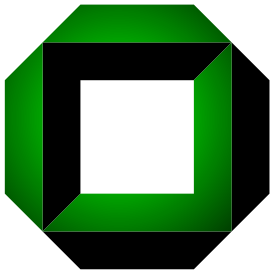

Universität Karlsruhe (TH)
Forschungsuniversität • gegründet 1825

Fakultät für Informatik
Lehrstuhl Programmierparadigmen - IPD Snelting

Eine Systematik für lokale Optimierungen

Studienarbeit von Johannes Franz

4. Juli 2009

Betreuer:
Dipl.-Inform. Matthias Braun

Verantwortlicher Betreuer:
Prof. Dr.-Ing. Gregor Snelting

Hiermit erkläre ich, die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt zu haben.

Ort, Datum Unterschrift

Inhaltsverzeichnis

1 Einleitung

In aktuellen Compilern werden lokale Optimierungen von Hand geschrieben. Dieser Code ist schwer lesbar und fehleranfällig. Vereinzelt stehen im Code die umzusetzenden Regeln in Form von Kommentaren dabei. Dies ist jedoch nicht systematisch und es ist nicht leicht herauszufinden, welche Regeln bereits implementiert wurden. Auch das Hinzufügen von Regeln ist zeitaufwendig.

Ziel dieser Arbeit ist es, lokale Optimierungen, die im Compiler zur Anwendung kommen, zu systematisieren und zu generieren. Dabei dient als Grundlage die am Institut entwickelte Zwischendarstellung Firm. Zu Firm gibt es eine Implementierung in Form einer Bibliothek mit Namen libFirm. Mit dieser Arbeit wird eine Sprache entwickelt, die lokale Optimierungsregeln für Firm beschreibt. Für diese Regeln wird ein Codegenerator implementiert, der C-Code für die Verwendung von libFirm erzeugt.

Firm ist eine graphbasierte Zwischendarstellung. Um einen kleinen Einblick von der graphischen Darstellung zu bekommen, betrachten wir beispielhaft die folgende einfache Funktion:

```
int func(int a, int b) {
  return  a + b;
}
```

Abbildung 1.1 zeigt die graphische Repräsentation. Die Pfeile stellen Abhängigkeiten dar und sind deshalb in umgekehrter Richtung zur Eltern/Kind Beziehung eingezeichnet. Mit Vorgängern eines Knotens sind die Operanden gemeint und die Nachfolger eines Knotens sind alle Knoten, die von diesem Knoten abhängen. Die Addition wird im Graphen durch einen Add-Knoten dargestellt, dessen zwei Vorgänger die Parameter der Funktion sind. Auf weitere Eigenschaften dieser Darstellung wird im nächsten Kapitel eingegangen.

Das zweite Kapitel beschäftigt sich mit den mathematischen Grundlagen und geht auf die Firm-Zwischendarstellung ein. Kapitel drei systematisiert die Optimierungen. Im viertel Kapitel wird die Sprache vorgestellt, mit der sich die Optimierungsregeln beschreiben lassen. Das darauffolgende Kapitel beschreibt die Implementierung des Codegenerators und im letzten Kapitel werden die handgeschriebene und die generierte Methode miteinander verglichen.

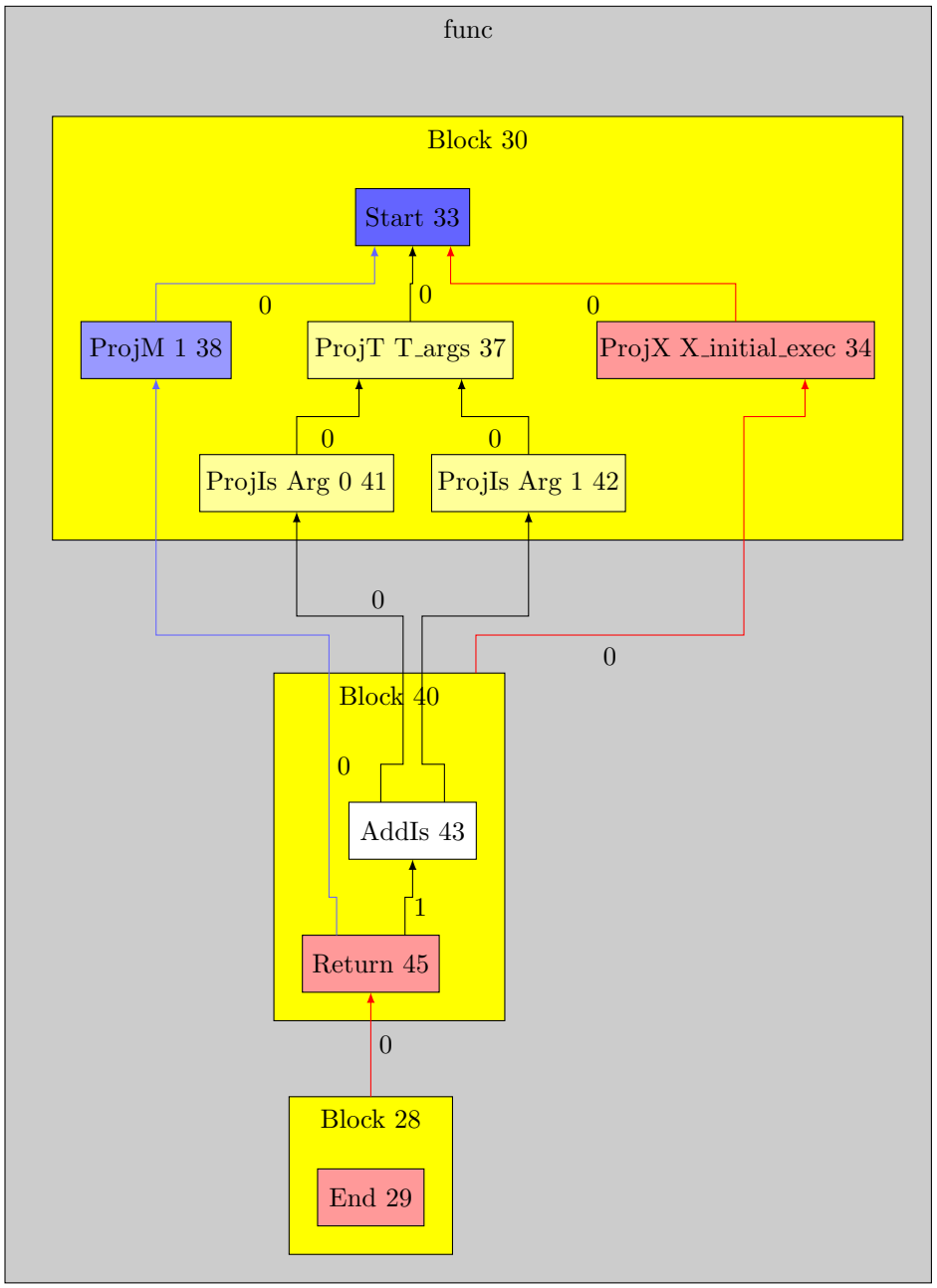

Abbildung 1.1: Graphdarstellung

2 Grundlagen

In diesem Kapitel wird definiert, was unter einer Optimierung in Firm zu verstehen ist. Es wird auf mathematische Grundlagen eingegangen und es werden die Grundlagen von Firm/libFirm beschrieben, die zum Verständnis für die lokalen Optimierungen notwendig sind.

2.1 Lokale Optimierungen in Firm

Firm basiert auf dem Konzept der expliziten Abhängigkeitsgraphen (EAGs), das aus der Dissertation von Martin Trapp [Tra01] hervorgegangen ist. Eine wichtige Eigenschaft von EAGs ist, dass bei Ersetzungen von Operationen die Abhängigkeiten erhalten bleiben, so dass diese nicht neu berechnet werden müssen. Lokale Optimierungen sind in dieser Arbeit somit Graphersetzungen, wobei sich die Lokalität auf die unmittelbare Nähe der Operanden im Graphen zu ihrer Benutzung bezieht.

Um Optimierungen auf einem Graphen durchzuführen, wird nach Teilgraphen gesucht, welche durch andere Teilgraphen ersetzt werden. Das allgemeine Problem der Suche nach Teilgraphen ist NP-vollständig und daher zu aufwendig. Deshalb wird jeder Knoten im Graphen einzeln als Wurzelknoten betrachtet. Ausgehend von diesem Wurzelknoten werden dessen Vorgänger bis zu einer endlichen Tiefe angeschaut. Wenn ein bestimmtes Muster auf diesen Teilgraphen passt, wird er durch einen der Regel entsprechenden Teilgraphen ersetzt.

2.2 Mathematische Gesetze

In diesem Abschnitt werden algebraischen Identitäten beschrieben, die Einfluss auf die Optimierungsregeln haben. Beispiele werden in C-Notation aufgeschrieben.

Kommutativität

Wenn eine Regel auf einen kommutativen Wurzelknoten definiert ist, gibt es auch eine duale Optimierungsregel dazu. Ein einfaches Beispiel ist eine Regel, die `a + 0` durch `a` ersetzt. Die duale Ersetzungregel transformiert `0 + a` nach `a`. Typische kommutative Operationen sind Addition, Multiplikation, bitweises Und, bitweises Oder und bitweises exklusives Oder.

Idempotenz

Unäre Operationen, die bei mehrfacher Anwendung keinen Unterschied zur einfachen Anwendung ergeben, sind idempotent.

$$f = (f \circ f)$$

Typkonversionen sind ein Beispiel für diesen Fall: `(int) (int) s` ist semantisch dasselbe wie `(int) s`.

Selbstinverse Funktionen

Eine ähnliche Eigenschaft ist es, wenn eine Operation sich bei zweifacher Anwendung wieder aufhebt. Eine solche Operation ist selbstinvers.

$$(f \circ f) = id$$

Dies ist zum Beispiel beim bitweisen Komplement ($\sim\sim$`x == x`) und beim unären Minus (`--x == x`) der Fall.

Assoziativität

Operationen, die unter beliebiger Klammerung im mathematischen Sinn identisch sind, sind strukturell unterschiedlich und erzeugen somit unterschiedliche Graphen. Mustererkennende Regeln im Graphen greifen jedoch nur auf eine konkrete Struktur. Es lohnt sich also unter Ausnutzung des Assoziativsgesetzes zu transformieren, um andere Regeln greifen zu lassen. Die typischen kommutativen Operationen, wie sie weiter oben angegeben sind, sind auch assoziativ.

Neutrale Elemente

Bei einer binären Operation kann es Konstanten geben, deren Verknüpfung keine Auswirkung haben. Typische Vertreter sind die `0` bei der Addition und die `1` bei der Multiplikation. Somit gibt es zum Beispiel eine Regel, die `x * 1` nach `x` transformiert.

Distributivität

Die Distributivität kann in zwei Richtungen ausgenutzt werden. Die folgenden beiden Ausdrücke sind identisch: `a*(b+c) == a*b + a*c`. Der Ausdruck auf der linken Seite spart eine Multiplikation. Der Ausdruck auf der rechten Seite läßt möglicherweise in einem anderen Kontext weitere Regeln greifen. Daher kann versucht werden zunächst nach der rechten Seite zu transformieren und am Ende diese Transformation falls nötig wieder zu reversieren.

2.3 Grundlagen von FIRM/LIBFIRM

Ein FIRM-Graph ist nach dem Prinzip der Statischen Einmalzuweisung (SSA-Form) aufgebaut, wodurch einige Optimierungen eine bessere Laufzeit bekommen. Eine kurze Einführung zur SSA-Form befindet sich in [Aho08]. Eine weitere Eigenschaft von FIRM ist, dass Variablen nicht mehr vorkommen, sondern als Werte von Knoten abstrahiert werden.

Um die FIRM Zwischendarstellung, wie sie in Abbildung 1.1 zu sehen ist, zu verstehen, kann sie sich als Überlagerung von mehreren Konzepten in einem einzigen Graphen vorgestellt werden:

Dazu gehört die Grundblockzugehörigkeit. Grundblöcke werden in Firm durch Knoten dargestellt. Jeder Knoten, außer einem Grundblock-Knoten, hat als seinen ersten Vorgänger den Grundblock-Knoten, zu dem er gehört. Die Grundblockzugehörigkeit wird in der Graphdarstellung abstrahiert, indem der Grundblock-Knoten alle Knoten umschließt, die ihn als Vorgänger haben.

Der Steuerfluss wird in FIRM durch Steuerfluss-Knoten wie Jmp oder Return dargestellt. Der Nachfolger eines Steuerfluss-Knoten ist der nächste Grundblock.

Die Modifikation des Speichers wird in FIRM durch Speicher-Knoten wie Load und Store modelliert. Wenn ein Knoten den Speicher potentiell verändert, ist einer seiner Vorgänger ein Speicherknoten und seine Nachfolger können den veränderten Speicher benutzen.

In der Zwischendarstellung FIRM können Knoten mehrere Operanden haben aber nur einen Ergebniswert. Die Operanden werden als Eingänge und der Ergebniswert als Ausgang bezeichnet. Wenn eine Operation mehrere Ergebnisse liefert, ist der Ergebniswert ein Tupelwert. Auf das einzelne Ergebnis wird mit einem Proj-Knoten zugegriffen, welcher das entsprechende Ergebnis projeziert. In Abbildung 1.1 werden aus dem Start-Knoten der Speicher, die Argumente und der initiale Steuerflussknoten projeziert.

2.3.1 Knotentypen in FIRM

Einen Überblick über die FIRM-Knoten gibt die Abbildung 2.1. Hier werden alle Knoten aufgelistet, die potentiell optimiert werden. Die Angabe der Stelligkeit lässt dabei den Fakt aus, dass alle FIRM-Knoten bis auf Knoten Block als ersten Vorgänger den Grundblock haben, zu dem sie gehören. Dieser hat aber nichts mit den Operanden der Operation zu tun und zählt deshalb nicht zur Stelligkeit. Eine gute jedoch veraltete Übersicht ist auch aus dem technischen Bericht über die FIRM Zwischendarstellung [TLB99] zu entnehmen.

2.3.2 Knotenmodi in FIRM

Jedem Ausgang eines Knotens ist ein Mode zugeordnet. Für jeden Knoten ist außerdem spezifiziert, welche Modi seine Eingänge haben dürfen. Die Modi der Eingänge eines Knoten stimmen immer mit den Modi der Ausgänge seiner Vorgänger über-

Knotenname	#Operanden	Kommentar
Abs	1	Absoluter Wert
Add	2	Addition
And	2	Bitweises Und
Block	n	Grundblock
Bound	4	Grenzüberprüfungen
Call	n	Prozeduraufruf
Cast	1	explizite Typumwandlungen in Klassenhierachien
Cmp	2	Vergleich von zwei Werten
Cond	1	Bedingter Sprung
Confirm	1	Repräsentiert abstraktes Wissen über einen Wert
Const	0	Konstante
Conv	1	Typkonversion (cast in C)
CopyB	3	Memcopy
Div	3	Integer Division
DivMod	3	Integer Division und Integer Rest
End	n	Das Ende des Kontrollflusses einer Prozedur
Eor	2	Exklusives Oder
Id	1	Identität (implementierungsbedingt)
Jmp	0	Unbedingter Sprung
Load	2	Laden eines Wertes
Minus	1	Unäres Minus
Mod	3	Integer Rest
Mul	2	Multiplikation
Mux	3	Bedingungsoperator
Not	1	Einerkomplement
Or	2	Bitweises Oder
Phi	n	Phi-Knoten (SSA-Darstellung)
Proj	1	Extrahiert einen einzelnen Wert aus einem Tuple-Knoten
Quot	3	Floating Point Division
Raise	2	Erzeugt eine Ausnahme
Rotl	2	Zyklische Verschiebung
Sel	3	Wählt ein Attribut eines Objektes aus
Shl	2	Linksverschiebung
Shr	2	Logische Rechtsverschiebung
Shrs	2	Arithmetische Rechtsverschiebung
Store	3	Speichern eines Wertes
Sub	2	Subtraktion
SymConst	0	Symbolische Konstante (erst bekannt nach Optimierungen)
Sync	n	Vereinigt Speicherbereiche
Tuple	n	Vereinigt mehrere Werte in Einen

Abbildung 2.1: Knotentypen in Firm

ein. Abbildung 2.2 listet alle in Firm vorhandenen Modi auf. Der Mode `BB` ist der Ausgangsmode von Grundblock-Knoten. Den Mode `X` haben Steuerfluss-Knoten. Knoten, die den Speicher modellieren, haben den Mode `M`. Ein Knoten mit mehr als einem Ergebnis bekommt den Mode `T`.

F	float
D	double
E	extended
Bs	signed byte
Bu	unsigned byte
Hs	signed short
Hu	unsigned short
Is	signed int
Iu	unsigned int
Ls	signed long
Lu	unsigned long
LLs	signed long long
LLu	unsigned long long
BB	basic block
X	execution
P	pointer
M	memory
T	tuple

Abbildung 2.2: Knotenmodi in Firm

2.3.3 Optimierungsalgorithmus

Lokale Optimierungen werden an zwei Stellen durchgeführt: bei der Konstruktion von neuen Knoten und jedesmal, wenn die Optimierungsphase für lokale Optimierungen aktiviert wird. Diese Optimierungsphase wird mehrmals durchlaufen, da andere Optimierungen den Graphen so verändern können, dass es sich wieder lohnt, den Graphen lokal zu optimieren. Dabei wird der Graph in umgekehrter Tiefensuche startend beim End-Knoten durchlaufen. Für jeden Knoten wird getestet, ob eine Regel anwendbar ist. Die Regeln müssen dabei sicherstellen, dass maximal nur eine Regel angewandt werden kann. Somit kann pro Muster nur eine Regel existieren. Kann ein Knoten optimiert werden, werden alle Benutzer dieses Knotens gekennzeichnet, um erneut optimiert zu werden. Damit wird der Fixpunktalgorithmus realisiert, der dafür sorgt, dass solange optimiert wird, bis keine Regel mehr anwendbar ist. Zu beachten ist außerdem, dass durch die umgekehrte Tiefensuche zunächst alle Vorgänger eines Knotens optimiert werden, bevor dieser selbst optimiert wird.

Optimierung bei der Graphkonstruktion

Durch die sofortige Optimierung bei Knotenkonstruktion, können Konstanten sofort gefaltet werden. Dabei wird überprüft, ob sich aus den Operanden eines neuen Knotens ein konstanter Wert errechnen lässt und gegebenenfalls ein Const-Knoten mit diesem neuen Wert zurückgegeben. Dadurch, dass die Operanden eines neuen Knotens zuerst erzeugt worden sind, wird sichergestellt, dass ein neuer Knoten, dessen Blätter nur aus Konstanten bestehen, zu einem Const-Knoten zusammengefaltet wird.

2.3.4 Normalisierung in libFirm

In libFirm werden neue Knoten normalisiert. Das bedeutet, dass bei kommutativen Operationen der Operand mit dem höchsten Grad an Konstantheit immer auf der rechten Seite steht. Dazu werden in libFirm drei Grade an Konstantheit definiert: `const_const`, `const_like` und `const_other`. In der aktuellen Implementierung hat der Const Knoten den Grad `const_const`. Den Grad `const_like` hat der Knoten SymConst. Alle anderen Knoten haben den Grad `const_other`. Als kommutative Operationen sind die Operationen Add, Mul, And, Or und Eor markiert. Zusätzlich wird, wenn beide Operanden den gleichen Grad an Konstantheit haben, der Operand mit dem größeren Knotenindex auf die rechte Seite gebracht. Der Knotenindex ist eine eindeutige Zahl, die den Knoten identifiziert.

3 Systematisierung von Optimierungen

Wie in Buch von Trapp [Tra01] beschrieben, werden Optimierungen in zwei Kategorien eingeteilt: Zum einen gibt es Optimierungen im eigentlichen Sinn, die die Laufzeit des Programms verringern. Diese werden als *kostenreduzierende Transformationen* bezeichnet. Zum anderen gibt es Optimierungen, die unter Umständen die Laufzeit des Programms erhöhen können, aber es ermöglichen, Optimierungen der ersten Art anzuwenden und somit am Ende zu einem besseren Ergebnis führen. Diese werden *normalisierende Transformationen* genannt. Normalisierungen dienen auch als Basis für spätere Optimierungen, zum Beispiel für das Finden von gemeinsamen Teilausdrücken.

Abgesehen von der groben Unterteilung in normalisierende und kostenreduzierende Optimierungen können Optimierungen weiterhin zu Bereichen gruppiert werden, die sich durchaus überlappen. Die meisten Optimierungen zählen zur Gruppe der algebraischen Vereinfachungen. Auch können architekturabhängige Optimierungen und FIRM-spezifische Optimierungen abgegrenzt werden.

3.1 Normalisierende Optimierungen

Aufgrund von Kommutativität, Assoziativität und Distributivität können semantisch gleiche Graphen durch unterschiedliche Strukturen repräsentiert werden. Wenn keine Vorbedingungen mit dem Graphen verknüpft sind, dann muss eine Optimierung in allen verschiedenen Varianten ausgeschrieben werden. Um diesen Aufwand zu reduzieren wird versucht, den Graphen zu normalisieren. Dies kann schon beim Graphaufbau geschehen, wie es bei den Konstanten, die auf die rechte Seite verschoben werden, der Fall ist. Dies kann aber auch in Form von Regeln ausgedrückt werden. Eine solche Regel ist zum Beispiel der Ersetzung der Subtraktion durch eine Addition und einer Negierung des zweiten Operanden. Eine weitere Normalisierung ist das Ersetzen von Shift- und Rotationsoperationen durch bitweises Und, Oder oder exklusives Oder. In der Folgenden Transformation wird die Shiftoperation im Wurzelknoten zu einer bitweisen Und Operation normalisiert: `(x & c1) >> c2` nach `(x >> c2) & (c1 >> c2)`.

3.2 Kostenreduzierende Optimierungen

Zunächst muss geklärt werden, welche Metrik für Kosten verwendet wird. Normalweise wird diese Metrik anhand der Anzahl der Operationen gemessen. Operationen

können auch feingranularer in billige und teure Operationen aufgeteilt werden. Eine Multiplikation ist um Größenordungen langsamer als eine Addition. So gibt es beispielsweise eine Optimierung, die eine Multiplikation durch Shift- und Additionsoperationen ersetzt. Auch Steuerfluss ist bedingt durch die heute Prozessorarchitektur und Sprungvorhersage teuer. So gibt es Regeln die versuchen, Cond-Knoten zu ersetzen.

3.3 Algebraische Vereinfachungen

Hierunter zählen Optimierungen, die sich algebraische Identitäten zu Nutze machen. Die Distributivität kann beim bitweisen Oder zur Anwendung kommen. So ist `(a | c) | (b | c)` identisch zu `(a | b) | c`. Der Ausdruck `(a | c) ^ (b | c)` kann zu `(a ^ b) & ~ c` vereinfacht werden.

3.4 Architekturabhängige Optimierungen

Häufig kann die Zielarchitektur ausgenutzt werden, um weitere Optimierungen durchzuführen. So kann das Muster `(a < 0) || (b < 0)` auf einer 32-Bit-Maschine zu `((a | b) >> 31)` umgeformt werden. Unter Ausnutzung des Zweierkomplements ist `~a + a` identisch zu `-1`.

3.5 FIRM-spezifische Optimierungen

Hierzu zählen Optimierungen auf FIRM-spezifischen Knoten wie Id, Tuple oder Confirm Knoten. Diese Entstehen als Artefakte aufgrund von anderen Optimierungen. So kann aus einem Knoten ein Id-Knoten werden, wenn dieser Knoten wegoptimiert wurde, aber kein Zugriff auf die Benutzer des Knotens zu dem Zeitpunkt der Optimierung besteht. Dieser wird mit einer weiteren FIRM-spezifischen Regel wegoptimiert.

Auch können als Artefakte von Optimierungen Bad-Knoten oder unerreichbare Blöcke entstehen. Diese kennzeichnen unerreichbaren Code, der durch entsprechende Regeln wegoptimiert wird.

4 Sprachentwurf

In diesem Kapitel werden die Entwurfsentscheidungen beschrieben, die zum Aufbau der Regelsprache geführt haben. Anschließend wird die Sprache selbst beschrieben.

Da es sich hier um Transformationen in einem Graphen handelt, sind die Optimierungsregeln Graphersetzungsregeln. Es müssen somit für jede Regel zwei Graphstrukturen beschrieben werden können: Eine für das Muster und die andere für den Ersetzungsgraphen.

Mit der Ausdrucksfähigkeit der Sprache nimmt auch die Komplexität zu. Daher gilt es abzuwägen zwischen der Vielfältigkeit der Muster, die mit der Sprache beschrieben werden können, und der Lesbarkeit und Handhabung der Sprache. Die folgende Entwurfskriterien haben während der Entwicklung der Sprache eine Rolle gespielt:

- Einfache Regeln können einfach aufgeschrieben werden

  Einfache und häufig vorkommende Muster sollen kompakt darstellbar sein. Sie haben somit Kommentarcharakter. Komplexe Muster sollen auch darstellbar sein, können aber eine komplexere Schreibweise erfordern. Sie können auch durch Kommentare erläutert werden.

- Möglichkeit, beliebigen Code ausführen zu lassen

  Um beliebig komplexe Regeln aufschreiben zu können, gibt es innerhalb der Sprache die Möglichkeit, einzelne Knoten um zusätzliche selbstdefinierte Bedingungen zu erweitern als auch neue Bedingungen zwischen mehreren Knoten zu definieren.

- Anpassung an Firm

  Da es sich hier um eine Codegenerierung speziell für die Bibliothek libFirm handelt, sind alle Knotennamen genauso wie in Firm benannt. Auch ähneln einfache Regeln den Kommentaren im handgeschriebenen Code.

Um die Sprache nicht zu komplex zu gestalten, mussten ein paar Einschränkungen gemacht werden. Mit der Sprache können nur endliche Muster beschrieben werden. Das schließt Muster aus, die Knoten mit beliebig vielen Vorgängern enthalten. Auch Muster, die Ketten beliebiger Länge enthalten, können mit dieser Sprache nicht beschrieben werden. Diese Muster können nur bis zu einer endlichen Tiefe umgesetzt werden. Solche Muster treten aber nicht häufig auf und können durch die Möglichkeit, beliebigen Code auszuführen, in eine Funktion ausgelagert werden.

Die Regeln werden in einer Regeldatei zusammengefasst. Da komplexe Regeln durchaus mehrere Zeilen belegen können, kann eine Regel durch Einrücken beliebig zeilenübergreifend aufgeschrieben werden. Das Ende einer Regel wird automatisch erkannt. Außerdem sind die üblichen C-Kommentare erlaubt: '//' am Anfang einer Zeile kennzeichnet diese Zeile als Kommentar. Dies ist hilfreich zum Kommentieren von komplexen Regeln. Mehrere Regeln können mit '/*' und '*/' umschlossen werden, um sie auszukommentieren. Das vereinfacht das Debuggen.

Eine Regel selbst besteht aus zwei Teilen, der Left Hand Side (LHS) und der Right Hand Side (RHS). Die LHS beschreibt das Muster für den Teilgraphen, der durch das Muster der RHS ersetzt wird. Die LHS und die RHS werden mit einem Separator `->` getrennt. Am Ende stehen eckige Klammern, die optional einen Debugstring enthalten. Dieser ruft ein Makro nach der Ausführung der Regel aus. Diese Makros sind für die Statistik und das Propagieren von Debuginformationen zuständig und sind bereits in libFirm vordefiniert.

Zusätzlich können an beliebiger Stelle Code-Knoten definiert werden, die in nachfolgenden Regeln verwendet werden können. Hierbei wird der Code in Anführungszeichen einem Bezeichner zugewiesen.

Mit der Backus-Naur Form wird der bisherige Teil der Grammatik formal beschrieben. Nichtterminale werden dabei in spitzen Klammern notiert und Terminale in Großbuchstaben.

```
<Regeln>          ::= <RegelElement> <Regeln>
<RegelElement>    ::= <Regel>  | <CodeDefinition>
<Regel>           ::= <LHS> "->" <RHS> "[" [ DEBUGSTRING ] "]"
<CodeDefinition>  ::= CODENODE "=" CODE
```

4.1 Das Muster

Auf der linken Seite wird das Muster beschreiben. Eltern/Kind-Beziehungen zwischen Knoten werden durch Anhängen der Kinder mit Klammern an den Knotennamen aufgeschrieben. Wenn ein Knoten keine Kinder hat, reicht der Knotenname aus.

So wird beispielsweise ein Eor-Knoten mit zwei konstanten Vorgängern folgendermaßen darstellt:

```
Eor(Const, Const)
```

Ein Knoten kann mit einem Code-Knoten umschlossen werden, um zusätzliche Bedingungen an den Knoten zu übergeben. Dabei wird bei der Codegenerierung jedes Vorkommen von `$0` im definierten Code durch die entsprechende Variable des Knotens ersetzt. Um beispielsweise das rechte Kind weiter auf eine Konstante mit dem Wert `0` einzuschränken, wird zusätzlich ein Code-Knoten definiert:

```
ZERO = “is_Const_null($0)“
Eor(Const, ZERO(Const))
```

Um auszudrücken, dass zwei Knoten gleich sind, werden Variablendeklarationen und Variablenreferenzen eingeführt. Variablennamen beginnen immer mit einem '$' Zeichen. Knotenvariablen bestehen aus mindestens einem Kleinbuchstaben. Wenn es sich um eine Deklaration und nicht um eine Referenz handelt, dann folgt auf den Namen ein ':'. Variablendeklarationen werden vor den Knotennamen geschrieben. Wenn der Knotentyp nicht relevant ist, kann er auch weggelassen werden. Auf Variablenreferenzen darf kein Knotenname folgen und zu jeder Variablenreferenz muss eine entsprechende Variablendeklaration existieren.

Beispielsweise wird ein Muster, dass einen Eor-Knoten darstellt, dessen Kinder identisch sind, so dargestellt:

```
Eor($a:, $a)
```

Für Optimierungsregeln werden nicht alle Modi benötigt. Insbesondere soll geprüft werden können, ob ein Knoten einer Gruppe von Modi entspricht. Deshalb sind in der Regelsprache neben den vorangegangenen Modi folgende Modi implementiert. Auf der rechten Seite steht die entsprechende Funktion aus libFirm, die diesen Test implementiert. Deren Argument ist der zu testende Modus.

F	mode_is_float
I	mode_is_int
P	mode_is_reference
S	mode_is_signed
N	mode_is_num
W	mode_wrap_around

In der Regelsprache werden die Ausgangsmodi durch Anhängen des entsprechenden Moduszeichens an den Knotennamen gefolgt vom Trennzeichen '_' ausgedrückt. Ein Modus kann auch negiert werden, indem ein '!' vor das Moduszeichen geschrieben wird. Zum Beispiel wird ein Mul-Knoten, der keinen vorzeichenunbehafteten Ausgangsmodus haben soll, so ausgedrückt:

```
Mul_!S
```

Ein weiterer häufig vorkommender Fall ist der Test auf Gleichheit von Modi. Die Regelsprache kann ausdrücken, dass die Modi von zwei Knoten gleich sind. Dazu werden wie bei Knotennamen Variablendeklarationen und Variablenreferenzen benutzt. Allerdings sind hier nur Großbuchstaben zugelassen. Eine Variablendeklaration wird vor den Modus, falls vorhanden, geschrieben und eine Variablenreferenz darf nur alleine auftauchen. Im Unterschied zu Variablenreferenzen für Knotennamen, können

Variablenreferenzen für Modi auch negiert werden. Im folgenden Beispiel wird ein Sub-Knoten beschrieben, dessen Moduswert überlaufen kann. Dieser Sub-Knoten hat als rechten Vorgänger einen Add-Knoten. Dessen rechter Vorgänger soll identisch sein mit dem linken Vorgänger des Sub-Knotens. Der linke Vorgänger soll nicht den Modus des Sub-Knotens haben. Zusätzlich ist er benannt. Die Variable wird auf der LHS nicht weiter verwendet, kann aber beispielsweise innerhalb der RHS verwendet werden.

```
Sub_$M:W($a:,Add($l:_!$M,$a))
```

Nach dem Muster können noch optional eine beliebige Menge an sogenannten Addons hingeschrieben werden. Auch hier kann die Variable `$l` verwendet werden.

4.1.1 Addons

Addons dienen dazu, beliebigen Code auszuführen. Sie werden wie die Code-Knoten definiert, können jedoch bis zu fünf Parameter haben. Dies war ausreichend und die meisten Addons genötigen in der Regel zwei bis drei Paramter. Als Parameter sind Knoten- und Modusreferenzen zugelassen. Zusätzlich dürfen sie auch eine vorangestellte Deklaration enthalten. Im definierten Code wird `$0` durch die neue Variable ersetzt und `$1` bis `$5` durch den entsprechenden Parameter. Auf diese Weise können Bedingungen aufgeschrieben werden, die von mehreren Knoten abhängen. Auch können mit dieser Schreibweise neue Knoten definiert werden, die auf der rechten Seite benutzt werden. Eine häufig vorkommende Regel für Phi-Knoten, die nur konstante Vorgänger haben wird hier beispielhaft für den Abs-Knoten angegeben.

```
CONSTPHI = “is_const_Phi($0)“
APPLYPHI_ABS = “$0 = apply_unop_on_phi($1, tarval_abs)“
Abs($p:CONSTPHI() ) $v:APPLYPHI_ABS($p) -> $v [ ALGSIM0_CONST_PHI ]
```

Mit dem Addon `$v:` wird ein neuer Knoten erzeugt, der durch die libFirm Funktion `apply_unop_on_phi` erzeugt wird. Der erste Parameter dieser Funktion ist der Knoten, der als `$p` deklariert wurde. Der zweite Parameter ist ein Zeiger auf die libFirm interne Funktion `tarval_abs`. Der so erzeugte Knoten ist der Rückgabewert für das Muster auf der rechten Seite.

4.2 Der Ersetzungsgraph

Die rechte Seite verhält sich ähnlich wie die linke Seite. Es gibt jedoch die Möglichkeit, mit Konstanten zu rechnen. Dabei handelt es sich um eine Mikrosprache, die sich von der bisher vorgestellten Notation unterscheidet. Zahlen (`0-9`), Wahrheitswerte (`t` oder `f`) und Variablenreferenzen von Const-Knoten dürfen unter Verwendung der in der Programmiersprache C üblichen Rechenzeichen verknüpft werden,

um neue Const-Knoten zu erzeugen. Beispielsweise kann ein Const-Knoten auf der rechten Seite in einen Const-Knoten mit negiertem Wert umgewandelt werden.

```
Sub($a:,Mul_$M:($b:,$c:Const)) -> Add($a,Mul_$M($b,Const(-$c))) []
```

Diese Schreibweise des Const-Knotens auf der rechten Seite besagt nicht, dass der Const-Knoten einen Vorgänger besitzt, sondern wird benutzt, um den Wert des Knotens anzugeben.

Weiterhin dürfen auf der rechten Seite keine Deklarationen auftauchen. Variablenreferenzen und Modusreferenzen sind jedoch notwendig.

4.3 Proj-Knoten

Proj-Knoten haben in Firm nur einen Vorgänger. Ein häufig verwendetes Attribut des Proj-Knotens ist dessen Projektionsnummer, welche die Nummer des Wertes angibt, die aus einem Tuple Wert projeziert wird. Um die Handhabung mit Projektionsnummern zu vereinfachen, wird bei Proj-Knoten die Projektionsnummer wie ein zweiter Vorgänger gehandhabt. Diese können genauso wie Knoten mit zusätzlichen Bedingungen in Form von Code-Knoten erweitert werden.

4.4 Abschlussbemerkungen

Die Sprache ist so entworfen worden, dass sie beim Aufschreiben viele Freiheitsgrade zulässt. Damit können Regeln so aufgeschrieben werden, dass sie besonders lesbar sind. Das beginnt beim Benennen von Code-Knoten. Um sie von den Firm-Knoten zu unterscheiden, sollten sie in Großbuchstaben geschrieben werden. Variablennamen sollten entsprechend ihrer Bedeutung mit l und r für den linken und rechten Vorgänger benannt werden, Wahrheitswerte entsprechend mit t und f. Auch die Platzierung von zusätzlichem Code kann, wenn er nicht von mehr als einem Parameter abhängt, sowohl an den Knoten geschrieben als auch als Addon realisiert werden.

5 Implementierung

Für die Implementierung wurde ein Parsergenerator benutzt. Das Standardwerkzeug hierfür ist `yacc`, welches einen LALR-Parser implementiert. Die Theorie hierzu wird im Buch von Aho, Lam, Sethi und Ullman beschrieben[Aho08]. In diesem Buch wird sehr detailliert beschrieben, wie ein Parser zu implementieren ist und yacc ist eine Implementierung nach diesem Buch. GNU `bison` ist eine erweiterte open source Nachimplementierung von yacc. Beide Tools legen die Programmiersprache C für den Benutzer des Tools fest. An der Universität in Groningen, Niederlanden, wurde bison komplett neu in C++ umgeschrieben[Bro]. Diese Implementierung heißt `bisonc++` und wird hier verwendet.

Bisonc++ ist für die syntaktische Analyse zuständig. Die lexikalische Analyse scannt eine Datei und liefert die Tokens für die syntaktische Analyse. Das Standardtool, dass mit yacc zusammenarbeitet, ist lex. Dafür existiert eine freie GNU Implementierung namens `flex`. Diese arbeitet auch mit `bisonc++` zusammen und kommt in dieser Arbeit zum Einsatz.

5.1 Architektur

Die Implementierung setzt sich aus dem Scanner, dem Parser und dem Driver zusammen. Der Driver legt den im Folgenden beschriebenen Ablauf fest. Zunächst liefert der Scanner die einzelnen Token, welche in der Datei `scanner/lexer.l` durch reguläre Ausdrücke beschrieben werden. Der Parser, der aus der in der Datei `parser/grammar.y` beschriebenen Grammatik generiert wird, prüft die Syntax der Regeln und gibt detaillierte Fehlermeldungen wie Zeilennummer der fehlerhaften Regel aus. Die Anknüpfungen in der Grammatik bauen zwei Syntaxbäume für jede Regel auf: einen für die LHS und einen für die RHS. Anschließend werden in der semantischen Analyse die Syntaxbäume auf weitere Fehler überprüft, die nicht durch die Grammatik ausgedrückt werden konnten. In der Codegenerierungsphase wird für jeden Knoten der seinen Bedingungen entsprechende Code generiert und mit dem Knoten verknüpft. Zum Schluss wird der generierte Code in eine Datei ausgegeben.

5.2 Algorithmus

Eine naive Implementierung erzeugt für jede Regel eigenen Code. Jedoch überlappen sich viele Muster und es entsteht viel redundanter Code, der zu einer Verlangsamung der lokalen Optimierung während der Übersetzungszeit führt. Ein Ansatz, ein besseres Ergebnis zu erzielen, ist es, mehrere Regeln zu überlagern. Dies wurde in dieser

Arbeit umgesetzt. Alle linken Seiten von Regeln, die denselben Wurzelknoten haben, werden in einem Überlagerungsbaum zusammengefasst. Dessen Knoten bestehen aus Listen von Knoten. Ein Knoten aus dieser Liste ist Repräsentant von verschiedenen Regeln, wenn dieser Knoten in allen Regeln dieselben Bedingungen hat. Sobald sich von der Wurzel beginnend die Knoten von zwei Regeln unterscheiden, bleiben diese auch getrennt und bilden separate Unterbäume. Abbildung 5.1 zeigt so einen Überlagerungsbaum. Die Nummern geben an, zu welchen Regeln der Knoten gehört. In diesem Fall werden drei Regeln überlagert, die für den Wurzelknoten denselben Code erzeugen.

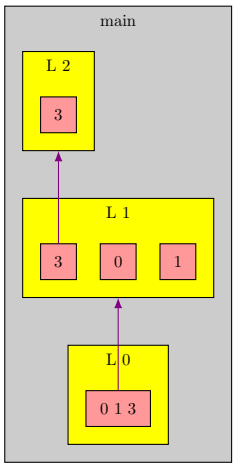

Abbildung 5.1: Beispielüberlagerungsbaum

Der Algorithmus für die Codeerzeugung aus einem Überlagerungsbaum funktioniert folgendermaßen: Der Überlagerungsbaum wird mit Tiefensuche durchlaufen und der mit einem Knoten verbundene Code ausgegeben. Dabei wird in jedem Knoten über seine inneren Knoten iteriert. Wenn ein Blatt mit einem inneren Knoten, der nur zu einer Regel gehört, erreicht wird, wechselt der Algorithmus in einen anderen Zustand und gibt im weiteren Durchlauf nur Code für diese Regel aus. Dabei wird die Regel gleichzeitig aus dem Überlagerungsbaum gelöscht. Nun wird die Tiefensuche wiederholt, bis alle Regeln abgearbeitet sind. Auf diese Weise wird sichergestellt, dass Code, der für mehrere Regeln gilt, nur einmal ausgegeben wird.

5.3 Interne Klassifizierung von Regeln

In libFirm wird unterschieden, ob es für die Ersetzung notwendig ist, neue Knoten zu erzeugen oder nicht. Eine Regel, die ein Muster durch einen seiner Vorgänger ersetzt, erzeugt keine neuen Knoten. Hierbei handelt es sich um eine *equivalent*-Regel. Andernfalls handelt es sich um eine *transform*-Regel. Der Grund für diese Trennung ist implementierungsbedingt auf die Verwendung der obstack Bibliothek zurück-

zuführen. In der Datei `ir/ir/iropt.c` sind alle lokalen Optimierungen von Hand ausgeschrieben. Hier gibt es für jeden Knoten, der optimiert wird, eine equivalent- und eine transform-Funktion. Der Codegenerator erkennt automatisch, ob es sich bei einer Regel um eine equivalent- oder eine transform-Regel handelt: Wenn auf der rechten Seite nur eine Variablenreferenz steht, die nicht zu einem Addon gehört, dann wird diese Regel als equivalent-Regel klassifiziert. Anderfalls handelt es sich um eine transform-Regel.

5.4 Benutzung

Der Generator liest die Regeln von der Standardeingabe. Zusätzlich können weitere Optionen angegeben werden. Mittels `-h` werden alle Optionen angezeigt. Die Option `-s` gibt Statistikinformationen, wie die Anzahl der geparsten Regeln, aus. Ein typischer Aufruf sieht folgendermaßen aus:

```
./generator -s < intput/intput.gen
```

Dies erzeugt die Datei `iropt.incl`. Die Datei `helper_functions.incl` wird zur Verfügung gestellt. Hier sind Hilfsfunktionen definiert und es können weitere hinzugefügt werden. Diese beiden Dateien werden in die ebenfalls gestellte Datei `iropt.c` eingebunden. Damit wird der generierte Code libFirm zur weiteren Compilierung bereitgestellt.

5.5 Debugging

Der Codegenerator bietet umfangreiche Debuggingfunktionen. Mit der Option `-d` wird der Generator im Debug-Modus gestartet. Es werden alle Parsing Schritte des LALR Automaten ausgegeben. Die Option `-v` gibt Ausgaben über den weiteren Zustand des Generators aus. Weiter existieren zwei Optionen zum Dumpen von internen Datenstrukturen. Für Firm gibt es bereits das `vcg`-Format zur Darstellung von Firm Graphen. Dieses kann mit dem ebenfalls für Firm entwickelten Programm `yComp` dargestellt werden. Dieses Format eignet sich auch gut für die Debuggingausgaben. Mit `-g` werden die linken Seiten jeder Regel im `vcg`-Format ausgegeben, sowohl für den Überlagerungsgraphen als auch für jede Regel separat, wobei die Zeilennummer in der Regeldatei zum leichteren Auffinden im Dateinamen mit angegeben ist. Die Option `-r` gibt die rechte Seite jeder Regel im `vcg` Format aus.

Das Debugging wird für die folgende algebraische Vereinfachungsregel demonstriert:

```
And(Or($a:, $b:), Not(And($a, $b))) -> Eor($a, $b) [ ALGSIMO_TO_EOR ]
```

Mit der Option `-g` wird die Datei `line_0122_And_lhs_transform.vcg` und mit der Option `-r` die Datei `line_0122_And_rhs.vcg` erzeugt. Zum leichteren Auffinden ist

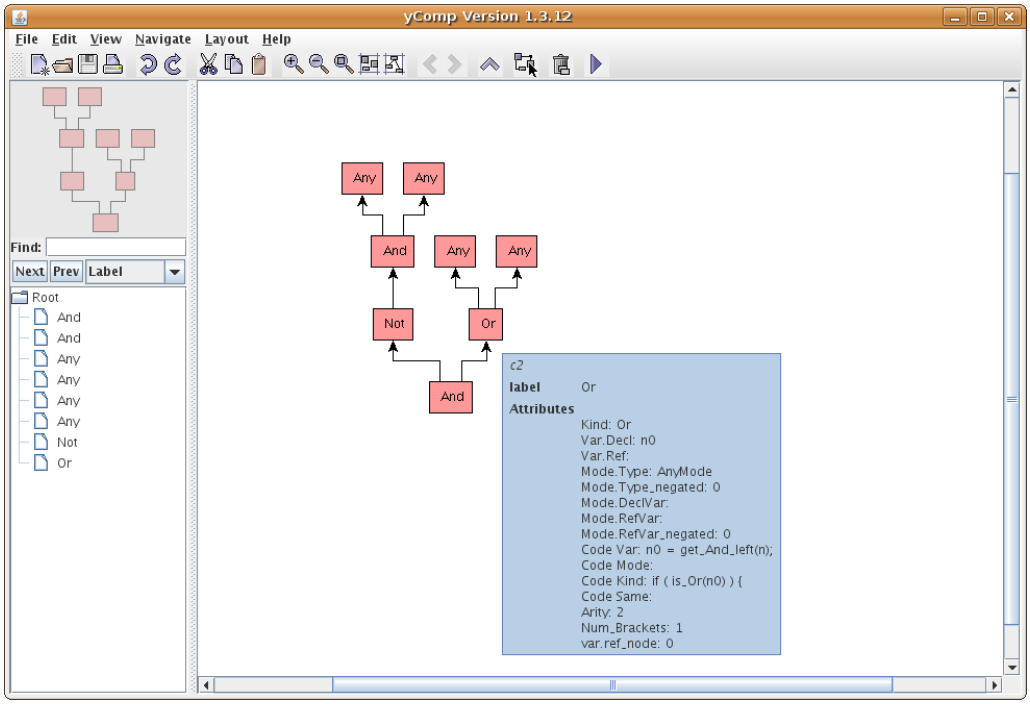

Abbildung 5.2: Beispiel für LHS in `yComp`

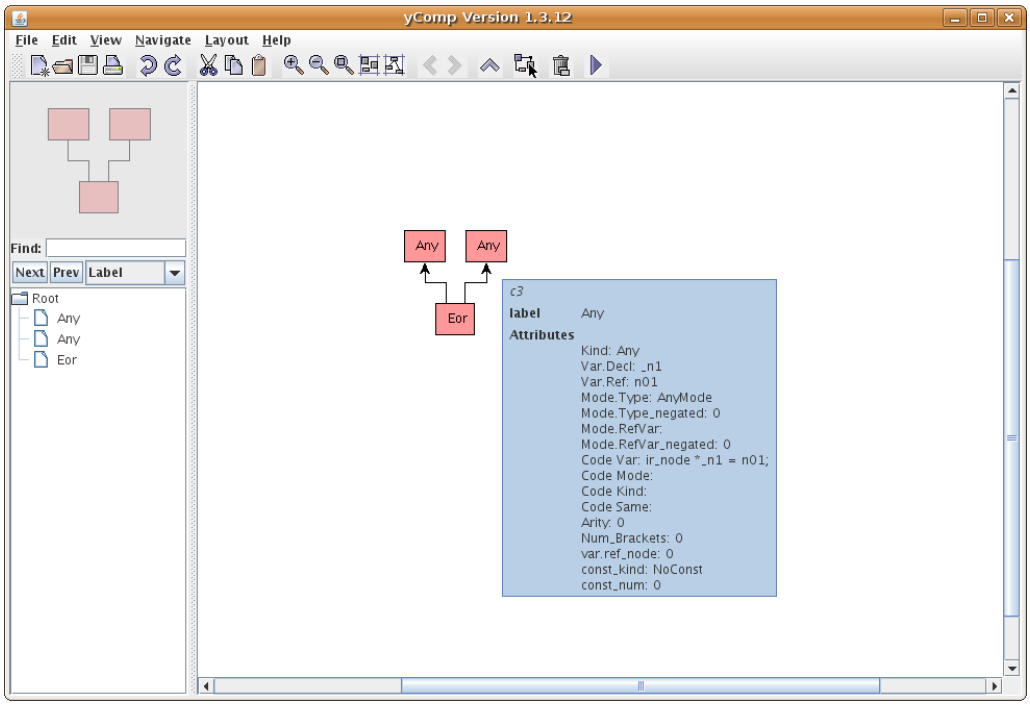

Abbildung 5.3: Beispiel für RHS in `yComp`

als Hilfe die Zeilennummer in der Regeldatei und der Wurzelknoten im Dateinamen mit angegeben. Die Regel wurde als transform-Regel erkannt. Diese beiden Dateien können graphisch wie in den Abbildungen 5.2 und 5.3 mit dem Tool `yComp` dargestellt werden. Genauso wie für FIRM-Graphen können die Attribute jedes Knotens angezeigt werden, wie in den Abbildungen zu sehen ist.

6 Evaluierung und Ausblick

In dieser Arbeit wurde eine Sprache zur Beschreibung von lokalen Optimierungen im Compiler entworfen. Die in der auf der Zwischensprache Firm basierenden Bibliothek libFirm vorhandenen Regeln wurden in dieser Sprache aufgeschrieben. Gleichzeitig wurde ein Programm implementiert, das C-Code für diese Regeln generiert.

6.1 Evaluierung

Es konnte gezeigt werden, dass eine automatische Generierung von lokalen Optimierungen möglich ist. Damit steht eine einfach zu verstehende Sprache zur Verfügung, die viele Implementierungsdetails versteckt. So braucht sich beispielsweise keine Gedanken mehr gemacht werden, ob eine Regel eine *equivalent-* oder eine *transform-*Regel ist. Dadurch können die Regeln in der Regeldatei übersichtlich in kompakter Form zusammen aufgeschrieben werden.

Im handgeschriebenen Code konnten weiterhin überflüssige Bedingungen gefunden werden. So werden einige Bedingungen mehrmals hintereinander überprüft oder sind gar nicht erfüllbar. Auch werden häufig beim Hinzufügen von neuen Regeln per Hand die Debugmakros vergessen. Die neue Schreibweise von Regeln macht diese Information offensichtlich.

Zu Testzwecken können Regeln einfach durch Auskommentieren entfernt und wieder hinzugefügt werden. Dies ist mit handgeschriebenem Code nicht so leicht möglich.

Auf der anderen Seite können Regeln, die sich sehr ähnlich sind, in handgeschriebener Form sehr kompakt und effizient hingeschrieben werden. Beispielsweise gibt es eine Funktion, die das Distributivgesetz für die Knoten And, Or und Eor anwendet, wobei für jeden Knoten dieselbe Funktion verwendet werden kann. Auch können Optimierungsregeln, die für alle binären Operationen gelten, für jeden binären Knoten wiederverwendet werden.

6.2 Statistik

Es wurden 400 Regeln auf 24 Wurzelknoten verteilt aus dem handgeschriebenen Code gewonnen. Diese Regeln konnten mit etwa 600 Zeilen aufgeschrieben werden, woraus circa 6.300 Zeilen lesbarer Code generiert wird. Somit konnten die Regeln mit nur einem Zehntel an Schreibaufwand ausgedrückt werden.

Insgesamt werden 7.000 Zeilen Code der handgeschriebenen Datei `iropt.c` durch circa 9.000 Zeilen Code der generierten Datei `iropt.c` ersetzt. Der Codeumfang hat sich durch die Generierung somit um 30% erhöht. Dies war zu erwarteten, da handgeschriebener Code kompakter aufgeschrieben werden kann.

Die physikalischen Codezeilen des Generators betragen 2.360 Zeilen, wobei Kommentare und Leerzeilen nicht mitgezählt wurden. Zusammen mit dem Umfang der Regeln ergeben sich somit 3.000 Zeilen an Schreibaufwand mit der Codegenerierung. Das ist weniger als die Hälfte von 7.000 Zeilen an Schreibaufwand vom handgeschriebenem Code.

6.3 Ausblick

Anhand der übersichtlichen Regeln ist es nun einfach zu sehen, welche implementiert wurden. In einer weiteren Arbeit können sich Optimierungsregeln von anderen Compilern wie dem `gcc` oder `llvm` angeschaut und gegebenenfalls ergänzt werden.

6.3.1 Spracherweiterungen

Es kann sich überlegt werden, in wieweit es sinnvoll ist, die Sprache um Unterstützung für Knoten mit einer beliegigen Anzahl an Vorgängern zu erweitern. Dies sind die Knoten Block, Call, End, Phi, Sync und Tuple. Solche Optimierungsregeln müssen mit der jetzigen Sprache in eine Funktion ausgelagert und mit Hilfe von Code-Knoten generiert werden.

6.3.2 Erweiterungen der Implementierung

Für die Codeeffizienz des generierten Codes lässt die Implementierung noch Potential offen. In dieser Implementierung wird die Überlagerung auf Knotenebene vorgenommen. Mit einem Knoten können beliebig viele Bedingungen verknüpft sein. In der Regel sind es zwischen zwei und drei Bedingungen. Knoten sind nur gleich, wenn sie in allen Bedingungen übereinstimmen. Wenn sie sich nur teilweise überschneiden, entsteht redundanter Code. Die Überlagerung könnte durch Aufteilung der Knoten in mehrere Knoten, die jeweils nur eine Bedingung enthalten, verbessert werden. Aufgrund des Sprachentwurfs hat der Regelschreiber jedoch Einfluss auf die Codeeffizienz. Jede Bedingung kann in Form eines Addons an die linke Seite angehängt werden. Auf diese Weise hat die Überlagerung keinen Effekt, da Addons bei der Überlagerung nicht berücksichtigt werden. Durch geschickte Platzierung von Code-Knoten an den Knoten kann die Coderedundanz reduziert werden.

Eine weitere Optimierung ordnet Bedingungen im generierten Code so an, dass Bedingungen, die für viele Regeln gelten, möglichst am Anfang geprüft werden. So ist die Wahrscheinlichkeit höher, dass ein Muster möglichst am Anfang gefunden wird und nicht nach vielen Negativtests. Dies kann mit der frühzeitigen Überprüfung von Knoten, die besonders häufig vorkommen, kombiniert werden. Dazu können

Statistiken über Häufigkeiten von Knoten herangezogen werden, die beispielsweise aus dem Durchlauf einer Testsuite gewonnen werden können.

Für jeden kommutativen Knoten in einem Muster verdoppelt sich die Anzahl an Regeln, die aufgeschrieben werden müssen, sofern keine Normalisierungsregeln existieren. Um diesen Aufwand zu reduzieren, kann über die automatische Generierung aller kommutativen Varianten zu einer Regeln nachgedacht werden.

Literaturverzeichnis

[Aho08] Aho, Alfred V. (Hrsg.): *Compiler : Prinzipien, Techniken und Werkzeuge*. München [u.a.] : Pearson Studium, 2008 (it Informatik)

[Bro] Brokken, Frank B.: *bison c++*. http://bisoncpp.sourceforge.net

[TLB99] Trapp, Martin ; Lindenmaier, Götz ; Boesler, Boris: Documentation of the Intermediate Representation FIRM / Universität Karlsruhe, Fakultät für Informatik. Version: Dec 1999. http://www/papers/firmdoc.ps.gz. Universität Karlsruhe, Fakultät für Informatik, Dec 1999 (1999-14). – Forschungsbericht

[Tra01] Trapp, Martin: *Optimierung objektorientierter Programme. Übersetzungstechniken, Analysen und Transformationen.*, University of Karlsruhe, Faculty of Informatik, Diss., Oct. 2001. http://www.amazon.de/exec/obidos/ASIN/3540423524/qid=1039790257/sr=1-1/