Universität Ulm, Fakultät für Informatik
Abteilung Programmiermethodik und Compilerbau
Leiter: Prof. Dr. Helmut Partsch

Erweiterte Compilierung von C/C++

Verbesserte Code-Optimierung durch Verwendung semantischer Zusatzinformationen

Dissertation
zur Erlangung des Doktorgrades
Doktor der Naturwissenschaften
(Dr. rer. nat.)
der Fakultät für Informatik
der Universität Ulm

vorgelegt von:

Mark Steffen Dettinger
aus Göppingen

2000

Amtierender Dekan:	Prof. Dr. Uwe Schöning
Gutachter:	Prof. Dr. Helmut Partsch
Gutachter:	Prof. Dr. Uwe Schöning
Tag der Promotion:	

Danksagung

An dieser Stelle möchte ich mich bei allen bedanken, die mich bei der Erstellung dieser Arbeit direkt oder indirekt unterstützt haben.

Mein besonderer Dank gilt meinem Betreuer Prof. Dr. Helmut Partsch, dem Leiter der Abteilung Compilerbau und Programmiermethodik, für seine großartige Unterstützung und die vielen Anregungen und Ratschläge während der letzten zwei Jahre.

Für sein Interesse an dieser Arbeit und das Zweitgutachten danke ich Prof. Dr. Uwe Schöning, dem Leiter der Abteilung Theoretische Informatik.

Mein Dank gilt außerdem der CAD-UL AG, insbesondere Peter Horn und Peter Lachner, für die finanzielle Ermöglichung dieser Arbeit und die Bereitstellung ihres Compiler-Quellcodes.

Für verschiedene Anregungen, Verbesserungsvorschläge, Literaturhinweise und die konstruktive Kritik an meiner Arbeit danke ich Stefan Sarstedt, Thilo Schwinn, Daniel Schwinn und Walter Guttmann. Für gelegentliche Tips im Umgang mit dem Textsatzprogramm LaTeX möchte ich mich bei Ulrike Schill und Jochen Rütschlin bedanken.

Schließlich danke ich Sandra, Karin und meinen Eltern für ihre Unterstützung meiner Aktivitäten.

Inhaltsverzeichnis

Kapitel 1 Einleitung

Diese Arbeit behandelt die Quellcode-Optimierung von C-Programmen, die mit semantischen Zusatzinformationen angereichert wurden. Es wird gezeigt, welche Arten von Zusatzinformationen für einen Compiler nützlich sind und wie er diese mit Verfahren, die bisher nur in der transformationellen Programmentwicklung Verwendung fanden, zu Optimierungen nutzen kann. Insbesondere wird die unseres Wissens erste Implementierung der Optimierung *Symbolic Finite Differencing* [Pai90] in einem C++-Compiler ausführlich beschrieben.

1.1 Warum überhaupt Codeoptimierung?

Die Geschwindigkeit der Rechner hat sich in den letzten Jahrzehnten alle 18 Monate verdoppelt. Ein heutiger PC besitzt damit etwa die tausendfache Leistungsfähigkeit eines preislich vergleichbaren PCs von 1985. Angesichts dieser Leistungssteigerung ist die Frage berechtigt, ob Codeoptimierung heutzutage überhaupt noch ein Thema ist. Sind die Rechner etwa nicht schon schnell genug, um auch schlechten Code in annehmbarer Zeit auszuführen?

Gute Codeoptimierung wird jedoch nach wie vor benötigt, wie die folgenden Gründe verdeutlichen:

- Zum einen steigt die Komplexität der Anwendungen parallel zum Fortschritt der Hardware. Die besser werdende Hardware ermutigt die Software-Hersteller zur Integration von immer aufwendigeren Technologien in ihre Produkte (wie z.B. in jüngster Zeit Spracherkennung). Die neuesten Anwendungen gehen dabei immer an die aktuelle Leistungsgrenze der Hardware. Eine Firma, die durch Codeoptimierung die Laufzeit ihrer Software um 50% senken kann, kann folglich ihre Technologie 18 Monate früher auf den Markt bringen als es ansonsten möglich wäre. Weitere Verbeserungen auf dem Gebiet der Optimierung wirken sich somit direkt als Wettbewerbsvorteil aus.

- Werkzeuge, die automatisch Software generieren, sind auf dem Vormarsch.

Automatisch generierte Software hat jedoch erfahrungsgemäß ein viel höheres Optimierungspotential als handgeschriebene Software. Der im vorigen Absatz genannte Zeitvorsprung, den man durch Optimierung gewinnen kann, kann daher noch höher ausfallen.

- Eingebettete Systeme gewinnen ebenfalls an Bedeutung. Diese Systeme arbeiten jedoch oft mit alten Prozessoren, da man bei ihnen aus Kostengründen gerne auf den Einsatz der neuesten Hardware verzichtet. Diese Systeme kamen folglich nicht in den Genuss der Leistungssteigerung der letzten Jahre. Damit sie trotzdem zufriedenstellend laufen, muss hier die Software entsprechend schnell sein.

- Weitere Leistungssteigerungen der Hardware werden in den nächsten Jahrzehnten immer schwieriger zu erreichen sein, da die Miniaturisierung der Mikroprozessoren sich langsam der physikalisch möglichen Grenze nähert. Wenn sich die Intervalle für eine jeweilige Leistungsverdopplung aber verlängern, erhöht sich der durch Codeoptimierung erreichbare Zeitvorteil — und verwandelt sich in einen absoluten Vorteil, wenn die physikalische Grenze erreicht ist.

- Wie wir in dieser Arbeit sehen werden, kann gute Codeoptimierung die Laufzeit auch um Größenordnungen verbessern, d.h. die Zeitkomplexität eines Algorithmus reduzieren. Dies kann von keiner noch so guten Hardware-Verbesserung erreicht werden.

- Ein von vielen Software-Ingenieuren vertretener Standpunkt ist schließlich, dass der Programmierer vor allem auf die Korrektheit und Wartbarkeit der Software achten sollte. Wenn er sich keine Gedanken mehr um Effizienz machen müsste, weil der Compiler ihm diese Aufgabe abnimmt, so würde dies die Software-Entwicklungszeiten und -kosten sowie die Zahl der in der Software enthaltenen Fehler erheblich senken.

1.2 Existierende Systeme

Zur Optimierung von Programmen existieren verschiedene Arten von Software-Systemen, zu denen außer den Compilern auch die Transformationssysteme und die Termersetzungssysteme zählen. Jedes dieser Systeme hat spezielle Eigenschaften, die es für jeweils andere Einsatzzwecke besonders geeignet machen. M. S. Feather stellt in [Fea87] mehrere Systeme vor und vergleicht die mit ihnen verfolgten Ziele.

1.2.1 Optimierende Compiler

Das meistverwendete System ist der optimierende Compiler. Er läuft vollautomatisch ab und wird insbesondere zur Optimierung imperativer Programme verwendet. Um auch die in der Praxis vorkommenden großen Programme von mehreren Millionen Codezeilen bewältigen zu können, ist Geschwindigkeit eine wichtige Anforderung. Dies erfordert den Einsatz von effizienten Algorithmen.

Man unterscheidet in Compilern zwischen der maschinenunabhängigen Optimierung des Quellprogramms und der maschinenabhängigen Optimierung des erzeugten Maschinencodes. Im Rahmen dieser Arbeit wird die erstere behandelt — die Quellcode-Optimierung, welche das höhere Optimierungspotential besitzt.

Zur Quellcode-Optimierung existiert ein inzwischen etabliertes Standard-Repertoire an Techniken, von denen die wichtigsten in Kapitel 2 vorgestellt werden.

Ein üblicher Compiler kann ein Programm typischerweise um einen Faktor zwischen 1 und 10 beschleunigen.

1.2.2 Transformationssysteme

Transformationssysteme sind interaktive Werkzeuge zur Programmentwicklung. Mit ihnen versucht der Benutzer, ein Programm durch sukzessive Anwendung semantikerhaltender Transformationen zu verbessern [PS83]. Man unterscheidet dabei üblicherweise drei Phasen der Programmentwicklung, nämlich

- die Herleitung eines funktionalen Programms aus einer Spezifikation duch Eliminierung deskriptiver Konstrukte,
- die Optimierung eines funktionalen Programms, z.B. durch Vereinfachung der Rekursionsstruktur,
- die Umwandlung eines funktionalen Programms in ein imperatives Programm.

Ein Transformationssystem kann dabei für eine oder mehrere dieser Phasen vorgesehen sein.

Durch Programmtransformation ist bei manchen Programmen eine Laufzeitverbesserung um Größenordnungen möglich, d.h. es kann eine echte Reduktion der Komplexität erreicht werden.

Allerdings benötigt der Benutzer dazu Intuition und Erfahrung. Da die Verfahren nur eingeschränkt automatisierbar sind, fanden sie bisher noch keine Verwendung in optimierenden Compilern.

Dazu kommt, dass imperative Programme aufgrund ihrer fehlenden referentiellen Transparenz bedeutend schwieriger zu handhaben sind als funktionale Programme. Insbesondere eine Sprache wie C, in der es Seiteneffekte, Zeiger und Kontrollstrukturen wie `goto` oder `setjmp/longjmp` gibt, macht den Einsatz der transformationellen Programmentwicklung besonders schwer. Martin Illsley behandelt in [Ill88] die Programmtransformation für eine Pascal-ähnliche imperative Sprache, in der Zuweisungen und Schleifen, aber keine Zeiger und `goto`-Anweisungen vorkommen.

Eine umfassende Einführung in die transformationelle Software-Entwicklung findet man in [Par90].

Beispiele für existierende Transformationssysteme sind das am Kestrel Institute entwickelte KIDS [Smi90] oder das an der TU München entworfene System CIP-S [BBB+85, BEH+87].

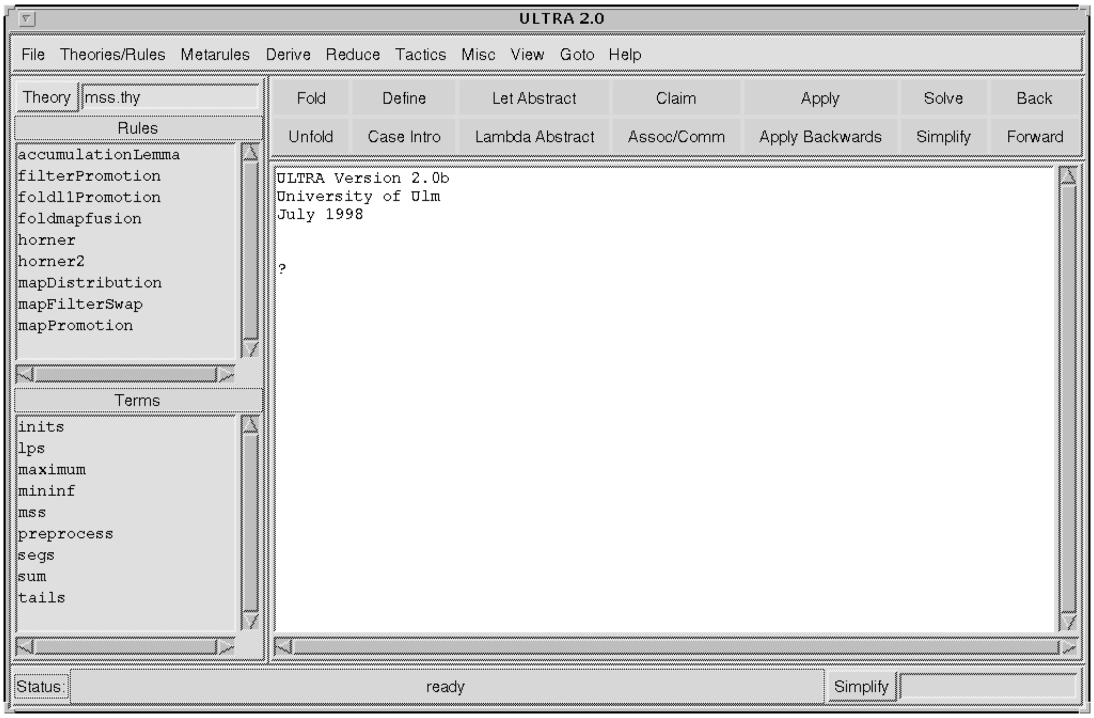


Abbildung 1.1: *Ulmer Transformationssystem ULTRA 2.0*

Seit 1997 wird an der Universität Ulm das Transformationssystem ULTRA [Det97, Sch98, Que98] entwickelt, mit dem Programme der Sprache *Gofer* (einer Haskell-Variante) transformiert werden können (Abb. 1.1). Die aktuelle Version ist frei erhältlich auf der Web-Seite http://www.informatik.uni-ulm.de/pm/ftp/ultra.html.

1.2.3 Termersetzungssysteme

Termersetzungssysteme kann man als automatische, eingeschränkte Transformationssysteme sehen. Sie bestehen aus einer Menge von Ersetzungsregeln, die wiederholt auf die Teilterme eines gegebenen Ausdrucks angewandt werden, bis keine Regel mehr angewandt werden kann.

Sie eignen sich damit hervorragend zur automatischen Vereinfachung von Ausdrücken. Das folgende Termersetzungssystem kann z.B. zur Vereinfachung von logischen Ausdrücken verwendet werden.

$$
\begin{array}{rcl}
and(x, true) & \rightarrow & x \\
and(x, false) & \rightarrow & false \\
or(x, true) & \rightarrow & true \\
or(x, false) & \rightarrow & x \\
not(not(x)) & \rightarrow & x
\end{array}
$$

$$\begin{aligned}
not(and(x,y)) &\rightarrow or(not(x), not(y))\\
not(or(x,y)) &\rightarrow and(not(x), not(y))\\
not(true) &\rightarrow false\\
not(false) &\rightarrow true
\end{aligned}$$

Angewandt auf den Ausdruck $not(and(x, false))$ würde das System — abhängig von der Reihenfolge der angewandten Regeln — nach zwei oder drei Schritten das Ergebnis $true$ liefern.

Eine allgemeine Einführung in die Termersetzungssysteme wird in [DJ90] gegeben.

1.2.3.1 Terminierung und Konfluenz

Die für diese Arbeit relevante Eigenschaft der Termersetzungssysteme ist ihre Automatisierbarkeit. Während es bei einem Transformationssystem keine feste Strategie gibt, nach der die anzuwendenden Regeln ausgesucht werden müssen, um ein bestimmtes Ziel zu erreichen, können bei einem Termersetzungssystem beliebige anwendbare Regeln auf beliebige Teilterme eines Ausdrucks angewandt werden, bis dessen einfachste Form erreicht ist.

Allerdings müssen dafür Einschränkungen bei den Regeln hingenommen werden. Die notwendigen Eigenschaften, die das Termersetzungssystem besitzen muss, sind *Terminierung* und *Konfluenz*.

Terminierung Ein Termersetzungssystem heißt *terminierend*, wenn für jeden beliebigen Ausdruck nach einer endlichen Zahl von Schritten eine Form erreicht wird, in der keine weitere Ersetzung mehr möglich ist (eine sogenannte *Normalform*).

Um die Terminierung eines Systems sicherzustellen, genügt es zu zeigen, dass bei allen Regeln die rechte Seite bezüglich einer geschickt gewählten Terminationsordnung kleiner als die linke Seite ist.

Konfluenz Ein terminierendes Termersetzungssystem heißt *konfluent*, wenn die Normalform, zu der ein Ausdruck durch das System reduziert wird, unabhängig von der Reihenfolge der angewandten Regeln ist.

1.3 Motivation und Zielsetzung der Arbeit

Aus der transformationellen Programmentwicklung sind einige mächtige Optimierungsverfahren bekannt, die in optimierenden Compilern bisher nur sehr eingeschränkt Verwendung fanden. Dies liegt vor allem daran, dass ein Compiler vollautomatisch ablaufen sollte, die besagten Verfahren aber nur eingeschränkt automatisierbar sind, da sie oft Wissen über semantische Eigenschaften des Programms erfordern.

Der theoretische Teil dieser Arbeit bestand nun darin, zu untersuchen, welche Möglichkeiten sich in der Codeoptimierung ergeben, wenn dem Compiler entsprechende semantische

Zusatzinformationen über die zu compilierenden Programme gegeben werden können. Die Codeoptimierung unter Zuhilfenahme solcher Informationen wird *erweiterte Compilierung* genannt [DFG+94, Fea87].

Im praktischen Teil wurde ein kommerzieller, bisher jedoch nicht optimierender C/C++-Compiler, um eine Möglichkeit, semantische Zusatzinformationen zu spezifizieren, und einen Code-Optimierer erweitert. Dabei wurden sowohl die klassischen Algorithmen als auch einige neue Verfahren implementiert, welche die Zusatzinformationen verwenden und Optimierungen ermöglichen, zu denen andere Compiler nicht in der Lage sind.

Mit C wurde eine existierende, weit verbreitete, aber auch „schwierige" Quellsprache gewählt. Eventuell auftretende Schwierigkeiten sollten bewusst *nicht* durch eine selbst definierte eingeschränkte Quellsprache umgangen werden.

Der entstandene Compiler – in den folgenden Kapiteln mit EPCOT (Experimental Prototype Compiler Of Tommorow) bezeichnet – bewältigt auch große Programme (mehrere Millionen Codezeilen) und ist für den Einsatz in der Industrie geeignet. In einer modifizierten Version wird EPCOT bereits von einer in Ulm ansässigen Software-Firma vertrieben. In dieser Version sind die klassischen Optimierungen enthalten. Die Möglichkeit, Zusatzinformationen zu spezifizieren, wurde jedoch deaktiviert.

1.4 Überblick über die folgenden Kapitel

Kapitel 2 gibt zunächst eine Einführung in die klassische Codeoptimierung. Die elementaren Verfahren, die in heutigen Compilern verwendet werden, werden vorgestellt.

In Kapitel 3 werden die Grenzen der elementaren Verfahren aufgezeigt. Verschiedene Arten von Zusatzinformationen, die dem Compiler helfen würden, diese Grenzen zu überwinden, werden beschrieben. Weiterhin werden einige Möglichkeiten diskutiert, wie einem Compiler diese Zusatzinformationen vermittelt werden könnten.

Kapitel 4 beschreibt, wie die Zusatzinformationen in den Compilationsvorgang integriert werden können, und stellt die Vorteile und die möglichen Einsatzgebiete der erweiterten Compilierung vor.

Kapitel 5 erläutert die Implementierung von EPCOT, einem C/C++-Compiler, der auf einem kommerziellen Compiler basiert und dem über Pragma-Direktiven Zusatzinformationen gegeben werden können.

Die Erfahrungen, die mit diesem erweiterten Compiler gemacht wurden, werden schließlich in Kapitel 6 diskutiert.

Kapitel 2 Klassische Codeoptimierung

Dieses Kapitel stellt zunächst die elementaren Verfahren der Codeoptimierung vor, die in Compilern verwendet werden. Dabei wird nur kurz auf das jeweilige Prinzip und auf einige der zu beachtenden Probleme eingegangen. Eine ausführliche Einführung in die Codeoptimierung wird von S. Muchnick in [Muc97] gegeben.

Die Verfahren, die in den folgenden Abschnitten vorgestellt werden, heißen Constant Propagation, Common Subexpression Elimination, Dead Code Elimination, Loop Unrolling und Finite Differencing. Da diese Bezeichnungen inzwischen international etabliert sind, verzichten wir hier auf eine Übersetzung der Namen.

Der Geschwindigkeitsgewinn, den man mit diesen Optimierungen erzielen kann, mag auf den ersten Blick gering erscheinen. Auch bezweifeln viele zunächst, dass in einem Programm, das von einem „guten" Programmierer geschrieben wurde, überhaupt einmal eine dieser Optimierungen zur Anwendung kommt.

Optimierungsmöglichkeiten können jedoch sehr versteckt auftreten, wie in Abschnitt 2.2 gezeigt wird. Außerdem kann durch die Anwendung einer Optimierung ein Domino-Effekt ausgelöst werden, der viele weitere Optimierungen ermöglicht. Nach einem *Loop Unrolling* (Abschnitt 2.4) ergeben sich z.B. in der Regel viele Möglichkeiten für die Verfahren *Constant Propagation*, *Common Subexpression Elimination* und *Dead Code Elimination*.

Es sei hier außerdem noch angemerkt, dass die heutigen optimierenden Compiler auf eben diesen Techniken basieren und damit bereits gute Ergebnisse erzielen. In den späteren Kapiteln werden wir außerdem nochmal auf diese Verfahren zurückkommen und zeigen, wie mit Hilfe semantischer Zusatzinformationen ihre Leistungsfähigkeit noch gesteigert werden kann.

2.1 Constant Propagation

Constant Propagation ersetzt Variablen durch Konstanten, wann immer dies möglich ist, d.h. wenn garantiert werden kann, dass eine Variable an einer bestimmten Stelle nur einen möglichen Wert haben kann.

Durch die Ersetzung können Ausdrücke entstehen, die nur noch Konstanten (und keine Variablen mehr) enthalten. Diese Ausdrücke können dann zur Compile-Zeit ausgewertet werden. Dies wird mit *Constant Folding* bezeichnet und wird generell als Unterroutine der Optimierung *Constant Propagation* implementiert.

Betrachten wir als Beispiel folgendes Programm:

```
a = 75;
b = 37;
if (n > 0) { b = 2 * a; }
c = b;
```

Bei der Berechnung von `b=2*a` hat `a` immer den Wert 75. Der Ausdruck `2*a` kann also zur Compile-Zeit ausgewertet werden.

```
a = 75;
b = 37;
if (n > 0) { b = 150; }
c = b;
```

Bei der Zuweisung `c=b` kann dagegen, ohne den Kontext des Programmstücks zu kennen, nicht bestimmt werden, ob `b` hier den Wert 37 oder 150 haben wird, da dies vom Ausgang des Vergleichs `n>0` abhängt. Hier ist also keine Optimierung möglich.

2.2 Common Subexpression Elimination

Die *Common Subexpression Elimination* versucht, unnötige Mehrfachberechnungen zu vermeiden.

Dazu werden identische seiteneffektfreie Ausdrücke A und B im Programm gesucht, für die gilt, dass alle Ausführungspfade durch das Programm, die zu B führen, auch über A laufen. Weiterhin darf auf den möglichen Pfaden von A nach B keine der Variablen, von denen der Wert des Ausdrucks abhängt, verändert werden.

Ist dies erfüllt, so kann man vor A eine Zuweisung einfügen, die das Ergebnis in einer Variable speichert, und die Ausdrücke A und B durch diese Variable ersetzen.

```
x = a*b*c + t;
y = a*b*c + 5;
```

Der Ausdruck `a*b*c` wird hier doppelt berechnet. Nach der Common Subexpression Elimination erhalten wir

```
cse = a*b*c;
x = cse + t;
y = cse + 5;
```

Die gemeinsamen Teilausdrücke können dabei auch versteckt auftreten, wie das folgende Beispiel zeigt.

```
int a[10][20][30];
a[i][j][0] = 2;
a[i][j][1] = 3;
a[i][j][2] = 5;
```

Der allen drei Zuweisungen gemeinsame Teilausdruck ist die Basisadresse von `a[i][j]`. Sie wird dreimal berechnet, und der Aufwand dieser Berechnung ist nicht zu unterschätzen:

```
a[i][j] = a + i * 20*30*sizeof(int) + j * 30*sizeof(int)
```

Es lohnt sich daher, die Adresse nur einmal zu berechnen und in einer neu eingeführten Variablen abzuspeichern.

```
p = a[i][j];
p[0] = 2;
p[1] = 3;
p[2] = 5;
```

2.3 Dead Code Elimination

Unter *Dead Code* versteht man Code, der nicht zur eigentlichen Berechnung des Programms beiträgt. Dieser Code kann gelöscht werden.

```
{
  b = f(4);
  a = g(5);
  c = b+7;
  a = h(x);
  return a+c;
}
```

Die Anweisung `a = g(5)` kann hier gestrichen werden, da auf den berechneten Wert nie zugegriffen wird. Zwei Zeilen später wird nämlich `a` erneut gesetzt, dazwischen findet jedoch kein Lesezugriff auf `a` statt. Die erste Zuweisung ist daher unnötig. Das verbesserte Programm lautet:

```
{
  b = f(4);
  g(5);
  c = b+7;
  a = h(x);
  return a+c;
}
```

Die Anweisung `g(5)` kann auch komplett gelöscht werden, sofern garantiert werden kann, dass `g(5)` keine Seiteneffekte besitzt.

Die Eliminierung redundanter Zuweisungen benötigt eine Analyse über die an jedem Punkt des Programms *lebenden Variablen*. Eine Variable gilt an einer gegebenen Programmstelle A als lebend, wenn ein Pfad von A zu einer Programmstelle B existiert, an der die Variable verwendet wird, und die Variable auf dem Pfad nicht redefiniert wird.

Eine genaue Beschreibung dieser Analyse wird in [NNH99] gegeben. Das Grundprinzip funktioniert folgendermaßen: Man geht das Programm in Rückwärtsrichtung durch, d.h. entgegen der Richtung der Kanten des Kontrollflussgraphen. Variablen beginnen bei Lesezugriffen zu leben und sterben bei Schreibzugriffen. Zuweisungen an tote Variablen sind dann redundant.

1. `return a+c`: a und c beginnen zu leben.
2. `a = h(x)`: a stirbt, x beginnt zu leben.
3. `c = b+7`: c stirbt, b beginnt zu leben.
4. `a = g(5)`: a würde sterben, ist aber bereits tot. Eine redundante Zuweisung wurde gefunden.

Zu beachten ist die Reihenfolge der Ereignisse: bei `c = c+1` stirbt `c` zuerst und beginnt dann wieder zu leben. Nach Abarbeitung der Anweisung ist `c` also eine lebende Variable.

Bei Feldern erweckt ein Lesezugriff auf ein unbekanntes Element (`a[i]` bei unbekanntem `i`) das ganze Feld zum Leben. Nach einem Lesezugriff auf eine komplett unbestimmte Variable (`*p` bei unbekanntem `p`) müssen alle Variablen als lebend betrachtet werden, um falsche Optimierungen zu vermeiden.

Die Information über die aktuell lebenden Variablen wird, wie oben gesagt, rückwärts durch das Programm propagiert. Aus diesem Grund ordnet man die Eliminierung redundanter Zuweisungen den Rückwärts-Optimierungen zu.
Im Gegensatz dazu gehören *Constant Propagation* und *CSE Elimination* zu den Vorwärts-Optimierungen, da deren zugehörige Programmanalysen die Information (die aktuelle Variablenbelegung bzw. die Menge der verfügbaren Ausdrücke) vorwärts durch das Programm propagieren.

2.4 Loop Unrolling

Im Vergleich zu anderen Instruktionen sind Sprünge extrem zeitaufwendig. Dies gilt vor allem in den neueren Prozessoren mit Pipeline-Verarbeitung. Diese Prozessoren können linearen Code sehr schnell ausführen. Nach einem Sprung muss die Pipeline jedoch wieder neu gefüllt werden, was Zeit kostet.

Wird eine Schleife nur eine kleine konstante Zahl oft durchlaufen, empfiehlt es sich daher, die Schleife zu entrollen. Dazu wird die Schleife durch die Initialisierung und die entsprechend lange alternierende Sequenz von Schleifenrumpf und Schleifeninkrement ersetzt.

Aus dem Programm

```
for (i=0; i<5; i++) f(i);
```

wird dabei

```
i=0; f(i); i++; f(i); i++; f(i); i++; f(i); i++; f(i); i++;
```

Mittels *Constant Propagation* lässt sich dies weiter vereinfachen zu

```
i=0; f(0); i++; f(1); i++; f(2); i++; f(3); i++; f(4); i++;
```

Eine abschließende *Dead Code Elimination* ergibt schließlich

```
f(0); f(1); f(2); f(3); f(4);
```

Ein Problem beim *Loop Unrolling* ist das eventuell sehr starke Anwachsen der Programmlänge. Hier ist dann gegebenenfalls zwischen Geschwindigkeit und Speicherplatzverbrauch abzuwägen.

Um die Korrektheit der Optimierung zu gewährleisten, muss der Compiler sicherstellen, dass der Schleifenindex im Rumpf nicht verändert werden kann - was auch versteckt über Zeiger oder in Unterroutinen geschehen könnte.

2.5 Finite Differencing

Finite Differencing ist auch unter dem Namen *Strength Reduction* bekannt. Der Grundgedanke dieses Verfahrens ist, in einer sich wiederholenden Berechnung, in der sich pro Iteration nur wenige Parameter leicht verändern, das Ergebnis der letzten Iteration wiederzuverwenden.

2.5.1 Historische Ursprünge

Wie Paige in [Pai90] erklärt, geht die Technik des *Finite Differencing* zurück auf Henry Briggs, einen Mathematiker des 16. Jahrhunderts.

Briggs stand vor der zeitraubenden Aufgabe, ein Polynom $p_0(x)$ vom Grad d an den Punkten $x_0, x_0 + \Delta, x_0 + 2\Delta, x_0 + 3\Delta, ...$ zu berechnen. Er entdeckte, dass das Polynom $p_i(x) = p_{i-1}(x + \Delta) - p_{i-1}(x)$ einen Grad von höchstens $d - i$ hat und damit $p_d(x)$ eine Konstante ist. Mittels Tabellierung konnte er damit die Werte von $p_i(x+\Delta)$ durch lediglich d Additionen aus den Werten von $p_i(x)$ (für $i = 0, 1, ..., d$) berechnen. (Eine Berechnung der Funktionswerte von $p_0(x)$ mittels Horner-Schema würde dagegen d Additionen und d Multiplikationen für jeden Punkt benötigen.)

Beispiel: $p_0(x) = x^3, x_0 = 0, \Delta = 1$

$$\begin{aligned}
p_0(x) &= x^3 \\
p_1(x) &= 3x^2 + 3x + 1 \\
p_2(x) &= 6x + 6 \\
p_3(x) &= 6
\end{aligned}$$

x	0	1	2	3	4	5	6
$p_3(x)$	6	6	6	6	6	6	6
$p_2(x)$	6	12	18	24	30	36	42
$p_1(x)$	1	7	19	37	61	91	127
$p_0(x)$	0	1	8	27	64	125	216

Tabelle 2.1: *Berechnung des Polynoms $p_0(x) = x^3$ durch Finite Differencing. Es gilt die Rekursion $p_{i-1}(x + \Delta) = p_i(x) + p_{i-1}(x)$, d.h. jede Zahl der Tabelle ergibt sich aus der Summe der Zahlen links neben und links über ihr.*

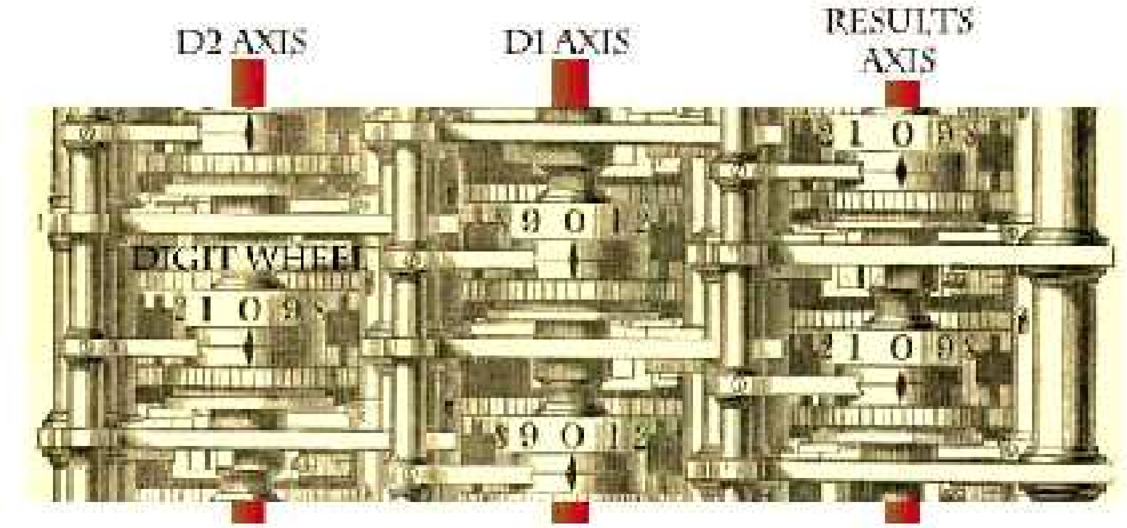


Abbildung 2.1: *Difference Engine No. 1 von Charles Babbage [Jon97]. Die Resultats-Achse zeigt die Werte des zu berechnenden Polynoms p_0 an, die D1-Achse die ersten Differenzen (p_1), die D2-Achse die zweiten Differenzen (p_2).*

Drei Jahrhunderte später baute Charles Babbage seine Maschine *Analytic Difference Engine*, die Briggs' Algorithmus ausführte, nachdem ihre Register mit den Werten von $p_i(x_0), i = 0, ..., d$ geladen worden waren (Abb. 2.1).

2.5.2 Anwendung in optimierenden Compilern

In optimierenden Compilern kann *Finite Differencing* eingesetzt werden, um die sukzessive Berechnung eines Polynoms innerhalb einer Schleife zu optimieren.

```
for (i=0, i<1000; i++)
  a[i] = i*i*i;
```

Wir ersetzen den Ausdruck `i*i*i` durch eine neue Variable `x`. `x` wird mit 0 initialisiert und hat damit in der ersten Iteration den richtigen Wert. Damit die Invariante $x = i^3$ in jeder Iteration erhalten bleibt, muss nun an der Stelle `i++` eine entsprechende Zuweisung an `x` eingefügt werden.

Es gilt: $(i+1)^3 - i^3 = 3i^2 + 3i + 1$.
Daraus ergibt sich das notwendige Update für `x`: `x += 3*i*i+3*i+1`.

```
for (i=0, x=0; i<1000; x+=3*i*i+3*i+1; i++)
  a[i] = x;
```

Wir haben nun eine Schleife, in der nur noch ein Polynom zweiten Grades berechnet wird. Das Verfahren wird nun rekursiv auf den Ausdruck `3*i*i+3*i+1` angewendet.

Wir ersetzen den Ausdruck durch eine weitere Variable `y`, initialisieren `y` mit 1 und fügen an der Stelle `i++` eine weitere Zuweisung ein, welche die neue Invariante $y = 3i^2 + 3i + 1$

erhält. Die dazu notwendige Zuweisung ergibt sich aus der Formel $(3(i+1)^2 + 3(i+1) + 1) - (3i^2 + 3i + 1) = 6i + 6$ zu `y += 6*i+6`.

```
for (i=0, x=0, y=1; i<1000; x+=y; y+=6*i+6; i++)
  a[i] = x;
```

Im dritten Schritt wird das Verfahren schließlich auf den Ausdruck `6*i+6` angewendet und wir erhalten das fertige Programm

```
for (i=0, x=0, y=1, z=6; i<1000; x+=y, y+=z, z+=6, i++)
  a[i] = x;
```

Anstelle von 2 Multiplikationen werden in diesem optimierten Programm pro Iteration 3 Additionen und 3 Zuweisungen ausgeführt. Auf den meisten Prozessoren bringt dies eine deutliche Zeitersparnis. Ein 80486-Prozessor benötigt beispielsweise 1-3 Taktzyklen für eine Addition oder Zuweisung, aber 13-42 Taktzyklen (abhängig von den zu multiplizierenden Zahlen) für eine Multiplikation [Nel91].

Kapitel 3 Semantische Zusatzinformationen

Programme besitzen oft Optimierungspotential, welches von einem üblichen Compiler nicht erkannt wird. Schuld daran ist das fehlende Wissen des Compilers über semantische Eigenschaften des zu übersetzenden Programms.

Eine Idee, dieses Problem zu lösen, ist, dem Compiler zusätzlich zum Quellcode weitere Informationen zur Verfügung zu stellen, die ihm dieses Wissen vermitteln.

Abschnitt 3.1 zeigt die Grenzen herkömmlicher Compiler auf. In Abschnitt 3.2 werden dann verschiedene Arten von Informationen beschrieben, die dem Compiler helfen würden, diese Grenzen zu überwinden. Wie die Übergabe der Informationen an den Compiler aussehen könnte, wird in Abschnitt 3.3 diskutiert. Wie sie letztendlich in EPCOT realisiert wurde, wird schließlich in Abschnitt 3.4 erläutert.

3.1 Grenzen heutiger Compiler

In vielen Fällen ist ein Programmstück zwar optimierbar, doch ist dazu semantisches Wissen erforderlich, welches der Compiler nicht hat.

Ein einfaches Beispiel dafür ist der Funktionsaufruf `pow(x,1.0)`, wobei `pow` hier die Potenzierungsfunktion aus der C-Standardbibliothek <math.h> sei. Natürlich ist dies zu `x` optimierbar. Um dies zu erkennen, müsste der Compiler jedoch die entsprechende Eigenschaft des Potenzierens kennen.

Es ist zwar kein Problem, dies als Spezialfall in den Compiler einzubauen, doch generell hat dieser Weg einige gravierende Nachteile:

- Erstens gibt es viele Fälle der obigen Art, z.B. `pow(x,0.0)`, `sin(0.0)`, `cos(0.0)`, `strlen("")` oder `printf("")`. Es wäre sehr zeitaufwendig, die entsprechenden Optimierungen für alle diese Spezialfälle in den Compiler einzubauen.
- Zweitens ist es gefährlich, anzunehmen, dass eine Funktion gewisse Eigenschaften besitzt, nur weil sie `pow` heißt und aus einer Datei namens <math.h> stammt. Würde sich der Benutzer eine eigene Funktion definieren, die zufällig

den gleichen Namen trägt, aber einen ganz anderen Zweck erfüllt, so würde der Compiler von falschen Voraussetzungen ausgehen und falschen Code generieren.

- Drittens könnte man auf diese Weise lediglich spezielle Optimierungen für die Standardbibliothek definieren. Schöner wäre es aber, wenn der Benutzer auch für selbstdefinierte Funktionen dem Compiler die für sie geltenden Eigenschaften nennen könnte.

3.2 Nützliche Zusatzinformationen

In diesem Abschnitt soll gezeigt werden, welche Zusatzinformationen über das zu compilierende Programm dem Compiler helfen würden.

3.2.1 Eigenschaften von Funktionen

3.2.1.1 Seiteneffekte

Unter einem *Seiteneffekt* verstehen wir im folgenden eine Änderung des „Berechnungszustandes" eines Programms. Es gibt zwei Arten von solchen Änderungen:

- Ein- und Ausgaben des Programms (`printf`, `scanf`, `fprintf`,...).
- Modifikationen von Variablen oder allokierten Speicherzellen, wobei diese direkt (`=`, `++`, `--`, `+=`, `*=`,...) oder indirekt über Bibliotheksfunktionen (`strcpy`) verursacht werden können.

Wir sagen, dass eine Funktion einen Seiteneffekt besitzt, wenn sie eine dieser Operationen durchführt und der Effekt auch nach dem Verlassen der Funktion noch sichtbar ist. Konkret heißt dies, dass wir Änderungen von *lokalen* Variablen *nicht* als Seiteneffekte betrachten. Ebenso sehen wir Modifikationen von allokierten Speicherzellen nur dann als Seiteneffekt an, wenn diese Zellen auch nach dem Verlassen der Funktion noch allokiert und erreichbar sind.

Das mögliche Vorhandensein eines Seiteneffekts kann den Compiler von einer Optimierung abhalten. Wenn in einem zu optimierenden Codestück eine Funktion aufgerufen wird, kann z.B. im allgemeinen nicht garantiert werden, dass nach dem Aufruf die aktuellen Annahmen über Werte von Variablen noch gelten.

```
a = 5;
f();
b = a;
```

Nur in Spezialfällen kann der Compiler hier `b = a` durch `b = 5` ersetzen, beispielsweise wenn `a` eine lokale Variable ist, deren Adresse an keiner Stelle genommen wird. Ansonsten ist generell nicht auszuschließen, dass `a` in `f` verändert wird.

Eventuell kann der Compiler noch mittels interprozeduraler Datenflussanalyse erkennen, ob `a` in `f` wirklich verändert werden kann. Unlösbar wird dieses Problem jedoch, wenn der Compiler den Quellcode von `f` gar nicht kennt, was z.B. bei allen Bibliotheksfunktionen der Fall ist.

```
a = 5;
x = sin(1.2);
b = a;
```

Wenn für den Programmierer hier auch offensichtlich ist, dass der Wert von `a` nicht von der Funktion `sin` verändert wird (sofern es sich um die Sinusfunktion aus der C-Standardbibliothek `<math.h>` handelt), kann der Compiler im allgemeinen nicht davon ausgehen. Wenn ihm die Seiteneffektfreiheit von `sin` jedoch als Zusatzinformation gegeben werden würde, so könnte er die Optimierung durchführen.

Ein anderer interessanter Fall tritt auf, wenn eine Funktion zwar Seiteneffekte hat, diese aber keinen Einfluss auf das restliche Programm haben. Dies ist z.B. der Fall, wenn eine Funktion zwar globale Datenstrukturen manipuliert oder temporäre Dateien liest und schreibt, aber nur, um Nebenrechnungen durchzuführen. Der Programmierer könnte eine solche Funktion als seiteneffektfrei oder sogar als transparent deklarieren und damit dem Compiler helfen.

3.2.1.2 Monotonie

Eine weitere Eigenschaft der Funktion `sin`, die für gewisse Optimierungen benötigt wird, ist ihre Monotonie.

Definition: Wir nennen eine Funktion *monoton*, wenn sie in jedem Kontext das selbe Ergebnis liefert, wann immer sie mit den selben Argumenten aufgerufen wird. Oder anders ausgedrückt: wenn ihr Rückgabewert nur von ihren Parametern abhängt, nicht aber von einem globalen Zustand.

Im Gegensatz zu einer seiteneffektfreien Funktion darf eine monotone Funktion Zuweisungen an globale Variablen durchführen. Sie darf dagegen nicht ihren Rückgabewert vom Wert einer globalen Variablen abhängig machen.

```
globalFlag = 0;
x = sin(1.2);
globalFlag = 1;
y = sin(1.2);
```

Um hier `y = sin(1.2)` zu `y = x` zu optimieren, reicht die Seiteneffektfreiheit von `sin` nicht aus. Theoretisch könnte ja die Variable `globalFlag` einen Einfluss auf die Berechnung haben, so dass der zweite Aufruf ein anderes Ergebnis liefert. Es ist die Monotonie von `sin`, die hier die Korrektheit der Optimierung garantiert.

Seiteneffektfreiheit und Monotonie ergeben zusammen die aus funktionalen Sprachen bekannte *referentielle Transparenz*.

Definition: Eine Funktion heißt *transparent*, wenn sie sowohl monoton als auch seiteneffektfrei ist.

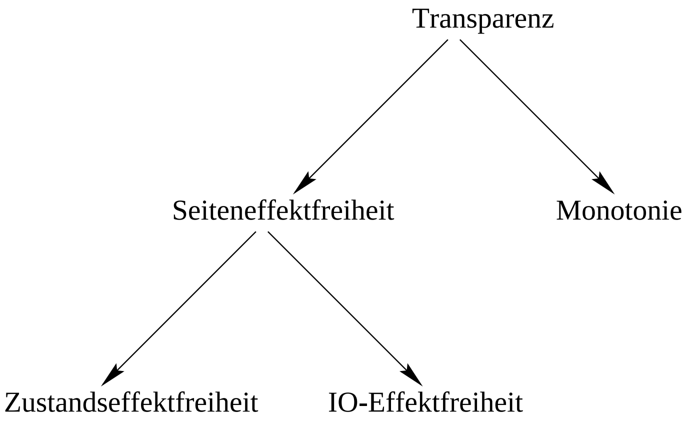


Abbildung 3.1: *Hierarchie der Eigenschaften von Funktionen (Pfeile stehen für logische Implikation)*

3.2.1.3 Klassifizierung von Seiteneffekten

Betrachten wir das folgende Codestück:

```
a = 5;
printf("%d",a);
b = a;
```

Würde man `printf` (fälschlicherweise) als seiteneffektfrei deklarieren, so würde dies hier dem Compiler zwar die Optimierung `b = 5` ermöglichen, doch er könnte dann auch die komplette Anweisung `printf("%d",a)` eliminieren, da sie ja angeblich keinen Seiteneffekt hat, also sowieso nichts bewirkt.

Lösen lässt sich das Dilemma durch eine genauere Klassifizierung von Seiteneffekten. Die Funktion `printf` ist zwar nicht seiteneffektfrei, da sie eine Ausgabe bewirkt, doch der globale Zustand wird von ihr nicht verändert. Man kann also unterscheiden zwischen *zustandseffektfreien* Funktionen, die keine Variablen verändern, und *IO-effektfreien* Funktionen, welche keine Ein- oder Ausgaben erzeugen. `printf` ist also eine zustandseffektfreie, nicht aber eine IO-effektfreie Funktion. Die obige Optimierung `b = 5` benötigt als Voraussetzung lediglich ersteres, das Eliminieren der kompletten Anweisung benötigt dagegen beide Vorausssetzungen.

Wir erhalten somit die in Abb. 3.1 gezeigte Hierarchie von Eigenschaften, die man den Funktionen eines Programms zuordnen kann. Für einen optimierenden Compiler wäre es hilfreich, zu wissen, welche Eigenschaften die einzelnen Funktionen erfüllen.

3.2.2 Transformationsregeln

Um dem Compiler z.B. zu sagen, dass er einen Ausdruck der Form `power(N,0)`, wobei `int power(int,int)` hier die Potenzierungsfunktion für ganze Zahlen sei und `N` für einen beliebigen seiteneffektfreien C-Ausdruck steht, durch `1` ersetzen kann, eignen sich *Transformationsregeln.*

Eine formale Einführung in den Transformationskalkül wird in [Pep84] gegeben. Ohne alle technischen Details wiederzugeben, sollen hier jedoch nur die wichtigsten Begriffe erläutert werden.

Transformationsregeln sind Inferenzen der Form `C` $\Rightarrow$ `Rel(I,O)`. Hierbei ist `C` eine Bedingung, unter der zwischen den Ausdrücken `I` und `O` die Relation `Rel` gilt. `I` und `O` können dabei implizit allquantifizierte freie Variablen enthalten, sogenannte *Schemavariablen*, die ihrerseits für beliebige Ausdrücke stehen können. `I` und `O` stellen daher eigentlich eine Verallgemeinerung von Ausdrücken, sogenannte *Ausdrucksschemata*, dar. Man nennt `C` die Anwendbarkeitsbedingung, `I` das *Eingabeschema* und `O` das *Ausgabeschema* der Transformationsregel. Typische in der Programmtransformation verwendete Relationen für `Rel` sind „ist äquivalent zu" oder „ist ersetzbar durch".

Die oben angesprochene Ersetzungsmöglichkeit lässt sich als Transformationsregel also folgendermaßen schreiben:

$$\mathit{SideEffectFree}(N) \Rightarrow \mathit{power}(N,0) \equiv 1$$

Die freie Variable `N` ist hier wie gesagt implizit allquantifiziert und steht für einen beliebigen Ausdruck der Programmiersprache C.

Die Anwendbarkeitsbedingung ist notwendig, damit nicht versehentlich Seiteneffekte wegoptimiert werden. Wie man leicht sieht, darf z.B. `power(i++,0)` *nicht* durch `1` ersetzt werden.

3.2.2.1 Anwendbarkeitsbedingungen

Das Stellen der korrekten Anwendbarkeitsbedingung für eine Transformationsregel in C verdient gesonderte Betrachtung.

Wir fordern, dass die Semantik des Programms durch die lokale Anwendung einer Regel nicht verändert werden darf. Da wir Transformationsregeln für C formulieren wollen, verwenden wir die vom ANSI-Kommitee definierte Semantik von C.

Wie oben gesagt, darf der Ausdruck `power(n,0)` nur dann durch `1` ersetzt werden, wenn `n` ein seiteneffektfreier Ausdruck ist. `power(n,1)` darf dagegen immer durch `n` ersetzt werden, auch dann, wenn `n` einen Seiteneffekt hat.

$$\begin{array}{rcl} \mathit{SideEffectFree}(N) & \Rightarrow & \mathit{power}(N,0) \equiv 1 \\ \mathit{True} & \Rightarrow & \mathit{power}(N,1) \equiv N \end{array}$$

Der entscheidende Unterschied zwischen den beiden Regeln liegt darin, dass in der zweiten Regel die Schemavariable N sowohl im Eingabe- als auch im Ausgabeschema auftritt. Sollte N einen Seiteneffekt besitzen, so würde dieser Seiteneffekt nach Anwendung der Regel immer noch auftreten.

Fordert man grundsätzlich die Seiteneffektfreiheit von allen im Eingabeschema vorkommenden Schemavariablen, ist man immer auf der sicheren Seite. Allerdings würde man damit einige Optimierungsmöglichkeiten verschenken.

Wir möchten daher im folgenden eine Vorgehensweise skizzieren, mit der entschieden werden kann, von welchen Schemavariablen die Seiteneffektfreiheit gefordert werden muss und von welchen nicht. Wir betrachten dazu die Regel

$$? \Rightarrow power(N, A + B) \equiv power(N, A) * power(N, B)$$

Wir bezeichnen die Seiteneffekte der an die Schemavariablen N, A und B gebundenen Ausdrücke mit S_N, S_A und S_B. Die Auswertungsreihenfolge der Operanden ist in ANSI C nur für die Operatoren `?:`, `&&` und `||` definiert. Ebenso ist die Reihenfolge, in der die Argumente einer Funktion ausgewertet werden, dem Compiler überlassen. Sollten die Ausdrücke N, A und B also alle Seiteneffekte besitzen, so könnten diese im Ausdruck $power(N, A + B)$ in vier möglichen Reihenfolgen auftreten: $S_N S_A S_B$, $S_N S_B S_A$, $S_A S_B S_N$ oder $S_B S_A S_N$. Ebenso sind beim Ausdruck $power(N, A) * power(N, B)$ verschiedene Reihenfolgen möglich (z.B. $S_A S_N S_B S_N$), die jedoch alle eines gemeinsam haben: S_N tritt zweimal auf.

Eine notwendige Korrektheitsbedingung ist jedoch, dass die Seiteneffekte mit und ohne Optimierung gleich oft auftreten. Es ist daher zumindest notwendig, die Seiteneffektfreiheit von N zu fordern, um die Korrektheit der Regel zu garantieren.

$$SideEffectFree(N) \Rightarrow power(N, A + B) \equiv power(N, A) * power(N, B)$$

Stellt man keine solche Forderung an A und B, so könnte sich nicht die Anzahl, sondern lediglich die Reihenfolge der Seiteneffekte ändern, wenn die Regel angewandt wird (von $S_A S_B$ nach $S_B S_A$ oder umgekehrt), falls der Compiler für Addition und Multiplikation unterschiedliche Auswertungsreihenfolgen verwendet.

Es gibt zwei Ansichten darüber, ob dies zulässig ist:

- Fordert man lediglich, dass sich das Programm mit und ohne Optimierung gemäß der Semantik von C laut ANSI-Spezifikation verhält, so ist es zulässig — denn die Reihenfolge der Operandenauswertung ist wie schon gesagt nun mal nicht definiert. Wer einen Ausdruck wie `power(n,(x=y)+(y=x))` verwendet, könnte dann eventuell ein unterschiedliches Verhalten beobachten, wenn er das optimierte und das nicht optimierte Programm ablaufen ließe. Beide Varianten wären aber korrekt.

- Fordert man dagegen zusätzlich, dass sich das Programm mit und ohne Optimierung gleich verhält, so ist es nicht zulässig. In diesem Fall müsste man dann auch noch die Seiteneffektfreiheit von A und B in die Anwendbarkeitsbedingung aufnehmen.

3.2.2.2 Syntaktische und semantische Prädikate

Es stellt sich die Frage, welche Prädikate notwendig sind, um Transformationsregeln für C formulieren zu können.

Ein wichtiges semantisches Prädikat wurde bereits im vorigen Abschnitt genannt: *SideEffectFree*. Mit diesem können viele Äquivalenzen, die in C allgemein keine Gültigkeit besitzen (wie $2 * x \equiv x + x$), in den entsprechenden Fällen anwendbar gemacht werden (hier: wenn $SideEffectFree(x)$ gilt).

In manchen Fällen sind auch syntaktische Prädikate hilfreich, beispielsweise, um die Anwendungsrichtung einer Ersetzungsregel zu steuern. Eine Regel wie

$$A * B \equiv B * A$$

mag zwar korrekt sein, ist aber für unsere Zwecke unbrauchbar, da sie immer wieder auf den selben Ausdruck anwendbar ist und damit zu einem nicht terminierenden Optimierer führt (sofern dieser nicht die Notbremse zieht, indem er Regeln, die zu oft angewendet wurden, deaktiviert). Möchte man aber die Regel nur so anwenden, dass Konstanten nach links und Variablen nach rechts gebracht werden (also um z.B. `x*3` durch `3*x` zu ersetzen), so könnte man dies schreiben als

$$Constant(C) \wedge Variable(X) \Rightarrow X * C \equiv C * X$$

$Constant$ und $Variable$ sind hier syntaktische Prädikate, die dann erfüllt sind, wenn ihr Argument von der entsprechenden Sorte ist.

Wünschenswert wäre weiterhin, wenn man auch gewöhnliche C-Ausdrücke als Bedingung stellen könnte. Die Regel

$$I < 8 \Rightarrow qsort(A, I, S, Cmp) \equiv bubblesort(A, I, S, Cmp)$$

würde beispielsweise besagen, dass ein Aufruf von `qsort` (der Quicksort-Funktion der C-Standardbibliothek) durch `bubblesort` ersetzt werden soll, sofern das Argument I, das die Anzahl der Elemente des zu sortierenden Feldes angibt, kleiner als acht ist.

3.2.3 Hardware-interne Informationen

Eine Optimierung, die eine Multiplikation eliminiert und dafür zwei Additionen einführt, macht nur dann Sinn, wenn die Multiplikation mehr als doppelt so lange dauert wie eine Addition. Dies hängt jedoch vom Prozessor der Zielmaschine ab. Ein Optimierer, der für einen speziellen Prozessor geschrieben wurde, kann daher auf einem anderen Prozessor den Code sogar verschlechtern.

Ideal wäre es, wenn man Informationen über die verwendete Hardware, wie z.B. die Zeiten, die für die elementaren Operationen benötigt werden, dem Compiler als Argument mitgeben könnte. Der Compiler hätte damit eine Tabelle der folgenden Art zur Verfügung:

Operation	Zeitbedarf
Zuweisung (Integer)	1
Addition (Integer)	1
Multiplikation (Integer)	13

Die Kosten eines Ausdrucks könnte man dann als Summe der Kosten aller in ihm enthaltenen Operatoren definieren. Die Anweisung `x = a*b+5` hätte damit einen Zeitbedarf von 15.

Dies ist allerdings nur eine Abschätzung, da der echte Zeitbedarf einer Operation auch von den Operanden (Variablen oder Konstanten) abhängen kann. Die Verwendung eines Cache oder die bei modernen Prozessoren übliche Pipeline-Verarbeitung machen die exakte Berechnung auf diese Weise sogar unmöglich, da hier noch berücksichtigt werden müsste, welche Ausdrücke der Prozessor bereits zuvor ausgewertet hat.

Als Richtwerte für den Vergleich von Ausdrücken sind die Ergebnisse der obigen Kostenfunktion jedoch durchaus geeignet.

Operation	8086/8088	68000	80486	K6
Integer-Addition	3-17	4-14	1-3	1
Integer-Multiplikation	80-160	70	13-42	3
Integer-Division	101-190	140-158	19-44	?

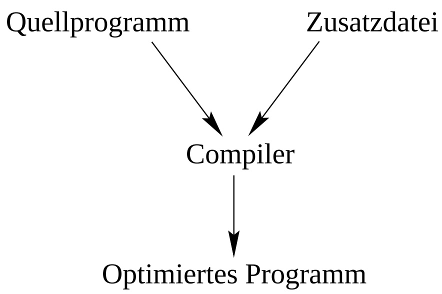
Tabelle 3.1: *Benötigte Taktzyklen einiger Operationen [Sca83, Cof84, Nel91]*

3.3 Übergabe der Informationen an den Compiler

Auf welche Art sollen dem Compiler nun diese Zusatzinformationen übergeben werden? Drei Möglichkeiten hierzu sollen in diesem Abschnitt vorgestellt werden.

3.3.1 Zusatzdatei

Der Compiler könnte zwei Dateien lesen: das Quellprogramm und - optional - eine Datei mit weiteren Informationen.

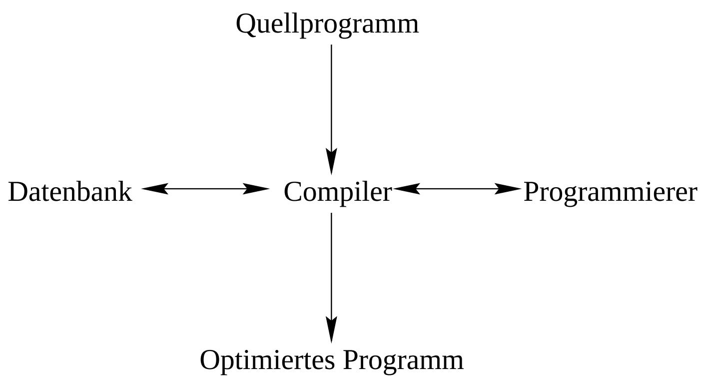


Die Aufteilung auf zwei Dateien fördert jedoch geradezu Konsistenzprobleme. Es steht zu befürchten, dass Programmierer den Quellcode verändern, aber vergessen, die dadurch evtl. ungültig gewordenen Informationen in der Zusatzdatei auf den neuesten Stand zu bringen.

3.3.2 Interaktion

Alternativ dazu könnte der Compiler während der Compilierung nach Zusatzinformationen fragen. Um den Benutzer dabei allerdings nicht unnötig zu nerven, müssten für den Einsatz in der Praxis bei dieser Lösung zwei Voraussetzungen erfüllt sein:

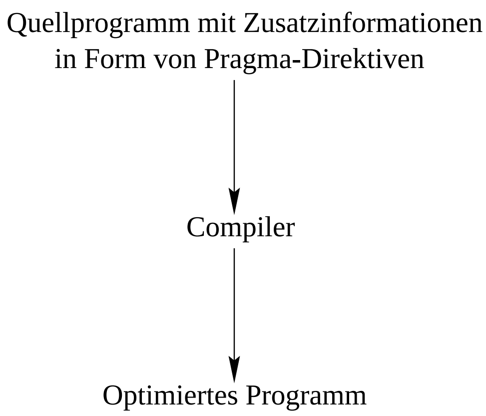
- Der Compiler sollte einmal gelernte Informationen in einer internen Datenbank speichern und nur dann neu erfragen, wenn sie sich seit der letzten Übersetzung geändert haben könnten.
- Der Compiler sollte nicht pauschal nach Informationen fragen, sondern gezielt und nur dann, wenn ihm die Antwort eventuell eine Optimierung ermöglichen würde.



Ein gravierender Nachteil der Verwendung einer Datenbank ist jedoch, dass die dem Compiler einmal gegebenen Informationen für den Benutzer unsichtbar sind. Es müsste ein Browser zur Verfügung stehen, mit dem kontrolliert werden kann, von welchen Informationen der Compiler zur Zeit ausgeht. Es sollte außerdem einen Editor geben, mit dem aus Versehen gegebene Falschinformationen wieder aus der Datenbank gelöscht werden können.

3.3.3 Pragma-Direktiven

Als dritte Möglichkeit kommen Pragma-Direktiven in Betracht. Diese sind in der Sprache C ohnehin dafür vorgesehen, dem Compiler zusätzliche Anweisungen zu geben. Sie eignen sich daher vorzüglich, um ein Programm mit Zusatzinformationen anzureichern.



Programmcode und Zusatzinformationen befinden sich bei dieser Lösung nahe beisammen in der selben Datei. Konsistenzprobleme fallen damit dem Programmierer eher auf und sind somit zumindest weniger wahrscheinlich als bei einer Aufteilung auf zwei Dateien.

Das angereicherte Programm ist weiterhin auch mit anderen Compilern übersetzbar, da unbekannte Pragma-Direktiven nach Konvention von einem Compiler ignoriert werden. (Es sei denn, der andere Compiler verwendet Pragmas des selben Namens und interpretiert sie anders.)

Die Lösung ist außerdem von einem programmiermethodischen Standpunkt aus interessant, da die Zusatzinformationen auch eine hervorragende Programmdokumentation darstellen. Wurde eine Funktion von ihrem Autor z.B. als transparent deklariert, so ist das nicht nur für den Compiler, sondern auch für andere Programmierer eine wertvolle Information.

3.4 Realisierung

Nach Abwägung der Vor- und Nachteile der vorgestellten Alternativen wurde in EPCOT die Lösung des Informationsübergabeproblems über die Pragma-Direktiven gewählt. Es folgt eine ausführliche Beschreibung der gewählten Syntax.

3.4.1 Eigenschaften von Funktionen

Eigenschaften von Funktionen werden mit dem Pragma `property` angegeben.

```
#pragma property Transparent(sin)
#pragma property Transparent(cos)
#pragma property StateEffectFree(printf)
```

Bisher existieren die Prädikate `SideEffectFree`, `StateEffectFree`, `IOEffectFree`, `Monotone` und `Transparent`.

3.4.2 Transformationsregeln

Die einfachsten in EPCOT formulierbaren Transformationsregeln sind Gleichheiten zwischen C-Ausdrücken. Da die Regeln allerdings nur von links nach rechts angewendet werden, stellen sie eigentlich Ersetzungsregeln und keine Gleichungen dar.

Um dem Compiler z.B. zu sagen, dass alle im Programm vorkommenden Ausdrücke der Form `cos(0.0)` zu `1.0` optimiert werden sollen, müsste man folgende Direktive ins Programm einfügen:

```
#pragma rule cos(0.0) === 1.0
```

Das neu eingeführte Symbol `===` steht hierbei für „ist ersetzbar durch“.

Die Regeln können auch allquantifizierte Schemavariablen erhalten, die ihrerseits für beliebige C Expressions stehen. Die Schemavariablen können dabei auf bestimmte Typen eingeschränkt werden. Die folgende Regel besagt, dass für alle Ausdrücke `n` vom Typ `int` die Ersetzung von `n*2` durch `n<<1` gültig ist.

```
#pragma rule ForAll int n : n * 2 === n << 1
```

In manchen Regeln will man den Schemavariablen dagegen keine Typbeschränkung auferlegen. Dafür steht das Schlüsselwort `type` zur Verfügung, das für einen beliebigen Typ steht. Die Distributivität des `?:`-Operators über Funktionsaufrufe ließe sich beispielsweise so ausdrücken:

```
#pragma rule ForAll int c; type f,e1,e2 :
             f(c?e1:e2) === c?f(e1):f(e2)
```

Streng genommen besteht hier natürlich schon eine Typbeschränkung. `e1` und `e2` müssen vom selben Typ `T` sein und `f` muss eine Funktion sein, die ein Argument vom Typ `T` als Eingabe nimmt. Dies wird allerdings ohnehin vom Compiler überprüft, auch bei abgeschalteter Optimierung. Entscheidend ist, dass, wenn das Pattern `f(c?e1:e2)` in einem typkorrekten C-Programm auftritt, die Regel ohne weitere Einschränkung der Typen von `f`, `e1` und `e2` immer angewandt werden kann.

Um einer Regel eine Anwendbarkeitsbedingung zu geben, steht das Symbol `=>` zur Verfügung, welches für „impliziert" steht.

```
#pragma rule ForAll int n: SideEffectFree(n) => n-n === 0
```

Zur Formulierung von Anwendbarkeitsbedingungen gibt es die Prädikate `Constant`, `SideEffectFree` und `C`.

- `SideEffectFree(expr)` ist erfüllt, wenn der Ausdruck `expr` keinen Seiteneffekt besitzt, d.h. wenn er keine Operatoren enthält, die direkt Seiteneffekte verursachen, und keine Aufrufe von Funktionen, die Seiteneffekte besitzen.
- `Constant(expr)` ist erfüllt, wenn `expr` eine Konstante im *syntaktischen* Sinn ist. `Constant(4+7)` liefert `false`, `Constant(11)` liefert `true`, obwohl beide Ausdrücke *semantisch* äquivalent sind.
- `C(expr)` ist erfüllt, wenn `expr`, ausgewertet mit der Semantik von `C`, zu `true` evaluiert. Ist eine Auswertung nicht möglich, weil `expr` freie Variablen enthält, so wird `false` zurückgeliefert.

Das Prädikat `C` wird benötigt, um die Grammatik eindeutig zu halten. Kommt im Programm beispielsweise eine Funktion mit dem Namen `Constant` vor, so kann nur mit Hilfe des Prädikats `C` zwischen den Bedingungen `Constant(expr)` und `C(Constant(expr))` unterschieden werden.

Bedingungen können außerdem mit dem Symbol '`!`' negiert und mit '`&&`' und-verknüpft werden.

```
ForAll int c,x : Constant(c) && !Constant(x) => x*c === c*x
```

Die vollständige Grammatik der Transformationsregeln befindet sich im Anhang C.

3.4.3 Hardware-interne Informationen

Mit dem Pragma `cost` kann der Benutzer die Kosten elementarer Operationen angeben. Die Syntax lautet `#pragma cost <operation> <value>`, wobei `<operation>` ein Schlüsselwort ist, das die Operation beschreibt, und `<value>` die Kosten dafür angibt.

```
#pragma cost iadd 1
#pragma cost isubtract 1
#pragma cost imultiply 13
#pragma cost iassign 2
#pragma cost passign 2
```

Eine vollständige Liste der Schlüsselwörter und der zugehörigen Operationen befindet sich im Anhang D.

Die mit dem Pragma `cost` gegebenen Operationskosten überschreiben die compiler-internen Voreinstellungen, die für Prozessoren wie den Pentium II oder K6 zufriedenstellende Ergebnisse liefern. Sollte der Benutzer also keine speziellen Operationskosten angeben, so gelten die vordefinierten Werte.

Das folgende Beispiel soll anhand der *Common Subexpression Elimination* verdeutlichen, wie sich die gegebenen Operationskosten auf die Optimierung auswirken können.

Die Anweisung

```
a[i*j+x*y] = a[y*x*2+j*i] + a[i*j+2*(y*x)];
```

wird, wenn keine speziellen Kostenangaben gemacht werden, von EPCOT folgendermaßen optimiert:

```
int cse0,cse1,cse2;
cse0 = i*j;
cse1 = x*y;
cse2 = a[cse0+(cse1<<1)];
a[cse0+cse1] = cse2 + cse2;
```

Die Kommutativität der Operatoren `+` und `*` für Integer-Zahlen wird bei der *CSE Elimination* berücksichtigt. Die Ausdrücke `a[y*x*2+j*i]` und `a[i*j+2*(y*x)]` sowie deren entsprechende Teilausdrücke wurden deshalb hier als identisch erkannt.

Angenommen, wir haben nun einen Prozessor, auf dem eine Integer-Multiplikation nur einen Taktzyklus benötigt. Wir fügen deswegen das folgende Pragma hinzu:

```
#pragma cost imultiply 1
```

Die beiden *CSE Eliminations* für `i*j` und `x*y` lohnen sich nun nicht mehr. Aus unserer Beispielanweisung wird

```
int cse2;
cse2 = a[i*j+(x*y)<<1];
a[i*j+x*y] = cse2+cse2;
```

Wir sind jedoch mit dieser Lösung nicht zufrieden, da eine Integer-Zuweisung auf unserem Prozessor 10 Taktzyklen braucht, und fügen deswegen hinzu:

```
#pragma cost iassign 10
```

EPCOT generiert nun die folgende Variante:

```
int* cse2;
cse2 = a + i*j + (x*y)<<1;
a[i*j+x*y] = *cse2 + *cse2;
```

Was ist hier passiert? Nun, der Ausdruck `a[i*j + (x*y)<<1]` ist identisch zu `*(a + i*j + (x*y)<<1)`. Da eine *Integer-Zuweisung* laut unserem Pragma teuer ist, eine *Pointer-Zuweisung* jedoch nicht, wählte der Compiler jetzt die Adresse `a+i*j+(x*y)<<1` als zu eliminierende *CSE*. Wenn wir nun aber noch das Pragma

```
#pragma cost passign 10
```

hinzufügen, so wird gar keine *CSE Elimination* mehr durchgeführt und es bleibt bei der ursprünglichen Anweisung.

3.5 Diskussion

In diesem Kapitel haben wir mit den Pragma-Direktiven eine einfache Möglichkeit vorgestellt, ein C/C++-Programm mit semantischen Zusatzinformationen anzureichern. Weiterhin wurden drei Arten von Informationen vorgestellt, die dem Compiler bei der Optimierung helfen würden: Eigenschaften von Funktionen, Ersetzungsregeln für Ausdrücke und Zeitkosten für elementare Operationen.

3.5.1 Verwandte Arbeiten

1994-97 wurde an der Technischen Universität Berlin das Projekt OPAL verfolgt. OPAL ist eine algebraische, rein funktionale Programmiersprache, die für Experimente mit der Spezifikation und Entwicklung funktionaler Programme entworfen wurde [DFG$^+$94]. In OPAL gibt es ebenfalls die Möglichkeit, das Programm mit Zusatzinformationen (dort LAWs genannt) anzureichern. Die LAWs von OPAL stellen ebenfalls Transformationsregeln dar und entsprechen unserem Pragma `rule`. Informationen über Funktionseigenschaften entfallen, da in OPAL natürlich alle Funktionen referentiell transparent sind. Die LAWs werden jedoch noch nicht vom OPAL-Compiler genutzt, sondern stellen vorerst lediglich „Kommentare“ dar. In einer zukünftigen Compiler-Version ist die Nutzung der LAWs jedoch geplant.

3.5.2 Erweiterungsmöglichkeiten

Selbstverständlich sind noch weitere Arten von Zusatzinformationen denkbar, von denen nun einige hier vorgestellt werden sollen.

3.5.2.1 Transformationsregeln für Anweisungen

Bisher sind beispielsweise mit den Transformationsregeln nur Ersetzungsregeln für *Ausdrücke* formulierbar. Denkbar wären aber auch Ersetzungsregeln für *Anweisungen*.

Mit folgender Regel könnte z.B. die Ersetzung von `for`- durch `while`-Schleifen spezifiziert werden.

```
#pragma rule ForAll init,cond,step,body :
                    for (init; cond; step) body
                    ===
                    init; while (cond) { body; step; }
```

Dies stellt an sich keine echte Optimierung dar, da die rechte Seite der Regel keine effizientere Version ist, sondern lediglich die Semantik von `for`-Schleifen mit Hilfe von `while`-Schleifen definiert. Als erste Vor-Optimierung eines Programms wäre diese Ersetzung aber durchaus vorstellbar. Algorithmen zur Schleifenoptimierung müssten dann nur für `while`-Schleifen geschrieben werden.

Ebenso könnte man mit Transformationsregeln für Anweisungen formulieren, dass bestimmte Anweisungen invers zueinander sind oder kombiniert werden können.

```
#pragma rule ForAll int i : i++;  i--;  === ;
#pragma rule ForAll int i,a,b : i+=a; i+=b; === i+=(a+b);
```

Optimierungen dieser Art werden *Instruction Combining* genannt und werden von einigen Compilern durchgeführt. Mit den entsprechenden Regeln wäre *Instruction Combining* jedoch auch auf benutzerdefinierte Funktionen erweiterbar.

```
#pragma rule push(); pop(); === ;

#pragma rule left(); left(); === uturn();
#pragma rule uturn(); forward(); uturn(); === backward();
```

3.5.2.2 Modifikation der Optimierungsstrategie

Einige Compiler gestatten dem Benutzer, mittels entsprechender Optionen beim Aufruf des Compilers die durchzuführenden Optimierungen zu wählen. Mit Pragma-Direktiven könnte man dies weiter verfeinern.

```
#pragma strategy { unroll; constant_propagation; cse; }
```

Zusätzlich zur Wahl, welche Optimierungen durchgeführt werden sollen, könnte man mit den Pragmas spezifizieren, in welcher Reihenfolge dies geschehen soll. (In der Regel beeinflusst dies das Ergebnis.) Auch könnten bestimmte Verfahren auf Wunsch mehrmals angewandt werden. Und schließlich könnte damit die Optimierungsstrategie an einer bestimmten Programmstelle umgeschaltet werden.

Für die elementaren Verfahren müssten entsprechende Schlüsselwörter zur Verfügung stehen, mit denen man dann die Gesamtstrategie zusammenbauen würde.

Es ist allerdings fraglich, ob beim Benutzer überhaupt Interesse besteht, von dieser Möglichkeit Gebrauch zu machen. Eventuell ist dies eher für den Compiler-Entwickler interessant, um schnell mit verschiedenen Strategien experimentieren zu können.

Kapitel 4 Erweiterte Compilierung

Dieses Kapitel beschreibt, wie die semantischen Zusatzinformationen vom Compiler für Codeoptimierungen genutzt werden können. Die Übersetzung unter Verwendung solcher Informationen wird *erweiterte Compilierung* genannt [DFG+94, Fea87].

Abschnitt 4.1 erläutert zunächst die grundlegenden Konzepte, mit denen die Zusatzinformationen in den Compilationsvorgang integriert werden können. In den folgenden Abschnitten werden dann verschiedene Anwendungsmöglichkeiten der erweiterten Compilierung vorgestellt.

Abschnitt 4.2 behandelt zunächst eine einfache Anwendung: die algebraische Vereinfachung von Ausdrücken — eine Optimierung, die sich mit Hilfe der Transformationsregeln, und ohne den Quellcode zu ändern, in den Compiler einbauen ließ. Abschnitt 4.3 stellt die um einiges aufwendigere Optimierung *Symbolic Finite Differencing* vor. In Abschnitt 4.4 wird gezeigt, wie für die Funktionen der C-Standardbibliothek spezielle Optimierungen definiert werden können, bei denen die Kenntnis über die Semantik der Funktionen verwendet wird. Abschnitt 4.5 erweitert dieses Konzept auf benutzerdefinierte abstrakte Datentypen. Schließlich wird in Abschnitt 4.6 diskutiert, welche Vorteile sich durch die Möglichkeit, den Optimierer über die Pragma-Direktiven zu konfigurieren, für den Compiler-Entwickler ergeben.

4.1 Grundprinzip

Wir haben drei Arten von Informationen (property, rule, cost), die wir nun in den Compilationsvorgang integrieren wollen. Dabei stellen sich jeweils zwei Fragen.

- Was ist zu tun, wenn die Information registriert wird? (Dies geschieht während der Syntaxanalyse des Programms, also während der Syntaxbaum konstruiert wird.)
- Wo und wie kann die Information später beim Optimieren des Syntaxbaums verwendet werden?

4.1.1 Funktionseigenschaften

Die Eigenschaften der Funktionen speichert man am besten in den Funktionsknoten im Syntaxbaum des Programms ab. Verwendet werden können die Eigenschaften von nahezu allen klassischen Optimierungen — nämlich immer dann, wenn analysiert werden muss, wie sich ein Funktionsaufruf auf die aktuellen Informationen auswirken kann.

Entsprechende Beispiele für die *Constant Propagation* und *CSE Elimination* wurden in Abschnitt 3.2.1 vorgestellt.

4.1.2 Operationskosten

Die Operationskosten legt man in einem globalen Feld ab, das zu Beginn der Compilierung Default-Werte für die Operationskosten für typische Prozessoren enthält. Mit dem cost-Pragma können diese Werte dann re-definiert werden. Zur geeigneten Verwendung der Informationen benötigt man eine Funktion `cost`, welche die Kosten eines Ausdrucks unter Zuhilfenahme des Feldes berechnet.

```
int cost (an_expr_node_ptr expr);
```

Hilfreich ist diese Funktion immer dann, wenn zwischen Alternativen abzuwägen ist, wie beispielsweise in der *CSE Elimination*: Lohnt es sich, einen gegebenen Ausdruck einer Variablen zuzuweisen, um ihn einmal weniger berechnen zu müssen?

4.1.3 Transformationsregeln

Die Transformationsregeln (rule) legt man in einer compiler-internen Datenbank ab. Um die Regeln zur Anwendung zu bringen, benötigt man noch ein Termersetzungssystem, d.h. eine Funktion `rewrite`, die einen Ausdruck als Eingabe nimmt und die gegebenen Regeln so lange anwendet, bis ein Ausdruck erreicht ist, auf den keine linke Seite einer Regel mehr passt.

```
void rewrite (an_expr_node_ptr expr);
```

Generell sind zwei Verwendungsmöglichkeiten von `rewrite` denkbar, welche beide in EPCOT vorkommen.

- Man kann den Compiler um einen allgemeinen Optimierungsdurchlauf erweitern, in dem die Prozedur `rewrite` auf jeden im Programm vorkommenden Ausdruck angewendet wird (Abschnitt 4.2).
- Man kann `rewrite` als Hilfsfunktion in komplexeren Optimierungen verwenden (Abschnitt 4.3).

Zusätzlich zur Anwendung aller gegebenen Regeln kann man noch das Falten von konstanten Ausdrücken (*Constant Folding*) in die Funktion `rewrite` integrieren. Diese Variante wurde in EPCOT gewählt. Bei der gegebenen Regel

```
#pragma rule ForAll int c; int e1,e2,e3 : e3+(c?e1:e2) === c?(e3+e1):(e3+e2)
```

würde ein Aufruf von `rewrite` den Ausdruck `1+(c?3:4)` also gleich zu `c?4:5` optimieren, anstatt nur zu `c?(1+3):(1+4)`.

4.2 Algebraische Vereinfachung von Ausdrücken

Die Optimierung *Constant Folding* ist bereits fest in EPCOT eingebaut. Ausdrücke der Form `(x*2)*3` oder `(x+2)+3` werden davon aber noch nicht erfasst. Um hier zu `x*6` bzw. `x+6` vereinfachen zu können, müsste der Compiler die Assoziativität der Multiplikation und Addition ausnutzen.

Ohne den Quellcode des Compilers zu ändern, kann dies dem Compiler nun durch Angabe der benötigten Transformationsregeln beigebracht werden. Das simple Einfügen folgender vier Zeilen in den Quellcode eines Programms bewirkt, dass der Compiler alle darin vorkommenden Ausdrücke der obigen Form nun optimiert.

```
#pragma rule ForAll int a,b,c : a*(b*c) === (a*b)*c
#pragma rule ForAll int a,b,c : a+(b+c) === (a+b)+c

#pragma rule ForAll int a,b : !Constant(a) && Constant(b) => a*b === b*a
#pragma rule ForAll int a,b : !Constant(a) && Constant(b) => a+b === b+a
```

Die ersten beiden Regeln weisen den Compiler an, bei den Operatoren `+` und `*` die Assoziativität auszunutzen, um Klammern nach links zu schieben. (Man beachte hier, dass die Transformationsregeln immer nur von links nach rechts angewendet werden.)

Die letzteren beiden Regeln bewirken, dass durch Vertauschen der Operanden Konstanten auf die linke Seite gebracht werden. Die Bedingung garantiert hier die Terminierung. Würden wir lediglich `ForAll int a,b :  a*b === b*a` schreiben, könnte die Regel wiederholt immer wieder auf den selben Ausdruck angewandt werden. Es würde allerdings auch ausreichen, die Bedingung `Constant(b)` zu verwenden, da *Constant Folding* in EPCOT Priorität über alle Transformationsregeln hat, und `!Constant(a)` daher implizit gilt, wenn eine der Regeln zum Einsatz kommt.

4.2.1 Konfluenz

Mit den gegebenen Regeln kann der Ausdruck `(x*2)*3` auf mehrere Arten zu `6*x` transformiert werden, wie in Abbildung 4.1 gezeigt wird.

Welcher Weg vom Compiler hier letztendlich genommen wird, hängt vom Algorithmus seines internen Termersetzungssystems ab. In diesem Beispiel enden alle Wege bei `6*x`. Dies ist kein Zufall, denn die verwendeten vier Regeln bilden ein *konfluentes* System (Abschnitt 1.2.3.1), wie sich mit der Methode der kritischen Paare [DJ90, vH97] zeigen ließe.

Es gibt jedoch leider auch Regelsysteme, die diese wünschenswerte Eigenschaft nicht haben. Fügt man die Regel

```
#pragma rule ForAll int n : 2 * n === n << 1
```

hinzu, so könnte die Optimierung sowohl bei `6*x` als auch bei `3*(x<<1)` enden.

Durch Hinzufügen einer (zugegebenermaßen unschönen) sechsten Regel kann die Konfluenz hier wieder hergestellt werden:

```
#pragma rule ForAll int a,b : a * (b << 1) === (2 * a) * b
```

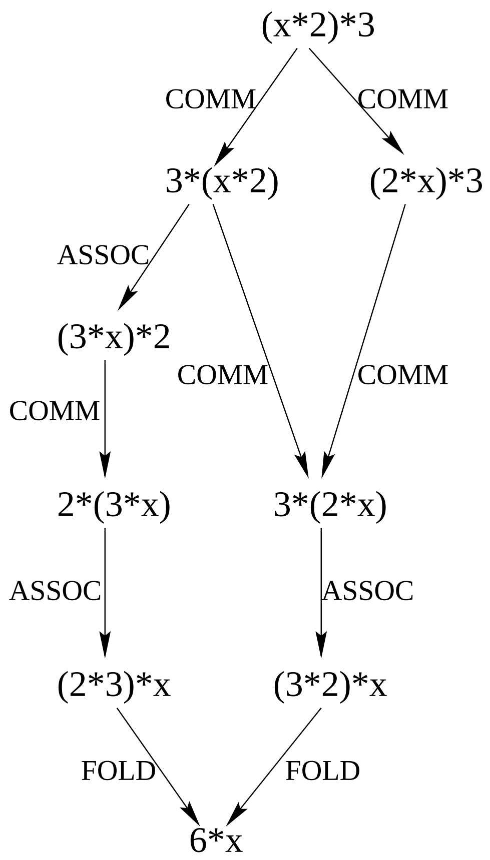


Abbildung 4.1: *Mögliche Wege der Optimierung des Ausdrucks (x*2)*3*

Vorerst ist der Benutzer selbst dafür verantwortlich, auf die Konfluenz seines Systems zu achten, sofern er diese wünscht. Denkbar wäre hierfür jedoch auch maschinelle Unterstützung. Konfluenz ist entscheidbar, und die Vervollständigung des Regelsystems könnte mit der sogenannten Knuth-Bendix-Vervollständigung [DJ90, vH97] automatisch durchgeführt werden. Allerdings kann die Anzahl der Regeln durch eine solche Vervollständigung exponentiell zunehmen, was sich verheerend auf die Compilierzeit auswirken würde.

Eine Alternative wäre, dem Benutzer Einfluss auf den Algorithmus des Termersetzungssystems zu geben. Man könnte z.B. die Regeln in Klassen einteilen und den Compiler anweisen, zuerst alle Regeln der Klasse 1 anzuwenden. Wenn damit keine weiteren Optimierungen mehr möglich sind, werden die Regeln der Klasse 2 angewandt und so weiter.

4.2.2 Terminierung

Der Benutzer ist ebenso dafür verantwortlich, dass sein Termersetzungssystem terminiert. Dies kann dadurch sichergestellt werden, dass Ausdrücke nur durch solche Ausdrücke ersetzt werden, die bezüglich einer gewählten Terminationsordnung kleiner sind.

4.2.3 Korrektheit

Ebenso muss beachtet werden, dass die Transformationen die Semantik des Programms nicht verändern. Es liegt in der Hand des Benutzers, dem Compiler zu sagen, welche Transformationen er durchführen darf.

Dem Compiler z.B. das assoziative Umklammern auch für den Datentyp `float` zu erlauben, könnte auf Grund von Rundungsfehlern zu verändertem Verhalten der compilierten Programme führen.

Streiten ließe sich auch über die Regel

```
#pragma rule ForAll int n : SideEffectFree(n) => n/n === 1
```

Wenn `n=0` wäre, so würde nämlich das unoptimierte Programm bei dem Ausdruck `n/n` mit einem Division-durch-Null-Fehler abstürzen, das optimierte Programm hingegen nicht. Falls dies im Interesse des Benutzers liegt, so kann er die obige Regel so stehen lassen. Will er jedoch, dass das optimierte Programm sich genau gleich verhält, so sollte er eine der folgenden Formulierungen wählen.

```
#pragma rule ForAll int n : SideEffectFree(n) && C(n!=0) => n/n === 1
#pragma rule ForAll int n : SideEffectFree(n) => n/n === n?1:(0/0)
```

Die erste Regel veranlasst den Compiler, die Ersetzung von `n/n` durch `1` nur dann durchzuführen, wenn er sich sicher ist, dass `n` einen Wert ungleich `0` hat. Das kann natürlich dazu führen, dass die Regel in vielen Fällen gar nicht zur Anwendung kommt.

Bei der zweiten Regel wird der Test in den optimierten Code eingebaut. `n/n` wird durch `n?1:(0/0)` ersetzt. Ein Test auf Verschiedenheit von `0` ist auf den meisten Prozessoren schneller als eine Division. Und sollte sich bei diesem Test `n=0` herausstellen, so wird der Division-durch-Null-Fehler „künstlich“ ausgelöst, um das erwartete Verhalten zu erzeugen.

4.2.4 Ergänzung des Regelsatzes

Zum Abschluss soll der Regelsatz ergänzt werden, um auch kompliziertere Vereinfachungen durchführen zu können, wie z.B. von `5-3*x*6+2*x*9+5` zu `10`.

Eine Möglichkeit, den Minus-Operator zu behandeln, ist, die entsprechenden eingeschränkten Assoziativ- und Kommutativgesetze dafür anzugeben.

```
#pragma rule ForAll int c1,c2,e : Constant(c1) && Constant(c2)
                => (c1-e)+c2 === (c1+c2)-e
```

Alternativ dazu könnte man auch versuchen, binäre Minus-Operatoren zu eliminieren, so dass die Gesetze für den Plus-Operator greifen.

```
#pragma rule ForAll int a,b : a-b === a+(-b)
```

Dann benötigen wir die spezielle Eigenschaft der `0` in der Multiplikation.

```
#pragma rule ForAll int n : SideEffectFree(n) => 0 * n === 0
```

Die Bedingung über Seiteneffektfreiheit ist hier notwendig. Ansonsten würde z.B. `0 * i++` zu `0` vereinfacht. Eine alternative Regel wäre hier allerdings

```
#pragma rule ForAll int n : 0 * n === n,0
```

`n` wird hier evaluiert, um einen eventuellen Seiteneffekt auszuführen. Das Ergebnis wird jedoch ignoriert; stattdessen wird `0` zurückgeliefert. Diese Regel ist vorzuziehen, da sie auch dann zum Einsatz kommt, wenn `n` einen Seiteneffekt besitzt. Der störende Kommaausdruck, der dabei in jedem Fall entsteht, lässt sich mit Hilfe einer zweiten Regel wieder beseitigen:

```
#pragma rule ForAll type a,b : SideEffectFree(a) => a,b === b
```

`0*x` würde damit also zuerst zu `x,0` und dann zu `0` vereinfacht werden.

Schließlich benötigen wir noch die Distributivität der Addition (und eventuell auch die der Subtraktion) über die Multiplikation.

```
#pragma rule ForAll int a,b,x : SideEffectFree(x) &&
                                Constant(a) && Constant(b)
                => a*x + b*x === (a+b)*x

#pragma rule ForAll int a,b,x : SideEffectFree(x) &&
                                Constant(a) && Constant(b)
                => a*x - b*x === (a-b)*x
```

Mit Hilfe dieser Regeln würde der obige Ausdruck `5-3*x*6+2*x*9+5` nun zu `10` vereinfacht.

Um dem Compiler diese Optimierung nun generell beizubringen, kann man die Regeln in eine Datei schreiben, die von allen zu compilierenden Programmen eingebunden wird (beispielsweise in `<stdio.h>`). Im Anhang B werden weitere allgemeine Regeln zur Vereinfachung von Ausdrücken vorgestellt.

4.3 Symbolic Finite Differencing

Finite Differencing kann weitaus mehr leisten als die Optimierung der Berechnung von Polynomen, die in Kapitel 2.5 vorgestellt wurde. 1977 zeigten R. Paige und J. T. Schwartz beispielsweise, wie - in der Sprache SETL - *Finite Differencing* in einem mengentheoretischen Kontext angewendet werden kann [PS77].

1990 schuf Robert Paige den theoretischen Rahmen zu einem allgemeinen Verfahren, für welches er den Begriff *Symbolic Finite Differencing* wählte [Pai90] und mit dem ganz allgemein wiederholte aufwendige Berechnungen durch inkrementelle Berechnungen ersetzt werden können.

Das Verfahren konnte jedoch nicht vollständig automatisiert werden und konnte daher bisher noch nicht in optimierenden Compilern eingesetzt werden. Wie wir hier zeigen werden, beschränkt sich der nicht automatisierbare Teil des Verfahrens jedoch hauptsächlich auf das Auffinden von algebraischen Gesetzen, die dem Compiler über die semantischen Zusatzinformationen vermittelt werden können. Mit deren Hilfe wird das *Symbolic Finite Differencing* somit für optimierende Compiler interessant.

4.3.1 Motivation

Wir betrachten das folgende Programm, welches die Zweierpotenzen von 2^0 bis 2^{30} berechnet und in ein Feld schreibt.

```
int power (int a, int b)
{
  if (b==0) return 1;
  return a*power(a,b-1);
}

void compute_powers()
{
  int i;
  for (i=0; i<=30; i++)
    a[i] = power(2,i);
}
```

Für jede Zahl `i` wird hier die Funktion `power(2,i)` aufgerufen. Die Laufzeit der Funktion `power` steigt linear mit dem Argument `i`, da, um 2^i zu berechnen, `i` Multiplikationen ausgeführt werden. Insgesamt werden im obigen Programm 1+2+...+29+30 = 465 Multiplikationen ausgeführt.

Ersetzt man den Ausdruck `power(2,i)` durch eine Variable, die mit 1 initialisiert und in jedem Schleifendurchlauf verdoppelt wird, so erhält man das folgende optimierte Programm, das mit 31 Multiplikationen auskommt.

```
void compute_powers()
{
  int i,x;
  for (i=0, x=1; i<=30; i++, x*=2)
    a[i] = x;
}
```

Doch wie soll der Compiler auf diese Lösung kommen? Benötigt wird hier semantisches Wissen über die Funktion `power`, das entweder gar nicht oder nur mit unvertretbar hohem Aufwand aus dem Quellcode abgeleitet werden kann. In diesem speziellen Beispiel könnte man sich zwar theoretisch noch vorstellen, dass der Compiler durch Programmanalyse die Semantik der Funktion `power` erkennt - im allgemeinen Fall ist dies aber unmöglich. Funktionen, die lediglich im Header-File deklariert sind, und deren Objektcode erst später hinzugelinkt wird, entziehen sich außerdem einer möglichen Analyse durch den Compiler.

Durch Angabe der entsprechenden Zusatzinformationen können wir aber den Compiler mit dem für die obige Optimierung benötigten Wissen ausstatten. Als erstes deklarieren wir dazu `power` als transparente Funktion.

```
#pragma property Transparent(power)
```

Weiterhin geben wir die wichtigsten algebraischen Eigenschaften der Funktion an, d.h. die Potenzgesetze $a^0 = 1$, $a^1 = a$ und $a^{b+c} = a^b * a^c$.

```
#pragma rule ForAll int a     : SideEffectFree(a) => power(a,0) === 1
#pragma rule ForAll int a     : power(a,1) === a
#pragma rule ForAll int a,b,c : SideEffectFree(a) =>
                                power(a,b+c) === power(a,b) * power(a,c)
```

Die Anwendbarkeitsbedingungen der ersten und dritten Regel sind notwendig, damit der Compiler keine Seiteneffekte wegoptimiert, beispielsweise in `power(i++,0)` oder `power(i++,b+c)`. In der zweiten Regel ist jedoch keine solche Bedingung notwendig. Weiterhin genügt es, in der dritten Regel die Seiteneffektfreiheit von `a` zu fordern. `b` und `c` dürfen dagegen Seiteneffekte haben.

Ausgestattet mit diesen Informationen kann EPCOT nun die obige optimierte Fassung von `compute_powers` herleiten.

Man beachte jedoch, dass `power(a,b)` hier nicht für negative `b` definiert ist. Die dritte Transformationsregel müsste daher eigentlich noch zwei weitere Anwendbarkeitsbedingungen besitzen, um wirklich korrekt zu sein:

```
#pragma rule ForAll int a,b,c : SideEffectFree(a) && C(b>=0) && C(c>=0) =>
                                power(a,b+c) === power(a,b) * power(a,c)
```

Ansonsten könnte der Compiler z.B. `power(2,3+(-1))` durch `power(2,3) * power(2,-1)` ersetzen. In der aktuellen Implementierung könnte EPCOT allerdings die Gültigkeit dieser beiden weiteren Bedingungen nicht beweisen und würde daher keine Optimierung durchführen. Um in dem gegebenen Programm zu zeigen, dass im Ausdruck `power(2,i)` die Bedingung `i>=0` erfüllt ist, wäre eine Intervallanalyse [NNH99] notwendig, die bestimmt, in welchem Bereich die Werte, die eine Variable annehmen kann, liegen können. Im obigen Programm wäre dies für `i` das Intervall [0,30].

4.3.2 Grundprinzip

Im Rumpf einer Schleife sei $E(x_1, ..., x_n)$ ein transparenter Ausdruck, dessen Berechnung aufwendig ist. $x_1, ..., x_n$ seien Variablen, die an einigen Stellen innerhalb der Schleife modifiziert werden. Wir nennen im folgenden $E(x_1, ..., x_n)$ einen *Flaschenhals*, die Variablen $x_1, ..., x_n$ *Induktionsvariablen* des Flaschenhalses und die Stellen, an denen die Induktionsvariablen verändert werden, *Modifikationsstellen*.

```
void compute_powers()
{
  int i;
  for (i=0; i<=30; i++)
    a[i] = power(2,i);
}
```

Flaschenhals:	`power(2,i)`
Induktionsvariable:	`i`
Modifikationsstelle:	`i++`

Das Ziel ist nun, den Flaschenhals $E(x_1, ..., x_n)$ durch eine Variable *out* zu ersetzen, die vor dem Eintritt in die Schleife entsprechend initialisiert wird, und durch Einfügen von entsprechenden Anweisungen an den Modifikationsstellen dafür zu sorgen, dass die Invariante $out = E(x_1, ..., x_n)$ erhalten bleibt, wann immer eine der Induktionsvariablen $x_1, ..., x_n$ verändert wird. Die Variable *out* nennt man auch *Ausgabevariable* der Invariante $out = E(x_1, ..., x_n)$.

Die einzufügenden Anweisungen werden *Differenz-Code* genannt. Wird eine Modifikationsstelle `dx` durch `D1; dx; D2;` ersetzt, so nennt man `D1` und `D2` Prä- und Post-Differenz-Code.

Die Optimierung lohnt sich genau dann, wenn die Berechnung des Flaschenhalses aufwendiger ist als die Berechnung des Differenz-Codes.

4.3.3 Induktionsvariablen und Modifikationsstellen

Anstatt für jeden Flaschenhals einzeln die Induktionsvariablen und deren Modifikationsstellen zu bestimmen, werden in EPCOT alle Variablen als potentielle Induktionsvariablen betrachtet. Vor der Optimierung einer Schleife wird für jede in der Schleife vorkommende Variable eine Liste ihrer Modifikationsstellen erstellt.

Bevor die Optimierung *Finite Differencing* angewandt wird, werden zwei Bedingungen überprüft:

- Alle Funktionsaufrufe müssen *zustandseffektfrei* sein. (Kap. 3.2.1.3) Ansonsten könnten vermeintliche Induktionsvariablen in den aufgerufenen Routinen verändert werden.
- Bei allen Zuweisungen muss die Zielvariable bestimmbar sein. Eine Zuweisung wie `*p=1`, bei der der Wert von `p` unbekannt ist, könnte ansonsten eine Induktionsvariable verändern. Oder anders gesagt: `*p=1` ist möglicherweise eine Modifikationsstelle, aber wir wissen nicht, von welcher Variable.

Ist eine dieser Voraussetzungen nicht erfüllt, so ist die Menge der Induktionsvariablen nicht bestimmbar und eine Optimierung mittels *Finite Differencing* nicht möglich. Ansonsten stellen in der Tat alle Variablen potentielle Induktionsvariablen dar und wir haben deren Modifikationsstellen alle erfasst.

Hat eine Variable keine Modifikationsstellen, liegt der Spezialfall einer schleifeninvarianten Variable vor.

Beispiel

```
for (j=0; j<100; j++)
{
   if (j%2==0)
      k+=2;
   else
      k--;
   printf("%d\n",i*j*k);
}
```

Induktionsvariable	Modifikationsstellen
i	keine (invariante Variable)
j	`j++`
k	`k+=2` und `k--`

4.3.4 Eliminierung eines Flaschenhalses

Der Algorithmus zur Eliminierung eines Flaschenhalses besteht, grob unterteilt, aus fünf Schritten.

1. Erzeuge eine neue Variable *out*.
2. Ersetze den Flaschenhals $E(x_1, ...x_n)$ durch *out*.
3. Füge direkt vor dem Eintritt in die Schleife die Initialisierung $out = E(x_1, ...x_n)$ ein.
4. Für jede Modifikationsstelle m_{ij} jeder Induktionsvariable x_i des Flaschenhalses: Versuche, den Differenz-Code zur Erhaltung der Invariante $out = E(x_1, ...x_n)$ zu generieren.
5. Ist der Differenz-Code für ein m_{ij} nicht berechenbar oder ist der akkumulierte Aufwand für den berechneten Differenz-Code zu hoch, so mache die bereits durchgeführten Programmänderungen rückgängig und breche ab. Ansonsten füge den Differenz-Code ins Programm ein.

Es bleibt das Problem, die Flaschenhälse zu erkennen. In der aktuellen Implementierung werden hier alle transparenten Ausdrücke im Schleifenrumpf als potentielle Flaschenhälse angesehen. Wenn sich die Berechnung des Differenzcodes als aufwendiger herausstellt als die eines vermeintlichen Flaschenhalses, so wird die Optimierung nicht durchgeführt.

4.3.5 Generierung des Differenz-Codes

Der Kern des symbolischen Finite Differencing - und gleichzeitig der Teil, der nicht ohne weiteres mechanisierbar ist - liegt in der Berechnung des Differenz-Codes.

Zur Erhaltung einer Invariante *inv* ist für jede Modifikationsstelle, die eine der Induktionsvariablen von *inv* verändert, der entsprechende Differenzcode zu generieren. Zuständig ist hierfür die Funktion `generate_difference_code`. Sie bekommt drei Argumente: die

Ausgabevariable out, in der die Invariante gespeichert ist, den symbolischen Wert von inv (also den Ausdruck $E(x_1, ..., x_n)$), und eine Anweisung dx, die eine der Variablen $x_1, ..., x_n$ verändert. Zurückgegeben wird der berechnete Prä-Differenz-Code, also eine Anweisung δout, die direkt *vor* dx ausgeführt werden muss, so dass nach der Ausführung von $\delta out; dx$ die Invariante $out = E(x_1, ..., x_n)$ wieder gilt.

Der Aufruf `generate_difference_code (x, power(2,i), i++)` liefert beispielsweise das Ergebnis `x*=2`. Das bedeutet, dass, um die Invariante `x = power(2,i)` zu erhalten, vor der Anweisung `i++;` die Anweisung `x*=2;` eingefügt werden muss.

`generate_difference_code` liefert den NULL-Zeiger zurück, wenn keine Anweisung gefunden werden kann, welche die Invariante erhalten würde.

Der Basisalgorithmus zur Generierung des Differenz-Codes arbeitet folgendermaßen:

1. Eine Zuweisung, die einen speziellen Zuweisungsoperator von C verwendet (`++`, `--`, `+=`,...) wird in eine gewöhnliche Zuweisung umgeschrieben. Aus `i++` wird also `i=i+1`.

2. In der zu erhaltenden Invariante E_0 wird die Induktionsvariable durch den Ausdruck, der ihr zugewiesen wird, ersetzt. Wir erhalten E_1. In `power(2,i)` wird `i` durch `i+1` ersetzt. Wir erhalten `power(2,i+1)`.

3. Das Termersetzungssystem wird mit E_1 gestartet und wir erhalten E_2. Aus `power(2,i+1)` wird hier z.B. `power(2,i)*2`.

4. Die ursprüngliche Invariante E_0 wird in E_2 gesucht und durch die Ausgabevariable out ersetzt. Wir erhalten E_3. Aus `power(2,i)*2` wird `x*2`.

5. Dieser Ausdruck muss out zugewiesen werden, um die Invariante zu erhalten. Der berechnete Prä-Differenz-Code ergibt sich somit zu $out = E_3$, in unserem Beispiel also zu `x=x*2`.

Die eigentliche Intelligenz des Verfahrens liegt hier in den Schritten 3 und 4. Die Verwendung von Zusatzinformationen beschränkt sich dabei auf Schritt 3. Hier wird das Termersetzungssystem verwendet - in der Hoffnung, den Ausdruck E_1 in eine Form zu bringen, bei der dann (in Schritt 4) leicht erkannt werden kann, mit welchen Operationen er aus E_0 herzustellen ist.

In Schritt 4 versucht der Basisalgorithmus lediglich, E_0 als Teilausdruck in E_2 wiederzufinden. Dies führt zwar schon recht häufig zum Erfolg, lässt sich jedoch weiter verfeinern:

- Werden in einer Schleife mehrere Invarianten erhalten, so kann man in E_2 auch nach den anderen Invarianten suchen (anstatt nur nach E_0). Ein Beispiel, wo diese Technik erfolgreich angewendet werden konnte, befindet sich im Abschnitt 4.3.6.

- E_0 muss nicht unbedingt ein echter Teilausdruck von E_2 im syntaktischen Sinne sein. Sei $E_0 = 3 * x + 7$ und $E_2 = 3 * x + 11$. Eine mögliche Lösung für E_3 ist hier $out + 4$.

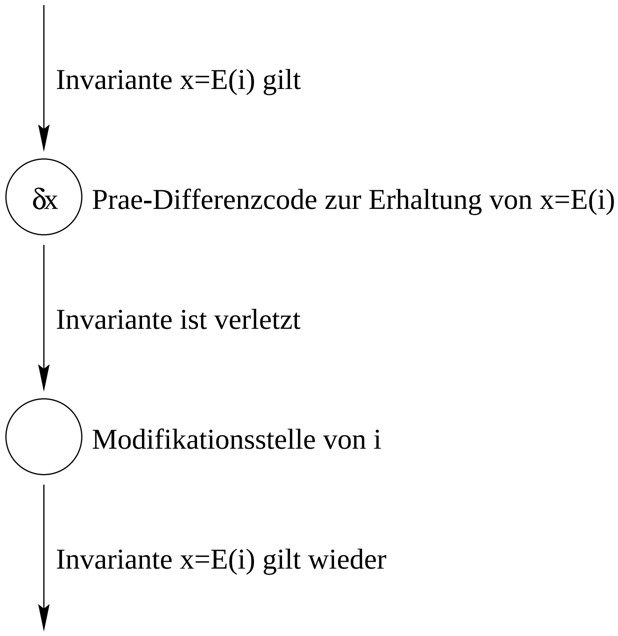


Abbildung 4.2: *Kurzfristige Invariantenverletzung an einer Modifikationsstelle*

4.3.6 Mehrere Invarianten

Werden in einer Schleife mehrere Invarianten gleichzeitig erhalten, so kann bei der Berechnung des Differenzcodes zu einer Invariante auch auf die anderen Invarianten zurückgegriffen werden. Ist z.B. `x=f(i)` und `y=f(i-5)` zu erhalten, so kann bei der Modifikation `i+=5` die letztere Invariante durch `y=x` erhalten werden.

Sind vor einer Modifikationsstelle jedoch mehrere Differenzcodestücke einzufügen, so kann es zwischen diesen Stücken zyklische Abhängigkeiten geben, die eine simple Hintereinanderausführung unmöglich machen. Die Invarianten werden an den Modifikationsstellen nämlich jeweils kurzfristig verletzt, und zwar *zwischen* dem Differenzcode und der eigentlichen Modifikation der Induktionsvariable (Abb. 4.2).

Beispiel: In dem Programmstück

```
for (i=0, sq=0; i<n; sq+=2*i+1, i++)
  square[i] = sq;
```

ist die Invariante $sq = i^2$ zwischen den Anweisungen `sq+=2*i+1` und `i++` verletzt. An dieser Stelle gilt stattdessen $sq = (i+1)^2$.

Normalerweise stört dies nicht, doch wenn zwischen dem Differenzcodestück δx und der Modifikationsstelle ein weiteres Differenzcodestück δy (zur Erhaltung einer zweiten Invariante) eingefügt werden soll, so darf in δy nicht auf x, die Ausgabevariable der verletzten Invariante, zugegriffen werden.

Dieses Problem lässt sich jedoch mit einer temporären Variablen lösen. Bevor x in δx verändert wird, kopiert man den Wert in eine neue Variable *tmp*. In δy greift man dann statt auf x auf *tmp* zu.

Das folgende Beispiel, in dem ein exponentieller Algorithmus zu einem Algorithmus mit linearer Zeitkomplexität optimiert wird, verdeutlicht diese Technik.

4.3.6.1 Beispiel: „Tribonacci-Zahlen“

Wir betrachten ein Programm, dass eine Variante der Fibonacci-Folge berechnet, in der jede Zahl die Summe ihrer *drei* Vorgänger ist. Für $n = 0, 1, 2$ definieren wir direkt die Funktionswerte 0, 0 und 1, für alle anderen Werte von n verwenden wir eine rekursive Definition, wobei für negative n rückwärts gerechnet wird.

```
int fib3 (int n)
{
  if (n==0 || n==1)
    return 0;
  else if (n==2)
    return 1;
  else if (n>=3)
    return fib3(n-3) + fib3(n-2) + fib3(n-1);
  else /* if (n<0) */
    return fib3(n+3) - fib3(n+2) - fib3(n+1);
}
```

Im Hauptprogramm geben wir die Folge aus für die Werte 1 bis n.

```
int n=38;

int main()
{
  int i;
  for (i=1; i<=n; i++)
    printf("%2d %10d\n",i,fib3(i));
  return 0;
}
```

Für `n=38` passt das Ergebnis gerade noch in eine 32-Bit-Integer-Zahl. Das ohne Optimierung compilierte Programm benötigt dafür auf einem PC mit 200 MHz jedoch etwa 45 Minuten, da die Rechenzeit für `fib3(n)` exponentiell mit `n` steigt. (`fib3(n)` hat in etwa die Zeitkomplexität $\Theta(1.84^n)$.)

Wir fügen nun zwei Pragmas hinzu, die dem Compiler Informationen über die Funktion `fib3` geben.

```
#pragma property Transparent(fib)
#pragma rule ForAll int n : fib3(n+1) === fib3(n) + fib3(n-1) + fib3(n-2)
```

Man beachte, dass die Transformationsregel falsch wäre, wenn die Funktion `fib3` nicht auch für negative Argumente definiert wäre. In diesem Fall müsste die Anwendbarkeitsbedingung `C(n>=2)` hinzugefügt werden.

Der Compiler versucht nun, den Flaschenhals `fib3(i)` im Hauptprogramm zu eliminieren, indem er eine neue Ausgabevariable `x` generiert und dann die Invariante `x = fib3(i)` zu erhalten versucht. Induktionsvariable ist `i` und deren einzige Modifikationsstelle ist `i++`. Mit den gegebenen Regeln leitet der Compiler den Differenzcode `x = x + fib3(i-1) + fib3(i-2)` her.

```
int main()
{
  int i,x;
  for (i=1, x=fib3(i);
       i<=n;
       x=x+fib3(i-1)+fib3(i-2), i++)
    printf("%2d %10d\n",i,x);
  return 0;
}
```

Wir haben nun zwar einen Flaschenhals eliminiert, dafür aber zwei neue (`fib3(i-1)` und `fib3(i-2)`) eingeführt, die sich jedoch wieder beseitigen lassen.

Die Invariante `y = fib3(i-1)` könnte man am einfachsten durch den Differenzcode `y=x` erhalten, da in `x` ja der Wert von `fib3(i)` gespeichert ist. Es ergibt sich hier jedoch das Problem, dass `y=x` nicht *direkt* vor der Anweisung `i++` eingefügt werden darf, da die Invariante `x=fib3(i)` hier *nicht* gilt. Sie wurde nämlich soeben durch den Differenzcode der anderen Invariante `x=x+...` verletzt und wird erst hinter der Anweisung `i++` wieder gültig.

Anders betrachtet besteht das Problem darin, die parallele Zuweisung

$$\begin{pmatrix} x \\ y \\ i \end{pmatrix} = \begin{pmatrix} x + y + fib(i-2) \\ x \\ i+1 \end{pmatrix}$$

zu sequentialisieren.

Da eine zyklische Abhängigkeit zwischen `x` und `y` besteht, benötigen wir eine temporäre Variable `tmp`. Den kompletten Differenzcode zur Erhaltung beider Invarianten können wir damit z.B. schreiben als `tmp=x, x=x+y+fib3(i-2), y=tmp, i++`.

Beim Eliminieren des letzten Flaschenhalses durch eine dritte Invariante `z=fib3(i-2)` tritt das selbe Problem auf. Der Differenzcode `z=y` direkt vor `i++` wäre falsch. Wir benötigen eine weitere temporäre Variable `tmp2`. Das fertig optimierte Programm lautet schließlich

```
int main()
{
  int i,x,y,z,tmp,tmp2;
  for (i=1, x=fib3(i), y=fib3(i-1), z=fib3(i-2);
       i<=n;
       tmp=x, tmp2=y, x=x+y+z, y=tmp, z=tmp2, i++)
    printf("%2d %10d\n",i,x);
  return 0;
}
```

Die Schleife enthält, abgesehen von der Initialisierung, nun überhaupt keine Aufrufe der Funktion `fib3` mehr. Die Zeitkomplexität des optimierten Programms ist nur noch linear und seine Ausführungszeit beträgt für `n=38` nur noch wenige Millisekunden.

4.4 Spezialisierte Bibliotheken

Bestimmte Konstrukte treten in Programmen oft in Kombination auf. Dabei kann es vorkommen, dass eine solche Kombination in manchen Fällen eine sehr ineffiziente Lösung darstellt. Die Funktion `strlen` wird beispielsweise häufig in Bedingungen zusammen mit einem Vergleich verwendet.

```
while (fscanf(input,"%s",s)!=0)
  if (strlen(s)<5)
    num_short_words++;
```

Hinter `strlen` verbirgt sich aber eine Schleife, die den String abläuft und die Zeichen zählt, bis die abschließende `0` auftritt. Sollte `s` im obigen Beispiel nun ein String der Länge 30000 sein, so würde die Auswertung von `strlen(s)` ziemlich lange dauern. Auch wenn das Ergebnis des Vergleichs `strlen(s)<5` eigentlich schon nach 5 Zeichen feststeht — die Schleife in der Funktion `strlen` läuft trotzdem zu Ende.

Die Transformationsregeln ermöglichen eine Vorgehensweise, mit der solche Ineffizienzen beseitigt werden können:

- Man erstellt eine Bibliothek, in der für Konstrukte, die in der Praxis oft in Kombination auftreten, effizientere spezialisierte Funktionen definiert werden.
- Zusätzlich definiert man Ersetzungsregeln, die dem Compiler mitteilen, wo diese Funktionen zum Einsatz kommen sollen.

Eine spezialisierte Funktion für `strlen(s)<n` könnte so aussehen:

```
int strlen_less_than (char* s, int n)
{
  while (*s && n>0) { s++; n--; }
  return n>0;
}
```

Und die Ersetzungsregel, die dem Compiler sagt, wann diese Funktion eingesetzt werden soll, lautet:

```
#pragma rule ForAll char* s; int n : strlen(s)<n === strlen_less_than(s,n)
```

Ausgestattet mit diesem Wissen würde der Compiler im obigen Programmstück den Ausdruck `strlen(s)<5` durch `strlen_less_than(s,5)` ersetzen. Das optimierte Programm hätte dann ein weitaus besseres Worst-Case-Laufzeitverhalten.

4.5 Abstrakte Datentypen

Zur Optimierung von Algorithmen, die auf abstrakten Datentypen wie Listen oder Mengen operieren, lässt sich *Finite Differencing* (Kap. 4.3) ebenfalls einsetzen. Der Programmierer müsste dazu einen Datentyp implementieren und zusätzlich Gesetze angeben, die für diesen Datentyp gelten und mit denen der Compiler den Differenzcode zur Erhaltung entsprechender Invarianten berechnen kann.

Für *Finite Differencing* werden Gesetze der Form $f(x \oplus \Delta) = f(x) \otimes f_\Delta(x)$ benötigt, d.h. Gesetze, die angeben, wie der Wert von $f(x \oplus \Delta)$ schneller berechnet werden kann, wenn der Wert von $f(x)$ bereits bekannt ist. Die Operation $\otimes$ und die Funktion f_Δ sollten hier natürlich einfacher zu berechnen sein als $\oplus$ und f.

Wir stellen im folgenden zwei Beispielanwendungen vor. Abschnitt 4.5.1 zeigt die Optimierung der Berechnung einer Mengendifferenz in einer Schleife. Die Komplexität des Schleifenrumpfes sinkt dadurch von $O(n \log n)$ auf $O(\log n)$.
In Abschnitt 4.5.2 wird ein quadratischer Algorithmus, der auf Listen operiert, in einen linearen Algorithmus umgeschrieben. Die Listenoperationen werden durch die Optimierung redundant und werden daher ebenfalls wegoptimiert. Das optimierte Programm kommt somit auch mit deutlich weniger Speicherplatz aus.

4.5.1 Mengen

Angenommen, wir haben einen Datentyp `set` für Mengen von ganzen Zahlen (int) implementiert. Als Funktionen stehen zur Verfügung:

```
bool elem (int x, set s);  // liefert, ob x Element von s ist
int size (set s);          // Anzahl Elemente in s

set insert (int x, set s); // Einfuegen von x in s
set delete (int x, set s); // Loeschen von x aus s

set set_union (set s1, set s2);        // s1 vereinigt mit s2
set set_intersection (set s1, set s2); // s1 geschnitten mit s2
set set_difference (set s1, set s2);   // s1 ohne s2
```

Algebraische Eigenschaften dieser Funktionen formulieren wir als Regeln, wie z.B.:

```
#pragma rule ForAll int x; set s1,s2 :
                set_difference(insert(x,s1),s2)
                  ===
                elem(x,s1) || elem(x,s2) ?
                  set_difference(s1,s2) :
                  insert(x,set_difference(s1,s2))

#pragma rule ForAll int x; set s :
                size(insert(x,s))
                  ===
                elem(x,s) ? size(s) : size(s)+1
```

`set` sei als balancierter, geordneter Binärbaum implementiert. Für Mengen mit n Elementen seien die Komplexitäten der Funktionen $O(\log n)$ für `elem`, `insert` und `delete`, und $O(n \log n)$ für `set_union`, `set_intersection` und `set_difference`.

Wird in einer Schleife nun aber wiederholt die Differenz zweier Mengen berechnet, die von einer Iteration auf die nächste nahezu unverändert bleiben, so lässt sich die Gesamtkomplexität weiter verbessern. Im folgenden Beispiel verändert sich lediglich `s1` durch Einfügen weiterer Elemente, während `s2` konstant bleibt.

```
for (i=0; i<100000; i++)
{
   s1 = insert(i,s1);
   ...
   ... set_difference(s1,s2) ...
   ...
}
```

Mittels *Finite Differencing* kann `set_difference(s1,s2)` durch eine neue Variable `s` ersetzt werden. Der Differenzcode zur Erhaltung der Invariante `s = set_difference(s1,s2)` an der Modifikationsstelle `s1 = insert(i,s1)` kann mit Hilfe der gegebenen Regeln berechnet werden. Wir erhalten

```
for (i=0, s=set_difference(s1,s2); i<100000; i++)
{
   s = elem(i,s1) || elem(i,s2) ? s : insert(i,s);
   s1 = insert(i,s1);
   ...
   ... s ...
   ...
}
```

Der Rumpf der optimierten Schleife benötigt bei n Elementen nur noch $O(\log n)$ Zeit statt $O(n \log n)$. Voraussetzung ist dabei allerdings, dass die Funktion `insert` nur einen Zeiger auf die eigentliche Datenstruktur zurückliefert und nicht die komplette Datenstruktur selbst. Ansonsten würde diese Struktur bei den Zuweisungen an `s` und `s1` kopiert werden. Die Komplexitätsreduktion wäre dann schwächer, nämlich nur von $O(n \log n)$ auf $O(n)$.

Eine Implementierung von Mengen durch geordnete Binärbäume und einer `insert`-Funktion, die logarithmische Komplexität besitzt, aber nicht destruktiv ist, so dass durch `a = insert(x,b)` die Menge `b` unverändert bleibt, ist nicht trivial, aber möglich. Eine solche Implementierung wird von A. Appel in [App98] gegeben.

4.5.2 Listen

Wir definieren einen Datentyp `list` für Listen von ganzen Zahlen und geben die dafür geltenden Gesetze an.

```
typedef struct element
{
  int head;
  struct element *tail;
} element;

typedef element* list;
```

Die Konstruktorfunktion `cons` nimmt eine Zahl `x` und eine Liste `xs` als Argument und liefert die Liste `(x:xs)` zurück.

```
list cons (int x, list xs)
{
```

```
  list l = (list)malloc(sizeof(element));
  l->head = x;
  l->tail = xs;
  return l;
}
#pragma property Transparent(cons)
```

`length` berechnet die Länge und `sum` die Summe einer Liste.

```
int length (list xs)
{
  return xs==NULL ? 0 : length(xs->tail)+1;
}
#pragma property Transparent(length)
#pragma rule ForAll int x; list xs: length(cons(x,xs)) === length(xs)+1

int sum (list xs)
{
  return xs==NULL ? 0 : sum(xs->tail) + xs->head;
}
#pragma property Transparent(sum)
#pragma rule ForAll int x; list xs: sum(cons(x,xs)) === sum(xs)+x
```

Das zu optimierende Hauptprogramm sei:

```
int main()
{
  list xs; int i;

  xs = NULL;
  for (i=1; i<=10000; i++)
    {
      if (i%5!=0)
          xs = cons(i,xs);
      else
          xs = cons(i,cons(42,xs));
      printf("Iteration #%d: List Length = %d  ",i,length(xs));
      printf("List Sum = %d  ",sum(xs));
      printf("Average = %f  ",sum(xs)/(double)length(xs));
    }
  return 0;
}
```

Die Schleife wird 10000 mal durchlaufen. In jeder Iteration wird die Liste `xs` um ein Element vergrößert: im Normalfall wird nur das Element `i` vorne angefügt, alle 5 Iterationen wird zusätzlich vorher das Element `42` angefügt. Nach jeder Iteration wird die aktuelle Listenlänge, die Listensumme und der Durchschnitt der Listenelemente berechnet.

Da die Liste während der Ausführung wächst, steigt die Zeit pro Iteration linear, was insgesamt eine quadratische Laufzeit ergibt. Würden wir die 10000 durch 20000 ersetzen, so würde das Programm viermal länger laufen.

Während des Finite Differencing werden zwei Flaschenhälse erkannt: `sum(xs)` und `length(xs)`. Beide hängen von der gleichen Induktionsvariable `xs` ab, welche zwei Modifikationsstellen besitzt. Zu beiden Stellen lässt sich mit den gegebenen Regeln der Differenzcode zur Erhaltung der entsprechenden Invarianten berechnen:

Invariante	Modifikationsstelle	(Prä-)Differenzcode
l = length(xs)	xs = cons(i,xs)	l++
s = sum(xs)	xs = cons(i,xs)	s+=i
l = length(xs)	xs = cons(i,cons(42,xs))	l+=2
s = sum(xs)	xs = cons(i,cons(42,xs))	s+=i+42

Das optimierte Programm lautet:

```
int main()
{
  list xs; int i,l,s;

  xs = NULL;
  for (i=1, l=length(xs), s=sum(xs); i<=10000; i++)
    {
      if (i%5!=0)
        l++, s+=i, xs=cons(i,xs);
      else
        l+=2, s+=i+42, xs=cons(i,cons(42,xs));
      printf("Iteration #%d: List Length = %d  ",i,l);
      printf("List Sum = %d  ",s);
      printf("Average = %f  ",s/(double)l);
    }
  return 0;
}
```

`xs` ist nun eine tote Variable. Es ist daher unnötig, die Liste überhaupt noch mitzuführen. Eine abschließende Dead Code Elimination ergibt somit:

```
int main()
{
  int i,l,s;

  for (i=1, l=0, s=0; i<=10000; i++)
    {
      if (i%5!=0)
        l++, s+=i;
      else
        l+=2, s+=i+42;
      printf("Iteration #%d: List Length = %d  ",i,l);
      printf("List Sum = %d  ",s);
      printf("Average = %f  ",s/(double)l);
    }
  return 0;
}
```

Dieses Programm läuft in linearer Zeit und ist äußerst effizient. Es kommt außerdem ohne komplexe Datenstrukturen aus - alle Listenoperationen wurden wegoptimiert.

4.6 Konfigurierbare Optimierung

Durch Hinzufügen neuer Transformationsregeln kann das Optimierungspotential des Compilers vom Benutzer selbst erweitert werden. Analog dazu können unerwünschte Optimierungen durch das Löschen von Regeln gezielt abgeschaltet werden.

Für den Compiler-Entwickler könnte dies ebenfalls interessant sein. Angenommen, ein Kunde beschwert sich darüber, dass der ihm verkaufte Compiler Multiplikationen mit Zweierpotenzen nicht durch die schnelleren Bit-Shifts ersetzt. Anstatt nun die fehlende Optimierung implementieren zu müssen und vermutlich erst Wochen später eine neue Version veröffentlichen zu können, kann dem Kunden hier sofort geholfen werden: Es genügt, ihm die entsprechenden Zeilen mitzuteilen, die er in sein Programm einfügen muss, um die gewünschte Optimierung zu bekommen:

```
#pragma rule ForAll int n : n*2  === n<<1
#pragma rule ForAll int n : n*4  === n<<2
#pragma rule ForAll int n : n*8  === n<<3
#pragma rule ForAll int n : n*16 === n<<4
#pragma rule ForAll int n : n*32 === n<<5
...
```

Denkbar wäre für den Compiler-Entwickler auch, lediglich einen generischen Codeoptimierer zu implementieren und die eigentlichen Optimierungsregeln in einer Datei <optimizations.h> zusammenzustellen, die mit dem Compiler ausgeliefert wird. In Programme, die mit Optimierung übersetzt werden sollen, müsste der Kunde dann die Datei <optimizations.h> einbinden.

Sollten nach der Auslieferung Probleme mit einzelnen Optimierungen gemeldet werden (was in der Praxis leider fast immer der Fall ist), kann den Kunden schnell mitgeteilt werden, wie durch Einfügen, Ändern oder Löschen der entsprechenden Pragma-Direktiven die fehlerhaften Optimierungen zu korrigieren bzw. gegebenenfalls abzuschalten sind. Dies ist eine weitaus bessere Lösung als der heute leider übliche Ratschlag „Ohne Optimierung compilieren und auf das nächste Update warten“.

Kapitel 5 Implementierung

Dieses Kapitel beschreibt die Implementierung von EPCOT. Grundkenntnisse über die Entwicklung von Compilern werden dabei bereits vorausgesetzt. Eine allgemeine Einführung in das Gebiet Compilerbau wird z.B. in [WM95] oder [ASU86] gegeben.

EPCOT stellt eine Erweiterung des CADUL-C++-Compilers dar, eines kommerziellen Compilers, der von CADUL, einer in Ulm ansässigen Software-Firma vertrieben wird. Der CADUL-C++-Compiler besteht aus ca. 300000 Zeilen C-Code, EPCOT aus noch etwa 15000 zusätzlichen Zeilen C++-Code.

Abschnitt 5.1 gibt einen Gesamtüberblick über den CADUL-C++-Compiler und erklärt die Datentypen, mit denen der Syntaxbaum eines zu übersetzenden Programms repräsentiert wird. Abschnitt 5.2 stellt dann die eigenen Erweiterungen vor, die in EPCOT vorgenommen wurden.

5.1 Überblick über den verwendeten Compiler

5.1.1 Architektur

Der Ablauf der Compilierung gliedert sich in die in Abb. 5.1 gezeigten Phasen. Das C++-Quellprogramm wird geparst und liegt danach als Syntaxbaum, der noch alle C++-Konstrukte enthält, im Speicher vor. In dieser Phase werden auch die Pragmas verarbeitet, von denen manche einen sofort wirkenden Effekt haben, d.h. das Parsen beeinflussen, und andere, die zur späteren Verarbeitung vorgesehen sind, vorerst nur im Syntaxbaum abgelegt werden. (Hier werden wir später die semantischen Zusatzinformationen integrieren.) In der Lowering-Phase werden die C++-Konstrukte eliminiert. Danach liegt ein reiner C-Syntaxbaum vor. In diese Phase ist außerdem das Inlining von Funktionen integriert, sofern der Compiler mit der entsprechenden Option aufgerufen wurde. Zwischen Lowering-Phase und Code-Generierung wird im EPCOT der Quellcode-Optimierer eingebaut werden, der den C-Syntaxbaum in einen effizienteren Baum transformiert. Der Code-Generator nimmt schließlich den C-Syntaxbaum und erzeugt daraus den Maschinencode. Abschließend wird dieser Code noch durch einen Peephole Optimizer optimiert.

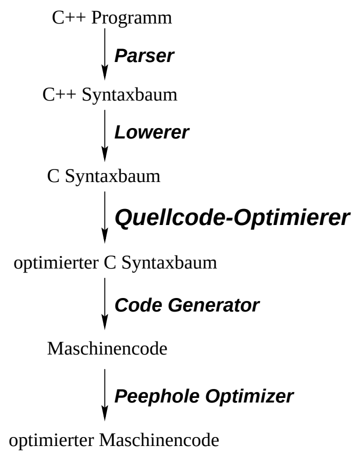


Abbildung 5.1: *Phasen der Compilierung*

5.1.2 Repräsentation des C-Syntaxbaums

C-Programme bestehen aus verschiedenen kombinierbaren Bausteinen wie Konstanten, Variablen, Routinen, Typen, Ausdrücken und Anweisungen. Jede dieser Arten wird im CADUL-Compiler durch einen eigenen Datentyp repräsentiert.

5.1.2.1 Konstanten

Der Datentyp zur Darstellung von Konstanten heißt `a_constant` und sieht - vereinfacht - wie folgt aus:

```
typedef struct a_constant
{
   a_constant_repr_kind kind;
   union
   {
     an_integer_value integer_value;
     a_float_value float_value;
     struct { ... } string;
     struct { ... } address;
     ...
   } variant;
};
```

Das Attribut `kind` gibt an, um welche Art von Konstante es sich handelt (Integer-, Float-, String-, Adress-Konstante, ...). Weitere Informationen stehen dann im entsprechenden Teil des Attributs `variant`. Ist `c` eine Stringkonstante (c.kind==ck_string), so erhält man die Länge des Strings beispielsweise mit `c.variant.string.length`.

5.1.2.2 Variablen

Der Datentyp zur Darstellung von Variablen heißt `a_variable`.

```
typedef struct a_variable
{
   a_source_correspondence source_corresp;
   a_type_ptr type;
   ...
   a_bit_field address_taken:1;
   ...
   a_variable_ptr next;
}
```

Auf den Namen einer Variablen `v` kann man mit `v.source_corresp.name` zugreifen. Allerdings ist dies der Name, der ihr nach dem *Name Mangling* bei der Konvertierung von C++ nach C zugewiesen wurde. Den ursprünglichen Namen erhält man mit `v.source_corresp.unmangled_name`. Ein nützliches Attribut ist das Flag `address_taken`. Es wird während der Syntaxanalyse gesetzt und gibt an, ob es eine Programmstelle gibt, an der die Adresse der Variable genommen wird.

5.1.2.3 Ausdrücke

Ein Ausdruck wird durch den Datentyp `an_expr_node` repräsentiert.

```
typedef an_expr_node* an_expr_node_ptr;

typedef struct an_expr_node
{
  an_expr_node_kind kind;
  union
  {
    a_constant constant;
    a_variable variable;
    struct {
      an_expr_operator_kind kind;
      an_expr_node_ptr operands;
    } operation;
  } variant;
  an_expr_node_ptr next;
};
```

Das Attribut `kind` gibt an, um welche Art von Ausdruck es sich handelt. Mögliche Werte sind hier z.B. `enk_variable` (Variable), `enk_variable_address` (Adresse einer Variablen), `enk_constant` (Konstante) oder `enk_operation` (Operation, d.h. Operator und Liste von Operanden).

Abhängig von der Art `kind` stehen dann im entsprechenden Teil des Attributs `variant` die für diese Art von Ausdruck relevanten Informationen. Handelt es sich bei einem Ausdruck `e` z.B. um eine Operation (`e.kind==enk_operation`), so erhält man mit `e.variant.operation.kind` den Operator und den ersten Operanden mit `e.variant.operation.operands`. Die Operanden sind untereinander mit dem Zeiger `next` verkettet, den zweiten Operanden erhält man also mit `e.variant.operation.operands->next`.

Abb. 5.2 verdeutlicht die Repräsentation anhand des Beispielausdrucks `x=y+4`.

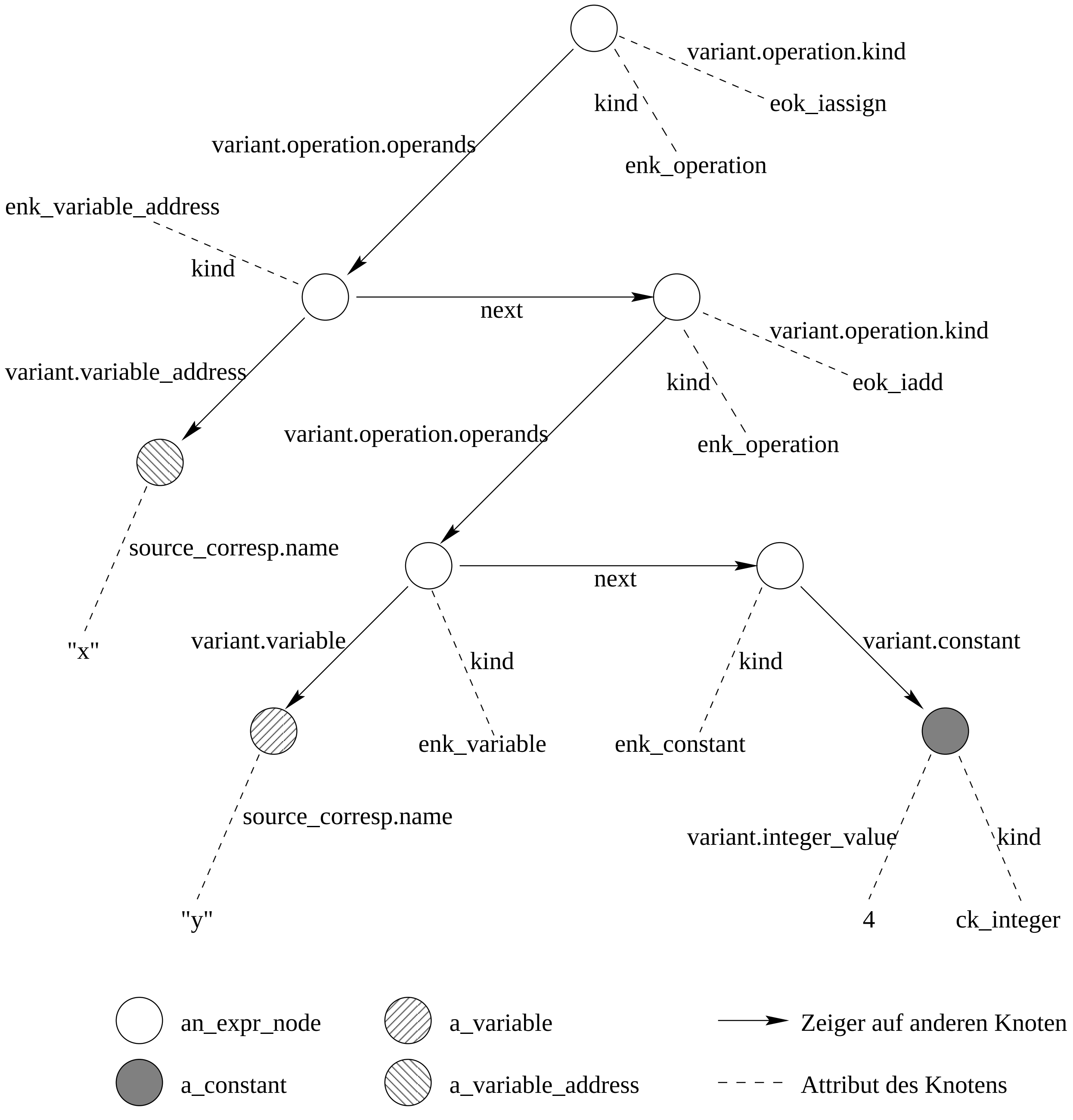


Abbildung 5.2: *Repräsentation des Ausdrucks $x = y + 4$*

5.1.2.4 Anweisungen

Zur Darstellung von Anweisungen gibt es den Datentyp `a_statement`. Wiederum gibt hier das Attribut `kind` an, um welche Art von Anweisung es sich handelt (While-Schleife, If-Anweisung, Block,...).

```
typedef struct a_statement
{
  a_statement_kind kind;
  union
  {
    // When kind == stmk_expr:
    an_expr_node_ptr expr;
    // When kind == stmk_if:
    struct {
      a_statement_ptr then_statement;
      a_statement_ptr else_statement;
    } conditional;
    // When kind == stmk_block:
    struct {
      a_statement_ptr statements;
      ...
    } block;
    ...
  } variant;
  a_statement_ptr next;
}
```

In Sequenzen von Anweisungen sind die einzelnen Anweisungen mit dem Zeiger `next` verkettet. Abb. 5.3 zeigt die Repräsentation des Programmstücks

```
{
  scanf("%d",&n);
  if (n<100)
     printf("lo");
  else
     printf("hi");
  sq = n*n;
}
```

5.1.2.5 Routinen

Funktionen werden mit dem Datentyp `a_routine` repräsentiert.

```
typedef struct a_routine
{
  a_type_ptr type;
  a_memory_region_number assoc_scope;
  ...
  a_routine_ptr next;
}
```

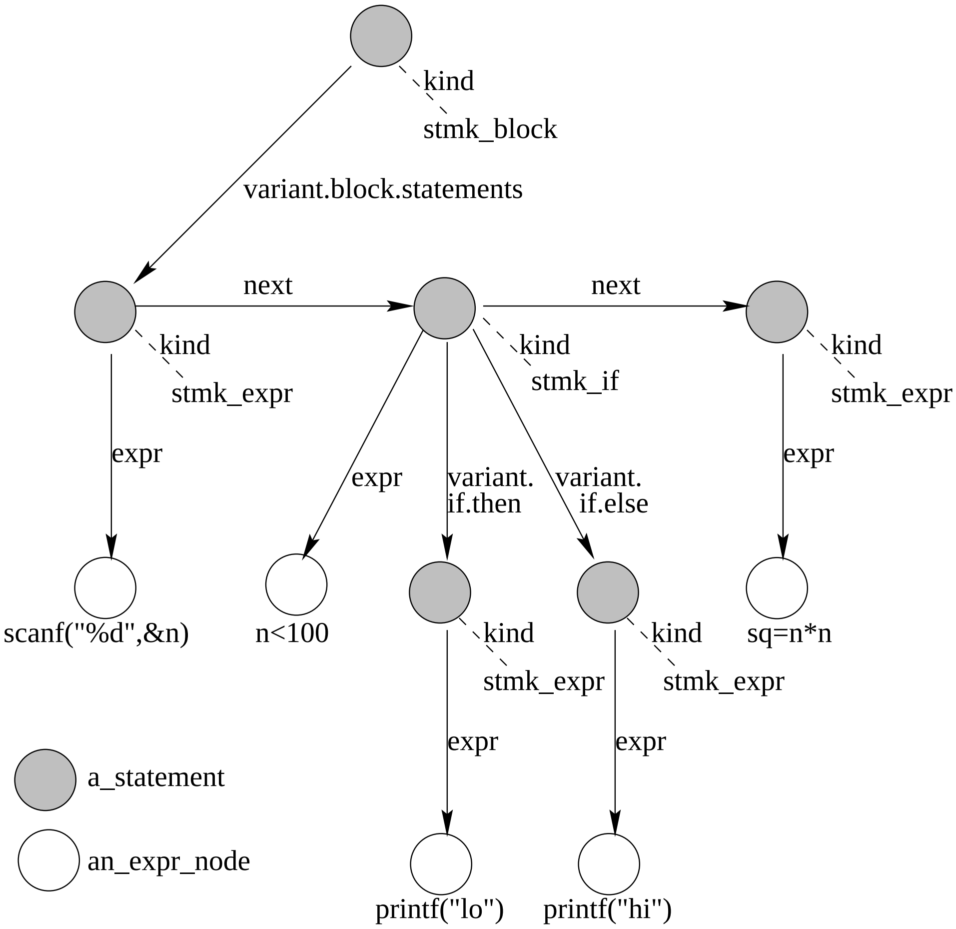


Abbildung 5.3: *Repräsentation von Anweisungen. Die hier dargestellte Anweisung ist ein Block, der seinerseits aus drei Anweisungen besteht. Die zweite davon ist ein Konditional mit Then- und Else-Zweig.*

Über das Attribut `assoc_scope` gelangt man indirekt an die Parameter und lokalen Variablen der Funktion und an den Rumpf der Funktion, welcher vom Typ `a_statement` ist. Die globale Variable `il_header` zeigt auf die Liste aller Funktionen, die im Programm vorkommen (Abb. 5.4).

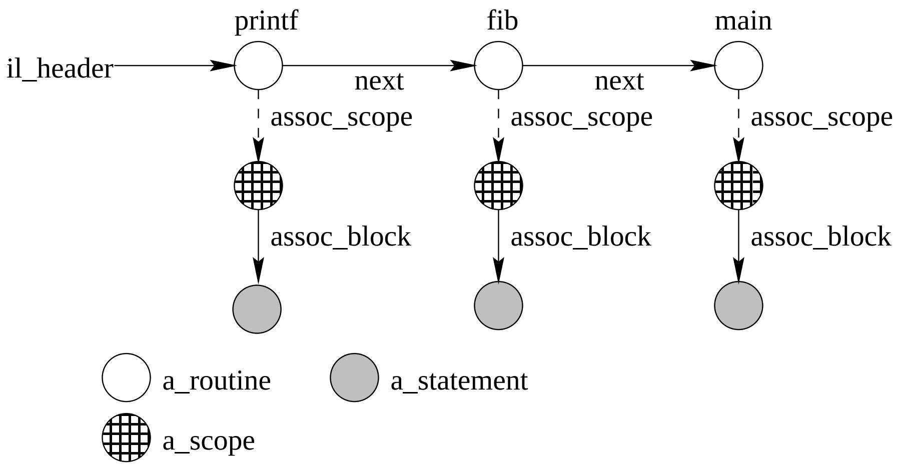


Abbildung 5.4: *Repräsentation eines kompletten Programms als Liste von Funktionen*

5.1.3 Diskussion alternativer Ansätze

Die vorgestellten Datenstrukturen waren, wie schon gesagt, bereits vorgegeben. Alternative Repräsentationen wären an einigen Stellen aber ebenfalls möglich gewesen.

Beim Schreiben des Quellcodeoptimierers, der ja auf den C-Syntaxbäumen operiert, stellte sich z.B. eine Schwäche der Repräsentation heraus:

Es gibt im Prinzip keine Unterscheidung zwischen *Ausdrücken* und *Listen von Ausdrücken*. Beide Typen werden durch `an_expr_node` repräsentiert - der Zeiger `next`, der zur Verkettung von Ausdrücken innerhalb einer Liste verwendet wird, ist Teil des Datentyps. Einige Algorithmen wurden dadurch komplizierter. Z.B. ist beim Überschreiben eines Knotens `e1` durch einen Knoten `e2` darauf zu achten, dass `e1.next` nicht auch überschrieben wird. Mit `e1 = e2` ersetzt man nicht nur `e1`, sondern die komplette Liste, die am Knoten `e1` beginnt.

Eine Alternative wäre hier die folgende Repräsentation:

```
typedef an_expr_list* an_expr_list_ptr;

typedef struct an_expr_list
{
  an_expr_node_ptr head;  // erstes Listenelement
  an_expr_list_ptr tail;  // Restliste
}
```

Ein Operator besitzt im CADUL-C++-Compiler einen Zeiger auf die Liste seiner Operanden. Auf den rechten Operanden eines binären Operators `op` greift man beispielsweise mit `op->variant.operands->next` zu. Diese Darstellung hat den Vorteil, dass Algorithmen, die den Syntaxbaum traversieren, leicht zu schreiben sind. Ein Nachteil ist dagegen, dass man schnell aus Versehen einen ungültigen Baum konstruieren kann, wie z.B. einen binären Operator mit einer Liste von drei Operanden. Eine Alternative wäre hier, die Operatoren nach ihrer Stelligkeit in Klassen zu unterteilen und jeweils entsprechend viele Zeiger auf die Operanden zu verwenden — also z.B. `left` und `right` für binäre Operatoren.

5.2 Eigene Erweiterungen

Der CADUL-C++-Compiler wurde im Rahmen dieser Arbeit sowohl um einige klassische Codeoptimierungen als auch um die eigentlichen Neuerungen, die die erweiterte Compilierung ausmachen, erweitert.

5.2.1 Grundlegende Funktionen und Datentypen

- Neu eingeführt wurde der Typ `a_symbolic_value`. Er stellt eine Vereinigung der Typen `a_constant`, `a_variable` und `an_expr_node` dar.

  ```
  typedef struct
  {
    a_symbolic_value_kind kind;
    union
    {
      a_constant_ptr constant;
      a_variable_ptr var;
      an_expr_node_ptr expr;
    } variant;
  } a_symbolic_value;
  ```

  In C++ könnte man `a_symbolic_value` auch als Oberklasse der zu vereinigenden Typen implementieren. Der CADUL-Compiler ist jedoch in C geschrieben, so dass diese „kleine Lösung“ einfacher durchzuführen war.

- Als hilfreiche Unterroutine von vielen Algorithmen stellte sich auch die Funktion `postorder_tree_walk` heraus, die eine Funktion `f` und einen Ausdruck `expr` als Argument nimmt, und dann `f` zuerst rekursiv auf die Teilausdrücke von `expr` und schließlich auf `expr` selbst anwendet.

  ```
  void postorder_tree_walk (void (*f) (an_expr_node_ptr),
                            an_expr_node_ptr expr)
  {
     an_expr_node_ptr e;
     if (expr==NULL) return;
     if (is_operation_node(expr))
        for (e=expr->variant.operation.operands;
             e!=NULL; e=e->next)
           postorder_tree_walk(f,e);
     f(expr);
  }
  ```

5.2.2 Vorwärts-Optimierungen

Constant Propagation und *CSE Elimination* werden gleichzeitig im einem Durchlauf erledigt. Dabei wird das Programm symbolisch ausgeführt und die Information über die aktuelle Variablenbelegung mitgeführt.

5.2.2.1 Datenstruktur

Als Datenstruktur zur Repräsentation dieser Information wurde eine Liste von Paaren `(a_variable, a_symbolic_value)` gewählt (Abb. 5.5).

```
typedef a_tuple* a_tuple_ptr;

typedef struct a_tuple
{
  a_variable_ptr var;
  a_symbolic_value value;
  a_tuple_ptr next;
} a_tuple;
```

Die Liste ist wiederum eingekapselt in eine Klasse `a_table`, welche die nötigen Funktionen bereitstellt, die auf der Liste operieren.

```
class a_table
{
  a_tuple_ptr root;
public:
  ...
  void assign (a_variable_ptr var,
               a_symbolic_value rhs);
  friend a_table operator & (const a_table& t1,
                             const a_table& t2);
  ...
}
```

5.2.2.2 Algorithmus

EPCOT führt nur intraprozedurale Analysen durch, d.h. jede Funktion wird einzeln optimiert. Dazu werden die Anweisungen jeder Funktion der Reihe nach symbolisch ausgeführt. Die aktuelle Variablenbelegung wird dabei in einem Objekt `tab` vom Typ `a_table` mitgeführt.

- Bei einer Zuweisung $lhs = rhs$ wird zuerst bestimmt, ob es sich bei lhs um eine Variable handelt. (lhs könnte ja ein Ausdruck wie `*p` sein.) Kann die Zielvariable der Zuweisung nicht bestimmt werden, so muss die aktuell bekannte Information vollständig gelöscht werden (`tab.erase()`). Ansonsten wird rhs evaluiert und die aktuelle Information entsprechend modifiziert (`tab.assign(lhs,rhs)`). Anschließend werden alle Paare $(var, value)$ in `tab` gelöscht, bei denen in $value$ die Variable lhs vorkommt.

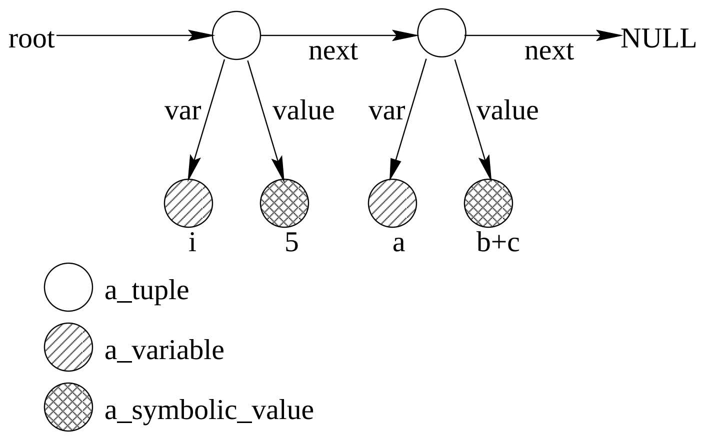


Abbildung 5.5: *Repräsentation der Information* $i = 5 \wedge a = b + c$

Beispiel: Die aktuelle Tabelle `tab` enthalte die Paare $(a, i * i)$ und $(j, 5)$, was die Information $a = i * i \wedge j = 5$ darstellt. Nun kommt die Zuweisung `i=6`. Das Paar $(i, 6)$ wird hinzugefügt. Das Paar $(a, i * i)$ muss gelöscht werden, da $a = i * i$ nun möglicherweise nicht mehr gilt.

- Bei einer Verzweigung (`if`, `switch`, `goto`) wird die Information `t` in die einzelnen Zweige hinein propagiert. Bei einer Zusammenführung, wie am Ende einer If-Anweisung oder bei einem Label, werden die Tabellen „geschnitten" (`tab = tab1 & tab2`), d.h. es werden diejenigen $(var, value)$-Paare übernommen, die in beiden Tabellen vorkommen.

- Bei Schleifen (`for`, `while`) wird eine Fixpunktberechnung durchgeführt, d.h. die Schleife wird solange symbolisch ausgeführt, bis sich die Tabelle nicht mehr ändert. Da in jeder Iteration dieser Berechnung die Tabelle um mindestens ein Paar kleiner wird, ist die Anzahl der Iterationen auf die Anzahl der im Programm vorkommenden Variablen beschränkt. (Wobei in der Praxis mehr als 2-3 Iterationen schon äußerst selten vorkommen.)

  Beispiel: Gegeben sei das Programmstück

  ```
  a = 17 * i;
  b = 19;
  while (j>10)
  {
    b++;
    j--;
  }
  n = 17 * i + b;
  ```

  In der ersten Iteration wird die Schleife mit der Tabelle $a = 17 * i \wedge b = 19$ betreten und mit der Tabelle $a = 17 * i \wedge b = 20$ verlassen. Beide Tabellen werden nun geschnitten, um die Information zu erhalten, die invariant bleibt. Wir erhalten damit $a = 17 * i$.

Die Schleife wird nun nochmal symbolisch ausgeführt, diesmal mit der Eingangsinformation $a = 17{*}i$. Man erkennt, dass am Ende der Iteration $a = 17{*}i$ immer noch gilt. Damit wurde ein Fixpunkt gefunden. $a = 17{*}i$ ist eine Schleifeninvariante — unabhängig davon, wie oft die Schleife ausgeführt wird.

Der Ausdruck `n = 17*i + b` kann somit zu `n = a + b` optimiert werden.

5.2.3 Rückwärts-Optimierungen

EPCOT's einzige Rückwärts-Optimierung ist die Entfernung unnötiger Zuweisungen (Abschnitt 2.3). Wir benötigen dazu eine Analyse über die an den jeweiligen Programmstellen lebenden Variablen.

5.2.3.1 Datenstruktur

Wir brauchen einen Datentyp für die Menge der lebenden Variablen.

```
class a_variable_set
{
public:
   a_variable_set_kind kind;
   a_link_ptr root;
}
```

`root` zeigt auf die Liste der in der Menge enthaltenen Variablen. Das Attribut `kind` kann zwei Werte annehmen: `DEAD` und `ALIVE`. Abhängig von diesem Wert gibt die Menge entweder die lebenden oder toten Variablen an. Will man also z.B. *alle* Variablen in die Menge der lebenden Variablen einfügen, so kann dies schnell erledigt werden, indem man die Liste löscht und `kind` auf `DEAD` setzt.

```
void insert_all()  // create the full set
{
   delete_list(root);
   root = NULL;
   kind = DEAD;
}
```

Es stehen außerdem Methoden zur Verfügung, die einzelne Variablen, alle globalen Variablen oder alle Variablen, deren Adresse genommen wurde, in eine Menge einfügen bzw. sie aus einer Menge löschen.

```
void insert (a_variable_ptr);
void kill (a_variable_ptr);
void insert_all();
void kill_all();
void insert_all_global_variables();
void insert_all_taken_variables();
...
```

Zwei Mengen vereinigen kann mit `s1 | s2`. Der Operator `|` wurde hierfür überladen.

```
friend a_variable_set operator | (const a_variable_set s1,
                                  const a_variable_set s2);
```

5.2.3.2 Algorithmus

Der Algorithmus verläuft analog zu den Vorwärts-Optimierungen, nur dass hier das Programm in entgegengesetzter Richtung symbolisch ausgeführt wird.

- Bei einem Ausdruck beginnen die Variablen zu leben, auf die zugegriffen wird. Kann nicht bestimmt werden, auf welche Variablen zugegriffen wird, beginnen im Zweifelsfall alle zu leben.
- Bei einer Zuweisung stirbt zunächst die Zielvariable. Kann diese nicht bestimmt werden, so stirbt keine Variable. Danach wird die rechte Seite der Zuweisung behandelt.
- Zusammengesetzte Zuweisungen wie `x+=y` werden behandelt wie die ihnen entsprechenden ausführlichen Versionen (`x=x+y`).
- Da wir das Programm entgegen der Kontrollflussrichtung symbolisch ausführen, muss das Ende einer `if`- oder `switch`-Anweisung als Verzweigung behandelt werden. Die Menge der lebenden Variablen wird hier in die einzelnen Zweige hinein propagiert. Analog stellt der Anfang einer `if`- oder `switch`-Anweisung eine Zusammenführung dar, bei der die einzelnen Mengen vereinigt werden.
- Bei Schleifen (`for`, `while`) wird wie bei den Vorwärts-Optimierungen eine Fixpunktberechnung durchgeführt, d.h. die Schleife wird solange symbolisch ausgeführt, bis sich die Menge der lebenden Variablen nicht mehr ändert.

5.2.4 Loop Unrolling

Kann bei einer Schleife die Anzahl der Iterationen zur Compile-Zeit bestimmt werden, so wird die Schleife entrollt, sofern die Programmlänge dadurch nicht zu stark anwachsen würde. Als Kriterium wurde hierfür das Produkt

*(Anzahl der Iterationen) * (Anzahl der Ausdrücke im Schleifenrumpf)*

gewählt, wobei dieses Produkt unter einem gewissen Schwellwert liegen muss. Zwei weitere Bedingungen, die erfüllt sein müssen, sind:

- Es muss sichergestellt sein, dass der Schleifenindex nicht im Schleifenrumpf verändert werden kann.
- Der Schleifenrumpf darf keine Sprungmarken (Labels) enthalten.

Für das eigentliche Entrollen ist schließlich die Funktion `unroll_for_loop` zuständig. Sie bekommt einen Zeiger auf eine Anweisung vom Typ `for (init, cond, step) body` sowie die Anzahl der Iterationen, stellt entsprechend viele Kopien von `step` und `body` her und ersetzt die Anweisung dann im Syntaxbaum durch die Sequenz `init; body; step; ... ; body; step;`.

5.2.5 Syntaxanalyse der Pragmas

Die Syntax von Pragma-Direktiven lautet nach Konvention:

`#pragma` *Schlüsselwort Datensatz*

Für den EPCOT-Compiler wurden drei neue Pragma-Direktiven eingeführt: `cost`, `property` und `rule`. Wird bei der Syntaxanalyse des Programms eines dieser Pragmas erkannt, so wird die zuständige Funktion (`cost_pragma`, `property_pragma`, `theorem_pragma`) aufgerufen, welche das Parsen des Datensatzes erledigt und die dadurch gewonnenen Daten in der entsprechenden Datenstruktur ablegt.

```
void theorem_pragma (a_pending_pragma_ptr ppp);
void property_pragma (a_pending_pragma_ptr ppp);
void cost_pragma (a_pending_pragma_ptr ppp);
```

Der Zeiger `a_pending_pragma_ptr` zeigt dabei auf die Liste der Tokens des zum Pragma gehörenden Datensatzes.

Das Parsen der Datensätze der `property`- und `cost`-Pragmas ist natürlich trivial und wird daher direkt erledigt. Die Grammatik der Transformationsregeln umfasst dagegen alle in C formulierbaren Ausdrücke sowie einige Meta-Symbole (`===, =>, ...`). Der hierfür erforderliche komplexe Parser wurde mit Bison [GNU95] generiert. Die vollständige Grammatik des Parsers wird im Anhang C angegeben.

5.2.6 Repräsentation der Transformationsregeln

Transformationsregeln werden mit einer baumartigen Darstellung repräsentiert. Konkret sieht der Datentyp folgendermaßen aus:

```
typedef struct a_rule* a_rule_ptr;

typedef struct a_rule
{
  a_rule_kind kind;
  union {
    // When kind == rk_predicate:
    struct {
      a_predicate_kind kind;
      an_expr_node_ptr operands;
    } predicate;
    // When kind == rk_all_quantor:
    struct {
      a_meta_variable_ptr var;
      a_rule_ptr body;
    } all_quantor;
    ...
  } variant;
} a_rule;
```

Das Attribut `kind` gibt dabei den Typ der einzelnen Knoten im Baum einer Regel an.

```
enum a_rule_kind
```

```
{
  rk_predicate,
  rk_all_quantor,
  rk_implication
};
```

Für den Knotentyp Prädikat (`rk_predicate`) wird mit dem Attribut `predicate.kind` unterschieden, um welches Prädikat es sich handelt.

```
enum a_predicate_kind
{
  pred_equivalence,
  pred_side_effect_free,
  pred_constant,
  pred_c_expr
};
```

Der Datentyp für die in den Transformationsregeln vorkommenden Schemavariablen heißt `a_meta_variable`.

```
typedef struct a_meta_variable
{
  a_variable_ptr var;
  a_meta_variable_ptr next;
};
```

Abbildung 5.6 zeigt die Darstellung der Transformationsregel

```
#pragma rule ForAll int c,e : Constant(c) => e*c === c*e
```

5.2.7 Termersetzungssystem und Transformationen

`rewrite(expr)` startet das Termersetzungssystem mit dem Ausdruck `expr`. In einem Postorder-Durchlauf werden die Transformationsregeln zunächst auf die Teilausdrücke von `expr` und dann auf den gesamten Ausdruck anzuwenden versucht.

Die Entscheidung, einen Postorder-Durchlauf zu verwenden, wurde aus implementierungstechnischen Gründen getroffen. Ein einzelner Durchlauf (ob Preorder oder Postorder) bringt in der Regel unvollständige Ergebnisse. Der Ausdruck `power(5,0*x)` würde bei einem Preorder-Durchlauf z.B. lediglich zu `power(5,0)` vereinfacht statt zu `1`, da erst nach der Anwendung der passenden Regel auf den Teilterm `0*x` die Anwendung einer weiteren Regel auf der äußeren Ebene möglich wird. Bei einem Postorder-Durchlauf kann allerdings das duale Problem vorkommen, nämlich dass durch die Anwendung einer Regel auf der äußersten Ebene eine weitere Anwendung auf einer inneren Ebene ermöglicht wird.

Verwendet man einen Postorder-Durchlauf, so lässt sich dieses Problem jedoch etwas leichter beheben, nämlich indem man, sobald ein Teilausdruck ersetzt wurde, das Termersetzungssystem rekursiv auf den neuen Teilausdruck anwendet. (Bei einem Preorder-Durchlauf funktioniert dies nicht.)

```
void rewrite (an_expr_node_ptr expr)
{
  postorder_tree_walk(use_rules,expr);
}
```

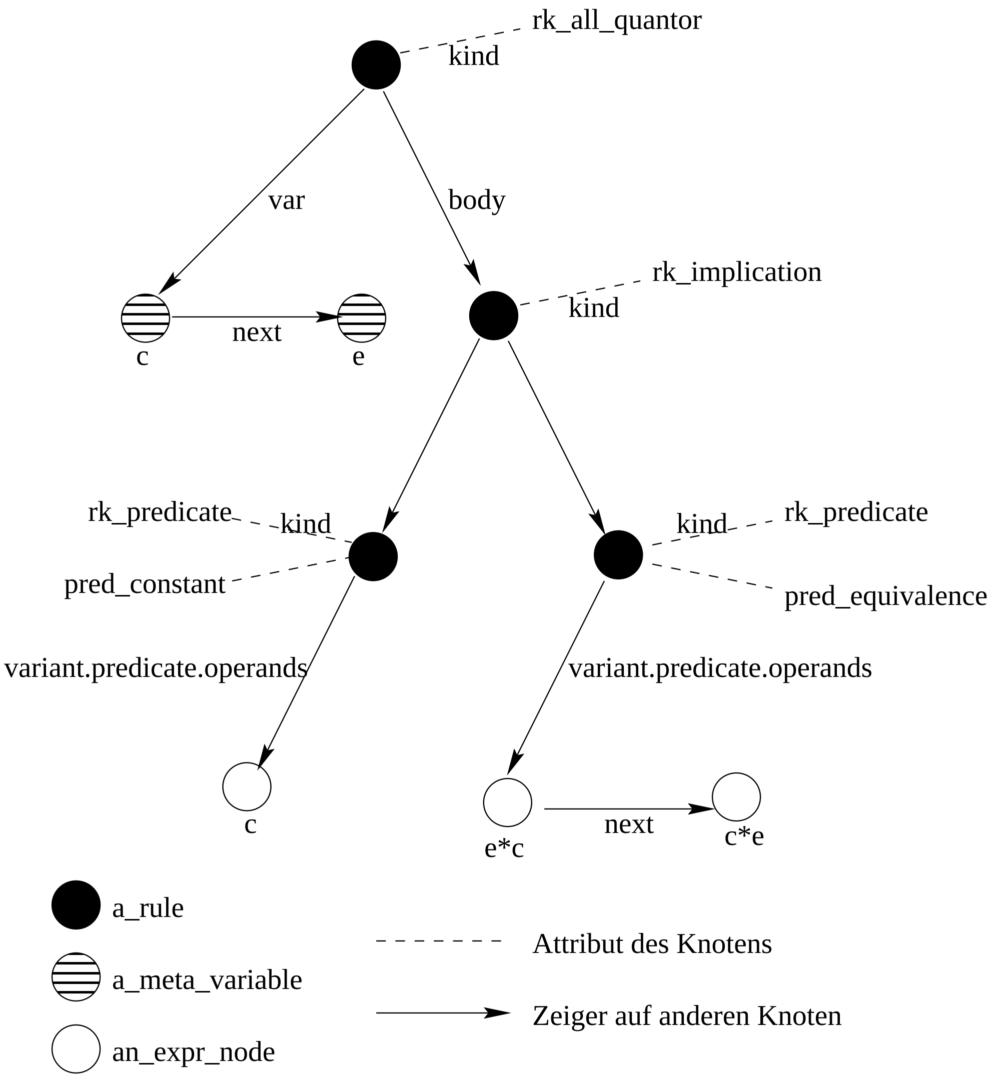


Abbildung 5.6: *Interne Darstellung der Transformationsregel* $\forall c, e : Constant(c) \Rightarrow e * c \equiv c * e$

Die Funktion `use_rules` wendet alle gegebenen Regeln der Reihe nach auf `expr` an.

```
void use_rules (expr)
{
  for (all transformation rules r)
    use_rule(r,expr);
}
```

`use_rule(r,expr)` versucht, die Regel `r` auf den Ausdruck `expr` anzuwenden.

```
void use_rule (a_rule_ptr r, an_expr_node_ptr expr);
```

- Dazu wird zunächst ein Pattern Matching zwischen der linken Seite der Regel und dem Ausdruck durchgeführt, bei der die in der Regel vorhandenen Metavariablen an die entsprechenden Teilausdrücke von `expr` gebunden werden. Ist das Matching erfolgreich, so wird die Anwendbarkeitsbedingung sowie die rechte Seite der Regel mit der gewonnenen Substitution instantiiert.

- Als nächstes wird die Bedingung überprüft. Eine Bedingung ist entweder eine Konjunktion, eine Negation, ein syntaktisches Prädikat (wie in `Constant(c) => a*c === c*a`) oder ein C-Ausdruck (wie in `n>=0 => f(n) === g(n)`). In den ersten beiden Fällen werden die Teilbedingungen rekursiv ausgewertet. Syntaktische Prädikate können leicht getestet werden. Ein C-Ausdruck wird mit der Routine für das *Constant Folding* ausgewertet. Schlägt das *Constant Folding* fehl, weil der Ausdruck Variablen enthält, so gilt die Bedingung - um auf der sicheren Seite zu sein - als nicht erfüllt.

- Ist die Bedingung erfüllt, so wird schließlich die Ersetzung durchgeführt, d.h. der Ausdruck `expr` wird im Programm durch die rechte Seite der Regel ersetzt.

- Nach einer Ersetzung wird `rewrite` rekursiv auf den neuen Ausdruck angewandt.

5.2.7.1 Pattern Matching

Eine Substitution ist eine endliche Abbildung von Metavariablen auf Ausdrücke. Die Standard Template Library (STL) von C++ stellt dafür bereits einen parametrisierten Datentyp `map` zur Verfügung, der nur noch instantiiert werden muss.

```
typedef map < a_meta_variable_ptr,
              an_expr_node_ptr,
              less <a_meta_variable_ptr> >
        a_substitution;
```

Ob eine Metavariable bereits an einen Ausdruck gebunden ist, lässt sich mit dem Makro `is_bound` feststellen.

```
#define is_bound(var,sub) ((sub).find(var)!=(sub).end())
```

Für eine gebundene Metavariable `var` liefert `sub[var]` den Ausdruck, auf den sie von der Substitution `sub` abgebildet wird. Analog lässt sich eine Substitution leicht modifizieren durch die Zuweisung `sub[var] = expr`.

Die Funktion `match` bekommt nun ein Pattern (d.h. einen Ausdruck, der Metavariablen enthalten kann) und einen weiteren Ausdruck als Argumente und liefert zurück, ob die Ausdrücke aufeinander passen. In einem dritten Argument wird zusätzlich die gefundene Substitution zurückgeliefert.

```
a_boolean match (an_expr_node_ptr pat, an_expr_node_ptr expr,
                 a_substitution& sub)
   // Performs a pattern matching
   // of a pattern "pat" against an expression "expr"
   // and returns TRUE if they match.
   // The found substitution is returned in "sub".
{
   if (is_meta_variable_node(pat))
   {
      a_meta_variable_ptr var = pat->variant.meta_variable;
      if (is_bound(var,sub))
         return match(sub[var],expr,sub);
      if (!match_types(var->type,expr->type) return FALSE;
      sub[var] = expr;
      return TRUE;
   }
   else // pattern matching of two expressions
   {
      if (pat->kind!=expr->kind) return FALSE;
      if (is_operation_node(expr))
      {
         if (!eq_operator_kinds(pat->variant.operation.kind,
                                expr->variant.operation.kind))
            return FALSE;
         return match_lists(pat->variant.operation.operands,
                            expr->variant.operation.operands,
                            sub);
      }
      return eq_expressions(pat,expr);
   }
}
```

Kapitel 6 Analyse

In diesem Kapitel wird der Prototyp-Compiler EPCOT unter verschiedenen Gesichtspunkten genauer betrachtet. Abschnitt 6.1 beschreibt, wie die Korrektheit der Implementierung getestet wurde. Abschnitt 6.2 geht auf die Leistungsfähigkeit des Compilers ein, d.h. auf die Qualität der Optimierungen, die er vornimmt. In Abschnitt 6.3 wird schließlich diskutiert, wie gut der Compiler für sehr große Programme geeignet ist, und welche Änderungen eventuell noch an ihm vorgenommen werden müssen.

6.1 Korrektheit

EPCOT wurde mit zwei kommerziellen Compiler-Test-Suiten auf Korrektheit überprüft, der *Nullstone Test Suite* und der *Perennial Test Suite*.

6.1.1 Perennial Test Suite

Die *Perennial Test Suite* besteht aus ca. 6000 C++-Programmen, die das komplette Spektrum der Sprache C++ systematisch abdecken, d.h. jedes mögliche Sprachkonstrukt wird irgendwo verwendet. Zu jedem Programm existiert außerdem eine Datei, in der die zu erwartende Ausgabe des Programms steht. Alle Programme werden nun mit dem zu testenden Compiler compiliert und ausgeführt. Ist die Ausgabe des Programms identisch zur erwarteten Ausgabe, so gilt der Test als bestanden. Werden alle Tests bestanden, so gilt der Compiler als korrekt.

EPCOT bestand alle Tests der *Perennial Suite*.

6.1.2 Nullstone Test Suite

Während die *Perennial Test Suite* als allgemeine Test Suite für C++-Compiler gedacht ist, ist die *Nullstone Test Suite*, die aus 5354 C-Programmen besteht, eine spezielle Test Suite für *optimierende* Compiler. Die Testprogramme besitzen jeweils hohes Optimierungspotential für eine bestimmte Optimierung. Compiliert man jedes Programm zweimal -

einmal mit aus- und einmal mit angeschalteter Optimierung - und vergleicht die Laufzeiten, kann man so neben der Korrektheit auch die Effektivität des Optimierers feststellen. Die Programme enthalten außerdem „Fallen“, d.h. typische Fehlerquellen für optimierende Compiler. Bei den Tests für *Loop Unrolling* gibt es beispielsweise Schleifen, deren Laufindizes im Rumpf indirekt über Zeiger verändert werden. Erkennt dies der Compiler nicht und entrollt eine solche Schleife, so fällt er bei diesem Test durch.

Die Ergebnisse des Testlaufs sind in Tabelle 6.1 wiedergegeben. Getestet wurden der EPCOT-Compiler und - zum Vergleich - auch der weit verbreitete, nicht kommerzielle GNU-Compiler.

Beide Compiler fielen bei einem Test der Sparte *Induction Variable Elimination* durch. Dieser Test wurde jedoch von beiden Compilern auch bei ausgeschalteter Optimierung nicht bestanden. Es war weiterhin nicht zu erkennen, warum der generierte Code falsch sein sollte, so dass es sich hier vermutlich um einen Bug in der Test Suite handelt.

Um echte Fehler handelt es sich jedoch bei den 59 fehlgeschlagenen Tests des GNU-Compilers, bei denen jeweils die Compilierung nicht innerhalb des vorgegebenen Zeitlimits von einer Minute terminierte. Es sieht so aus, als ob der GNU-Compiler für *Instruction Combining* und *Common Subexpression Elimination* Algorithmen verwendet, die im schlechtesten Fall exponentielle Laufzeit haben. In „normalen“ Programmen tritt dieser Fall anscheinend nie ein - bei den besagten 59 Testprogrammen der *Nullstone Suite* aber schon. Eine anschließende Stichprobe mit einem der Testprogramme ergab, dass die Compilierung auch nach 45 Minuten noch nicht terminiert hätte.

6.2 Leistungsfähigkeit

6.2.1 Nullstone Test Suite

Die Laufzeit der Programme der Nullstone Test Suite wird durch die Optimierungen von EPCOT im Schnitt um 53% gesenkt.

Tabelle 6.2 zeigt das Ergebnis des Tests. Die Testprogramme bestanden jeweils aus einer Schleife, deren Rumpf entsprechendes Optimierungspotential besaß. Gezählt wurden die Iterationen pro Sekunde, mit der diese Schleife vom compilierten Code durchlaufen wurde.

In einem zweiten Testlauf wurde EPCOT wiederum mit dem GNU-Compiler (aufgerufen mit der Option -O) verglichen. Tabelle 6.3 zeigt das Ergebnis des Vergleichs.

Im Gesamtergebnis ist EPCOT dem GNU-Compiler zwar um 23% unterlegen, doch liegt dies hauptsächlich daran, dass die meisten der getesteten Optimierungen in EPCOT noch nicht implementiert sind. Zieht man nur die implementierten Optimierungen in Betracht, d.h. Constant Folding, Constant Propagation, CSE Elimination, Dead Code Elimination, Expression Simplification, Strength Reduction, Integer Multiply Optimization und Loop Unrolling/Collapsing, so schneidet EPCOT in sieben dieser neun Disziplinen besser als der GNU-Compiler ab.

Als Zusatzinformationen wurden EPCOT nur allgemein gültige Informationen über die Sprache C und die C-Standardbibliothek gegeben (siehe Anhang B) und keine speziellen

NULLSTONE 98 Release 4.5					
		EPCOT	EPCOT	GNU	GNU
Optimization	Tests	Compile Errors	Execution Errors	Compile Errors	Execution Errors
Alias Optimization (by type)	66	.	.	.	.
Alias Optimization (const-qualified)	9	.	.	.	.
Alias Optimization (by address)	44	.	.	.	.
Bitfield Optimization	3	.	.	.	.
Branch Elimination	15	.	.	.	.
Instruction Combining	2008	.	.	40	.
Constant Folding	56	.	.	.	.
Constant Propagation	15	.	.	.	.
CSE Elimination	2128	.	.	19	.
Dead Code Elimination	256	.	.	.	.
Integer Divide Optimization	92	.	.	.	.
Expression Simplification	163	.	.	.	.
If Optimization	69	.	.	.	.
Function Inlining	33	.	.	.	.
Induction Variable Elimination	4	.	1	.	1
Strength Reduction	2	.	.	.	.
Hoisting	38	.	.	.	.
Loop Unrolling	16	.	.	.	.
Loop Collapsing	3	.	.	.	.
Loop Fusion	2	.	.	.	.
Unswitching	2	.	.	.	.
Block Merging	1	.	.	.	.
Cross Jumping	4	.	.	.	.
Integer Modulus Optimization	92	.	.	.	.
Integer Multiply Optimization	99	.	.	.	.
Address Optimization	22	.	.	.	.
Pointer Optimization	15	.	.	.	.
Printf Optimization	3	.	.	.	.
Forward Store	3	.	.	.	.
Value Range Optimization	30	.	.	.	.
Tail Recursion	4	.	.	.	.
Register Allocation	47	.	.	.	.
Narrowing	3	.	.	.	.
SPEC Conformance	2	.	.	.	.
Static Declarations	1	.	.	.	.
String Optimization	4	.	.	.	.
Overall Totals	5354	0	1	59	1

Tabelle 6.1: *Korrektheitstest von EPCOT und GNU. Bis auf einen Test (der vermutlich einen Bug in der Test Suite darstellt) bestand EPCOT alle Tests. In 59 Tests terminierte der GNU Compiler nicht innerhalb des Zeitlimits von einer Minute.*

NULLSTONE 98 Release 4.5				
Optimization	Sample Size	Not Optimized	Optimized	Delta
Alias Optimization (by type)	66 tests	302772	302416	+0%
Alias Optimization (const-qualified)	9 tests	226207	228381	+0%
Alias Optimization (by address)	44 tests	222746	222693	+0%
Bitfield Optimization	3 tests	2074412	2066435	+0%
Branch Elimination	15 tests	1725035	6331525	+72%
Instruction Combining	2008 tests	541578	532691	-1%
Constant Folding	56 tests	536687	13065348	+95%
Constant Propagation	15 tests	843613	3078816	+72%
CSE Elimination	2128 tests	91703	260789	+64%
Dead Code Elimination	256 tests	2377907	5669251	+58%
Integer Divide Optimization	92 tests	1111255	1080298	-2%
Expression Simplification	163 tests	3294492	4403061	+25%
If Optimization	69 tests	1241312	1255800	+1%
Function Inlining	33 tests	747803	740612	+0%
Induction Variable Elimination	4 tests	15455	21301	+27%
Strength Reduction	2 tests	17426	97423	+82%
Hoisting	38 tests	23651	31969	+26%
Loop Unrolling	13 tests	3480289	3572223	+2%
Loop Collapsing	3 tests	17926	169323	+89%
Loop Fusion	2 tests	15651	46261	+66%
Unswitching	2 tests	21216	28159	+24%
Block Merging	1 tests	118153	114975	-2%
Cross Jumping	4 tests	294710	266504	-9%
Integer Modulus Optimization	92 tests	1003039	963491	-3%
Integer Multiply Optimization	99 tests	1676244	2391201	+29%
Address Optimization	22 tests	597473	987948	+39%
Pointer Optimization	15 tests	902636	906012	+0%
Printf Optimization	3 tests	5698	5693	+0%
Forward Store	3 tests	127140	3170785	+95%
Value Range Optimization	30 tests	159587	616175	+74%
Tail Recursion	4 tests	3555	4573	+22%
Register Allocation	47 tests	5035	5008	+0%
Narrowing	3 tests	83753	138749	+39%
Static Declarations	1 tests	73912	509017	+85%
String Optimization	4 tests	725401	723164	+0%
Average Nullstone Rate	5349 tests	705870	1534238	+53%

Tabelle 6.2: *Test von EPCOT's Leistungsfähigkeit. Die Zahlen geben die Iterationen pro Sekunde an, die bei der Ausführung der nicht optimierten bzw. optimierten Programme gemessen wurden. Die letzte Spalte zeigt die durch die Optimierung erzielte Zeitersparnis.*

NULLSTONE 98 Release 4.5				
Optimization	Sample Size	GNU	EPCOT	Delta
Alias Optimization (by type)	66 tests	1686206	302416	-82%
Alias Optimization (const-qualified)	9 tests	701963	228381	-67%
Alias Optimization (by address)	44 tests	1237109	222693	-81%
Bitfield Optimization	3 tests	674054	2066435	+67%
Branch Elimination	15 tests	5499581	6331525	+13%
Instruction Combining	2008 tests	1222183	532691	-56%
Constant Folding	56 tests	4710931	13065348	+63%
Constant Propagation	15 tests	4753407	3078816	-35%
CSE Elimination	2128 tests	165878	260789	+36%
Dead Code Elimination	256 tests	4373929	5669251	+22%
Integer Divide Optimization	92 tests	1406082	1080298	-23%
Expression Simplification	163 tests	3725517	4403061	+15%
If Optimization	69 tests	2519354	1255800	-50%
Function Inlining	33 tests	924254	740612	-19%
Induction Variable Elimination	4 tests	8735	21301	+58%
Strength Reduction	2 tests	49184	97423	+49%
Hoisting	38 tests	37670	31969	-15%
Loop Unrolling	13 tests	3823410	3572223	-7%
Loop Collapsing	3 tests	44426	169323	+73%
Loop Fusion	2 tests	17941	46261	+61%
Unswitching	2 tests	53665	28159	-47%
Block Merging	1 tests	123652	114975	-7%
Cross Jumping	4 tests	307079	266504	-13%
Integer Modulus Optimization	92 tests	1355578	963491	-28%
Integer Multiply Optimization	99 tests	1710649	2391201	+29%
Address Optimization	22 tests	1805684	987948	-45%
Pointer Optimization	15 tests	947195	906012	-4%
Printf Optimization	3 tests	5410	5693	+4%
Forward Store	3 tests	42734	3170785	+98%
Value Range Optimization	30 tests	217075	616175	+64%
Tail Recursion	4 tests	6455	4573	-29%
Register Allocation	47 tests	5116	5008	-2%
Narrowing	3 tests	22219198	138749	-99%
Static Declarations	1 tests	268101	509017	+47%
String Optimization	4 tests	3433077	723164	-78%
Average Nullstone Rate	5349 tests	2002356	1534238	-23%

Tabelle 6.3: *Vergleich von EPCOT und GNU bezüglich ihrer Leistungsfähigkeit. Die Zahlen geben die Iterationen pro Sekunde an, die bei der Ausführung der optimierten Programme gemessen wurden. Die letzte Spalte zeigt den Unterschied zwischen den Compilern.*

Informationen über die Testprogramme, da dies ein unfairer Vorteil für EPCOT gewesen wäre. Die Testprogramme enthalten nämlich oft Konstrukte der folgenden Art:

```
a = gen_int(4);
b = a+a+a+a+a+a+a+a+a;
c = a+a+a+a+a+a+a+a+a;
```

`gen_int` ist dabei die identische Abbildung. Die Einkapselung der 4 in diese Funktion macht eine *Constant Propagation* unmöglich und zwingt den Compiler zur *CSE Elimination*. Würde man EPCOT nun aber die Regel

```
#pragma rule ForAll int n : gen_int(n) === n
```

geben, so hebt man diesen Effekt wieder auf, und die effektivere Optimierung *Constant Propagation* kommt viel öfter zum Einsatz.

6.2.2 ACM Programming Contest

Da die Programme der Nullstone Suite eigentlich unnatürliche Programme sind, wurde ein zweiter Test mit acht Programmen durchgeführt, die für den ACM Southwestern European Programming Contest 1998 entworfen worden waren [1]. Die Programme haben eine Länge von durchschnittlich 150 Zeilen und stellen Lösungen zu typischen Informatikproblemen dar. Bei der Entwicklung der Programme wurde aufgrund der Art des Wettbewerbs sehr auf Effizienz geachtet, so dass zu erwarten war, dass sie vom Compiler nur noch sehr schwach optimiert werden können.

Programm	Laufzeit ohne Optimierung	Laufzeit mit Optimierung	Steigerung
Bugs	0.99	0.99	0 %
Centauri	2.09	2.09	0 %
Dice	0.31	0.31	0 %
Fastfood	3.23	1.96	65 %
Fuses	0.31	0.31	0 %
Games	0.99	0.71	39 %
Optimal	1.14	1.04	10 %
Slides	4.85	2.90	67 %

Tabelle 6.4: *Laufzeiten der Musterlösungen zu den im ACM Southwestern European Programming Contest 1998 gestellten Probleme in Sekunden, mit und ohne vorheriger Quellcodeoptimierung.*

Bei vier von acht Programmen waren noch Geschwindigkeitssteigerungen möglich, überraschenderweise in zwei Fällen sogar signifikant um 65 bzw. 67 %. Der größte Beitrag wurde dabei jeweils von der Optimierung *Finite Differencing* verursacht. Ein Vergleich des nicht optimierten und des optimierten Quellcodes der Programme „Fastfood“ und „Slides“ befindet sich im Anhang A.

[1] http://www.informatik.uni-ulm.de/acm/swerc98/

6.2.3 Laufzeitverbesserung um Größenordnungen

Laufzeitverbesserungen um Größenordnungen, d.h. echte Reduktionen in der Komplexität, sind zwar möglich, treten aber eher bei akademischen Beispielen auf. Einige Beispiele dafür wurden in den Abschnitten 4.3 und 4.5 gegeben.

6.3 Skalierbarkeit

Der Compiler ist grundsätzlich für den kommerziellen Einsatz geeignet. Auch große Applikationen, die aus 100000 oder mehr Codezeilen bestehen, stellen für ihn kein Problem dar.

Einschränkend muss hier jedoch gesagt werden, dass die in der Industrie zu compilierenden Programme bisher natürlich noch nicht mit Zusatzinformationen angereichert sind. Da zu erwarten ist, dass die Kunden von der Möglichkeit, Transformationsregeln zu definieren, eher spärlichen Gebrauch machen werden, wurden für die kommerzielle Version einige Optimierungen, die eigentlich auch über die Regeln formulierbar wären, hart im Compiler verdrahtet. In der aktuell vertriebenen Version des Compilers ist die Möglichkeit, über die Pragma-Direktiven Zusatzinformationen anzugeben, sogar deaktiviert.

Sollten in einem großen Programm sehr viele Transformationsregeln verwendet werden, so könnte sich dies mit einer längeren Compilierzeit störend bemerkbar machen. Die Regeln werden nämlich zur Zeit als verkettete Liste repräsentiert. Wenn für einen Ausdruck eine passende Regel gesucht wird, wird im Worst Case die ganze Liste durchlaufen und ein Pattern Matching mit jeder Regel durchgeführt. Die Compilierzeit, die vom Optimierer benötigt wird, steigt daher linear mit der Anzahl der Regeln.

Die Zeit für ein Pattern Matching liegt, wie Messungen auf einem PC mit 200 MHz ergeben haben, im Durchschnitt bei etwa 10 Mikrosekunden. Die Compilierzeit eines Programms, dass aus 100000 Ausdrücken besteht, steigt also durch jede weitere Transformationsregel um etwa eine Sekunde.

In einer zukünftigen Programmversion müsste also die interne Repräsentation der Regeldatenbank verbessert werden. Ein Ansatz dazu wäre, ähnliche Regeln zusammenzufassen.

Angenommen, das Programm enthält die Regeln

```
#pragma rule ForAll int n : n*2  === n<<1
#pragma rule ForAll int n : n*4  === n<<2
#pragma rule ForAll int n : n*8  === n<<3
#pragma rule ForAll int n : n*16 === n<<4
#pragma rule ForAll int n : n*32 === n<<5
```

so könnte man, anstatt diese 5 Regeln in einer Liste zu repräsentieren, dies intern sinngemäß zusammenfassen zu

```
#pragma rule ForAll int n : n*{1,2,4,8,16,32} === n<<{1,2,3,4,5}
```

Bei einem Ausdruck wie `x+y`, der an der Wurzel einen anderen Operator als `*` hat, wüsste man dann bereits nach einem fehlgeschlagenen Pattern Matching, dass auf ihn keine der Regeln anwendbar ist.

Kapitel 7 Zusammenfassung und Ausblick

Das Ziel dieser Arbeit war, zu zeigen, welche Möglichkeiten sich in der Quellcodeoptimierung von C/C++-Programmen ergeben, wenn dem Compiler semantische Zusatzinformationen über das Programm vermittelt werden können. Verschiedene vielversprechende Einsatzgebiete dieser sogenannten *erweiterten Compilierung* wurden aufgezeigt. Implementierungstechnische Aspekte wurden anhand des im Rahmen dieser Arbeit entwickelten Prototyp-Compilers EPCOT diskutiert.

7.1 Zusammenfassung

Die wichtigsten Beiträge dieser Arbeit sind:

- Die Identifikation der Arten von semantischen Zusatzinformationen, die einem Compiler bei der Codeoptimierung helfen würden (Abschnitt 3.2).
- Die Vorstellung eines einfachen Weges, ein Programm mit solchen Informationen anzureichern (Abschnitte 3.3 und 3.4).
- Die Vorstellung möglicher Verwendungszwecke der Zusatzinformationen, die über gewöhnliche Compilierung hinausgehen, wie z.B. Optimierung in Abhängigkeit von der Hardware (Abschnitt 3.4.3) oder Konfiguration des Optimierers für spezielle Anwendungsgebiete (Kapitel 4).
- Die Automatisierung des Verfahrens *Symbolic Finite Differencing* und dessen Integration in einen Compiler (Abschnitt 4.3).
- Die Entwicklung eines in der Industrie einsetzbaren Compilers, in dem die in dieser Arbeit beschriebenen Verfahren implementiert sind (Kapitel 5).

In Kapitel 3 werden zunächst Eigenschaften von Funktionen, Operationskosten und vor allem Transformationsregeln für C-Ausdrücke als geeignete semantische Zusatzinformationen vorgestellt. Verschiedene Wege, diese Informationen einem C/C++-Compiler zugänglich zu machen, werden diskutiert. Als vielversprechendsten Weg sehen wir hier Pragma-Direktiven an.

Kapitel 4 stellt die Vorteile und die möglichen Einsatzgebiete der erweiterten Compilierung vor. Beispielsweise können algebraische Optimierungen wie die Vereinfachung von Ausdrücken in Abschnitt 4.2 dem Compiler schnell beigebracht werden, ohne seinen eigentlichen Quellcode zu ändern. Für die Funktionen der C-Standardbibliothek können, wie in Abschnitt 4.4 gezeigt, spezielle Optimierungen definiert werden, bei denen die Kenntnis über die Semantik der Funktionen genutzt wird. Zu abstrakten Datentypen (Abschnitt 4.5) können algebraische Gesetze angegeben werden, die dem Compiler algorithmische Optimierungen ermöglichen, welche die Komplexität senken können.

Das interessanteste Anwendungsgebiet ist aber sicherlich die Automatisierung von bisher nur in interaktiven Systemen eingesetzten Optimierungsverfahren. Ein Beispiel hierfür ist *Symbolic Finite Differencing*, welches in Abschnitt 4.3 vorgestellt wird. Es wird gezeigt, wie man unter Zuhilfenahme von semantischen Zusatzinformationen dieses Verfahren auch in einem optimierenden Compiler zum Einsatz bringen kann.

Kapitel 5 beschreibt die Implementierung von EPCOT, dem Compiler, der im Laufe dieser Arbeit entstanden ist und in dem diese Ideen verwirklicht sind. Dabei wird auf die Datentypen, die zur Repräsentation des Programms und der Zusatzinformationen verwendet werden, sowie auf die Algorithmen des Optimierers eingegangen.

Die Erfahrungen, die mit EPCOT gemacht wurden, werden schließlich in Kapitel 6 diskutiert.

7.2 Ausblick

Ein Gebiet für mögliche zukünftige Forschung ist die Erweiterung und Verfeinerung der Arten von Zusatzinformationen, die dem Compiler über die Pragma-Direktiven gegeben werden können.

Beispielsweise ließe sich die Hierarchie der Funktionseigenschaften (Abb. 3.1) noch weiter verfeinern. Wir haben in Abschnitt 3.2.1.3 den Begriff der zustandseffektfreien Funktion definiert. Die Standardfunktion `scanf` ist nicht zustandseffektfrei, da sie ja an der ihr übergebenen Adresse eine Eingabe entgegennimmt. Allerdings ist dies der einzige Seiteneffekt von `scanf`, und dass der Compiler nach einem Aufruf von `scanf("%d",&n)` die Information über alle Variablen verwirft, ist schade.

```
a = 5;
scanf("%d",&n);
b = a;
```

Hier wäre eventuell eine neue Klasse von Funktionen angebracht, deren Seiteneffekte sich nur auf die Argumente auswirken.

Die Überprüfung der Anwendbarkeitsbedingungen der Transformationsregeln ist ebenfalls noch stark verbesserbar. Bisher wird die Bedingung einer Regel lediglich mit der gefundenen Substitution instantiiert und dann ein *Constant Folding* aufgerufen. Bei vielen Bedingungen ist jedoch mit diesem einfachen Verfahren die Erfülltheit nicht zu entscheiden. Betrachten wir dazu das folgende Programmstück:

```
for (i=1; i<100; i++)
  f(i*i);

#pragma ForAll int n : C(n<10000) => f(n) === g(n)
```

Die Regel soll ausdrücken, dass für Argumente kleiner 10000 die Funktion `f` durch die spezialisierte Funktion `g` ersetzt werden soll.

Ein *Constant Folding* des Ausdrucks `i*i<10000` geht aber natürlich schief, da in dem Ausdruck die Variable `i` enthalten ist. Die Regel kommt folglich nicht zum Einsatz. Bei Betrachtung des Kontexts fällt aber auf, dass `i` nur Werte zwischen 1 und 99 annehmen kann und `i*i<10000` daher immer erfüllt ist. Der Compiler müsste hier eine Intervallanalyse [NNH99] durchführen, um zu erkennen, dass hier $1 \leq i \leq 99$ gilt. Ferner würde ein automatisches Beweissystem benötigt, das dann die Aussage $1 \leq i \leq 99 \Rightarrow i * i < 10000$ beweisen kann.

Eine andere interessante Fragestellung ist, wie man die Sprache der Pragma-Direktiven erweitern muss, um damit komplette Optimierungen beschreiben zu können. Wie in Abschnitt 4.2 gezeigt wurde, kann man die algebraische Vereinfachung von Ausdrücken dem EPCOT-Compiler durch die Angabe entsprechender Pragma-Direktiven beibringen, ohne den Quellcode des Compilers zu ändern. *Symbolic Finite Differencing* war hingegen zu komplex, um vollständig durch Pragma-Direktiven beschrieben zu werden. Pragmas werden hier zwar verwendet, um dem Compiler programmspezifische Informationen zu geben - der eigentliche Algorithmus des Finite Differencing ist jedoch hart im Compiler implementiert und umfasst ca. 1000 Zeilen Code. Wenn die Sprache der Pragma-Direktiven ausdrucksstark genug wäre, um auch solche komplexen Optimierungen zu spezifizieren, hätte man einen echten konfigurierbaren Compiler geschaffen, der vom Benutzer beliebig erweitert werden kann. Wenn die dazu notwendige Sprache allerdings genau so komplex wie C++ wäre, so hätte man eigentlich gar nichts gewonnen, denn dann könnte man dem Benutzer auch gleich den Quellcode des Compilers geben und ihm sagen, er könne ja weitere Optimierungen einbauen, wenn er wolle. Die Kunst besteht hier darin, eine einfache Sprache zu finden, in der dennoch komplexe Optimierungen spezifiziert werden können.

Anhang A Beispiele für Finite Differencing

Beispielprogramme, in denen das *Finite Differencing* von EPCOT erfolgreich angewendet werden konnte, wurden an verschiedenen Stellen innerhalb dieser Arbeit vorgestellt. In diesem Kapitel möchten wir noch drei weitere interessante Fallbeispiele präsentieren.

Im Beispiel A.1 wird gezeigt, wie nichtlineare Polynome in mehreren Variablen mittels Finite Differencing durch simple Additionen ersetzt werden. In anderen Compilern ist diese Optimierung zumeist nur für *lineare* Polynome in *einer* Variable implementiert.

Die Beispiele A.2 und A.3 sind Musterlösungen zu den Aufgaben *Fast Food* und *Sorting Slides*, welche im ACM Southwestern European Programming Contest 1998 gestellt wurden. Die Geschwindigkeit dieser Programme konnte von EPCOT um 65 bzw. 67 % gesteigert werden (Abschnitt 6.2.2).

A.1 Polynome in mehreren Variablen

Die folgende Schleife enthält den Flaschenhals `i*j*k`, ein Polynom in drei Variablen. Die Modifikationsstellen der Induktionsvariablen sind `i++`, `j++` und `k++`.

```
for (c=read(); c!='.'; c=read())
{
  if      (c=='i') i++;
  else if (c=='j') j++;
  else if (c=='k') k++;
  print(i*j*k);
}
```

Dem Compiler bekannt sei die Distributivität der Multiplikation über die Addition, die Neutralität der 1 bezüglich der Multiplikation, sowie die Zustandseffektfreiheit der Funktionen `read` und `print`.

```
#pragma rule ForAll int a,b,c : SideEffectFree(c) => (a+b)*c === a*c+b*c
#pragma rule ForAll int a,b,c : SideEffectFree(c) => c*(a+b) === c*a+c*b
#pragma rule ForAll int n : 1*n === n
#pragma rule ForAll int n : n*1 === n
```

```
#pragma property StateEffectFree(read)
#pragma property StateEffectFree(print)
```

Mit Hilfe dieser Regeln kann der Differenz-Code für alle drei Modifikationsstellen berechnet werden. Es ergibt sich das Programm:

```
for (c=read(), x=i*j*k; c!='.'; c=read())
{
  if (c=='i') x+=j*k, i++;
  if (c=='j') x+=i*k, j++;
  if (c=='k') x+=i*j, k++;
  print(x);
}
```

In dem generierten Differenz-Code sind nun drei neue Flaschenhälse enthalten (`j*k`, `i*k` und `i*j`), die ebenfalls mit *Finite Differencing* beseitigt werden können.

```
for (c=read(), x=i*j*k, pij=i*j, pik=i*k, pjk=j*k;
     c!='.'; c=read())
{
  if (c=='i') x+=pjk, pij+=j, pik+=k, i++;
  if (c=='j') x+=pik, pij+=i, pjk+=k, j++;
  if (c=='k') x+=pij, pik+=i, pjk+=j, k++;
  print(x);
}
```

Insgesamt wurden die zwei Multiplikationen pro Iteration durch drei Additionen und Zuweisungen ersetzt. Dies kann auf manchen Prozessoren auch eine Verschlechterung sein. Ob die Optimierung von EPCOT tatsächlich durchgeführt wird, hängt daher auch von den Einstellungen der Kosten der Operationen `*` und `+=` ab (Kapitel 3.2.3).

Eine Schwäche ist hier, dass der Compiler noch nicht erkennt, dass sich die drei Bedingungen gegenseitig ausschließen und daher wirklich nur drei (und nicht neun) Additionen/Zuweisungen pro Iteration ausgeführt werden. Bei einer für den 80486-Prozessor geeigneten Kosteneinstellung (z.B. 2 Zyklen für `+=`, 13 Zyklen für `*`) wird die Optimierung aber immer noch als lohnend betrachtet ($9 * 2 < 2 * 13$).

A.2 Fast Food

Im Problem *Fast Food* ist eine Straße gegeben, entlang der n Restaurants stehen, und zwar an den Kilometersteinen $d_1, d_2, ..., d_n$. Jedes Restaurant wird von genau einem Depot beliefert, aber ein Depot kann auch mehrere Restaurants beliefern. Es sollen nun k Restaurants ausgewählt werden, an denen die Depots errichtet werden sollten, so dass die Summe der Distanzen der Restaurants zu ihren jeweils nächstliegenden Depots minimiert wird.

Das folgende Programm verwendet dazu die Technik des *Dynamischen Programmierens* [Sch97, CLR90].

```
#include "stdio.h"
#include "math.h"

#define MAX_STORES 200
#define MAX_DEPOTS 30

FILE *inp;

int pos[MAX_STORES];
int best[MAX_DEPOTS][MAX_STORES+1];
short back[MAX_DEPOTS][MAX_STORES+1];
int cost[MAX_STORES][MAX_STORES+1];
int k,n;

int caseno = 1;
```

Mit `cost[a][b]` bezeichnen wir den Beitrag zur Distanzsumme, den wir erhalten, wenn wir ein Depot errichten, dass die $b-a$ Restaurants $a, a+1, ..., b-1$ beliefert.

```
void compute_cost()
{
  int i,j,l,med,err;

  for(i=0;i<n;i++)
    for(j=i+1;j<=n;j++)
      {
        /* the median is always the best position */
        med = pos[i+(j-i)/2];
        err = 0;
        for(l=i;l<j;l++)
          err += abs(pos[l] - med);
        cost[i][j] = err;
      }
}
```

Durch *Finite Differencing* ließ sich hier die Berechnung der Adresse des Elements `cost[i][j]` verbessern.

```
void compute_cost(void)
{
  auto int *_sr12;
  auto int i; auto int j; auto int l; auto int med; auto int err;

  for (i=0; i<n; i++)
    for (j=1+i, _sr12 = (((int *)(cost+i))+(1+i));
         j<=n; _sr12++ , j++)
      {
        med = ((pos[i + ((j - i) >> 1)]));
        err = 0;
        for(l=i; l<j; l++)
          err += abs(pos[l] - med);
        (*_sr12) = err;
      }
}
```

Wir bezeichnen mit `best[a][b]` die minimale Distanzensumme, die sich ergibt, wenn `a+1` Depots auf die ersten `b` Restaurants verteilt werden. Was wir letztendlich berechnen wollen, ist der Wert `best[k-1][n]`. Das Feld `back` verwenden wir dann, um die eigentliche Lösung zu rekonstruieren.

```
void solve_case()
{
  int i,j,b,first,last;
  int sol[MAX_DEPOTS];

  compute_cost();

  for(j=1;j<=n;j++)
    {
      best[0][j] = cost[0][j];
      back[0][j] = 0;
    }
  for(i=1;i<k;i++)
    for(j=1;j<=n;j++)
      {
        best[i][j] = 9999999;
        for(b=i;b<j;b++)
          if(cost[b][j]+best[i-1][b] < best[i][j])
            {
              best[i][j] = cost[b][j] + best[i-1][b];
              back[i][j] = b;
            }
      }
  for(i=k-1,j=n;i>=0;j=back[i--][j]) sol[i] = back[i][j];

  printf("Chain %d\n",caseno++);
  for(i=0;i<k;i++)
    {
      first = sol[i];
      last = i!=k-1?sol[i+1]-1:n-1;
      printf("Depot %d at restaurant %d serves restaurant",
             i+1,first+(last-first+1)/2+1);
      if(first == last)
        printf(" %d\n",first+1);
      else
        printf("s %d to %d\n",first+1,last+1);
        }
  printf("Total distance sum = %d\n\n",best[k-1][n]);
}
```

Insgesamt sechs Flaschenhälse wurden hier durch *Finite Differencing* beseitigt.

```
void solve_case(void)
{
  auto short *_sr52; auto int *_sr40; auto int *_sr31;
  auto short *_sr25; auto int *_sr24; auto int *_sr22;
  auto int i; auto int j; auto int b; auto int first; auto int last;
  auto int sol[30];
```

```
  compute_cost();
  for (j=1, _sr25 = (((short *)(back)) + 1),
       _sr24 = (((int *)(cost)) + 1) ,
       _sr22 = (((int *)(best)) + 1); j <= n;
       _sr22++, _sr24++, _sr25++, j++)
    {
      *_sr22 = ((*_sr24));
      *_sr25 = ((short)0);
    }
  for (i=1; i<k; i++)
    for (j=1, (_sr52 = (((short *)(back + i)) + 1))),
         (_sr40 = (((int *)(best + i)) + 1)); (j <= n);
         (_sr40++) , ((_sr52++) , (j++)))
      {
        *_sr40 = 9999999;
        for (b=i, _sr31 = (((int *)(best+((-1)+i)))+i);
             b<j; (_sr31++) , (b++))
          if (((((int *)(cost+b))[j]))+(*_sr31)) < *_sr40)
            {
              *_sr40 = (((((int *)(cost+b))[j]))+(*_sr31));
              *_sr52 = ((short)b);
            }
      }
  ...
}
```

Die restlichen Funktionen blieben bei der Optimierung unverändert:

```
int read_case()
{
  int i;
  fscanf(inp,"%d %d",&n,&k);
  if(n == 0) return 0;
  skip_line();
  for(i=0;i<n;i++)
    fscanf(inp,"%d",pos+i);
  return 1;
}

int main()
{
  inp = fopen("fastfood.in","r");
  while(read_case()) solve_case();
  fclose(inp);
  return 0;
}
```

A.3 Sorting Slides

Im Problem „Sorting Slides“ ist eine Menge von durchsichtigen, mit Zahlen beschrifteten Folien gegeben, die so angeordnet ist, dass sich die Folien teilweise überlappen. Die Aufgabe besteht darin, alle Paare von Folien und Zahlen zu bestimmen, die eindeutig einander zugeordnet werden können (Abb. A.1).

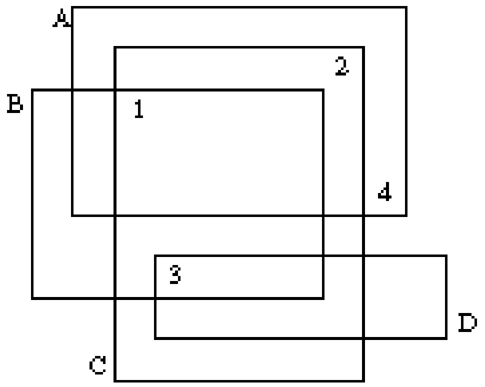


Abbildung A.1: *Die 4 befindet sich eindeutig auf Folie A. Damit muss sich die 2 auf Folie C befinden. Für Folie D kommt nur die 3 in Frage. Daraus folgt, dass Folie B mit 1 beschriftet ist.*

Abstrakt gesehen gilt es hier, die Schnittmenge aller maximalen Matchings zwischen Folien und Zahlen zu finden, wobei eine Folie mit einer Zahl durch eine Kante verbunden ist, wenn sich die Zahl innerhalb des Rechtecks befindet, das durch die Folie gegeben ist.

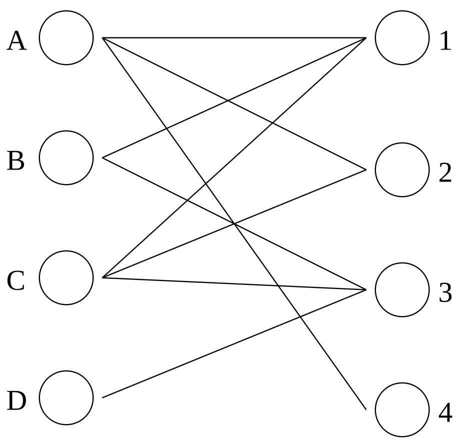


Abbildung A.2: *Äquivalentes Matchingproblem auf bipartitem Graphen. 4 Kanten sind auszuwählen, so dass jedem Buchstaben genau eine Zahl zugeordnet ist und umgekehrt.*

Die folgende Musterlösung baut den in Abb. A.2 gegebenen Graphen auf, und überprüft dann für jede Kante (u, v), ob nach dem Entfernen dieser Kante der verbleibende Graph immer noch ein maximales Matching enthält. Falls nicht, so ist die Kante (u, v) auf jeden Fall in einem maximalen Matching enthalten, und die Zahl v ist eindeutig der Folie u zuzuordnen.

```
#include "stdio.h"
#include "assert.h"

#define MAXSLIDES 32

FILE *inp;
int n,l[MAXSLIDES][4],c[MAXSLIDES][2];
int caseno=1;
int edge[MAXSLIDES][MAXSLIDES],e[2*MAXSLIDES][2*MAXSLIDES];
int reach[2*MAXSLIDES],back[2*MAXSLIDES];
```

Die Funktion `read_data` liest die Eingabe (Koordinaten der Folien und Zahlen) für die einzelnen Testfälle.

```
int read_data()
{
  int i,j;

  fscanf(inp,"%d",&n);
  for(i=0;i<n;i++)
    for(j=0;j<4;j++) fscanf(inp,"%d",l[i]+j);
  for(i=0;i<n;i++) fscanf(inp,"%d %d",c[i],c[i]+1);
  return n;
}
```

Beim Optimieren wurde hier die innere Schleife entrollt:

```
int read_data(void)
{
  auto int i; auto int j;

  fscanf(inp, "%d", (&n));
  for (i = 0; (i < n); i++)
    {
      fscanf(inp, "%d", (((int *)(l+i))));
      fscanf(inp, "%d", (((int *)(l+i))+1));
      fscanf(inp, "%d", (((int *)(l+i))+2));
      fscanf(inp, "%d", (((int *)(l+i))+3));
    }
  for (i=0; (i<n); i++)
    fscanf(inp,"%d %d",((int *)(c+i)),(((int *)(c+i))+1));
  return n;
}
```

Die Funktion `inside(i,j)` testet, ob sich die Zahl `j` innerhalb der Folie `i` befindet.

```
int inside(int i, int j)
{
  return (l[i][0] <= c[j][0] && c[j][0] <= l[i][1] &&
          l[i][2] <= c[j][1] && c[j][1] <= l[i][3]);
}
#pragma property SideEffectFree(inside)
```

Sie blieb jedoch unverändert, da die mehrfach vorkommenden Ausdrücke (wie `c[j][0]`) nicht teuer genug sind, um eine *CSE Elimination* zu rechtfertigen.

Die Funktion `matching` versucht, in dem durch das Feld `edge` gegebenen Graphen ein maximales Matching zu finden und liefert bei Erfolg `true` zurück.

```
int matching()
{
  int i,j,k,ch,m=0;

  for(i=0;i<2*n+2;i++)
    for(j=0;j<2*n+2;j++)
      e[i][j] = 0;
  for(i=0;i<n;i++)
    {
      e[0][i+2] = 1;
      e[n+i+2][1] = 1;
      for(j=0;j<n;j++)
        e[i+2][j+n+2] = edge[i][j];
    }
  while(1)
    {
      reach[0] = 1;
      for(i=1;i<2*n+2;i++) reach[i] = 0;
      ch = 1;
      while(ch)
        {
          ch = 0;
          for(i=0;i<2*n+2;i++)
            if(reach[i])
              for(j=0;j<2*n+2;j++)
                if(e[i][j] && !reach[j])
                  {
                    ch = 1;
                    reach[j] = 1;
                    back[j] = i;
                    break;
                  }
        }
      if(reach[1])
        {
          m++;
          for(i=1;i>0;i=back[i])
            {
              e[back[i]][i] = 0;
              e[i][back[i]] = 1;
            }
          continue;
        }
      break;
    }
  return (m == n);
}
#pragma property SideEffectFree(matching)
```

Hier waren einige Optimierungen durch *Finite Differencing* und *Expression Simplification* möglich. Die verbesserte Funktion lautet:

```
int matching(void)
{
  auto int _sr43; auto int *_sr35; auto int _sr31; auto int _sr29;
  auto int *_sr16; auto int _sr9; auto int *_sr7; auto int _sr6;
  auto int i; auto int j; auto int ch; auto int m = 0;

  for ((i = 0) , (_sr9 = (2 + (n << 1))); (i < _sr9); i++)
    for (((j = 0) , (_sr7 = ((int *)(e + i)))) , (_sr6 = _sr9);
         (j < _sr6); (_sr7++) , (j++))
      (*_sr7) = 0;

  for (i = 0; (i < n); i++)
    {
      (((int *)e)[2 + i]) = 1;
      (((int *)(e + ((2 + n) + i)))[1]) = 1;
      for ((j = 0) , (_sr16 = ((int *)(edge + i)));
           (j < n); (_sr16++) , (j++))
        (((int *)(e + (2 + i)))[(2 + j) + n]) = ((*_sr16));
    }

  while (1)
    {
      (reach[0]) = 1;
      for ((i = 1) , (_sr29 = (2 + (n << 1))); (i < _sr29); i++)
        (reach[i]) = 0;
      ch = 1;
      while (ch)
        {
          ch = 0;
          for (i = 0; (i < (2 + (n << 1))); i++)
            if ((reach[i]))
              {
                for (((j = 0) , (_sr35 = ((int *)(e + i)))) ,
                       (_sr31 = (2 + (n << 1))); (j < _sr31);
                     (_sr35++) , (j++))
                  {
                    if (((*_sr35)) && (!((reach[j]))))
                      {
                        ch = 1;
                        (reach[j]) = 1;
                        (back[j]) = i;
                        goto __T1;
                      }
                  }
              __T1:;
              }
        }
      if ((reach[1]))
        {
          m++;
```

```
            for ((i = 1) , (_sr43 = ((back[1]))); (i > 0);
                 (_sr43 = ((back[_sr43]))) , (i = ((back[i]))))
              {
                  {  (((int *)(e + _sr43))[i]) = 0; }
                  {  (((int *)(e + i))[_sr43]) = 1; }
              }
            goto __T2;
          }
        goto __T3;
      __T2:;
      }
   __T3:;
    return (int)(m == n);
  }
```

Die Optimierung ist hier leider noch nicht perfekt. Z.B. wurde nicht erkannt, dass `n` eine konstante Variable ist und damit `2+(n<<1)` nur einmal berechnet werden müsste.

Die Funktion `process_data` konstruiert den Graphen und testet anschließend alle Kanten daraufhin, ob sie entfernt werden können und der verbleibende Graph dann immer noch ein maximales Matching enthält. Kanten, die nicht entfernt werden können, sind Teil der Lösung und werden ausgegeben.

```
void process_data()
{
  int i,j,k,f;

  for(i=0;i<n;i++)
    for(j=0;j<n;j++)
      edge[i][j] = inside(i,j);
  assert(matching());
  printf("Heap %d\n",caseno++);
  f = 0;
  for(i=0;i<n;i++)
    for(j=0;j<n;j++)
      if(edge[i][j])
        {
          edge[i][j] = 0;
          if(!matching())
            {
              printf("(%c,%d) ",i+'A',j+1); f = 1;
              for(k=0;k<n;k++)
                {
                  if(k != i) edge[k][j] = 0;
                  if(k != j) edge[i][k] = 0;
                }
            }
          edge[i][j] = 1;
        }
  if(!f) printf("none");
  printf("\n\n");
}
```

In dieser Funktion wurden insgesamt drei Optimierungen durch *Finite Differencing* durchgeführt:

```
void process_data(void)
{
  auto int *_sr51; auto int *_sr48; auto int *_sr46;
  auto int i; auto int j; auto int k; auto int f;

  for (i=0; i<n; i++) {
    for (j=0, _sr46=((int *)(edge+i)); j<n; _sr46++, j++)
      {  (*_sr46) = (inside(i, j)); }
  }
  assert(matching());
  printf("Heap %d\n", (caseno++));
  f = 0;
  for (i = 0; (i < n); i++)
    for ((j = 0) , (_sr51 = ((int *)(edge + i)));
         (j < n); (_sr51++) , (j++))
      if ((*_sr51))
        {
          (*_sr51) = 0;
          if (!(matching()))
            {
              printf("(%c,%d) ", (65 + i), (1 + j));
              f = 1;
              for ((k = 0) , (_sr48 = ((int *)(edge+i)));
                   (k < n); (_sr48++) , (k++))
                {
                  if (k!=i) { ((int *)(edge+k))[j] = 0; }
                  if (k!=j) { (*_sr48) = 0; }
                }
            }
          (*_sr51) = 1;
        }
  if (!(f))  {  printf("none"); }
  printf("\n\n");
  return;
}
```

Das Hauptprogramm, das nicht weiter optimiert werden konnte, lautet schließlich:

```
int main()
{
  inp = fopen("slides.in","r");
  while(read_data())
    process_data();
  fclose(inp);
  return 0;
}
```

Anhang B Aktuelle Regelbasis

B.1 Allgemeine Regeln für die Sprache C

Kommutativität

```
#pragma rule ForAll int c,e   : Constant(c) => e*c === c*e
#pragma rule ForAll int c,e   : Constant(c) => e+c === c+e
#pragma rule ForAll int c,e   : Constant(c) => e|c === c|e
#pragma rule ForAll int c,e   : Constant(c) => e&c === c&e
#pragma rule ForAll int c,e   : Constant(c) => e^c === c^e
```

Assoziativität

```
#pragma rule ForAll int a,b,c : Constant(a) &&& Constant(b) =>
                a*(b*c) === (a*b)*c
#pragma rule ForAll int a,b,c : Constant(a) &&& Constant(b) =>
                a+(b+c) === (a+b)+c
#pragma rule ForAll int a,b,c : Constant(a) &&& Constant(b) =>
                a|(b|c) === (a|b)|c
#pragma rule ForAll int a,b,c : Constant(a) &&& Constant(b) =>
                a&(b&c) === (a&b)&c
#pragma rule ForAll int a,b,c : Constant(a) &&& Constant(b) =>
                a^(b^c) === (a^b)^c
```

Distributivität

```
#pragma rule ForAll int x,y,z : SideEffectFree(x) => x*y+x*z === x*(y+z)
#pragma rule ForAll int x,y,z : SideEffectFree(x) => x*y+z*x === x*(y+z)
...
```

Reflexivität von Relationen

```
#pragma rule ForAll type e : SideEffectFree(e) => e==e === 1
#pragma rule ForAll type e : SideEffectFree(e) => e>=e === 1
#pragma rule ForAll type e : SideEffectFree(e) => e<=e === 1
#pragma rule ForAll type e : SideEffectFree(e) => e!=e === 0
#pragma rule ForAll type e : SideEffectFree(e) => e>e === 0
#pragma rule ForAll type e : SideEffectFree(e) => e<e === 0
```

Vereinfachung von Bedingungen

```
#pragma rule ForAll type x,y : !(x<y)  === x>=y
#pragma rule ForAll type x,y : !(x>y)  === x<=y
#pragma rule ForAll type x,y : !(x<=y) === x>y
#pragma rule ForAll type x,y : !(x>=y) === x<y
#pragma rule ForAll type x,y : !(x==y) === x!=y
#pragma rule ForAll type x,y : !(x!=y) === x==y

#pragma rule ForAll type x,y : (x<y)==1 === x<y
...
#pragma rule ForAll type x,y : (x<y)==0 === x>=y
...
```

Vereinfachung von Gleichungen

```
#pragma rule ForAll int a,b,x : Constant(a) &&& Constant(b)
                => a==(b+x) === (a-b)==x
#pragma rule ForAll int a,b,x : Constant(a) &&& Constant(b)
                => a==(x-b) === (a+b)==x
#pragma rule ForAll int a,x : Constant(a) => a==a*x === 1==x
...
```

Zusammenfassung von Shifts

```
#pragma rule ForAll type e; int c1,c2 :
                Constant(c1) &&& Constant(c2) =>
                (e >> c1) >> c2 === e >> (c1+c2)
#pragma rule ForAll type e; int c1,c2 :
                Constant(c1) &&& Constant(c2) =>
                (e << c1) << c2 === e << (c1+c2)
#pragma rule ForAll int e1,e2,c : SideEffectFree(c) =>
                (e1<<c) & (e2<<c) === (e1&e2) << c
#pragma rule ForAll int e1,e2,c : SideEffectFree(c) =>
                (e1>>c) & (e2>>c) === (e1&e2) >> c
#pragma rule ForAll int e1,e2,c : SideEffectFree(c) =>
                (e1<<c) | (e2<<c) === (e1|e2) << c
#pragma rule ForAll int e1,e2,c : SideEffectFree(c) =>
                (e1>>c) | (e2>>c) === (e1|e2) >> c
```

AND und OR

```
#pragma rule ForAll int x : SideEffectFree(x) => x || x === x
#pragma rule ForAll int x : SideEffectFree(x) => x && x === x
```

Doppelte Negation

```
#pragma rule ForAll int x : !!x === x
#pragma rule ForAll int x : ~~x === x
#pragma rule ForAll int x : -(-x) === x
#pragma rule ForAll int e1,e2 : (-e1)*(-e2) === e1*e2
```

Neutrale Elemente

```
#pragma rule ForAll int n : 0+n === n
#pragma rule ForAll int n : n-0 === n
#pragma rule ForAll int n : 0*n === n,0
#pragma rule ForAll int n : 1*n === n
#pragma rule ForAll int n : n/1 === n
#pragma rule ForAll int n : n%1 === n,0
#pragma rule ForAll int n : 0&n === n,0
#pragma rule ForAll int n : ~0&n === n
#pragma rule ForAll int n : 0|n === n
#pragma rule ForAll int n : 0^n === n
#pragma rule ForAll int n : SideEffectFree(n) => n/n === n?1:(1/0)
#pragma rule ForAll int n : SideEffectFree(n) => n-n === 0
```

Regeln für den ?: Operator

```
#pragma rule ForAll int c : c?1:0 === c
#pragma rule ForAll int c : c?0:1 === !c
```

Distributivität von ?: über Funktionsaufrufe

```
#pragma rule ForAll type f,c,e1,e2 : f(c?e1:e2) === c?f(e1):f(e2)
```

Distributivität von ?: über verschiedene Operatoren

```
#pragma rule ForAll int c; type e1,e2,e3 :
                (c?e1:e2)!=e3 === c?(e1!=e3):(e2!=e3)
#pragma rule ForAll int c; type e1,e2,e3 :
                (c?e1:e2)==e3 === c?(e1==e3):(e2==e3)
#pragma rule ForAll int c; int e1,e2,e3 : e3+(c?e1:e2) === c?(e3+e1):(e3+e2)
#pragma rule ForAll int c; int e1,e2,e3 : e3*(c?e1:e2) === c?(e3*e1):(e3*e2)
#pragma rule ForAll int c; int e1,e2,e3 : (c?e1:e2)-e3 === c?(e1-e3):(e2-e3)
...
```

Multiplikationen mit Zweierpotenzen

```
#pragma rule ForAll int n :          2 * n === n <<  1
#pragma rule ForAll int n :          4 * n === n <<  2
#pragma rule ForAll int n :          8 * n === n <<  3
#pragma rule ForAll int n :         16 * n === n <<  4
#pragma rule ForAll int n :         32 * n === n <<  5
...
#pragma rule ForAll int n : 1073741824 * n === n << 30

#pragma rule ForAll int n :         -2 * n === -(n <<  1)
#pragma rule ForAll int n :         -4 * n === -(n <<  2)
#pragma rule ForAll int n :         -8 * n === -(n <<  3)
#pragma rule ForAll int n :        -16 * n === -(n <<  3)
...
#pragma rule ForAll int n :-1073741824 * n === -(n << 30)
```

B.2 <ctype.h>

```
#pragma property Transparent(isalnum)
#pragma property Transparent(isalpha)
#pragma property Transparent(isdigit)
#pragma property Transparent(iscntrl)
#pragma property Transparent(isupper)
#pragma property Transparent(islower)
...
#pragma property Transparent(toupper)
#pragma property Transparent(tolower)
```

B.3 <stdio.h>

```
#pragma property StateEffectFree(printf)
#pragma property StateEffectFree(fprintf)
```

B.4 <string.h>

```
#pragma property Transparent(strlen)
#pragma property Transparent(memcmp)
#pragma property Transparent(strcmp)
#pragma property Transparent(strncmp)
#pragma property Transparent(memchr)
#pragma property Transparent(strchr)
#pragma property Transparent(strstr)
```

B.5 <math.h>

```
#pragma property Transparent(sin)
#pragma property Transparent(cos)
#pragma property Transparent(tan)
#pragma property Transparent(asin)
#pragma property Transparent(acos)
#pragma property Transparent(atan)
#pragma property Transparent(atan2)
#pragma property Transparent(sinh)
#pragma property Transparent(cosh)
#pragma property Transparent(tanh)
#pragma property Transparent(exp)
#pragma property Transparent(log)
#pragma property Transparent(log10)
#pragma property Transparent(sqrt)

#pragma rule sin(0.0) === 0.0
#pragma rule cos(0.0) === 1.0

#pragma property Transparent(pow)
```

```
#pragma rule ForAll double x : pow(x,1.0) === x
#pragma rule ForAll double x : SideEffectFree(x) => pow(x,0.0) === 1.0
#pragma rule ForAll double x,a,b : SideEffectFree(x) =>
                                   pow(x,a+b) === pow(x,a)*pow(x,b)
```

Anhang C Grammatik der Transformationsregeln

Transformationsregeln haben die Syntax `#pragma rule` *rule_definition*.
Die Grammatik für *rule_definition* lautet:

```
rule_definition : 'ForAll' decl_list ':' equiv_or_impl
                | equiv_or_impl

decl_list       : declaration
                | declaration ';' decl_list

declaration     : TYPE meta_ident_list

equiv_or_impl   : equivalence
                | implication

equivalence     : EXPR '===' EXPR

implication     : condition '=>' equivalence

condition       : atom
                | '!' atom
                | atom '&&' condition

atom            : predicate '(' expr ')'

predicate       : 'Constant'
                | 'C'
                | 'SideEffectFree'
                | 'StateEffectFree'
                | 'IOEffectFree'
                | 'Transparent'
                | 'Monotone'

meta_ident_list : meta_ident
                | meta_ident ',' meta_ident_list
```

```
meta_ident : IDENTIFIER
```

Die drei Nichtterminale `IDENTIFIER`, `EXPR` und `TYPE` sind in der Grammatik groß geschrieben, da sie Konstrukte darstellen, die bereits in der Grammatik von C definiert sind.

IDENTIFIER steht für einen beliebigen Variablennamen, d.h. einen alphanumerischen String. `EXPR` steht für einen C-Ausdruck und `TYPE` für einen C-Datentyp (int, float, ...).

Anhang D Schlüsselworte der Operationen

Die Kosten elementarer Operationen können mit dem Pragma `cost` angegeben werden. Die Operationen werden dabei mit den folgenden Schlüsselworten spezifiziert.

Schlüsselwort	Operation
indirect	Pointer de-reference („*“ operator)
inegate	Integer negation
fnegate	Floating negation
not	Logical complement („!“ operator)
cast	Type cast
complement	Integer bitwise complement („~“ operator)
ipost_incr	Integer post increment
ipost_decr	Integer post decrement
ipre_incr	Integer pre increment
ipre_decr	Integer pre decrement
fpost_incr	Floating post increment
fpost_decr	Floating post decrement
fpre_incr	Floating pre increment
fpre_decr	Floating pre decrement
ppost_incr	Pointer post increment
ppost_decr	Pointer post decrement
ppre_incr	Pointer pre increment
ppre_decr	Pointer pre decrement
iadd	Integer addition
isubtract	Integer subtraction
imultiply	Integer multiplication
idivide	Integer division

Schlüsselwort	Operation
ieq	Integer equality
ine	Integer inequality
igt	Integer greater than
ilt	Integer less than
ige	Integer greater than or equal
ile	Integer less than or equal
iassign	Integer assignment
fadd	Floating addition
fsubtract	Floating subtraction
fmultiply	Floating multiplication
fdivide	Floating division
feq	Floating equality
fne	Floating inequality
fgt	Floating greater than
flt	Floating less than
fge	Floating greater than or equal
fle	Floating less than or equal
fassign	Floating assignment
padd	Pointer addition
psubtract	Pointer subtraction
passign	Pointer assignment
remainder	(„%“ operator)
pdiff	Pointer difference
peq	Pointer equality
pne	Pointer inequality
pgt	Pointer greater than
plt	Pointer less than
pge	Pointer greater than or equal
ple	Pointer less than or equal
iadd_assign	Integer add assign operator
isubtract_assign	Integer subtract assign operator
imultiply_assign	Integer multiply assign operator
idivide_assign	Integer divide assign operator
remainder_assign	Remainder assign operator
fadd_assign	Floating add assign operator
fsubtract_assign	Floating subtract assign operator
fmultiply_assign	Floating multiply assign operator
fdivide_assign	Floating divide assign operator

<table>
<tr><th>Schlüsselwort</th><th>Operation</th></tr>
<tr><td>padd_assign</td><td>Pointer add assign operator</td></tr>
<tr><td>psubtract_assign</td><td>Pointer subtract assign operator</td></tr>
<tr><td>shiftl_assign</td><td>Left shift assign operator</td></tr>
<tr><td>shiftr_assign</td><td>Right shift assign operator</td></tr>
<tr><td>and_assign</td><td>Bitwise and assign operator</td></tr>
<tr><td>or_assign</td><td>Bitwise or assign operator</td></tr>
<tr><td>xor_assign</td><td>Exclusive or assign operator</td></tr>
<tr><td>subscript</td><td>C subscripting operation („[]“ operator)</td></tr>
<tr><td>field</td><td>„.“ operator when result is the address of the field</td></tr>
<tr><td>value_field</td><td>„.“ operator when result is the value of the field</td></tr>
<tr><td>shiftl</td><td>Left shift („<<“ operator)</td></tr>
<tr><td>shiftr</td><td>Right shift („>>“ operator)</td></tr>
<tr><td>and</td><td>Bitwise and („&“ operator)</td></tr>
<tr><td>or</td><td>Bitwise or („|“ operator)</td></tr>
<tr><td>xor</td><td>Exclusive or („^“ operator)</td></tr>
<tr><td>comma</td><td>The comma operator</td></tr>
<tr><td>land</td><td>Logical intersection („&&“ operator)</td></tr>
<tr><td>lor</td><td>Logical union („||“ operator)</td></tr>
<tr><td>question</td><td>Conditional expression („?:“ operator)</td></tr>
<tr><td>call</td><td>A call of a function or a subroutine</td></tr>
</table>

Literaturverzeichnis

[App98] A.W. Appel. *Modern Compiler Implementation in C.* Cambridge University Press, 1998.

[ASU86] A. Aho, R. Sethi und J. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley Series in Computer Science, 1986.

[BBB+85] F. Bauer, R. Berghammer, M. Broy, W. Dosch, F. Geiselbrechtinger, R. Gnatz, E. Hangel, W. Hesse, B. Krieg-Brückner, A. Laut, T. Matzner, W. Möller, F. Nickel, H. Partsch, P. Pepper, K. Samelson, M. Wirsing und H. Wössner. *The Munich Project CIP. Volume I: The Wide Spectrum Language CIP-L (Lecture Notes in Computer Science, Volume 183).* Springer Verlag, 1985.

[BEH+87] F. Bauer, H. Ehler, A. Horsch, B. Möller, H. Partsch, O. Paukner und P. Pepper. *The Munich Project CIP. Volume II: The Transformation System (Lecture Notes in Computer Science, Volume 292).* Springer Verlag, 1987.

[CLR90] T. Cormen, C. Leiserson und R. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[Cof84] J.W. Coffron. *Programmierung des 8086/8088.* SYBEX-Verlag GmbH, Düsseldorf, 1984.

[Det97] M. Dettinger. *ULTRA: Ein interaktives Transformationssystem für funktionale Programme.* Diplomarbeit, Universität Ulm, 1997.

[DFG+94] K. Didrich, A. Fett, C. Gerke, W. Grieskamp und P. Pepper. *OPAL: Design and Implementation of an Algebraic Programming Language.* In: Jürg Gutknecht (Herausgeber): *Programming Languages and System Architectures, International Conference, Zurich, Switzerland, March 1994*, Band 782 der Reihe *Lecture Notes in Computer Science*, Seiten 228–244. Springer, 1994.

[DJ90] N. Dershowitz und J. Jouannaud. *Rewrite Systems*, Band B der Reihe *Handbook of Theoretical Computer Science*, chapter 6, Seiten 243 – 320. Elsevier, 1990.

[Fea87] M.S. Feather. *A Survey and Classification of some Program Transformation Approaches and Techniques.* In: L.G.L.T. Meertens (Herausgeber): *Program specification and transformation, Proceedings of the IFIP TC2/WG 2.1 Working Conference on Program Specification and Transformation, Bad Tölz, Germany, 15-17 April, 1986*, Seiten 165–195, 1987.

[GNU95] GNU. *Bison: The YACC-compatible Parser Generator.* Available via World Wide Web on http://www.gnu.org/, 1995.

[Ill88] M. Illsley. *Transforming Imperative Programs.* Doktorarbeit, University of Edinburgh, 1988.

[Jon97] S. Jones. *From the Abacus to Wintermute: Fragments from a History of Computing.* Available via World Wide Web on http://www.culture.com.au/brain_proj/babbage.htm, 1997.

[Muc97] S.S. Muchnik. *Advanced Compiler Design & Implementation.* Morgan Kaufmann Publishers, Inc., 1997.

[Nel91] R.P. Nelson. *80386/486 Handbuch für Programmierer.* Microsoft Press, 1991.

[NNH99] F. Nielson, H.R. Nielson und C. Hankin. *Principles of Program Analysis.* Springer Verlag, 1999.

[Pai90] R. Paige. *Symbolic Finite Differencing.* Technischer Bericht New York University, 1990.

[Par90] H. Partsch. *Specification and Transformation of Programs: A formal approach to software development.* Springer Verlag, 1990.

[Pep84] P. Pepper. *A simple calculus for program transformations (inclusive of induction).* Science of Computer Programming 9, Seiten 221–262, 1984.

[PS77] R. Paige und J. Schwartz. *Expression Continuity and the Formal Differentiation of Algorithms*, 1977.

[PS83] H. Partsch und R. Steinbrüggen. *Program Transformation Systems.* Computing Surveys, Vol. 15, No. 3, Seiten 199–236, 1983.

[Que98] T. Quell. *ULTRA: Typisierung und Programmtransformation.* Diplomarbeit, Universität Ulm, 1998.

[Sca83] L.J. Scanlon. *Die 68'000er: Grundlagen und Programmierung.* AT Verlag Aarau, 1983.

[Sch97] U. Schöning. *Algorithmen - kurz gefasst.* Spektrum Akademischer Verlag, 1997.

[Sch98] J. Schmid. *ULTRA: Nichtdeterminismus und Programmtransformation.* Diplomarbeit, Universität Ulm, 1998.

[Smi90] D. Smith. *KIDS: A Semiautomatic Program Development System.* IEEE Transactions on Software Engineering, Vol. 16, No. 9, 1990.

[vH97] F. von Henke. *Theorie und Praxis des maschinellen Beweisens.* Vorlesungsskript SS97 an der Universität Ulm, 1997.

[WM95] R. Wilhelm und D. Maurer. *Compiler Design.* Addison-Wesley, 1995.

Curriculum Vitae

Mark Steffen Dettinger

28. Dezember 1970
Geboren in Göppingen

September 1981 - Mai 1990
Erich-Kästner-Gymnasium Eislingen

Juli 1990 - September 1991
Bundeswehr

Oktober 1991 - August 1997
Studium der Informatik an der Universität Ulm

29. August 1997
Abschluss des Informatik-Diploms

September 1997 - Februar 2000
Wissenschaftlicher Mitarbeiter in der Abteilung „Compilerbau und Programmiermethodik“ der Universität Ulm