Fachhochschule Köln
Fachbereich für Informatik
Wahlpflichtfach Compiler- und Interpreterbau
Sommersemester 2015
Dozent: Prof. Dr. Erich Ehses

Code-Optimierungsstrategien

Tail Recursion und Last-Call-Optimierung
Inline Expansion
Constant Folding und Copy Propagation
Dead-Code-Elimination

Arno v. Borries, Mtr. Nr. 11085034
Jan Phillip Kretzschmar, Mtr. Nr. 11096203

Inhaltsverzeichnis

1. Code-Optimierung

„In almost every computation a great variety of arrangements for the succession of the processes is possible, and various considerations must influence the selection amongst them for the purposes of a Calculating Engine. One essential object is to choose that arrangement which shall tend to reduce to a minimum the time necessary for completing the calculation."[1]

- Ada Lovelace

Das Optimieren von Programmcode, d.h. die Umwandlung eines Programms in ein semantisch äquivalentes mit weniger Ressourcenverbrauch, ist eine wichtige Funktionalität moderner Compiler. Tatsächlich ist die Entwicklung heutiger Softwaresysteme ohne optimierende Compiler ein schwer vorstellbares Unterfangen. Handoptimierung wird mit der steigenden Komplexität von Prozessor- und Speicherarchitekturen nicht nur technisch immer schwieriger, sie ist auch zeitaufwändig. Beides zwingt zur Konzentration auf besonders erfolgversprechende Optimierungsstrategien. Der optimierende Compiler ist von diesen beiden Problemen erheblich weniger betroffen und kann daher eine viel größere Bandbreite verschiedener Optimierungen einsetzten, auch solche, die trotz ihrer Komplexität nur einen geringen Optimierungseffekt haben. Daher scheint es im wesentlichen unmöglich, ein nicht-triviales Programm von Hand auch nur annähernd so sehr zu optimieren, wie dies die gebräuchlichen Compilerprogramme automatisch tun.

Optimierungen von Programmcode können grundsätzlich in zwei Richtungen vorgenommen werden. Einerseits kann eine Reduktion der Laufzeit eines Programms angestrebt werden, andererseits ein verringerter Speicherbedarf des Programms. Diese beiden Ziele stehen in einem gewissen Widerspruch zueinander, es muss also eine Abwägung getroffen werden, welches für den Einsatz der Software von größerer Bedeutung ist. Der Compiler muss dennoch, unabhängig vom Zweck der Optimierung, für eine im Prinzip unendliche Zahl von verschiedenen Programmen eine korrekte Übersetzung in die Zielsprache liefern.[2] Ein *perfekter* Compiler würde - im Gegensatz zu einem optimierenden – in jedem Falle gleich die Ausgabe des Programms liefern (es gäbe also keine Laufzeit mehr, nur C*ompile Time*). Ein solcher Compiler ist bereits auf theoretischer Ebene unmöglich, da er zum Erreichen dieses Ziels gezwungen wäre, das Halteproblem zu lösen.

1 Menabrea, Luigi „Sketch of the Analytical Engine Invented by Charles Babbage, with Notes upon the Memoir by the Translator“ In: *Bibliothèque Universelle de Genève* **82** , October 1842

2 Vgl. Aho, Alfred et al: *Compilers – Priciples, Techniques & Tools (Second Edition)*, Boston, 2007, S.14

Den offensichtlichen Vorteilen von optimiertem Code stehen im wesentlichen zwei Nachteile gegenüber. Erstens verliert der optimierte Code für gewöhnlich ganz erheblich an Menschenlesbarkeit. Dies ist eher ein Problem bei handoptimiertem Code als bei Compileroptimiertem, da ja der (hoffentlich) gut lesbare Quellcode in der Ausgangssprache (hoffentlich) weiterhin zur Verfügung steht. Zweitens benötigt Optimierung wie bereits erwähnt Zeit und Ressourcen. Dies gilt natürlich auch für optimierende Compiler. Obwohl diese Problematik im Laufe der letzten drei Jahrzehnte erheblich an Brisanz verloren hat, ist es auch heute noch üblich, dass erst am Ende der Entwicklung großer Softwaresysteme alle Optimierungsoptionen des Compilers verwendet werden. Vorher ist der Fokus eher auf einer möglichst schnellen Compilierung zu Testzwecken.

Die üblichen, in Compilern implementierten Optimierungsalgorithmen setzen einen bereits vollständig erstellten abstrakten Syntaxbaum voraus, d.h. sie werden vor oder während der Codegenerierungsphase des Compilers durchgeführt. Allerdings ist es ein ganz eigenes Problem, die optimale Reihenfolge für die Anwendung dieser zahlreichen Algorithmen auf ein gegebenes Programm zu ermitteln, da es verschiedene komplexe Interaktionen zwischen den einzelnen Optimierungsschritten gibt. Die Darstellung einiger dieser Interaktionen soll, neben der detaillierten Dokumentation exemplarischer Techniken, das Ziel dieser Arbeit sein.

Als Grundlage für dieses Projekt im Rahmen des Wahlpflichtfaches Compiler- und Interpreterbau dient den Autoren der zu diesem Zweck bereitgestellte Tiny-C-Compiler. Dieser Compiler ist in Java implementiert und übersetzt einen Teil des Befehlssatzes der Sprache C in Java-Bytecode. Die hier beschriebenen Optimierungsstrategien sind daher, ganz im Sinne dieser Designentscheidung, allesamt Maschinenunabhängig.

2. Tail recursion und Last-Call-Optimierung

2.1 Grundlagen der Tail recursion und Last-Call-Optimierung

Rekursion beschreibt den Selbstaufruf einer Funktion. Generell verbraucht Rekursion viel Platz auf dem Stack, da mit jeder neuen Rekursionsebene ein neuer Stack-Frame auf den Stack geschoben wird. Bei einer Funktion, deren Anweisungen ansonsten wenig Ressourcen verbrauchen, kann dieser Overhead größer werden, als die intendierte Funktionalität.

Verglichen damit kommen iterativ formulierte Funktionen (etwa mittels einer Do-While-Schleife), mit einem einzigen Stack-Frame aus. Darüber hinaus wird hierbei die Anzahl der Sprungbefehle

minimiert. Aus Optimierungs-Sicht erscheint es also im Allgemeinen zweckmäßig, Rekursion so weit wie möglich zu vermeiden. Tatsächlich ist es theoretisch möglich, alle rekursiv formulierten Funktionen ebenso gut iterativ zu programmieren.[3]

Je nach dem zu Grunde gelegten Paradigma der Programmiersprache und des Programmierers oder auch im Interesse der Lesbarkeit des Programmcodes ist es dennoch häufig nötig oder zumindest sinnvoll, Funktionen rekursiv zu formulieren. Solche Erwägungen zu den Vorteilen der rekursiven Formulierung entfallen implizit, sobald der Programmcode in einer Form vorliegt, bei der im Interesse der Maschinennähe auf Menschenlesbarkeit bewusst verzichtet wird. Eine wünschenswerte eingebaute Optimierungsfunktion eines Compilers ist folglich die automatische Umwandlung von rekursiven Funktionsaufrufen in iterative Schleifen bei der Übersetzung in die Zielsprache.

Eine solche Umwandlung ist nicht in jedem Fall gleich leicht zu bewältigen. Häufig beschränken sich die Bemühungen in diese Richtung daher auf die Behandlung von *endrekursiv* formulierten Funktionen. Endrekursion (engl. *tail recursion*) bezeichnet dabei den Selbstaufruf einer Funktion in der letzten Anweisung vor ihrer Return-Anweisung. In diesem Fall werden die zusätzlichen Frames auf dem Stack nicht mehr benötigt, da nach dem Auffinden der Lösung des rekursiven Problems alle auf dem Stack gespeicherten Rücksprünge bis zum ersten Aufruf der Funktion verworfen werden können.

Dieser letzte Funktionsaufruf einer Funktion kann zu einer Schleife aufgelöst werden, indem der aktuelle Stack, ohne neue Rücksprünge und Übergabeparameter auf ihn zu legen, weiterverwendet wird und lediglich die aktuelle Funktion wieder von vorne beginnt. Dies ist äquivalent mit einer *Goto*-Verknüpfung zum Beginn der Funktion.

Eine Generalisierung dieser Endrekursionsoptimierung ist die sogenannte Last-Call-Optimierung. Der Begriff *Last Call* bzw. *Tail Call* bezeichnet einen Funktionsaufruf, der als letzte Anweisung einer beliebigen Funktion vor der Return-Anweisung steht. Da nach dieser Anweisung nichts mehr in der aktuellen Funktion geschieht, kann dieser Funktionsaufruf durch eine Goto-Anweisung ersetzt werden.

Dieses Vorgehen stößt jedoch auf Probleme, sobald lokale Variablen zur Parameterübergabe an den letzten Funktionsaufruf genutzt werden, dann ist der Funktionsaufruf nicht mehr die letzte Anweisung der Funktion, da nach diesem Funktionsaufruf noch die lokalen Variablen wieder vom Stack verschwinden müssen. Dies kann jedoch durch eine Abwandlung der Stack-Verwaltung gelöst

3 Siehe Aho et al, S.73.

werden, indem alle lokalen noch existierenden Variablen vor dem Rücksprung entfernt werden. Jedoch ist dies nicht Teil der eigentlichen Last-Call-Optimierung sondern nur ein Problem, welches diese Optimierung unter gewissen Umständen verhindern kann, obwohl es nicht unbedingt sofort ersichtlich ist, da auf Code-Ebene der Funktionsaufruf immer noch die letzte Anweisung der Funktion ist.

Die Last-Call-Optimierung (und damit auch die Endrekursionsoptimierung) lässt sich auf verschiedene Arten implementieren. Zum einen kann dies bereits im abstrakten Syntaxbaum durch neue Knoten oder Transformationen des Baumes gelöst werden. Der Baum wird zum Zeitpunkt der Erstellung dahingehend abgeändert, dass bei einem Last Call anstatt dem Funktionsaufruf-Knoten ein Last-Call-Knoten auf diese Funktion eingefügt wird. Dieser Last-Call-Knoten wird dann bei der Codegenerierung dahingehend aufgelöst, dass eine Goto-Anweisung zum Beginn der aufgerufenen Funktion angelegt wird und die nun nicht mehr vorhandenen Übergabeparameter der aufgerufenen Funktion dennoch verfügbar sind.

2.2 Implementierung der Last-Call-Optimierung im Tiny-C-Compiler

Eine andere Möglichkeit ist ein Visitor, der den gesamten Baum traversiert und die entsprechenden Funktionsaufrufe nur markiert. Diese Variante bietet gegenüber der ersteren die Freiheit, direkt mehrere Optimierungen in einer Traversion durchzuführen. Im Folgenden soll erläutert werden, wie die Last-Call-Optimierung im Tiny-C-Complier auf Grundlage dieser Strategie realisiert ist, indem der abstrakte Syntaxbaum durch den **LastCallVisitor** traversiert wird. Wir betrachten im einzelnen einige zentrale Methoden dieses Visitors.

Trifft der Visitor auf einen Funktionsknoten, setzt er seine Klassenvariable *lastCall* zunächst auf TRUE und beginnt, den Funktions-Anweisungsblock weiter zu bearbeiten. Außerdem wird bei Void-Funktionen die Klassenvariable *isVoid* ebenfalls auf TRUE gesetzt, bei allen Anderen auf FALSE:

```
protected void visitFunction(FunctionEntry f) {
      fkt = f;
      isVoid = (f.getResultType() == VOID_TYPE);
      lastCall = true;
      f.getBlock().accept(this);
}
```

Der LastCallVisitor arbeitet alle Anweisungen der Funktion in umgekehrter Reihenfolge ab, so lange seine Klassenvariable *lastCall* auf TRUE gesetzt ist:

```
public void visit(StatementSequence s) {
      ListIterator<INode> iter = s.stmts.listIterator(s.stmts.size());
      while (lastCall) iter.previous().accept(this);
}
```

Die Klassenvariable *lastCall* bleibt weiterhin gesetzt, wenn die erste der abgearbeiteten Anweisungen (also die letzte der Funktion) ein Return ist. Bei Nicht-void-Funktionen (d.h. wenn die Klassenvariable *isVoid* auf FALSE steht, siehe oben) wird ausserdem der Ausdruck des Return-Statements durch den Visitor evaluiert. Er könnte schließlich weitere Funktionsaufrufe beinhalten:

```
public void visit(ReturnStmt s) {
	lastCall = true;
	if (!isVoid) s.expression.accept(this);
}
```

Trifft der Visitor direkt vor dem Return auf einen Funktionsaufruf, prüft er ob es sich um einen rekursiven Aufruf handelt, und markiert diesen mit Hilfe der Instanzvariable *lastCall* als Last Call. Danach wird die Klassenvariable *lastCall* auf FALSE gesetzt. Auf diese Weise wird sichergestellt, dass nur der letzte rekursive Funktionsaufruf vor dem Return als Last Call markiert wird, da der Rest der Anweisungen davor nicht mehr betrachtet wird:

```
public void visit(FunctionCall s) {
	if (lastCall && s.function.equals(fkt)) {
		s.lastCall = true;
		System.out.println(fkt.getName() + ": last call");
	}
	lastCall = false;
}
```

Trifft der Visitor direkt vor dem Return auf eine If-Anweisung, muss er eine Fallunterscheidung treffen. Möglicherweise befindet sich ein Return im Then-Teil. Unabhängig davon muss aber auf jeden Fall noch geprüft werden, ob sich im evtl. vorhandenen Else-Teil ebenfalls noch ein Return findet. Daher muss der Wert der Klassenvariable *lastCall* für den Else-Teil zwischengespeichert werden. Beide Teile werden dann vom Visitor genauso weiterbehandelt, wie ein Funktionsblock:

```
public void visit(IfStmt s) {
	boolean lastInElse = lastCall;
	s.thenPart.accept(this);
	if (s.elsePart != null) {
		lastCall = lastInElse;
		s.elsePart.accept(this);
	}
	lastCall = false;
}
```

Alle anderen Anweisungen führen sofort zum Setzen der Klassenvariable *lastCall* auf FALSE und damit zum Abbruch der weiteren Suche in diesem Funktionsknoten. Dieses Vorgehen stellt sicher, dass nur Funktionsaufrufe direkt vor dem Return-statement als Last call markiert werden und ist auch eine effektive Methode zur Verkürzung der Laufzeit des Compilers:

```
protected void otherStatement(INode s) {
      lastCall = false;
}
```

Am Ende der Traversion des gesamten abstrakten Syntaxbaumes sind auf diese Weise alle Last Calls für die weitere Bearbeitung korrekt markiert.

3. Inline Expansion

3.1 Grundlagen der Inline expansion

Inline Expansion bzw. *Inlining* bezeichnet das Ersetzen eines Funktionsaufrufs mit dem Anweisungsblock der gerufenen Funktion. Dies eliminiert den Function-Call-Overhead, d.h. das Anlegen des Stack-Frames für den Funkionsaufruf, verbraucht keine Register um Argumente zu übergeben und im Falle eines *Call-by-reference* muss nichts dereferenziert werden. Auf diese Weise verkürzt Inlining die Laufzeit. Andererseits wird das Programm umfangreicher, sobald der Funktionsaufruf mehr als ein mal im Programmcode vorkommt. Es handelt sich also um ein Tauschgeschäft, wobei Speicherverbrauch gegen Ausführungsgeschwindigkeit gehandelt wird. Obwohl es dabei möglich ist, den Speicherverbrauch soweit zu erhöhen, dass der *Instruction Cache* geflutet wird (was wiederum eine erhebliche Verlängerung bei der Laufzeit mit sich bringt), sind die Vorteile bei der Laufzeit für gewöhnlich größer als die Nachteile beim Speicherverbrauch.[4]

Eine Reihe weiterer Optimierungen benötigen einen möglichst weit expandierten Code um ihr Optimierungspotential voll zu entfalten.[5] So wird es etwa wesentlich leichter möglich, die optimale Allokation von Registern zu Variablen zu entscheiden, wenn ein möglichst langes Code-Segment vorliegt. Dadurch wird auch die Dead-Code-Elimination effektiver. Die durch diese Technik ermöglichten und verbesserten Optimierungen sind zahlreich, jedoch wird in dieser Arbeit Inline Expansion nur im Zusammenhang mit der Umwandlung von Last Calls in Schleifen näher betrachtet werden.

3.2 Implementierung von Inline Expanison im Tiny-C-Compiler

Im Folgenden soll gezeigt werden, wie die durch den LastCallVisitor an Hand des Tiny-C-Compilers markierten Funktionen mit Hilfe einiger Methoden des **JavaCodeGenerationVisitor** expandiert werden.

---

4 Insbesondere bei *kleinen* Caches: Siehe Chen et al, „The Effect of Code Expansion Optimizations on Instruction Cache Design“, *IEEE Transactions on Computers*, vol.42, **9**, September 1993 S.1056

5 Siehe Allen, R. und Johnson, S., “Compiling C for vectorization, parallelism, and inline expansion”, *Proceedings of the 1988 ACM Conference on Programming Language Design and Implementation*, June 1988, S.242ff

Der Visitor traversiert den abstrakten Syntaxbaum und wandelt die Knoten in Java-Bytecode-Anweisungen um. Stößt er dabei auf einen Funktionsaufruf-Knoten, erzeugt er zunächst die nötigen Anweisungen, um die Argumente des Aufrufs auf den Stack zu schieben. Danach trifft er eine Fallunterscheidung: Wenn die Instanzvariable lastCall des Knotens auf TRUE steht, der Funktionsaufruf also als Last Call markiert ist, speichert er die gerade behandelten Argumente statt dessen als lokale Variablen in den von der Java Virtual Machine dafür vorgesehenen lokalen Variablenarray des Stack-Frames der aufrufenden Funktion. Danach wird zum Anfang des aufgerufenen Funktionsblocks gesprungen, um die Bytecode-Anweisungen in diesem zu generieren und nacheinander in die aufrufende Funktion zu schreiben. Schließlich wird die Klassenvariable deleteReturn auf TRUE gesetzt. Für den Fall, dass der Funktionsaufruf nicht als Last Call markiert ist, wird statt dessen der Code für das anlegen eines Stack-Frames für eine statische Funktion generiert:

```
public void visit(FunctionCall f) {
      for (INode n : f.arguments) n.accept(this);
      if (f.lastCall) {
            Type[] parameterTypes = f.function.getTypesOfParameters();
            for (int addr = parameterTypes.length-1; addr >= 0; addr--) {
                  Type t = parameterTypes[addr];
                  mv.visitVarInsn((t==INT_TYPE) ? ISTORE : ASTORE, addr);
            }
            mv.visitJumpInsn(GOTO, beginLabel);
            deleteReturn = true;
      }
      else mv.visitMethodInsn(INVOKESTATIC, className, f.name,
f.function.getSignature());
}
```

Trifft der Visitor auf einen Return-Knoten, setzt er zunächst die Klassenvariable deleteReturn auf FALSE. Return-Anweisungen, die Ausdrücke enthalten, werden weiter evaluiert, da sie noch Funktionsaufrufe beinhalten können. Bleibt bei der Evaluation einer solche Return-Anweisung deleteReturn weiterhin FALSE (d.h. der Ausdruck enthielt keinen Funktionsaufruf oder der enthaltene Funktionsaufruf war nicht als endrekursiv markiert), wird die dem Rückgabewerttyp entsprechende Bytecode-Return-Anweisung generiert. Für Void-Funktionen wird nachvollziehbarer Weise die Bytecode-Return-Anweisung für Void-Funktionen generiert. Für den Fall, dass sich die Funktion, zu der die Return-Anweisung gehört, als Last-Call-formuliert herausgestellt hat, also als solche markiert war, wird das Return-Statement einfach überhaupt nicht übersetzt, da es überflüssig geworden ist und damit eingespart werden kann.

```
	public void visit(ReturnStmt s) {
		if (s.expression != null) {
			deleteReturn = false;
			s.expression.accept(this);
			if (!deleteReturn) mv.visitInsn(s.resultType == INT_TYPE ?
IRETURN : ARETURN);
			}
		else mv.visitInsn(RETURN);
	}
```

Nach dieser Traversion des Syntaxbaums ist der Bytecode für das Programm vollständig generiert. Dabei wurden sämtliche Funktionen (*callees*), deren Aufrufe als Last Call markierten war, in die sie aufrufenden Funktionen (*callers*) geinlined. Da dieser Vorgang rekursiv abgelaufen ist, sind damit auch alle rekursiven Schleifen in iterative verwandelt.

4. Constant Folding, Copy Propagation und Dead-Code-Elimination

4.1 Grundlagen von Constant Folding, Copy Propagation und Dead-Code-Elimination

Die Optimierungsstrategie, Programmcode auf Ausdrücke hin zu untersuchen, die zu Konstanten gemacht werden können, wird als *Constant Folding* bezeichnet. Bestandteile konstanter Ausdrücke können Literale sein, so z.B. ein bekannter String oder Integer, oder aber Variablen, deren Wert zur Compile Time bekannt ist. Diese Bestandteile der Ausdrücke werden durch Operatoren miteinander verknüpft. Solche Ausdrücke können aufgelöst und durch ihr Ergebnis (eine Konstante) ersetzt werden. Dieses Vorgehen nennt man *Constant-* bzw *Copy Propagation*. Für sich alleine sind diese Optimierungsstrategien bereits recht mächtig. An einem Ausdruck wie (23*15/5+154*3) kann man sofort sehen, dass sich erheblich Zeit sparen lässt, wenn man ihn nur einmal ausrechnet und durch 531 ersetzt, statt ihn jedes mal zur Laufzeit wieder berechnen zu lassen. Der Nutzen dieser Optimierung geht aber noch weiter. Eine Prüfung, ob eine bekannte Konstante einem bestimmten Wert entspricht oder ob sie sich in einem bestimmten Wertebereich befindet, kann offensichtlich zur Compile-Time durchgeführt werden. Die Anweisung

```
x = 5;
if (x > 3) return a;
else return b;
```

kann zu einem einzigen Return reduziert werden, wobei ein Vergleich und der damit verbundene bedingte Sprung eingespart werden. Der eingesparte Else-Teil der If-Anweisung wäre auch ohne die Optimierung nie erreicht worden, es handelt sich also um Dead Code. Für den Fall, dass das

Programm bereits in *Static-Single-Assignment-Form* vorliegt, können Constant Folding, Copy Propagation und bestimmte Formen von Dead-Code-Elimination gemeinsam durchgeführt werden.[6]

4.2 Implementierung der Optimierungen für den Tiny-C-Compiler

Implementiert wurden diese Optimierungen für den Tiny-C-Compiler durch die Autoren in einem eigenen Visitor, dem **DeadCodeEliminationVisior**. Im Folgenden soll anhand seiner Methoden gezeigt werden, welche Maßnahmen er beim Traversieren des abstrakten Syntaxbaumes unternimmt. Das Prinzip ist das bereits erläuterte: es werden Variablen, sobald sie einen konstanten Wert erhalten, in einer *Hashmap* gespeichert und jedes nachfolgende Auftreten dieser Variablen durch ihren konstanten Wert ersetzt, bis die Variable einen anderen Wert erhält. Gleichzeitig werden auch Bedingungen von Schleifen und Verzweigungen auf konstante Ausdrücke überprüft und unerreichbarer Code wird gelöscht.

Zunächst ein paar Bemerkungen zu den Klassenvariablen:

reachedReturn spezifiziert, ob der aktuelle Funktionsblock sein Ende (d.h. seine Return-Anweisung) erreicht hat. Aller Code danach wird eliminiert.

isChanged zeigt an, ob der aktuelle Knoten (in einer Bedingung oder Zuweisung) konstant ist oder verändert wurde.

changedVal beinhaltet den Knoten, der den aktuellen Knoten ersetzen soll.

returnNeeded spezifiziert, ob eine Return-Anweisung oder das Ergebnis einer Berechnung benötigt wird. Diese Information wird bei allen Operationen verwendet, um sicherzustellen, dass sie nur für Kontrollstrukturen und Zuweisungen ausgeführt werden.

Die Hashmap *constants* enthält alle aktuell bekannten Variablen, die tatsächlich Konstanten sind.

```
public class DeadCodeEliminationVisitor implements IVisitor {
      private boolean reachedReturn = false;
      private boolean isChanged = false;
      private INode changedVal;
      private boolean returnNeeded = false;
      private HashMap<String, INode> constants;
      …
}
```

4.2.1 Funktionen und Operationen

Die Klassenvariablen werden wie gezeigt initialisiert. Wir werfen als nächstes einen Blick auf die Methoden zur Behandlung von Funktionen und Operationen.

---

6 Siehe Wegman, Mark N; Zadeck, F. Kenneth, "Constant Propagation with Conditional Branches", *ACM Transactions on Programming Languages and Systems* **13** (2), April 1991, S.208.

Beim Bearbeiten der Wurzel des Syntaxbaumes, d.h. dem Knoten für das Programm als Ganzes, werden nach der Initialisierung der Konstanten alle Funktionen und die globalen Anweisungen abgearbeitet:

```
public void visit(Program p) {
      //Initialize constants:
      constants = new HashMap<String, INode>();
      p.stmts.accept(this);
      for (FunctionEntry f : p.listOfFunctions) {
            isChanged = false;
            //Reset constants:
            constants.clear();
          visitFunction(f);
      }
      isChanged = false;
}
```

Die Methode *visitFunction* ist ein Hilfsfunktion, die nur dazu dient, den Anweisungsblock einer Funktion im weiteren Verlauf abzuarbeiten. Die Parameter der Funktion können an dieser Stelle nicht der HashMap zugefügt werden, da sie keine Konstanten sind.

```
protected void visitFunction(FunctionEntry f) {
      reachedReturn = false;
      //Run through StatementSequence:
      f.getBlock().accept(this);
      isChanged = false;
}
```

Trifft der Visitor auf einen Funktionsaufruf, werden dessen Argumente weiter evaluiert:

```
public void visit(FunctionCall f) {
      for (int i = 0; i < f.arguments.size(); i++) {
            isChanged = false;
            INode arg = f.arguments.get(i);
            returnNeeded = true;
            arg.accept(this);
            if(isChanged) {
                  f.arguments.set(i,changedVal);
            }
      }
      isChanged = false;
}

public void visit(CallStmt f) {
      f.functionCall.accept(this);
}
```

Beim Bearbeiten einer Operation mit zwei Operanden (*BinOp*-Knoten) trifft der Visitor zunächst eine Unterscheidung nach dem Typ der Operation (*INT_TYPE* oder *STRING_TYPE*) . In beiden Fällen wird dann weiter geprüft, ob beide Operanden konstant sind. In diesem Fall wird die Operation aufgelöst und die dadurch erzeugte neue Konstante in die Klassenvariable *changedVal* eingetragen. Die Klassenvariable *isConstant* wird auf TRUE gesetzt. Aus Platzgründen (es gibt

einige verschiedene Operatoren) muss an dieser Stelle auf eine Darstellung des Quellcodes der Methode verzichtet werden.

Trifft der DeadCodeEliminationVisitor auf eine Operation mit nur einem Operanden (UnOp-Knoten), wird ähnlich verfahren. Ist der Operand konstant, wird sein Wert in *constantVal* gespeichert und *isChanged* gesetzt.

```
	public void visit(UnOp op) {
	if(returnNeeded) {
		isChanged = false;
			op.accept(this);
			if(isChanged) {
				//SOLVE UNOP If operand is constant
				if(op.operator == CHS_OP){
					changedVal = new IntLiteral(-((IntLiteral)
op.operand).number);
				} else if(op.operator == NOT_OP){
					changedVal = new IntLiteral(((IntLiteral)
op.operand).number == 0);
				} else if(op.operator == SCONV_OP){
					Integer i = new Integer(((IntLiteral)
op.operand).number);
					changedVal = new StringLiteral(i.toString());
				} else {
					//unknown BinOp.operator
					isChanged = false;
				}
			}
		} else {
			isChanged = true;
			changedVal = new NullStatement();
		}
	}
```

4.2.2 Kontrollstrukturen

Wir wenden uns nun der Behandlung der verschiedenen Kontrollstrukturen zu. Beim Besuchen eines *StatementSequence*-Knotens wird die gesamte Sequenz traversiert und alle Elemente hinter der Return-Anweisung entfernt.

```
	public void visit(StatementSequence b) {
	int iPos = 0;
	while(iPos < b.stmts.size() && !reachedReturn) {
		INode stmt = b.stmts.get(iPos);
		returnNeeded = false;
		isChanged = false;
		stmt.accept(this);
		if(isChanged) {
			b.stmts.set(iPos, changedVal);
			iPos--;
		}
		iPos++;
	}
```

```
        //Remove all statements behind a return:
        while(iPos < b.stmts.size()) {
              b.stmts.remove(iPos);
              iPos++;
        }
        isChanged = false;
    }
```

Die Behandlung der If-Anweisung durch den Visitor ist aus drei Gründen wichtig:

1. Ist eine Bedingung eine Konstante, wird die If-Anweisung entfernt und nur der lebendige Pfad verwendet.
2. Beide Teile des If-Else-Anweisungsblocks müssen darauf geprüft werden, ob in ihnen eine Return-Anweisung vorkommt. Wenn ja, wird *reachedReturn* gesetzt.
3. Es muss geprüft werden, ob beide Teile der Anweisung eine Variable auf den gleichen (konstanten) Wert setzten. Wenn nicht, muss diese Variable aus der HashMap *constants* gelöscht werden.

Auf die Darstellung des Quellcodes der visit-Methode für die If-Anweisung wird – wiederum aus Platzgründen – ebenso verzichtet wie auf die entsprechenden Methoden für die *While*- und die von den Autoren dem Compiler hinzugefügte *For-Schleife*. In diesen beiden Fällen ist vor allem die Prüfung relevant, ob die Schleife überhaupt ausgeführt wird und wenn ja, ob die Anweisungen hinter der Schleife jemals erreicht werden.

Beim Besuch eines Return-Statements werden etwaige Parameter des Ausdrucks weiter evaluiert und *reachedReturn* auf TRUE gesetzt:

```
    public void visit(ReturnStmt s) {
          if (s.expression != null) {
                returnNeeded = true;
                s.expression.accept(this);
                if(isChanged) {
                      s.expression = changedVal;
                      isChanged = false;
                }
          }
          reachedReturn = true;
    }
```

Auch in einer *Print*-Anweisung können sich Ausdrücke verbergen, die evaluiert werden müssen:

```
     public void visit(PrintStmt s) {
          for (int i = 0; i < s.expressions.size(); i++) {
                returnNeeded = true;
                s.expressions.get(i).accept(this);
                if(isChanged) s.expressions.set(i, changedVal);
          }
          isChanged = false;
    }
```

4.2.3 Variablen & Literale

Wie bereits beim Bearbeiten der Wurzel des Syntaxbaumes zu erkennen ist, wird für jede Funktion die Hashmap *constants* entleert. In dieser Hashmap sind alle Variablen enthalten, die aktuell tatsächlich einen konstanten Wert besitzen. Variablen werden im Syntaxbaum durch Knoten vom Typ *VarRef* repräsentiert. Die Knoten vom Typ *VarExpr* enthalten dabei immer eine Referenz auf andere Variablen und reichen ihren Aufruf somit nur an ihre referenzierte Variable weiter. Wird eine *VarRef* in der Hashmap gefunden, wird ihr Wert in *changedVal* geschrieben, so dass die übergeordnete Operation weiß, dass eine Konstante vorhanden ist.

```
	public void visit(VarRef v) {
		if(returnNeeded) {
			INode varTMP = constants.get(v.name);
			if(varTMP != null) {
				isChanged = true;
				changedVal = varTMP;
			} else {
				isChanged = false;
			}
		} else {
			isChanged = true;
			changedVal = new NullStatement();
		}
	}
```

Literale werden immer in *changedVal* geschrieben, um die übergeordente Operation ggf. auflösen zu können. Da ein *ReadNode* eine Benutzereingabe ist, ist sein Wert immer variabel, es wird somit lediglich der Aufforderungsstring besucht und die Anweisung durch ein NullStatement ersetzt, falls das Ergebnis der Benutzereingabe ungenutzt bleibt.

```
	public void visit(IntLiteral lit) {
		isChanged = true;
		if(returnNeeded) changedVal = lit;
		else changedVal = new NullStatement();
	}

	public void visit(StringLiteral lit) {
		isChanged = true;
		if(returnNeeded) changedVal = lit;
		else changedVal = new NullStatement();
	}

	public void visit(ReadNode s) {
		if(returnNeeded) {
			s.prompt.accept(this);
			isChanged = false;
		} else {
			isChanged = true;
			changedVal = new NullStatement();
		}
	}
```

Da Variablen einen Wert in einem *AssignStmt* erhalten, wird die Visit-Methode für diese Knoten genutzt, um Variablen in der Hashmap *constants* zu verwalten. Dazu wird zunächst die rechte Seite einer Zuweisung ausgewertet. Wird für diese ein konstanter Wert zurückgegeben, kann dieser Wert in der Hashmap für die Variable gespeichert werden und die Zuweisung gelöscht werden.

```
	public void visit(AssignStmt s) {
		isChanged = false;
		returnNeeded = true;
		s.rightHandSide.accept(this);
		if(isChanged) {
			if(changedVal instanceof IntLiteral || changedVal instanceof
StringLiteral) {
			//Variable can be replaced with a constant!

	constants.put(((VarRef)s.leftHandSide.reference).descriptor.getName(),
changedVal);
			//Remove AssignStmt
			isChanged = true;
			changedVal = new NullStatement();
		} else if(changedVal instanceof NullStatement){
			//ERROR occured; remove variable from constants
			isChanged = false;

	constants.remove(((VarRef)s.leftHandSide.reference).descriptor.getName());
		} else {
			//Something else got replaced:
			s.rightHandSide = changedVal;
			isChanged = false;
			//Constant may be a variable now!

	constants.remove(((VarRef)s.leftHandSide.reference).descriptor.getName());
		}
	} else {
		//Constant may be a variable now!
		isChanged = false;

	constants.remove(((VarRef)s.leftHandSide.reference).descriptor.getName());
	}
    }
```

4.2.4 Hilfsfunktionen

Es wurde eine Funktion *copy* implementiert, mit der Kopien von Knoten erzeugt werden können. Diese wird genutzt, um die Hashmap in der If-Anweisung zu duplizieren. Sie kann auch genutzt werden, um eine Kopie der StatementSequence einer Funktion beim Inlining zu erstellen. Zusätzlich wurde noch eine Funktion *count* erstellt, die für einen (Teil-)Baum die Anzahl von Anweisungen zählt. Dies sollte ebenfalls im Inlining genutzt werden, um zu entscheiden, ob für einen Funktionsaufruf stattdessen Inlining verwendet werden soll.

```
public interface INode {
      void accept(IVisitor v);
      INode copy();
      int count();
}
```

Die letztendliche Ausführung liegt in der Klasse **parser/Compile**. Hier wird die Methode *optimization* definiert und ausgeführt. In ihr wird ein neuer DeadCodeEliminationVisitor erzeugt und der abstrakte Syntaxbaum des Programms von diesem traversiert.

```
private static void optimization(Program thisProgram) {
    out.println("Optimization of Syntax Tree");
    thisProgram.accept(new DeadCodeEliminationVisitor());
}
```

Abschließend wird noch diese Methode in der Funktion **compile** aufgerufen bevor der Java-Byte-Code erzeugt wird.

5. Ausblick

Der Tiny-C-Compiler kann aufgrund seiner Zielsprache und seiner vorwiegend didaktischen Intention natürlich nicht den Vergleich mit Compilern aufnehmen, die tatsächlich auf die Erzeugung von optimiertem Assembler-Code ausgelegt sind. Wir begnügen uns an dieser Stelle nur mit einem kurzen Blick auf die implementierten Optimierungen. Constant Folding und Copy Propagation sind bei heutigen Compilern Standard. Die Umwandlung von Endrekursion in iterative Schleifen ist weniger weit verbreitet. Java z.B. unterstützt automatisches Inlining von Funktionen zunächst einmal überhaupt nicht. Dies ist eine Designentscheidung u.a. um im Falle einer *Exception* einen kompletten, menschenlesbaren *Function-Stack* ausgeben zu können. Inzwischen, d.h. mit der Verbreitung der *Just-In-Time-Compilierung* von Java-Bytecode, löst sich dieser Konsens zunehmend auf. Ähnlich wird bei Python argumentiert: Rekursion sollte nicht wegoptimiert werden, sondern Programmierer sollten ihren Code iterativ schreiben. Der weit verbreitete gcc C-Compiler verwandelt Endrekursion automatisch in Iteration, die C-Compiler icc, suncc und msvc dagegen nicht. Einige funktionale Sprachen verlangen von Compilern explizit diese Umwandlung.

Wir haben in dieser Arbeit gezeigt, wie die Markierung endrekursiver Funktionen zu ihrer Umwandlung in iterative Schleifen mittels Inline expansion ermöglicht. Ebenso beruht unsere Strategie zur Eliminierung von totem Code auf Constant Folding und Constant Propagation. Die Interdependenzen hören an dieser Stelle natürlich nicht auf. Als nächster Schritt würde z.B. in Frage kommen, sich die verschiedenen Schleifen im Code näher anzuschauen. Die Techniken auf diesem

Gebiet sind erwartungsgemäß zahlreich, man könnte etwa mit *Loop Fusion* beginnen. Zwei Schleifen im gleichen Block, die eine gleiche Anzahl Iterationen haben, und deren Daten nicht voneinander abhängen, können zu einer Schleife zusammengefasst werden, was die Hälfte der Sprungbefehle einspart. Natürlich ist es sofort einsichtig, dass es, hat man diese Optimierung implementiert, sinnvoll ist, Schleifen auf die gleiche Anzahl von Iterationen „zurechtzustutzen“, indem man einige Iterationen vor der Schleife separat ausführt (*Loop Peeling*). Um den gesamten Overhead einer Schleife zu eliminieren, kann sie natürlich auch in eine längere Folge identischer Anweisungsblöcke zerlegt werden, was sich bei wenigen Iterationen anbietet (*Loop Unrolling*). Man kann auch bestimmte Berechnungen, die in der Orginal-Schleife in jeder Iteration gemacht wurden, stattdessen bereits vor der Schleife ausführen, was die Laufzeit erheblich verbessern kann (*Loop Invariant Code Motion*). Allerdings wird dann auch eine Umwandlung der rechenzeitintensiveren *For*- und *While*- in schnellere *Do/While*-Schleifen nötig (*Loop Inversion*).

Wann würden diese Schleifenoptimierungen gemacht werden? Je größer der Codeblock, der untersucht wird, desto wahrscheinlicher, dass man Optimierungsmöglichkeiten vorfindet. Dies würde dafür sprechen, die Schleifen nach dem Inlining zu behandeln. Entrollte Schleifen, die in Wirklichkeit toter Code sind oder solche, die überkompliziert formuliert sind, werden danach mit Hilfe des Constant Folding wegoptimiert bzw. vereinfacht.

Wie sich zeigt, bietet fast jede zusätzlich implementierte Optimierung neue Ansatzpunkte für weitere Optimierungen. Noch schnellerer Zielcode wird dabei jedes mal mit noch langsamerer Compilierung erkauft. Darüber hinaus werden die Compiler selbst auf diese Weise mit der Zeit zu immer monströseren Programmen. Als Werkzeuge für die Softwareentwicklung sind die modernen, immer besser optimierenden Complier dennoch unverzichtbar.

6. Verwendete Literatur

- Aho, Alfred et al: *Compilers – Priciples, Techniques & Tools (Second Edition)*, Boston, 2007.
- Allen, R. und Johnson,S., “Compiling C for vectorization, parallelism, and inline expansion”, *Proceedings of the 1988 ACM Conference on Programming Language Design and Implementation*, June 1988, S.241-249.
- Chen et al, „The Effect of Code Expansion Optimizations on Instruction Cache Design“, *IEEE Transactions on Computers*, vol.42, **9**, September 1993 S.1045-1057.
- Menabrea, Luigi „Sketch of the Analytical Engine Invented by Charles Babbage, with Notes upon the Memoir by the Translator“ In: *Bibliothèque Universelle de Genève* **82** , October 1842.
- Wegman, Mark N; Zadeck, F. Kenneth, "Constant Propagation with Conditional Branches", *ACM Transactions on Programming Languages and Systems* **13** (2), April 1991, S.181–210.