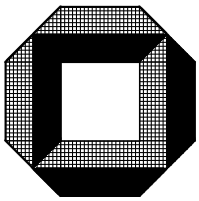

Universität Karlsruhe (TH)
Institut für
Programmstrukturen
und Datenorganisation
Lehrstuhl Professor Goos

Erweiterung eines statischen Übersetzers zu einem Laufzeitübersetzungssystem

Marcus Denker

Diplomarbeit

Verantwortlicher Betreuer: Prof. Dr. Gerhard Goos
Betreuender Mitarbeiter: Dipl.-Inform. Florian Liekweg

März 2004

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

Karlsruhe, den 22.03.2004

Marcus Denker

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Aufgabenstellung

Aufgabe der Diplomarbeit ist die Entwicklung eines Laufzeitübersetzers aus einem existierenden, statischem Übersetzer.

Dazu ist basierend auf der Zwischensprache des existierenden Systems eine Datenbasis zu entwickeln, die für die Laufzeitübersetzung geeignet ist.

An diese Datenbasis ist ein *Front-End* und ein *Back-End* anzuschliessen (oder existierende Module sind anzupassen). Das System soll den vollständigen Entwicklungszyklus unterstützen, vom Aufbau und Modifikation der Datenbasis durch das Einlesen von Quelltext und die Verwendung von bereits gesammelten Laufzeitdaten über die Durchführung von Optimierungen bis zur Generierung von Zielcode.

Das resultierende System soll die Funktionalität des existierenden Systems beibehalten, und die Integration weiterer Analysen ermöglichen, wie weiterer statischer Analysen als auch Laufzeitoptimierungen.

1.2 Motivation

Laufzeitübersetzer erlauben Optimierungen unter Verwendung von Informationen, die erst zur Laufzeit des Programms verfügbar sind.

Der Übersetzer muss das als Quelltext vorliegende Programm in einem ersten Schritt übersetzten. Nun werden beim Programmlauf Informationen (z. B. über die an einem Methodenaufruf vorkommenden Klassen) gesammelt.

Unter Ausnutzung dieser Informationen kann nun der Code besser optimiert werden.

Dieser Prozess kann viele Male wiederholt werden: Auch der nun generierte Code kann in einem weiteren Schritt optimiert werden. Also reicht eine Neuübersetzung aus dem Quellcode nicht aus, stattdessen muss der Übersetzer die Ergebnisse des vorherigen Laufes in einer Datenbasis speichern und in den nächsten Übersetzerlauf einbeziehen.

1.3 Zielkriterien

Aus der Aufgabenstellung ergeben sich folgende Zielkriterien für diese Arbeit:

1. Vollständiger Entwicklungszyklus unterstützt
2. Der Übersetzer erlaubt weitreichende Optimierungen
3. Integration in ein vorhandenes System

1.3.1 Vollständiger Entwicklungszyklus unterstützt

Der Übersetzer soll später vollständig transparent in das System Integriert werden. Das System soll sowohl neuen (als Quellcode eingegebenen) Code übersetzen als auch vorhandenen Code unter Ausnutzung von zusätzlichen Informationen weiter optimieren können.

1.3.2 Der Übersetzer erlaubt Optimierungen

Grund für die wiederholte Übersetzung ist die Möglichkeit, weitere Optimierungen durchzuführen. Der Übersetzer soll daher eine Zwischendarstellung bereitstellen, auf der diese Optimierungen durchgeführt werden können.

1.3.3 Integration in ein vorhandenes System

Der Entwurf soll für ein bestehenden System realisiert werden. Für den Test des Übersetzers ist dabei entscheidend, dass die Ausgabe des Übersetzers auch im System verwendbar ist: Die übersetzten Methoden sollen zu Testzwecken ausführbar sein.

1.4 Überblick über die Arbeit

In Kapitel 2 folgt der Entwurf eines Übersetzers, der die in Abschnitt 1.3 definierten Anforderungen erfüllt.

Das Kapitel 3 beschreibt einen Teil des Entwurfs (den SSA Abbau) genauer.

Die Umsetzung des Entwurfs ist dann Thema des Kapitels 5.

Die Ergebnisse und einige vergleichende Messungen stellt das Kapitel 6 vor.

Das Kapitel 7 analysiert dann die erzielten Ergebnisse anhand der Anforderungen und gibt einen Ausblick auf zukünftige Entwicklungen.

Kapitel 2

Entwurf

2.1 Überblick

Im folgenden Kapitel werden wir einen Übersetzer entwerfen, der den Kriterien aus Kapitel 1.3 entspricht. Dazu betrachten wir die Funktionsweise klassischer Übersetzer und leiten daraus einen Ansatz für den Entwurf her. Dieser wird dann im zweiten Teil des Kapitels ausführlich erläutert.

2.2 Funktionsweise klassischer Übersetzer

Die Aufgabe der Übersetzung wird zur Reduktion der Komplexität in mehrerer Phasen aufgeteilt. In den meisten Übersetzern findet sich folgende Aufteilung (siehe [GW84] und [ASU86]).

1. Syntaktische Analyse
    - Symbolentschlüsselung
    - Zerteilung
2. Semantische Analyse
3. Zwischencode Erzeugung
4. Zwischencode Optimierung (globale Optimierungen)
5. Maschinencode Erzeugung

6. Maschinencode Optimierung (lokale Optimierungen)

Die syntaktische Analyse liest die als Text vorliegende Eingabe. Als Erstes wird der Strom von einzelnen Zeichen in einen Strom von Symbolen transformiert (Symbolentschlüsselung). Anschließend werden diese Symbole durch den Zerteiler nach der Grammatik in einen Syntaxbaum (AST, *Abstract Syntax Tree*) überführt.

Der Syntaxbaum muss nun semantisch analysiert werden: Hier wird die Namensanalyse und bei Sprachen mit statischer Typisierung auch eine Typanalyse durchgeführt. Ergebnis ist ein attributierter Syntaxbaum.

Die nächste Phase ist die Erstellung eines Zwischencodes. Auf diesem kann der Übersetzer nun Optimierungen durchführen.

Der Zwischencode wird dann in Maschinencode übersetzt. Dieser wird lokal optimiert (z.B. mittels Schlüssellochoptimierung).

2.2.1 Bytecode-orientierte Sprachen

In Bytecode-orientierten objektorientieren Sprachen (z. B. Java oder Smalltalk) ist der Übersetzer einfacher aufgebaut: Die Zielsprache, der *Bytecode*, ist semantisch der Quellsprache sehr ähnlich. Es werden nur sehr einfache Optimierungen durchgeführt.

Der Ablauf ist folgender:

1. Syntaktische Analyse
2. Semantische Analyse
3. Erzeugung eines stackorientierten Bytecode

In diesem Fall ist also keine Zwischensprache nötig: Man generiert den Bytecode direkt aus dem unoptimierten, attributierten AST.

2.2.2 Bytecode als Zwischensprache

Wenn man diesen Aufbau mit dem eines klassischen Übersetzers vergleicht, so sieht man, dass die Schritte exakt den Schritten 1-3 eines klassischen Übersetzers entsprechen. Man kann den Bytecode also nicht nur als Zielsprache verstehen (ausgeführt von einer virtuellen Maschine), sondern auch als Zwischensprache.

Die Sichtweise des Bytecode als Zwischensprache ist auch naheliegend, da modernen virtuelle Maschinen den Bytecode in Maschinencode übersetzen, also die Schritte 4-6 implementieren.

2.3 Wahl einer Datenbasis: Bytecode

Unser System soll eine wiederholte Übersetzung erlauben. Dabei sollen die Ergebnisse der Übersetzungen in einer Datenbasis gespeichert und für die nächste Übersetzung wiederverwendet werden.

Wenn wir uns den Aufbau eines Bytecode-orientierten Systems ansehen, so findet sich eine geeignete Datenbasis im Bytecode selbst: Er wird zwischen den Übersetzungsläufen gespeichert, da das System ihn für die Ausführung benötigt.

Wir werden im Folgenden daher einen Bytecode-nach-Bytecode Übersetzer entwerfen.

2.4 Zwischensprache SSA

Der Bytecode dient als Datenbasis zwischen den Übersetzerläufen. Als Zwischensprache für die Optimierungsphase des Übersetzers ist er aber nicht geeignet. Als Zwischensprache wollen wir eine SSA-Darstellung implementieren.

2.5 Aufbau des Übersetzers

Abbildung 2.1 zeigt den Aufbau eines Bytecode-nach-Bytecode Übersetzers im Überblick: Er liest Bytecodes ein, baut daraus eine Zwischensprache (SSA) auf und erzeugt abschließend, nach Optimierung, wieder Bytecodes.

Die Abbildung zeigt den Übersetzer in einer klassischen Form: Links die Eingabe, die durch eine Abfolge von Transformationen die Ausgabe ergibt. Da Ein-und Ausgabe den gleichen Zwischencode verwenden, kann die Ausgabe in einer weiteren Übersetzung als Eingabe dienen.

Die Abbildung 2.2 zeigt diese Sicht.

Wir wollen nun die einzelnen Phasen des Übersetzers genauer betrachten.

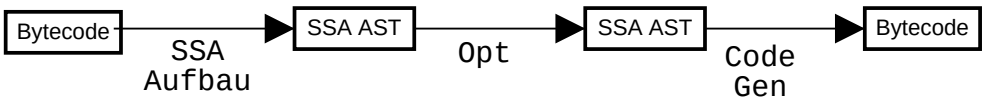

Abbildung 2.1: Struktur des Übersetzers

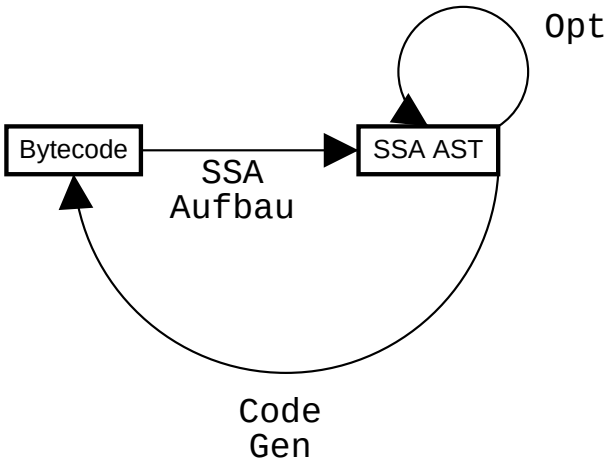

Abbildung 2.2: Der Übersetzer liest seine Ausgabe

2.5.1 Analyse des Bytecodes und Aufbau SSA

Erster Schritt ist die Analyse des Bytecodes. Dazu wird der Bytecode symbolisch ausgeführt und dabei ein Baum aufgebaut. Dieser wird dann in SSA-Darstellung (*Static Single Assigment Form*) transformiert. Eine Einführung in SSA findet sich in [CFR+91].

2.5.2 Optimierungen auf SSA

Die SSA-Darstellung bietet viele Vorteile für die Optimierungsphase. Die Algorithmen sind einfacher und können alle die SSA-Darstellung als Zwischenprache verwenden, was bei anderen Repräsentationen nicht der Fall ist.

2.5.3 SSA Abbau

Die SSA-Darstellung enthält Phi-Funktionen. Für diese kann direkt kein Bytecode erzeugt werden. Daher muss der Übersetzer die SSA-Darstellung in eine nicht-SSA Form transformieren.

Wir werden das verwendete Verfahren für den Abbau der SSA-Darstellung im Kapitel 3 genauer betrachten.

2.5.4 Codegenerierung

Die Aufgabe dieser Phase der Übersetzung ist die Transformation des nun nicht mehr in SSA-Darstellung vorliegenden Baumes in Bytecode, also in eine Reihung von 8bit Zahlen.

Diese Phase lässt sich in zwei Teile gliedern: Erzeugen einer maschinenahen Repäsentation und anschließende Übersetzung dieses symbolischen Codes in Bytecode.

2.6 Zusammenfassung

In diesem Kapitel haben wir, nach einem Überblick über die Funktionsweise klassischer Übersetzer, einen Entwurf vorgestellt. Dieser soll die Anforderungen aus dem ersten Kapitel erfüllen.

Wir haben die einzelnen Phasen des entworfenen Übersetzers beschrieben.

Bevor wir den Entwurf realisieren, wird das nächste Kapitel den SSA Abbau genauer darstellen.

Kapitel 3

SSA Abbau

3.1 Überblick

Dieses Kapitel beschreibt den SSA Abbau. Dazu stellen wir kurz die klassischen Verfahren vor. Dabei gehen wir speziell auf die Probleme dieser Verfahren ein.

Thema des zweiten Teils ist dann das *Phi-Kongruenzverfahren* zum Abbau der SSA-Darstellung nach [SJGS99].

3.2 Klassische Verfahren

Das einfachste Verfahren nach [CFR+91] fügt Zuweisungen in die Vorgängerblöcke ein. Ein einfaches Beispiel zeigt Bild 3.1.

Man sieht, dass die Phi-Funktion entfernt und stattdessen die Zuweisung $a3 := a1$ und $a3 := a2$ in die entsprechenden Vorgängerblöcke eingefügt wurden.

Die eingefügten Kopien können durch Kopienweiterschaltung zu großen Teilen enfernt werden. Dies ist in den meisten aktuellen Implementierungen Teil der Registerzuteilung.

3.2.1 Probleme

In [BCHS98] wird gezeigt, dass dieses Verfahren nicht funktioniert, wenn Code umgeordnet wird. In Abschnitt 3.9 werden wir solche zwei Beispiele

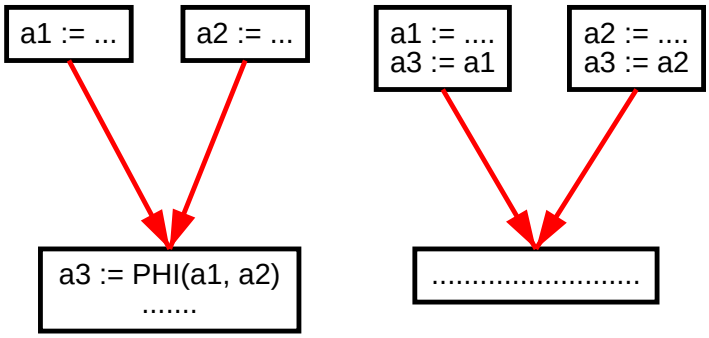

Abbildung 3.1: Entfernen von Phi-Funktionen

(*Lost Copy* und *Swap*) kennenlernen.

Briggs et. al. schlagen in [BCHS98] ein modifiziertes Verfahren vor, das solche Fälle erkennt und zusätzliche Kopien einfügt.

Aber auch mit dieser Modifikation bleibt das Verfahren für unseren Entwurf problematisch: Der Bytecode-nach-Bytecode Übersetzer hat keine Registerzuteilungsphase, die die unnötigen Kopien entfernen kann. Wir müssen also entweder eine solche Phase zur Entfernung der Kopien implementieren, oder ein Verfahren zum Phi-Abbau einsetzen, dass nur solche Kopien einfügt, die wirklich notwendig sind.

3.3 Das Phi-Kongruenz Verfahren

Ein solches Verfahren findet sich in [SJGS99]. Es baut auch in den Problematischen Fällen die SSA Form korrekt ab und fügt weit weniger Kopien ein als das Standardverfahren.

3.3.1 Grundidee

Die Grundidee des Verfahrens beruht auf folgernder Beobachtung:

Selektiert eine Phi-Funktion zwischen identischen Variablen, so kann man die Phi-Funktion durch eine einfache Zuweisung ersetzen:

$$a1 := PHI(a1, a1)$$

kann ersetzt werden durch

$$a1 = a1$$

Ist nun, wie in diesem Fall, auch das Ziel der Phi-Funktion gleich den Eingabevariablen, so kann man den Ausdruck entfernen.

Die Idee des Verfahrens ist nun, das Programm so zu transformieren, dass die Eingabevariablen aller Phi-Funktionen identisch sind. Die Phi-Funktionen können dann einfach entfernt werden.

3.4 Einige Definitionen

Für die Darstellung des Verfahrens benötigen wir einige Definitionen.

3.4.1 Variablen einer Phi-Funktion

Wir verstehen unter der Menge der Variablen einer Phi-Funktion die Vereinigung zwischen den Quellvariablen der Phi-Funktion und dem Ziel, also der Variable, an die zugewiesen wird.

Beispiel:

$$a_3 = PHI(a_1, a_2) \tag{3.1}$$

a_3 ist Zielvariable, a_1 und a_2 sind Quellvariablen. Alle drei Variablen sind Teil der Phi-Funktion.

3.4.2 Phi Kongruenzen

Definition 1 (Phi-Verbundenheit) *Gegeben sei eine Variable x aus einer Phi Funktion. Wir definieren:*

`PhiVerbunden`$(x) = \{y|x$ *und* y *sind Teil einer Phi-Funktion oder es existiert ein* z *mit: Sowohl* x *und* z *als auch* y *und* z *sind Teil einer Phi-Funktion*$\}$

Nach dieser Definition sind also zum einen alle Variablen einer Phi-Funktion verbunden, zum anderen sind alle Variablen von Phi-Funktionen dann verbunden, wenn diese Phi-Funktionen mindestens eine gleiche Variable enthalten.

Ein Beispiel:

$$a_3 = PHI(a_1, a_2)$$
$$a_5 = PHI(a_3, a_4)$$

Hier sind über die gemeinsame Variable a_3 auch a_1 und a_4 Verbunden.

Definition 2 (Phi Kongruenzklasse) *Unter der* Phi-Kongruenzklasse *der Variable* x *verstehen wir die reflexive und transitive Hülle von* `PhiVerbunden`(x).

Eine Phi-Kongruenzklasse definiert also die Menge der Variablen, die über Phi-Funktionen verbunden sind.

Definition 3 (Phi-Kongruenz Eigenschaft) *Eine SSA-Form hat die* Phi-Kongruenz Eigenschaft*, wenn man alle Vorkommen von Variablen einer Kongruenzklasse durch einen Repräsentanten ersetzen kann, ohne die Semantik des Programms zu ändern.*

Die Phi-Kongruenz Eigenschaft liegt dann vor, wenn die Kongruenzklassen keine überlappenden Lebensdauern haben.

Das Verfahren zum SSA Aufbau aus einem Strukturbaum von Cytron et. al. generiert eine SSA Form mit dieser Eigenschaft.

3.4.3 CSSA und TSSA

Definition 4 (CSSA) *Eine SSA-Darstellung nennen wir CSSA (*conventional SSA*), wenn sie die* Phi-Kongruenz Eigenschaft *besitzt.*

Nach Aufbau hat SSA diese Eigenschaft. Durch Optimierungen kann sie aber zerstört werden. So ändert z.B. Kopienweiterschaltung die Lebensdauer der Variablen, Interferenzen innerhalb der Phi-Kongruenzklassen können entstehen, die Phi-Kongruenz Eigenschaft gilt nicht mehr.

Definition 5 (TSSA) *Erfüllt die SSA Form die* Phi-Kongruenz Eigenschaft *nicht, so nennen wie die Form TSSA (*transformed *SSA)*

Das Ergebnis der Optimierungsphase des Übersetzers ist nicht unbedingt CSSA, sondern TSSA.

3.4.4 Lebendigkeit und Interferenz

Definition 6 (Lebendigkeit) *Eine Variable v ist* lebendig *an einer Stelle p eines Programms, wenn es einen Pfad zu einer Benutzung der Variable gibt, ohne das v auf diesem Pfad neu definiert wird.*

Dabei sind für Programme in SSA zwei wichtige Eigenschaften zu beachten:

1. Die Benutzung einer Phi-Quell-Variable passiert im entprechenden Vorgängerblock.
2. Die Phi-Funktionen werden alle am Beginn einer Grundblocks gleichzeitig ausgewertet

Die Phi-Funktionen stehen also am Anfang eines Grundblocks und arbeiten mit Werten, die am Ende der Vorgängerblöcke benutzt werden. Für unseren Algorithmus ist daher Lebendigkeit nur an zwei Stellen interessant: Anfang und Ende der Grundblöcke.

Definition 7 (LiveIn) *$LiveIn(B)$ bezeichnet die Menge aller lebendigen Variablen am Anfang des Grundblocks B.*

Definition 8 (LiveOut) *$LiveOut(B)$ bezeichnet die Menge der lebendigen Variablen am Ende des Grundblocks B.*

Definition 9 (Interferenz) *Zwei Variablen interferieren, wenn die Lebensdauer dieser Variablen überlappt.*

Wenn eine Menge Variablen nicht interferiert, kann man diese Menge durch eine Variable ersetzen, ohne das sich die Semantik des Programms ändert.

3.5 Gesamtüberblick

Mit den im letzten Abschnitt erläuterten Definitionen können wir nun das Verfahren beschreiben. Es wird in 3 Schritten ablaufen:

1. Transformation TSSA nach CSSA.
2. Ersetzen aller Variablen einer Phi-Kongruenzklasse durch einen Repräsentanten.
3. Löschen aller Phi-Funktionen.

3.6 Transformation in CSSA

Wir werden zwei Verfahren die Transformation von TSSA nach CSSA vorstellen. Das einfache Verfahren ist dabei dem klassischen dahingehend ähnlich, dass pessimistisch Kopien eingefügt werden.

Das Verfahren II ist eine Verbesserung des ersten Verfahrens, es fügt weit weniger Kopien ein, ist aber aufwendiger. Wir werden in Kapitel 6 beide Verfahren in Hinsicht auf die Anzahl der eingefügten Kopien und der Laufzeit untersuchen.

3.6.1 Verfahren I

Das einfachste Verfahren ist dem Standardverfahren ähnlich: Wir fügen Kopien für alle Quellen in die entsprechenden Vorgängerblöcke ein. Zusätzlich wird eine Kopie für die Quellvariable hinter die Phi-Funktionen eingefügt.

Dadurch wird erreicht, dass das Programm die Phi-Kongruenzeigenschaft hat und damit in TSSA Form vorliegt.

Bild 3.2 zeigt ein Beispiel.

Dieses Verfahren erzeugt TSSA Form, hat aber den Nachteil, dass es sehr viele Kopien einfügt, die nicht nötig sind.

3.6.2 Verfahren II

Die Grundidee des verbesserten Verfahrens besteht darin, nur diejenigen Kopien einzufügen, die nötig sind, um die CSSA Form zu erreichen.

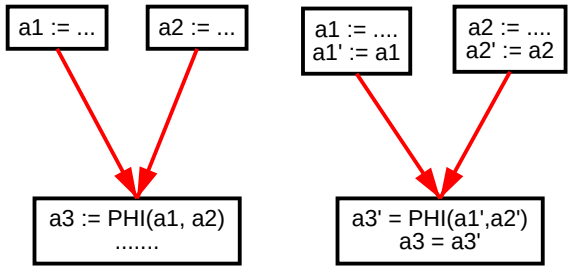

Abbildung 3.2: Einfache Transformation nach CSSA

So sind Kopien nur dann nötig, wenn die Variable zu einer Interferenz beiträgt. Zum Auflösen dieser Interferenzen werden nicht immer Kopien für beide Variablen nötig. Das Verfahren verwendet Datenflussinformationen in Form von *LiveIn* und *LiveOut* Mengen, um zu entscheiden, welche Kopien benötigt werden.

3.7 Beschreibung Algorithmus Verfahren II

Die Grundidee des Verfahrens besteht darin, die Phi-Kongruenzklassen schrittweise aufzubauen und immer dann Kopien einzufügen, wenn ohne die Kopie eine Interferenz entstehen würde.

Wir iterieren dazu über alle Phi-Funktionen aller Grundblöcke.

3.7.1 Kongruenzklassen aufbauen

Zu Beginn initialisieren wir alle Klassen so, das jede Variable Element in seiner eigenen Kongruenzklasse ist.

Dann gehen wir alle Phi-Funktionen durch. Für jedes Paar Variablen aus einer Phi-Funktion testen wir, ob die Kongruenzklassen der beiden Variablen überlappen.

3.7.2 Auf Interferenz testen

Wenn die Variablen x_i und x_j interferieren, so müssen Kopien eingefügt werden.

Seien L_j und L_i die den Variablen x_i und x_j zugeordneten Vorgängerblöcke.

Sind beide Variablen Phi-Quellvariablen, so müssen wir 4 Fälle unterscheiden. Dazu definieren wir

$M_1 = phiKongruenzKlasse(x_i) \cup LiveOut(L_j)$

$M_2 = phiKongruenzKlasse(x_j) \cup LiveOut(L_i)$.

1. M_1 **ist nicht leer,** M_2 **ist leer:** Eine Kopie für x_i wird vorgemerkt.
2. M_1 **ist leer,** M_2 **ist nicht leer:** Eine Kopie für x_j wird vorgemerkt.
3. M_1 **ist nicht leer,** M_2 **ist nicht leer:** Kopien für x_i und x_j werden vorgemerkt.
4. M_1 **ist leer,** M_2 **ist leer:** Eine Kopie für x_i oder x_j. Wir merken uns beide Kanditaten, die Entscheidung welche Kopie eingefügt wird, treffen wir nachdem alle Interferenzen der Phi-Funktion bekannt sind.

Falls eine der Variablen Ziel einer Phi-Funktion ist, so müssen wir die gleichen vier Fälle unterscheiden, wobei sich die Definition für M_1 und M_2 ändert. O.B.d.A. sei x_i die Zielvariable:

$M_1 = phiKongruenzKlasse(x_i) \cup LiveOut(B)$

$M_2 = phiKongruenzKlasse(x_j) \cup LiveIn(B)$.

Dabei ist B der Grundblock, in den die Phi-Funktion vorkommt.

3.7.3 Verzögerte Kopien

Im vierten Fall müssen wir eine Kopie für eine von zwei möglichen Variablen einfügen. Hier kann eine geschickte Wahl der Variable bedeuten, dass weniger Kopien eingefügt werden müssen.

Wir haben diese Fälle in die Tabelle T eingetragen: Bei Interferenz von x_i und x_j fügen wir x_i dem Eintrag $T(x_j)$ hinzu, x_j entspechend $T(x_i)$. Die Tabelle enhält also für jede Variable eine Mengen von interferierenden Variablen.

Nun können wir diese Tabelle nach der Größe dieser Mengen sortieren und dann der Reihe nach auflösen, indem wir für die Variablen Kopien einfügen.

Mit Hilfe dieser Heuristik stellt das Verfahren sicher, dass zuerst für diejenigen Variablen Kopien eingefügt werden, die mit mehreren Variablen Interferenzen bilden. Mit einer Kopie können evtl. gleichzeitig mehrere dieser Interferenzen aufgelöst werden.

3.7.4 Kopien einfügen

Nun fügen wir die als nötig erkannten Kopien in das Programm ein. Dabei passen wir auch die Interferenz- und Lebendigkeitsinformationen an, da diese für die nächste Iteration benötigt werden.

3.7.5 Kongruenzklassen anpassen

Im letzten Schritt müssen wir die Kongruenzklassen anpassen. Das Verfahren hat sichergestellt, dass alle Variablen der Phi-Funktion nicht interferieren. Das bedeutet, dass wir einfach die Kongruenzklassen aller Variablen der Phi-Funktion vereinigen können.

Damit ist eine Phi-Funktion abgearbeitet, das Verfahren wird nun für alle weiteren Phi-Funktionen aller Grundblöcke wiederholt.

3.8 Phi-Funktionen löschen

Wenn alle Phi-Funktionen abgearbeitet sind, haben wir Kopien so eingefügt, dass das Programm die Phi-Kongruenz-Eigenschaft hat. Wir können also die Variablen der einzelnen Kongruenzklassen durch einen Repräsentanten ersetzen und die Phi-Funktionen löschen.

3.9 Beispiele

Der Algorithmus wird nun anhand der Beispiele *Lost Copy* und *Swap* erläutert.

3.9.1 *Lost Copy*

Bild 3.3 zeigt das Lost Copy Problem vor (a) und nach SSA Aufbau und Kopienweiterschaltung (b).

Wenn wir das klassische Verfahren aus Kapitel 3.2.1 anwenden, so geht die Kopie $x = y$ verloren.

Wir wollen nun das Phi-Kongruenz-Verfahren anwenden. Dazu inititalisieren wir die Phi-Kongruenzklassen. Alle Konguenzklassen enhalten nur die Variable selbst.

Nun gehen wir alle Phi-Funktionen durch. In unserem Beispiel gibt es nur eine Phi-Funktion, daher wird der folgende Schritt nur für diese durchgeführt.

Interferierende Paare feststellen

In diesem Fall interferiert nur ein Paar: x_2 und x_3.

Interferenzen auflösen

Diese Intereferenz soll nun aufgelöst werden. Da x_2 Ziel der Phi-Funktion ist, bilden wir folgenden Mengen:

$M_1 = phiKongruenzKlasse(x_2) \cup LiveOut(L2)$

$M_2 = phiKongruenzKlasse(x_3) \cup LiveIn(L2)$.

Die Phi-Kongruenzklassen enhalten jeweils die Variable selber. Ausserdem ist x_2 ist in $LiveOut(L2)$ und x_3 nicht in $LiveIn(L2)$.

Also: M1 nicht leer, M2 leer. Wir müssen also eine Kopie für x_2 einfügen (Siehe Abschnitt 3.7.2).

Bild 3.3 (c) zeigt das Ergebnis. Es wurde eine Kopie eingefügt.

3.9.2 *Swap*

Als zweites Beispiel betrachten wir das *Swap* Program. Das Bild 3.4 zeigt den Code vor SSA Aufbau (a) in CSSA-Darstellung (b) und nach Kopienweiterschaltung (c).

Der erste Schritt ist die Initialisierung der Phi-Kongruenzklassen: Jede Variable ist Teil Element der eigenen Kongruenzklasse.

Dann berechnen wir die Lebendigkeit der Variablen am Begin und Ende der Grundblöcke:

$$LiveIn(L2) = \{x_2, y_2\}$$
$$LlveOut(L2) = \{x_2, y_2\}$$

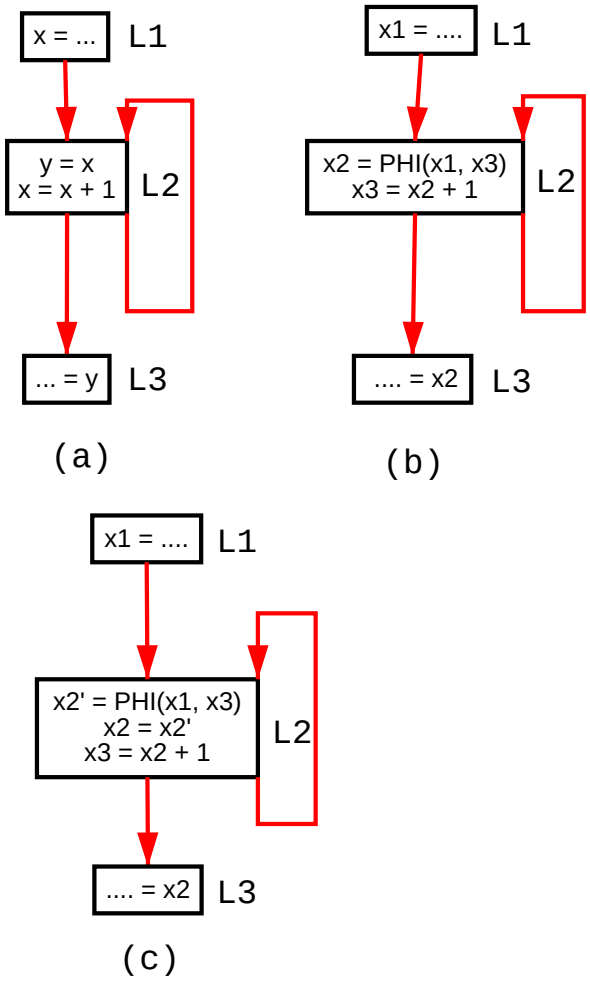

Abbildung 3.3: Lost Copy

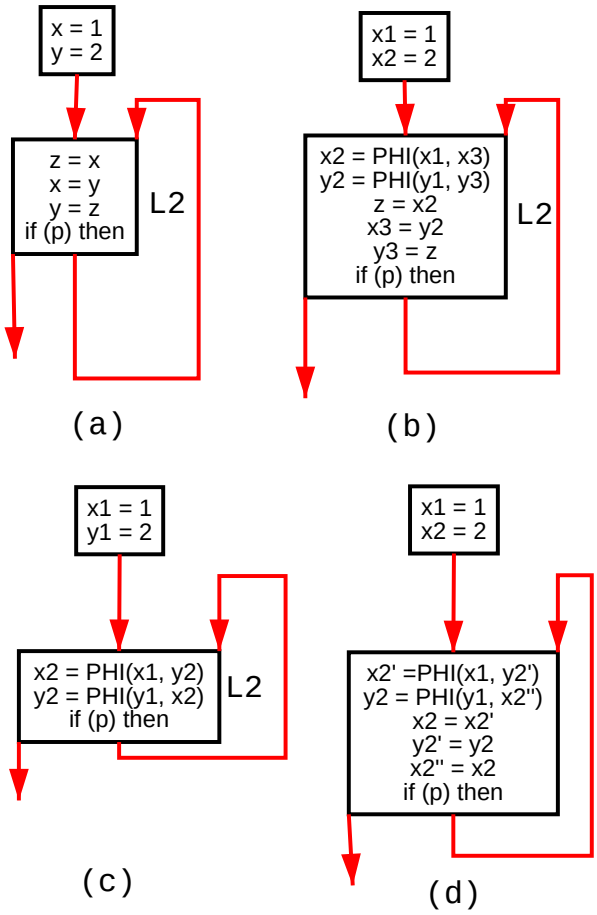

Abbildung 3.4: Swap

Nun gehen wir alle Phi-Funktionen durch.

Erste Phi-Funktion

x_1 und y_2 interferieren, y_2 ist $LiveIn(L2)$, x_2 ist $liveOut(L2)$. Daher müssen wir Kopien für beide Variablen einfügen.

Als nächstes werden $LiveIn$ und $LiveOut$ angepasst:

$$LiveIn(L2) = \{x'_2, y_2\}$$
$$LlveOut(L2) = \{x_2, y'_2\}$$

Nun verschmelzen wir die Phi-Kongruenzklassen aller Variablen der Phi-Funktion.

Zweite Phi-Funktion

x_1 und y_2 interferieren, y_2 ist nicht $LiveIn(L2)$, x_2 ist nicht $liveOut(L2)$. Daher müssen wir Kopien für eine der beiden Variablen einfügen.

Das Bild 3.4 (d) zeigt das Ergebnis: Es wurden drei Kopien eingefügt.

3.10 Zusammenfassung

Der Übersetzer muss die SSA-Darstellung abbauen. In diesem Kapitel haben wir die klassischen Verfahren und das *Phi-Kongruenz Verfahren* vorgestellt.

Bevor wir dieses Verfahren für unseren Übersetzer implementieren, gibt das nächste Kapitel einen Überblick über die Systeme und Rahmenwerke, die wir für die Implementierung verwenden werden.

Kapitel 4

Überblick existierende Infrastruktur

4.1 Überblick

Der Entwurf aus Kapitel 2 wurde innerhalb des Squeak-Systems realisiert. Für eine diesem System sehr ähnliche Plattform, *VisualWorks*, steht mit dem AOStA Rahmenwerk eine Implementierung eines Teiles des Übersetzers (Analyse des Bytecodes und Aufbau SSA) zur Verfügung. Der Übersetzer baut auf diesem System auf.

Wir wollen nun nach einer kurzen Einführung in Squeak das AOStA Rahmenwerk betrachten. Abschliessend folgt ein Überblick über den Assemblierer `IRBuilder`.

4.2 Verwendete Werkzeuge

Folgende existierende Subsysteme haben wir für die Implementierung verwendet:

- Das Squeak-System in der Version 3.7alpha
  (virtuelle Maschine, Entwicklungsumgebung)
- ClosureCompiler
  (eine Smalltalk-nach-Bytecode Übersetzer. Autor: Anthony Hannan)
- AOStA (SSA Rahmenwerk. Autor: Elliot Miranda)

- `IRBuilder`
  (ein symbolischer Assemblierer für Squeak Bytecode. Autor: Anthony Hannan)

4.3 Squeak

Das Squeak-System entstand als direkte Weiterentwicklung von Smalltalk 80 im Zuge von Alan Kays *Squeak* Projekt. Das System wird z. B. in [IKM+97] oder [GR01] ausführlich beschrieben. Hier soll ein kurzer Überblick ausreichen.

Squeak verwendet eine virtuelle Maschine, die eine Kellerarchitektur bereitstellt. Neben Befehlen zur Manipulation dieses Kellers (*push* und *pop*) gibt es Befehle zum Zugriff auf die lokalen Variablen der Methode und die Variablen der Instanzen.

Für die Ablaufsteuerung gibt es zum einen den Befehl *send* für den Aufruf von Methoden, zum anderen Sprünge für Schleifen und Bedingungen.

Die virtuelle Maschine implementiert in der aktuellen Version einen Interpreter, erste experimentelle Laufzeitübersetzer existieren.

Die verwendetet Sprache ist ein Dialekt von Smalltalk ([GR83]), es wurden aber auch Übersetzer für weitere Sprachen implementiert. Es gibt z. B. Übersetzer für die Sprachen E-Lang ([MS03]) und Logo ([Pap80]). Außerdem wurde im Rahmen des OpenCroquet Projekts ([SKRR03][SKRR04]) ein Übersetzer für Javascript implementiert.

4.4 Das AOStA Rahmenwerk

Die Implementierung des Übersetzers verwendet Teile eines SSA Rahmenwerks, das für *VisualWorks* (eine kommerzielle Smalltalk implementierung) entwickelt wird. Dieses System wird hier kurz vorgestellt.

4.4.1 Das AOStA Projekt

Ziel des AOStA Projekts ist die Realisierung der von Self ([Hö94]) bekannten dynamischen, adaptiven Optimierung in Smalltalk. AOStA ist ein *Open Source* Projekt unter der Leitung von Eliot Miranda (Firma Cincom). Einen Überblick findet man in [Bon03], ausführlichere Informationen bietet [Mir02].

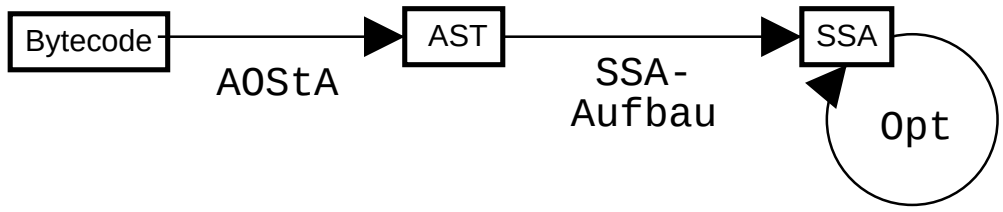

Abbildung 4.1: AOStA

Im Zuge des AOStA Projektes sind Teile eines Übersetzers (Bytecodeanalyse und SSA Aufbau) entstanden, die wir für die Implementierung unseres Entwurfs nutzen wollen.

4.4.2 Aufbau AOStA

Das AOStA-Rahmenwerk besteht aus drei Teilen. Am Begin steht die Analyse des Bytecodes und der Aufbau eines AST. Dieser wird dann im zweiten Schritt in SSA-Darstellung transformiert (siehe 4.1).

Der dritte Teil des AOStA Systems ist ein Rahmenwerk für Optimierungen, als Beispiel werden einige einfache Optimierungen bereitgestellt.

SSA Abbau und Codegenerierung sind nicht implementiert.

4.4.3 Eigenschaften

Die von AOStA erstellt SSA-Darstellung beschränkt sich auf die temporären Variablen der Methoden. Instanzvariablen werden nicht in SSA-Darstellung transformiert.

Der SSA Aufbau führt keine Wertnummerierung durch, identische Teilausdrücke werden nicht optimiert.

Definions-Nutzungs Informationen

AOStA stellt Definions-Nutzungs Informationen bereit. Jede Benutzung einer Variable hat also eine Referenz auf die Definion dieser Variable, jede Definition kennt alle ihre Benutzungen.

4.4.4 Typinformationen

Das AOStA-System erlaubt zu jeder Variable Typinformationen zu notieren. Smalltalk ist eine dynamisch getypte Sprache. Daher kann beim Aufbau der SSA-Darstellung nur für statisch feststellbare Fälle der Typ (also die Klasse) der Variablen festgestellt werden.

Transformationen

Für die Optimierungsphase stellen die Knotenklassen der AOStA SSA-Darstellung einfache Transformationen zur Verfügung. Man kann Knoten löschen oder durch einen anderen Knoten ersetzen. Dabei sorgt AOStA dafür, dass die Defininions-Nutzungs Verkettung korrekt bleibt.

Besucher

Mit Hilfe des Musters *Besucher* (siehe [GHJV97]) kann man die SSA-Darstellung durchlaufen. AOStA stellt dazu abstrakte Klassen bereit, die verschiedene Besuchsmuster definieren:

- `AOBreadthFirstDominatorTreeVisitor`
  In Reihenfolge Dominator-Baum Breitensuche.
- `AODepthFirstDominatorTreeVisitor`
  In Reihenfolge Dominator-Baum Tiefensuche
- `AONodeUnorderedVisitor` ungeordnet

4.4.5 Optimierungen in AOStA

Mit den Besucherklassen und den Transformationen lassen sich nun Optimierungen implementieren. AOStA stellt zwei solche Optimierungen als Bespiele bereit.

Konstantenweiterschaltung

Die Konstantenweiterschaltung (*constant propagation*) ersetzt Zugriffe auf eine Variable, der ein konstanter Wert zugewiesen wird, im gesamten Programm durch diese Konstante.

Realisiert ist die Konstantenweiterschaltung in AOStA als Unterklasse von `AOBreadthFirstDominatorTreeVisitor`. Der Baum wird also in Ausführungsreihenfolge durchsucht. Zuweisungen von Konstanten werden in einer Hash-Tabelle gespeichert. Beim Besuch einer Variable wird in dieser Tabelle nachgeschlagen und die eingetragenen Variablen dann durch den konstanten Wert ersetzt.

Entfernung toten Codes

Diese Optimierung entfernt Definitionen von Variablen, die im Programm nicht genutzt werden. Durch die im Baum enthaltene Definition-Benutzungs Information ist dies leicht festzustellen:

Ist die Menge der Nutzungen leer, so wird die definierte Variable niemals genutzt. Falls die Definition keine Seiteneffekte hat, ist nicht nötig und kann entfernt werden.

4.5 Der symbolische Assemblierer `IRBuilder`

Der `IRBuilder` stellt die Funktionalität eines symbolischen Assemblierers für Squeak Bytecodes bereit.

4.5.1 Beispiel

Das folgende Beispiel erzeugt eine Methode, die einfach die Zahl `1` zurückgibt.

```
ir := IRBuilder new
            rargs: #(self);
            pushLiteral: 1;
            returnTop;
            ir.
```

Nach Erzeugen einer `IRBuilder` Instanz (`IRBuilder new`) werden nacheinander die Methoden `#rargs:`, `#pushLiteral:` und `#returnTop` aufgerufen. Der Aufruf von `#ir` schliesst die Befehle ab. Die so erzeugte, bytecodenahe Zwischensprache kann nun in Bytecode umgesetzt werden:

```
aCompiledMethod := ir compiledMethod.
```

Der generierte Bytecode:

```
5 <76> pushConstant: 1
6 <7C> returnTop
```

Der Aufruf der Methode `#compiledMethod` erzeugt nicht einzig eine Reihung mit den Bytcodes, sondern ein vollständiges Objekt der Klasse `CompiledMethod`, das man direkt im System verwenden kann.

Man kann die Methode ausführen:

```
aCompiledMethod valueWithReceiver: nil arguments: #()
```

oder einer bestehende Klasse als Methode hinzufügen:

```
Float addSelector: #test withMethod: aCompiledMethod.
```

Hier haben wir das generierte `CompiledMethod` Objekt als Methode `#test` in der Klasse `Float` installiert. Diese neue Methode kann nun aufgerufen werden, sowohl bei existierenden als auch neuen Instanzen der Klasse `Float`.

4.5.2 Symbolische Sprungziele

Sehr wichtig für den Einsatz des `IRBuilder` im Codegenerator ist die Möglichkeit Sprungziele symbolisch zu definieren. Hier ein einfaches Beispiel eines `if-else`.

```
 1 < 2 ifTrue: [^10] ifFalse: [^20]
```

Wir wollen Bytecode erzeugen, wie er durch den Übersetzer für diesen Code erzeugt wird:

```
ir := IRBuilder new
            rargs: #(self);
            pushLiteral: 1;
            pushLiteral: 2;
            send: #<;
            jumpAheadTo: #true if: true;
```

```
pushLiteral: 10;
returnTop;
jumpAheadTo: #continue;
jumpAheadTarget: #true;
pushLiteral: 20;
returnTop;
jumpAheadTarget: #continue;
ir.
```

Daraus erzeugt `IRBuilder` den Bytecode

```
13 <76> pushConstant: 1
14 <77> pushConstant: 2
15 <B2> send: <
16 <A8 02> jumpTrue: 20
18 <20> pushConstant: 10
19 <7C> returnTop
20 <21> pushConstant: 20
21 <7C> returnTop
```

Der `IRBuilder` hat also aus den symbolisch definierten Zielen nun Sprünge generiert, die feste Positionen im Bytecode als Ziel haben.

4.6 Zusammenfassung

Dieses Kapitel hat die existierenden Systeme und Rahmenwerke vorgestellt, auf die die Realisierung des Übersetzers aufbauen wird. Nach einem kurzen Überblick über das Squeak-System haben wir das SSA-Rahmenwerk AOStA vorgestellt und an Squeak angepasst.

Eine Einführung in den symbolische Assemblierer `IRBuilder` schliesst dieses Kapitel ab.

Im nächsten Kapitel folgt nun die Umsetzung des Übersetzers basierend auf die hier vorgestellten Systeme.

Kapitel 5

Umsetzung

5.1 Überblick

Wir wollen nun den Entwurf aus Kapitel 2 implementieren. Der Entwurf sieht folgenden Aufbau vor:

- Analyse des Bytecodes und Aufbau SSA
- Optimierungen auf SSA
- SSA Abbau
- Code Generierung

Wir werden nun die Implementierung der einzelnen Teile des Übersetzers beschreiben.

5.2 Aufbau SSA-Darstellung

Wie wir im letzten Kapitel gesehen haben, können wir einige bestehende Systeme wiederverwenden. Für den ersten Teil des Übersetzers verwenden wir das AOStA Rahmenwerk. Dieses analysiert den Bytecode und baut eine SSA-Darstellung auf.

5.2.1 Portierung AOStA

AOStA ist für *VisualWorks* Smalltalk entwickelt worden. *VisualWorks* und Squeak basieren beide auf dem Xerox Smalltalk 80 System ([GR83]).

Beide Systeme wurden aber seit 1983 getrennt weiterentwickelt und weisen daher einige Unterschiede auf.

Diese Unterschiede betreffen z. B. die Bibliotheken. Es sind einige Anpassungen am Code nötig, um AOStA unter Squeak übersetzen zu können.

Der entscheidende Unterschied ist aber der verwendete Bytecode.

Beispielhaft zeigen wir die Unterschiede in zwei Bereichen: Die Codierung von *Closures* und ein einfaches Beispiel für unterschiedliche Bytecodes.

5.2.2 *Closures*

Smalltalk Blockobjekte können sowohl lokale Variablen definieren, als auch Variablen der Methode oder eines umschließenden Blocks referenzieren.

In Squeak werden alle diese Variablen als lokale Variablen der Methode realisiert. Eine andere Möglichkeit ist, den Blockobjekten eigene lokale Variablen zuzuordnen und auf die Variablen der Umgebung korrekt zuzugreifen: Die Blockobjekte sind dann anonyme Funktionen höherer Ordnung, wie sie aus funktionalen Sprachen bekannt sind. Diese *Closure*-Semantik (siehe [Lan64]) wurde in *Visualworks* implementiert.

In der aktuellen Squeak Version ist aber eine *Closure*-Semantik nicht vorhanden.

Es existiert eine experimenteller Smalltalk-nach-Bytecode Übersetzer für Squeak, der *Closure*-Semantik unterstützt. Die nötigen Änderungen an der virtuellen Maschine sind schon Teil von Squeak 3.6.

Für diese Arbeit verwenden wir daher diesen experimentellen Übersetzer in einer leicht angepassten Version, die es erlaubt, den Übersetzer parallel zum alten Übersetzer im Squeak-System zu betreiben.

Damit unterztützt Squeak *Closures* semantisch. Der verwendete Syntax, also die Codierung z. B. der Variablen, ist aber verschieden. Wir müssen AOStA entsprechend anpassen.

5.2.3 Bytecode für Schleifenerkennung

VisualWorks Smalltalk verwendet eine virtuelle Maschine mit Laufzeitübersetzer. Aus diesem Grund wurden die Bytecodes um einen `LoopHead` Befehl erweitert. Jeder Rücksprung hat als Ziel einen solchen `LoopHead` Bytecode. Dies vereinfacht das Erkennen von Schleifen und erlaubt daher einen einfachere Analysephase des Laufzeitübersetzers.

Auch AOStA setzt diesen `LoopHead` Befehl voraus. Daher muss der Squeak Compiler angepasst werden und den entsprechenden Code erzeugen. Der verwendete Bytecode ist unbelegt und wird von der virtuellen Maschine ignoriert.

5.3 Optimierungen auf SSA

Das AOStA Rahmenwerk stellt einige einfache Optimierungen bereit: Entfernung von totem Code (*Dead Code Elimination*) und Konstanten-Weiterschaltung (*Constant Propagation*). Diese beiden Optimierungen sind aber bzgl. des SSA Abbaus nicht interessant.

Wie in Kapitel 3.4 erwähnt, sind die Phi-Kongruenzklassen nach SSA Aufbau überlappungsfrei. Es liegt also bereits CSSA Form vor. Im Folgenden soll eine einfache Optimierung implementiert werden, die die Lebensdauer von Variablen ändert und dadurch Interferenzen zwischen Variablen erzeugt.

5.3.1 Kopienweiterschaltung

Die Kopienweiterschaltung ist eine solche Optimierung. Sie ist sehr ähnlich zur Konstantenweiterschaltung. Hier ein einfaches Beispiel:

```
BB5: [
      t2a := 1.
      s3a := t2a.
      ^s3a]
```

Die Zuweisung $s3a := t2a$ kann gelöscht werden, wenn man alle Vorkommen von $s3a$ durch $t2a$ ersetzt:

```
BB5: [
```

```
t2a := 1.
^t2a]
```

Die Implementierung verwendet die Definitions-Benutzt Information: Wir durchlaufen (mittels eines Besuchers) den Baum. Sobald wir auf eine Zuweisung der Form $var1 = var2$ treffen, ersetzen wir alle Benutzungen von $var1$ durch $var2$.

5.3.2 Entfernen leerer Grundblöcke

AOStA baut eine Darstellung auf, die leere Grundblöcke enthält. Hintergrund ist, dass der Smalltalk-nach-Bytecode Übersetzer keine bedingten Rücksprünge erzeugt, sondern stattdessen einen bedingen Sprung in einen Leeren Grundblock mit unbedingtem Rückspung.

Wir implementieren einen Besucher, der Leere Grundblöcke entfernt. Dabei müssen wir den Sprungbefehl so ändern, dass er auf das korrekte Ziel zeigt. Außerdem müssen wir auch die Nachfolger- und Vorgängerzeiger der Grundblöcke entsprechend anpassen.

5.4 SSA Abbau

Das AOStA Rahmenwerk bietet einzig Module zur Analyse des Bytecodes, dem Aufbau der SSA-Darstellung und der Optimierung. Ein SSA Abbau ist nicht implementiert.

Daher haben wir einen SSA Abbau nach dem Phi-Kongruenzverfahren implementiert, wie es in Kapitel 3.3 erläutert wurde. Das Verfahren hat drei Schritte:

1. Transformation in CSSA
2. Ersetzen aller Phi-Kongruenzklassen durch einen Repräsentanten
3. Entfernen der Phi-Funktionen

5.4.1 Berechnung Lebendigkeit und Interferenz

Das Verfahren benötigt einige Informationen, die AOStA nicht bereitstellt: Zum einen Informationen zur Lebendigkeit am Beginn und Ende der Grund-

blöcke (*LiveIn* und *LiveOut*), zum anderen Informationen zur Interferenz der Variablen.

Wir erweitern also die AOStA Klasse `AOMethodNode` um einen Interferenzgraphen und die Klasse `AOBasicBlockNode` um *LiveIn*- und *LiveOut*-Mengen.

Appel stellt in [App98] einen Algorithmus vor, der Lebendigkeit und Interferenz für eine SSA-Darstellung mit Definition-Benutzt Informationen berechnet:

Für jede Variable läuft man rückwärts bis zur Definition der Variable. Wenn man dabei auf die Benutzung einer anderen Variable trifft, so kann man eine Interferenz eintragen. Bei Blockende bzw. Blockstart aller durchlaufenen Blöcke ergänzt man die *liveIn/liveOut* Mengen der entsprechenden Blöcke.

5.4.2 Transformation in TSSA

Wir haben die beiden Verfahren implementiert. Technisch sind diese über das Besucher Muster als Klasse `TSSA2CSSA` und `TSSA2CSSASimple` realisiert.

Verfahren I

Das einfache Verfahren soll Kopien für alle Variablen der Phi-Funktionen einfügen. Wir realisieren es über eine Klasse `TSSA2CSSASimple`. Dies ist ein Besucher, der den Baum ungeordnet durchläuft und die Kopien für alle Variablen der Phi-Funktionen einfügt.

Verfahren II

Wenn wir die Lebendigkeit und Interferenz berechnet haben, können wir das in Kapitel 3 beschriebene Verfahren anwenden.

Die Klasse `TSSA2CSSA` implementiert es über einen Besucher.

5.4.3 Entfernen der Phi-Funktionen

Unter der Vorraussetzung, dass die Phi-Konruenzen sich nicht überlappen, ist der SSA Abbau sehr einfach:

1. Jede Variable, die innerhalb einer Phi-Fkt vorkommt, wird durch einen Repräsentanten ihrer Phi-Kongruenzklasse ersetzt.

2. Die Phi-Funktionen werden gelöscht.

Die Implementierung verwendet einen Visitor `CSSARemovePhis`. Wurzel des SSA-Baums ist das Knotenobjekt für die Methode. Dieses wird als erstes besucht, hier können wir also den ersten Schritt implementieren: Wir ersetzen alle Variablen der Phi-Kongruenzklasse durch einen Repräsentanten. Dann durchlaufen wir den Baum und löschen alle Phi-Funktionen.

5.5 Codegenerierung

Nach SSA Abbau ist die Zwischendarstellung in einer Form, aus der sehr einfach Bytecode erzeugt werden kann.

5.5.1 Überblick über die Bytecode-Erzeugung

Wie in Kapitel 2.5.4 dargestellt teilt sich diese Phase in zwei Teile: Erzeugen einer maschinennahen Zwischensprache und anschließender Generierung des Bytecodes.

Für die Assemblierung setzen wir ein schon vorhandes Modul ein: Den `IRBuilder`, der in Abschnitt 4.5 beschrieben wurde.

5.5.2 Codeerzeugung

Mit Hilfe des `IRBuilder` lässt sich nun sehr einfach Code erzeugen: Der Besucher durchläuft den Baum und ruft die entsprechenden Methoden einer `IRBuilder` Instanz auf.

Trifft der Codeerzeugungsbesucher beispielswiese auf einen Knoten der Klasse `MessageSendNode` (also einen Methodenaufruf), wird folgender Code ausgeführt:

```
visitMessageSendNode: aNode
        self visitNode: aNode receiver.
        self visitNodes: aNode arguments.
        builder send: aNode selector.
```

Der Codeerzeuger wird also erst weiter den Empfänger der Nachricht und dann die Argumente besuchen. Danch folgt ein Aufruf der `IRBuilder` Instanz: Ein Bytecode zur Aufruf der Methode wird erzeugt.

5.6 Ein Beispiel

Nun soll das Zusammenspiel der einzelnen Phasen des Übersetzers anhand eines einfachen Beispiels gezeigt werden. Wir verwenden dazu eine einfache `while` Schleife:

```
exampleWhileFalse
        | a |
        a := 0.
        [a > 10] whileFalse: [a := a + 1].
        ^a.
```

Der Smalltalk-nach-Bytecode Übersetzer generiert für `exampleWhileFalse` optimierten Code: Es werden keine Blockobjekte generiert, sondern der Code wird offen eingebaut und durch Sprünge kodiert:

```
9 <75> pushConstant: 0
10 <68> popIntoTemp: 0
11 <7E> loopHead
12 <10> pushTemp: 0
13 <20> pushConstant: 10
14 <B3> send: >
15 <A8 06> jumpTrue: 23
17 <10> pushTemp: 0
18 <76> pushConstant: 1
19 <B0> send: +
20 <68> popIntoTemp: 0
21 <A3 F4> jumpTo: 11
23 <10> pushTemp: 0
24 <7C> returnTop
```

Man sieht hier in Zeile 11 auch ein Beispiel für den `loopHead` Befehl.

Als nächstes analysiert AOStA den Bytecode und baut die SSA-Darstellung auf:

```
BB9: [
       t1a := 0]
BB11: [
      t1b := PHI(t1a, t1c).
      if (t1b > 10) goto BB23]
BB17: [
      t1c := t1b + 1.
      goto BB11]
BB23: [
      s2a := t1b.
      ^s2a]
```

Man sieht, dass in Block Nr. 11 eine Phi-Funktion eingefügt wurde.

Die Kopienfortschaltung optimiert einzig den Ausdruck im letzten Block. Nach Entfernen der Phi-Funktion erhält man dann:

```
BB9: [
           t2a := 0]
BB11: [
           if (t2a) > 10:) goto BB23]
BB17: [
           t2a := t2a + 1.
          goto BB11]
BB23: [
          ^t2a]
```

Diesen Baum durchläuft dann der Codegenerierungs-Besucher und erzeugt Bytecode identisch zur Eingabe.

5.7 Zusammenfassung

Der Entwurf aus Kapitel 2 wurde implementiert. Als nächstes werden wir die Ergebnisse vorstellen.

Kapitel 6

Ergebnisse

6.1 Überblick

Nachdem wir den Entwurf aus Kapitel 2 implementiert haben, wollen wir nun die Ergebnisse darstellen.

Nach einem Überblick über den Stand der Implementierung werden einige vergleichende Messungen des SSA Abbaus vorgenommen.

6.2 Stand der Implementierung

Der Entwurf wurde wie in Kapitel 3 beschrieben implementiert.

Die Codegenerierungsphase ist noch nicht ganz vollständig: So fehlen einige Optimierungen wie beispielsweise Kodierung sehr einfacher Methoden im Kopf des Methodenobjekts (*Quick Methods*).

Außerdem sind primitive Methoden, also der Aufruf von Code, der als Teil der virtuellen Machine implementiert ist, noch nicht realisiert.

6.3 Tests und Evaluierung

Für den Test des Übersetzers wollen wir Squeak Bytecode übersetzen lassen. Dabei werden wir zwei Gruppen von Testdaten verwenden:

- Eine Sammlung von Beispielen

- Der Bytecode des gesamten Squeak-Systems

Die Beispiele sind dabei so gewählt, dass keine im Codegenerator fehlende Funktionalität benötigt wird. Hier kann also der gesamte Übersetzer, einschließlich Codegenerierung, getestet werden.

Für den Test des Übersetzer ohne Codegenerator, also Aufbau SSA, Optimierung, SSA Abbau, kann der Code des gesamten Squeak-Systems (und damit auch der Code des Übersetzers) verwendet werden.

6.3.1 Sammlung von Testfällen

Die Sammlung von Testfällen enthält auch die Beispiele `LostCopy` und `Swap` aus Kapitel 3. Daneben ca. 90 weitere Methoden, die im Zuge der Entwicklung des Übersetzers entstanden.

Mittels `SUnit` [Bec] lassen sich nun automatisch ablauffähige Testsammlungen erstellen. Die Testsammlungen enthalten insgesamt ca. 600 Tests.

Die Testsammlungen decken alle Phasen des Übersetzers ab. Daneben werden auch weitere Klassen getestet, wie z.B. die Berechnung der Lebendigkeit.

Wir wollen zwei Testklassen genauer betrachten.

Test SSA Aufbau

Dieser Test baut die SSA-Darstellung auf und tested dann die Definitions-Benutzungs-Informationen und die Rückverkettung der enstandenen SSA-Darstellung.

```
testExampleSwap
self shouldnt: [
     methodNode := parser parse: (AOStAExamples
                   compiledMethodAt: #exampleSwap).
     AOCheckDefUseChains check: methodNode.
     AOCheckParentNodeConnection new visitNode: methodNode.
] raise: Error.
```

Test Codegenerierung

Die Klasse `AOStACodeGenVisitorTest` testet die Codegenerierung. Im Folgenden sehen wir den Test für das Beispiel `Swap`:

```
testSwap
   |  method |
   aOMethodNode := AOBytecodeToStaticSingleAssignment
           parse: (AOStAExamples compiledMethodAt: #exampleSwap).

    AOStaticOptimizer optimize:  aOMethodNode. self checkTree.

   AOLivenessCalculator analyse: aOMethodNode.
   TSSA2CSSA new visitNode: aOMethodNode. self checkTree.

   CSSARemovePhis new visitNode: aOMethodNode.
   self checkTreeNonSSA.

   method := AOStACodeGenVisitor new genMethod: aOMethodNode.
   self assert: (method valueWithReceiver: nil arguments: #()) = 2
```

Man erkennt sehr gut die einzelnen Phasen der Übersetzung:

1. Analyse Bytecode und Aufbau SSA
2. Optimierung (`AOStaticOptimizer`)
3. Berechnung Lebendigkeit (`AOLivenessCalculator`)
4. Transformation *TSSA* nach *CSSA*(`TSSA2CSSA`)
5. Entfernung der Phi-Funktionen (`CSSARemovePhis`)
6. Codegenerierung (`AOStACodeGenVisitor`)

Nach jedem Übersetzungsschritt testen wir einige Eigenschaften (z.B. korrekte Rückverkettung) des transformierten Baums. Am Ende wird die generierte Methode ausgeführt und getestet, ob der Rückgabewert korrekt ist.

6.3.2 Ein statischer Übersetzer

Wir wollen das gesamte System als Testdaten für unseren Übersetzer verwenden.

Dazu erstellen wir einen statischen Übersetzer, der mit Hilfe des Laufzeitübersetzers einzelne Klassen übersetzt.

Die Klasse `AORecompiler` für dazu für alle Methoden der Klasse folgende Schritte durch:

- Erzeugen von Bytecode (ClosureCompiler)
- SSA-Darstellung aufbauen mit AOStA
- Optimierung
- Phi-Funktionen entfernen
- Zählen der eingefügten Kopien
- Codegenerierung (optional)
- Installation des Codes im System (optional)

Dabei ist die Codegenerierung optional. Dies ist nötig, da der Codegenerator noch nicht für das gesamte System Code erzeugen kann.

Mit dem `AORecompiler` können wir nun das System übersetzen. Im nächsten Abschnitt werden wir einige Messungen vorstellen.

6.4 SSA Abbau

Wir wollen nun die beiden Verfahren zum SSA Abbau in drei Bereichen genauer untersuchen:

- Anzahl eingefügter Kopien
- Performanz Abbau
- Performanz Code

6.4.1 Eingefügte Kopien

Als Testdaten verwenden wir zum einen die Beispielklasse, die im Zuge der Entwicklung entstand (90 Methoden), Zum anderen einen Teil des Squeak-Systems: Alle Klassen, die mit den Buchstaben A-I anfangen. (719 Klassen mit 13235 Methoden). Die Ergebnisse zeigt Tabelle 6.1.

Ohne Optimierungen liegt CSSA-Form vor: Das Verfahren II fügt keine Kopien ein. Auch nach Optimierung (Kopienweiterschaltung) ist die Anzahl der eingefügten Kopien gering.

Tabelle 6.1: Anzahl der eingefügten Kopien

	Kopien Bespiele	Kopien Squeak-Klassen
Verfahren I	126	23891
Verfahren II ohne Opt.	0	0
Verfahren II mit Opt.	12	1023

Tabelle 6.2: Übersetzung der Beispiele

Verfahren	Zeit
I	36430 msec
II	36325 msec

6.4.2 Performanz SSA Abbau

Der SSA Abbau durch Verfahren ist relativ aufwendig. Wir wollen untersuchen, inwiweit dies durch die geringere Anzahl von eigefügten Kopien kompensiert wird.

Tabelle 6.2 zeigt die Dauer der Übersetzung der Beispiele (90 Methoden).

Der Unterschied ist gering. Wir reduzieren die Methoden auf 6 Beispiele, in denen 18 Kopien mit Verfahren I und 4 Kopien mit Verfahen 2 eingefügt werden. Neben der Gesamtlaufzeit von jeweils 10 Läufen wollen wir auch den konstanten Aufwand, also den Lauf des Übersetzers ohne SSA Abbau, messen.

Tabelle 6.3 zeigt die Ergenisse. Das Verfahren II ist schneller als das einfachere Verfahren I.

Dabei muss man bedenken, dass im einfachen Fall noch die Laufzeit einer

Tabelle 6.3: Laufzeit des SSA Abbaus

	Laufzeit	Laufzeit SSA Abbau
Verfahren I	21058 msec	1077 msec
Verfahren II	20360 msec	379 msec
ohne Abbau	19981 msec	0 msec

Tabelle 6.4: Laufzeit Code

Verfahren	Zeit
Verfahren I	3,4 sec
Verfahren II	2,6 sec

Optimierungphase nach dem SSA Abbau hinzugerechnet werden muss. Diese ist nötig, da die Kopien, wenn sie nicht minimiert werden, die Performanz verschlechtern. Dies wollen wir im nächsten Abschnitt messen.

6.4.3 Performanz Code

Wir führen Code aus, jeweils mit SSA Abbau nach beiden Verfahren. Dabei verwenden wir folgendes Beispiel:

```
exampleWhileTrue
| a |
a := 1000.
[a > 0] whileTrue: [ a := a - 1].
^a.
```

Verfahren I fügt drei Kopien ein, das zweite Verfahren keine. Wir wiederholen die Ausführung 10000 mal (siehe Tabelle 6.4).

Wir sehen, dass die eingefügten Kopien einen großen Einfluss auf die Geschwindigkeit des Codes haben.

6.5 Zusammenfassung

In diesem Kapitel haben wir die Implementierung des Übersetzers untersucht. Für die dazu notwendigen Tests haben wir zum einen eine Sammlung von Beispielen implementiert, zum anderen den Übersetzer in das Squeak-System integriert.

Wichtig war für uns insbesondere der Abbau der SSA-Darstellung. Das verwendete Verfahren wurde auf die Anzahl der eingefügten Kopien untersucht.

Kapitel 7

Zusammenfassung und Ausblick

7.1 Zusammenfassung

Im letzen Kapitel wurden die Ergebnisse vorgestellt. Wir Bewerten nun die Arbeit anhand der Zielkriterien. Abschließend folgt ein Ausblick auf mögliche zukünftige Entwicklungen.

7.2 Bewertung

In Abschnitt 1.3 wurden einige Zielkriterien definiert, die der Übersetzer erfüllen soll:

1. Vollständiger Entwicklungszyklus unterstützt
2. Der Übersetzer erlaubt weitreichende Optimierungen
3. Integration in ein vorhandenes System

Wir haben als Datenbasis Bytecode gewählt. Dies erlaubt den existierenden Smalltalk-nach-Bytecode Übersetzer für das Lesen von Quelltexten zu verwenden. Bereits übersetzter Bytecode kann der Laufzeitübersetzer einlesen und wieder Bytecode ausgeben. Der Übersetzer deckt damit den vollständigen Entwicklungszyklus ab, Kriterium 1 wurde erfüllt.

Wir haben für die Zwischensprache des Übersetzers eine SSA-Darstellung implementiert. Diese wurde über das AOStA Rahmenwerk realisiert, das weitreichende Möglichkeiten für die Implementierung von Optimierungen bietet. Somit ist auch das Kriterium 2 erfüllt.

Der Übersetzer wurde in Squeak integriert. Wie Kapitel 4 zeigt, kann man Methoden übersetzen und den generierten Bytecode im System installieren. Dies erfüllt Kriterium 3.

Der entwickelte Übersetzer erfüllt also alle in Abschnitt 1.3 definierten Ziele.

7.3 Ausblick

Im Folgenden werden wir einige Bereiche möglicher weiterer Entwicklungen darstellen.

7.3.1 Verbesserungen AOStA

Das für die Realisierung des Übersetzers verwendete AOStA Rahmenwerk hat einige Schwachstellen gezeigt. Besonders die Kodierung der Phi-Funktionen sollte verbessert werden.

Eingabevariablen Phi-Funktionen

In AOStA werden die Eingabevariablen der Phi-Funktionen als Menge representiert, die Zuordnung zu den Grundblöcken geht verloren. Diese Information ist aber für die Berechnung der Lebendigkeit und SSA Abbau nötig.

Für diese Arbeit haben wir daher die Knotenklasse der Phi-Funktionen um eine Tabelle erweitert, die jeder Phi-Eingabevariable den entsprechenden Grundblock zuordnet. Diese Tabelle kann man sehr einfach während des SSA Aufbaus füllen.

Eine zukünfige AOStA Version sollte diese Tabelle mit der vorhandenen Kodierung der Phi-Eingabevariablen vereinheitlichen.

Kodierung Phi-Zuweisung

AOStA kodiert Phi-Funktionen als eine Selektorfunktion, die einer Variable über eine normale Zuweisungsoperation zugewiesen wird.

Um diese Phi-Zuweisungen von normalen Zuweisungen zu unterscheiden, bietet AOStA den Test `#isPhiFunctionAssigmentNode`.

Diese Kodierung hat einige Nachteile: Meist ist man nicht an der Phi-Funktion im Sinne von AOStA, sondern der gesamten Phi-Zuweisung in-

teressiert.

Im Besucher muss man also nicht die Phi-Knoten besuchen, sondern alle Zuweisungen, und dann testen, ob es sich um eine Phi-Zuweisung handelt. Dies bedeutet für die Implementierungen von z. B. Optimierungen und Phi-Abbau einen unnötigen Aufwand.

AOStA sollte stattdessen für die komplette Phi-Zuweisung eine eigene Knotenklasse bereitstellen.

7.3.2 Vereinfachung Smalltalk Übersetzer

Der Smalltalk-nach-Bytecode Übersetzer führt einige einfache Optimierungen durch. So werden Methodenaufrufe für Kontrollstrukturen offen eingebaut.

Diese Optimierungen können nun in den Laufzeitübersetzer integriert werden, eine Optimierung im Smalltalk Übersetzer ist nicht nötig.

7.3.3 Laufzeitinformationen für Optimierung

Der entwickelte Übersetzer kann Code unter Ausnutzung zusätzlicher Informationen weiter optimieren. Beispiele für solche Informationen sind etwa Statistiken über die auftretenden Klassen an einer Stelle eines Methodenaufrufs im Programm.

Die virtuelle Maschine muss entsprechend angepasst werden, diese Informationen bereitzustellen.

7.3.4 Zwischensprache

Eine weitere interessante Frage ist, ob man in einem solchen System wirklich Bytecode als Zwischensprache verwenden soll: Bytecode ist nicht sehr gut geeignet für diesen Zweck, da die Analyse relativ aufwendig ist. Die Stärke des Squeak Bytecodes ist die einfache Interpretierbarkeit. Unter Vorraussetzung eines Laufzeitübersetzers sind andere Zwischendarstellungen vielleicht interessanter. (Siehe z. B. SafeTSA [ADvRF01] und [KF96]).

Literaturverzeichnis

[ADvRF01] Wolfram Amme, Niall Dalton, Jeffery von Ronne, and Michael Franz. SafeTSA: A type safe and referentially secure mobile-code representation based on static single assignment form. In *Proceedings of the ACM SIGPLAN'01 Conference on Programming Language Design and Implementation*, pages 137–147. ACM Press, 2001.

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

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Priciples, Techniques and Tools.* Addison-Wesley, 1986.

[BCHS98] Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. Practical Improvements to the Construction and Destruction of Static Single Assignment Form. *Software Practice and Experience*, 28(8):859–881, 1998.

[Bec] Kent Beck. Simple Smalltalk Testing With Patterns. unpublished.

[Bon03] Paolo Bonzini. Inside AOStA. In *ESUG 2003*, 2003.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[GHJV97] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns.* Addison-Wesley, 1997.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, 1983.

[GR01] Mark Guzdial and Kim Rose, editors. *Squeak, Open Personal Computing for Multimedia*. Prentice-Hall, 2001.

[GW84] G. Goos and W. Waite. *Compiler Construction*. Springer Verlag, 1984.

[Hö94] Urs Hölzle. Adaptive Optimization for SELF: Reconciling High Performance with Exploratory Programming. Thesis CS-TR-94-1520, Stanford University, Department of Computer Science, 1994.

[IKM+97] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the Future: The Story of Squeak, A Practical Smalltalk Written in Itself. In *Conference Proceedings of OOPSLA '97, Atlanta*, volume 32(10) of *ACM SIGPLAN Notices*, pages 318–326. ACM, 1997.

[KF96] Thomas Kistler and Michael Franz. A Tree-Based Alternative to Java Byte-Codes. Technical Report ICS-TR-96-58, 1996.

[Lan64] P.J. Landin. The mechanical evaluation of expressions. *Computer J.*, (6(4)), 1964.

[Mir02] Eliot Miranda. A Sketch for an Adaptive Optimizer for Smalltalk written in Smalltalk. unpublished, 2002.

[MS03] Mark S. Miller and Jonathan S. Shapiro. Paradigm Regained: Abstraction Mechanisms for Access Control. Technical Report HPL-2003-222, HP Laboratories Palo Alto, 2003.

[Pap80] Seymour Papert. *Mindstorms: Children, Computer and Powerful Ideas*. Basic Books, 1980.

[SJGS99] Vugranam C. Sreedhar, Roy Dz-Ching Ju, David M. Gillies, and Vatsa Santhanam. Translating Out of Static Single Assignment Form. In *Proceedings of the 6th International Symposium on Static Analysis*, pages 194–210. Springer-Verlag, 1999.

[SKRR03] David A. Smith, Alan Kay, Andreas Raab, and David P. Reed. Croquet, A Collaboration System Architecture. In *C5: Conference on Creating, Connecting and Collaborating through Computing*, 2003.

[SKRR04] David A. Smith, Alan Kay, Andreas Raab, and David P. Reed. Croquet - A Menagerie of New User Interfaces. In *C5: Conference on Creating, Connecting and Collaborating through Computing*, 2004.