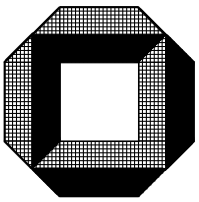

Universität Karlsruhe (TH)
Institut für
Programmstrukturen
und Datenorganisation
Lehrstuhl Professor Goos

Entwurf von Optimierungen für Squeak

Marcus Denker

Studienarbeit

Betreuender Mitarbeiter:
Dipl. - Inform. Dirk Heuzeroth

Januar 2003

Inhaltsverzeichnis

1 Einleitung **4**

1.1 Gliederung 4

2 Das Squeak-System: Hintergrund und Anforderungen **5**

2.1 Überblick 5

2.2 Geschichte 5

2.3 Squeak im Einsatz 6

2.4 Technische Anforderungen 9

2.4.1 Dynamische Metaprogrammierung 9

2.4.2 Portabilität 9

2.4.3 Erwartetes Verhalten 10

2.4.4 Performanz 10

2.5 Zusammenfassung 10

3 Die aktuelle virtuelle Maschine **11**

3.1 Die Implementierung von Squeak 11

3.1.1 Die virtuelle Maschine 11

3.1.2 ObjectMemory 12

3.1.3 Interpreter 13

3.1.4 Primitive Funktionen 13

3.1.5 Portierung 14

3.2 Bewertung 14

3.2.1 Performanz 15

3.2.2 Erwartetes Verhalten 15

3.3 Zusammenfassung 16

4 J3: Ein Squeak-Laufzeitübersetzer **17**

4.1 Projektziele 17

4.2 Architektur 17

4.2.1 Squeak Abstract Machine (SAM) 18

4.2.2 Optimierung und Codegenerierung 18

4.2.3 Der Laufzeitassembler CCG 19

4.2.4 Beispiel 19

4.3 Performanz 20

4.3.1 Messung der Interpreterleistung 21

4.3.2 Leistung des Gesamtsystems 21

4.4 Portabilität 22

4.5 Bewertung 23

4.6 Zusammenfassung 23

5 Entwurf SX (*Squeak eXtreme*) **24**

5.1 Überblick 24

5.2 Ziele 24

5.3 Architekturüberlegungen 24

5.3.1 Bytecode als Zielsprache 25

5.3.2 Übersetzer in Smalltalk 27

5.3.3 Laufzeitübersetzer ohne JIT 27

5.4 Weitere Konsequenzen 29

5.4.1 Ein statischer Smalltalk Übersetzer 29

5.4.2 Anbindung von Werkzeugen über die Zwischensprache 29

6 Zusammenfassung **30**

A Quellcode des *tinyBenchmark* **31**

1 Einleitung

Das Squeak-System möchte eine flexible Programmierumgebung und ein mächtiges multimediales Autorensystem für Kinder bereitstellen. Die besonderen Eigenschaften eines solchen Systems stellen besondere Anforderungen an die Implementierung.

Die Studienarbeit zeigt die Probleme der bestehenden Implementierungen auf und stellt einen verbesserten Entwurf vor.

1.1 Gliederung

Das Kapitel 2 gibt einen Überblick über das existierende Squeak-System. Nach einer kurzen Vorstellung des Squeak-Projektes gehe ich insbesondere auf die Anforderungen ein, die Squeak an die Implementierung des Systems stellt.

Das Kapitel 3 zeigt, wie das aktuelle System diese Anforderungen durch Verwendung einer virtuellen Maschine erfüllt und stellt die Zielkriterien auf, die ein verbessertes System erfüllen muß.

Den ersten Versuch einer Verbesserung stelle ich in Kapitel 4 vor. Das Kapitel stellt den Squeak-Laufzeitübersetzer J3 vor, erläutert die Ergebnisse des J3 Projektes und stellt dar, inwieweit sie die Kriterien aus Kapitel 3 erfüllen.

Die Erfahrungen aus dem J3-Projekt bilden die Grundlage meines neuen Entwurfes. Dieser soll die Vorteile des ursprünglichen interpretativen Systems mit denen des Laufzeitübersetzers verbinden. Dazu definiert er eine neue virtuelle Maschine in Kapitel 5. Dieses Kapitel gibt außerdem einen Ausblick über zukünftige Entwicklungen.

2 Das Squeak-System: Hintergrund und Anforderungen

2.1 Überblick

Dieses Kapitel stellt das aktuelle Squeak-System kurz vor. Nach einem kurzen Überblick zeigt Abschnitt 2.4 die besonderen Anforderungen auf, die Squeak an die Implementierung der virtuellen Maschine stellt.

2.2 Geschichte

Das Squeak-Projekt wurde 1995 von einer kleinen Forschungsgruppe (geleitet von Alan Kay) bei Apple gestartet. Nach 5 Jahren bei *Walt Disney Imagineering* arbeitet die Gruppe nun selbständig als *Viewpoints Research Institute*[1].

Die Anfänge von Squeak lassen sich weiter bis ans Xerox PARC zurückverfolgen. Squeak basiert auf dem Smalltalk 80 System, das in den Jahren 1971 – 1980 unter der Leitung von Alan Kay am Xerox PARC entwickelt wurde.

Ziel des Smalltalk-Projektes war es, einen Computer für Kinder (das *Dynabook*) zu entwickeln, ein interaktives Buch. Das Smalltalk System war Prototyp für Betriebsystem, Benutzeroberfläche, Programmiersprache und Entwicklungssystem dieses Kindercomputers.

Im Zuge des Smalltalk-Projektes wurde ein erstaunliches System geschaffen, das seiner Zeit weit voraus war: Es war das erste System mit einer modernen, fensterbasierten graphischen Benutzeroberfläche. Das System war mit der ersten modernen objektorientierten Programmiersprache nach Simula implementiert. Diese basierte, wie Java 20 Jahre später, auf einer bytecodebasierten virtuellen Maschine mit automatischer Speicherbereinigung.

Das Ziel des Smalltalk-Projektes, das Dynabook, wurde aber nie erreicht. Das Smalltalk-System wurde stattdessen in Richtung eines professionellen Entwicklungsystems weiterentwickelt und als solches 1983 kommerzialisiert.

Squeak versteht sich als Fortführung des Smalltalk Projektes, es ist in gewisser Weise ein neuer Versuch, das *Dynabook* zu verwirklichen.

[1] `www.viewpointsresearch.org`

2.3 Squeak im Einsatz

Das *Learning Research Institute* entwickelt mit Squeak ein System, mit dem Kinder spielerisch mit neuen Ideen umgehen und diese lernen sollen.

Das Squeak-System wird in Schulen (z. B. der *Open Charter School* in Los Angeles/USA) im Unterricht eingesetzt. Die Squeak-Umgebung erlaubt es den Kindern, spielerisch die im Mathematikunterricht gelernten Ideen kennen zu lernen.

Dazu bietet Squeak eine graphische Skriptsprache, mit deren Hilfe sich sowohl einfache als auch komplexere Animationen und Simulationen erstellen lassen. Alle Änderungen an diesen Skripten sind sofort spürbar, man kann sehr einfach und mit wenig Aufwand verschiedene Möglichkeiten ausprobieren.

Als Beispiel dient im folgenden ein typisches Projekt für die fünfte oder sechste Klasse: Ein Auto soll von einem Programm gesteuert über eine Straße fahren.

Nachdem das Kind ein Auto gemalt hat, kann es sich alle Befehle anzeigen lassen, die jedes Bildschirmobjekt beherrscht (mittels des *Betrachters*, siehe Abbildung 1). Wenn man einen dieser Befehle auf eine freie Stelle des Bildschirm zieht, entsteht ein neues Programm (siehe Abbildung 2).

Abbildung 3 zeigt das Ergebnis des Projektes. Man erkennt drei Skripte: Eines bringt das Auto dazu, vorwärts zu fahren (`Auto gehe vorwärts um 5`), die beiden weiteren Skripte drehen das Auto in die richtige Richtung, um auf der Straße zu bleiben. Nach einem Klick auf den „Go" Knopf werden die Skripte nun immer wieder, von einer globalen Uhr gesteuert, aufgerufen. Diese „tickende" Art der Methodenausführung erlaubt es, komplexe Kontrollstrukturen erst zu einem späteren Zeitpunkt einzuführen.

Die Kinder können auch zur Laufzeit mit den erstellten Skripten experimentieren: Während das Programm läuft kann man Parameter ändern, neue Befehle hinzufügen oder ganze neue Skripte erstellen. Dadurch werden die Auswirkungen der einzelnen Änderungen sofort sichtbar. Man erkennt sehr schnell Zusammenhänge und Funktionsweisen der einzelnen Befehle.

Das Ergebnis ist ein Auto, das programmgesteuert auf der Straße fährt. Wenn alle Kinder ihre Autos fertig entwickelt haben, veranstaltet die Klasse ein Autorennen: Die Autos werden zum Rechner des Lehrers gesendet, und schon kann das Rennen beginnen (siehe Abbildung 4).

Abbildung 1: Der Betrachter

Abbildung 2: Drehe das Auto um fünf Grad im Uhrzeigersinn

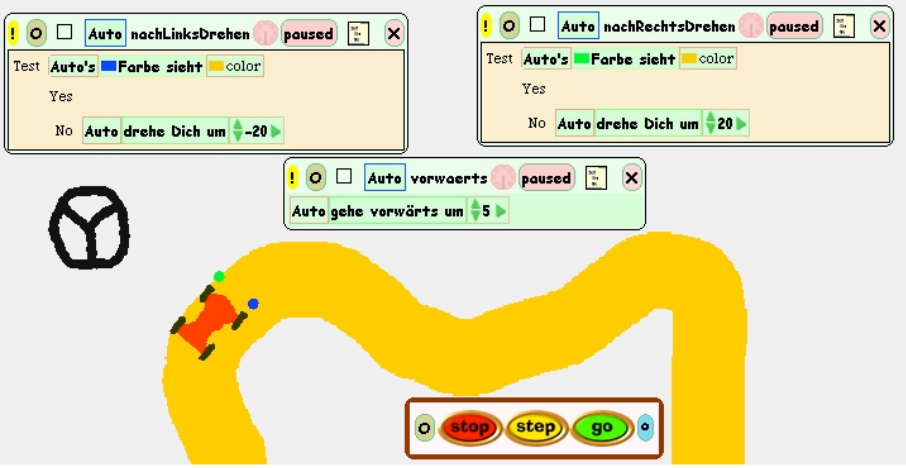

Abbildung 3: Ein programmiertes Auto in Squeak

Abbildung 4: Autorennen in Squeak

2.4 Technische Anforderungen

Als Konsequenz aus dem im vorigen Kapitel dargestellten Einsatz von Squeak als Lernmedium ergeben sich einige interessante technische Anforderungen, die das System erfüllen muß.

2.4.1 Dynamische Metaprogrammierung

Unter dynamischer Metaprogrammierung versteht man die Änderung eines Programms durch ein zweites Programm (das Metaprogramm) zur Laufzeit.

Ein Kind, das mit einem einfachen Squeak-Skript eine Animation erstellt, verwendet Squeak zur Metaprogrammierung: Mit Squeak wird ein Programm erstellt und ausgeführt. Die Metaprogrammierung ist dynamisch, da man das Programm nicht anhalten muß: Es lassen sich neue Methoden zu einem Objekt zur Laufzeit unter Beibehaltung des Zustandes hinzufügen. Natürlich ist auch die Änderung bestehender Methoden oder die Definition neuer Klassen zur Laufzeit möglich.

Eine der Voraussetzungen für dynamische Metaprogrammierung ist, daß man die Ausführung eines Programms zu jeder Zeit anhalten, den Code verändern und dann das Programm weiterlaufen lassen kann.

Dies hat sich gerade für ein Lernsystem als sehr wichtig erwiesen: Es erlaubt, sehr einfach den Ablauf eines Programms zu erforschen.

Ein Edit-Compile-Run Zyklus, wie man ihn von vielen Programmiersprachen kennt, ist eher kontraproduktiv: Er macht jedes spielerisch-explorative Umgehen mit dem Entwicklungssystem unmöglich

2.4.2 Portabilität

Im Schulbetrieb kommen viele verschiedene Betriebsysteme zum Einsatz. Der Ausbildungsmarkt ist in den USA historisch bedingt immer noch sehr heterogen, da noch in weiten Teilen Apple-Systeme eingesetzt werden. In Deutschland finden langsam Linux-Systeme Verwendung in der Schule, da diese eindeutige Kostenvorteile bieten.

Die heterogene Umgebung erfordert ein portables System: Squeak selber muß auf allen Systemen lauffähig sein, aber auch die von den Schülern geschaffenen Inhalte sollen sowohl Zuhause als auch auf dem Schulrechner benutzbar sein.

2.4.3 Erwartetes Verhalten

Das Verhalten eines Squeak-Programmes sollte dem entsprechen, was der Benutzer erwartet. Ein System zeigt *erwartetes Verhalten* (engl. *expected behaviour*, siehe [27]), wenn es sich so verhält, wie es im Quellcode vorgegeben wurde.

Ein Interpreter, der den Quellcode direkt interpretiert, zeigt ein solches Verhalten. Falls man optimierende Transformationen zur Leistungssteigerung einsetzen möchte, müssen diese semantikerhaltend sein. Kapitel 3 zeigt, daß dies in der aktuellen Implementierung nicht der Fall ist.

2.4.4 Performanz

Die bis jetzt genannten Anforderungen an die Squeak-Implementierung (dynamische Metaprogrammierung, erwartetes Verhalten und Portabilität) haben alle einen entscheidenden Nachteil: Jede naive Implementierung ist sehr ineffizient.

Praktisch einsatzfähig ist das System nur, wenn es trotz weitgehender Verwirklichung dieser Anforderungen eine ausreichende Gesamtperformanz liefert.

Gerade für Multimediasysteme wird in vielen Bereichen (etwa Grafik oder Musik) ein sehr effizientes System benötigt. Dies ist besonders dann der Fall, wenn man das System auch auf portablen Endgeräten einsetzen möchte, bei denen die Leistungsfähigkeit durch einen geringen Stromverbrauch begrenzt wird.

2.5 Zusammenfassung

Squeak bietet eine Plattform, mit der Kinder spielerisch mit Animationen und Simulationen experimentieren können. Ein solches System stellt besondere technische Anforderungen an die Implementierung, besonders in den Bereichen dynamischer Metaprogrammierung, erwartetem Verhalten und Performanz.

Das nächste Kapitel zeigt, wie das aktuelle System diese Anforderungen verwirklicht und welche Probleme diese Implementierung aufweist.

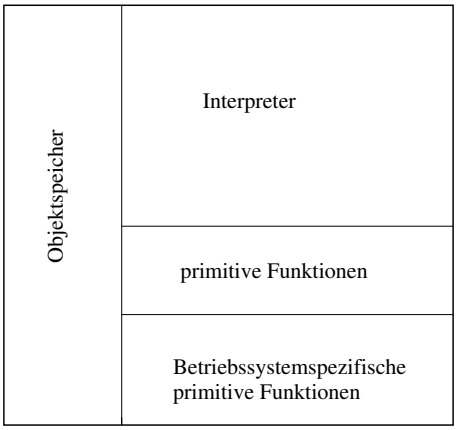

Abbildung 5: Struktur der virtuellen Maschine

3 Die aktuelle virtuelle Maschine

Dieses Kapitel stellt das aktuelle Squeak-System vor und diskutiert, wie es die in Abschnitt 2.4 aufgestellten Anforderungen erfüllt und welche Probleme sich aus dem Entwurf ergeben.

3.1 Die Implementierung von Squeak

Squeak versucht, möglichst wenige der von dem Betriebsystem angebotenen Dienste zu nutzen. Dadurch vereinfacht sich die Portierung und Squeak bleibt über alle Systeme kompatibel.

Das Squeak-System enthält eine umfangreiche Klassenbibliothek (ca. 1700 Klassen). Diese stellt neben Grundlegendem (etwa Datentypen wie Integer und Float oder einem mächtigen Behälterrahmenwerk (engl. *collection hierarchy*) auch Klassen für graphische Oberflächen, Netzwerkprotokolle und Multimedia zur Verfügung.

3.1.1 Die virtuelle Maschine

Die Implementierung der virtuellen Maschine hält sich weitgehend an die Referenzimplementierung der Smalltalk 80 VM in [7], die genauen Unterschiede diskutiert [12].

Die virtuelle Maschine von Squeak gliedert sich in drei Teile (siehe Abbildung 5):

- Objektspeicher (*ObjectMemory*)
- Interpreter
- Primitive Funktionen für elementare Aufgaben

Die folgenden Abschnitte stellen diese Teile kurz genauer vor. Eine ausführliche Beschreibung findet sich in [24].

3.1.2 ObjectMemory

Der *ObjectMemory* ist das Speicherverwaltungssystem der virtuellen Maschine. Er implementiert das Objektmodell und stellt eine automatische Speicherbereinigung bereit.

Die Speicherbereinigung verwendet *Generation Scavanging* [26]. Das Verfahren macht Gebrauch von der Beobachtung, daß viele Objekte nur für eine sehr kurze Zeit nach ihrer Erzeugung benötigt werden.

Der Speicher der virtuellen Maschine wird in mindestens zwei Bereiche eingeteilt. Im Falle von Squeak existiert ein kleiner Bereich, in dem neue Objekte erzeugt werden (der sog. *Newspace*) und ein großer Speicherbereich für langlebige Objekte (*Oldspace*).

Da der *Newspace* nur wenige 100 Kilobyte umfaßt, dauert eine Speicherbereinigung nur eine kurze Zeit. Sie kann also sehr häufig durchgeführt werden. Da ein sehr großer Anteil der Objekte nur eine kurze Lebensdauer hat, überleben nur wenige Objekte die Bereinigung des *Newspace*.

Mit der Zeit werden aber auch langlebige Objekte erzeugt. Wenn die Anzahl der Objekte nach der Bereinigung des *Newspace* einen Schwellwert überschreitet, so werden alle diese Objekte dem *Oldspace* zugeordnet.

Der *Oldspace* wird nur sehr selten bereinigt. Solch ein Fall tritt z. B. auf, wenn nicht mehr genug Speicher für neue Objekte vorhanden ist.

Die Speicherverwaltung von Squeak wird in [12] ausführlich beschrieben.

3.1.3 Interpreter

Die virtuelle Maschine interpretiert einen kellerbasierten Bytecode, der im Hinblick auf Kompaktheit und eine relativ effiziente Interpretierbarkeit entwickelt wurde.

Kompakt ist der Bytecode z. B. dadurch, daß die virtuelle Maschine extra Bytecodes für häufig vorkommende Folgen von einfachen Operationen bereitstellt.

Effizient interpretierbar wird der Bytecode durch Sprungbefehle. Diese erlauben dem Smalltalk Übersetzer, die Methoden der Kontrollstrukturen wie *ifTrue:* oder *whileTrue:* offen einzubauen.

Die insgesamt 248 verschiedenen Bytecodes lassen sich in folgende Gruppen gliedern:

Keller Manipulation: Diese Bytecodes ändern den Keller der virtuellen Maschine: z. B. push bzw. pop von lokalen Variablen.

Sprünge: Einige wenige Methoden werden offen eingebaut (z. B. `ifTrue:`). Dazu werden Sprungbefehle benötigt.

Nachrichten versenden: Methodenaufrufe. Neben dem allgemeinen `send` Bytecode existieren einige extra Bytecodes für häufig vorkommende Aufrufe (z. B. `sendAdd`).

Rücksprünge Rücksprung aus Methoden und Blöcken.

Der Bytecode kennt also keine primitive Typen. Es werden lediglich Nachrichten an Objekte gesendet.

3.1.4 Primitive Funktionen

Die primitiven Funktionen erlauben es, Code aufzurufen, der direkt in der virtuellen Maschine implementiert ist. Es gibt zwei Gründe, auf primitive Funktionen zurückzugreifen: Zum einen Operationen, die nicht in Squeak implementiert werden können (weil sie z. B. das Typsystem umgehen oder Betriebssystemroutinen aufrufen). Zum anderen werden primitive Funktionen zur Steigerung der Geschwindigkeit eingesetzt.

Die vorhandenen primitiven Funktionen lassen sich in folgende Klassen einteilen:

Ein-/Ausgabe: Tastatur, Maus, Bildschirm über plattformabhängige C-Funktionen

Arithmetik: Grundlegende arithmetische Operationen auf Integer/Float.

Erzeugen neuer Objekte: Aufrufe der Speicherverwaltung.

Prozeßmanipulation: Prozesse starten, anhalten und beenden.

Ausführungskontrolle: Erzeugung neuer Ausführungskontexte von Methoden und Blöcken.

Performanz: Primitive Funktionen zur Beschleunigung des Systems.

3.1.5 Portierung

Die virtuelle Maschine ist zu einem großen Teil in *Slang* implementiert, dies ist ein vereinfachter Smalltalk-Dialekt, der sehr einfach nach C übersetzt werden kann.

Neben dem so implementierten Interpreter, der automatischen Speicherbereinigung (*Garbage Collector*) und den primitiven Funktionen werden für eine vollständige virtuelle Maschine noch einige systemnahe Funktionen zur Ein- und Ausgabe (Netzwerk, Graphik, Maus, Tastatur) benötigt, insgesamt nur ca. 4000 Zeilen C-Code.

Einzig dieser überschaubare Teil der virtuellen Maschine ist spezifisch für das jeweils eingesetzte Betriebssystem, der weitaus größte Teil ist portabel.

Da nur sehr wenig Code für ein neues System implementiert werden muß, ist Squeak mit sehr geringem Aufwand zu portieren (siehe [21]). Anfang 2001 lief Squeak auf ca. 20 Systemen, darunter alle Windows-Systeme, Unix und MacOS. Squeak wurde auch schon erfolgreich auf PDAs portiert, wie z. B. Sharp, Zaurus und PocketPC.

3.2 Bewertung

Schüler und Lehrer setzen die in diesem Kapitel vorgestelle virtuelle Maschine auf verschiedenen Plattformen erfolgreich ein. Das System ist auf aktueller (und damit schneller) Hardware ausreichend performant für den praktischen Einsatz. Die aktuelle virtuelle Maschine erfüllt die Kriterien aus Abschnitt 2 weitgehend: Das System ist hoch portabel, ausreichend schnell

und realisiert dynamische Metaprogrammierung durch den Einsatz eines Interpreters.

Auf den ersten Blick scheinen damit die Zielkriterien aus Kapitel 2.4 erfüllt, aber bei einer genaueren Betrachtung erkennt man viele Probleme. Das Performanzproblem des Interpreters diskutiert Abschnitt 3.2.1. Abschnitt 3.2.2 erläutert, warum das System durch nicht-transparente Optimierungen auch kein *erwartetes Verhalten* garantiert.

3.2.1 Performanz

Die Entscheidung, ein Multimedia-System mittels einer interpretierten Sprache zu implementieren, ist ungewöhnlich. Gerade bei Multimedia-Systemen wird eine hohe Leistung benötigt, da solche Systeme Graphik und Klang in Echtzeit verarbeiten müssen.

Alle kritischen Teile des Systems wurden daher mittels primitiver Funktionen als Teil der virtuellen Maschine implementiert. Der Einsatz von primitiven Funktionen zur Geschwindigkeitssteigerung hat jedoch viele Nachteile. Primitive Funktionen liegen als Binärcode vor und sind damit nicht so einfach änderbar wie Smalltalk-Code: Die primitiven Funktionen sind Teil der virtuellen Maschine, nicht Teil der Klassenbibliothek.

Für diese Teile gilt also keines der Zielkriterien außer einem: Sie sind schnell. Aber es ist keine Metaprogrammierung möglich, man kann den Code nicht anhalten und nicht modifizieren, ohne eine neue virtuelle Maschine zu erstellen.

Das gesamte System ist dadurch wenig flexibel.

3.2.2 Erwartetes Verhalten

Wie in Abschnitt 3.1.3 erläutert, verwendet Squeak einen relativ komplexen Satz von Bytecodes. Der Bytecode enthält z. B. Sprungbefehle für die offene Kodierung von Methoden. Außerdem gibt es eine Menge von Spezialbefehlen z. B. für arithmetische Operationen.

Das Problem ist, daß diese Optimierungen nicht transparent eingesetzt werden: Der Smalltalk-nach-Bytecode Übersetzer (siehe [2]) implementiert offenen Einbau von Methoden für wenige Spezialfälle (z. B. `ifTrue:`). Diese Spezialfälle sind hart im Übersetzer kodiert, daher wird beispielsweise jede

Änderung der Methode `False>>ifTrue:` (etwa zum Hinzufügen von Ausgaben zur Fehlersuche) niemals wirksam.

Da die Optimierung nur für wenige, bereits existierende Spezialfälle erfolgt, können Kontrollstrukturen, die der Benutzer hinzufügt, nicht optimiert werden.

3.3 Zusammenfassung

Das aktuelle Squeak-System versucht die Anforderungen aus Kapitel 2.4 durch den Einsatz eines Bytecodeinterpreters zu erfüllen.

Da ein Interpreter aber zu langsam ist, führt der Smalltalk-Übersetzer Optimierungen durch, die für den Benutzer nicht transparent sind. Diese Teile des Systems erfüllen die Anforderungen nicht.

Das im folgenden Kapitel 4 beschriebene J3 Projekt implementiert einen Laufzeitübersetzer, der das Problem der schlechten Performanz des Interpreters angeht.

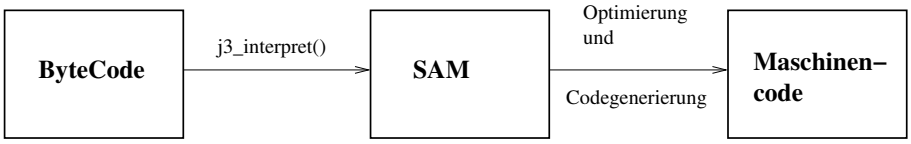

Abbildung 6: Architektur J3

4 J3: Ein Squeak-Laufzeitübersetzer

Abschnitt 3.2.1 hat die schlechte Performanz der interpretierenden virtuellen Maschine als Problem identifiziert. Dieses Problem will der J3-Laufzeitübersetzer [19] durch Übersetzen des Bytecodes in Maschinencode zur Programmlaufzeit beheben. Die Arbeitweise und Eigenschaften von J3 beschreibt dieses Kapitel.

4.1 Projektziele

Das Ziel des J3 Projektes ist, die Ausführungsgeschwindigkeit von Squeak durch Erzeugen von Maschinencode zur Laufzeit zu erhöhen. Der erzeugte Code soll eine hohe Qualität haben (d. h. er soll performant sein). Wichtig ist, daß sich der Compiler relativ einfach auf neue Architekturen portieren läßt.

Der Squeak J3 wurde von Ian Piumarta (INRIA Rocquencourt) für PowerPC Systeme entwickelt und von mir im Zuge eines Praktikums bei *Walt Disney Imagineering* auf Intel x86 Systeme portiert.

4.2 Architektur

Der Squeak J3 ersetzt den Bytecode-Interpreter der normalen virtuellen Maschine. Viele andere Teile, etwa die primitiven Funktionen und die automatische Speicherbereinigung (den sog. *ObjectMemory*) kann J3 wiederverwenden.

Der Übersetzer besteht aus drei Teilen: Transformation in eine Zwischenrepräsentation, Optimierung und schließlich Codegenerierung (siehe Abbildung 6).

4.2.1 Squeak Abstract Machine (SAM)

Im ersten Schritt transformiert J3 die Bytecodes (siehe Kapitel 3.1.3) in eine Zwischenrepräsentation (siehe Abbildung 6).

Diese SAM genannten Operationen sind ein stark vereinfachter Bytecode, statt 248 verschiedenen Bytecodes gibt es nur 22 grundlegende SAM-Instruktionen, dazu kommen noch 12 Instruktionen (Add, Sub, etc), die einzig der besseren Lesbarkeit und dem Verständnis sowohl des generierten SAM-Codes, als auch der Implementierung des Optimierers dienen.

4.2.2 Optimierung und Codegenerierung

Der Laufzeitübersetzer führt auf Basis des SAM-Zwischencodes einige einfache Optimierungen durch. Der Optimierer bringt die im Code vorkommenden Sprungbefehle in in eine Form, die es später dem Codegenerator ermöglichen, besseren Code zu erzeugen.

Ein Beispiel ist ein vom Smalltalk-Übersetzer erzeugter bedingter Rücksprung:

```
label: ....
                    jumpTrue: label
                    ...
```

Dieser läßt sich umformen in einen bedingten Vorwärtssprung und einen unbedingten Rücksprung, die die Sprungvorhersage korrekt erkennen kann:

```
label:          ....
                    jumpFalse: label2
                    jumpTo: label
label2:         ...
```

Anschließend findet eine weitere einfache Optimierung während der Codegenerierung statt. Verwendet wird eine Form der Codegenerierung mit Maschinensimulation (*Delayed Code Generation* (DCG) siehe [22]).

Die Maschinensimulation simuliert eine Kellermaschine, alle Operanden der Maschine werden mittels Deskriptoren beschrieben.

Mit Hilfe eines simulierten Kellers kann der Codegenerator so zur Übersetzungzeit konstante Ausdrücke berechnen.

Zur Generierung von Maschinenbefehlen werden dem Codegenerator Funktionen bzw. Makros bereitgestellt, die von der Plattform abstrahieren und eine prozessorunabhängige assemblernahe Sprache darstellen. Diese Routinen müssen für jede Architektur implementiert werden. Als Laufzeitassembler wird *CCG* eingesetzt, der im nächsten Abschnitt näher beschrieben wird.

Eine ausführlichere Beschreibung des J3-Übersetzers findet sich in [19].

4.2.3 Der Laufzeitassembler CCG

Der Codegenerator muß den erzeugten Maschinencode direkt im Speicher erzeugen. Außerdem wird der Codegenerator bei der ersten Ausführung einer Methode zur Laufzeit aufgerufen, die Codegenerierung muß daher auch sehr schnell geschehen. Der Laufzeitübersetzer kann also nicht, wie bei statischen Übersetzern üblich, den vom Betriebsystem bereitgestellten Assembler nutzen.

Der Squeak J3 verwendet als Laufzeitassembler den von Ian Piumarta entwickelten `CCG` (siehe [20]). Dies ist ein Präprozessor, der es erlaubt, Assembleranweisungen direkt in C (bzw. C++) Code einzubetten. Die Assembleranweisungen werden dann mit Hilfe des `CCG`-Präprozessors durch ANSI-C Präprozessormakros ersetzt, die der C-Übersetzer als konstant erkennt und zu einfachen und effizienten Zuweisungen umformt.

Abbildung 7 zeigt den Einsatz von CCG im Überblick: Ziel ist ein Programm, das seinen eigenen Code im Speicher manipuliert.

Dazu wird ein C-Programm mit CCG-Anweisungen erstellt (Datei.ccg in Abbildung 7). Der CCG-Präprozessor erzeugt daraus ein C-Programm (Datei.c), das Makro-Definitionen des CCG-Systems verwendet. Der C-Präprozessor und C-Übersetzer erzeugen dann das Endprodukt: Ein ausführbares Programm, das selber zur Laufzeit neuen ausführbaren Maschinencode erzeugen kann (Datei.o).

4.2.4 Beispiel

Das folgende Beispiel verdeutlicht die Funktionsweise des `CCG`-Laufzeitassemblers.

Gegeben sei die Assembleranweisung:

```
#[ movl $42, %eax ]#
```

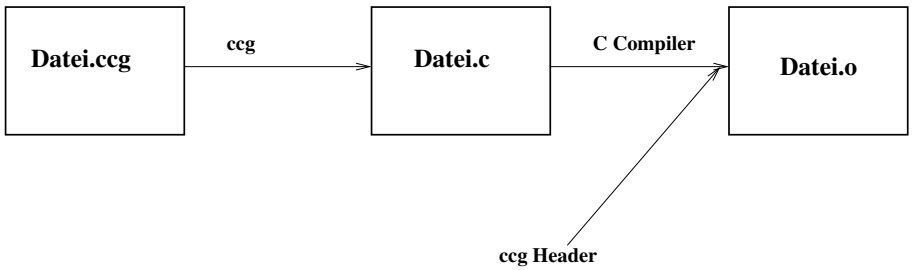

Abbildung 7: Der CCG Präprozessor

Der Präprozessor `CCG` ersetzt diese innerhalb der eckigen Klammern durch die entsprechenden Makrobefehle für den C Präprozessor:

```
MOVLir(42,_EAX);
```

Das Makro `MOVLir` wird in den vom *CCG*-System [20] bereitgestellten C-Header Dateien definiert. Nach Optimierung entspricht dies folgendem C-Code:

```
*(char *)(someAdress)     = 0xB8; /* insn: move imm32 -> %eax */
*(char *)(someAdress + 1) = 0x2A; /* constant: 42 */
```

Der C-Übersetzer kann den Code also so stark optimieren, daß er zur Erzeugung von Maschinenbefehlen zur Laufzeit nur Konstanten in den Speicher schreiben muß.

4.3 Performanz

Die erreichte Steigerung der Ausführungsgeschwindigkeit ist trotz des einfachen Codegenerators schon sehr zufriedenstellend.

Zur Messung der Leistungsteigerung der virtuellen Maschine enthält die Squeak-Klassenbibliothek zwei verschiedene Benchmarks. Einer ermittelt die Leistung der virtuellen Maschine, der zweite zeigt die Leistung des Gesamtsystems.

Tabelle 1: Vergleich *tinyBenchmarks* PowerPC G4 (450Mhz)

	Bytecodeausführung	Methodenaufrufe
Interpreter:	19393939 bytecodes/sec	880528 sends/sec
J3-Übersetzer:	52159739 bytecodes/sec	6575704 sends/sec
Beschleunigungsfaktor:	2.7	7.47

4.3.1 Messung der Interpreterleistung

Will man die Ausführungsgeschwindigkeit einer virtuellen Maschine beurteilen, so interessiert zum einen, wie schnell sie den Bytecode verarbeitet, also wieviele Bytecodes pro Sekunde eines realen, kleinen Programms sie ausführt. Dies der erste Meßwert des *tinyBenchmarks* (siehe erste Spalte in Tabelle 1).

Zum anderen ist aber auch die Geschwindigkeit von Methodenaufrufen entscheidend: Der Smalltalk-Übersetzer baut nur wenige Methoden offen ein: Einzig einige bestehende Sprachkonstrukte wie Schleifen und If-Abfragen optimiert er zu Sprüngen. Jeden anderen Methodenaufruf kann er nicht optimieren, ein effizienter Aufruf von Methoden (*send*) ist daher sehr wichtig. Der zweite Meßwert des *tinyBenchmarks* gibt daher die Anzahl von Methodenaufrufen pro Sekunde an (siehe zweite Spalte in Tabelle 1).

Der Quellcode des Benchmarks findet sich in Anhang A.

Der *tinyBenchmark* gibt die Leistungsfähigkeit der virtuellen Maschine wieder, aber er testet lediglich die Ausführungsgeschwindigkeit unter der Voraussetzung, daß ausschließlich Bytecode ausgeführt wird. Dies ist aber in einem realen Programm niemals der Fall: Alle zeitkritischen Teile des Systems sind als primitive Funktionen implementiert. Ein Squeak-Programm verbringt einen großen Teil der Zeit mit der Ausführung dieser primitiven Funktionen, die der Übersetzer nicht beschleunigen kann. Die Beschleunigung des Gesamtsystems durch eine schnellere virtuelle Maschine reduziert sich daher um die Menge der primitiven Funktionen.

4.3.2 Leistung des Gesamtsystems

Die Leistung des Gesamtsystems messen die sogenannten *macroBenchmarks*, die eher die in realen Programmen zu erwartende Geschwindigkeitssteigerung wiedergeben. Bei den *macroBenchmarks* werden Squeak-Programme

Tabelle 2: Vergleich *macroBenchmarks* PowerPC G4 (450Mhz)

Benchmark	Decompile	morphic tiles	slang	ctxstpSim
Interpreter	44245	56616	71890	26059
Übersetzer	19835	34206	38587	12364
Beschleunigungsfaktor	2.2	1.6	1.8	2.1

Benchmark	InterprSim	browsers	FreeCel
Interpreter	6987	9402	7718
Übersetzer	3740	6709	6930
Beschleunigungsfaktor	1.8	1.4	1.1

als Benchmark verwendet, die auch im normalen Alltag auftreten. Dazu zählen unter anderem das Übersetzen von Smalltalk-Code, das Generieren von C-Code für den Interpreter und einige weitere alltägliche Aktivitäten, wie z. B. das Öffnen mehrerer Editoren.

Tabelle 2 zeigt, daß die bei diesem Benchmark beobachtete Beschleunigung weit hinter den *tinyBenchmarks* zurückbleibt.

4.4 Portabilität

Ein Ziel der Entwicklung des J3 war es, den Laufzeitübersetzer mit relativ geringem Aufwand auf neue Architekturen portieren zu können. Es wurde zuerst ein Backend für PowerPC Prozessoren implementiert.

Zum Test der Portierbarkeit wurde das J3-System auf Intel x86 portiert. Das Ergebnis ist positiv: Der Aufwand betrug ca. 4 Monate für einen Entwickler, der weder mit Laufzeitübersetzern im allgemeinen noch J3 im besonderen vertraut war.

Dabei ist aber zu beachten, daß der Laufzeitassembler *CCG* schon auf Intel Systemen eingesetzt wurde. Eine zusätzliche Portierung des Assembler, wie sie z. B. für Intel StrongARM nötig wäre, würde den Aufwand einer Portierung mindestens verdoppeln.

4.5 Bewertung

Das J3-Projekt hat gezeigt, daß auch ein relativ einfacher und leicht portierbarer Laufzeitübersetzer die Ausführungsgeschwindigkeit steigern kann.

Die erreichte Beschleunigung (ca. Faktor 5-7) wird es möglich machen, in einigen Fällen auf den Einsatz von primitiven Funktionen zu verzichten. Dazu müßte der Laufzeitübersetzer aber auf allen System verfügbar sein.

Hier zeigt sich ein Problem des Ansatzes: Die Portierung eines Übersetzers ist, auch im Fall von J3, doch immer noch viel aufwendiger als die des Interpreters (der einfach nur mit Hilfe eines C-Compilers übersetzt werden muß).

Daher wird der Laufzeitübersetzer den Interpreter nie ersetzen können: Es wird immer zwei virtuelle Maschinen geben, die fast vollständig getrennt voneinander gewartet und weiterentwickelt werden müssen.

Ein weiteres Problem ist die Implementierungssprache C++: Die Entwicklungsumgebung ist, besonders bei der Fehlersuche, bei weitem nicht so mächtig wie Squeak. Außerdem hat sich gezeigt, daß es sehr viel schwieriger für die anderen Squeak-Entwickler ist, mit dem J3-System zu experimentieren als mit dem Squeak-Interpreter. Dies liegt natürlich zum einen an der viel höheren Komplexität eines Übersetzers, darüberhinaus spielt aber auch die Implementierungssprache eine Rolle: Ein in Squeak implementiertes System ist für die Entwickler viel einfacher zu verstehen und zu verändern.

4.6 Zusammenfassung

Durch einen Laufzeitübersetzer kann, trotz einfacher Architektur, eine entscheidende Beschleunigung erreicht werden. Der J3-Laufzeitübersetzer hat aber einzig die Beschleunigung der Ausführung des existierenden Squeak-Bytecodes zum Ziel. Die Probleme, die dieser Bytecode mit sich bringt (siehe Kapitel 3.2.2), behebt er daher nicht.

Eine Architektur, die auch diese Probleme lösen möchte, muß daher einen anderen Weg gehen. Kapitel 5 stellt einen entsprechenden Entwurf für eine solche Architektur (SX *Squeak eXtreme*) vor.

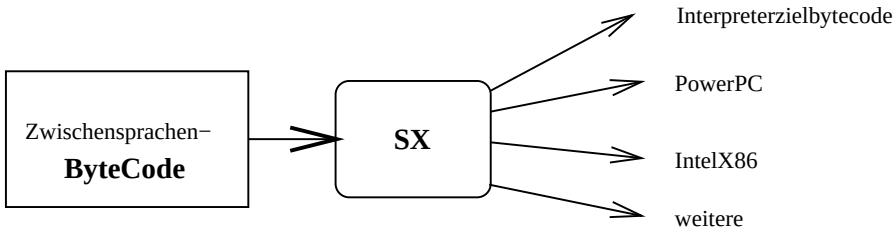

Abbildung 8: Architektur SX

5 Entwurf SX (*Squeak eXtreme*)

5.1 Überblick

In diesem Kapitel stelle ich meinen neuen Entwurf für die Squeak virtuelle Maschine vor. Dieser versucht alle Anforderungen aus Kapitel 2.4 zu erfüllen.

5.2 Ziele

Die Ziele, an denen sich der neue Entwurf messen lassen muß, sind also

- Steigerung der Ausführungsgeschwindigkeit (mindestens so wie J3).
- Einfache Portierung.
- Einfacheres Experimentieren durch Implementierung in Smalltalk
- Verbesserung des *erwarteten Verhaltens*.

5.3 Architekturüberlegungen

Die grundlegende Idee der SX-Architektur ist, die Vorteile, die ein Laufzeitübersetzer bietet, für weitere Verbesserungen auszunutzen. Das J3-Projekt hat gezeigt, daß ein Laufzeitübersetzer sinnvoll ist. J3 war aber nicht vollständig ins System integriert: Das Ziel war, die Ausführung des bestehenden Bytecodes zu beschleunigen.

Eine sehr problematische Eigenschaft des Squeak-Systems sind die Optimierungen für Spezialfälle, die der Smalltalk-nach-Bytecode-Übersetzer

durchführt. Wenn ein Laufzeitübersetzer vorhanden ist, kann man auf diese Optimierungen aber vollständig verzichten: Es existiert dann ja schon eine Optimierung innerhalb des Laufzeitübersetzers. Für Optimierungen auf Bytecode-Ebene besteht daher kein Grund, sobald der Laufzeitübersetzer zur Anwendung kommt. Da der J3-Laufzeitübersetzer nur Maschinencode erzeugen kann, kann man ihn nur einsetzen, wenn die Zielmaschine unterstützt wird. In allen anderen Fällen muß man auf den Interpreter zurückgreifen.

Aber ein Übersetzer kann ja nicht nur Code für existierende Maschinen, sondern auch für die virtuelle Maschine erzeugen (siehe Abbildung 8). In diesem Fall wären die Optimierungen des Smalltalk-nach-Bytecode-Übersetzers nicht mehr nötig. Die Optimierungen finden dann vollständig transparent im Optimierer des Laufzeitübersetzers statt.

Der J3-Übersetzer war in C++ implementiert. Dies hat einige Nachteile gezeigt: Zur Entwicklung kann man die mächtige Squeak-Entwicklungsumgebung nicht verwenden, die Fehlersuche ist daher sehr schwierig. SX soll deshalb in Smalltalk, nicht C++, implementiert sein.

Der Hauptgrund für die Entscheidung, C++ als Implementierungssprache für J3 zu verwenden, war die benötigte Geschwindigkeit des Laufzeitübersetzers. Der J3 generiert Code nur zur Laufzeit, direkt vor Ausführung einer Methode. Dagegen soll der SX-Entwurf prüfen, ob man nicht den einmal übersetzten Code speichern kann. Durch einen inkrementellen Ansatz könnte so der Einfluß der Geschwindigkeit des Laufzeitübersetzers auf die interaktiven Eigenschaften gemindert werden.

Die drei zentralen Eigenschaften des SX-Entwurfs sind also:

- Bytecode ist eine optionale Zielsprache, nicht Zwischensprache.
- Das System sollte in Squeak implementiert sein.
- Laufzeitübersetzung nicht *Just in Time*

Im folgenden werden diese Ideen weiter erläutert und gezeigt, welche weiteren Vorteile sich daraus ergeben.

5.3.1 Bytecode als Zielsprache

Der Ansatz von SX ist, die virtuelle Maschine genau wie auch die realen Maschinen als Zielsystem des Laufzeitübersetzers zu behandeln.

Daraus ergeben sich einige interessante Vorteile:

- Als Zwischensprache muß kein Bytecode mehr verwendet werden.
- Es wird möglich, den generierten Bytecode, wie auch den Binärcode, vom Laufzeitübersetzer optimieren zu lassen.

Die Verwendung von Bytecode als Zwischensprache bei Smalltalk und Java ist für eine virtuelle Maschine mit Laufzeitübersetzer eher negativ: Kellerbasierter Bytecode ist denkbar schlecht geeignet als Zwischensprache für einen Übersetzer. Daher ist der erste Schritt aller Laufzeitübersetzer, aus dem Bytecode wieder eine für Optimierungen geeignete Zwischenrepräsentation zu gewinnen.

Wenn der Laufzeitübersetzer auch Code für die virtuelle Maschine erzeugen kann (für den Fall, daß der Prozessor der Zielmaschine nicht unterstützt wird), dann gibt es keinen Grund, einen komplexen, einfach zu interpretierenden aber schwierig zu übersetzenden Bytecode als Zwischensprache zu verwenden. Daher verwendet das SX-System einen sehr stark vereinfachtem Bytecode, der möglichst nahe an der Sprache Smalltalk ist. Insbesondere enthält er keine Optimierungen. Alle Optimierungen führt der SX-Laufzeitübersetzer auf einer SSA-basierten Zwischendarstellung durch, die er aus dem Eingabebytecode erzeugt.

Alternativ kann man den abstrakten Syntaxbaum direkt als Zwischensprache verwenden. Franz zeigt in [6], daß ein komprimierter AST sogar kompakter ist als typische bytecodebasierte Codierungen. Gegen die Verwendung eines AST als Zwischensprache spricht, daß man in diesem Fall die Werzeuge der Squeak-Entwicklungsumgebung wie z. B. den Debugger nicht mit geringem Aufwand anpassen kann.

Neben einer besseren Zwischensprache erlaubt dieser Entwurf auch, den Bytecode durch den Laufzeitübersetzer optimieren zu lassen. Fast alle typischen Optimierungen, die ein Übersetzer durchführt, sollten auch dann eine Geschwindigkeitsteigerung zur Folge haben, wenn Code für einen Interpreter generiert wird. Den Squeak-Bytecode und die virtuelle Maschine muß man in diesem Fall aber anpassen. Man kann beispielsweise vorkommende Aufrufziele im Bytecodestrom direkt puffern, um diese statistische Information bei der Codegenerierung ausnutzen zu können.

In einem solchen System besteht insbesondere kein Grund mehr für die in Kapitel 3.2.2 kritisierten Optimierungen des Bytecodes: Diese kann der Optimierer des Laufzeitübersetzers transparent durchführen.

5.3.2 Übersetzer in Smalltalk

Die aktuelle virtuelle Maschine ist in einer Teilmenge von Smalltalk (*Slang*) implementiert, die nach C übersetzt wird. Diese Lösung erlaubt es, die virtuelle Maschine mit der mächtigen Entwicklungsumgebung des Squeak-Systems zu entwickeln. Die hat sich besonders bei der Fehlerbehebung als sehr hilfreich herausgestellt.

Die Sprache *Slang* ist aber so stark vereinfacht (keine Polymorphie, keine Objekte), daß man viele der Vorteile von Smalltalk nicht nutzen kann.

Eine sehr interessante Möglichkeit wäre, den Übersetzer direkt, sozusagen *reflexiv* in Squeak zu implementieren: Nach dem Start interpretiert ein einfacher Interpreter den Laufzeitübersetzer, während dieser sich selber übersetzt. Nach dieser ineffizienten Startphase würde der Interpreter nicht mehr eingesetzt, sondern jetzt kann der nun in Maschinensprache vorliegende Laufzeitübersetzer die Ausführung vollständig übernehmen.

Der experimentelle Java JIT-Compiler *OpenJIT* hat gezeigt, das ein solcher Entwurf tatsächlich praktisch zu verwirklichen ist (siehe [18] und [17]), daher wird das SX-System diesen Ansatz verfolgen.

5.3.3 Laufzeitübersetzer ohne JIT

Die meisten Laufzeitübersetzer speichern den generierten Code nicht dauerhaft, sondern verwenden einen sehr kleinen Cache (meist weniger als 1MB), in dem sie den Maschinencode nach einer LRU-Strategie verwalten. Dieser Cache wird insbesondere nicht auf dem Hintergrundspeicher gesichert, der Code muß also mindestens bei jedem Programmstart neu generiert werden.

Wenn eine Methode das erste mal aufgerufen wird (oder wenn sie aus dem Cache gelöscht wurde), erzeugt ein schneller Übersetzer *just in time* den entsprechenden Code. Dabei wird die Ausführung des Programms angehalten.

Das entscheidende Entwurfskriterium für Laufzeitübersetzer ist also einzig die Geschwindigkeit, mit der sie Code generieren können. Alles andere muß sich diesem Ziel unterordnen. Es ist klar, daß Geschwindigkeit bei der Übersetzung meist auf Kosten der Geschwindigkeit des erzeugten Codes erkauft wird: Ein Just-In-Time-Übersetzer kann nur triviale Optimierungen durchführen.

Will man aufwendige Optimierungen durchführen, so muß man auf ein zweistufiges System zurückgreifen: Ein Compiler erzeugt schnell schlechten Code,

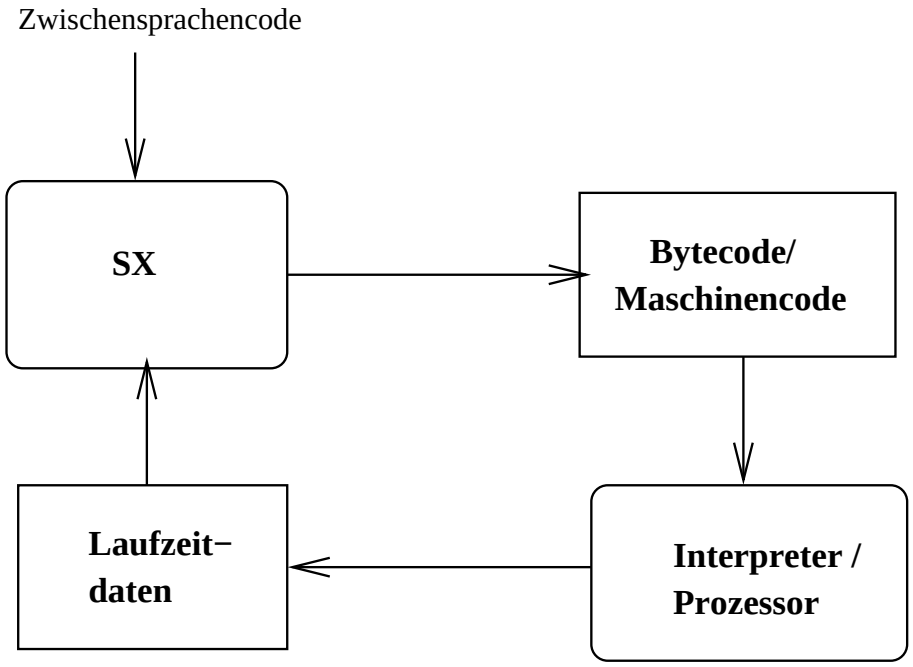

Abbildung 9: Übersetzung zur Laufzeit

ein zweiter kann auf Anforderung solche Stellen des Programms mit voller Optimierung im Hintergrund neu übersetzen, bei denen es sich lohnt. Dies werden also die Stellen sein, die häufig ausgeführt werden (sog. *Hot Spots*). Bei einer solchen Neuübersetzung kann der Übersetzer zudem Informationen zur Optimierung verwenden, die erst zur Laufzeit verfügbar sind, z. B. statistische Informationen über Aufrufziele von polymorphen Methodenaufrufen (siehe Abbildung 9).

Diese Idee wurde vom Self-Projekt (siehe [3] und [25]) als erstes implementiert und wird heute bei einigen modernen Java-Systemen eingesetzt (z. B. Sun HotSpot [4]).

Die IBM *Jalapeno/Jikes* Research VM (siehe [1]) geht noch einen Schritt weiter und speichert den erzeugten Code persistent: Selbst nach einem Neustart ist das Programm sofort ohne Übersetzungsphase einsetzbar. Da das SX-System in Smalltalk implementiert wird, sind alle erzeugten Datenstrukturen Smalltalk-Objekte und werden daher vom System persistent gespei-

chert.

Ein solcher Entwurf erlaubt es, eine virtuelle Maschine zu implementieren, die alle Vorteile eines statischen Übersetzers mit denen eines Laufzeitübersetzers verbindet. Insbesondere kann man aufwendige Optimierungen durchführen.

5.4 Weitere Konsequenzen

Der in diesem Kapitel vorgestelle Entwurf eines Squeak-Laufzeitübersetzers könnte noch einige weitere interessante Vorteile haben, die nicht direkt mit der virtuellen Maschine in Zusammenhang stehen. Interessant dabei sind zwei Aspekte: Die Wiederverwendung des Übersetzerrahmenwerks für einen statischen Übersetzer und die Auswirkungen einer AST-basierten Zwischensprache auf die Entwicklungsumgebung.

5.4.1 Ein statischer Smalltalk Übersetzer

Mit Hilfe des Übersetzerrahmenwerks kann man sehr einfach einen statischen Smalltalk-Übersetzer erstellen. Diesen kann man z. B. dazu einsetzen, einen einfachen Interpreter zu erstellen, der zum *Bootstrapping* des Systems benötigt wird.

5.4.2 Anbindung von Werkzeugen über die Zwischensprache

Die Squeak-Kinder-Skriptsprache *Etoys* erlaubt es, jede Methode des Systems als graphisches Skript darzustellen und zu manipulieren. Für diese Funktionalität erstellt der Zerteiler des Smalltalk-nach-Bytecode-Übersetzers eine AST-Darstellung. Dieser Mechanismus kann durch eine AST-basierte Zwischensprache stark vereinfacht und besser ins System integriert werden.

Auch viele Werkzeuge wie z. B. der *Refactoring Browser* [23] arbeiten auf Basis einer Baumdarstellung. Auch hier könnte die AST-Zwischendarstellung hilfreich sein.

6 Zusammenfassung

In dieser Studienarbeit habe ich die Anforderungen dargestellt, die bei der Implementierung des Kinderprogrammiersystems Squeak auftreten. Nach einem kurzen Überblick über die aktuelle interpreterbasierte Implementierung habe ich gezeigt, daß diese die Anforderungen nur bedingt erfüllt.

Zwei Verbesserungen wurden diskutiert: Der Laufzeitübersetzer J3 und mein darauf aufbauender, verbesserter Entwurf (SX).

Der von mir im Rahmen eines Auslandspraktikums auf Intel-Rechner portierte Laufzeitübersetzer J3 hat gezeigt, daß ein schwerwiegendes Problem des Squeak-Systems, die mangelnde Performanz des Interpreters, durch einen relativ einfachen und portablen Laufzeitübersetzer verbessert werden kann.

Das J3-Experiment hat aber auch weitere Probleme aufgezeigt. Diese führen zu einem neuen Entwurf, der versucht, alle dargestellen Probleme zu lösen. Grundlage des SX Systems ist die Idee, daß der Laufzeitübersetzer nicht Maschinencode, sondern auch Bytecode generieren kann. Dies erlaubt eine Vereinfachung des verwendeten Zwischencodes im Squeak-System. Dadurch können die Probleme gelöst werden, die durch nicht transparente Optimierungen des Squeak-Bytecodes entstanden sind.

A Quellcode des *tinyBenchmark*

```
tinyBenchmarks
  "Report the results of running the two tiny Squeak benchmarks.
   | t1 t2 r n1 n2 |
   n1 _ 1.
   [t1 _ Time millisecondsToRun: [n1 benchmark].
   t1 < 1000] whileTrue:[n1 _ n1 * 2].

   n2 _ 28.
   [t2 _ Time millisecondsToRun: [r _ n2 benchFib].
   t2 < 1000] whileTrue:[n2 _ n2 + 1].

   ^ ((n1 * 500000 * 1000) // t1) printString, ' bytecodes/sec; ',
    ((r * 1000) // t2) printString, ' sends/sec'

benchFib  "Handy send-heavy benchmark"
   ^ self < 2
        ifTrue: [1]
        ifFalse: [(self-1) benchFib + (self-2) benchFib + 1]

benchmark  "Handy bytecode-heavy benchmark"
   | size flags prime k count |
   size _ 8190.
   1 to: self do:
       [:iter |
       count _ 0.
       flags _ (Array new: size) atAllPut: true.
       1 to: size do:
          [:i | (flags at: i) ifTrue:
             [prime _ i+1.
             k _ i + prime.
             [k <= size] whileTrue:
                [flags at: k put: false.
                   k _ k + prime].
             count _ count + 1]]].
  ^ count
```

Literatur

[1] Bowen Alpern, C. R. Attanasio, Anthony Cocchi, Derek Lieber, Stephen Smith, Ton Ngo, John J. Barton, Susan Flynn Hummel, Janice C. Sheperd, and Mark Mergen. Implementing jalapeño in Java. In *OOPSLA'99 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 34(10) of *ACM SIGPLAN Notices*, pages 314–324, Denver, CO, October 1999. ACM Press.

[2] Vassili Bykov. The Hitch Hiker's Guide to the Smalltalk Compiler. *Smalltalk Chronicles*, 2(1), March 2000.

[3] C. Chambers, D. Ungar, and E. Lee. An Efficient Implementation of SELF a Dynamically-Typed Object-Oriented Language Based on Prototypes. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 24, pages 49–70, New York, NY, October 1989. ACM Press.

[4] David Detlefs and Ole Agesen. The Case for Multiple Compilers. *OOPSLA '99 VM Workshop: Simplicity, Performance and Portability in Virtual Machine Design*, 1999.

[5] Peter Deutsch and Alan M. Schiffman. Efficient Implementation of the Smalltalk-80 System. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 297–302. ACM, January 1984.

[6] Michael Franz and Thomas Kistler. Slim Binaries. *CACM*, 40(12):87–94, December 1997.

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

[8] Mark Guzdial. *Squeak, Object-Oriented Design with Multimedia Applications*. Prentice-Hall, 2000.

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

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

[11] Dan Ingalls. Design Principles Behind Smalltalk. *BYTE Magazine*, August 1981. `http://users.ipa.net/~dwighth/`.

[12] 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, October 1997.

[13] Alan C. Kay. Microelectronics and the Personal Computer. *Scientific American*, 237(3):231–244, September 1977.

[14] Alan C. Kay. Computer, Networks and Education. *Scientific American*, pages 138–148, September 1991.

[15] Alan C. Kay. The Early History of Smalltalk. In *History of Programming Languages*, pages 511–579. ACM Press/Addison-Wesley, 1996.

[16] Alan C. Kay and Adele Goldberg. Personal Dynamic Media. *Computer*, 10(3):31–41, March 1977.

[17] S. Matsuoka, H. Ogawa, K. Shimura, and K. Kimura, Y.and Hotta. OpenJIT – A Reflective Java JIT Compiler. In *Proceedings of OOPSLA '98 Workshop on Reflective Programming in C++ and Java*. ACM, October 1998.

[18] H. Ogawa, K. Shimura, S. Matsuoka, F. Maruyama, Y. Sohda, and Y. Kimura. OpenJIT: An Open-Ended, Reflective JIT Compiler Framework for Java. In *ECOOP 2000*, volume 1850 of *LNCS*. Springer, June 2000.

[19] Ian Piumarta. J3 for Squeak. unpublished, `http://www-sor.inria.fr/~piumarta/squeak/unix/zip/j3-2.6.0/doc/j3/`.

[20] Ian Piumarta. ccg: a Tool for Writing Dynamic Code Generators. OOPSLA 99 Workshop on Simplicity, Performance and Portability in Virtual Machine Design, November 1999.

[21] Ian Piumarta. *Squeak, Open Personal Computing for Multimedia*, chapter Porting Squeak, pages 215–261. Prentice-Hall, 2001.

[22] Ian K. Piumarta. *Delayed Code Generation in a Smalltalk-80 Compiler*. PhD thesis, University of Manchester, September 1992.

[23] Donald Bradley Roberts. Practical Analysis for Refactoring. Technical Report UIUCDCS-R-99-2092, University of Illinois at Urbana-Champaign, April 1999.

[24] Tim Rowledge. *Squeak, Open Personal Computing for Multimedia*, chapter A Tour of the Squeak Object Engine, pages 185–214. Prentice-Hall, 2001.

[25] David Ungar and Randall B. Smith. Self: The power of simplicity. *ACM SIGPLAN Notices*, 22(12):227–242, December 1987.

[26] David M. Ungar. Generation Scavenging: A Non-disruptive High-Performance Storage Reclamation Algorithm. In Peter Henderson, editor, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 157–167, Pittsburgh, Pennsylvania, April 1984. Also distributed as *ACM SIGPLAN Notices 19*(5):157–167, May, 1984 and *Software Engineering Notes 9*(3):157–167, May, 1984.

[27] P. T. Zellweger. Interactive Source-Level Debugging of Optimized Programs. Technical Report CSL-84-5, Xerox Corporation, Palo Alto Research Center, Palo Alto, CA, May 1984.