Smalltalk – ein Blick hinter die Kulissen

Christian Baumann
Mat.Nr.: 0126091
Version 1.0

16. Dezember 2006

Inhaltsverzeichnis

Einleitung

Diese Arbeit soll einen Einblick „unter die Haube“ der Programmiersprache Smalltalk geben. Smalltalk wurde von Sprachen wie Lisp oder Simula beeinflusst und hat selbst auch viele der heutigen Sprachen inspiriert [Wiki]. Smalltalk wurde zwar bereits vor 30 Jahren entwickelt, besaß aber schon alle Ansätze einer modernen objektorientierten Sprache. Es zeichnet sich vor allem durch drei Eigenschaften aus:

1. Der Zustand des Systems wird in Objekten gespeichert.
2. Die Abarbeitung erfolgt mittels Nachrichtenaustausch zwischen den Objekten.
3. Das Verhalten von Objekten wird in zugehörigen Klassen beschrieben.

Wie auch viele andere moderne objektorientierte Programmiersprachen, läuft auch Smalltalk auf einer virtuellen Maschine, die auf der Basis eines Stacks arbeitet. Zwecks Performancesteigerung und aus Platzersparnis wird der Code in eine für die virtuelle Maschine besser ausführbaren Zwischencode – den sog. Bytecode – übersetzt. Im folgenden werden die Architektur der virtuellen Maschine und der Bytecode näher beschrieben.

1 Die virtuelle Maschine

1.1 Architektur

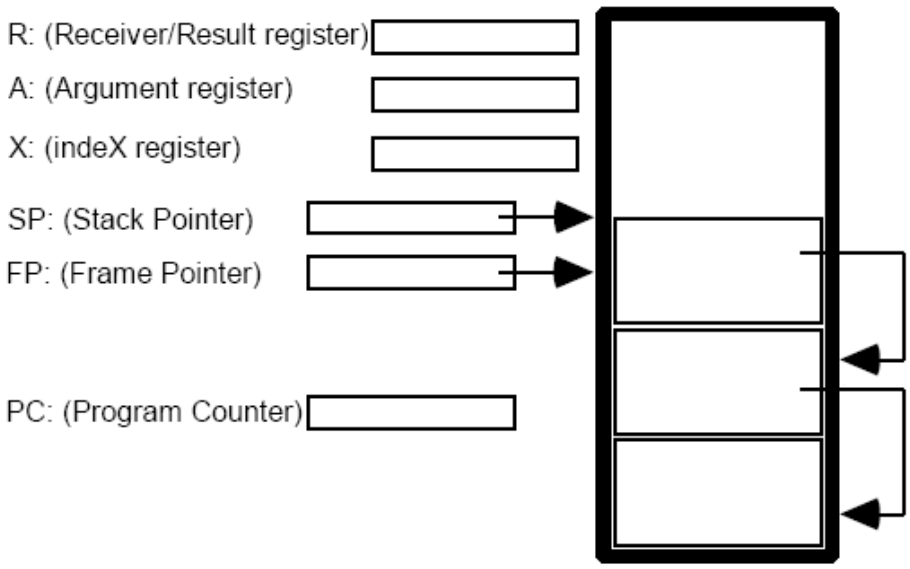

Abbildung 1: Aufbau des Smalltalk–Stacks

Das Ausführungsmodell der virtuellen Maschine von Smalltalk besteht aus einem Stack und einigen Spezialregistern. Der Stack besteht aus einer Reihe von objektorientierten Zeigern (Referenzen), wobei das SP–Register (Stack Pointer) auf den letzten Eintrag zeigt (Abbildung 1). Es können auch individuelle Referenzen auf den Stack gelegt oder von ihm entfernt werden indem einfach der SP inkrementiert oder dekrementiert wird [Wir99].
Aus logischer Sicht ist der Stack in sog. Stack–Frames (auch Activation Records genannt) organisiert, die den Zustand individueller Methodenaufrufe repräsentieren. Der Pointer auf das aktuelle Frame steht im FP–Register (Frame Pointer). Jeder Frame enthält eine Referenz auf den vorherigen Frame. Das PC–Register (Program Counter) enthält einen Wert, der die nächste auszuführende Instruktion identifiziert.
Das Register R beinhaltet einen Zeiger des Objekts, das der Empfänger einer Nachrichten–Sendeinstruktion ist. Nach der Rückkehr aus der Nachricht beinhaltet es den Rückgabewert der Prozedur.
Das Register A wird dazu verwendet das letzte (linkeste) Argument einer Prozedur zu übergeben. Die anderen Argumente werden über den Stack übergeben. Während des Aufrufs der Prozedur ist das Register A flüchtig. Das bedeutet, dass nach der Rückkehr aus der Prozedur der Wert des Registers undefiniert ist.
Das Register X dient als Index oder Basisregister für indirekte Feldadressierungen von Feldern von Objekten. Diverse Instruktionen, die explizit indirekte Zugriffe durchführen, benutzen auch das X–Register. Auch dieses Register ist während des Aufrufs von Prozeduren flüchtig.

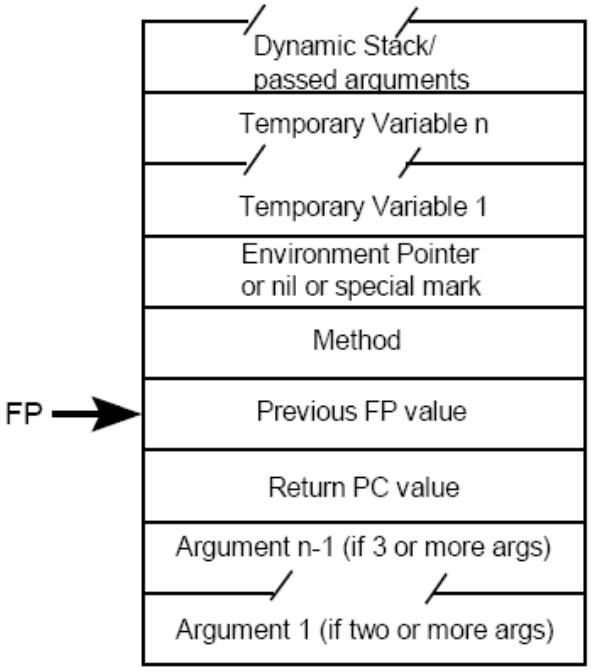

Abbildung 2: Stack–Frame Format

1.2 Stack–Frame Format

Ein Stack–Frame (Activation Record) stellt den Zustand eines Methodenaufrufs dar (Abbildung 2). Dazu werden eine Reihe von Referenzen auf dem Stack abgelegt, die den Zustand vor dem Aufruf der Methode festhalten, damit dieser Zustand nach der Rückkehr aus der Methode wiederhergestellt werden kann.
Im Speziellen werden der aktuelle Program Counter (PC) und der aktuelle Frame Pointer (FP) und evtl. ein Environment Pointer auf dem Stack abgelegt. Außerdem werden die Argumente, einige temporäre Variablen und eine Referenz der Methode selbst für den Methodenaufruf übergeben.

1.3 Kontexte

Lokale Variablen werden nicht auf dem Stack abgelegt, sondern in sog. Kontexten gespeichert. Ein Kontext kann als Scope der momentan sichtbaren Variablen in einem Codeabschnitt gesehen werden. Ein Stack–Frame kann über den Environment–Pointer eine Referenz auf einen Kontext besitzen.
Kontexte können auch verschachtelt sein. Das erste Feld in einem Kontext beinhaltet eine Referenz auf seinen umgebenden Kontext oder `nil`, wenn kein umgebender Kontext existiert.

1.4 Blöcke

Ein Block ist ein Objekt, welches einen Codeblock in einer vordefinierten Umgebung (Environment) darstellt. Stößt die virtuelle Maschine während der Programmausführung auf einen literalen Codeblock (Code in eckigen Klammern), so wird automatisch ein neues Block–Objekt

dafür angelegt. Das Block–Objekt beinhaltet eine Kopie des Environment–Pointers des aktuellen Stack–Frames (also des Stack–Frame, der zu der Zeit der Generierung des Blocks gerade aktiv war). Außerdem enthält es eine Referenz auf die kompilierte Methode, die den ausführbaren Code aus dem Block enthält. Jeder literale Block wird in eine davon unabhängige kompilierte Methode kompiliert.

2 Bytecodes

2.1 Allgemeines

Bei der Programmiersprache Smalltalk handelt es sich um einen Interpreter. Das bedeutet, dass der Code erst zur Laufzeit in die Zielsprache des Systems übersetzt wird. Um dies möglichst effizient zu lösen verwenden die meisten Interpreter (und so auch Smalltalk) eine Zwischensprache. Das Programm wird in diese Zwischensprache kompiliert und dann zur Laufzeit interpretiert.
Zur Steigerung der Performance verwendet Smalltalk als Zwischensprache sog. Bytecode. Das sind einfache 1 bis 3 Byte große Anweisungen, die von der virtuellen Maschine sehr effizient ausgeführt werden können. Die virtuelle Maschine selbst arbeitet mit einem Stack auf dem alle Operationen durchgeführt werden. Die Argumente für eine Operation werden auf den Stack gepusht, die Operation wird ausgeführt (dazu werden die Argumente vom Stack gepopt) und das Ergebnis wird wieder auf dem Stack abgelegt [Bud87].
Ein weiterer Vorteil der Bytecodes ist natürlich auch, dass Programme sehr wenig Speicher benötigen. Um sich die Nutzung der Bytecodes besser vorstellen zu können muss man sich die verwendeten Instanzvariablen und temporären Variablen in einem Array abgelegt vorstellen. Der Zugriff auf die einzelnen Variablen erfolgt über einen Index. Also ähnlich wie ein Arrayzugriff.

2.2 Aufbau eines Bytecodes

Es gibt ungefähr ein Dutzend verschiedene Arten von Operationen. Es werden aber ein paar mehr definiert um genügend „Spielraum“ für Optimierungen zu haben, aber 16 sollten in diesem Fall reichen. Die Operation wird durch ein Tag (ein sog. Opcode) gefolgt von Zusatzinformationen dargestellt. Meistens ist diese Zusatzinformation aber nur ein einfacher Integerwert.
Da man angenommen hat, dass es maximal 16 verschiedene Operationen gibt, kommt man mit 4 Bits (nibble) für den Opcode aus. Die anderen 4 Bits kann man für die Wertübergabe nutzen. Konkret wird der Opcode in den oberen 4 Bits und der Wert in den unteren 4 Bits abgelegt. Dies ist die einfachste Form einer Operation, die nur 1 Byte Speicher benötigt.

Größe	4	4
Feld	*Opcode*	*Wert*

Bei dieser Form zeigt sich allerdings ein Problem: als Wert hat man natürlich auch nur 16 verschiedene Werte zur Verfügung. Das bedeutet man könnte lediglich 16 verschiedene Instanzvariablen ansprechen. In der Praxis werden zwar nie mehr benötigt, aber man wollte das System nicht zu sehr einschränken. Aus diesem Grund wird der Operation ein weiteres Byte angehängt (insgesamt also 2 Bytes), das den Wert beinhaltet (z.B. den Index der Instanzvariable).

2.3 Opcodes

Erweitertes Opcode–Format (opcode 0) Die oberen 4 Bits sind immer 0, die unteren 4 Bits stellen den eigentlichen Opcode dar gefolgt von 8 Bits (1 Byte) für den Wert.

Größe	4	4	8
Wert	0	*Opcode*	*Wert*

Zugriff auf eine Instanzvariable (opcode 1) Die Instanzvariable mit dem angegebenen Index wird auf dem Stack abgelegt. Möchte man eine Instanzvariable mit einem Index größer 15 ansprechen, so muss man das erweiterte Operatorformat verwenden.

Zugriff auf Argumente oder temporäre Variablen (opcode 2) Die Argumente und temporären Variablen werden in einem gemeinsamen Array gespeichert: dem Kontext. Die Operation legt das Element mit dem angeforterten Index auf dem Stack ab. Auch hier gilt wieder, wenn man Elemente mit einem Index größer als 15 ansprechen will, so muss man das erweiterte Operatorformat verwenden.

Zugriff auf Literale (opcode 3) Das Element aus dem Literalarray mit dem angegebenen Index wird auf dem Stack abgelegt. Das Literalarray kann jede Art von Literal, dass zum Zeitpunkt des Parsens bekannt war, beinhalten (Characters, Strings, Integer, Symbole, Arrays). Die Ausnahme bilden die Klassen, die erst zur Laufzeit bekannt sind.

Zugriff auf Klassenobjekte (opcode 4) Zum Zeitpunkt des Parsens wird ein Symbol, das den Namen der Klasse repräsentiert ins Literalarray geschrieben. Der Wert beinhaltet den Index des Symbols der gewünschten Klasse im Literalarray. Während der Ausführung dieser Operation wird die Beschreibung der Klasse geladen und am Stack abgelegt.

Speichern in Instanzvariablen (opcode 6) Das aktuelle Topelement des Stacks wird gepopt und in die Instanzvariable mit dem angegebenen Index geschrieben.

Speichern in temporäre Variablen (opcode 7) Das aktuelle Topelement des Stacks wird gepopt und in den Speicher des Kontextarrays am angegebenen Index geschrieben. Obwohl Argumente als auch temporäre Variablen im Kontextarray liegen kann der Parser unterscheiden, dass kein Befehl Argumente überschreiben würde.

Senden von Nachrichten (opcode 8) Hierbei wird angenommen, dass der Empfänger der Nachricht und die erforderlichen Argumente für die Nachricht auf dem Stack abgelegt worden sind. Das Wertfeld beinhaltet die Anzahl der Argumente, die der Nachricht übergeben werden. Wenn mehr als 16 Argumente übergeben werden sollen, muss das erweiterte Operatorformat verwendet werden. Das angehängte Byte wird als Index im Literalarray interpretiert. Das an der Stelle gespeicherte Symbol wird als Repräsentation des Nachrichtenselektors gewertet.

Größe	4	4	8
Wert	8	*Anzahl der Argumente*	*Index der Nachricht*

Senden von Nachrichten an super (opcode 9) Die Felder sind die selben wie in der vorigen Nachricht. Das Objekt, das `self` und `super` repräsentiert ist das selbe und liegt im Kontextarray an erster Stelle.

Erstellen von Blöcken (opcode 14) Das Wertfeld dieses Operators beinhaltet die Anzahl der Argumente für den Block. Ist diese nicht Null, so beinhaltet das zweite Byte die Position im Kontextarray, wo die Argumente für den Block abgelegt werden sollen. Das dritte Byte enthält die Größe (in Bytecodes) der Instruktionen im Block. Die Instruktionen für den Block stehen unmittelbar nach diesem Byte.

Größe	4	4	8	8
Wert	14	*Anzahl der Argumente*	*Position der Argumente*	*Blockgröße*

Spezialinstruktionen (opcode 15) Der Wert gibt die Instruktion an, die ausgeführt werden soll. Diese Instruktionen benötigen entweder keine Argumente oder mehr als 15 Argumente und würden damit nicht von der Kompaktheit profitieren. Mit Hilfe dieser Instruktionen kann man beispielsweise das oberste Stackelement duplizieren bzw. löschen. Außerdem gibt es einige Instruktionen mit denen sich verschiedene Werte zurückgeben lassen (`return` aus einer Methode). Die nächste Gruppe von Instruktionen werden für Verzweigungen und auch Schleifen benutzt. Mit ihrer Hilfe kann man eine gewisse Anzahl an Bytes auslassen (vorwärts wie rückwärts). Spezialversionen von diesen Instruktionen können das sogar bedingt ausführen, z.B. wenn das oberste Stackelement `true` bzw. `false` ist. Die letzte Instruktion wird dazu verwendet primitivere Operationen durchzuführen.

3 Zusammenfassung

Diese Arbeit sollte einen tieferen Einblick in die Programmiersprache Smalltalk bieten. Es wurde der Aufbau der virtuellen Maschine und der zugehörigen Komponenten näher erläutert. Das zentrale Element der virtuellen Maschine ist der Stack.
In einem weiteren Abschnitt wurde der Aufbau von Bytecode erklärt. Dieser Bytecode ist eine Zwischensprache die auf der virtuellen Maschine läuft und in die Smalltalk–Programme übersetzt werden.

Literatur

[Kra83] Glen Krasner, editor. *Smalltalk–80: Bits of History, Words of Advice.* Addison–Wesley, 1983.

[Bud87] Timothy Budd, editor. *A Little Smalltalk.* Addison–Wesley, 1987.

[Wir99] Alan Wirfs–Brock, Pat Caudill. *A Smalltalk Virtual Machine Architectural Model.* Instantiations, Inc, 1999.

[Wiki] Wikipedia, http://en.wikipedia.org/wiki/Smalltalk. *Smalltalk* Wikipedia, 2006.