Effiziente Virtuelle Maschinen für funktionale Programmiersprachen

Proseminar Programmiersysteme WS 2003/04
Lehrstuhl Programmiersysteme – Prof. Gert Smolka
Universität des Saarlandes

Steffen Heil
Betreuer: Thorsten Brunklaus
02. April 2004

1. Einleitung

Die Ausführung von Programmen in funktionalen Sprachen erfordert die Übersetzung dieser Programme die Maschinensprache der Zielplattform oder in einen Bytecode, der auf der Zielplattform interpretiert wird. Beide Ansätze haben unterschiedliche Implikationen im Bezug auf Effizienz und Portabilität.

1.1. Native-Code Compiler

Native-Code Compiler übersetzen die Programme direkt in die Maschinensprache der Zielplattform. Der erzeugte Code kann sehr stark für die Zielplattform optimiert werden und die Ausführung erzeugt keinen interpretativen Overhead. Der Nachteil dieser Compiler ist die starke Anpassung des Compilers and die Zielplattform. Die Komponenten zur Code-Erzeugung und Optimierung müssen für jede potentielle Zielplattform neu erstellt oder größtenteils verändert werden.

1.2. Virtuelle Maschinen

Compiler für Virtuelle Maschinen erzeugen einen Bytecode, der für die Virtuelle Maschine (VM) optimiert ist. Die VM selbst wird im Maschinencode der Zielplattform erstellt oder einer gut

optimierbaren Sprache, die bereits auf vielen potentiellen Zielplattformen unterstützt wird. Für C existieren viele wohl bekannte Compiler für die meisten Plattformen, daher erscheint die Implementierung der VM in C sinnvoll.

Der Compiler für die funktionale Sprache erzeugt keinen Maschinencode, sondern ByteCode für die VM. Auf der Zielplattform wird die VM ausgeführt, die wiederum das eigentliche Programm interpretiert. Die Fetch/Decode-Stufen der nächsten Instruktion und das Verzweigen zur eigentlichen Ausführung der Instruktion innerhalb der VM verzögert die Ausführung des ByteCodes gegenüber dem Native Code. Diese Differenz wird interpretativer Overhead genannt. Er kann reduziert werden, wenn die VM hochspezialisierte Befehle beherrscht, und somit weniger Instruktionen die Fetch/Decode-Stufen durchlaufen müssen.

Zur Portation des Systems auf neue Zielplattformen genügt es, die relativ simple VM auf die neue Plattform zu portieren. Der Bytecode ist nicht von der Plattform abhängig. Außerdem können durch den Einsatz kompatibler VMs auf verschiedenen Rechnern Programme verteilt ausgeführt werden.

Nur bei Verwendung einer Virtuellen Maschine scheint eine einfache Portierung möglich.

2. Effizienz bei funktionalen Sprache

Um Aussagen über die Effizienz verschiedener Ausführungsmodelle treffen zu können, müssen die wichtigsten Konstrukte in funktionalen Sprachen bestimmt werden. Offensichtlich werden funktionale Sprachen von Abstraktionen und Applikationen dominiert.

2.1. Abstraktionen

Bei Abstraktionen müssen wir unterscheiden zwischen kaskadierten, einstelligen und mehrstelligen Abstraktionen. Kaskadierte Funktionen bieten die meiste Funktionalität, da sie partielle Applikationen unterstützen. Einstellige Funktionen, die mehrere Parameter akzeptieren sollen, müssen diese in Form von Argumenten-Tupeln annehmen. Das Erzeugen dieser Tupel erfordert aber eine Allokation.

Mehrstellige Funktionen können auf verschiedene Arten realisiert werden. Einerseits ist die "direkte" Umsetzung durch Argumenten-Register möglich, was jedoch das Kopieren der Argumenten-Werte in die Argumenten-Register vor dem Funktionsaufruf und das Kopieren der Argumenten-Register in lokale Variablen nach dem Funktionsaufruf erfordert. Eine weitere Möglichkeit zur Realisierung mehrstelliger Funktionen ist die Umsetzung als einstellige Funktionen mit Argumenten-Tupeln, was jedoch die schon genannten Nachteile hat. Die letzte Möglichkeit ist die Darstellung mehrstelliger Funktionen als kaskadierte Funktionen. Bei der naiven Auswertung von Mehrfachapplikationen müssen jedoch unnötige Closures gebaut werden. Falls die Darstellung in kaskadierter Form mindestens eben so schnell wie die direkte Darstellung realisiert werden kann, so ist die kaskadierte Form vorzuziehen.

2.2. Applikationen

Applikationen treten in funktionalen Sprachen in zwei Varianten auf: Einzelapplikation und Mehrfachapplikation. Die Mehrfachapplikation sollte in der Regel semantisch identisch sein zur sukzessiven Einzelapplikation. In der Regel wird die Mehrfachapplikation aber effizienter sein. Wenn die kaskadierte Form für die Darstellung von mehrstelligen Funktionen verwendet wird, ist eine sehr effiziente Ausführung von Mehrfachapplikationen unabdingbar. Allerdings können bei Mehrfachapplikationen in funktionalen Sprachen zwei Sonderfälle auftreten: Unterversorgung und Überversorgung. Diese müssen gesondert beachtet werden.

Es erscheint also sinnvoll, mehrstellige Funktionen als kaskadierte Funktionen darzustellen. Ziel muss es aber sein, kaskadierte Funktionen mit Mehrfachapplikation möglich effizient auszuführen.

3. Kaskadierung und Applikation

Der verwendeten Sprache sollte eine strikte left-to-right Auswertung zu Grunde liegen.

3.1. Naive Auswertung

Bei der naiven Auswertung der Mehrfachapplikation $M\ N_1\ N_2$ würde zunächst die Applikation $M\ N_1$ ausgewertet und das Ergebnis würde auf N_2 angewandt. Um jedoch der Zwischenergebnis darzustellen bzw. zu speichern muss eine Closure erzeugt werden. So müssen bei der sukzessiven Applikation von k Argumenten mindestens k-1 Closures erzeugt werden. Closures werden auf dem Heap alloziiert. Der Speicherbedarf steigt damit ebenso wie die benötige Rechenzeit, da Allokationen und die spätere Garbage Collection sehr rechenzeit-intensiv sind.

3.2. Direkte Mehrfachapplikation

Bei der direkten Mehrfachapplikation werden zunächst alle Parameter ausgewertet und auf den Stack gelegt. Bei der Auswertung der Funktion können dann alle Parameter nach Bedarf benutzt werden, ohne dass die Auswertung unterbrochen werden muss. Es müssen also keine unnötigen Closures erzeugt werden.

Bei dieser Auswertungsstrategie entsteht jedoch ein Unterschied zwischen der Mehrfachapplikation und der sukzessiven Einzelapplikation: Bei der Mehrfachapplikation von $M\ N_1\ N_2$ würde in der Reihenfolge M, N_1, N_2, $M\ N_1\ N_2$ ausgewertet, bei der sukzessiven Einzelapplikation hingegen würde in der Reihenfolge M, N_1, $M\ N_1 = a$, N_2, $a\ N_2$ ausgewertet. Diese Auswertungsreihenfolgen unterscheiden sich insbesondere darin, ob N_2 vor oder nach der Applikation $M\ N_1$ ausgewertet wird.

Dies läst sich mit einem Beispiel verdeutlichen:

```
exception Abs   exception Right
val f = fn x => (raise Abs; fn y => y)

Left-To-Right Evaluation Order

f 1 (raise Right)       => raise Right
(f 1) (raise Right)     => raise Abs
```

Dieses Problem tritt nicht auf, wenn man generell statt einer left-to-right Auswertung eine right-to-left Auswertung verwendet: Bei der Mehrfachapplikation von M N_1 N_2 würde in der Reihenfolge N_2, N_1, M, N_1, M N_1 N_2 ausgewertet, bei der sukzessiven Einzelapplikation würde in der Reihenfolge N_2, N_1, M, M N_1 = a, a N_2 ausgewertet. Beide Auswertungsstrategien werten N_2 vor der Applikation M N_1 aus.

```
Right-To-Left Evaluation Order

f 1 (raise Right)       => raise Right
(f 1) (raise Right)     => raise Right
```

4. Die abstrakte Maschine

Die abstrakte Maschine besteht aus einem Datenspeicher und einem Interpreter. Der Speicher hat eine automatische Garbage Collection.

Zum Interpreter verwendet eine Codepointer, der auf den nächsten auszuführenden Befehl im Instruktionsspeicher zeigt. In einer Umgebung verwaltet er die lokalen Bezeichnerbindungen. Über einen Stack werden Aufrufparameter und Ergebnisse übergeben und Closures von unterbrochenen Auswertungen abgelegt. Zusätzlich wird ein Akkumulator benutzt, um Zwischenergebnisse schneller verfügbar zu machen.

Die Semantik der Maschine wird durch den Befehlssatz und die Operationale Semantik bestimmt.

5. Speicher und Darstellung von Werten

Der Speicher der Virtuellen Maschine ist als eine Reihe von 32-bit Worten organisiert. Auch die Zellen des Stacks, der Inhalt des Akkumulators und die Werte in der Umgebung sind 32-Bit Worte. Alle Werte müssen also mit 32-Bit dargestellt werden können.

Um Integer-Werte, konkrete Datentypen und Datenblöcke (wie z.B. Strings) einheitlich darzustellen, verwendet die Maschine eine Unterscheidung von "unboxed" und "boxed" Werten. "Unboxed", also direkt als Wert, werden nur Integer-Werte dargestellt. Alle anderen Werte

werden als Referenz in den Speicher (in der Regel im Heap) dargestellt. Die Maschine muss also nur zwischen "boxed" und "unboxed" unterscheiden können.

5.1. Integer-Werte

Als Integer-Werte erlaubt die Maschine Werte im Bereich von -2^31 bis 2^31-1. Um diese Werte darzustellen, werden genau 31 Bit benötigt. Diese werden in den Bits 1-31 abgelegt. Bit 0 wird konstant auf 1 gesetzt. Jeder Wert, der im Bit 0 eine 1 hat, ist ein Integer.

Bei arithmetischen Operationen muss dieses Bit gesondert beachtet werden. Bei der Addition muss beispielsweise immer 1 subtrahiert werden, damit die Summe und die Darstellung korrekt sind.

5.2. Speicherblöcke

Alle Werte, deren Bits 0-1 den Wert 00 haben, sind Zeiger auf Blöcke im Speicher. Damit sind Zeiger klar von Integern zu unterscheiden, deren letztes Bit 1 ist.

n	GC	Tag
$Feld_1$		
...		
$Feld_n$		

Alle Speicherblöcke haben einen einheitlichen Header. Dieser Header ist das erste 32-Bit Wort des Speicherblocks. Er ist aufgeteilt in drei Bereicht: Die Bits 10-31 enthalten die Größe des Speicherblocks exklusive Header. Die Bits 8-9 sind für die Garbage Collection reserviert. Die Bits 0-7 geben an, welchen Typ der Inhalt dieses Speicherblocks beschreibt. Diese Typangabe wird Tag genannt.

5.3. Unstrukturierte Blöcke

Unstrukturierte Blöcke haben den Tag 255. Innerhalb dieser Blöcke können Daten abgelegt werden, die von der Maschine nicht weiter interpretiert werden. Insbesondere enthalten unstrukturierte Blöcke keine Zeiger auf andere Blöcke, so dass der Garbage Collector den Inhalt unstrukturierter Blöcke ignorieren kann.

Ein Beispiel für die Verwendung von unstrukturierten Blöcken sind Strings. Strings werden nullterminiert im Datenbereich eines Blocks abgelegt. Da der Speicher aus 32-Bit Worten besteht und Strings in der Regel Folgen von 8-Bit-Zeichen sind, werden diese zu 4-Zeichen Worten zusammengefasst und mit 02, 003, 0004, oder 00005 aufgefüllt. Die Länge eines Strings lässt sich damit schnell berechnen mit der Formal: Länge des Speicherblocks * 4 – letztes Byte.

5.4. Closures

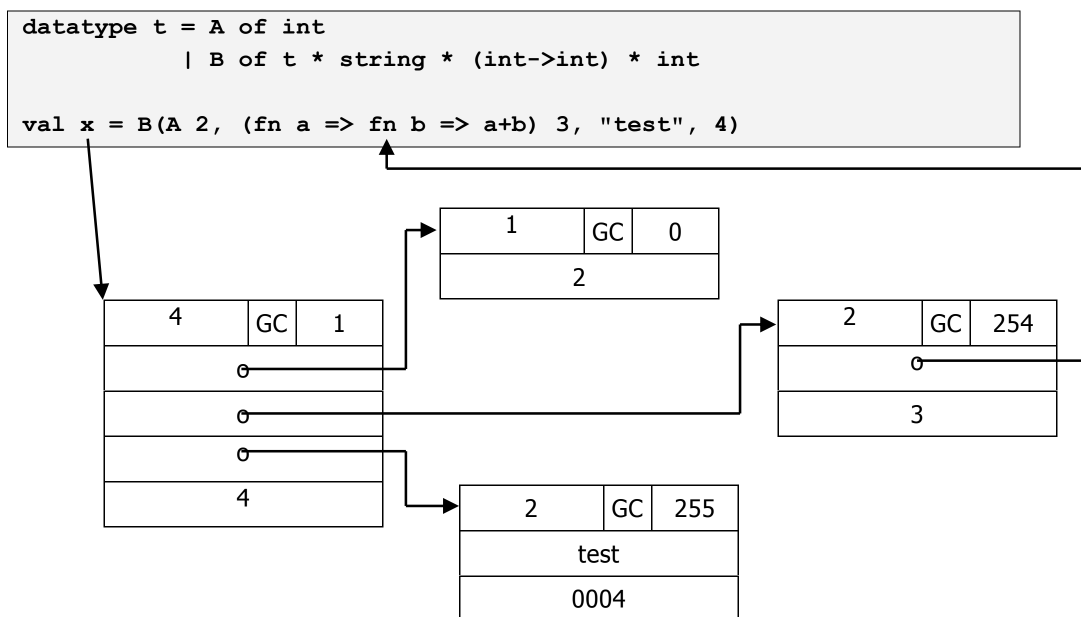
Closures haben den Tag 254. Im ersten Datenfeld wird der Code-Zeiger abgelegt. Die folgenden Worte enthalten die Werte der Umgebung.

5.5. Konkrete Datentypen

Konkrete Datentypen haben als Tag die Variantennummer des Konstruktors. Die Varianten werden in Reihenfolge der Deklaration durchnummeriert, beginnend mit null. Die folgenden Worte enthalten die Elemente der Datenstruktur. Dies sind wieder boxed oder unboxed Werte.

Für konkrete Datentypen mit mehr als 254 Varianten muss eine andere Darstellung verwendet werden, da die Tags 254 und 255 eine andere Bedeutung haben. In diesem Falle wird der Tag 0 verwendet und die Variantennummer als Integer in das erste Feld des Datenblocks geschrieben. Die folgenden Felder enthalten dann die Elemente der Datenstruktur.

5.6. Beispiel



6. Darstellung von Abstraktionen

Abstraktionen werden in λ-Abstraktionen mit de-Bruijn-Darstellung ausgedrückt. Beispiele:

```
val f = fn a => fn b => a + b
val f = λ . λ . <1> + <0>

val f = fn a => fn b => fn x => a + b
val f = λ . λ . λ . <2> + <1>
```

Bezeichnerbindungen werden durch generische Lambdas ersetzt; benutzende Auftreten von Bezeichnern werden durch Indices ersetzt, die angeben, wie viele Lambdas im Syntaxbaum nach oben übersprungen werden müssen, um das passende Lambda zu finden.

Dadurch verringert sich die Umgebung von einer Funktion von Bezeichnern nach Werten zu einer einfachen sortierten Liste von Werten. Beispiel:

```
val f = λ . λ . λ . <2> + <1>
        ↑
f : [], o

val g = f 5 3

g :  [3, 5],  o
```

7. Die Operationale Semantik

7.1. Der Befehlssatz

Die vollständige Maschine hat 9 Instruktionsgruppen mit insgesamt über 122 Instruktionen. Die folgenden Instruktionen genügen jedoch für Umsetzung einfacher Funktionen:

Code	**Akku**	**Env.**	**Stack**	**Code**	**Akku**	**Env.**	**Stack**
Access(k); c	a	$[v_0..v_n]$	s	c	v_k	$[v_0..v_n]$	s

Access(k) - kopiert das k. Element der Umgebung in den Akkumulator. Es entspricht der de-Bruijn-Notation der gebundenen Variablen <n>.

Code	**Akku**	**Env.**	**Stack**	**Code**	**Akku**	**Env.**	**Stack**
Reduce(c'); c	a	e	s	c'	a	e	<c,e> :: s

Reduce(c) - führt die Instuktionen-Folge c aus und legt den Wert des Akkumulators auf den Stack. Dies entspricht der Auswertung eines Ausdrucks. Jede in Reduce eingeschlossene Instruktionsfolge muss mit Return enden.

Code	**Akku**	**Env.**	**Stack**	**Code**	**Akku**	**Env.**	**Stack**
Return	a	e	<c_0, e_0> :: s	c_0	a	e_0	a :: s
Return	(c_0,e_0)	e	v :: s	c_0	(c_0, e_0)	e_0	v :: s

Return - beendet die Auswertung eines Ausdrucks. Falls auf dem Stack ein Aufrufrahmen liegt, so ist die Auswertung regulär abgeschlossen, die Auswertung des äußeren Ausdrucks wird fortgesetzt. Befindet sich auf dem Stack aber ein Wert, so kann kein Rücksprung zu letzten Auswertung erfolgen. Dies kann aber nur dann zutreffen, wenn eine Überversorgung mit Argumenten vorliegt. Da das Programm aber typkorrekt gewesen sein muss, muss das Zwischenergebnis eine Closure sein. Mit dieser wird dann die Auswertung fortgesetzt.

Code	Akku	Env.	Stack	Code	Akku	Env.	Stack
Grab; c	a	e	v :: s	c	a	v :: e	s
Grab; c	a	e	<c_0, e_0> :: s	c_0	a	e_0	(Grab;c,e) :: s

Grab - nimmt das oberste Element [das nächste Argument] vom Stack und fügt sie als neues erstes Element in die Umgebung ein. Dies entspricht einem Lambda in der Lambda/de-Bruijn-Darstellung einer Abstraktion. Falls kein weiterer Wert auf dem Stack liegt (sondern ein Aufruf-rahmen), so liegt eine Unterversorgung vor. In diesem Falle muss die Auswertung unterbrochen werden, das Ergebnis der Auswertung ist dann die Closure der unterbrochenen Auswertung.

Code	Akku	Env.	Stack	Code	Akku	Env.	Stack
ConstInt(i); c	a	e	s	c	i	e	s

ConstInt(i) - legt die Integer-Konstante i in den Akkumulator.

Code	Akku	Env.	Stack	Code	Akku	Env.	Stack
AddInt; c	a	e	v :: s	c	a + v	e	s
SubInt; c	a	e	v :: s	c	a – v	e	s

AddInt/SubInt - Addiert/Subtrahiert das oberste Element des Stacks zum/vom Akkumulator.

7.2. Die Übersetzungsfunktion

Folgende Funktion übersetzt Ausdrücke in Lambda/de-Bruijn-Notation in Instruktionsfolgen:

```
[ <k> ]          --> Access(k)
[ λ . N ]        --> Grab; [ N ]
[ M N ]          --> Reduce( [ N ]; Return ); [ M ]
[ i ]            --> ConstInt(i)
[ N1 + N2 ]      --> Reduce( [ N2 ]; Return ); [ N1 ]; AddInt
[ N1 - N2 ]      --> Reduce( [ N2 ]; Return ); [ N1 ]; SubInt
```

7.3. Übersetzungsbeispiel

```
( fn a => fn b => a + b ) ( ( fn x => x ) 1 ) 2

( λ . λ . <1> + <0> ) ( ( λ . <0> ) 1 ) 2

[ ( λ . λ . <1> + <0> ) ( ( λ . <0> ) 1 ) 2 ]
=
Reduce( ConstInt(2); Return );
Reduce( Reduce( ConstInt(1); Return );
        Grab; Access(0); Return );
Grab; Grab; Reduce( Access(0); Return );
            Access(1); AddInt; Return
```

8. Analyse

In diesem Befehlssatz werden Closures nur durch die Instruktionen Grab und Reduce angelegt.

Grab erzeugt genau dann eine Closure, wenn keine weiteren Argumente auf dem Stack liegen. Dass Grab ausgeführt wird, bedeutet aber, dass der Code ein weiteres Argument erwartet, es liegt also eine Unterversorgung vor. In diesem Falle muss eine Closure erzeugt werden, da das Ergebnis eine Abstraktion ist.

Reduce erzeugt Closures um die Auswertung eines Ausdrucks zu unterbrechen, bis ein Argument innerhalb dieses Ausdrucks ausgewertet ist. Diese Closures können aber nicht in die Umgebung eingefügt werden, und müssen daher nicht auf dem Heap angelegt werden. Es genügt, diese Closures auf dem Stack anzulegen.

8.1. Applikation mit Unterversorgung

Falls einer Applikation nicht genügend Argumente übergeben werden, so wird die Virtuelle Maschine bei der Ausführung auf eine Grab-Instruktion stoßen, obwohl kein Argument (Wert) mehr auf dem Stack liegt. An diesem Punkt wird die Auswertung unterbrochen und die erzeugte Closure als Ergebnis zurückgeliefert. Beispiel:

```
( fn a => fn b => 1 ) 2
( λ . λ . 1 ) 2

Reduce( ConstInt(2); Return );
  Grab; Grab; Access(1);
```

Dieser Ausdruck wird wie folgt ausgeführt:

Code	Akku	Env.	Stack
Reduce(ConstInt(2); Return); Grab; ...	v	e	s
ConstInt(2); Return	v	e	<Grab;..., e> :: s
Return	2	e	<Grab;..., e> :: s
Grab; Grab; Access(1);	2	e	2 :: s
Grab; Access(1);	2	2 :: e	s
			(Grab; ..., 2::e) :: s

Auf dem Stack liegt die erwartete Closure.

8.2. Applikation mit Überversorgung

Falls einer Applikation zu viele Argumente übergeben werden, so wird die Virtuelle Maschine bei der Ausführung auf eine Return-Instruktion, obwohl kein Aufrufrahmen auf dem Stack liegt. Dieser Punkt kann aber nur erreicht werden, wenn das Ergebnis der Auswertung des Ausdrucks eine Abstraktion darstellt, da das übersetzte Programm typkorrekt war. Diese Auswertung kann nun sofort mit den noch vorhandenen Argumenten fortgesetzt. Beispiel:

```
( fn a => a ) ( fn b => 1 ) 2
( λ . <0> ) ( λ . 1 ) 2

Reduce( ConstInt(2); Return );
Reduce( Grab; ConstInt(1); Return );
Grab; Access(0); Return
```

Dieser Ausdruck wird wie folgt ausgeführt:

Code	**Akku**	**Env.**	**Stack**
Reduce(ConstInt(2); ...); Reduce ...	v	e	s
ConstInt(2); Return	v	e	<Reduce..., e> :: s
Return	2	e	<Reduce..., e> :: s
Reduce(Grab; ...); Grab; ...	2	e	2 :: s
Grab; ConstInt(1); Return	2	e	<Grab..., e> :: 2 :: s
Grab; Access(0); Return	2	e	(Grab;..., e) :: 2 :: s
Access(0); Return	2	(Grab;...,e)::e	2 :: s
Return	(Grab;...,e)	(Grab;...,e)::e	2 :: s

An diesem Punkt findet Return keinen Aufrufrahmen auf dem Stack und führt daher die Auswertung mit der Closure im Akkumulator fort.

Code	**Akku**	**Env.**	**Stack**
Grab; ConstInt(1); Return	(Grab;...,e)	e	2 :: s
ConstInt(1); Return	(Grab;...,e)	2 :: e	s
Return	1	2 :: e	s
			1 :: s

Auf dem Stack liegt das erwartete Ergebnis.

9. Die reale Maschine

9.1. Weitere Optimierungen

Die reale Maschine kann weiter optimiert werden. Entgegen den zuvor vorgestellten Optimierungen kann mit diesen kleineren Optimierungen nur ein konstanter Kostenfaktor eingespart werden.

Durch einen Cache wird der Zugriff auf die Umgebung beschleunigt. Dazu wird die Umgebung logisch in eine Liste der obersten Elemente (mit konstanter Länge) und in eine Umgebungsliste unterteilt. Die oberen Elemente werden in den Registern der Zielplattform gespeichert. Der Zugriff auf diese Register ist deutlich schneller als Speicherzugriffe in die reguläre Umgebung.

Durch spezialisierte Instruktionen für häufige Instruktionen (wie z.B. let, konstante Konstruktoren, Endrekursion, etc.) kann die Anzahl der ausgeführten Instruktionen deutlich reduziert werden. Beispielsweise ist die von der Übersetzungsfunktion erzeugte Befehlssequenz für Ar-

gumente, die Abstraktionen sind, sehr ineffizient, da die Auswertung des Arguments versucht wird, obwohl diese mit einem Grab beginnt. Durch die Einführung einer Cur(ofs) kann dies vermieden werden. Es ersetzt genau dann ein Reduce, wenn der Block innerhalb von Reduce mit einem Grab beginnt. Dies erspart vor allem das Anlegen von Closures auf dem Stack.

Globale Variablen werden nicht in der Umgebung gebunden, sondern in eine globale Variablenliste eingetragen. Zum Zugriff auf diese Liste stehen spezialisierte Befehle zur Verfügung. Durch diese globale Variablenliste wird die Umgebung stark verkleinert. Daher sinkt der Aufwand beim erzeugen von Closures.

10. Zusammenfassung

Durch die Änderung der Ausführungsreihenfolge von left-to-right zu right-to-left kann die Maschine Mehrfachapplikationen kaskadierter Funktionen ausführen, ohne temporäre Closures zu erzeugen. Die Ausführung von Applikationen kaskadierter Funktionen ist mindestens eben so schnell wie die Ausführung von mehrstelligen Funktionen mit direkter Implementierung. Die Umsetzung von mehrstelligen Funktionen als kaskadierte Funktionen verringert also nicht die Ausführungsgeschwindigkeit. Die Maschine kann besonders im Zusammenspiel mit Unter- und Überversorgung von Abstraktionen hohe Geschwindigkeitsvorteile erzielen. Dieser Vorteil entsteht einerseits durch die entfallenden Allokationen bei der Erzeugung von Closures und andererseits durch den wesentlich geringeren Speicherbedarf.

11. Quelle

Xavier Leroy, The ZINC experiment:
an economical implementation of the ML language.
Technical report 117, INRIA, 1990