Compilerbau WS05/06

Implementierung Funktionaler Sprachen

Hans-Wolfgang Loidl,

Ludwig-Maximilians Universität, München

24. Januar 2006

`http://www.tcs.ifi.lmu.de/lehre/WS05-06/Compiler/`

Implementierung funktionaler Sprachen

Überblick: Implementierung funktionaler Sprachen.

Fokus auf Grundkonzepte funktionaler Sprachen:

- Funktionen höherer Ordnung (“higher-order functions”)
- Bedarfsauswertung (“lazy evaluation”)

Verschiedene Ansätze zur Implementierung:

- Umgebungsbasierte Modelle
- Graphreduktion
- Datenfluß

Funktionen höherer Ordnung

Funktionen höherer Ordnung sind solche Funktionen, die Funktionen als Argumente haben können oder eine Funktion als Wert liefern. D.h. Funktionen sind **Werte erster Klasse**.

Beispiel: `map` in ML:

```
map f [] = []
map f (x::xs) = (f x)::(map f xs)
```

Dabei sind verschachtelte Gültigkeitsbereiche (“nested scopes”) erlaubt. D.h. eine Funktion kann auf Variablen im statisch umgebenden Block zugreifen.

```
incBy : int => (int => int)
incBy n = h
          where h m = n+m
```

Closures

Ohne verschachtelte Funktionen können Funktionen höherer Ordnung einfach durch Code-Zeiger repräsentiert werden (siehe C).

Eine Deklaration einer Funktionsvariable reserviert Platz für einen (Code-)Zeiger.

Eine Zuweisung speichert diesen Zeiger. Zwischensprache:

```
MOVE(TEMP(t3), NAME(L4))
```

Ein Funktionsaufruf über eine Funktionsvariable wird zu einem indirekten Sprung. Zwischensprache:

```
CALL(TEMP(t3),...)
```

Closures

Im `incBy` Beispiel funktioniert das nicht, da von der zurückgegebenen Funktion `h` auf die freie Variable `n` zugegriffen wird.

In einem Program

```
incBy : int -> (int -> int)     main = let  inc = incBy 1
incBy n = h                                 n = 2
          where h m = n+m                in   inc 5
```

muss in der Bindung von `inc` der Parameter zu `incBy` gespeichert werden, damit der Rumpf zu `1+5` und nicht etwa zu `2+5` expandiert.

Eine Funktionsvariable muss daher zusätzlich zum Code noch Information über die Werte der freien Variablen mitführen:

$$\mathrm{Closure} = \mathrm{code} + \mathit{environment}$$

Repräsentation von Closures

Als Repräsentation einer Closure kann ein Objekt mit einer Methode und einer Instanzvariablen für jede freie Variable im Rumpf der Funktion verwendet werden.

Mit verschachtelten Funktionen ist die Programmauswertung nicht mehr rein stack-orientiert. Z.B. muss auch nach dem Verlassen der `incBy` Funktion der Wert von `n` noch zugreifbar sein, um `inc` ausführen zu können.

Dazu wird ein “escaping-variable record” im heap verwaltet. Jeder Stack frame verweist auf einen solchen record. Auch closures verwenden diese Repräsentation um die Werte freier Variablen zu speichern. Zusätzlich enthält dieser Record noch einen statischen Link, d.h. einen Zeiger auf einen Record mit den freien Variablen in der statischen Umgebung.

$$\mathrm{Environment} = \mathrm{frame} + \mathrm{escaping\ variable\ record}$$

Optimierungen in (rein) funktionalen Sprachen

Rein funktionale Sprachen sind **referentiell transparent**, d.h. wenn ein Funktionsaufruf $f\ x$ zu einem Wert a auswertet, so kann im Programm jedes Auftreten von $f\ x$ durch a ersetzt werden, ohne dass sich die Semantik ändert.

Dies ermöglicht Gleichheitsschließen (“equational reasoning”).

Dies ist nicht möglich wenn eine Sprache Seiteneffekte (zB: Variablenzuweisung) zuläßt.

Ein Hauptvorteil funktionaler Sprachen ist die reiche Möglichkeit zur Optimierung des erzeugten Codes.

Aussichtsreiche Optimierungen:

- Inlining
- Tail call elimination

Diese Optimierungen reduzieren vor allem die Kosten von Funktionsaufrufen.

Inlining

Ersetzen eines Funktionsaufrufs durch den Rumpf der Funktion, nach Substitution der aktuellen Parameter für die formalen Parameter.

Motivation: In funktionalen Sprachen sind einzelne Funktionen oft sehr kurz.

Zu beachten bei dieser Optimierung:

- Richtige Benennung der Variablen.

  Lösung: sämtliche Variablen in einem frühen Compiler-pass eindeutig benennen.

- Komplexe aktuelle Parameter sollen zuvor neuen Variablen zugewiesen werden.

- Vermeidung von duplizierten Auswertungen und von Aufblähung des Codes.

Heuristiken zum Entscheiden ob inlining sinnvoll ist.

Tail call elimination

Eine Funktion `f` is tail-rekursiv, wenn alle rekursiven Aufrufe der Funktion ihren Resultatwert direkt zurückgeben. Beispiel:

```
sumAcc : int list => int => int
sumAcc [] a = a
sumAcc (x::xs) a = sumAcc xs (a+x)
```

Dies ist effizienter als eine nicht-tail-rekursive Version, da man im erzeugten Code nach Beendigung des rekursiven Aufrufs `sumAcc xs (a+x)` direkt zum Aufrufer von `sumAcc` springen kann.

Zum Vergleich, hier die nicht-tail-rekursive Version

```
sum : int list => int
sum [] = 0
sum (x::xs) = x+(sum xs)
```

Das Resultat `sum xs` wird zur Berechnung der Summe benötigt.

Implementierung von Tail Calls

Zwei Beobachtungen ermöglichen die Erzeugungeffizienteren Codes:

- Lokale Variablen im Aufrufer (`x`) werden nach dem Tail Call nicht mehr benötigt
- Als return Adresse von `sumAcc xs (a+x)` kann direkt die Return Adress von `sumAcc` verwendet werden.

Code für Tail Calls:

1. Parameterübergabe (MOVEs zu Registern oder Frame-Positionen).
2. Rückschreiben der callee-save Register.
3. Pop des Frames der aufrufenden Funktion (`sumAcc (x::xs) a`).
4. Direkter Sprung zur aufgerufenen Funktion (`sumAcc xs (a+x)`).

Oft sind Parameter bereits in Registern, womit (1) entfällt. Funktionen die wenige Register benötigen (nur caller-save Register), benötigen keinen stack Frame.

Im Idealfall, kann der Tail Call direkt als Sprung übersetzt werden.

Effizienz Mantra:

$$\textit{Tail call recursion} = \textit{Loop}$$

Erweiterungen von MiniJava

Um diese Konzepte zu demonstrieren muß MiniJava erweitert werden.

FunJava MiniJava mit Funktionen höherer Ordnung und Seiteneffekten.

PureFunJava MiniJava mit Funktionen höherer Ordnung ohne Seiteneffekte (rein funktional).

LazyJava MiniJava mit Funktionen höherer Ordnung unter Verwendung von Bedarfsauswertung (“lazy evaluation”).

Einbindung in den Compiler

MiniJava muß um folgende Konstrukte erweitert werden:

- **Funktionstypen**: da Funktionen an Varibalen zugewiesen werden können, muß es möglich sein deren Typ zu deklarieren.
- **Blockstruktur**: das `return` statement kann nun am Ende eines Programmblocks stehen und definiert den Resultatwert dieses Blocks: `{ ... return x; }`
- **Bedingte Expression**: das `if` Konstrukt arbeitet nun über Expressions statt Statements, und definiert damit einen Wert.

Beispiel:

```
type intFun = int -> int;

class C {  public intfun add (int n) {
                public int h (int m) { return n+m; }
                return h; }  }
```

Restriktionen für eine rein funktionale Sprache

Um Seiteneffekte zu verbieten sind folgende Restriktionen auf MiniJava nötig:

- Keine Variablenzuweisungen, mit Ausnahme von Initialisierungen
- Keine Zuweisungen auf Felder dynamischer Datentypen, mit Ausnahme von Initialisierungen
- Keine Aufrufe zu Systemfunktionen mit Seiteneffekten

Um I/O Verhalten zu realisierern, kann **continuation-based I/O** verwendet werden.

Beispiel: all I/O Funktionen erhalten als zusätzliches Argument eine Funktion (“continuation”), die den Wert der gesamten Funktion berechnet:

```
type answer // builtin type
type cont = () -> answer

class ContIO {
 ...
 public answer putByte (int i, cont c);
}
```

Grundsätzliche Terminologie

Unterscheide: Eigenschaften der Semantik und der Auswertung.

Semantik von Programmiersprachen:

Eine Funktion f ist **strikt** gdw. $\mathrm{f}\ \bot = \bot$.

Auswertung von Programmen:

Eager Evaluation wertet die Argumente zu einer Funktion aus, bevor der Rumpf der Funktion betreten wird.

Lazy evaluation

Auswertungsprinzip: “Ein Ausdruck wird erst dann ausgewertet, wenn er zur Berechnung des Resultates benötigt wird.”

Vorteile:

- Vereinfacht das **logische Schließen** über Programme: Gleichheitsschließen *(“equational reasoning”)*. Ersetzen von Funktionsaufruf durch Rumpf erfordert eine derartige Auswertung. Beispiel:

```
f x y = if x>0 then x
                else y


-- Logisches Schliessen (lazy evaluation)
f 1 (1/0) --> if 1>0 then 1 else (1/0) --> 1

-- Auswertung (eager evaluation)
f 1 (1/0) --> ERROR
```

Lazy evaluation

Vorteile:

- Vermeidet unnötige Arbeit beim **Auswerten**, indem nicht verwendete Ausdrücke nicht ausgewertet werden. Beispiel:

```
-- ignoriert das 2. Argument
f 1 (fact 298357230985) --> 1
```

Gleichheitsschließen ist zulässig wenn die Sprache frei von Seiteneffekten ist.

Unterschied zwischen strikten und nicht-strikten Sprachen besteht nur im Terminationsverhalten der Programme.

Lazy evaluation ist **eine** Möglichkeit eine Sprache mit nicht-strikter Semantik zu implementieren.

Hauptvorteil nicht-strikter Sprachen: **Modularität** (siehe *Hughes, 1989*).

Formen der Parameterübergabe

Die Unterschiede zwischen lazy und eager evaluation manifestieren sich vor allem in der Parameterübergabe:

Call-by-value: Auswertung der Argumente vor dem Betreten der Funktion. Der Ausdruck $\mathrm{f\ a\ b}$ für eine Funktion mit der Definition $\mathrm{f\ x\ y\ =\ B}$ wird wie folgt übersetzt:

```
eval a
bind a to x
eval b
bind b to y
eval B
```

Formen der Parameterübergabe

Call-by-name: Alle Werte im Program sind Funktionswerte, repräsentiert als sogenannte *“thunks”*: Funktionen, die ihre Werte erst auf Anfrage auswerten.

Eine Variable vom Typ `int` wird zu einer Funktion vom Typ `() -> int`.

Erzeugung einer Variable (in Variablendeklarationen und bei Funktionsaufrufen) wird zur Erzeugung einer Funktion (**keine Auswertung**).

Zugriff auf eine Variable wird zu Funktionsapplikation, mit `()` als Argument.

```
let  x = 5+7
in   x+x
```

```
let  x z = 5+7
in   (x ())+(x ())
```

Formen der Parameterübergabe

Problem von call-by-name: mehrfache Auswertung von Ausdrücken.

Call-by-need: Wie call-by-name, vermeidet aber mehrfache Ausführung der neu eingeführten Funktionen. Jeder “thunk” enthält einen “memo slot”, der nach Auswertung das Resultat des Ausdrucks enthält.

Mittels einer full-laziness transformation, können weitere Mehrfachauswertungen eliminiert werden (siehe “loop invariant hoisting”).

Im Graphreduktionsmodell wird diese “Memoisation” durch das überschreiben des ursprünglichen Graphen erreicht.

Kurz:

$$\mathrm{Lazy\ evaluation} = \mathrm{call\ by\ name} + \mathrm{sharing}$$

Implementierung von call-by-need

Ein “thunk” wird als Objekt mit folgenden Komponenten definiert:

- Funktion, zur Auswertung
- memo slot, für das Resultat
- Variablenliste, mit Werten der freien Variablen (siehe “closures”)

Beispiel: Der Rumpf des Programs

```
let incBy5 = incBy 5
in  twenty = incBy5 15
```

wird übersetzt zu

```
twentyThunk twenty = new twentyThunk(...)
```

mit den Klassen:

```
class intThunk { public int eval ();
                 int memo;
                 boolean done; }

class c_int_int { public int exec (int x); }
class intFuncThunk { public c_int_int eval ();
                     c_int_int memo;
                     boolean done;  }

class twentyThunk extends intThunk {
 intFuncThunk incBy5;
 public int exec() {
  if (!done) {
   memo = incBy5.eval().exec(15);
   done = true;
  }
  return memo;
 }
 twentyThunk(fv1) { this.incBy5 = fv1; }
}
```

Implementierung von call-by-need

Zur Auswertung von `twentyThunk twenty = new twentyThunk(...)` werden die freien Variablen als Argumente für den Konstruktor benötigt.

Dazu können Transformationen wie “closure conversion” oder “lambda lifting” verwendet werden.

Die `eval` Methode wertet den Ausdruck aus, oder liefert das bereits bekannte Resultat.

Optimierungen nicht-strikter funktionaler Sprachen

Durch Verwendung referentieller Transparenz im engeren Sinn, sind weitere Möglichkeiten zur Optimierung gegeben.

- **Invariant hoisting (full laziness transformation)** : Ausdrücke die nicht von den Parametern einer Funktion abhängen, können aus der Funktion herausgezogen werden.

  Dies ist nur gültig, wenn die herausgezogene Variable lazy ausgewertet wird!

- **Dead code removal**: Eliminierung von code, der Variablen bindet, die nicht weiter benötigt werden.

  Aber: falls der eliminierte code nicht terminiert, wird unter eager evaluation dadurch das Terminationsverhalten verändert.

- **Deforestation**: Eliminierung von Datenstrukturen, die nur als Zwischenwerte benötigt werden.

  Mit lazy evaluation können unendliche Datenstrukturen verwendet werden.

Strictness Analysis

Implementierung von lazy evaluation ist teuer, da zahlreiche thunks erzeugt werden.

Oft kann gezeigt werden, dass ein Programmausdruck immer benötigt wird. In dem Fall, kann für diesen Ausdruck eager evaluation verwendet werden, ohne dass dies der nicht-strikten Semantik widerspricht.

```
f x y = x+y

g x y = if x>0 then x else y

h s y = if y>0 then (s::(h s (y-1))) else []
```

Ist eine Funktion strikt in einem Argument, so muss kein thunk für dieses Argument erzeugt werden.

Eine Striktheitsanalyse (“strictness analysis”) ermittelt für welche Ausdrücke dies möglich ist.

Strictness Analysis

Verschiedene Ansätze zu statischen Programmanalysen:

- **Abstrakte Interpretation**: Man definiert einen vereinfachten Domain ausgewertet (z.B: 2 Werte: definitiv strikt, vielleicht nicht-strikt). Ausgehend von bekannten Primitiven werden abstrakte Funktionen für alle Programmfunktionen ermittelt. Das abstrakte Programm wird über diesen Domain ausgewertet.
- **Typ Inferenz**: Man definiert eine Logik (Typsystem), das als “Typ” die Striktheit einer Funktion modeliert. Dann können Standardalgorithmen zur Typinferenz verwendet werden.

SECD Maschine

Ein **umgebungsbasiertes** Maschinenmodell, d.h. die Variablen-Wert Abbildungen werden als Datenstrukturen in der abstrakten Maschine kodiert.

Historisch erste dedizierte Maschine zur Ausführung funktionaler Sprachen. Verwendet eine spezielle Datenstruktur (“dump”) zur Speicherung von Closures. Ansonsten stackbasiert.

Grundlage für frühe Lisp Implementierungen.

Komponenten:

- **S**: stack
- **E**: environment, Abbildung von Variablen auf Werte
- **C**: code, Rest des auszuführenden Programs
- **D**: dump, letzte Umgebung der Maschine; wird beim verlassen einer Funktion gelesen

Im Fall der Auswertung einer Closure, muss der derzeitige Zustand (stack, environment, code) auf den Dump gepusht werden.

Auswertungsregeln der SECD Maschine

$$
\begin{aligned}
&eval : stack \Rightarrow env \Rightarrow code \Rightarrow dump \\
&eval\ (x :: s)\ e\ []\ [] && = && x \\
&eval\ (x :: s)\ e\ []\ (s_1, e_1, c_1) :: ds && = && eval\ (x :: s_1)\ e_1\ c_1\ ds \\
&eval\ s\ e\ (\texttt{x} :: c)\ d && = && eval\ ((e[\texttt{x}]) :: s)\ e\ c\ d \\
&eval\ s\ e\ ((\texttt{LAMBDA}\ v\ b) :: c)\ d && = && eval((\texttt{CLOS}\ v\ b\ e) :: s)\ e\ c\ d \\
&eval\ ((\texttt{CLOS}\ v\ b\ e') :: a :: s)\ e\ (\texttt{APPLY} :: c)\ d && = && eval\ []\ (\texttt{bind}\ v\ a) :: e\ b\ (s, e, c) :: d \\
&eval\ (\texttt{PRIM}\ f) :: a :: s\ e\ \texttt{APPLY} :: c\ d && = && eval\ ((f\ a) :: s)\ e\ c\ d \\
&eval\ s\ e\ ((f\ a) :: c)\ d && = && eval\ s\ e\ (a :: f :: \texttt{APPLY} :: c)\ d
\end{aligned}
$$

Graphreduktion

Die bisher gezeigten Techniken bauen Konzepte funktionaler Sprachen in die Implementierung einer einfachen Objekt-orientierten Sprache ein.

Einer der populärsten Ansätze zur Implementierung (rein) funktionaler Sprachen, wie Haskell, ist **Graphreduktion**.

Dabei werden sowohl Program als auch Daten als Graphstrukturen im Heap repräsentiert. Die Auswertung eines Programs entspricht der Ausführung des Codes im Knoten eines Graphs. Nach dem Auswerten wird der Graph durch sein Resultat überschrieben.

Vorteile:

- Gemeinsame Programmteile werden nur einmal ausgeführt
- Einheitliche Darstellung von Daten und Programmteilen $\Longrightarrow$ vereinfacht die Realisierung von “lazy evaluation”.
- Unabhängige Subgraphen können in jeder Reihenfolge, insbesondere auch parallel ausgeführt werden.

Weiterführendes Material

- *Functional Programming*, A. Field, P. Harrison, Addison Wesley, 1988.

  Genereller Überblick zu funktionalen Sprachen und deren Implementierung.

- *Why Functional Programming Matters*, John Hughes, in *The Computer Journal*, 32(2):98–107, 1989. `http://www.cs.chalmers.se/~rjmh/Papers/whyfp.ps`

  Vorteile von funktionaler Programmierung und lazy Evaluation

- *Implementing functional languages: a tutorial*, Simon Peyton Jones and David Lester. Prentice Hall, 1992. `http://research.microsoft.com/Users/simonpj/Papers/pj-lester-book/student.pdf.gz`

  Implementierung mittels Graphreduktion (Text online; literate Code).

- *Implicit Parallel Programming in pH*, R. Nikhil and Arvind, Morgan Kaufmann Publishers, 2001. ISBN 1-55860-644-0.

  Implementierung mittels Datenflußgraphen, insbesondere für parallele Ausführung.