Fachrichtung 6.2 — Informatik
Universität des Saarlandes
Tutorenteam der Vorlesung Programmierung 1

Programmierung 1 (Wintersemester 2012/13)

Erklärung 4

(Tripeldarstellung von Prozeduren)

Hinweis: Dieses Blatt enthält eine zusätzliche Erklärung erstellt von den Tutoren. Für die Richtigkeit besteht daher keine Gewähr.

Die Erklärung sowie ihr Thema sind weder für die Klausur relevant noch irrelevant.

1 Wie beschreibt man eine Prozedur?

In der Vorlesung haben wir gesehen, dass die Umgebung das Ergebnis eines Ausdrucks verändern kann. So wird der Ausdruck $e_1 + e_2$ in der Umgebung

$$V_1 = [e_1 := 5, e_2 := 7]$$

zu 12 ausgewertet, in der Umgebung

$$V_1 = [e_1 := 27, e_2 := 13]$$

zu 40 und in der leeren Umgebung kann der Ausdruck wegen den freien Bezeichnern e_1 und e_2 erst gar nicht ausgewertet werden.
Das gleiche gilt auch für Prozeduren: Die Prozedur

```
fun f x = x + e
```

liefert für f 5 in der Umgebung

$$V_1 := [e := 5]$$

den Wert 10, in der Umgebung

$$V_2 := [e := 27]$$

den Wert 32. Aus diesem Grund reicht, um eine Prozedur vollständig zu charakterisieren, der Code nicht aus. Deshalb wird in Kapitel 2 die Tripeldarstellung zur Beschreibung von Prozeduren eingeführt.
Diese sieht aus wie folgt: Das Tripel besteht nun aus dem *Code*, d.h. der eigentlichen Prozedurdeklaration, dem *Typ* der Prozedur und der *Umgebung*, in der die Prozedur deklariert wurde.

2 Wie kann man die Tripeldarstellung von einer beliebigen Prozedur aufstellen?

Zuerst schaut man sich den Code der Prozedur an. Bei den meisten Prozeduren ist es möglich, einfach den gesamten Code abzuschreiben (für Spezialfälle siehe die Abschnitte zu Let-Ausdrücken und Tripeldarstellungen von Abstraktionen). Diesen schreibt man an die erste Stelle des Prozedurtripels. Die Typen werden weggelassen, da diese ja an der zweiten Stelle des Prozedurtripels stehen. Wir betrachten das Programm:

```
val a = 7+5
fun f (x:int) = x*a
```

Der Code in der Tripeldarstellung für f müsste nun

```
fun f x = x * a
```

lauten. In diesem Fall nimmt die Prozedur f ein x des Typs *int* als Argument und hat den Ergebnistyp *int*. Damit ist der Typ der Prozedur $int \to int$. Schauen wir uns den Code unserer Prozedur an, so sehen wir, dass in diesem der Bezeichner a frei auftritt, a ist also kein Parameter der Prozedur. Also müssen wir in der Umgebung a an den oben definierten Wert $7 + 5 = 12$ binden und erhalten damit die Umgebung:

$$[a := 12]$$

Insgesamt wird f an folgendes Prozedurtripel gebunden:

```
f:=(fun f x = x * a, int -> int, [a:=12])
```

Enthält der Code einer Prozedur g nun in der obigen Umgebung eine Anwendung der Prozedur f, so ist es notwendig, diese (bzw. ihre gesamte Tripeldarstellung) ebenfalls der Umgebung der Tripeldarstellung der Prozedur g zu binden (denn eine Prozedur ist ja auch ein Wert). Zusammenfassend heißt das für die Tripeldarstellung:

(a) Code ohne Typen abschreiben

(b) Prozedurtyp ermitteln

(c) in der Umgebung freie Bezeichner an ihre Werte binden

3 Tripeldarstellung von Abstraktionen

Die Tripeldarstellung von Abstraktionen funktioniert nach dem gleichen Prinzip: Man schreibt den Code der Abstraktion ab, bestimmt den Typ und die Umgebung. Ein Beispiel: Es sei folgende Val-Deklaration gegeben:

```
val f = fn (x:int) =>fn (y:int) => (x = y)
```

Ist nun gefragt, wie die Tripeldarstellung aussieht, an die f gebunden wird, so gilt:

(a) Ohne Typen den Code der Abstraktion abschreiben, an den gebunden wird, also:

```
fn x => fn y => (x = y)
```

val gehört nicht mehr zum Code der Prozedur hinzu!

(b) Der Typ, an den f gebunden ist lautet: $int \to int \to bool$

(c) Es tauchen keine freien Bezeichner auf, die Umgebung ist also leer.

Damit erhalten wir als Tripeldarstellung:

```
f:=(fn x => fn y => (x = y), int -> int -> bool, [])
```

4 Let-Ausdrücke

Betrachten wir nun folgenden Ausdruck:

```
let
  val a = 7
  fun f x = x+a
in f
end
```

Ist nun nach der Tripeldarstellung gefragt, zu der dieser Ausdruck auswertet, so geht man wieder nach oben genanntem Schema vor:

(a) Was ist der Code der Prozedur? Der Code entspricht dem Ausdruck zwischen *in* und *end*. Dabei müssen wir den Ausdruck in eine Abstraktion umschreiben:

```
val g = fn x => x + a
```

und können dann unser Vorgehen für Abstraktionen anwenden. Also erhalten wir schließlich als Code

```
fn x => x + a
```

(b) Was ist der Typ? Dazu müssen wir uns den Argumenttyp von f anschauen, dieser ist *int*. Wir addieren das Argument zu einer ganzen Zahl, d.h. wir haben als Ergebnistyp wieder *int* und erhalten somit den Prozedurtypen $int \rightarrow int$.

(c) Wie sieht die Umgebung aus? Dazu müssen wir das Programm zwischen *let* und *in* betrachten. Alle die dort gebundenen Bezeichner werden zur Berechnung des Ausdrucks gebunden (a nur zur Berechnung von f selbst), d.h. wir erhalten als Umgebung:

```
[f:=(fn x = x+a, int ->int, [a:=7])]
```

Nun müssen wir die einzelnen Teile nur noch zur Tripeldarstellung zusammensetzen.

5 Tripeldarstellung kaskadierter Prozeduren

Was passiert nun, wenn ein Argument bei einer kaskadierten Prozedur schon "eingelöst" wurde? In diesem Fall verkürzt sich die Tripeldarstellung der Prozedur einfach um ein Argument und wir erhalten in unserem Code einen freien Bezeichner mehr, der im dritten Teil der Tripeldarstellung, der Umgebung, gebunden wird.
Man betrachte folgende zwei Prozeduren:

```
fun f (x:int) (y:int) = x + y
val g = f 7
```

Gehen wir wieder unsere Fragen durch:

(a) Was ist der Code? Am besten löst man die kaskadierte Prozedur wieder in Abstraktionen auf:

```
fun f (x:int) = fn (y:int) => x + y
```

Dann erhalten wir als Code für g die hintere Abstraktion und gehen somit wieder wie für Abstraktionen vor:

```
fn y => x + y
```

(b) Was ist der Typ? Auch hier erhalten wir als Typen wieder $int \rightarrow int$.

(c) In unserem Code der Prozedur g taucht der freie Bezeichner x auf, der aber schon gebunden wurde, d.h. wir erhalten als Umgebung:

```
[x:=7]
```

6 Muss man die Prozedur selbst bei rekursiven Prozeduren wieder in die Umgebung schreiben?

Man betrachtet die Prozedur:

```
fun f x = f x
```

Würden wir bei der Ausführung dieser Deklaration f wieder in seine Umgebung schreiben, so müssten wir das in der Umgebung wieder machen, und in der Umgebung von f in der Umgebung wieder, ... Damit hätten wir Divergenz in der Definition! D.h. die Prozedur selbst wird bei der Deklaration selbst nie in die Umgebung geschrieben. Um trotzdem rekursive Aufrufe ausführen zu können, wird bei der Ausführung der Prozeduranwendung die Prozedur zusammen mit der Bindung der Argumente in die Umgebung geschrieben, in der die Prozedur ausgeführt wird.[1]

[1] nach dem Buch: Gert Smolka, Programmierung - Eine Einführung in die Informatik in Standard ML