14. Übungsblatt zu Programmierung (WS 1999/2000)

Andreas Rossberg, Prof. Gert Smolka

Abgabe: Freitag, 11. Februar 2000 vor der Vorlesung

Vorgegegebene Module finden Sie unter `http://www.ps.uni-sb.de/courses/info1-ws99/uebung/sml/blatt14/`.

Aufgabe 14.1: Bingungsanalyse (5 Punkte)

Annotieren Sie für folgendes Programm die Ergebnisse der Bindungsanalyse wie im Skript, und geben Sie alle frei auftretenden Bezeichner an.

```
val nil = [1,2,3,4]
fun f x = fn x => x::nil
val f = fn x => fn x => f x x
val f = f
datatype ('a,'b) t = A | B of 'a | C of 'b | D
val A = D
val (B A) = C C
val (B E) = C C
val a = A
val b = let val a = 0 in 1 end
val c = a
fun g y z = let val y = g y y in g y end y
```

Hinweis: Den Quelltext des Programmes finden sie zum bequemen Ausdrucken auch im Netz.

Aufgabe 14.2: Abstrakte Syntax (5 Punkte)

Die abstrakte Syntax der Sprache F lässt sich durch folgende Typen beschreiben:

```
structure Syntax =
  struct
    type con = int
    type id  = string

    datatype typ = Bool
                 | Int
                 | Arrow of typ * typ

    datatype exp = Con of con
                 | Id  of id
                 | If  of exp * exp * exp
                 | App of exp * exp
                 | Abs of id * typ * exp
  end
```

Betrachten Sie nun folgenden Ausdruck in der Sprache F:

```
fn x : int => (fn f : int -> int -> int =>
               if equal x 0 then 1 else f x x) add
```

(a) Geben Sie einen SML-Ausdruck an, der den F-Ausdruck gemäß der abstrakten Syntax beschreibt.

(b) Geben Sie den F-Ausdruck als mathematisches Objekt in mathematischer Notation an.

Aufgabe 14.3: Freie Bezeichner (8 Punkte)

Gegeben sei eine Struktur `Set`, die Mengen implementiert. Schreiben Sie eine Prozedur

```
freeIds: Syntax.exp -> Syntax.id Set.set
```

die für einen Ausdruck der Sprache F die Menge der freien Bezeichner ermittelt.

Hinweis: Schreiben Sie eine Hilfsprozedur

```
freeIds': Syntax.id Set.set -> Syntax.exp -> Syntax.id Set.set
```

so dass `freeIds' X e` die Menge aller Bezeichner liefert, die in `e` frei auftreten und nicht in `X` sind (`X` ist die Menge der bereits als gebunden vermerkten Bezeichner).

Aufgabe 14.4: Elaboration (15 Punkte)

Im Netz finden Sie eine unvollständige Version eines Interpreters für F, inklusive einer kurzen Beschreibung. In den verbleibenden Aufgaben soll dieser zu einem direkt ausführbaren Programm vervollständigt werden.

(a) Implementieren Sie die Struktur `Type` gemäß folgender Signatur:

```
signature TYPE =
  sig
    type typ = Syntax.typ

    val constant: Syntax.con -> typ
    val base:     typ Env.env
  end
```

Die Prozedur `constant` liefert den Typ einer Konstanten. Die Umgebung `base` beschreibt die initiale statische Umgebung von F:

```
add:     int -> int -> int
sub:     int -> int -> int
mul:     int -> int -> int
equal:   int -> int -> bool
less:    int -> int -> bool
```

(b) Implementieren Sie die statische Analyse in der Struktur `Elab` gemäß folgender Signatur:

```
signature ELAB =
  sig
    datatype error = IdUnbound of Syntax.id
                   | IfConditionNotBool
                   | IfBranchesMismatch
                   | AppNoFunction
                   | AppArgumentMismatch

    exception Error of error
    val elab: Type.typ Env.env -> Syntax.exp -> Type.typ    (* Error *)
  end
```

Die Prozedur `elab` ermittelt den Typ eines Ausdrucks relativ zu einer Typumgebung entsprechend den Regeln der statischen Semantik für F. Im Falle eines ungebundenen Bezeichners oder eines Typfehlers im Ausdruck soll eine Ausnahme `Error` mit entsprechendem Inhalt geworfen werden.

Aufgabe 14.5: Evaluation (15 Punkte)

Schreiben Sie nun die verbleibenden Module, um F-Ausdrücke mit dem Interpreter auswerten zu können.

(a) Implementieren Sie zunächst die Komponente `Value` gemäß folgender Signatur:

```
signature VALUE =
  sig
    type oper
    datatype value = Num  of int
                   | Op   of oper
                   | Proc of Syntax.id * Syntax.exp * value Env.env

    exception Error
    val constant: Syntax.con -> value
    val applyOp:  oper * value -> value (* Error *)
    val base:     value Env.env
  end
```

Der Typ `value` repräsentiert Werte der Sprache F. Dabei werden boolesche Werte durch Zahlen dargestellt: 0 steht für `false`, 1 für `true`. Vordefinierte Operatoren werden durch den abstrakten Typ `oper` repräsentiert.

Die Prozedur `constant` wertet eine Konstante aus, `applyOp` wendet einen Operator auf einen Wert an. Im Falle eines Typfehler soll die Ausnahme `Error` geworfen werden. Die initiale Wertumgebung wird durch `base` gegeben. Sie soll genau die unter Aufgabe 14.4 aufgeführten Bezeichner binden.

Hinweis: Alle vordefinierten Operatoren sind kaskadiert. Implementieren Sie den Typ `oper` deshalb wie folgt:

```
datatype oper = Add | Sub | Mul | Equal | Less
              | Add' of int | Sub' of int | Mul' of int
              | Equal' of int | Less' of int
```

Die ersten fünf Konstruktoren repräsentieren die unapplizierten Operatoren. Die verbleibenden repräsentieren das Ergebnis der Anwendung eines Operators auf ein erstes Argument.

(b) Implementieren Sie nun das Modul `Eval`:

```
signature EVAL =
  sig
    exception Error
    val eval: Value.value Env.env -> Syntax.exp -> Value.value
  end
```

Die Prozedur `eval` wertet einen Ausdruck gemäß der dynamischen Semantik relativ zu einer Wertumgebung aus. Falls ein Typfehler oder ein ungebundener Bezeichner auftritt, wird die Ausnahme `Error` geworfen.

Hinweis: Solange ihre Implementation von Elaborator und Evaluator korrekt und die initialen Umgebungen konsistent sind, können die Ausnahmen `Eval.Error` und `Value.Error` und damit die Fehlermeldung `Runtime error` des Interpreters nie auftreten!