6. Übungsblatt „Programmiersprachen“ SS 99

Leif Kornstaedt, Prof. Gert Smolka

Ausgabe: 28. Mai 1999 **Abgabe:** 4. Juni 1999

Diesmal sollen Sie einen Elaborator für eine Teilsprache von F2 programmieren. Dabei lernen Sie, wie polymorphe Typinferenz funktioniert. Außerdem üben Sie den Umgang mit Referenzen, imperativen Datenabstraktionen und getrennt übersetzten Modulen. Der Elaborator soll mit Moscow ML implementiert werden.

Der Elaborator soll allgemeinste Typen für Ausdrücke der Form

$$\begin{array}{rcl} a & ::= & x \\ & \mid & a_1 \ a_2 \\ & \mid & \texttt{fn}\ x\ \texttt{=>}\ a \\ & \mid & \texttt{val}\ x\ \texttt{=}\ a_1\ \texttt{in}\ a_2 \end{array}$$

gemäß der statischen Semantik von F2 inferieren. Dabei heißt τ ein allgemeinster Typ für T und a, wenn gilt:

a. $T \vdash a \Rightarrow \tau$

b. Für alle τ' mit $T \vdash a \Rightarrow \tau'$ gilt: τ' kann aus τ durch Instanzieren von Typvariablen erhalten werden.

Dabei nehmen wir vereinfachend an, dass die Typumgebung T keine freien Typvariablen enthält.

Alle für die Elaboration notwendigen Typen und Typschemata werden in einer imperativen Datenstruktur, die wir *Typbasis* nennen, alloziert. Die Typbasis wird durch ein Modul mit der Signatur

```
type ty
type tycon = string

exception Unification of string

val var: unit -> ty
val cons: tycon * ty list -> ty

val beginScheme: unit -> unit
val endScheme: ty -> unit

val instantiate: ty -> ty

val unify: ty * ty -> unit          (* Unification *)

val toString: ty -> string
```

realisiert. Typen und Typschemata werden beide als Werte des Typs `ty` repräsentiert. Dabei werden Typen mit den entsprechenden trivialen Typschemata identifiziert, die keine Typvariablen quantifizieren.

Die Operationen `var` und `cons` allozieren Typvariablen und konstruierte Typen. Die primitiven Typkonstruktoren `->` und `*` werden dabei genauso wie die anderen Typkonstruktoren behandelt.

Um ein Typschema zu allozieren, wird zuerst die Operation `beginScheme` ausgeführt. Dann wird der Typ des Schemas durch die Operationen `var` und `cons` und durch Rückgriff auf bereits allozierte Typen aufgebaut. Schließlich wird der aufgebaute Typ durch die Operation `endScheme` in ein Schema umgewandelt, das alle Typvariablen quantifiziert, die zwischen `beginScheme` und `endScheme` erzeugt wurden.

Die Operation `instantiate` alloziert zu einem Typschema einen neuen Typ, der eine allgemeinste Instanz des Schemas darstellt (also eine Kopie des Schemas, in der die quantifizierten Variablen durch neu allozierte Variablen ersetzt sind).

Die Operation `unify` versucht, zwei gegebene Typen durch Binden von Typvariablen zu unifizieren (das heißt gleichzumachen). Wenn die beiden Typen nicht gleichgemacht werden können, wird die Ausnahme `Unification` geliefert.

Die für die Elaboration notwendigen Bindungen von Bezeichnern an Typschemata werden in einer imperativen Datenstruktur abgelegt, die im Übersetzerbau als *Symboltabelle* bekannt ist. Da die betrachtete Teilsprache von F2 keine Muster hat, kann auf die Repräsentation des Bezeichnerstatus verzichtet werden. Die Symboltabelle verwaltet für jeden Bezeichner einen Keller von Typschemata. Es gibt Operationen, die zu einem Bezeichner ein Typschema hinzufügen (`push`) oder wegnehmen (`pop`). Die Operation `top` liefert zu einem Bezeichner das zuletzt eingetragene Typschema.

Der Elaborator inferiert jetzt für einen Ausdruck a einen allgemeinsten Typ wie folgt:

a. $a = x$: Sei σ das x von der Symboltabelle zugeordnete Typschema und τ eine allgemeinste Instanz von σ. Liefere τ.

b. $a = a_1\ \ a_2$: Inferiere allgemeinste Typen τ_1 und τ_2 für a_1 und a_2. Alloziere eine neue Typvariable τ. Unifiziere τ_1 mit τ_2 `->` τ. Liefere τ.

c. $a =$ **`fn`** x `=>` a': Alloziere eine neue Typvariable τ_1. Trage τ_1 für x in der Symboltabelle ein. Inferiere einen allgemeinsten Typ τ' für a'. Mache den Eintrag in die Symboltabelle rückgängig. Liefere τ_1 `->` τ'.

d. $a =$ **`val`** x `=` a_1 **`in`** a_2: Wende die Operation `beginScheme` der Typbasis an. Inferiere einen allgemeinsten Typ τ_1 von a_1. Wende die Operation `endScheme` der Typbasis auf τ_1 an. Trage τ_1 für x in die Symboltabelle ein. Inferiere einen allgemeinsten Typ τ_2 für a_2. Mache den Eintrag in der Symboltabelle rückgängig. Liefere den Typ τ_2.

Die Typbasis ist so konzipiert, dass sie mit geschachtelten Anwendungen der Operationen `beginScheme` und `endScheme` korrekt umgeht.

Die Typbasis repräsentiert Typen und Typschemata einheitlich und unterscheidet stattdessen zwischen quantifizierten und nicht-quantifizierten Variablen. Auf eine explizite Repräsentation der Menge der quantifizierten Variablen eines Schemas kann verzichtet werden, da Typschemata nicht in Typschemata geschachtelt werden.

Die Typbasis repräsentiert Typen und Typschemata durch Werte des Typs `ty`, der wie folgt definiert ist:

```
type level = int
type tycon = string

datatype typeterm =
    TVar of level
  | TCons of tycon * ty list
  | TPtr of ty
withtype ty = typeterm ref
```

Der Konstruktor `TPtr` markiert die Zeiger, die beim Binden einer Typvariablen eingeführt werden. Jede Variable hat eine ganzzahlige Tiefe (englisch *level*). Tiefe -1 bedeutet, dass es sich um eine quantifizierte Typvariable handelt. Die nichtnegativen Tiefen werden für nicht-quantifizierte Typvariablen verwendet.

Die Typbasis verwaltet eine mit 0 initialisierte Arbeitstiefe, die durch die Operationen `beginScheme` und `endScheme` jeweils um 1 erhöht beziehungsweise erniedrigt wird. Die Operation `endScheme` generalisiert zudem alle Typvariablen, die in dem als Argument gegebenen Typ vorkommen und deren Tiefe gleich der bisherigen Arbeitstiefe ist. Generalisieren einer Typvariablen bedeutet, sie durch die Tiefe -1 als quantifiziert zu markieren.

Die Unifikationsoperation muss beim Binden von Variablen die Tiefe beachten:

a. Wenn zwei Variablen unterschiedlicher Tiefe unifiziert werden sollen, muss die Variable mit der größeren Tiefe gebunden werden.

b. Wenn eine Variable α mit einem nichtvariablen Typ τ unifiziert werden soll, muss die Tiefe aller Variablen in τ, deren Tiefe größer als die Tiefe von α ist, auf die Tiefe von α erniedrigt werden.

Wichtiger Hinweis. *Auf den Webseiten zur Vorlesung finden Sie die Signaturen und Implementationen der Module `Type`, `Symbol` und `Expression` des Elaborators. Die Implementation liegen allerdings nur in übersetzter Form (für Moscow ML) vor. Damit sind Sie in der Lage, die Augaben unabhängig voneinander zu bearbeiten und die Funktion ihrer Module zu testen.*

Bei Aufgabe VI.5 handelt es sich um eine Sonderaufgabe, durch deren Bearbeitung Bonuspunkte erworben werden können.*

Aufgabe VI.1: Symboltabellen (10 Punkte)

In dieser Aufgabe sollen Sie ein Modul `Symbol` implementieren, das Symboltabellen als abstrakten Datentyp zur Verfügung stellt. `Symbol` soll folgende Signatur haben:

```
type id = string

type 'a t

exception Error

val new: unit -> 'a t
val push: 'a t * id * 'a -> unit
val pop: 'a t * id -> unit          (* Error *)
val top: 'a t * id -> 'a            (* Error *)
```

Eine Symboltabelle `'a t` ist eine Abbildung von Bezeichnern `id`, dargestellt als Zeichenketten, auf Keller von Einträgen des Typs `'a`. Die Operationen verhalten sich wie folgt:

- `new ()` erzeugt eine neue Symboltabelle. Diese enthält keine Einträge.
- `push (`t`, `i`, `a`)` trägt eine Bindung $i \mapsto a$ in die Symboltabelle t ein, wodurch ein eventuell bereits vorhandener Eintrag für i verdeckt wird.
- `pop (`t`, `i`)` löscht die zuletzt eingetragene Bindung für i aus der Symboltabelle t. Eine eventuell verdeckte Bindung für i wird dadurch wieder sichtbar.
- `top (`t`, `i`)` liefert den letzten Eintrag für i in der Symboltabelle t.

Aufgabe VI.2: Initiale Typbindungen (7 Punkte)

Schreiben Sie eine Prozedur

```
initializeEnv: Type.ty Symbol.t -> unit
```

die die Module `Type` und `Symbol` benutzt und Typbindungen entsprechend der folgenden Tabelle in eine gegebene Symboltabelle einträgt:

Bezeichner	Typ
`one`	`int`
`true`	`bool`
`false`	`bool`
Bezeichner	**Definition**
`id`	**`fn`** `x => x`
`before`	**`fn`** `x =>` **`fn`** `y => x`
`o`	**`fn`** `f =>` **`fn`** `g =>` **`fn`** `x => f (g x)`
`pair`	**`fn`** `x =>` **`fn`** `y =>` **`fn`** `z => (z x) y`
`fst`	**`fn`** `pair => pair (`**`fn`** `a =>` **`fn`** `b => a)`
`snd`	**`fn`** `pair => pair (`**`fn`** `a =>` **`fn`** `b => b)`

Aufgabe VI.3: Abstrakte Syntax (4 Punkte)

Das Modul `Expression` definiert eineabstrakte Syntax für die hier betrachtete Teilsprache von F2:

```
type id = string

datatype exp =                  (* a ::= *)
    VarExp of id                   (* x *)
  | AppExp of exp * exp            (* a1 a2 *)
  | FnExp of id * exp              (* fn x => a *)
  | ValExp of id * exp * exp       (* val x = a1 in a2 *)
```

Schreiben Sie SML-Ausdrücke, die zu der abstrakten Syntax der folgenden F2-Ausdrücke evaluieren:

a. `val x = (fn z => val y = fn v => v in y z) in x`

b. `fn x => (o fst) (pair x)`

Aufgabe VI.4: Elaborator (12 Punkte)

Implementieren Sie den beschriebenen Elaborator durch ein Modul `Elab` mit der folgenden Signatur:

```
val elab: Type.ty Symbol.t * Expression.exp -> string option
```

Wenn der Ausdruck in der durch die Symboltabelle gegebenen Umgebung nicht typisiert werden kann, soll `NONE` geliefert werden. Ansonsten soll der inferierte Typ als Zeichenreihe geliefert werden. Die Konversion soll mit der Prozedur `Type.toString` vorgenommen werden.

Testen Sie Ihren Elaborator mit den Ausdrücken aus Aufgabe VI.3 und der Prozedur aus Aufgabe VI.2. Weitere Testausdrücke sind im Web verfügbar.

Aufgabe VI.5*: Unifikation von Typen (15 Punkte)

Mit Ausnahme der Prozedur `unify` haben wir die Implementation des Moduls `Type` im Web angegeben. Vervollständigen Sie diese Implementation, indem sie eine passende Prozedur `unify` schreiben.