Praktische Informatik 3 WS 2009/10

9. Übungsblatt

Ausgabe: 7. 1. 2009
Abgabe: 21. 1. 2010

Berthold Hoffmann <hof>
Dominik Dietrich <dodi>
Christian Maeder <maeder>
Julia Seiter <jseiter>
Jasper van de Ven <jasper>

19 *Polymorphe Typinferenz* — *6 Punkte*

Im Aufgabenblatt 6 hatten Sie schon die Unifikation von zwei Typen implementiert. Dies ist Voraussetzung für die *polymorphe Typinferenz* in funktionalen Sprachen. Eine Funktionsanwendung FA ist *wohlgetypt*, wenn der Ausdruck F eine Funktion von einem Typ $t \to r$ ist und der Ausdruck A einen Wert des Typs u liefert, so dass u und der Argumenttyp t von F unifiziert werden können. Dazu wird die allgemeinste Substitution σ mit $t^\sigma = u^\sigma$ berstimmt, der *allgemeinster Unifizierer*. Der Typ der Funktionsanwendung FA ist dann r^σ, also der Resultattyp r von F, der mit der Substitution σ instanziiert wird.

Beispiel. Der Ergebnistyp der Funktionsanwendung `id '*'` ist `Char`, weil das Argument `'*'` vom Typ `Char` ist, die Funktion `id` vom Typ `a` $\to$ `a` und die Unifikation des Argumenttyps `a` mit `Char` mit der Substitution $\sigma = \{$`a` $\mapsto$ `Char`$\}$ gelingt.

Seien also Typen, Substitutionen und die Unifikationsprozedur wie in Blatt 6 gegeben. Dann betrachten wir folgende geschachtelte Funktionsanwendungen: Ein *Term* ist entweder eine bloßer Bezeichner für eine Funktion (d.h. eine Konstante) oder eine Applikation eines Terms auf einen anderen (wobei der erste Term einen Funktionstyp haben muss). Der Datentyp dazu ist ein binärer Baum:

```
data Term = Const String | Appl Term Term
```

Jeder Konstanten ist ein (potentiell polymorpher) Typ zugeordnet, der in einer Umgebung nachgeschaut werden kann. Als Umgebung kann eine endliche Abbildung `Map String Type` verwendet werden.

Es soll nun der Typ eines Terms berechnet werden. Allein der Typ einer Konstanten ergibt sich aber nicht nur durch bloßes Nachschauen in der Umgebung. Der polymorphe Typ einer Konstanten muss mit "frischen" Typvariablen instanziiert werden. Zu diesem Zweck muss ein Zähler Ein- und Ausgabe der Typinferenzfunktion sein.

```
infer :: Env → Int → Term → Maybe (Int, Type)
```

Der Eingabezähler enthält immer die höchste bisher bekannte Nummer einer Typvariablen.

Alle Typvariablen in der Umgebung sind immer größer null, um ein einfaches Addieren des Eingabezählers zu ermöglichen.

Der Typ der Funktion `id` in der Umgebung (*env*) sei z.B.

```
TyCons "(→)" [TyVar 1, TyVar 1]
```

Wäre die höchste bekannte Nummer einer Typvariablen 3, dann erhielte man folgendes Ergebnis für `infer env 3 (Const "id")`:

```
Just (4, TyCons "(→)" [TyVar 4, TyVar 4])
```

Der Ergebniszähler ist das Maximum der Eingabe und der frischen Typvariablennummern. Zu beachten ist, dass frische Typvariablen für verschiedene Typvariablen verschieden bleiben müssen. Das Ergebnis zu einem Tupeltyp in der Umgebung: `TyCons "(,)"[TyVar 1, TyVar 2]` wäre also (ausgehend von 3):

```
Just (5, TyCons "(,)" [TyVar 4, TyVar 5])
```

Dieses Erzeugen von frischen Typvariablen ist nötig, damit der gleiche Funktionsname mit unterschiedlichen Instanzen in einem Applikationsterm vorkommen kann. Zum polymorphen Typ `TyCons "(→)"[TyVar 1, TyVar 1]` von `id` aus der Umgebung müssen also für zwei Vorkommen von `id` in einem Applikationsterm zwei unterschiedliche Instanzen erzeugt werden, z.B.:

`TyCons "(→)"[TyVar 4, TyVar 4]` und `TyCons "(→)"[TyVar 5, TyVar 5]`

Dabei stehen die frischen Typvariablen für noch unbekannte Typen.

Um nun den Typ einer Applikation zu inferieren, wird rekursiv der Typ der Funktion und des Arguments berechnet, wobei der Ergebniszähler der ersten rekursiven Berechnung Eingabe für die zweite ist.

Der Typ der Funktion muss natürlich ein Funktionstyp sein, den man am Typkonstruktor `(→)` und den zwei Argumenttypen erkennt![1] Ist der Typ der Funktion nicht von dieser Form, dann hat man einen Typfehler gefunden (das Ergebnis ist `Nothing`), andernfalls muss der erste Argumenttyp mit dem Typ des Arguments unifiziert werden. Schlägt die Unifikation fehl, liegt wieder ein Typfehler vor; anderfalls kann die Ergebnissubstitution auf den zweiten Argumenttyp vom Funktionstyp angewendet werden, um den Ergebnistyp zu liefern.

Testen Sie die Typinferenz am Beispiel `id id '*'`:

[1] Den exotischen Fall, dass dieser Typ eine Typvariable a ist, die erst durch einen Funktionstyp $a \to b$ substituiert werden muss, darf ignoriert werden.

```
Appl (Appl (Const "id") (Const "id")) (Const "'*'")
```

mit `id` wie oben und `'*'` vom Typ `TyCons "Char"[]`. Enthält der Ergebnistyp keine Variablen mehr, hat man einen monomorphen Ergebnistyp gefunden und kann den Ausgabezähler ignorieren.

20 *Überladen von Funktionen erster Ordnung* *4 Punkte*

Die obige polymorphe Typinferenz für Funktionen höherer Ordnung wollen wir einer Typinferenz für monomorphe Funktionen erster Ordnung mit Überladung gegenüberstellen. Zur Repräsentation von Typen erster Ordnung benötigen wir nur einfache Zeichenketten, während Funktionen durch ihren Namen, ihre Argumenttypen und ihren Ergebnistyp charakterisiert werden. Die Applikationsterme für Funktionen erster Ordnung sind ein Spezialfall der sogenannten *Rose-Trees*:

```
data Term1 = Term1 String [Term1]
```

D.h ein solcher Term ist ein Funktions- oder Konstantenname mit einer Liste von Argumententermen. Den Namen wollen wir für die Typinferenz wieder in einer Umgebung nachsehen. Da aber ein Name nur in Verbindung mit den Argumenttypen eine Funktion eindeutig bezeichnet, brauchen wir ein verschachtelte Umgebung

```
type NextEnv = Map String (Map [String] String)
```

Ist z.B. die zweistellige Funktion `plus` für `Int` und `Float` definiert, dann hätten wir folgende geschachtelte Umgebung:

```
fromList [("plus", fromList [(["Int","Int"],"Int"),
                             (["Float","Float"] "Float")]),
          ("1",    fromList [([], "Int")]),
          ("1.0",  fromList [([], "Float")])                ]
```

Programmieren Sie die Typinferenz bzw. die Überladungsauflösung für solche Terme:

```
resolve :: NestEnv → Term1 → Either String String
```

Statt `Maybe` und `Nothing` nutzen Sie `Either` und `Left`, um Typfehler genauer zu beschreiben. Ein `Right`-Ergebnis enthält den Ergebnistyp als String.

Tipps:

- Wer mit seiner Implementierung von Blatt 6 nicht zufrieden ist, darf statt dessen die Musterlösung auf der Webseite benutzen.

[Die Idee zu dieser Aufgabe stammt von **Christian Maeder**.]

Dieses ist Version 3 vom 7. Januar 2010.