Fakultät IV Elektrotechnik und Informatik
Informatik 1 WS 2001/02, Jähnichen / Flik / Lechtchinsky / Neunast / Tutoren

Aufgabenblatt 7

letzte Aktualisierung: 08. December, 12:18

Ausgabe: *7.12.2001*
Abgabe: *07./08.01.2002* *Prozent:* *100*

Thema: *Bäume; Auswertung von Ausdrücken*

Der zweite von drei Tests findet in den Tutorien am 07./08.01.02 statt.

1. Aufgabe (10 Prozent): Typinferenz

Diese Aufgabe steht nicht in einem inhaltlichen Zusammenhang mit den restlichen Aufgaben des Übungsblattes. Sie ist allerdings notwendig, da bei der Bearbeitung des 5. und 6. Aufgabenblattes noch erhebliche Probleme bei der Ableitung eines Typs aus einer gegebenen Anwendung auftraten.

1.1. (Tut) Gegeben sei eine Funktion `f` sowie die Funktionen `+'Nat`, `succ'Nat` und die Konstante `2'Nat`. `f1` und `f2` dienen als Hilfsfunktionen von `f`.

```
FUN f: nat -> (nat -> nat) -> nat ** nat -> nat
DEF f == \\a. \\b. \\c,d. ( g(2+f1(a)(b(e),c))
                           WHERE
                           (e,g) == f2(c,d,b) )
```

Welche Funktionalitäten müssen `f1` und `f2` haben, damit die Definition von `f` typkorrekt ist? Welche Typen haben die Ausdrücke `f`, `f(2)`, `f(2)(succ)` und `f(2)(succ)(2,2)`?

1.2. Permutationen einer Sequenz (10 Prozent) Wir betrachten eine kompakte Funktion `perms`, mit der alle möglichen Permutationen der Elemente einer Sequenz natürlicher Zahlen generiert werden können (wobei die Sequenz keine mehrfach auftretenden Elemente enthalten sollte, da es sonst in der Ergebnissequenz zu Wiederholungen kommt).

Die Hilfsfunktion ‚`-`' löscht dabei aus einer Sequenz natürlicher Zahlen das erste Auftreten eines vorgegebenes Elements und hat die Funktionalität `FUN - : seq[nat] ** nat -> seq[nat]`.

Die (polymorphe) Funktion `fold` verallgemeinert ihr Gegenstück von Blatt 6 auf beliebige Typen und besitzt die Funktionalität `FUN fold:` $(\alpha$ `**` β `->` $\alpha)$ `**` α `->` `seq[`β`]` `->` α mit α und β als Typparametern.

```
FUN perms: seq[nat] -> seq[seq[nat]]
DEF perms(S)  ==
  IF <>?(S) THEN
    <> :: <>
  ELSE
    fold(\\a1,b1. fold(\\a2,b2. (b1::b2)::a2,a1)(perms(S-b1)),<>)(S)
  FI
```


Es gilt beispielsweise `perms`$(\langle 1,2,3\rangle) = \langle\langle 3,1,2\rangle,\langle 3,2,1\rangle,\langle 2,1,3\rangle,\langle 2,3,1\rangle,\langle 1,2,3\rangle,\langle 1,3,2\rangle\rangle$.

Um Teilausdrücke eindeutig benennen zu können, schreiben wir diese Funktion in semantisch äquivalenter Weise um:

```
DEF perms(S)  ==
  IF <>?(S) THEN
    empty1 :: empty2
    WHERE
    empty1 == <>
    empty2 == <>
  ELSE
    result
    WHERE
    result == foldf1(S)
    foldf1 == fold(f1,empty3)
    empty3 == <>
    f1     == \\a1,b1. ( foldf2(perms(S-b1))
                         WHERE
                         foldf2 == fold(f2,a1)
                         f2     == \\a2,b2. (b1 cons1 b2) cons2 a2 )
    cons1  == \\b1,b2. b1::b2
    cons2  == \\b1_cons1_b2,a2. b1_cons1_b2::a2
  FI
```

Gebt tabellarisch die Typen der Teilausdrücke `empty1`, `empty2` , `result`, `foldf1`, `empty3`, `f1`, `foldf2`, `f2`, `cons1` und `cons2` an.

2. Aufgabe (Tut): Einfache Ausdrücke

Die zweite bis sechste Aufgabe dieses Übungsblattes beinhalten Teilabschnitte einer größeren Aufgabe. Es soll eine Datenstruktur implementiert werden, die sowohl einfache Ausdrücke, als auch Variablen und Lambda-Ausdrücke speichern kann. Diese Datenstruktur soll später zur Evaluierung benutzt werden. (Die Auswertung von Rekursionen fällt unter die freiwilligen Zusatzaufgaben.) Diese Datenstruktur soll des weiteren so strukturiert werden, dass sie in verschiedenen Modulen zusammengefasst werden kann. Zuletzt soll eine Ausgabefunktion für jeden in dieser Datenstruktur gespeicherten Ausdruck implementiert werden.

2.1. Repräsentation von Ausdrücken (Tut) Entwerft einen Datentyp, mit dessen Hilfe Ausdrücke repräsentiert werden können.

2.2. Auswertung von Ausdrücken (Tut) Implementiert eine Funktion `evaluate`, die einen solchen Ausdruck auswertet. Verwendet dabei den Datentyp `safeNat` von Aufgabenblatt 6, um Fehler darstellen zu können.

3. Aufgabe (30 Prozent): Variablen

3.1. `let`-Bindungen (Tut) Wie muß der Datentyp `expr` erweitert werden, um Variablen und `let`-Bindungen repräsentieren zu können? Was muß an der Funktion `evaluate` verändert werden, damit auch solche Ausdrücke ausgewertet werden können?

3.2. Modularisierung (Tut) Wie kann der Interpreter sinnvoll modularisiert werden?

3.3. Speichern von Bindungen (30 Prozent) Implementiert die Struktur `Environment`, die den abstrakten Datentyp `environment` zum Speichern von Variablenbindungen zur Verfügung stellt. Die Struktur soll die folgenden Funktionen zur Verfügung stellen:

`FUN emptyenv : environment`
liefert ein leeres Environment.

`FUN lookup : environment ** denotation -> safeNat`
liefert die Bindung der gegebenen Variablen oder einen Fehler.

`FUN bind : environment ** denotation ** safeNat -> environment`
erweitert ein Environment um die gegebene Bindung.

4. Aufgabe (40 Prozent): Lambda-Ausdrücke

4.1. Lambda-Ausdrücke (Tut) Wie muß der Interpreter erweitert werden, um einstellige Lambda-Ausdrücke repräsentieren und auswerten zu können?

4.2. Repräsentation von Lambda-Ausdrücken und Funktionswerten (20 Prozent) Modifiziert den Interpreter so, daß Lambda-Ausdrücke, Funktionswerte und Funktionsanwendungen repräsentiert werden können. Definiert hierzu einen Datentyp `safeVal`, der natürliche Zahlen, Funktionswerte und Fehler repräsentieren kann. Implementiert sichere Arithmetikfunktionen auf Werten vom Typ `safeVal`.

4.3. Evaluierung von Lambda-Ausdrücken (20 Prozent) Modifiziert die Funktion `eval` so, daß Lambda-Ausdrücke und Funktionsanwendungen verarbeitet werden können.

5. Aufgabe (20 Prozent): Pretty-Printing

Um die Funktionsweise des Interpreters überprüfen zu können, sollen alle bisher definierten Typen auch ausgegeben werden.

5.1. Ausgabe von `SafeVal` (Tut) In OPAL können Typen üblicherweise mit der Funktion ‘ zu einer `denotation` umgewandelt werden (OASYS verwendet standardmäßig diese Funktion, um Werte selbstdefinierter Datentypen auszugeben). Implementiert die Funktion ‘ für den Typ `safeVal`.

5.2. Ausgabe von Ausdrücken (10 Prozent) Implementiert die Funktion ‘, die einen Ausdruck in eine Denotation umwandelt. Dabei sollen `let`-Ausdrücke wie folgt eingerückt werden:

```
LET y == x
IN
  LET z == y
  IN
    z
```

Die binären Operationen `plus`, `minus`, `mult` und `div` sollen dabei durch die Infix-Operatoren $+$, $-$, $*$ und / repräsentiert werden.

5.3. Präzedenzen (10 Prozent) Erweitert die Funktion ‘ so, daß die Präzedenzen der Operatoren berücksichtigt und unnötige Klammern vermieden werden.

6. Aufgabe: Freiwillige Zusatzaufgaben

Da dieses Aufgabenblatt eine Bearbeitungszeit von fast einem Monat hat (Ferien mitgerechnet), sind die folgenden Aufgaben für all diejenigen gedacht, die ihren Interpreter noch weiter verbessern möchten. Die Bearbeitung dieses Teil ist freiwillig, es werden also keine Punkte angerechnet.
Er bietet jedoch tiefere Einblicke in die Arbeitsweise funktionaler Programmierung und ist damit für all diejenigen interessant, die ein erweitertes Verständnis erwerben möchten.

6.1. Unterstützung rekursiver Funktionen (0 Prozent) Rekursive Funktionen benötigen im Unterschied zu allen bisherigen Ausdrücken zwingend ein Abbruchkriterium. Von daher wird die Implementierung einer IF-Abfrage notwendig.

Ansonsten muss auch bei einer rekursiven Funktion festgehalten werden, wann welcher Variable welcher Wert zugewiesen wird. Darüber hinaus sollte auch die Rekursionstiefe und der Programmteil festgehalten werden, von dem aus die Rekursion gestartet ist.

Es ist also sinnvoll, entweder `Environment` oder `Binding` zu ändern, um zusätzliche Information unterzubringen.

6.2. Implementierung in einer imperativen Sprache (0 Prozent) OASYS übersetzt derzeit OPAL Source-Code zu C Source-Code. Ein Interpreter, der in eine imperative Programmiersprache übersetzt wird, kann natürlich auch direkt in einer imperativen Programmiersprache geschrieben werden.

Implementiert den in den Aufgaben 2. bis 5. enwickelten Interpreter in einer imperativen Programmiersprache eurer Wahl. Vergleicht den erforderlichen Entwicklungs- und Implementationsaufwand beider Varianten.