3. Übungsblatt zu Programmierung 1, WS 2010/11

Prof. Dr. Gert Smolka, Christian Doczkal, M.Sc.
www.ps.uni-sb.de/courses/prog-ws10/

Lesen Sie im Buch: Kapitel 3

Aufgabe 3.9 Deklarieren Sie eine Prozedur $prod : (int \to int) \to int \to int$, die für $n \geq 0$ die Gleichung

$$prod\, f\, n = 1 \cdot (f\, 1) \ldots \cdot (f\, n)$$

erfüllt. Deklarieren Sie außerdem mithilfe von *prod* eine Prozedur $fac : int \to int$, die für $n \geq 0$ die Fakultät $n!$ berechnet (siehe Aufgabe 1.26 auf S. 21). Die Prozedur *fac* soll nicht rekursiv sein.

Aufgabe 3.10 Deklarieren Sie mithilfe der höherstufigen Prozedur *sum* eine Prozedur $sum' : (int \to int) \to int \to int \to int$, die für $k \geq 0$ die Gleichung

$$sum'\, f\, m\, k = 0 + f\,(m+1) \cdot \cdots + f\,(m+k)$$

erfüllt. Die Prozedur sum' soll nicht rekursiv sein.

Aufgabe 3.11 Geben Sie die Baumdarstellung des folgenden Typs an:

$$(int \to bool) \to (bool \to real) \to int \to real$$

Aufgabe 3.12 Geben Sie geschlossene Abstraktionen an, die die folgenden Typen haben:

a) $(int * int \to bool) \to int \to bool$
b) $(int \to real) \to (bool \to int) \to int * bool \to real * int$

Die Abstraktionen sollen nur mit Prozeduranwendungen, Tupeln und Bezeichnern gebildet werden. Konstanten und Operatoren sollen nicht verwendet werden.

Aufgabe 3.14 Deklarieren Sie eine Prozedur $mul : int \to int \to int$, die das Produkt zweier Zahlen x und $n \geq 0$ gemäß der Gleichung

$$x \cdot n = 0 \underbrace{+ x \ \ldots \ + x}_{n\text{-mal}}$$

durch Addieren berechnet. Die Prozedur *mul* soll mithilfe der Prozedur *iter* formuliert werden und nicht rekursiv sein.

Aufgabe 3.15 Geben Sie eine Abstraktion *e* an, sodass die Ausführung des Ausdrucks *first x e* für alle *x* divergiert.

Aufgabe 3.16 Deklarieren Sie mit *first* eine Prozedur $divi : int \to int \to int$, die zu $x \geq 0$ und $m > 0$ dasselbe Ergebnis liefert wie Division mit dem Operator *div*. Hinweis: Für $x \geq 0$ und $m > 0$ liefert *div* die größte ganze Zahl k mit $k \cdot m \leq x$.

Aufgabe 3.20 Deklarieren Sie polymorphe Prozeduren wie folgt:

$$\begin{aligned} id: &\quad \forall\alpha.\ \alpha \to \alpha \\ com: &\quad \forall\alpha\ \beta\ \gamma.\ (\alpha \to \beta) \to (\beta \to \gamma) \to \alpha \to \gamma \\ cas: &\quad \forall\alpha\ \beta\ \gamma.\ (\alpha * \beta \to \gamma) \to \alpha \to \beta \to \gamma \\ car: &\quad \forall\alpha\ \beta\ \gamma.\ (\alpha \to \beta \to \gamma) \to \alpha * \beta \to \gamma \end{aligned}$$

Ihre Prozeduren sollen nicht rekursiv sein. Machen Sie sich klar, dass die angegebenen Typschemen das Verhalten der Prozeduren eindeutig festlegen.

Aufgabe 3.21 Dieter Schlau ist ganz begeistert von polymorphen Prozeduren. Er deklariert die Prozedur

```
fun 'a pif (x:bool, y:'a, z:'a) = if x then y else z
val α pif : bool * α * α → α
```

und behauptet, dass man statt eines Konditionals stets die Prozedur *pif* verwenden kann:

$$\textit{if } e_1 \textit{ then } e_2 \textit{ else } e_3 \quad \rightsquigarrow \quad \textit{pif}(e_1, e_2, e_3)$$

Was übersieht Dieter? Denken Sie an rekursive Prozeduren und daran, dass der Ausdruck, der das Argument einer Prozeduranwendung beschreibt, vor dem Aufruf der Prozedur ausgeführt wird.

Aufgabe 3.24 (Let und Abstraktionen) Dieter Schlau ist begeistert. Scheinbar kann man Let-Ausdrücke mit Val-Deklarationen auf Abstraktion und Applikation zurückführen:

$$\textit{let val } x = e \textit{ in } e' \textit{ end} \quad \rightsquigarrow \quad (\textit{fn } x\colon t \Rightarrow e')\,e$$

Auf der Heimfahrt von der Uni kommen ihm jedoch Zweifel an seiner Entdeckung, da ihm plötzlich einfällt, dass Argumentvariablen nicht polymorph getypt werden können. Können Sie ein Beispiel angeben, das zeigt, dass Dieters Zweifel berechtigt sind? Hilfe: Geben Sie einen semantisch zulässigen Let-Ausdruck an, dessen Übersetzung gemäß des obigen Schemas scheitert, da Sie keinen passenden Typ t für die Argumentvariable x finden können.

Aufgabe 3.27 Geben Sie Deklarationen an, die monomorph getypte Bezeichner wie folgt deklarieren:

a: $\textit{int} * \textit{unit} * \textit{bool}$

b: $\textit{unit} * (\textit{int} * \textit{unit}) * (\textit{real} * \textit{unit})$

c: $\textit{int} \to \textit{int}$

d: $\textit{int} * \textit{bool} \to \textit{int}$

e: $\textit{int} \to \textit{real}$

f: $\textit{int} \to \textit{real} \to \textit{real}$

g: $(\textit{int} \to \textit{int}) \to \textit{bool}$

Verzichten Sie dabei auf explizite Typangaben und verwenden Sie keine Operator- und Prozeduranwendungen.

Aufgabe 3.30 Betrachten Sie den Ausdruck

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

a) Geben Sie die Baumdarstellung des Ausdrucks an.
b) Markieren Sie die definierenden Bezeichnerauftreten durch Überstreichen.
c) Stellen Sie die lexikalischen Bindungen durch Pfeile dar.
d) Geben Sie alle Bezeichner an, die in dem Ausdruck frei auftreten.
e) Bereinigen Sie den Ausdruck durch Indizieren der gebundenen Bezeichnerauftreten.

Aufgabe 3.32 Betrachten Sie die bereinigte Deklaration

```
fun f (x, y) = (fn z => (fn u => (fn v => u) z) y) x
```

a) Geben Sie die Baumdarstellung der Deklaration an.
b) Stellen Sie die lexikalischen Bindungen der Deklaration durch Pfeile dar.
c) Geben Sie die statischen Bezeichnerbindungen an, die durch die semantische Analyse bestimmt werden.
d) Geben Sie eine möglichst einfache, semantisch äquivalente Deklaration an, die ohne Abstraktionen gebildet ist.

Aufgabe 3.33 (Exists) Deklarieren Sie eine Prozedur

$$\mathit{exists} : \mathit{int} \to \mathit{int} \to (\mathit{int} \to \mathit{bool}) \to \mathit{bool}$$

die testet, ob eine Prozedur für mindestens eine Zahl zwischen zwei gegebenen Zahlen den Wert *true* liefert. Orientieren Sie sich an der Prozedur *forall*.

Aufgabe 3.36 Schreiben Sie eine Prozedur $\mathit{reduce} : \mathit{int} \to \mathit{int} \to \mathit{int} * \mathit{int}$, die zu zwei Zahlen $n, p \geq 2$ das eindeutig bestimmte Paar (m, k) liefert, sodass $n = m \cdot p^k$ gilt und m nicht durch p teilbar ist.

Aufgabe 3.37 Deklarieren Sie mit *iterup* eine Prozedur

a) *power*, die zu x und n die Potenz x^n liefert.
b) *fac*, die zu $n \geq 0$ die n-te Fakultät $n!$ liefert.
c) *sum*, die zu f und n die Summe $0 + f\,1 \ldots + f\,n$ liefert.
d) *iter′*, die zu n, s und f dasselbe Ergebnis liefert wie *iter n s f*.