Technische Universität München
Institut für Informatik
Prof. Tobias Nipkow, Ph.D.
Florian Haftmann

SS 2008
30. 05. 2008

Perlen der Informatik 2
4. Übung

1 Zum Warmwerden

▷ Definieren sie die Fakultätsfunktion `fac` auf natürlichen Zahlen primitiv rekursiv. Sie können dazu die Multiplikation `m * n` auf natürlichen Zahlen verwenden.

▷ Definieren sie die Folge der Fibonacci-Zahlen f_n als Funktion `fib :: nat ⇒ nat`. Die Fibonacci-Zahlen gehorchen folgender Rekursionsvorschrift:

f_0 `= 0`
f_1 `= 1`
f_n `=` f_{n-1} `+` f_{n-2}

▷ Definieren sie eine Funktion `sum`, die eine Liste natürlicher Zahlen aufsummiert. Die Additon `m + n` auf natürlichen Zahlen ist bereits definiert.

▷ Zeigen Sie die folgenden Aussagen:

```
sum_append: sum (xs @ ys) = sum xs + sum ys
sum_rev: sum (rev xs) = sum xs
```

Eine Liste von Listen lässt sich durch Aneinanderhängen der einzelnen Listen »flachklopfen«. Dazu gibt es bereits eine vordefinierte Funktion in der Listentheorie `List.thy`. Weiterhin liefert die Funktion `length` die Länge einer Liste.

▷ Formulieren und beweisen Sie: die Länge einer flachgeklopften Liste von Listen ist die Summe der Längen der einzelnen Listen. Zum Formulieren ist die Funktion `map` hilfreich (oder definieren sie eine geeignete Hilfsfunktion).

▷ Zeigen Sie eine ähnliche Eigenschaft über flachgeklopfte Listen und `sum`.

2 Assoziationslisten

Assoziationslisten sind Listen von Paaren, die endliche partielle Abbildungen repräsentieren.

▷ Definieren sie zwei Operationen

```
lookup :: ('a × 'b) list ⇒ 'a ⇀ 'b
update :: 'a × 'b ⇒ ('a × 'b) list ⇒ ('a × 'b) list
```

zum Nachschlagen von Werten und Aktualisieren von Assoziationslisten.

▷ Beweisen oder widerlegen Sie:

lookup_update: *lookup (update (k, v) xs) k = Some v*

update_swap: *x* $\neq$ *y* $\Longrightarrow$

lookup (update (x, a) (update (y, b) xs)) z =

lookup (update (y, b) (update (x, a) xs)) z

▷ Definieren Sie eine Funktion *delete* :: *'a* $\Rightarrow$ *('a* $\times$ *'b) list* $\Rightarrow$ *('a* $\times$ *'b) list*, die einen Eintrag aus einer Assoziationsliste löscht.

▷ Zeigen oder widerlegen Sie: *lookup_delete_same*: *lookup (delete k xs) k = None*

Unsere Assoziationslisten haben das Problem, dass sich bei wiederholten Updates jedesmal neuer Müll in der Liste ansammelt, der nicht mehr relevant ist.

▷ Definieren sie eine Funktion *normalize*, die alle nicht mehr benötigten Paare aus der Liste löscht.

Man erwartet natürlich, dass die normalisierten Listen sich noch genauso verhalten wie das Original.

▷ Zeigen Sie die entscheidende Korrektheitseigenschaft: *lookup_normalize*: *lookup (normalize xs) k = lookup xs k*. Wahrscheinlich benötigen Sie hier wieder ein geeignetes Hilfslemma, eine Eigenschaft über das Zusammenspiel von *delete* und *lookup*.

3 Ein verifizierter Mini-Compiler

3.1 Quellspache: Einfache artihmetische Ausdrücke

Wie definieren einfache arithmetische Ausdrücke über den ganzen Zahlen. Diese bestehen aus Numeralen (≫-7≪, ≫42≪), Variablen (die wir der Einfachheit halber mit natürlichen Zahlen bezeichnen) und den binären Operationen *Add* und *Mult* (siehe hierzu auch Abschnitt 3.3. des Tutorials). Repräsentiert werden diese Ausdrücke durch folgenden Datentyp:

```
datatype aexp =
  Num int
  | Var nat
  | Add aexp aexp
  | Mult aexp aexp
```

Der Wert von Variablen wird aus einer *Umgebung* (environment) abgerufen. Das ist einfach eine Funktion von Variablenname (bei uns: natürliche Zahl) auf Wert (ganze Zahl):

```
types
  env = "nat ⇒ int"
```

Um die Semantik dieser Ausdrücke zu erklären, definieren wir eine Auswertungsfunktion:

```
consts
  eval :: "env ⇒ aexp ⇒ int"
```

```
primrec
  "eval env (Num n) = n"
  "eval env (Var v) = env v"
  "eval env (Add e1 e2) = eval env e1 + eval env e2"
  "eval env (Mult e1 e2) = eval env e1 * eval env e2"
```

Wenn von vornherein feststeht, in welcher Umgebung ein Ausdruck ausgewertet werden soll, kann man die Werte der Variablen auch direkt in den Ausdruck einsetzen und erhält einen Ausdruck ohne Variablen.

▷ Definieren Sie dazu eine Funktion `elim_var`.

▷ Zeigen Sie: Die Auswertung eines Ausdrucks in einer Umgebung liefert das gleiche Ergebnis wie die Auswertung des gleichen Ausdrucks, in dem zuvor alle Variablen eliminiert worden sind, in einer beliebigen (anderen) Umgebung.

3.2 Zielarchitektur: Eine einfache Stackmaschine

Nun definieren wir eine Stackmaschine, die einfache arithmetische Berechnungen ausführen kann. Sie hat folgenden Befehlssatz:

```
datatype instr =
  IStore int
  | IVar nat
  | IAdd
  | IMult
```

Dabei haben die einzelnen Instruktionen folgende Semantik:

1. `IStore k`: Lege den Wert `k` auf den Stack
2. `IVar n`: Lege den Wert der Variablen `n` auf den Stack
3. `IAdd`: Nimm die beiden obersten Werte vom Stack, addiere sie und lege das Ergebnis wiederum auf den Stack
4. `IMult`: Nimm die beiden obersten Werte vom Stack, multipliziere sie und lege das Ergebnis wiederum auf den Stack

Die Stackmaschine verarbeitet in einem Schritt eine Instruktion `instr` innerhalb einer Umgebung `env` auf einem Eingangszustand des Stack und endet mit einem Ausgangszustand des Stack. Den Stack modellieren wir als `int list`.

Zum Abarbeiten der Instruktionen `IAdd` und `IMult` benötigen wir eine Hilfsfunktion, die zwei Elemente vom Stack nimmt, sie mit einer Operation verknüpft und das Ergebnis wieder auf den Stack legt. Für diese (nicht-rekursive) Definition können wir **definition** verwenden.

```
definition apply_stack :: "('a ⇒ 'a ⇒ 'a) ⇒ 'a list ⇒ 'a list" where
  "apply_stack f xs = f (hd xs) (hd (tl xs)) # tl (tl xs)"

consts
  exec :: "env ⇒ instr ⇒ int list ⇒ int list"
```

primrec

```
"exec env (IStore n) s  = n # s"
"exec env (IVar v) s = env v # s"
"exec env IAdd s = apply_stack (op +) s"
"exec env IMult s = apply_stack (op *) s" —
```
[1]

Bislang haben wir nur definiert, was in einem Verarbeitungsschritt (*instr*) passiert. Die Stackmaschine soll aber eine Liste von Instruktionen hintereinander abarbeiten:

lexec :: *env* ⇒ *instr list* ⇒ *int list* ⇒ *int list*,

▷ Definieren Sie *lexec* in geeigneter Weise!

3.3 Der Compiler

Wir werden nun einen kleinen Compiler von arithmetischen Ausdrücken in die Sprache der Stackmaschine schreiben und seine Korrektheit zeigen.

Der Compiler übersetzt arithmetische Ausdrücke in Listen von Instruktionen für die Stackmaschine.

▷ Geben Sie die Definition von *compile* :: *aexp* ⇒ *instr list*, an!

▷ Formulieren und Beweisen Sie die Korrektheit des Compilers. Dazu werden sie wahrscheinlich Hilfslemmata benötigen. Auch kann es nötig sein, die zu beweisende Aussage zu generalisieren. Hilfreiche Hinweise dazu finden Sie in Abschnitt 3.2. des Tutorials.

Hinweis: am 6. Juni findet die Übung *ausnahmsweise* im Alonzo Church (01.09.014) statt.

[1] Wie in ML bazeichnet die Syntax *op +*, dass ein Infix-Operator in Präfix-Position verwendet wird.