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

SS 2008
06. 06. 2008

Perlen der Informatik 2
5. Übung

1 Ein verifizierter Mini-Compiler

1.1 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.

1.2 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!

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

1.3 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.

1.4 Arithmetische Ausdrücke mit Seiteneffekten

- Sehen Sie sich den Abschnitt über `fun_upd` der Theorie `Fun` in der Isabelle-Library an.
- Erweitern Sie den Datentyp `aexp` um eine Alternative `Assign`. Sie soll für Zuweisungen `x = e` stehen, die einer Variable `x` den Wert des Ausdrucks `e` geben. Der Wert der Zuweisung selbst ist dabei ebenfalls der Wert der rechten Seite.
- Ändern Sie die Auswertungsfunktion `eval` so, dass sie mit den neuen Ausdrücken mit Seiteneffekten zurecht kommt. Da ein Ausdruck jetzt nicht nur einen Wert besitzt, sondern auch die Belegung von Variablen ändern kann, werden Sie den (Rückgabe-)typ von `eval` ändern müssen.
- Definieren Sie eine Funktion `is_const :: aexp ⇒ bool`, die Ausdrücke ohne Seiteneffekte identifiziert. Beweisen Sie: wenn `is_const e` gilt, ändert die Auswertung von `e` die Variablenbelegung nicht.
- Erweitern Sie den Datentyp `instr` der Maschineninstruktionen um einen Befehl `IUpd x`, der den obersten Wert auf dem Stack in die Variable `x` schreibt (der Wert selbst soll auf dem Stack liegen bleiben). Passen Sie die Definitionen der Maschinen-Semantik `exec` und `lexec` entsprechend an. Sie werden auch hier die (Rückgabe-)typen ändern müssen.
- Passen Sie den Compiler den neuen Ausdrücken und Instruktionen an. Beweisen Sie die Korrektheit Ihres Compilers.