Types Summer School 2007
Coq-lab
Verification of a Mini-Compiler

The purpose of this exercise is to use Coq to verify the correctness of a mini-compiler. The source language is a single expression involving integer constants, variables and additions. The target language is a assembly-like language with a single accumulator and an infinite set of registers. A template file for this exercise is available at

`www.lri.fr/~filliatr/types-summer-school-2007/compiler.v`

where the occurrences of the comment `(* TO BE COMPLETED *)` must be replaced by Coq definitions, statements or proofs.

1 Source Language

1.1 Syntax

The abstract syntax of the input language is the following:

$$\begin{array}{rll} e ::= & n & \text{literal constant} \\ \mid & x & \text{variable} \\ \mid & e+e & \text{addition} \end{array}$$

Q1. Define an inductive type `expr:Set` for the abstract syntax of the source language. Literal constants will have type `nat`. Variable names will be represented by an abstract type `string:Set`.

1.2 Semantics

Given a state s assigning values to variables, the semantics E of an expression is immediate. It is recursively defined as follows:

$$\begin{array}{rcl} E(n) & = & n \\ E(x) & = & s(x) \\ E(e_1+e_2) & = & E(e_1)+E(e_2) \end{array}$$

Q2. The type of states is defined as `state:=string->nat`. Define the semantics of the source language as a function `E` of type `state->expr->nat`.

2 Target Language

The target language is an assembly-like language with an accumulator and an infinite set of registers.

2.1 Syntax

The syntax of the target language is simply a list of instruction. The abstract syntax of instructions is the following:

$$
\begin{array}{rl}
i ::= & \texttt{LI}\ n \\
| & \texttt{LOAD}\ r \\
| & \texttt{STO}\ r \\
| & \texttt{ADD}\ r
\end{array}
$$

where n is a literal constant and r a register name.

Q3. Define an inductive type `instr:Set` for the abstract syntax of the target language. Register names will be represented by natural numbers (type `nat`).

2.2 Semantics

The semantics of the four instructions is the following:

- `LI` n loads the immediate value n in the accumulator;
- `LOAD` r loads the contents of register r in the accumulator;
- `STO` r stores the contents of the accumulator in register r;
- `ADD` r adds the contents of register r to the accumulator.

Q4. Define an inductive type `cell:Set` to represent either the accumulator or some register. A state of the assembly machine is called a *store*. The type of stores is simply defined as `store := cell -> nat` *i.e.* a store is a mapping from cells to values.

Q5. Define a function `update : store -> cell -> nat -> store` which updates some given store by assigning a value to a cell. Hint: You need a decidable equality over cells to define this function. So you have to prove first the following lemma:

```
Lemma cell_eq_dec : forall c1 c2 : cell, {c1 = c2} + {c1 <> c2}.
```

Q6. Define the semantics of a single instruction as a store transformer `Si : store -> instr -> store`. Then define the semantics of a list of instructions as another store transformer `Sl : store -> list instr -> store`.

3 Compilation

The compilation schema is simple: different variables are mapped to different registers and, since there is a finite number of variables, the infinitely many remaining registers can be used to perform the evaluation of the expression.

3.1 The Compiler

Let e be an expression and m an assignment from its variables to registers. Let r be a register greater than those used in m for the variables of e. Then the compilation of e is a list of instructions $C_r(e)$ defined as follows:

$$\begin{aligned} C(n) &= \texttt{LI}\ n \\ C(x) &= \texttt{LOAD}\ m(x) \\ C(e_1+e_2) &= C_r(e_1) \mathbin{+\!\!+} \texttt{STO}\ r \mathbin{+\!\!+} C_{r+1}(e_2) \mathbin{+\!\!+} \texttt{ADD}\ r \end{aligned}$$

where $\mathbin{+\!\!+}$ denotes the catenation of lists. When this list of instructions is executed, it stops with the value of e in the accumulator.

Q7. The assignment m is simply defined as a function of type `symt := string -> nat`. Define the compiler as a recursive function `C : symt -> nat -> expr -> list instr`.

3.2 Correctness

We are now going to prove the correctness of this compiler. The correctness statement is the following:

> Let e be an expression, s a state, m an assignment from variables to registers, s' a store and r a register. If for any variable x we have
>
> - $m(x) < r$;
> - x has the same value in s than $m(x)$ in s'
>
> then the list of instructions $l = C_r(e)$ is such that
>
> - the execution of l in store s' ends up with an accumulator containing the value of e in s;
> - any register smaller than r is untouched by the execution of l.

Q8. State this theorem above in Coq.

Q9. Prove this theorem. Here are some hints:

- It is useful to prove that `Sl` s $(l_1 \mathbin{+\!\!+} l_2) =$ `Sl` (`Sl` s l_1) l_2 and to use this lemma to simplify some statements (using `repeat rewrite` for instance).
- It may be useful to prove the second part of the theorem first *i.e.* that registers smaller than r are untouched by the execution. It is indeed independent of the first part of the theorem and needed to prove the first part.