Technische Universität München
Institut für Informatik
Prof. Tobias Nipkow, Ph.D.
Brian Huffman, Peter Lammich

WS 2011/12
2. 12. 2011

Semantics of Programming Languages

Exercise Sheet 7

Exercise 7.1 Type coercions

Adding and comparing integers and reals can be allowed by introducing implicit conversions: Adding an integer and a real results in a real value, comparing an integer and a real can be done by first converting the integer into a real. Implicit conversions like this are called *coercions*.

1. Modify, in the theory *Types*, the inductive definitions of *taval* and *tbval* such that implicit coercions are applied where necessary.
2. Extend the datatype *com* by a loop construct *DO a TIMES c* which executes the command *c* exactly *a* times, where *a* is an arbitrary arithmetic expression of integer type.
3. Adapt all proofs in the theory *Types* accordingly.

Hint: Isabelle already provides the coercion functions *nat*, *int*, and *real*.

Homework 7 Register Machine for Arithmetic Expressions

Submission until Wednesday, December 14, 2011, 12:00 (noon).

Compiler.thy defines a function *acomp* that compiles an arithmetic expression, resulting in a program that runs on a stack machine. In this homework, you will define and verify a similar compiler for a different machine architecture: the *register machine*.

The register machine does not have a stack, but instead uses a set of registers to store values. Registers are indexed by type *reg*, which is a synonym for type *nat*. (Effectively, the machine has infinitely many registers.) The contents of the entire register file can be modeled as a function from registers to values (type *mstate*).

type_synonym *reg* = *nat*
type_synonym *mstate* = *"reg ⇒ val"*

The instruction set for the register machine comprises three instructions:

- *LOAD r x* stores the value of variable x in register r.
- *LOADI r v* stores the immediate value v in register r.
- *ADD* r_{dest} r_1 r_2 reads values from registers r_1 and r_2, and stores their sum in register r_{dest}.

datatype *rinst* = *LOAD reg vname* | *LOADI reg val* | *ADD reg reg reg*

The assignment consists of three parts. In each part you have much flexibility in choosing how to formalize everything.

1. Formalize the operational semantics of the register machine. The final machine state (type *mstate*) depends on the program (type *rinst list*), the state of the variables (type *state*), and the initial machine state. You may define it as a function or as an inductive relation, whichever you prefer.
2. Define a compiler for arithmetic expressions. This should take an expression of type *aexp* as input, and give back a program of type *rinst list*. Your compiler must ensure that when a value is saved in a register, it does not get overwritten before it needs to be used again. Therefore a simple function of type *aexp* ⇒ *rinst list* probably will not work; you may need to add extra input and/or output parameters to keep track of which registers to use and which not to use. (Don't worry about using registers efficiently—remember that you have an infinite supply of them, so use as many as you want.)
3. Formulate and prove a correctness property for your compiler. For any arithmetic expression *a* :: *aexp* and variable state *s* :: *state*, the compiled program for *a* should result in a final machine state where some appropriate register contains a value equal to *aval a s*. Of course, the formulation of this theorem must depend on the design of your compiler function.