Technische Universität München
Institut für Informatik
Prof. Tobias Nipkow, Ph.D.
Andrei Popescu, Peter Lammich

WS 2012/13
27. 11. 2012

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 Jump-Chain Optimization

Submission until Tuesday, December 4, 2012, 10:00am. One sub-task of this homework is optional, and gives 5 bonus points.

In this homework, you shall implement a transformation that eliminates chained jumps from machine code. To keep things simple, we write a function that only optimizes a jump at a given index *i*, and only follows chained jumps one level deep.

If you do not manage to prove a lemma, set a sorry there and try the remaining lemmas. Thus, you'll get a partial score, even if you cannot prove one of the main lemmas. This especially applies to the bonus-lemma!

First, write a function that updates the *i*th element of a list, and prove the lemmas below:

primrec *list_updatei* :: "'*a list* ⇒ *int* ⇒ '*a* ⇒ '*a list*" **where**
— Hint: It's easiest to define this function recursively over the list, similar to *op* !!

lemma *updatei_other*[*simp*]: "*i*≠*j* ⟹ *list_updatei l i x* !! *j* = *l* !! *j*"

lemma *updatei_this*[*simp*]: "⟦0≤*i*; *i* < *isize l*⟧ ⟹ *list_updatei l i x* !! *i* = *x*"

lemma *updatei_length*[*simp*]: *"length (list_updatei l i x) = length l"*

Then, write a function that looks at index i, and, if there is a jump that points to a jump, updates the jump offset to do both jumps at once. For simplicity, you only need to optimize unconditional jumps here.

Hints:

- Be careful if the first jump jumps out of the program, i.e., to a negative program counter, or a program counter beyond the size of the program. In those cases, you must not change the program.
- When computing the offset for the optimized jump, remember that jump offsets are relative to the instruction **after** the jump.

definition *opt_at* :: *"int ⇒ instr list ⇒ instr list"* **where**

To get a feeling for your function, show the following lemma:

lemma *opt_at_size*[*simp*]: *"isize (opt_at i P) = isize P"*

Now show that a step of the optimized program can be simulated by arbitrarily many (actually, one or two) steps of the original program:

lemma *sim_opt_at_aux*:
shows *"opt_at i P ⊢ c → c' ⟹ P ⊢ c →∗ c'"*

Hints: First, identify the cases where *opt_at* does not change the program. Then, handle the case that the program counter of c is not at index i. Finally, show the interesting case, where the modified instruction is actually executed.

Note that you do not need induction for this proof!

Use the above lemma to show that an arbitrary execution of the optimized program has a corresponding execution of the original program.

lemma *sim_opt_at*:
"opt_at i P ⊢ c →∗ c' ⟹ P ⊢ c →∗ c'"

Hint: To get the induction through, you must instantiate the induction rule:

apply (*induct "opt_at i P" _ _ rule*: *exec.induct*)

For 5 bonus points, show the opposite direction, i.e., that for any execution of the original program to a terminating configuration, there is also an execution of the original program to that terminating configuration:

lemma *opt_at_sim*:
"P ⊢ c →∗ (isize P,s_fin,stk_fin) ⟹
opt_at i P ⊢ c →∗ (isize P,s_fin,stk_fin)"

Hints:

- This proof is hard. Before you try this task, write a sorry here and complete the remaining exercise sheet.

- You will need an induction over the number of steps taken by the original program, such that you can apply the induction hypothesis for any smaller number of steps. See *Comp_Rev.thy* for an n-step version of *exec*, and theorem *nat_less_induct* for an appropriate induction rule.
- If the simplifier fails to prove obvious goals, there might be too many rules, e.g., *exec1_def*, in the simpset. Using *simp only*: or removing those rules from the simpset may help.

Similar to source-level program equivalence $\sim$, we can also define machine-level program equivalence. We use a big-step version here, that only regards the results of the machine programs

definition *mi_equiv* (**infix** "$\sim_m$" 50) **where**
"$P \sim_m P' \equiv$
$(\forall s\ stk\ s'\ stk'.$
$(P \vdash (0,s,stk) \rightarrow* (isize\ P,s',stk'))$
$\longleftrightarrow (P' \vdash (0,s,stk) \rightarrow* (isize\ P',s',stk'))$
)"

Show that $\sim_m$ is an equivalence relation:

lemma *mi_refl*[*simp*]:
lemma *mi_sym*:

lemma *mi_trans*[*trans*]:

Show that *opt_at* preserves equivalence

lemma *opt_at_correct*: "*opt_at i P* $\sim_m$ *P*"