Technische Universität München
Institut für Informatik
Prof. Tobias Nipkow, Ph.D.
Kevin Kappelmann, Lukas Stevens

WS 24/25
21.11.2024

Semantics of Programming Languages

Exercise Sheet 6

Exercise 6.1 Compiler optimization

A common programming idiom is *IF b THEN c*, i.e., the else-branch consists of a single *SKIP* command.

1. Look at how the program *IF Less (V ''x'') (N 5) THEN ''y'' ::= N 3 ELSE SKIP* is compiled by *ccomp* and identify a possible compiler optimization.
2. Implement an optimized compiler *ccomp2* which reduces the number of instructions for programs of the form *IF b THEN c*. Try to finish *ccomp2* without looking up *ccomp*!
3. Extend the proof of *comp_bigstep* to your modified compiler.

```
value "ccomp (IF Less (V ''x'') (N 5) THEN ''y'' ::= N 3 ELSE SKIP)"

fun ccomp2 :: "com ⇒ instr list" where
  "ccomp2 SKIP = []" |
  "ccomp2 (x ::= a) = acomp a @ [STORE x]" |
  "ccomp2 (c₁;;c₂) = ccomp2 c₁ @ ccomp2 c₂" |
  "ccomp2 (WHILE b DO c) =
   (let cc = ccomp2 c; cb = bcomp b False (size cc + 1)
    in cb @ cc @ [JMP (−(size cb + size cc + 1))])"

value "ccomp2 (IF Less (V ''x'') (N 5) THEN ''y'' ::= N 3 ELSE SKIP)"

lemma ccomp_bigstep:
  "(c,s) ⇒ t ⟹ ccomp2 c ⊢ (0,s,stk) →* (size(ccomp2 c),t,stk)"
```

Exercise 6.2 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*.

When doing this, all expressions will have a type – hence you can define *taval*/*tbval* as functions.

1. In the theory *HOL−IMP.Types* (copy it first), re-write the inductive definitions of *taval*/*tbval* as functions, and mody *atyping*/*btyping* such that implicit coercions are applied where necessary.
2. Adapt all proofs in the theory *HOL−IMP.Types* accordingly.

Hint: Isabelle already provides the coercion function *real_of_int* (*int* ⇒ *real*).

Homework 6.1 Compilation of exceptions

Submission until Wednesday, November 27, 23:59pm.

In the previous homework, we extended IMP with the exception throwing and handling constructs *THROW* and *ATTEMPT _ CLEANUP _*. In this homework you have to extend the command compiler *ccomp* to deal with these two constructs. The main idea is simple: a *THROW* is compiled to a *JMP* to the *CLEANUP* code. The new *ccomp* should have type *nat* ⇒ *com* ⇒ *instr list*. The additional *nat* parameter has a similar purpose as the *nat* parameter of function *bcomp*: it tells *ccomp* how far beyond the end of the generated code the code should jump in case of a *THROW*. If execution of the source code terminates with *SKIP*, execution of the compiled code should terminate 1 step beyond end of the compiled code; if execution of the source code terminates with *THROW*, execution of the compiled code should jump *n+1* steps beyond the end of compiled code.

Define the adapted compiler:

fun *ccomp* :: *"nat ⇒ com ⇒ instr list"*

Now adapt the correctness statement by replacing *a* with an appropriate term (without introducing a new constant) and prove it correct.

lemma *ccomp_bigstep*:
"(c,s) ⇒ (c′,t) ⟹ ccomp n c ⊢ (0,s,stk) →∗ (size(ccomp n c) + a,t,stk)"

Homework 6.2 Left and Right Movers

Submission until Wednesday, November 27, 23:59pm.

A semaphore is a counter which can be incremented and decremented by parallel processes, however, decrement has to wait until the counter is greater 0. This ensures that the counter is never negative.

Semaphores can be used to synchronize the access of processes to resources.

We model the possible operations (increment, decrement, unrelated) on semaphores as follows:

datatype *action* = *Up* (*char list*) | *Down* (*char list*) | *Other*

Define the effect of an action on a state. Here, the state holds the values of the semaphores. Assume that other actions do not modify the state.

inductive *exec* :: *"action ⇒ state ⇒ state ⇒ bool"*

Next, we want to develop a scheduler for two processes. The actions of the processes are modeled as lists.

We use a small-step approach, i.e., we define a configuration that contains the remaining actions of the two processes and the current semaphore state:

type_synonym *config* = *"action list × action list × state"*

Then, you have to define a relation *step* such that *step c l c′* means that in the configuration c one action is scheduled, and the resulting configuration is c'. The label l labels an action a and indicates on which process (1 or 2) a acts on:

datatype *label* = *P1 action* | *P2 action*

inductive *step* :: *"config ⇒ label ⇒ config ⇒ bool"*

A well-known result on semaphores is that down-operations are right-movers and up-operations are left movers.

Show that down-operations are right-movers, i.e. a down operation on one process can be exchanged with a subsequent operation on the other process. Intuitively, this moves the down-operation to the right in the interleaving sequence.

With the right automation, this proof can be made very automatic and solved by a one-liner. However, the aim of this homework is to understand better how the automation works. Thus, **do a step-by-step proof**:

- solve every goal with a single *by* with a single method (no , or ;)
- do not use proof methods more powerful than *auto* (i.e., isar proof patterns and single-step methods are allowed but *fastforce*, *blast*, ... are not.)
- use at most a single *simp*, *intro*, or *elim* modifier per method
- do not declare *simp*, *intro*, or *elim* rules to be used automatically

  Hint: You might want to state a similar lemma about *exec* first. Note that there are methods to perform a single *intro* or *elim* step (similar to *rule*).

lemma *step_shift*:
  assumes *"step c1 (P1 (Down x)) c2"*
    and *"step c2 (P2 a) c3"*
 shows *"∃ ch. step c1 (P2 a) ch ∧ step ch (P1 (Down x)) c3"*

Homework 6.3 Locking Order

Submission until Wednesday, November 27, 23:59pm. 6 bonus points, hard - use automation again! (Bonus points count towards your score but not the maximum.)

Another well-known result is that a locking-order implies deadlock freedom: Assume that there is an ordering on locks, such that a process may only acquire locks which are greater than all locks it has already acquired. Moreover, assume that a process eventually releases all acquired locks. Then, there are no deadlocks.

Note that locks can be simulated by semaphores initialized to 1.

We define well-formed action sequences as follows:

well_formed_aux A (Down x # l) = (well_formed_aux (insert x A) l ∧ (∀ y∈A. y < x))

well_formed_aux A (Up x # l) = (well_formed_aux (A − {x}) l ∧ x ∈ A)

well_formed_aux A (Other # l) = well_formed_aux A l

well_formed_aux A [] = (A = {})

well_formed ≡ well_formed_aux {}

Note that the additional parameter A captures the locks that the process has already acquired. For simplicity, we use the lexicographic ordering on semaphore names as lock ordering, from *HOL−Library.List_Lexorder* and *HOL−Library.Char_ord*.

Moreover, we define the initial state, a final state, a deadlocked state, and a step without an explicit label:

init ≡ λ_. 1

fun *final* **where** *"final ([],[],_) ⟷ True" | "final _ ⟷ False"*
definition *"deadlocked c ≡ ¬final c ∧ (∀ c' a. ¬step c a c')"*
abbreviation *"step' c c' ≡ ∃ a. step c a c'"*

Your task is to prove that schedules of well-formed action sequences cannot deadlock:

theorem *deadlock_freedom*:
  assumes *WF1*: *"well_formed l1"*
    and *WF2*: *"well_formed l2"*
    and *STEPS*: *"star step' (l1,l2,init) c'"*
 shows *"¬deadlocked c'"*

Here are some hints on one possible way of proving this: Try to find a suitable invariant on configurations, i.e., a predicate that holds for the initial configuration, and is preserved by a step. Having established such a predicate, you can easily prove that it holds for any reachable configuration:

$\llbracket star\ R\ c0\ c';\ I\ c0;\ \bigwedge c\ c'.\ \llbracket I\ c;\ R\ c\ c' \rrbracket \Longrightarrow I\ c' \rrbracket \Longrightarrow I\ c'$

The invariant should contain enough information about the configuration and the acquired locks to get through the following (informal) argument:

If a state is stuck, there are two cases: 1) Both processes want to acquire locks (wlog a and b) which are not free. Due to locking order, the locks are held by the respective other process. Again, due to locking order, this implies $a > b$ and $a < b$, which is a contradiction.

2) Another possibility for stuck states is that one process is already finished. However, well-formedness ensures that a finished process has released all its locks.