Concrete Semantics with Isabelle/HOL

Exercise Sheet 7

Exercise 7.1  Type checker as recursive functions

Reformulate the inductive predicates $\Gamma \vdash a : \tau$, $\Gamma \vdash b$ and $\Gamma \vdash c$ as three recursive functions

fun atype :: "tyenv ⇒ aexp ⇒ ty option"
fun bok :: "tyenv ⇒ bexp ⇒ bool"
fun cok :: "tyenv ⇒ com ⇒ bool"

and prove

lemma atyping_atype: "(\Gamma \vdash a : \tau) = (atype \Gamma a = Some \tau)"
lemma btyping_bok: "(\Gamma \vdash b) = bok \Gamma b"
lemma ctyping_cok: "(\Gamma \vdash c) = cok \Gamma c"

Exercise 7.2  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 "$x" $(V "$x") (N 5) THEN "$y" ::= N 3 ELSE SKIP
   is compiled by $ccomp$ and identify a possible compiler optimization.
2. Implement an optimized compiler (by modifying $ccomp$) which reduces the number of instructions for programs of the form IF $b$ THEN $c$.
3. Extend the proof of $ccomp\_bigstep$ to your modified compiler.
Homework 7.1  Concurrency and Reynolds’s Criterion (12 points + 3 bonus points)

Submission until Monday 22.06.2015 10:00 AM. Please send your submissions to jasmin.blanchette@mpi-inf.mpg.de and mathias.fleury@mpi-inf.mpg.de with “[CONCRETE7]” in the subject of your email. If your name is Förnamn Efternamn, please call your theory file Efternamn_Förnamn.thy.

In this exercise, we will look at a simple programming language with shared-memory concurrency (i.e., multithreading). The semantics is given by interleaving small steps from concurrent threads—i.e., we have a so-called interleaving semantics.

Our programming language consists of the following commands:

```
datatype pcom =
  Skip — skip
| Asn vname aexp — nonatomic assignment
| Asn_Atom vname aexp — atomic assignment
| Seq pcom pcom — sequential composition
| Par pcom pcom — parallel composition, written || in examples
```

Parallel composition can be seen as starting two threads (commands), each running independently of the other. The two threads’ executions are interleaved (interwoven). For example, the program

```
x := 0;
y := 1;
{
  x := x + y;
  x := x + y
  ||
  x := 5;
  y := 100
}
```

(using some ad hoc syntax) ends up in one of the following states if the assignments are executed atomically (i.e., in one small step):

```
{x := 5, y := 100}
{x := 6, y := 100}
{x := 7, y := 100}
{x := 205, y := 100}
```

The result depends on the precise interleaving of the four instructions in the parallel block. If the assignments are nonatomic, there are even more possibilities, depending on the order of evaluation. For example, in any of the \(x + y\) right-hand sides, \(x\) could be evaluated before \(x := 5\) is executed whereas \(y\) could be executed after both \(x := 5\) and \(y := 100\) have taken place.
One can imagine that atomic assignments are implemented using mutexes or other mechanisms. Here, we simply assume that they exist side by side with nonatomic assignments. We prefer to use nonatomic assignments whenever possible, because these are implemented more efficiently.

**Step A:** To fully model concurrency in the presence of nonatomic assignments, we will need a small-step semantics even for arithmetic expressions. Define one:

```lean
inductive small_step_aexp : "state ⇒ aexp ⇒ aexp ⇒ bool" ("a a a 10")
abbreviation
small_steps_aexp : "state ⇒ aexp ⇒ aexp ⇒ bool" ("a a a 10")
where
"s a a a b a b" = star (small_step_aexp s) a b
```

**Step B:** Define a small-step semantics for our parallel command language. Loading a value from a variable or storing a value into a variable can be considered atomic and hence modeled by one step.

```lean
inductive small_step : "pcom × state ⇒ pcom × state ⇒ bool" ("p p 10")
abbreviation
small_steps : "pcom × state ⇒ pcom × state ⇒ bool" (infix "→∗p" 55)
where
"ps p qt" = star small_step ps qt
```

**Step C:** Some assignments have the same behavior irrespective of whether they are executed atomically or not. An obvious example is `x := 5`, which consists of a single memory store and hence of a single step in the small-step semantics. However, John Reynolds observed that there is a wider class of commands where this is possible. The following material is paraphrased from Gregory R. Andrew’s book *Foundations of Multithreaded, Parallel, and Distributed Programming*.

**At-Most-Once Property** (or **Reynolds’s Criterion**): A critical reference in an expression is a reference to a variable that is changed by another process. An assignment command `x := a` satisfies the at-most-once property if either (1) `a` contains at most one critical reference and `x` is not read by another process, or (2) `a` contains no critical references, in which case `x` may be read by other processes. This is called the at-most-once property because there can be at most one shared variable, and it can be referenced at most once.

If an assignment command meets the requirements of the at-most-once property, then execution of the command will appear to be atomic. A few examples will help clarify this.

Both assignments in the following program satisfy the property:

```plaintext
x := 0;
y := 0;
{| x := x + 1
```

---

3
\[ y := y + 1 \]
\}

There are no critical references in either process. The final values of \( x \) and \( y \) are both 1. Both assignments in this program also satisfy the property:

\[ x := 0; \]
\[ y := 0; \]
\{ \]
\[ x := y + 1 \]
\[ y := y + 1 \]
\} \]

The first process references \( y \) (one critical reference), but \( x \) is not read by the second process. The final value of \( x \) is either 1 or 2, and the final value of \( y \) is 1. The first process will see \( y \) either before or after it is incremented, but making \( x := y + 1 \) atomic would change nothing to that fact.

As a final example, neither assignment below satisfies the at-most-once property:

\[ x := 0; \]
\[ y := 0; \]
\{ \]
\[ x := y + 1 \]
\[ y := x + 1 \]
\} \]

The final values of \( x \) and \( y \) could be 1 and 1, if the processes read \( y \) and \( x \) before either assigns to them.

Your task is to write a function \textit{unatom} that replaces all atomic assignments that satisfy the at-most-once criterion by nonatomic assignments, as an optimization.

\[ \text{fun unatom :: } \textit{pcom} \Rightarrow \textit{pcom} \]

\textbf{Step D:} Try out all four examples introduced so far, using the \textit{value} command. Use \textit{Asn_Atom} to represent \( := \) in the input, and see which assignments remain atomic. Explain your results briefly.

\textbf{Step E:} Formulate some soundness and completeness theorems for the \textit{unatom} function. Provide an informal argument to why the theorems hold. Which direction is the most difficult and why?

\textbf{Step F (3 bonus points):} Prove one of the two directions in Isabelle. Warning: Do not waste too much time on this exercise.