Technische Universität München
Institut für Informatik
Prof. Tobias Nipkow, Ph.D.
Fabian Huch, Kevin Kappelmann

WS 23/24
28.11.2023

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 Compiling the ol' SWITCHeroo

Submission until Monday, December 4, 23:59pm.

Adapt the compiler for the switch construct from the last homework, and prove it correct! The machine instructions are slightly changed: instead of *JMPLESS*/*JMPGE*, we now have *JMPEQ*/*JMPNE*.

Hint: Take some inspiration from the approach for the original IMP!

fun *ccomp* :: *"com ⇒ instr list"*

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

Homework 6 Type Inference

Submission until Monday, December 4, 23:59pm.

Types are wonderful. But it is tedious to write them down all the time. In this exercise, we will hence specify and prove correct a type inference scheme. Unlike type checking schemes, which take a program, a type environment, and a type as input, (c.f. *atyping*, *ctyping*), type inference schemes take a program and a partial type environment as input and return an updated (partial) type environment. The type inference's goal is to extend the partial type environment to suit the given program.

For this purpose, we extend the co-domain of type environments by an unknown, denoting that we do not yet know the type of a variable. If the type inference encounters a program part that determines the type of an unknown, it should update the type environment accordingly. If the type inference encounters a program part that does not match the already determined type, it should fail.

type_synonym *ptyenv = "vname ⇒ ty option"*

For simplicity, we want a one-pass type inference, that is we want to visit each part of the program only once. Unfortunately, this causes a problem: Consider the possible types for the expression $(x+y)+(x+2.3)$. Clearly, we have that both x and y must be reals. However, when type inference is done in a depth-first-search fashion, it will see $x+y$ first and infer x and y to be undetermined. Only later, once it sees the second term *x+2.3*, it has to somehow go back and set y to be *real* as well, even though y does not occur in the second term.

To avoid this problem, we require that all variables that we see in expressions already have a determined type and let type inference fail otherwise. In practice, this means that input variables of the program still need to be explicitly typed, but all other variables will be inferred.

First extend the type checking rules for arithmetic and boolean expressions to partial type environments. Due to the reasons just explained, the check should only succeed if the types of all variables occurring in the expression are determined.

inductive *check_aexp* :: *"ptyenv ⇒ aexp ⇒ ty ⇒ bool"*
inductive *check_bexp* :: *"ptyenv ⇒ bexp ⇒ bool"*

To check whether our specifications are correct, we want to compare them to *atyping* and *btyping*, respectively. For this purpose, we need to explain how a partial type environment might be extended to a type environment.

A type environment is an instance of an partial type environment if the two match on all variables with determined types:

is_inst Γ *p*Γ ≡ ∀ *x T*. *p*Γ *x* = *Some T* ⟶ Γ *x* = *T*

Now show that your specifications are correct:

lemma *atyping_if_check_aexp*:
  assumes *"check_aexp p*Γ *a T"*
   and *"is_inst* Γ *p*Γ*"*
 shows *"atyping* Γ *a T"*

lemma *check_aexp_if_atyping*:
  assumes *"atyping* Γ *a T"*
 shows *"check_aexp* (λ*x*. *Some* (Γ *x*)) *a T"*

lemma *btyping_if_check_bexp*:
  assumes *"check_bexp p*Γ *b"*
   and *"is_inst* Γ *p*Γ*"*
 shows *"btyping* Γ *b"*

lemma *check_bexp_if_btyping*:
  assumes *"btyping* Γ *b"*
 shows *"check_bexp* (λ*x*. *Some* (Γ *x*)) *b"*

Next, write an inductive predicate that extends a partial typing according to a command. For an assignment, the type of the assigned variable is determined to have the type of the right hand side expression. If the assigned variable is already determined to have a different type, no typing for the program should be inferred.

For an if-statement, the inferred types for the then and else branches must be combined. A combination is not possible if a variable is determined to have two different types in the branches. In that case, no typing for the program should be inferred. Use the predicate *combines_to* to express this constraint in your definition:

inductive *infer_com* :: *“ptyenv ⇒ com ⇒ ptyenv ⇒ bool”*

As a test, show that your type inference works for the following program:
test_c ≡ *''x'' ::= Ic 0;; IF Less (V ''x'') (Ic 2) THEN SKIP ELSE ''y'' ::= Rc (10 / 10);; ''y'' ::= Plus (V ''y'') (Rc (31 / 10))*

lemma *type_test_c*:
 “∃ pΓ. infer_com (λ_. None) test_c pΓ ∧ ctyping (λx. case (pΓ x) of Some T ⇒ T | _ ⇒ Ity) test_c”

As sketched below, a safe way to prove such a lemma is to apply the introduction rules manually. Of course, you may also try to automate this proof. Note that you may have to adjust the applied introduction rules for your solution.

```
  unfolding test_c_def
  apply (rule exI)
  apply (rule conjI)
  apply (rule infer_com.intros)
  apply (rule infer_com.intros)
  apply (rule infer_com.intros)
  apply (rule check_aexp.intros)
  apply (rule refl)
```

and so on ...

Prove that your inference is monotone with respect to its input and output environments:

```
lemma is_inst_if_is_inst_if_infer:
    assumes “infer_com pΓ c pΓ'”
      and “is_inst Γ pΓ'”
  shows “is_inst Γ pΓ”
```

Finally, prove soundness of your type inference.

It may be advantageous to prove some auxiliary lemmas about *is_inst* and *combines_to* rather then proving these things in the main proof.

```
lemma ctyping_if_infer_com:
    assumes “infer_com pΓ c pΓ'”
      and “is_inst Γ pΓ'”
  shows “ctyping Γ c”
```