Technische Universität München
Institut für Informatik
Prof. Tobias Nipkow, Ph.D.
Simon Wimmer

WS 2016/17
13. 12. 2016

Semantics of Programming Languages

Exercise Sheet 8

Exercise 8.1 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 8.2 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*.

1. Modify, in the theory *Types*, the inductive definitions of *taval* and *tbval* such that implicit coercions are applied where necessary.
2. Adapt all proofs in the theory *Types* accordingly.

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

Homework 8.1 A Typed Language

Submission until Tuesday, December 20, 2016, 10:00am.

Use the template file `hw08_tmpl.thy`.

We unify boolean expressions *bexp* and arithmetic expressions *aexp* into one expressions language *exp*. We also define a datatype *val* to represent either integers or booleans.

We then give a type system and small semantics. Your task is to show preservation and progress of the type system, i.e. replace all `oops` by valid proofs.

Preparation 1: We define unified values and expressions:

datatype *val* = *Iv int* | *Bv bool*
datatype *exp* =
  N int | *V vname* | *Plus exp exp* | *Bc bool* | *Not exp* | *And exp exp* | *Less exp exp*

Evaluation is now defined as an inductive predicate only working when the types of the values are correct:

inductive *eval* :: "*exp* ⇒ *state* ⇒ *val* ⇒ *bool*" **where**
"*eval* (*N i*) *s* (*Iv i*)" |
"*eval* (*V x*) *s* (*s x*)" |
"*eval* a_1 *s* (*Iv* i_1) ⟹ *eval* a_2 *s* (*Iv* i_2) ⟹ *eval* (*Plus* a_1 a_2) *s* (*Iv* (i_1 + i_2))" |
"*eval* (*Bc v*) *s* (*Bv v*)" |
"*eval b s* (*Bv bv*) ⟹ *eval* (*Not b*) *s* (*Bv* (¬ *bv*))" |
"*eval* b_1 *s* (*Bv* bv_1) ⟹ *eval* b_2 *s* (*Bv* bv_2) ⟹ *eval* (*And* b_1 b_2) *s* (*Bv* (bv_1 ∧ bv_2))" |
"*eval* a_1 *s* (*Iv* i_1) ⟹ *eval* a_2 *s* (*Iv* i_2) ⟹ *eval* (*Less* a_1 a_2) *s* (*Bv* (i_1 < i_2))"

Preparation 2: The small-step semantics are as before, we just replaced *aval* and *bval* with *eval*.

inductive
  small_step :: "(*com* × *state*) ⇒ (*com* × *state*) ⇒ *bool*" (**infix** "→" *55*)
where
Assign: "*eval a s v* ⟹ (*x* ::= *a*, *s*) → (*SKIP*, *s*(*x* := *v*))" |
IfTrue: "*eval b s* (*Bv True*) ⟹ (*IF b THEN* c_1 *ELSE* c_2,*s*) → (c_1,*s*)" |
IfFalse: "*eval b s* (*Bv False*) ⟹ (*IF b THEN* c_1 *ELSE* c_2,*s*) → (c_2,*s*)" |

...

Preparation 3: We introduce the type system.

datatype *ty* = *Ity* | *Bty*

type_synonym *tyenv* = "*vname* ⇒ *ty*"

inductive *etyping* :: "*tyenv* ⇒ *exp* ⇒ *ty* ⇒ *bool*"
 ("(*1_*/ ⊢/ (_ :/ _))")
where
"Γ ⊢ *N i* : *Ity*" |
"Γ ⊢ *V x* : Γ *x*" |
"Γ ⊢ a_1 : *Ity* ⟹ Γ ⊢ a_2 : *Ity* ⟹ Γ ⊢ *Plus* a_1 a_2 : *Ity*" |
"Γ ⊢ *Bc v* : *Bty*" |
"Γ ⊢ *b* : *Bty* ⟹ Γ ⊢ *Not b* : *Bty*" |
"Γ ⊢ b_1 : *Bty* ⟹ Γ ⊢ b_2 : *Bty* ⟹ Γ ⊢ *And* b_1 b_2 : *Bty*" |
"Γ ⊢ a_1 : *Ity* ⟹ Γ ⊢ a_2 : *Ity* ⟹ Γ ⊢ *Less* a_1 a_2 : *Bty*"

inductive *ctyping* :: "*tyenv* ⇒ *com* ⇒ *bool*" (**infix** "⊢" *50*) **where**
Skip_ty: "Γ ⊢ *SKIP*" |

Assign_ty: "$\Gamma \vdash a : \Gamma\ x \Longrightarrow \Gamma \vdash x ::= a$" |
Seq_ty: "$\Gamma \vdash c_1 \Longrightarrow \Gamma \vdash c_2 \Longrightarrow \Gamma \vdash c_1;;c_2$" |
If_ty: "$\Gamma \vdash b : \mathit{Bty} \Longrightarrow \Gamma \vdash c_1 \Longrightarrow \Gamma \vdash c_2 \Longrightarrow \Gamma \vdash \mathit{IF}\ b\ \mathit{THEN}\ c_1\ \mathit{ELSE}\ c_2$" |
While_ty: "$\Gamma \vdash b : \mathit{Bty} \Longrightarrow \Gamma \vdash c \Longrightarrow \Gamma \vdash \mathit{WHILE}\ b\ \mathit{DO}\ c$"

We define a state typing *styping* to describe the type context of a state.

fun *type* :: "$\mathit{val} \Rightarrow \mathit{ty}$" **where**
"$\mathit{type}\ (\mathit{Iv}\ i) = \mathit{Ity}$" |
"$\mathit{type}\ (\mathit{Bv}\ r) = \mathit{Bty}$"
definition *styping* :: "$\mathit{tyenv} \Rightarrow \mathit{state} \Rightarrow \mathit{bool}$" (**infix** "$\vdash$" 50) **where**
"$\Gamma \vdash s \longleftrightarrow (\forall x.\ \mathit{type}\ (s\ x) = \Gamma\ x)$"

Task 1: Show preservation and progress on expressions:

lemma *epreservation*: "$\Gamma \vdash a : \tau \Longrightarrow \mathit{eval}\ a\ s\ v \Longrightarrow \Gamma \vdash s \Longrightarrow \mathit{type}\ v = \tau$"
lemma *eprogress*: "$\Gamma \vdash a : \tau \Longrightarrow \Gamma \vdash s \Longrightarrow \exists v.\ \mathit{eval}\ a\ s\ v$"

Task 2: Show progress and preservation on commands:

theorem *progress*: "$\Gamma \vdash c \Longrightarrow \Gamma \vdash s \Longrightarrow c \neq \mathit{SKIP} \Longrightarrow \exists cs'.\ (c,s) \rightarrow cs'$"
theorem *styping_preservation*: "$(c,s) \rightarrow (c',s') \Longrightarrow \Gamma \vdash c \Longrightarrow \Gamma \vdash s \Longrightarrow \Gamma \vdash s'$"
theorem *ctyping_preservation*: "$(c,s) \rightarrow (c',s') \Longrightarrow \Gamma \vdash c \Longrightarrow \Gamma \vdash c'$"
theorem *type_sound*:
"$(c,s) \rightarrow\!* (c',s') \Longrightarrow \Gamma \vdash c \Longrightarrow \Gamma \vdash s \Longrightarrow c' \neq \mathit{SKIP} \Longrightarrow \exists cs''.\ (c',s') \rightarrow cs''$"

Hint: For most of the proof work, you should be able to closely follow the proofs in the original IMP theory.

Homework 8.2 A Type System for Physical Units

Submission until Tuesday, December 20, 2016, 10:00am.

Start with a fresh copy of *Types.thy*. We will define a language that only computes on real values but attaches a physical unit to every constant. The binary operators are addition and multiplication (*op* $*$ in Isabelle/HOL). The semantics shall get stuck if trying to add or compare values with different physical units.

Define a type system that uses physical units as types. Well-typed programs must not add or compare values with different physical units. Adapt the theory up to the *type_sound*-theorem, i.e., show that in a well-typed program, every reachable non-skip state can make another step. Some steps of this development are detailed below.

Note: Please turn in two separate files for the two homework exercises.

A unit is either an elementary unit (Newton or Meters), or a product of units.

datatype *unit* = *N* | *M* | *Prod unit unit*

We only consider real values but attach units to values:

type_synonym *val* = *"real* $\times$ *unit"*

datatype *aexp* = *Pc val* | *V vname* | *Plus aexp aexp* | *Mult aexp aexp*

You will need to define an equality predicate *unit_eq* :: *unit* $\Rightarrow$ *unit* $\Rightarrow$ *bool* on units. Note that e.g. *Prod N M* should be the same as *Prod M N*.

The types are simply all possible units:

type_synonym *ty* = *unit*

It is easy to read types from values in our setting: they are already attached to them. Thus a well-typed state is expressed as follows:

definition *styping* :: *"tyenv* $\Rightarrow$ *state* $\Rightarrow$ *bool"* (**infix** "$\vdash$" *50*)
where "$\Gamma \vdash s \longleftrightarrow (\forall x.\ unit\_eq\ (snd\ (s\ x))\ (\Gamma\ x))$"