Technische Universität München
Fakultät für Informatik
Dr. Peter Lammich
Simon Wimmer

WS 2018/19
15. 01. 2019

Semantics of Programming Languages

Exercise Sheet 12

Exercise 12.1 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 12.2 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 (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 12.1 Redundant Assignments

do this in preparation of exam
so print slides first

Submission until Tuesday, January 22, 2019, 10:00am.

In this exercise we will consider a variant of IMP extended with arrays. Your task will be to adopt the type system for this variant, and to adapt the proofs for progress and preservation of the type system. Arrays can either contain only integer or only real values. The template will guide you through the necessary steps.

Arithmetic Expressions
The type of elementary values as before:

datatype *pval* = *Iv int* | *Rv real*

We use an additional type of for array values. They are either arrays of reals or of integers:

datatype *val* = *Ia "int ⇒ int"* | *Ra "int ⇒ real"*

type_synonym *vname* = *string*

type_synonym *state* = *"vname ⇒ val"*

Arithmetic expressions with arrays, analogously to what we have seen before:

datatype *aexp* = *Ic int* | *Rc real* | *Vidx vname aexp* | *Plus aexp aexp*

Add two typing rules for *Vidx*, one each for the case of an integer and a real array:

inductive *taval* :: *"aexp ⇒ state ⇒ pval ⇒ bool"* **where**
taval_Ic: *"taval (Ic i) s (Iv i)"* |
taval_Rc: *"taval (Rc r) s (Rv r)"* |
taval_PlusInt: *"taval a1 s (Iv i1) ⟹ taval a2 s (Iv i2)*
⟹ taval (Plus a1 a2) s (Iv(i1+i2))" |
taval_PlusReal: *"taval a1 s (Rv r1) ⟹ taval a2 s (Rv r2)*
⟹ taval (Plus a1 a2) s (Rv(r1+r2))" |
— New:

Note that we explicitly do not declare these theorems as [*elim*!]:

inductive_cases *taval_elims*:
"taval (Ic i) s v" *"taval (Rc i) s v"*
"taval (V x) s v"
"taval (Plus a1 a2) s v"
— New
"taval (Vidx x i) s v"

Boolean Expressions Nothing changes for Boolean expressions:

datatype *bexp* = *Bc bool* | *Not bexp* | *And bexp bexp* | *Less aexp aexp*

inductive *tbval* :: *"bexp ⇒ state ⇒ bool ⇒ bool"* **where**
"tbval (Bc v) s v" |
"tbval b s bv ⟹ tbval (Not b) s (¬ bv)" |
"tbval b1 s bv1 ⟹ tbval b2 s bv2 ⟹ tbval (And b1 b2) s (bv1 & bv2)" |
"taval a1 s (Iv i1) ⟹ taval a2 s (Iv i2) ⟹ tbval (Less a1 a2) s (i1 < i2)" |
"taval a1 s (Rv r1) ⟹ taval a2 s (Rv r2) ⟹ tbval (Less a1 a2) s (r1 < r2)"

Syntax of Commands% **datatype**
com = *SKIP*
| *Seq com com* ("_;; _" [60, 61] 60)

| *If* *bexp com com* (*"IF _ THEN _ ELSE _"* [*0*, *0*, *61*] *61*)
| *While* *bexp com* (*"WHILE _ DO _"* [*0*, *61*] *61*)
— New:
| *AssignIdx vname aexp aexp* (*"_[_] ::= _"* [*1000*, *0*, *61*] *61*)
| *ArrayCpy vname vname* (*"_[] ::= _"* [*1000*, *1000*] *61*)
| *ArrayClear vname* (*"CLEAR _[]"* [*1000*] *61*)

Small-Step Semantics of Commands Add rules for array assignment, clear, and copy to the small-step semantics. It should only be possible to assign integer values to integer arrays, and real values to real arrays. Clear should reset the array to a zero-only array of the same type.

inductive
small_step :: "(*com* × *state*) ⇒ (*com* × *state*) ⇒ *bool*" (**infix** "→" *55*)
where
Seq1: "(*SKIP*;;*c*,*s*) → (*c*,*s*)" |
Seq2: "(*c1*,*s*) → (*c1'*,*s'*) ⟹ (*c1*;;*c2*,*s*) → (*c1'*;;*c2*,*s'*)" |
IfTrue: "*tbval b s True* ⟹ (*IF b THEN c1 ELSE c2*,*s*) → (*c1*,*s*)" |
IfFalse: "*tbval b s False* ⟹ (*IF b THEN c1 ELSE c2*,*s*) → (*c2*,*s*)" |
While: "(*WHILE b DO c*,*s*) → (*IF b THEN c*;; *WHILE b DO c ELSE SKIP*,*s*)" |
— New:
lemmas *small_step_induct* = *small_step.induct*[*split_format*(*complete*)]

The Type System We still use the same types:

datatype *ty* = *Ity* | *Rty*

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

Add a typing rule for *Vidx*:

inductive *atyping* :: "*tyenv* ⇒ *aexp* ⇒ *ty* ⇒ *bool*"
("(*1_*/ ⊢/ (_ :/ _))" [*50*,*0*,*50*] *50*)
where
Ic_ty: "Γ ⊢ *Ic i* : *Ity*" |
Rc_ty: "Γ ⊢ *Rc r* : *Rty*" |
Plus_ty: "Γ ⊢ *a1* : τ ⟹ Γ ⊢ *a2* : τ ⟹ Γ ⊢ *Plus a1 a2* : τ" |
— New:
declare *atyping.intros* [*intro*!]
inductive_cases [*elim*!]:
"Γ ⊢ *V x* : τ" "Γ ⊢ *Ic i* : τ" "Γ ⊢ *Rc r* : τ" "Γ ⊢ *Plus a1 a2* : τ" "Γ ⊢ *Vidx x i* : τ"

Nothing changes for Boolean expressions:

inductive *btyping* :: "*tyenv* ⇒ *bexp* ⇒ *bool*" (**infix** "⊢" *50*)
where
B_ty: "Γ ⊢ *Bc v*" |
Not_ty: "Γ ⊢ *b* ⟹ Γ ⊢ *Not b*" |
And_ty: "Γ ⊢ *b1* ⟹ Γ ⊢ *b2* ⟹ Γ ⊢ *And b1 b2*" |

Less_ty: “$\Gamma \vdash a1 : \tau \Longrightarrow \Gamma \vdash a2 : \tau \Longrightarrow \Gamma \vdash$ *Less a1 a2*”

declare *btyping.intros* [*intro*!]
inductive_cases [*elim*!]: “$\Gamma \vdash$ *Not b*” “$\Gamma \vdash$ *And b1 b2*” “$\Gamma \vdash$ *Less a1 a2*”

Add typing rules for array assignment, clear, and copy. *Hint:* One rule for each construct suffices.

inductive *ctyping* :: “*tyenv* ⇒ *com* ⇒ *bool*” (**infix** “⊢” *50*) **where**
Skip_ty: “$\Gamma \vdash$ *SKIP*” |
Seq_ty: “$\Gamma \vdash c1 \Longrightarrow \Gamma \vdash c2 \Longrightarrow \Gamma \vdash c1;;c2$” |
If_ty: “$\Gamma \vdash b \Longrightarrow \Gamma \vdash c1 \Longrightarrow \Gamma \vdash c2 \Longrightarrow \Gamma \vdash$ *IF b THEN c1 ELSE c2*” |
While_ty: “$\Gamma \vdash b \Longrightarrow \Gamma \vdash c \Longrightarrow \Gamma \vdash$ *WHILE b DO c*” |
AssignIdx_ty: “$\Gamma \vdash i : Ity \Longrightarrow \Gamma \vdash a : \tau \Longrightarrow \Gamma(x) = \tau \Longrightarrow \Gamma \vdash x[i] ::= a$” |
Clear_ty: “$\Gamma \vdash$ *CLEAR x*[]” |
Copy_ty: “$\Gamma\ x = \tau \Longrightarrow \Gamma\ y = \tau \Longrightarrow \Gamma \vdash x[] ::= y$”

declare *ctyping.intros* [*intro*!]
inductive_cases [*elim*!]:
“$\Gamma \vdash c1;;c2$”
“$\Gamma \vdash$ *IF b THEN c1 ELSE c2*”
“$\Gamma \vdash$ *WHILE b DO c*”
— New
“$\Gamma \vdash x[i] ::= a$”
“$\Gamma \vdash$ *CLEAR x*[]”
“$\Gamma \vdash x[] ::= y$”

Well-typed Programs Do Not Get Stuck The type of elementary values:

fun *ptype* :: “*pval* ⇒ *ty*” **where**
“*ptype* (*Iv i*) = *Ity*” |
“*ptype* (*Rv r*) = *Rty*”

The type of array values:

fun *type* :: “*val* ⇒ *ty*” **where**
“*type* (*Ia i*) = *Ity*” |
“*type* (*Ra r*) = *Rty*”

lemma *ptype_eq_Ity*[*simp*]: “*ptype v* = *Ity* ⟷ (∃ *i*. *v* = *Iv i*)”
by (*cases v*) *simp_all*

lemma *ptype_eq_Rty*[*simp*]: “*ptype v* = *Rty* ⟷ (∃ *r*. *v* = *Rv r*)”
by (*cases v*) *simp_all*

lemma *type_eq_Ity*[*simp*]: “*type v* = *Ity* ⟷ (∃ *i*. *v* = *Ia i*)”
by (*cases v*) *simp_all*

lemma *type_eq_Rty*[*simp*]: “*type v* = *Rty* ⟷ (∃ *r*. *v* = *Ra r*)”
by (*cases v*) *simp_all*

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

Adapt the proofs for progress and preservation:

theorem *apreservation*:
"$\Gamma \vdash a : \tau \Longrightarrow taval\ a\ s\ v \Longrightarrow \Gamma \vdash s \Longrightarrow ptype\ v = \tau$"

theorem *aprogress*: "$\Gamma \vdash a : \tau \Longrightarrow \Gamma \vdash s \Longrightarrow \exists\, v.\ taval\ a\ s\ v$"

theorem *bprogress*: "$\Gamma \vdash b \Longrightarrow \Gamma \vdash s \Longrightarrow \exists\, v.\ tbval\ b\ s\ v$"

theorem *progress*:
"$\Gamma \vdash c \Longrightarrow \Gamma \vdash s \Longrightarrow c \neq SKIP \Longrightarrow \exists\, cs'.\ (c,s) \rightarrow cs'$"

theorem *styping_preservation*:
"$(c,s) \rightarrow (c',s') \Longrightarrow \Gamma \vdash c \Longrightarrow \Gamma \vdash s \Longrightarrow \Gamma \vdash s'$"

theorem *ctyping_preservation*:
"$(c,s) \rightarrow (c',s') \Longrightarrow \Gamma \vdash c \Longrightarrow \Gamma \vdash c'$"
abbreviation *small_steps* :: *"com* $*$ *state* $\Rightarrow$ *com* $*$ *state* $\Rightarrow$ *bool"* (**infix** *"*$\rightarrow*$*" 55*)
where *"x* $\rightarrow*$ *y == star small_step x y"*

corollary *type_sound*:
"$(c,s) \rightarrow* (c',s') \Longrightarrow \Gamma \vdash c \Longrightarrow \Gamma \vdash s \Longrightarrow c' \neq SKIP$
$\Longrightarrow \exists\, cs''.\ (c',s') \rightarrow cs''$"
apply(*induction rule*:*star_induct*)
apply (*metis progress*)
by (*metis styping_preservation ctyping_preservation*)

Hint: Note that the original proofs are highly automated. Do not expect your proofs to be quite as automated! Use Isar. Explicit rule inversion can be helpful. Recall that this can be achieved with a proof snippet of the following form:
from ‹*taval a s v*› *show ?case proof cases*

Homework 12.2 Absolute Adressing

Submission until Tuesday, January 22, 2019, 10:00am. This homework is worth 5 bonus points.

The current instruction set uses *relative addressing*, i.e., the jump-instructions contain an offset that is added to the program counter. An alternative is *absolute addressing*, where jump-instructions contain the absolute address of the jump target.

Write a semantics that interprets the three types of jump instructions with absolute addresses. Write a function that converts a program from relative to absolute addressing. Show that the semantics match wrt. your conversion.

definition *cnv_to_abs* :: *"instr list ⇒ instr list"*

abbreviation
exec_abs :: *"instr list ⇒ config ⇒ config ⇒ bool"* ("(_/ $\vdash_a$ (_ →*/ _))" *50*)

theorem *cnv_to_abs_correct*: *"cnv_to_abs P* $\vdash_a$ *c* →* *c′* ⟷ *P* ⊢ *c* →* *c′"*

Also define an inverse function

definition *cnv_to_rel* :: *"instr list ⇒ instr list"*

and prove:

theorem *"P* $\vdash_a$ *c* →* *c′* ⟷ *cnv_to_rel P* ⊢ *c* →* *c′"*

Hint: The inverse direction should be easy once you have the first part.