Technische Universität München
Fakultät für Informatik
Prof. Tobias Nipkow, Ph.D.
Fabian Huch

WS 2020/21
7.12.2020

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*.

1. Modify, in the theory *HOL−IMP.Types* (copy it first), the inductive definitions of *taval* and *tbval* 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 Absolute Adressing

Submission until Sunday, Dec 13, 23:59.

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 3 types of jump instructions with absolute addresses.

fun *iexec_abs* :: *"instr ⇒ config ⇒ config"*

definition *exec1_abs* :: *"instr list ⇒ config ⇒ config ⇒ bool"*
(*"(_/ ⊢ₐ (_ →/ _))" [59,0,59] 60*)

lemma *exec1_absI* [*intro*]:
"⟦c' = iexec_abs (P!!i) (i,s,stk); 0 ≤ i; i < size P⟧ ⟹ P ⊢ₐ (i,s,stk) → c'"

abbreviation *exec_abs* :: *"instr list ⇒ config ⇒ config ⇒ bool"*
(*"(_/ ⊢ₐ (_ →*/ _))" 50*)

Now write a function that converts a program from absolute to relative addressing:

cnv_to_rel :: *instr list ⇒ instr list*

Finally show that the semantics match wrt. your conversion. Hints:

- First write a function that converts each instruction, depending on its address. Then use the function *index_map*, that is defined below, to convert a program.
- Prove the theorem for a single step first.

fun *index_map* :: *"(int ⇒ 'a ⇒'a) ⇒ int ⇒ 'a list ⇒ 'a list"* **where**
"index_map f i [] = []"
| *"index_map f i (x#xs) = f i x # index_map f (i+1) xs"*

Start with proving the following basic facts about *index_map*, which may be helpful for your main proof!

lemma *index_map_len*[*simp*]: "*size* (*index_map f i l*) = *size l*"
lemma *index_map_idx*[*simp*]: "$\llbracket$ $0 \le i$; $i <$ *size l* $\rrbracket \Longrightarrow$ *index_map f k l* !! $i = f$ $(i + k)$ (l !! i)"

theorem *cnv_correct*: "$P \vdash_a c \rightarrow* c' \longleftrightarrow$ *cnv_to_rel* $P \vdash c \rightarrow* c'$"

Homework 6.2 Algebra of Commands

Submission until Sunday, Dec 13, 23:59.

We define an extension of the language with parallel composition (‖).

datatype
com = *SKIP*
| *Assign vname aexp* ("_ ::= _" [1000, 61] 61)
| *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)
| *Par com com* (**infix** "‖" 59)

inductive
small_step :: "*com* ∗ *state* ⇒ *com* ∗ *state* ⇒ *bool*" (**infix** "→" 55) **where**
— sequential part as before
ParL: "$(c1,s) \rightarrow (c1',s') \Longrightarrow (c1 \parallel c2,s) \rightarrow (c1' \parallel c2,s')$" |
ParLSkip: "$(SKIP \parallel c,s) \rightarrow (c,s)$" |
ParR: "$(c2,s) \rightarrow (c2',s') \Longrightarrow (c1 \parallel c2,s) \rightarrow (c1 \parallel c2',s')$" |
ParRSkip: "$(c \parallel SKIP,s) \rightarrow (c,s)$"

Your task will be to prove various algebraic laws for the small-step equivalence. For that, we define the *nsteps* relation. Custom induction rules for small step and nsteps are provided below.

lemmas *small_step_induct* = *small_step.induct*[*split_format*(*complete*)]

inductive
nsteps :: "*com* ∗ *state* ⇒ *nat* ⇒ *com* ∗ *state* ⇒ *bool*" ("_ →^_ _" [60,1000,60]999) **where**
zero_steps[*simp*,*intro*]: "$cs \rightarrow\hat{}0\ cs$" |
one_step[*intro*]: "$cs \rightarrow cs' \Longrightarrow cs' \rightarrow\hat{}n\ cs'' \Longrightarrow cs \rightarrow\hat{}(Suc\ n)\ cs''$"

lemmas *nsteps_induct* = *nsteps.induct*[*split_format*(*complete*)]

We consider the small-step pre-order relation ⪯:

definition *small_step_pre* :: "*com* ⇒ *com* ⇒ *bool*" (**infix** "⪯" 50) **where**
"$c \preceq c' \equiv (\forall s\ t\ n.\ (c,s) \rightarrow\hat{}n\ (SKIP,\ t) \longrightarrow (\exists\ n' \ge n.\ (c',\ s) \rightarrow\hat{}n'\ (SKIP,\ t)))$"

Based on the pre-order on programs, define an equivalence relation ≈ on programs.

definition *small_step_equiv* :: "*com* ⇒ *com* ⇒ *bool*" (**infix** "≈" 50)

Now prove commutativity and associativity of $\|$. You are free to do either automatic or Isar proofs. In the former case, make sure to set up some proof automation first.

theorem *Par_commute*: "$c \parallel d \approx d \parallel c$"
theorem *Par_assoc*: "$(c \parallel d) \parallel e \approx c \parallel (d \parallel e)$"

Homework 6.3 Type Inference (Bonus Exercise)

Submission until Sunday, Dec 13, 23:59. This is a bonus exercise worth 4 points.

Specifying the types of variables is annoying, in particular, as they are mostly clear from the program anyway.

In this exercise, you shall implement and prove correct a type inference scheme. The type inference goes through the program similar to *atyping*, *btyping*, *ctyping*. But instead of only checking whether the specified types match the program, it computes matching types.

For this purpose, we extend types by an unknown value, which means that we do not yet know the type of that variable. If the type inference encounters a program part that determines the type of a variable typed with unknown, it will update the type environment accordingly. If type inference encounters a program part that does not match the already determined typing, it fails.

type_synonym *ety* = *"ty option"*
type_synonym *etyenv* = *"vname ⇒ ety"*

For efficiency (and simplicity) we want a one-pass type inference, i.e., we want to visit each part of the program only once. However, this causes a problem: Consider the possible types for 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 top-down fashion, it will see $x + y$ first, and infer x and y to be undetermined. Only later, if it sees the second term, it has to somehow go back and set y to be *real* too, although y does not occur in the second term.

To avoid this effect, we will assume that variables that we see in expressions have already a determined type, and let type inference fail otherwise. This means, that input variables of the program still need to be explicitly typed.

Define the following predicates, which determine the type of an arithmetic/Boolean expression. A type should only be returned if the types of all variables occurring in the expression are determined.

inductive *infer_aty* :: *"etyenv ⇒ aexp ⇒ ty ⇒ bool"*
inductive *infer_bty* :: *"etyenv ⇒ bexp ⇒ bool"*

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

definition *is_inst* :: *"tyenv ⇒ etyenv ⇒ bool"*

where "$is\_inst\ \Gamma\ e\Gamma \equiv \forall x\ \tau.\ e\Gamma\ x = Some\ \tau \longrightarrow \Gamma\ x = \tau$"

Show that type inference infers a valid typing, i.e., all instances of the inferred typing are valid:

theorem *ainfer*: **assumes** "$infer\_aty\ e\Gamma\ a\ \tau$" **and** "$is\_inst\ \Gamma\ e\Gamma$" **shows** "$atyping\ \Gamma\ a\ \tau$"
theorem *binfer*: **assumes** "$infer\_bty\ e\Gamma\ b$" **and** "$is\_inst\ \Gamma\ e\Gamma$" **shows** "$btyping\ \Gamma\ b$"

Next, write a predicate that extends a typing according to a command. On 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.

On an if-statement, the inferred types for the then and else part must be combined. If combination is not possible, because a variable is determined to have two different types in the then and else part, no typing for the program should be inferred. This is expressed by the following predicate:

definition *combine* :: "$etyenv \Rightarrow etyenv \Rightarrow etyenv \Rightarrow bool$" **where**
"$combine\ e\Gamma_1\ e\Gamma_2\ e\Gamma \equiv e\Gamma = e\Gamma_1 ++ e\Gamma_2\ \wedge$
$(\forall x\ \tau_1\ \tau_2.\ e\Gamma_1\ x = Some\ \tau_1 \wedge e\Gamma_2\ x = Some\ \tau_2 \longrightarrow \tau_1 = \tau_2)$"

inductive *infer_cty* :: "$etyenv \Rightarrow com \Rightarrow etyenv \Rightarrow bool$"

As a test, show that your type inference works for the following program

abbreviation "$test\_c \equiv$
$''x''$::=*Ic 0*;;
(*IF Less* (*V* $''x''$) (*Ic 2*) *THEN SKIP ELSE* $''y''$ ::= *Rc 1.0*);;
$''y''$ ::= *Plus* (*V* $''y''$) (*Rc 3.1*)"

lemma "$\exists\, e\Gamma'.\ infer\_cty\ (\lambda\_.\ None)\ test\_c\ e\Gamma'$"

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 probably have to adjust the applied introduction rules to your solution!

apply (*rule exI*)

apply (*rule infer_cty.intros*)
 apply *simp*
 apply (*rule infer_cty.intros*)
  apply (*rule infer_cty.intros*)
apply *simp*
  apply (*rule infer_aty.intros*) — and so on ...

Finally, prove the following theorem:

theorem *infer_typing*: **assumes** "$infer\_cty\ e\Gamma\ c\ e\Gamma'$" **and** "$is\_inst\ \Gamma\ e\Gamma'$" **shows** "$ctyping\ \Gamma\ c$"

Hint: You will need some auxiliary lemmas. The main idea is that *infer_cty* only determines more types, but does not change already determined ones, and that if type inference for *aexp* and *bexp* works on a type environment, it also works on a more determined type environment. You may use (($\subseteq_m$), look it up using *find_theorems*!) to express that a type environment is less determined than another one.

Moreover, it may be advantageous to prove some auxiliary lemmas about ($\subseteq_m$), *is_inst*, *combine* and the relation of these concepts, rather then proving these things in the main proof.