Technische Universität München
Institut für Informatik
Prof. Tobias Nipkow, Ph.D., Dr. Peter Lammich
Johannes Hölzl

WS 2014/15
18. 11. 2014

Semantics of Programming Languages

Exercise Sheet 6

Exercise 6.1 A different instruction set architecture

We consider a different instruction set which evaluates boolean expressions on the stack, similar to arithmetic expressions:

- The boolean value *False* is represented by the number *0*, the boolean value *True* is represented by any number not equal to *0*.
- For every boolean operation exists a corresponding instruction which, similar to arithmetic instructions, operates on values on top of the stack.
- The new instruction set introduces a conditional jump which pops the top-most element from the stack and jumps over a given amount of instructions, if the popped value corresponds to *False*, and otherwise goes to the next instruction.

Modify the theory *Compiler* by defining a suitable set of instructions, by adapting the execution model and the compiler and by updating the correctness proof.

Exercise 6.2 Deskip

Define a recursive function

fun *deskip* :: *"com ⇒ com"*

that eliminates as many *SKIP*s as possible from a command. For example:

deskip (*SKIP*;; *WHILE b DO* (*x* ::= *a*;; *SKIP*)) = *WHILE b DO x* ::= *a*

Prove its correctness by induction on *c*:

lemma *"deskip c ∼ c"*

Remember lemma *sim_while_cong* for the *WHILE* case.

Homework 6.1 While Free Programs

Submission until Tuesday, November 25, 10:00am.

a) Show that while-free programs always terminate, i.e., show that for any while-free command and any state, the big-step semantics yields a result state.

b) Show that non-terminating programs contain a while loop, i.e., show that all commands, for which there is a state such that the big-step semantics yields no result, contain a while loop.

Homework 6.2 Absolute Adressing

Submission until Tuesday, November 25, 10:00am.

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"* ("(_/ $\vdash_a$ (_ →/ _))" [59,0,59] 60)
lemma *exec1_absI* [*intro*]:
"⟦c' = *iexec_abs* (P!!i) (i,s,stk); 0 ≤ i; i < *size* P⟧ ⟹ P $\vdash_a$ (i,s,stk) → c'"

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

Write a function that converts a program from absolute to relative addressing:

cnv_to_rel :: *instr list ⇒ instr list*

Show that the semantics match wrt. your conversion.

$P \vdash_a c \rightarrow\!* c' \longleftrightarrow$ *cnv_to_rel* $P \vdash c \rightarrow\!* c'$

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*"
— Map with index
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*]: "$size\ (index\_map\ f\ i\ l) = size\ l$"
— *index_map* preserves size of list
lemma *index_map_idx*[*simp*]: "$\llbracket\ 0 \leq i;\ i < size\ l\ \rrbracket$
$\Longrightarrow index\_map\ f\ k\ l\ !!\ i = f\ (i+k)\ (l!!i)$"
— *index_map* commutes with list indexing

Homework 6.3 Control Flow Graphs

Submission until Tuesday, November 25, 2014, 10:00am. This homework is worth **5 bonus points**.

From Homework 4.1:

type_synonym $('q,'l)$ *lts* = "$'q \Rightarrow 'l \Rightarrow 'q \Rightarrow bool$"
inductive *word* :: "$('q,'l)$ *lts* $\Rightarrow 'q \Rightarrow 'l$ *list* $\Rightarrow 'q \Rightarrow bool$" **for** δ
where
empty: "*word* δ *q* [] *q*"
| *prepend*: "$[\![\delta\ q\ l\ qh;\ word\ \delta\ qh\ ls\ q']\!] \Longrightarrow word\ \delta\ q\ (l\#ls)\ q'$"

A control flow graph is a labeled transition system (cf. Homework 4.1), where the edges are labeled with actions:

datatype *action* =
EAssign vname aexp — Assign variable
| *ETest bexp* — Only executable if expression is true
| *ESkip*
type_synonym $'q$ *cfg* = "$('q,action)$ *lts*"

Intuitively, the control flow graph is executed by following a path and applying the effects of the actions to the state.

Define the effect of an action to a state. Your function shall return *None* if the action is not executable, i.e., a test of an expression that evaluates to *False*:

fun *eff* :: "*action* $\Rightarrow$ *state* $\rightharpoonup$ *state*" **where**

Lift your definition to paths. Again, only paths where all tests succeed shall yield a result $\neq$ *None*.

fun *eff_list* :: "*action list* $\Rightarrow$ *state* $\rightharpoonup$ *state*" **where**

The control flow graph of a WHILE-Program can be defined over nodes that are commands. Complete the following definition. (Hint: Have a look at the small-step semantics first)

inductive *cfg* :: "*com cfg*" **where**
cfg_assign: "*cfg* (*n* ::= *e*) (*EAssign n e*) (*SKIP*)"
| *cfg_Seq2*: "$[\![$ *cfg c1 e c1'* $]\!] \Longrightarrow$ *cfg* (*c1*;;*c2*) *e* (*c1'*;;*c2*)"

Prove that the effects of paths in the CFG match the small-step semantics:

lemma *eq_path*: "$(c,s) \rightarrow\!* (c',s') \longleftrightarrow (\exists \pi.\ word\ cfg\ c\ \pi\ c' \wedge eff\_list\ \pi\ s = Some\ s')$"

Hint. Prove the lemma for a single step first.