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

WS 23/24
7.11.2023

Semantics of Programming Languages

Exercise Sheet 3

Exercise 3.1 Reflexive Transitive Closure

A binary relation is expressed by a predicate of type $R :: {'s} \Rightarrow {'s} \Rightarrow bool$.

Intuitively, $R\ s\ t$ represents a single step from state s to state t.

The reflexive, transitive closure R^* of R is the relation that contains a step $R^*\ s\ t$, iff R can step from s to t in any number of steps (including zero steps).

Formalize the reflexive transitive closure as an inductive predicate:

inductive *star* :: *"('a ⇒ 'a ⇒ bool) ⇒ 'a ⇒ 'a ⇒ bool"* **for** *r*

When doing so, you have the choice to append or prepend a step. In any case, the following two lemmas should hold for your definition:

lemma *star_prepend*: *"⟦r x y; star r y z⟧ ⟹ star r x z"*

lemma *star_append*: *"⟦ star r x y; r y z ⟧ ⟹ star r x z"*

Now, formalize the star predicate again, this time the other way round (append if you prepended the step before or vice versa):

inductive *star'* :: *"('a ⇒ 'a ⇒ bool) ⇒ 'a ⇒ 'a ⇒ bool"* **for** *r*

Prove the equivalence of your two formalizations:

lemma *"star r x y = star' r x y"*

Exercise 3.2 Avoiding Stack Underflow

A *stack underflow* occurs when executing an instruction on a stack containing too few values—e.g., executing an *ADD* instruction on an stack of size less than two. A well-formed sequence of instructions (e.g., one generated by *comp*) should never cause a stack underflow.

In this exercise, you will define a semantics for the stack-machine that throws an exception if the program underflows the stack.

Modify the *exec1* and *exec* - functions, such that they return an option value, *None* indicating a stack-underflow.

fun *exec1* :: "*instr* ⇒ *state* ⇒ *stack* ⇒ *stack option*"
fun *exec* :: "*instr list* ⇒ *state* ⇒ *stack* ⇒ *stack option*"

Now adjust the proof of theorem *exec_comp* to show that programs output by the compiler never underflow the stack:

theorem *exec_comp*: "*exec* (*comp a*) *s stk* = *Some* (*aval a s* # *stk*)"

Exercise 3.3 A Structured Proof on Relations

We consider two binary relations T and A and assume that T is total, A is antisymmetric and T is finer than A, i.e., $T\ x\ y$ implies $A\ x\ y$ for all x, y. Show with a structured, Isar-style proof that then A finer than T (without proof methods more powerful than simp!):

lemma
  assumes *total*: "$\forall x\ y.\ T\ x\ y \lor T\ y\ x$"
    and *anti*: "$\forall x\ y.\ A\ x\ y \land A\ y\ x \longrightarrow x = y$"
    and *subset*: "$\forall x\ y.\ T\ x\ y \longrightarrow A\ x\ y$"
 shows "$A\ x\ y \longrightarrow T\ x\ y$"

Homework 3.1 Compiling bexps

Submission until Monday, November 13, 23:59pm.

Consider a stack machine with additional binary operations as instructions:

datatype *instr* = *LOADI int* | *LOAD* (*char list*) | *bADD* | *bSUB* | *bMAX* | *bMIN*

The implementation of their execution is straightforward by extending the known *exec1* function, e.g.:

exec1 bMAX ux (*j* # *i* # *stk*) = *max i j* # *stk*

This machine admits compilation of *aexp*s exactly like in the lecture (the compilation function is called *acomp*). Your job now is it to define compilation of *bexp*s. Since the machine computes on integers, the result should be *1* if the expression evaluates to *True* and *0* otherwise.

fun *bcomp* :: "*bexp* ⇒ *instr list*"

Show that you construction is correct!

theorem *exec_bcomp*: "*exec* (*bcomp b*) *s stk* = (*if bval b s then 1 else 0*) # *stk*"

Homework 3.2 Regular Expressions

Submission until Monday, November 13, 23:59pm.

In this exercise, we take a nostalgic trip back to your *Introduction to the Theory of Computation* class (also known as *THEO* at TUM).

A *word* is a list of characters over some alphabet, which we keep polymorphic.

type_synonym *'a word = "'a list"*

A language is a set of words.

type_synonym *'a lang = "'a word set"*

Here are some basic operations on languages, namely concatenation and exponentiation:

$concat\ L\ M \equiv \{v\ @\ w\ |v\ w.\ v \in L \wedge w \in M\}$

$pow\ L\ 0 = \{[]\}$

$pow\ L\ (Suc\ n) = concat\ L\ (pow\ L\ n)$

The following two lemmas are useful for automation. Theorems with the attribute *intro* are automatically applied as backward rules.

lemma *empty_mem_pow_zero* [*simp*, *intro*]: "$[] \in pow\ L\ 0$"
  by (*auto simp*: *concat_def*)

lemma *append_mem_pow_SucI* [*intro*]:
  "$v \in L \Longrightarrow w \in pow\ L\ n \Longrightarrow (v\ @\ w) \in pow\ L\ (Suc\ n)$"
  by (*auto simp*: *concat_def*)

We next consider a simplified form of regular expressions.

datatype *'a rexp = Atom 'a | Concat ('a rexp) ('a rexp) | Or ('a rexp) ('a rexp) | Star ('a rexp)*

Define a recursive function that computes the language of a regular expression.

fun *lang* :: *"'a rexp ⇒ 'a lang"*

Hint: use the functions *concat* and *pow*. For the *Star* case, use one of the following:

$lang\ (Star\ r) = (\bigcup n.\ ...)$

$lang\ (Star\ r) = \{w.\ ...\}$

Define a recursive function that pulls *Or*s outward whenever a *Concat* meets an *Or*, i.e. replace terms such as $(a|b)c$ by $ab|bc$. For simplicity, you should ignore new clashes of *Or*s and *Concat*s created by your function (you will run into termination troubles otherwise). For example, $((a|b)c)d$ should be transformed to $(ab|bc)d$, not $abd|bcd$, and $((a|b)|c)d$ should be transformed to $(ad|bd)|cd$.

fun *or_outward* :: *"'a rexp ⇒ 'a rexp"*

Example:

value $\text{“}or\_outward\ (Star\ (Concat\ (Concat\ (Or\ (Atom\ ''a'')\ (Atom\ ''b''))\ (Atom\ ''c''))\ (Atom\ ''d''))) =$
$Star\ (Concat\ (Or\ (Concat\ (Atom\ ''a'')\ (Atom\ ''c''))\ (Concat\ (Atom\ ''b'')\ (Atom\ ''c'')))\ (Atom\ ''d''))\text{”}$

Show that your transformation doesn't change the language!

lemma *lang_or_outward_eq_lang*: $\text{“}lang\ (or\_outward\ r) = lang\ r\text{”}$

Next, define an inductive predicate that decides whether a word is in the language of a regular expression *without* using *lang*:

inductive *in_lang* :: $\text{“}'a\ rexp \Rightarrow\ 'a\ word \Rightarrow bool\text{”}$

Prove that *lang* and *in_lang* coincide by proving the following lemmas. Hint: you do not need to invent any additional lemmas.

lemma *mem_lang_if_in_lang*: $\text{“}in\_lang\ r\ w \Longrightarrow w \in lang\ r\text{”}$

lemma *in_lang_Star_if_mem_powI*:
$\text{“}(\bigwedge w.\ w \in lang\ r \Longrightarrow in\_lang\ r\ w) \Longrightarrow$
$w \in pow\ (lang\ r)\ n \Longrightarrow in\_lang\ (Star\ r)\ w\text{”}$

Hint: use *in_lang_Star_if_mem_powI* for the following proof:

lemma *in_lang_if_mem_lang*: $\text{“}w \in lang\ r \Longrightarrow in\_lang\ r\ w\text{”}$

corollary *in_lang_iff_mem_lang*: $\text{“}in\_lang\ r\ w \longleftrightarrow w \in lang\ r\text{”}$