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

WS 2021/22
14.12.2021

Semantics of Programming Lectures

Exercise Sheet 8

Starting from this sheet, we will use Isabelle 2021-1. Additionally, install the linter component from https://github.com/isabelle-prover/isabelle-linter.

Exercise 8.1 Knaster-Tarski Fixed Point Theorem

The Knaster-Tarski theorem tells us that for each set P of fixed points of a monotone function f we have a fixpoint of f which is a greatest lower bound of P. In this exercise, we want to prove the Knaster-Tarski theorem.

First we give a construction of the greatest lower bound of all fixed points P of the function f. This is the union of all sets u smaller than P and $f\ u$. Then the task is to show that this is a fixed point, and that it is the greatest lower bound of all sets in P.

Let us define *Inf_fixp*:

definition *Inf_fixp* :: "$('a\ set \Rightarrow 'a\ set) \Rightarrow 'a\ set\ set \Rightarrow 'a\ set$" **where**
"$Inf\_fixp\ f\ P = \bigcup\{u.\ u \subseteq \bigcap P \cap f\ u\}$"

To work directly with this definition is a little cumbersome, we propose to use the following two theorems:

lemma *Inf_fixp_upperbound*: "$X \subseteq \bigcap P \Longrightarrow X \subseteq f\ X \Longrightarrow X \subseteq Inf\_fixp\ f\ P$"
by (*auto simp*: *Inf_fixp_def*)

lemma *Inf_fixp_least*: "$(\bigwedge u.\ u \subseteq \bigcap P \Longrightarrow u \subseteq f\ u \Longrightarrow u \subseteq X) \Longrightarrow Inf\_fixp\ f\ P \subseteq X$"
by (*auto simp*: *Inf_fixp_def*)

Now prove, that *Inf_fixp* is acually a fixed point of f.

Hint: First prove $Inf\_fixp\ f\ P \subseteq f\ (Inf\_fixp\ f\ P)$, this will be used for the other direction. It may be helpful to first think about the structure of your proof using pen-and-paper and then translate it into Isar.

lemma *Inf_fixp*:
assumes f: "*mono f*"
and P: "$\bigwedge p.\ p \in P \Longrightarrow f\ p = p$"
shows "$Inf\_fixp\ f\ P = f\ (Inf\_fixp\ f\ P)$"

Now we prove that it is a lower bound:

lemma *Inf_fixp_lower*: "$Inf\_fixp\ f\ P \subseteq \bigcap P$"

And that it is the greatest lower bound:

lemma *Inf_fixp_greatest*:
  assumes "$f\ q = q$"
    and "$q \subseteq \bigcap P$"
  shows "$q \subseteq Inf\_fixp\ f\ P$"

Exercise 8.2 While combinator

So far, all functions that we defined were required to terminate. However, there is also a while-combinator in HOL. For instance, the IMP-program
WHILE Less (*V ''x''*) (*N 3*) *DO ''x''* ::= *Plus* (*V ''x''*) (*N 2*)
could be stated in HOL as follows:

value "*while* (λx::*nat*. $x < 3$) (λx. $x + 2$) *0*"

Take a look at the definition. What is surprising about it? Can you state and refute (using *nitpick*) lemmas about involved constants that should at first glance hold?

Using *while*, define an *exec* function for commands:

fun *exec* :: "*com* $\Rightarrow$ *state* $\Rightarrow$ *state*"

Example:

value "(*exec* (*WHILE Less* (*V ''x''*) (*N 3*) *DO ''x''* ::= *Plus* (*V ''x''*) (*N 2*)) <>) *''x''*"

Show that exec is correct:

lemma "$(c,s) \Rightarrow t \Longrightarrow exec\ c\ s = t$"

Homework 8.1 Be Original!

Submission until Sunday, Jan 8, 23:59pm. Think up a nice topic to formalize yourself, for example

- Prove some interesting result about automata/formal language theory
- Formalize some results from mathematics
- Show properties of some interesting algorithm
- ...

You will have time until after the winter break (Jan 8) to finish the project; during this time, regular homework load will be reduced. *In total, this exercise will be worth 20 points, plus bonus points for nice submissions.*

This week, you should find a topic and start to formalize some concepts from it. Be creative! The topic can be from any area; your project will also be judged on creativity. Until the end of this week (Dec 19), send an e-mail with your planned topic to the tutor (3 points).

You are also welcome to discuss your plans with the tutor. This is, however, not a necessity by any means.

For the actual project:

- you should set yourself a time limit before you start
- incomplete/unfinished formalizations are welcome and will be graded
- comment your formalization well, such that we can see what it does/is intended to do
- the code quality of your formalization also matters, so make sure to use the linter add-on component

Homework 8.2 Call Inlining (bonus)

Submission until Sunday, Dec 19, 23:59pm. This is a bonus exercise worth 2 (easy part) + 5 (hard part) points.

Consider an extension of IMP with procedures:

datatype *com* = *SKIP* | *Assign* (*char list*) *aexp* | *Seq com com* | *If bexp com com* | *While bexp com* | *CALL* (*char list*)

The big-step semantics is extended with a procedure environment *penv*. The new rule is:

$pe \vdash (pe\ p,\ s) \Rightarrow t \Longrightarrow pe \vdash (CALL\ p,\ s) \Rightarrow t$

Define a relation *inlines* between the original com' and the new version, such that the semantics are equivalent (warm-up):

inductive *inlines* :: *"penv ⇒ com ⇒ com′ ⇒ bool"* **for** *pe*

code_pred *inlines* **.**

The easy part Show correctness of your inlining:

theorem *inline_correct*: *"inlines pe c c′ ⟹ pe ⊢ (c, s) ⇒ t = (c′, s) ⇒ t"*

However, this inlining depends on the derivation for *inlines*, which does not exist for recursive programs.

The hard part We want to show that an inlining exists for every non-recursive program. First, define the set of recursive calls, i.e. all *pname*s that can occur in the execution of a *com*.

Hint: Recall the transitive reflexive closure.

definition *rec_pnames* :: *"penv ⇒ com ⇒ pname set"*

With that, define non-recursive programs, and show that for those, an inlining exists. Since any *penv* defined by source code would be finite, we may also assume that that *rec_pnames* is finite.

definition *nonrec* :: *"penv ⇒ com ⇒ bool"*
lemma *example1*: *"nonrec (λp. if p = ''end'' then SKIP else CALL ''end'') (SKIP;;CALL ''proc'')"*

lemma *example2*: *"¬nonrec (λ_. CALL ''rec'') (CALL ''proc'')"*

Hint: For the induction, find a way to encode the size of a *penv* and *com* into natural number, and induct over that using *less_induct*. For *finite sets*, *sum f A* sums up all elements after applying *f* to them.

theorem *inline_nonrec*:
  assumes *finite*: *"finite (rec_pnames pe c)"*
    and *nonrec*: *"nonrec pe c"*
  shows *"∃ c'. inlines pe c c'"*