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

WS 2022/23
13.12.2022

Semantics of Programming Languages

Exercise Sheet 8

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 ⇒ 'a set) ⇒ 'a set set ⇒ 'a set"* **where**
"Inf_fixp f P = ⋃{u. u ⊆ ⋂P ∩ 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 ⊆ ⋂P ⟹ X ⊆ f X ⟹ X ⊆ Inf_fixp f P"*
by (*auto simp*: *Inf_fixp_def*)

lemma *Inf_fixp_least*: *"(⋀u. u ⊆ ⋂P ⟹ u ⊆ f u ⟹ u ⊆ X) ⟹ Inf_fixp f P ⊆ 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 ⊆ 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 *mono*: *"mono f"*
and *P*: *"⋀p. p ∈ P ⟹ 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 ⊆ ⋂P"*

And that it is the greatest lower bound:

lemma *Inf_fixp_greatest*:
assumes "*f q = q*"
and "*q* ⊆ ⋂*P*"
shows "*q* ⊆ *Inf_fixp f P*"

Exercise 8.2 Denotational Semantics

Define a denotational semantics for REPEAT-loops.

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)
| *Repeat com bexp* ("(REPEAT _/ UNTIL _)" [0, 61] 61)

inductive
big_step :: "*com* × *state* ⇒ *state* ⇒ *bool*" (**infix** "⇒" 55)
where
Skip: "(*SKIP*,*s*) ⇒ *s*" |
Assign: "(*x* ::= *a*,*s*) ⇒ *s*(*x* := *aval a s*)" |
Seq: "⟦ (*c*$_1$,*s*$_1$) ⇒ *s*$_2$; (*c*$_2$,*s*$_2$) ⇒ *s*$_3$ ⟧ ⟹ (*c*$_1$;;*c*$_2$, *s*$_1$) ⇒ *s*$_3$" |
IfTrue: "⟦ *bval b s*; (*c*$_1$,*s*) ⇒ *t* ⟧ ⟹ (*IF b THEN c*$_1$ *ELSE c*$_2$, *s*) ⇒ *t*" |
IfFalse: "⟦ ¬*bval b s*; (*c*$_2$,*s*) ⇒ *t* ⟧ ⟹ (*IF b THEN c*$_1$ *ELSE c*$_2$, *s*) ⇒ *t*" |
WhileFalse: "¬*bval b s* ⟹ (*WHILE b DO c*,*s*) ⇒ *s*" |
WhileTrue:
"⟦ *bval b s*$_1$; (*c*,*s*$_1$) ⇒ *s*$_2$; (*WHILE b DO c*, *s*$_2$) ⇒ *s*$_3$ ⟧
⟹ (*WHILE b DO c*, *s*$_1$) ⇒ *s*$_3$"

Proof automation:

lemmas [*intro*] = *big_step.intros*
lemmas *big_step_induct* = *big_step.induct*[*split_format*(*complete*)]

inductive_cases *SkipE*[*elim*!]: "(*SKIP*,*s*) ⇒ *t*"
inductive_cases *AssignE*[*elim*!]: "(*x* ::= *a*,*s*) ⇒ *t*"
inductive_cases *SeqE*[*elim*!]: "(*c1*;;*c2*,*s1*) ⇒ *s3*"
inductive_cases *IfE*[*elim*!]: "(*IF b THEN c1 ELSE c2*,*s*) ⇒ *t*"
inductive_cases *WhileE*[*elim*]: "(*WHILE b DO c*,*s*) ⇒ *t*"
type_synonym *com_den* = "(*state* × *state*) *set*"

definition *W* :: "(*state* ⇒ *bool*) ⇒ *com_den* ⇒ (*com_den* ⇒ *com_den*)" **where**
"*W db dc* = (λ*dw*. {(*s*,*t*). *if db s then* (*s*,*t*) ∈ *dc O dw else s*=*t*})"

fun *D* :: "*com* ⇒ *com_den*" **where**
"*D SKIP* = *Id*" |
"*D* (*x* ::= *a*) = {(*s*,*t*). *t* = *s*(*x* := *aval a s*)}" |
"*D* (*c1*;;*c2*) = *D*(*c1*) *O D*(*c2*)" |

"D (IF b THEN c1 ELSE c2)
= {(s,t). if bval b s then (s,t) ∈ D c1 else (s,t) ∈ D c2}" |
"D (WHILE b DO c) = lfp (W (bval b) (D c))"

Exercise 8.3 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?

lemma *"the (opt :: bool option) = x ⟹ opt = Some x"*
nitpick[*expect=genuine*]
oops

— *undefined* is an element of its type - for unit type, it must be ().
lemma *"the None ≠ ()"*
nitpick[*expect=genuine*]
oops

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

fun *exec* :: *"com ⇒ state ⇒ 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) ⇒ t ⟹ exec c s = t"*

Does the other direction hold?

Homework 8.1 Denotational Semantics

Submission until Monday, Dec 19, 23:59pm.

We again consider the extension of IMP with non-determinism from exercise sheet 5. However, this time, we add a construct $LOOP\ c$ for non-deterministic looping. The idea is that $LOOP\ c$ can non-deterministically decide to either stop iteration and do nothing or to execute the loop body c for one more time.

datatype *Defs.com = Defs.com.SKIP | Defs.com.Assign (char list) aexp | Defs.com.Seq Defs.com Defs.com | Defs.com.If bexp Defs.com Defs.com | Defs.com.While bexp Defs.com | Or Defs.com Defs.com | ASSUME bexp | Loop Defs.com*

First extend the big-step semantics with this new construct:

inductive
big_step :: *"com × state ⇒ state ⇒ bool"* (**infix** *"⇒" 55*)
where
Skip: *"(SKIP,s) ⇒ s" |*
Assign: *"(x ::= a,s) ⇒ s(x := aval a s)" |*
Seq: *"⟦ (c₁,s₁) ⇒ s₂; (c₂,s₂) ⇒ s₃ ⟧ ⟹ (c₁;;c₂, s₁) ⇒ s₃" |*
IfTrue: *"⟦ bval b s; (c₁,s) ⇒ t ⟧ ⟹ (IF b THEN c₁ ELSE c₂, s) ⇒ t" |*
IfFalse: *"⟦ ¬bval b s; (c₂,s) ⇒ t ⟧ ⟹ (IF b THEN c₁ ELSE c₂, s) ⇒ t" |*
WhileFalse: *"¬bval b s ⟹ (WHILE b DO c,s) ⇒ s" |*
WhileTrue: *"⟦ bval b s₁; (c,s₁) ⇒ s₂; (WHILE b DO c, s₂) ⇒ s₃ ⟧ ⟹ (WHILE b DO c, s₁) ⇒ s₃" |*
OrLeft: *"⟦ (c₁,s) ⇒ s' ⟧ ⟹ (c₁ OR c₂,s) ⇒ s'" |*
OrRight: *"⟦ (c₂,s) ⇒ s' ⟧ ⟹ (c₁ OR c₂,s) ⇒ s'" |*
Assume: *"bval b s ⟹ (ASSUME b, s) ⇒ s" |*
— Your cases here:

declare *big_step.intros* [*intro*]
lemmas *big_step_induct = big_step.induct*[*split_format*(*complete*)]

inductive_cases *skipE*[*elim*!]: *"(SKIP,s) ⇒ t"*
inductive_cases *AssignE*[*elim*!]: *"(x ::= a,s) ⇒ t"*
inductive_cases *SeqE*[*elim*!]: *"(c1;;c2,s1) ⇒ s3"*
inductive_cases *OrE*: *"(c1 OR c2,s1) ⇒ s3"*
inductive_cases *IfE*[*elim*!]: *"(IF b THEN c1 ELSE c2,s) ⇒ t"*
inductive_cases *WhileE*[*elim*]: *"(WHILE b DO c,s) ⇒ t"*

Now, give a denotational semantics for this language:

type_synonym *com_den = "(state × state) set"*

fun *D* :: *"com ⇒ com_den"* **where**
"D SKIP = Id" |
"D (x ::= a) = {(s,t). t = s(x := aval a s)}" |
"D (c1;;c2) = D(c1) O D(c2)" |
"D (IF b THEN c1 ELSE c2)
= {(s,t). if bval b s then (s,t) ∈ D c1 else (s,t) ∈ D c2}" |
"D (WHILE b DO c) = lfp (W (bval b) (D c))"
— Your cases here:

Then correct the proof of the equivalence theorem between big-step and denotational semantics:

theorem *denotational_is_big_step*:
"(s,t) ∈ D(c) = ((c,s) ⇒ t)"

Use theory *HOL−IMP.Denotational* as a template for the proof!

Homework 8.2 Be Original!

Submission until Monday, Jan 9, 23:59pm. Think up a nice topic to formalize yourself! It can be from any area of mathematics, computer science, etc., but should contain some interesting proof(s) – mere definitions or implementations are not interesting.

Creativity is encouraged and will be graded, but keep in mind that formalizations can often be more difficult than anticipated. Set yourself realistic goals! You are also welcome to discuss your project with us beforehand.

Comment your formalization well such that we can see what it is intended to do.

Incomplete or unfinished formalizations are welcome and will be graded (but clean them up so it is obvious what is there and what is missing).

The project will run until the end of the winter holiday, and the regular homework load is strongly reduced during that time.

In total, this exercise will be worth 15 points, plus bonus points for nice submissions.