Semantics of Abstract Imp

Gert Smolka, Saarland University

October 28, 2019

We study a simple imperative language Imp computing with abstract states. We consider three complementary semantic disciplines for Imp: Big-step semantics, small-step semantics, and step-indexed semantics. We show that the three semantics agree. Big-step and small-step semantics are obtained with recursive inductive predicates and require interesting inductive proofs.

1 States, Actions, Tests

We have in mind a simple imperative language that can write and read registers realized with an abstract memory. All registers hold values of the same type (e.g., integers). We abstract away from concrete values and assume the following types and functions:

$$s : \mathsf{State} \;:\; \mathbf{T}$$

$$a : \mathsf{Action} \;:\; \mathbf{T} \qquad \alpha : \mathsf{Action} \to \mathsf{State} \to \mathsf{State}$$

$$b : \mathsf{Test} \;:\; \mathbf{T} \qquad \beta : \mathsf{Test} \to \mathsf{State} \to \mathsf{B}$$

Concrete examples for actions are assignments like $x := 2$ and $x := x + y$. Concrete examples for tests are comparisons like $x > 0$ and $x \le y + z$. Note that the semantics of actions and tests is given by the functions α and β.

For examples it will be useful to assume an action and a test

$$\mathsf{skip} : \mathsf{Action} \qquad\qquad \mathsf{tt} : \mathsf{Test}$$

such that $\alpha\,\mathsf{skip}\,s = s$ and $\beta\,\mathsf{tt}\,s = \mathsf{T}$ for all states s.

2 Commands

The **commands** of Imp are obtained with actions, sequentialisations, conditionals, and loops:

$$c : \mathsf{Com} \;::=\; a \mid c_1; c_2 \mid \mathsf{if}\; b\; c \mid \mathsf{while}\; b\; c$$

Commands are executed on states. The execution of a command on a state may terminate or not terminate. If execution of a command on a state terminates, it yields a state. We speak of the initial and the final state of an execution. Informally, we may describe the semantics of commands as follows:

- Action a: The state is transformed with αa.
- Sequentialisation $c_1; c_2$: First execute c_1, then execute c_2.
- Conditional $\mathsf{if}\ b\ c$: If b yields T, execute c, otherwise do nothing.
- Loop $\mathsf{while}\ b\ c$: execute c as long as b yields T.

Nontermination comes into play through loops: The loop $(\mathsf{while}\ \mathsf{tt}\ \mathsf{skip})$ does not terminate.

3 Big-Step Semantics

The big-step semantics of Imp describes a recursive interpreter that given a command and a state computes the resulting state. Since the interpretation of a command does not necessarily terminate, we formalize the interpreter with an inductive predicate

$$\vdash : \mathsf{State} \to \mathsf{Com} \to \mathsf{State} \to \mathsf{P}$$

defined with the following rules (using the notation $s, c \vdash s'$):

$$\frac{}{s, a \vdash \alpha a s} \qquad \frac{s, c_1 \vdash s' \qquad s', c_2 \vdash s''}{s, c_1; c_2 \vdash s''}$$

$$\frac{\beta b s = \mathsf{T} \qquad s, c \vdash s'}{s, \mathsf{if}\ b\ c \vdash s'} \qquad \frac{\beta b s = \mathsf{F}}{s, \mathsf{if}\ b\ c \vdash s}$$

$$\frac{\beta b s = \mathsf{T} \qquad s, c \vdash s' \qquad s', \mathsf{while}\ b\ c \vdash s''}{s, \mathsf{while}\ b\ c \vdash s''} \qquad \frac{\beta b s = \mathsf{F}}{s, \mathsf{while}\ b\ c \vdash s}$$

Note that the recursions coming with the rule for sequentialisation and the first rule for loops are binary. Without the first rule for loops the rules yield an always terminating interpreter since each recursion step employs a smaller command.

Fact 1 (Functionality) If $s, c \vdash s'$ and $s, c \vdash s''$, then $s' = s''$.

Proof By induction on $s, c \vdash s'$. ∎

Exercise 2 (Nontermination) Informally, for every command c, execution of the loop $\mathsf{while}\ \mathsf{tt}\ c$ does not terminate. Prove $\neg \exists s'.\ s, \mathsf{while}\ \mathsf{tt}\ c \vdash s'$ for all states s and all commands c.

4 Small-Step Semantics

We specify a second semantics for Imp that models single execution steps and provides for an iterative interpreter. The iterative interpreter may be seen as a machine operating on a state and a stack of commands. The machine stops if the stack is empty. Otherwise, the first command on the stack is executed, possibly updating the state and the stack. We describe the atomic execution steps of the machine with an inductive predicate

$$\succ : \mathsf{State} \to \mathcal{L}(\mathsf{Com}) \to \mathsf{State} \to \mathcal{L}(\mathsf{Com}) \to \mathbf{P}$$

defined with the following rules (using the notation $s, C \succ s', C'$).

$$\begin{array}{rcll}
s,\ a :: C & \succ & \alpha a s,\ C & \\
s,\ c_1; c_2 :: C & \succ & s,\ c_1 :: c_2 :: C & \\
s,\ \mathsf{if}\ b\ c :: C & \succ & s,\ c :: C & \text{if } \beta b s = \mathsf{T} \\
s,\ \mathsf{if}\ b\ c :: C & \succ & s,\ C & \text{if } \beta b s = \mathsf{F} \\
s,\ \mathsf{while}\ b\ c :: C & \succ & s,\ c :: \mathsf{while}\ b\ c :: C & \text{if } \beta b s = \mathsf{T} \\
s,\ \mathsf{while}\ b\ c :: C & \succ & s,\ C & \text{if } \beta b s = \mathsf{F}
\end{array}$$

To iterate execution steps $s, C \succ s', C'$, we define the **reflexive transitive closure** $\succ^*$ of $\succ$ as an inductive predicate:

$$\frac{}{s, C \succ^* s, C} \qquad \frac{s, C \succ s', C' \qquad s', C' \succ^* s'', C''}{s, C \succ^* s'', C''}$$

Informally, we may describe $\succ^*$ as follows:

$$s, C \succ^* s', C' \ \leftrightarrow\ s, C \succ \cdots \succ s', C'$$

To show that the small-step semantics agrees with the big-step semantics, we will prove the equivalence

$$s, [c] \succ^* s', [] \ \leftrightarrow\ s, c \vdash s' \tag{1}$$

in the next section. Note that $\succ$ is a non-recursive inductive predicate. So all recursion of the small-step semantics is in the extension $\succ^*$ to the reflexive transitive closure.

Fact 3

1. Subsumption: If $s, C \succ s', C'$, then $s, C \succ^* s', C'$,
2. Transitivity: If $s, C \succ^* s', C'$ and $s', C' \succ^* s'', C''$, then $s, C \succ^* s'', C''$.

Proof Subsumption is straightforward. Transitivity follows by induction on $s, C \succ^* s', C'$. ∎

Exercise 4 Prove the following properties of the small-step semantics.

a) Functionality: If $s, C \succ s', C'$ and $s, C \succ s'', C''$, then $s' = s''$ and $C' = C''$.

b) Adjunction: If $s, C \succ s', C'$, then $s, C + D \succ s', C' + D$.

c) Adjunction: If $s, C \succ^* s', C'$, then $s, C + D \succ^* s', C' + D$.

5 Agreement of Small-Step with Big-Step Semantics

We now show that the small-step semantics agrees with the big-step semantics of Imp as specified by the equivalence (1). The two directions require different proofs. Both directions are interesting and provide an excellent opportunity to get acquainted with proofs based on inductive predicates, both on paper and with Coq.

The direction from big-step to small-step semantics

$$s, c \vdash s' \quad \to \quad s, [c] \succ^* s', []$$

follows by induction on the big-step predicate $\vdash$. To use the inductive hypothesis, one needs the adjunction property (c) from Exercise 4. The proof can be simplified if one proves the more general claim

$$s, c \vdash s' \quad \to \quad \forall C.\ s, c :: C \succ^* s', C$$

building in the adjunction property.

Lemma 5 If $s, c \vdash s'$, then $s, c :: C \succ^* s', C$.

Proof By induction on $s, c \vdash s'$ using Fact 3, where C is quantified in the inductive hypothesis. ∎

The direction from small-step to big-step semantics

$$s, [c] \succ^* s', [] \quad \to \quad s, c \vdash s'$$

follows by induction on the reflexive transtive closure $\succ^*$. For the induction to go through, we need to generalize to

$$s, C \succ^* s', [] \quad \to \quad s, C \vdash s'$$

where the **generalized big-step predicate** $s, C \vdash s'$ for command stacks C is defined as one would expect:

$$\frac{}{s, [] \vdash s} \qquad\qquad \frac{s, c \vdash s' \qquad s', C \vdash s''}{s, c :: C \vdash s''}$$

The step case of the induction follows with the following lemma.

Lemma 6 (Absorption) $s, C \succ s', C' \;\to\; s', C' \vdash s'' \;\to\; s, C \vdash s''$.

Proof Case analysis on $s, C \succ s', C'$ with inversion of $s', C' \vdash s''$ (Exercise 9). For instance, we have the case $s, c_1; c_2 :: C \succ s, c_1 :: c_2 :: C$ for sequentialization, where we have to prove

$$s, c_1 :: c_2 :: C \vdash s'' \;\to\; s, c_1; c_2 :: C \vdash s''$$

which is straightforward. ∎

Lemma 7 $s, C \succ^* s', [] \;\to\; s, C \vdash s'$.

Proof Induction on $s, C \succ^* s', []$ using Lemma 6. ∎

Theorem 8 $s, c \vdash s'$ if and only if $s, [c] \succ^* s', []$.

Proof Follows with Lemmas 5 and 7. ∎

Exercise 9 Verify the following inversion lemmas:

a) If $s, [] \succ^* s'$, then $s = s'$.

b) If $s, c :: C \vdash s'$, then $s, c \vdash s_1$ and $s_1, C \vdash s'$ for some s_1.

c) If $s, [c] \vdash s'$, then $s, c \vdash s'$.

Exercise 10 For Lemma 6, verify the case for loops where the test is satisfied.

6 Step-Indexed Semantics

The big-step predicate $s, c \vdash s'$ describes a recursive interpreter for Abstract Imp. The interpreter can be realized as a procedure $\mathsf{Com} \to \mathsf{State} \to \mathsf{State}$ in programming languages. Since constructive type theory can only express total functions, such a function cannot be expressed in type theory, however. Neither, a total function

$$\mathsf{Com} \to \mathsf{State} \to \mathcal{O}(\mathsf{State})$$

deciding non-termination can be expressed in type theory

since termination is undecidable in general for Abstract Imp. However, we can express a *step-indexed interpretation function*

$$\mathsf{N} \to \mathsf{Com} \to \mathsf{State} \to \mathcal{O}(\mathsf{State})$$

that yields the final state if it can be computed with the recursion depth given as first argument.

Figure 1 shows the definition of the step-indexed interpreter. We have given it a higher-order formulation separating the handling of the **step index** (the first

$$
\begin{aligned}
\sigma &: \mathsf{N} \to \mathsf{Com} \to \mathsf{State} \to \mathcal{O}(\mathsf{State}) \\
\sigma\, 0 &:= \lambda cs.\emptyset \\
\sigma\, (\mathsf{S}n) &:= \sigma'(\sigma\, n)
\end{aligned}
$$

$$
\begin{aligned}
\sigma' &: (\mathsf{Com} \to \mathsf{State} \to \mathcal{O}(\mathsf{State})) \to \mathsf{Com} \to \mathsf{State} \to \mathcal{O}(\mathsf{State}) \\
\sigma'\, F a\, s &:= {}^\circ\alpha a s \\
\sigma'\, F (c_1; c_2) s &:= \mathsf{seq}\, (Fc_1)\, (Fc_2)\, s \\
\sigma'\, F (\mathsf{if}\, b\, c) s &:= \mathsf{if}\ \beta b s\ \mathsf{then}\ Fcs\ \mathsf{else}\ {}^\circ s \\
\sigma'\, F (\mathsf{while}\, b\, c) s &:= \mathsf{if}\ \beta b s\ \mathsf{then}\ \mathsf{seq}\, (Fc)\, (F(\mathsf{while}\, b\, c))\, s\ \mathsf{else}\ {}^\circ s
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{seq} &: (\mathsf{State} \to \mathcal{O}(\mathsf{State})) \to (\mathsf{State} \to \mathcal{O}(\mathsf{State})) \to \mathsf{State} \to \mathcal{O}(\mathsf{State}) \\
\mathsf{seq}\, f g s &:= \mathsf{match}\ f s\ [\, \emptyset \Rightarrow \emptyset \mid {}^\circ s' \Rightarrow g s'\,]
\end{aligned}
$$

Figure 1: Step-indexed interpreter

argument) and the interpretation of commands. Note that for a given step index a command is interpreted as a function $\mathsf{State} \to \mathcal{O}(\mathsf{State})$ where a result $\emptyset$ indicates that the final state could not be computed with the given step index. The auxiliary function seq handles sequentialization of two functions $\mathsf{State} \to \mathcal{O}(\mathsf{State})$, a feature used for the interpretation of sequentializations and loops.

We will prove that the step-indexed interpreter agrees with the big-step semantics

$$s, c \vdash s' \ \leftrightarrow\ \exists n.\ \sigma n c s = {}^\circ s'$$

We will also prove that the step-indexed interpreter is *monotone*:

$$\sigma n c s = {}^\circ s' \ \to\ \sigma(\mathsf{S}n) c s = {}^\circ s'$$

Lemma 11 $\sigma n c s = {}^\circ s' \to s, c \vdash s'$.

Proof By induction on n with c, s, and s' quantified followed by case analysis on c in the successor case. Each case is straightforward. Direction $\to$ of Exercise 16 is useful. ∎

Lemma 12 $\sigma n c s = {}^\circ s' \ \to\ \sigma(\mathsf{S}n) c s = {}^\circ s'$.

Proof By induction on n with c, s, and s' quantified followed by case analysis on c in the successor case. Each case is straightforward. Direction $\to$ of Exercise 16 is useful. ∎

Theorem 13 (Monotonicity) $m \le n \to \sigma m c s = {}^\circ s' \to \sigma n c s = {}^\circ s'$.

Proof By induction on n. The successor case follows with Lemma 12. ∎

Lemma 14 $s, c \vdash s' \to \exists n.\ \sigma n c s = {}^\circ s'$.

Proof Induction on $s, c \vdash s'$ using monotonicity (Theorem 13) for sequentialization and loops. Direction $\leftarrow$ of Exercise 16 is useful. ∎

Theorem 15 (Agreement) $s, c \vdash s' \leftrightarrow \exists n.\ \sigma n c s = {}^\circ s'$.

Proof Lemmas 14 and 11. ∎

Exercise 16 Prove $\mathtt{seq}\ f g s = {}^\circ s' \leftrightarrow \exists s''.\ f s = {}^\circ s'' \wedge g s'' = {}^\circ s''$. Note that both directions of this lemma are useful in the Coq proofs.

Exercise 17 Prove $\sigma m c s = {}^\circ s_1 \to \sigma n c s = {}^\circ s_2 \to s_1 = s_2$.

Exercise 18 Define a step-indexed big-step semantics $s, c \vdash^n s'$ such that

$$s, c \vdash^n s' \leftrightarrow \sigma n c s = {}^\circ s'$$

Prove the equivalence.

7 Remarks

Several of the proofs (e.g., Lemmas 5 and 6) require the verification of many cases involving many variables. This is tedious and error-prone if done by hand. With Coq, the verifications can be done semi-automatically using the eauto tactic. The automation of the inversions needed for Lemma 6 requires a custom tactic applying inversion lemma (b) from Exercise 9 to the assumptions. See the accompanying Coq development for more information, where the proof script for Lemma 6 is a one-liner. Doing this proof in detail by hand requires dozens of lines.