IMP — Big Step Dynamic Semantics

Neil Ghani and Conor McBride

October 18, 2018

1 Syntax

We concern ourselves with the fragment of IMP given by the grammar:

```
<iexp>
  ::= <number>
    | <var>
    | <iexp> <iop> <iexp>
    | new <var> := <iexp> in <iexp>
    | do <command> return <iexp>

<iop> ::= + | -

<command>
  ::= {<block>}
    | <var> := <iexp>
    | if (<bexp>) <command> else <command>
    | while (<bexp>) <command>
    | new <var> := <iexp> in <command>

<block>
  ::=
    | <command>; <block>

<bexp>
  ::= <bit>
    | <bexp> & <bexp>
    | <bexp> \| <bexp>
    | ! <bexp>
    | <iexp> <comparator> <iexp>

<bit> ::= 0 | 1

<comparator> ::= == | != | < | > | <= | >=
```

That is, we have removed (for now) the apparatus for defining and invoking new functions and commands.

2 Program Variables versus Metavariables, Metapatterns

Specific, programs may contain variables, which refer to memory allocated for the storage of values.

When we talk about programs in general, we give names to their parts. Those names are not program variables, but rather part of our language for talking about programs, so we try to be careful to call them *meta*variables. E.g., we might talk about the

$$\texttt{new}\ x := e_0\ \texttt{in}\ e_1$$

construct. Here, the metavariables e_0 and e_1 stand for arbitrary integer expressions, and (perhaps confusingly), the metavariable x stands for an arbitrary program variable, like `fred` or `x27`.

It is common practice to adopt metavariable naming conventions to allow you to guess more easily what a metavariable stands for, given what it is called. It is good to be explicit about these conventions. E.g., we shall typically use e to stand for integer expressions and x, y, z to stand for program variables.

We call $\texttt{new}\ x := e_0\ \texttt{in}\ e_1$ a *metapattern*, because it stands for a whole class of expressions — those whose top-level construct is `new`..., with metavariables naming the parts so that we can talk about them later.

3 Stores, Scope and Shadowing

Let us focus on program variables. These are labels for mutable storage of integers. Correspondingly, when a variable is used as an integer expression, it stands for the integer value stored, and when a variable stands on the left of an assignment command, the stored value is to be replaced. Neither makes sense if the variable is not in *scope*. Programs should make use of only those variables which are locally known to refer to stored values: to be 'in scope' is to be such a variable.

How do variables come into scope? In the above fragment, that is the role of the $\texttt{new}\ x := e_0\ \texttt{in}\ e_1$ construct. The expression e_0 is to be computed in the current scope (which may or may not involve a variable called x); its value v_0 is used to initialise a new local variable x which is in scope for the evaluation of e_1; when e_1 has been evaluated, the original scope is restored. So

```
new x := 6 in do x := x + 1 return x
```

yields the value 7. Crucially, $\texttt{new}\ x := e_0\ \texttt{in}\ e_1$ does not corrupt any existing memory labelled with the name x. Rather, such memory becomes inaccessible to e_1, because, there and only there, variable x refers to the newly introduced local store. That is

```
new x := 37 in (new x := 42 in do x := x + 1 return x) + x
```

yields 80, because the two `new x` constructs introduce distinct local store. It is the inner `x`, initialised to 42, which is in scope for the `do ... return ...`, and so incremented to 43 and returned. The `x` to the right of + refers is the outer one, initialised to 37 and unchanged throughout.

We shall need to account for stores scope and shadowing in any explanation of how IMP programs run. A store is a finite sequence (order matters) of assignments of integer values to variables. The same variable may appear more than once.

$$\begin{array}{rcl} \langle \mathit{store} \rangle & ::= & \\ & | & \langle \mathit{store} \rangle, \langle \mathit{var} \rangle := \langle \mathit{integer} \rangle \end{array}$$

We may use σ as a metavariable to stand for stores as we develop our semantics.

We shall need to write metapatterns which characterise (or 'match') stores with particular properties, containing metavariables which stand for parts of those stores. We write ψ to stand for store metapatterns (which makes it a metametavariable!).

- a metavariable σ is a metapattern which matches any contiguous part of a store (including, perhaps, a whole store)
- $x := v$ matches the (part of a) store which assigns value v to the variable that x stands for

- if ψ_0 and ψ_1 are store metapatterns, then ψ_0, ψ_1 is also a metapattern, matching any store which can be split at some point into a left part matching ψ_0 and a right part matching ψ_1
- metapattern $x\backslash\psi$ matches any part of a store which matches ψ and does *not* contain an assignment for x

Examples:

- σ matches any store
- $x := v$ matches the store which consists exactly of the assignment of v to x
- $\sigma, x := v$ matches the store whose rightmost assignment is of v to x, with σ being everything to the left of that assignment
- $\sigma_0, x := v, x\backslash\sigma_1$ matches any store which contains at least one assignment to x, with v being the rightmost value assigned to x, because the $x\backslash\sigma_1$ matches only stores σ_1 which do not contain x; to the left, we have σ_0, which may contain zero or more other assignments to x

By convention, scopes are ordered by *locality*, with more local things to the right of less local things. The last example, in particular, allows us to pick out the *most local* x which is the x that is in scope. The xs in σ_0 are *shadowed*.

4 Judgements and Rules

We make logical statements about parts of programs which we call *judgments*. Judgements are not just any old nonsense: they come in particular shapes called *judgement forms*. The job of a judgement form is to characterise an aspect of what there is to know. Judgement forms can be given a grammar. E.g.,

$$\langle \mathit{judgement} \rangle ::= \langle \mathit{store} \rangle \,|\, \langle \mathit{iexp} \rangle \Downarrow \langle \mathit{store} \rangle \,|\, \langle \mathit{integer} \rangle$$

We often state informally what is the intuition for a given judgment form — what it tells us. E.g.,

$$\sigma_0 \,|\, e \Downarrow \sigma_1 \,|\, v$$

means 'starting with store σ_0, expression e evaluates yielding final store σ_1 and integer value v' (where u, v are metavariables standing for integer values).

We then write *rules* which tell us how to *derive* (i.e., prove) that judgements hold. Rules are often presented in 'natural deduction' form:

$$\frac{J_1 \quad \ldots \quad J_n}{J}$$

means 'to show judgement J, you must in turn show each of tyhe judgments $J_1, \ldots, J_n$'. We say J is the *conclusion* of the rule and the J_i are its *premises*. Whatever else anybody may have told you, we always work *backwards* from a judgment we want to establish to find a *derivation* which establishes it. We can stop when we reach a rule with zero premises: such a rule is called an *axiom*. E.g.

$$\frac{}{\sigma \,|\, v \Downarrow \sigma \,|\, v}$$

is an axiom stating that an expression which takes the form of an integer value evaluates to that very value, without changing the state.

We should add that the positions for things in judgements have *modes*: they are either *inputs* or *outputs*. In our example judgement form, the modes are

input		input		output		output
σ_0	$\|$	e	$\Downarrow$	σ_1	$\|$	v

We tend to keep inputs to the left and outputs to the right.

Modes help us to read and understand rules.

- The inputs of a rule's conclusion are metapatterns (naming the parts of the inputs with metavariables);
- the inputs of a rule's premises are metaexpressions (which make use of known metavariables but do not introduce new metavariables);
- the outputs of a rule's premises are metapatterns (which extract useful information from the premises);
- the outputs of a rule's conclusion are metaexpression (which may use all the metavariables in the rule).

That is, we read rules *clockwise*, from the inputs of the conclusion, up and through the premises left-to-right, then down to the conclusion's outputs. Think of a rule as a server for its conclusion (you ask it to establish a fact) and a client for its premises (it needs to establish that the right conditions hold, in order to serve you your fact). The modes explain the communication in that process. E.g.,

$$\frac{\sigma_0 \mid e_0 \Downarrow \sigma_1 \mid v_0 \quad \sigma_1 \mid e_1 \Downarrow \sigma_2 \mid v_1}{\sigma_0 \mid e_0 + e_1 \Downarrow \sigma_2 \mid v_0 + v_1}$$

tells us how to evaluate addition expressions. The conclusion's input patterns tell us the starting store σ_0 and the two subexpressions e_0 and e_1; the first premise uses σ_0 and e_0 for its inputs, and we learn from its outputs σ_1 (the store after evaluating e_0) and v_0 (the value of e_0); the second premise uses σ_1 as the starting store for e_1, giving us a resulting store σ_2 and the value v_1 of e_1; the conclusion's outputs then say that the final store is σ_2 and the value of the whole expression is $v_0 + v_1$. Note that the $+$ in the input is *syntax* (the operator in the language grammar), while the $+$ in the outputs is actual addition of integers.

5 Big-step Operational Semantics for IMP

Let's put all these pieces together and give a 'big-step' operational semantics for IMP. It's a *semantics* because it says what programs mean. It's *operational* because it characterises meaning by saying what things do. It's *big-step* because judgments characterise complete behaviour in one go.

Metavariable conventions Let us write

e	for an	$\langle iexp \rangle$	v	for an	$\langle integer \rangle$
p	for a	$\langle bexp \rangle$	b	for a	$\langle bit \rangle$
c	for a	$\langle command \rangle$			
cs	for a	$\langle block \rangle$			
σ	for a	$\langle store \rangle$			

where $\langle integer \rangle$ characterises not only the $\langle number \rangle$s, but also the negative values.

Judgement forms

- $\sigma_0 \mid e \Downarrow \sigma_1 \mid v$
  starting with store σ_0, integer expression e evaluates yielding final store σ_1 and integer value v
- $\sigma_0 \mid p \Downarrow \sigma_1 \mid b$
  starting with store σ_0, boolean expression p evaluates yielding final store σ_1 and bit value b
- $\sigma_0 \mid c \Downarrow \sigma_1$
  starting with store σ_0, executing command c terminates, yielding final store σ_1

- $\sigma_0 \mid cs \Downarrow \sigma_1$
  starting with store σ_0, executing block cs terminates, yielding final store σ_1

There's a key *invariant* about the store to be preserved by all of these judgements: the store 'after' will always contain the same *variables* in the same order as the store 'before', even if their *values* have changed.

We may now write the rules for each judgement form. It is traditional to write the judgement form in a box at the top of the table of the rules which use that judgement form in its conclusion.

$$\boxed{\sigma_0 \mid e \Downarrow \sigma_1 \mid v}$$

$$\frac{}{\sigma \mid v \Downarrow \sigma \mid v}$$

$$\frac{}{\sigma, x := v, x \backslash \sigma' \mid x \Downarrow \sigma, x := v, \sigma' \mid v}$$

$$\frac{\sigma_0 \mid e_0 \Downarrow \sigma_1 \mid v_0 \quad \sigma_1 \mid e_1 \Downarrow \sigma_2 \mid v_1}{\sigma_0 \mid e_0 + e_1 \Downarrow \sigma_2 \mid v_0 + v_1}$$

$$\frac{\sigma_0 \mid e_0 \Downarrow \sigma_1 \mid v_0 \quad \sigma_1 \mid e_1 \Downarrow \sigma_2 \mid v_1}{\sigma_0 \mid e_0 - e_1 \Downarrow \sigma_2 \mid v_0 - v_1}$$

$$\frac{\sigma_0 \mid e_0 \Downarrow \sigma_1 \mid v_0 \quad \sigma_1, x := v_0 \mid e_1 \Downarrow \sigma_2, x := v_2 \mid v_1}{\sigma_0 \mid \texttt{new}\ x := e_0\ \texttt{in}\ e_1 \Downarrow \sigma_2 \mid v_1}$$

$$\frac{\sigma_0 \mid c \Downarrow \sigma_1 \quad \sigma_1 \mid e \Downarrow \sigma_2 \mid v}{\sigma_0 \mid \texttt{do}\ c\ \texttt{return}\ e \Downarrow \sigma_2 \mid v}$$

Note how the rule for variables carefully chooses the most local occurrence of the variable in the store and gives back the same store. Note how the rule for `new` introduces a fresh local part of the store for its second premise: this local store is discarded when producing the final store in the conclusion, ensuring that the invariant is respected.

Let us have commands and blocks, next.

$$\boxed{\sigma_0 \mid c \Downarrow \sigma_1}$$

$$\frac{\sigma_0 \mid cs \Downarrow \sigma_1}{\sigma_0 \mid \{cs\} \Downarrow \sigma_1}$$

$$\frac{\sigma_0 \mid e \Downarrow \sigma_1, x := v_1, x \backslash \sigma_1' \mid v}{\sigma_0 \mid x := e \Downarrow \sigma_1, x := v, \sigma_1'}$$

$$\frac{\sigma_0 \mid p \Downarrow \sigma_1 \mid 0 \quad \sigma_1 \mid c_0 \Downarrow \sigma_2}{\sigma_0 \mid \texttt{if}\ (p)\ c_1\ \texttt{else}\ c_0 \Downarrow \sigma_2}$$

$$\frac{\sigma_0 \mid p \Downarrow \sigma_1 \mid 1 \quad \sigma_1 \mid c_1 \Downarrow \sigma_2}{\sigma_0 \mid \texttt{if}\ (p)\ c_1\ \texttt{else}\ c_0 \Downarrow \sigma_2}$$

$$\frac{\sigma_0 \mid p \Downarrow \sigma_1 \mid 0}{\sigma_0 \mid \texttt{while}\ (p)\ c \Downarrow \sigma_1}$$

$$\frac{\sigma_0 \mid p \Downarrow \sigma_1 \mid 1 \quad \sigma_1 \mid c \Downarrow \sigma_2 \quad \sigma_2 \mid \texttt{while}\ (p)\ c \Downarrow \sigma_3}{\sigma_0 \mid \texttt{while}\ (p)\ c \Downarrow \sigma_3}$$

$$\frac{\sigma_0 \mid e \Downarrow \sigma_1 \mid v \quad \sigma_1, x := v \mid c \Downarrow \sigma_2, x := v_2}{\sigma_0 \mid \texttt{new}\ x := e\ \texttt{in}\ c \Downarrow \sigma_2}$$

$$\boxed{\sigma_0 \mid cs \Downarrow \sigma_1}$$

$$\frac{}{\sigma \mid \Downarrow \sigma}$$

$$\frac{\sigma_0 \mid c \Downarrow \sigma_1 \quad \sigma_1 \mid cs \Downarrow \sigma_2}{\sigma_0 \mid c;\ cs \Downarrow \sigma_2}$$

Finally, we need boolean expressions. Watch carefully how 'and' and 'or' work.

$$\boxed{\sigma_0 \mid p \Downarrow \sigma_1 \mid b}$$

$$\frac{}{\sigma \mid b \Downarrow \sigma \mid b}$$

$$\frac{\sigma_0 \mid p_0 \Downarrow \sigma_1 \mid 0}{\sigma_0 \mid p_0 \& p_1 \Downarrow \sigma_1 \mid 0}$$

$$\frac{\sigma_0 \mid p_0 \Downarrow \sigma_1 \mid 1 \quad \sigma_1 \mid p_1 \Downarrow \sigma_2 \mid b}{\sigma_0 \mid p_0 \& p_1 \Downarrow \sigma_2 \mid b}$$

$$\frac{\sigma_0 \mid p_0 \Downarrow \sigma_1 \mid 1}{\sigma_0 \mid (p_0 | p_1) \Downarrow \sigma_1 \mid 1}$$

$$\frac{\sigma_0 \mid p_0 \Downarrow \sigma_1 \mid 0 \quad \sigma_1 \mid p_1 \Downarrow \sigma_2 \mid b}{\sigma_0 \mid (p_0 | p_1) \Downarrow \sigma_2 \mid b}$$

$$\frac{\sigma_0 \mid p \Downarrow \sigma_1 \mid 0}{\sigma_0 \mid !p \Downarrow \sigma_1 \mid 1}$$

$$\frac{\sigma_0 \mid p \Downarrow \sigma_1 \mid 1}{\sigma_0 \mid !p \Downarrow \sigma_1 \mid 0}$$

$$\frac{\sigma_0 \mid e_0 \Downarrow \sigma_1 \mid v_0 \quad \sigma_1 \mid e_1 \Downarrow \sigma_2 \mid v_1}{\sigma_0 \mid e_0 \ cmp \ e_1 \Downarrow \sigma_2 \mid v_0 \ cmp \ v_1}$$

Where we expect all the comparison operators *cmp* to be defined for integer values yielding one bit results.

The rules for 'and' and 'or' make explicit that if the first subexpression determines the outcome, the second is *not* evaluated. This is known as *short-circuiting*.