Typed IMP

Neil Ghani and Conor McBride

November 6, 2018

1 Introduction

We've seen a fragment of IMP, whose grammar has been constructed with care:

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

This grammar is very careful to separate the integer expressions, the Boolean expressions, and the commands. Everything fits together properly, and we get a fairly sensible dynamic semantics as a result, but there are issues:

- Although we keep the types apart, we can still have *scope* errors in grammatically correct programs: i.e., trying to access variables with no corresponding entry in the store.
- We are restricted to integer variables only.

- We see duplication of the `new` construct between integer expressions and commands.
- If we wanted to add another type (strings, perhaps), we would need new nonterminal symbols in the grammar and more repetition.
- If we wanted to add types of arrays of things with any element type we want (including arrays of arrays of things, etc.), we would need infinitely many nonterminal symbols in the grammar.

Perhaps it is a mistake to expect the *grammar* to do all the hard work for us, when we could use a sensible *static semantics* instead. In this document, we give that a try.

2 Smash the Grammar!

Let us start by making the grammar simpler but more chaotic.

```
<exp>
  ::= <number>
    | <bit>
    | <place>
    | <place> := <exp>
    | {<block>}
    | ! <exp>
    | <exp> <op> <exp>
    | new <var> := <exp> in <exp>
    | if (<exp>) <exp> else <exp>
    | while (<exp>) <exp>

<block>
  ::=
    | <exp>; <block>

<place>
  ::= <var>

<bit>
  ::= false | true

<op> ::= == | != | < | > | <= | >= | & | \| | + | -
```

As you can see, the grammar is smaller but works less hard to stop us making fools of ourselves, e.g., by writing

```
if ({}) 7 else 2 + (x := true)
```

But that's what the static semantics is for!

(If you're wondering why `<place>` has been made to exist, it's because we're going to add arrays, so that variables are not the only places where things can be stored.)

(If you're wondering why bits are now called `false` and `true` instead of 0 and 1, that's to distinguish them from the numbers 0 and 1.)

3 Types

For now, let us have three types, which may instantiate type metavariables σ and τ in judgements.

$$\langle \textit{type} \rangle ::= \texttt{bool} \mid \texttt{int} \mid \texttt{void}$$

The type `void` is for *commands*.

If we know the types of the variables in the store, we should be able to figure out the types of expressions. Let us have typing contexts, Γ, given by the grammar

$$\langle \mathit{context} \rangle ::= \quad | \ \langle \mathit{context} \rangle, \langle \mathit{var} \rangle : \langle \mathit{type} \rangle$$

We may then have typing judgement forms for expressions e, places p and blocks es.

Let us have the rules for places, which effectively look up the rightmost occurrence of a variable in the context.

$$\boxed{\Gamma \vdash x \leftarrow \tau}$$

$$\frac{}{\Gamma, x : \tau \vdash x \leftarrow \tau} \qquad \frac{\Gamma \vdash x \leftarrow \tau \quad x \neq y}{\Gamma, y : \sigma \vdash x \leftarrow \tau}$$

The type of a block is the type of the last thing in the block, with `void` being the type of the empty block. The judgement form is designed to keep track of the last thing we remember seeing. That convention allows us to drop the `do` c `return` e construct from the language, as $\{c; e;\}$ now does its job.

$$\boxed{\Gamma \vdash \sigma | es : \tau}$$

$$\frac{}{\Gamma \vdash \tau | \ : \tau} \qquad \frac{\Gamma \vdash e : \sigma' \quad \Gamma \vdash \sigma' | es : \tau}{\Gamma \vdash \sigma | e; es : \tau}$$

We collect the types of our infix operators in a set (known as a *signature*) which gets named Σ. It is a set of quadruples, giving each operator its two input types and its output type, respectively

$$\Sigma = \{(+, \mathtt{int}, \mathtt{int}, \mathtt{int}), (-, \mathtt{int}, \mathtt{int}, \mathtt{int}), (\&, \mathtt{bool}, \mathtt{bool}, \mathtt{bool}), \ldots, (<, \mathtt{int}, \mathtt{int}, \mathtt{bool}), \ldots\}$$

It even makes sense to *overload* operators

$$\{(==, \mathtt{int}, \mathtt{int}, \mathtt{bool}), (==, \mathtt{bool}, \mathtt{bool}, \mathtt{bool})\} \subset \Sigma$$

but we should avoid ambiguity by ensuring that the operator and the input types determine the output type.

$$(op, \sigma_0, \sigma_1, \tau_0), (op, \sigma_0, \sigma_1, \tau_1) \in \Sigma \Rightarrow \tau_0 = \tau 1$$

Now we can give the types of expressions:

$$\boxed{\Gamma \vdash e : \tau}$$

$$\frac{}{\Gamma \vdash n : \mathtt{int}} \qquad \frac{}{\Gamma \vdash \mathtt{false} : \mathtt{bool}} \qquad \frac{}{\Gamma \vdash \mathtt{true} : \mathtt{bool}}$$

$$\frac{\Gamma \vdash p \leftarrow \tau}{\Gamma \vdash p : \tau} \qquad \frac{\Gamma \vdash p \leftarrow \tau \quad \Gamma \vdash e : \tau}{\Gamma \vdash p := e : \mathtt{void}}$$

$$\frac{\Gamma \vdash \mathtt{void} | es : \tau}{\Gamma \vdash \{es\} : \tau}$$

$$\frac{\Gamma \vdash e : \mathtt{bool}}{\Gamma \vdash !e : \mathtt{bool}} \qquad \frac{\Gamma \vdash e_0 : \sigma_0 \quad \Gamma \vdash e_1 : \sigma_1 \quad (op, \sigma_0, \sigma_1, \tau) \in \Sigma}{\Gamma \vdash e_0 \ op \ e_1 : \tau}$$

$$\frac{\Gamma \vdash e_0 : \sigma \quad \Gamma, x : \sigma \vdash e_1 : \tau}{\Gamma \vdash \mathtt{new} \ x := e_0 \ \mathtt{in} \ e_1 : \tau}$$

$$\frac{\Gamma \vdash e_b : \mathtt{bool} \quad \Gamma \vdash e_t : \tau \quad \Gamma \vdash e_f : \tau}{\Gamma \vdash \mathtt{if} \ (e_b) \ e_t \ \mathtt{else} \ e_f : \tau}$$

$$\frac{\Gamma \vdash e_b : \mathtt{bool} \quad \Gamma \vdash e_l : \mathtt{void}}{\Gamma \vdash \mathtt{while} \ (e_b) \ e_l : \mathtt{void}}$$

Observe that we may have places of any type (even `void`!) and that the assignment rule checks that what we put in a place respects that type. The whole assignment has type `void`, because it is a command.

The rule for blocks ensures that `void` is the type of {} by making it the initial 'type seen most recently'.

The operator rule just looks up the relevant information in the signature.

We have *one* rule for `new` $x := e_0$ `in` e_1 which gives a new local x the type of its initialising expression and allows the body to have whatever type we choose.

The rule for `if` (e_b) e_t `else` e_f insists that the condition is of type `bool`, and that the branches have the *same* type, which is the type of the whole thing. We may now write expressions like

```
if (x >= y) then x else y
```

to compute the maximum of two integer values.

Note that `while` (e_b) e_l insists on a loop body of type `void`: that is perhaps slightly picky, but we must surely give the whole while-loop type `void`, as a loop whose condition is initially false is unlikely to give us anything more informative.

Moreover, the rules for typing are *syntax-directed* (there is one rule for each construct, and it invokes the rules only for subconstructs) and *deterministic* (we can treat the type of the construct as an *output* and the other data (context, type last seen in a block, the given construct) as *inputs*). There is a *type synthesis* algorithm!

4 What Happens to the Dynamic Semantics?

We need to make a few small adjustments to the presentation of the dynamic semantics. For one thing, commands are now lumped in with expressions, so we have to give them values. We may simply invent a trivial value, $\star$, for commands.

$$\langle \mathit{value} \rangle ::= \langle \mathit{integer} \rangle \mid \texttt{false} \mid \texttt{true} \mid \star$$

and we have a typing judgement for values (which needs no context)

$$\boxed{v : \tau}$$

$$\frac{}{n : \texttt{int}} \qquad \frac{}{\texttt{false} : \texttt{bool}} \qquad \frac{}{\texttt{true} : \texttt{bool}} \qquad \frac{}{\star : \texttt{void}}$$

We can say when a *store* fits a *context*:

$$\frac{}{\ :\ } \qquad \frac{\sigma \;:\; \Gamma \quad v : \tau}{\sigma, x := v \;:\; \Gamma, x : \tau}$$

That is, we think of contexts as the types of stores, with the empty store fitting the empty context and each extension having its designated type.

We may now have judgment forms

- $\sigma_0 \mid e \Downarrow \sigma_1 \mid v$
  starting with store σ_0, expression e yields final store σ_1 and value v
- $\sigma_0 \mid v_0 \setminus es \Downarrow \sigma_1 \mid v_1$
  starting with store σ_0 and last seen value v_0, block es yields final store σ_1 and value v_1
- $\sigma \mid x \Rightarrow v$
  accessing variable x in store σ yields value v
- $\sigma_0 \mid x := v \Downarrow \sigma_1$
  updating variable x with value v in store σ_0 yields store σ_1

Let us have edited highlights of the rules. For expressions, we now have

$$\frac{\sigma \mid x \;\Rightarrow\; v}{\sigma \mid x \;\Downarrow\; \sigma \mid v} \qquad \frac{\sigma_0 \mid e \;\Downarrow\; \sigma_1 \mid v \quad \sigma_1 \mid x := v \;\Downarrow\; \sigma_2}{\sigma_0 \mid x := v \;\Downarrow\; \sigma_2 \mid \star} \qquad \frac{\sigma_0 \mid \star \setminus es \;\Downarrow\; \sigma_1 \mid v}{\sigma_0 \mid \{es\} \;\Downarrow\; \sigma_1 \mid v}$$

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

$$\frac{\sigma_0 \mid e_b \;\Downarrow\; \sigma_1 \mid \texttt{false}}{\sigma_0 \mid \texttt{while}\ (e_b)\ e_l \;\Downarrow\; \sigma_1 \mid \star}$$

$$\frac{\sigma_0 \mid e_b \;\Downarrow\; \sigma_1 \mid \texttt{true} \qquad \sigma_1 \mid e_l \;\Downarrow\; \sigma_2 \mid \star \qquad \sigma_2 \mid \texttt{while}\ (e_b)\ e_l \;\Downarrow\; \sigma_3 \mid \star}{\sigma_0 \mid \texttt{while}\ (e_b)\ e_l \;\Downarrow\; \sigma_3 \mid \star}$$

For blocks,

$$\frac{}{\sigma \mid v \setminus \;\Downarrow\; \sigma \mid v} \qquad \frac{\sigma_0 \mid e \;\Downarrow\; \sigma_1 \mid v_1 \quad \sigma_1 \mid v_1 \setminus es \;\Downarrow\; \sigma_2 \mid v_2}{\sigma_0 \mid v_1 \setminus e; es \;\Downarrow\; \sigma_2 \mid v_2}$$

For looking up places,

$$\frac{}{\sigma, x := v \mid x \;\Rightarrow\; v} \qquad \frac{\sigma \mid x \;\Rightarrow\; v \quad x \neq y}{\sigma, y := v' \mid x \;\Rightarrow\; v}$$

For updating places,

$$\frac{}{\sigma, x := v_0 \mid x := v \;\Downarrow\; \sigma, x := v} \qquad \frac{\sigma_0 \mid x := v \;\Downarrow\; \sigma_1 \quad x \neq y}{\sigma_0, y := v' \mid x := v \;\Downarrow\; \sigma_1, y := v'}$$

5 What to Prove?

We ought to get repaid with some confidence about the behaviour of the dynamic semantics, in return for imposing a static semantics.

For example, if we know $\sigma_0 \;:\; \Gamma$ and $\Gamma \vdash p \leftarrow \tau$, then there *must* be a v such that $v : \tau$ and $\sigma \mid p \;\Rightarrow\; v$. The static semantics checks that all variables accesses are properly in scope, with a type given by the context, so if the store fits the context, a variable lookup must succeed and give us a value of the correct type, to boot.

Moreover, if we know $\sigma_0 \;:\; \Gamma$, $\Gamma \vdash p \leftarrow \tau$ and $v : \tau$, then there *must* be a σ_1 such that $\sigma_0 \mid p := v \;\Downarrow\; \sigma_1$ and, moreover, $\sigma_1 \;:\; \Gamma$. Updating a store which fits a context by filling a place of known type with a value of that type must give us a store which fits the same context.

For expressions, we cannot be quite so definitive, because there are still expression like

$$\texttt{while}\ (\texttt{true})\ \{\}$$

which do not terminate.

However, we can say that if $\Gamma \vdash e : \tau$ and $\sigma_0 \;:\; \Gamma$ and $\sigma_0 \mid e \;\Downarrow\; v \mid \sigma_1$, then $v : \tau$ and $\sigma_1 \;:\; \Gamma$. That is, we should have the *type preservation* property. (Think about how to state the necessary result for blocks.)

We should aim for more than preservation, though. When a well typed program fails to evaluate, there should be a very good reason. Here, it should really be true that getting stuck in a loop is the only reason why a well typed program can fail to produce a value. In a small step semantics it is easy to state the *progress* property, namely that every well typed program either has stopped already, or will take a step to another program of the same type.

In a big step semantics, it is more usual to refine judgements with a notion of 'fuel' which is spent by while-loops. Evaluation can stop by running out of fuel, as well as by producing a value. We model this outcome by adding an extra value, $\bot$, pronounced "bottom" in every type:

$$\frac{}{\bot : \tau}$$

Our evaluation judgment forms are now

$$\sigma_0 \mid e \Downarrow_n \sigma_1 \mid v \qquad \sigma_0 \mid es \setminus v_0 \Downarrow_n \sigma_1 \mid v_1$$

where the extra n is the amount of fuel supplied, meaning that each loop body may run at most n times. Crucially, we demand

$$\frac{\sigma_0 \mid e_b \Downarrow_0 \sigma_1 \mid \texttt{true}}{\sigma_0 \mid \texttt{while}\ (e_b)\ e_l \Downarrow_0 \sigma_1 \mid \bot}$$

$$\frac{\sigma_0 \mid e_b \Downarrow_{n+1} \sigma_1 \mid \texttt{true} \qquad \sigma_1 \mid e_l \Downarrow_{n+1} \sigma_2 \mid \star \qquad \sigma_2 \mid \texttt{while}\ (e_b)\ e_l \Downarrow_n \sigma_3 \mid v}{\sigma_0 \mid \texttt{while}\ (e_b)\ e_l \Downarrow_{n+1} \sigma_3 \mid v}$$

The rationale is this: in every rule premise for a subexpression, the fuel stays the same; in every other premise, the fuel decreases. In every case, something is getting smaller, so we cannot keep going for ever. We can decorate all our usual rules this way. We also need rules to check for success and to propagate failure: abandon evaluation at the first sign of $\bot$! E.g.,

$$\frac{\sigma_0 \mid e_b \Downarrow_{n+1} \sigma_1 \mid \bot}{\sigma_0 \mid \texttt{while}\ (e_b)\ e_l \Downarrow_{n+1} \sigma_3 \mid \bot} \qquad \frac{\sigma_0 \mid e_b \Downarrow_{n+1} \sigma_1 \mid \texttt{true} \qquad \sigma_1 \mid e_l \Downarrow_{n+1} \sigma_2 \mid \bot}{\sigma_0 \mid \texttt{while}\ (e_b)\ e_l \Downarrow_{n+1} \sigma_3 \mid \bot}$$

In rules with subevaluations, we must make sure that each does not yield $\bot$ before we do any more work.

$$\frac{}{\sigma_0 \mid \bot \setminus es \Downarrow_n \sigma_0 \mid \bot} \qquad \frac{\sigma_0 \mid e \Downarrow_n \sigma_1 \mid v_1 \quad v_1 \neq \bot \quad \sigma_1 \mid v_1 \setminus es \Downarrow_n \sigma_2 \mid v_2}{\sigma_0 \mid v_1 \setminus e; es \Downarrow_n \sigma_2 \mid v_2}$$

Refining all the rules with fuel and $\bot$ takes effort rather than imagination. We may then prove the following:

- If $\sigma_0 \mid e \Downarrow_n \sigma_1 \mid v$ and $v \neq \bot$, then $\sigma_0 \mid e \Downarrow_{n+1} \sigma_1 \mid v$. (That is, failure is never caused by supplying *more* fuel.)
- If $\Gamma \vdash e : \tau$ and $\sigma_0 : \Gamma$, then for any amount of fuel n, $\sigma_0 \mid e \Downarrow_n \sigma_1 \mid v$ for some $v : \tau$ and $\sigma_1 : \Gamma$. (So, we always get a value of the right type, but it might be $\bot$. Even if it is, the final store when we ran out of fuel still fits the context.)

6 Arrays

We can extend Typed IMP with arrays, by adding a production to the grammar of types

$$\langle type \rangle ::= \ldots \mid \texttt{array}(\langle type \rangle)$$

which means we can nest arrays to any depth.

We need to extend the language of *places* to allow us array lookup

$$\langle place \rangle ::= \ldots \mid \langle place \rangle[\langle exp \rangle]$$

and add a typing rule

$$\frac{\Gamma \vdash p \leftarrow \texttt{array}(\tau) \quad \Gamma \vdash e \leftarrow \texttt{int}}{\Gamma \vdash p[e] \leftarrow \tau}$$

We need to extend the language of expressions with some means to create arrays. One way to do this is

$$\langle exp \rangle ::= \ldots \mid \texttt{array}(\langle exp \rangle)$$

which creates an array whose *default* value in every place is the value of the expression, so the typing rule is

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{array}(e) : \texttt{array}(\tau)}$$

We should further extend our notion of *value* with some means to represent arrays. We can say that an array is given by a function which maps integers to values. For any such function f, we may update it by taking

$$f[i \mapsto v](j) = \begin{cases} v & \text{if } i = j \\ f(j) & \text{if } i \neq j \end{cases}$$

and our $\texttt{array}(e)$ construct kicks us off with the function which gives e's value for every index. Thes function values have array types: sampling one stored value is enough to find out the type of the array's elements.

$$\frac{f(0) : \tau}{f : \texttt{array}(\tau)}$$

However, we are not out of the woods. We now have places which can contain arbitrary expressions, so they now need a notion of value. These are sometimes called 'l-values', meaning values which can stand to the left of $:=$ because they refer to places in the store.

$$\langle \mathit{lvalue} \rangle ::= \langle \mathit{var} \rangle \mid \langle \mathit{lvalue} \rangle [\langle \mathit{integer} \rangle]$$

We need a judgement for evaluating places to l-values, l which can mutate the store.

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

$$\frac{}{\sigma \mid x \Downarrow \sigma \mid x} \qquad \frac{\sigma_0 \mid p \Downarrow \sigma_1 \mid l \quad \sigma_1 \mid e \Downarrow \sigma_2 \mid n}{\sigma_0 \mid p \Downarrow \sigma_2 \mid l[n]}$$

The assignment rule becomes

$$\frac{\sigma_0 \mid p \Downarrow \sigma_1 \mid l \quad \sigma_1 \mid e \Downarrow \sigma_2 \mid v \quad \sigma_2 \mid l := v \Downarrow \sigma_3}{\sigma_0 \mid p := e \Downarrow \sigma_3 \mid \star}$$

because it has to evaluate where to put the thing, as well as what to put there.

Our lookup judgement should now properly look up lvalues instead. Our update judgement needs to update at a given lvalue, too.

$$\frac{\sigma \mid l \Rightarrow v}{\sigma \mid l[n] \Rightarrow v(n)} \qquad \frac{\sigma_0 \mid l \Rightarrow u \quad \sigma_0 \mid l := u[n \mapsto v] \Downarrow \sigma_1}{\sigma_0 \mid l[n] := v \Downarrow \sigma_1}$$

That is to say, to look up an array access, look up the whole array and access it. To update one entry of an array, look up the whole array, then update the whole thing (with one entry changed).

And that's it! We added a feature by asking (and answering)

- What new types do we need?
- What do we need to add to the grammar to make use of those types?
- What new typing rules do we need, for our new constructs?
- What new notions of value do we need for those types?
- How do the new constructs evaluate?