EECS 590
Ivy Operational Semantics

Steven Schaefer
stschaef@umich.edu

December 13th, 2022

1 Introduction

1.1 What is Ivy?

Ivy [2] is a language to aid in the design and verification of systems. Because of its simplicity and convenient interface, it allows one to reason easily about concurrent and distributed systems.

Here, we provide some small step operational semantics of a stripped down version of Ivy. For a first treament of the language semantics, we ignore some constructs that do not (greatly) change the behavior of Ivy programs. Some of the ignored cases include the semantics behind modules, objects, monitors, definitions, importing actions, exporting actions, and some syntactic sugar. Additionally, we can assume that actions are implemented at the same time as they are declared. None of these assumptions changes what a user may accomplish with the language; instead, the same behaviors may be encoded in the language fragment presented. In the future, it would be interesting to give a formal treatment to these parts of the language; however, for now I don't believe they fundamentally affect why people care about Ivy.

Typically, many Ivy programs are presented similar to a specification of a system rather than an executable program. That is, instead of a user providing an explicit sequence of actions in a `.ivy` file, one could imagine a program verfier — like `ivy_check` [1] — generating and testing these action traces. In our presentation of Ivy's semantics, we have these action traces written explicitly. For example, consider a program containing these declarations,

```
action connect(x:client,y:server) = {
    require semaphore(y);
    link(x,y) := true;
    semaphore(y) := false
}

action disconnect(x:client,y:server) = {
    require link(x,y);
    link(x,y) := false;
    semaphore(y) := true
}
```

The program's global state is only affected if we have an explicit trace of actions like

```
call connect(c1, s1); call disconnect(c1, s1); ...
```

Because the invoked actions are explicitly given in this way, we may analyze their effect on program state by evaluating them with an interpreter for our stripped down language.

1.2 Project Contribution

This project attempts to give some formal semantics for analyzing Ivy, which do not seem to be present in the community. To this end, I give formal inference rules for small-step semantics on a stripped down version of the Ivy language. Additionally, for this stripped

down language, I also provide typing rules for expressions in Ivy and rules to check if commands are well-formed.

To sanity check these proposed rules and give some empirical evidence that these semantics are correct, I implemented an Ivy interpreter and type-checker in OCaml.

2 Semantics

In Fig. 1 we find a description of the Ivy-like syntax. Effectively, the expressions behave like terms in first-order logic with the addition of callable actions — potentially with side effects — and non-determinism.

We assume that declarations and assertions are correctly scoped and stored in variable context Γ — which tracks what variables have been declared along with their types — and theory context Θ — which tracks what assertions are valid at a given point in a program. Note, some assertions — like `axiom` — must be true everywhere, while a `require` assertion need only hold as a precondition to an action.

As our writeable locations, we have variables that are explicitly defined in the program — either locally or globally — as well as the values of declared functions. We can effectively interpret functions as arrays indexed by their arguments, and thus we may edit what is stored at those indices. These variables and function values will determine the program's state and are contained in store σ.

2.1 Things Not Considered

Syntactic Sugar There are many instances of Ivy constructs that can be entirely described via other constructs. Here are a couple cases of syntactic sugar that we ignored for this reason. First, commands like

```
if some x:t .  s(x){...}
```

are ignored because these are equivalent to

```
if ∃x:t.s(x){...}
```

In Ivy expressions capitalized, free variables are implicitly universally quantified. We may assume that these have been translated into their equivalent expressions with explicit quantifiers.

In Ivy, $+ : \alpha \times \alpha \to \alpha$ and $<: \alpha \times \alpha \to$ `bool` are defined for all types α. We ignore these, as they are treated like uninterpreted functions. That is, if a user of this toy model of Ivy wanted to replicate the behavior of $+$ in proper Ivy, they could do so by explicitly declaring an add function on the type they're interested in. Similarly, if they wish to have a theory of arithmetic — say on `int`'s — they could declare the proper types and axiomatize them appropriately. While it may be convenient to have these as primitive to the language, we can encode their behavior through our existing constructs.

Time Constraints Due to the short amount of time around the project, there were some features that I would have liked to finish in more detail but could not.

First of all, I had trouble coding up enumerated types in my Ivy interpreter because of a parsing issue. Additionally, I did not have enough time to get callable actions — with optional return types — or the handling of assertions [1] in my interpreter. The lack of callable actions is not greatly detrimental, as a user could do some simple rewriting to dodge this construct. However, the lack of proper assertions in the interpreter does change the semantics of the language greatly. Similarly, I do not handle errors.

Interestingly, these last two features — the invoking of actions and assertions — have a lot

[1] In some of the code examples, there may appear to be assertions. These are handled by the language frontend, but simply map to a `skip` command on the backend.

Expressions

⟨*expr*⟩ :: = ⟨*x*⟩
- | ⟨*f*⟩ (⟨*expr*⟩, . . . , ⟨*expr*⟩)
- | `true`
- | `false`
- | ¬ ⟨*expr*⟩
- | ⟨*expr*⟩ ⊕ ⟨*expr*⟩, ⊕ ∈ {∧, ∨, →, ⇔}
- | ∀ ⟨*var*⟩ . ⟨*expr*⟩
- | ∃ ⟨*var*⟩ . ⟨*expr*⟩
- | `call` ⟨*action*⟩
- | ∗ (non-deterministic choice)

⟨*value*⟩ ::= `true`
- | `false`
- | ⟨*x*⟩
- | ⟨*f*⟩(⟨*value*⟩,..., ⟨*value*⟩)

Commands/Declarations

⟨*declaration*⟩ :: = `var` ⟨*var*⟩ (local variable declaration)
- | `individual` ⟨*var*⟩ (global variable declaration)
- | `type` ⟨*type*⟩
- | `type` ⟨*type*⟩ = {$\tau_1, \ldots, \tau_n$} (enumerated type)
- | `function` ⟨*f*⟩(⟨*var*⟩ : ⟨*type*⟩) : ⟨*type*⟩

⟨*command*⟩ :: = ⟨*x*⟩ := ⟨*expr*⟩
- | $f(\tau_{n_1}, \ldots, \tau_{n_m})$:= ⟨*expr*⟩
- | ⟨*command*⟩; ⟨*command*⟩
- | `if` ⟨*expr*⟩ { ⟨*command*⟩ }
- | `if` ⟨*expr*⟩ { ⟨*command*⟩ } `else` { ⟨*command*⟩ }
- | `for` ⟨*var*⟩ `in` τ { ⟨*command*⟩ }
- | `while` ⟨*expr*⟩ { ⟨*command*⟩ }
- | `skip`

⟨*value_cmd*⟩ ::= `skip`

Assertions

⟨*assertion*⟩ :: = `require` ⟨*expr*⟩
- | `ensure` ⟨*expr*⟩
- | `assume` ⟨*expr*⟩
- | `axiom` ⟨*expr*⟩

Evaluation Contexts

⟨*E*⟩ :: = [•]
- | ⟨*E*⟩ ⊕ ⟨*expr*⟩, ⊕ ∈ {∧, ∨, →, ⇔}
- | ⟨*v*⟩ ⊕ ⟨*E*⟩
- | ¬⟨*E*⟩
- | ⟨*E*⟩ `if` ⟨*expr*⟩ `else` ⟨*expr*⟩
- | ⟨*v*⟩ `if` ⟨*E*⟩ `else` ⟨*expr*⟩
- | ⟨*v*⟩ `if` ⟨*v*⟩ `else` ⟨*E*⟩
- | $x := E$
- | `if` ⟨*E*⟩ `then` { ⟨*command*⟩ }
- | `if` ⟨*E*⟩ `then` { ⟨*command*⟩ } `else` { ⟨*command*⟩ }

⟨*E_cmd*⟩ ::= [•]
- | ⟨*E_cmd*⟩;⟨*command*⟩

Figure 1: Ivy-like Syntax

of overlap semantically. If the precondition to an action is not met and the action is called, then the desired behavior in Ivy is for the action to return with no effect on global state.

2.2 Syntax

In Fig. 1, we find a formal presentation of Ivy syntax. We take $\langle x \rangle$, $\langle f \rangle$ to be syntactic categories containing variable names and function names, respectively. Here, *functions* refer to global state variables. In this setting, it is useful to think of these as akin to arrays indexed by their arguments.

For simplicity, we take all types to be enumerated, even if not stated as such. A priori, the Ivy language that exists out in the world can reason about an unbounded number of elements in its type. However, when used for actual verification purposes, a verifier will usually instantiate these types are particular sizes. Because we wish to implement an interpreter, we require that all types are either enumerated explicitly or are given a size so that they are enumerated implicitly. [2] Likewise, we will consider formulas like $\bigwedge_{y \in \tau} e[x \mapsto y]$ as finitary. That is, because types τ are finite we may explicitly instantiate quantified expressions. When $\tau = \{\tau_1, \ldots, \tau_n\}$,

$$\left(\bigwedge_{y \in \tau} e[x \mapsto y] \right) \Leftrightarrow e[x \mapsto \tau_1] \wedge \cdots \wedge e[x \mapsto \tau_n]$$

Take variable context Γ to contain all the typing information explicitly declared as well as the new information that may be gained from the declared through a type inference procedure like Hindley-Milner. Θ will contain current axioms and assertions, and we must importantly evaluate each expression with respect to the constraints described in Θ. In the setting of the interpreter, we require everything to be explicitly typed. I did not have time to implement type inference on top of the type checking and small step evaluation.

Note that $\Gamma, E[*] \models * : \tau$ means that this occurence of $*$ was inferred to be of type τ in the context of the entire program. Two occurences of $*$ in the same program may inhabit two distinct types, as the evaluation context surrounding each occurence may allow different type inferences.

[2] This implicit enumeration does not appear in the formal syntax described in this paper, but is implemented in the interpreter.

2.3 Theory Context

The theory context Θ is meant to track the set of logical obligations the program state must obey in the current scope. We write $\sigma \models \Theta$ to denote that state σ satisfies all of the constraints laid out in Θ. When evaluating expressions, we also must instantiate the current axioms and conjoin them to the expression. This is not formally expressed, but we can imagine that $\langle expr \rangle$ behaves instead as some sort of pre-expression category that must be instantiated with respect to the axioms of Θ_{ax}.

For example, suppose $\Theta_{\text{ax}} = \{\phi_1, \phi_2\}$. If we wish to evaluate e in the context Θ, then to get the correct value we need to instantiate Θ_{ax} in e. This means instead of evaluating e in our formal semantics, we should be evaluating $e' = \phi_1 \wedge \phi_2 \wedge e$.

In the formal semantics, we comprehend Θ mathematically. That is, if we are checking if $\sigma \models \Theta$, we don't worry about the decision procedure. In general, first order logic is undecidable so this may appear worrying; however, here all of our types are finite so there exists a brute force decision procedure.

To omit a redundant premise across all inference rules in Fig. 4, we add Fig. 2 to ensure that computations only step if they respect the theory context. Use $\Theta_{\text{ax}}, \Theta_{\text{pre/post}}$ to denote the set of `axiom` assertions and the set of `require`/`ensure` assertions, respectively.

I am attempting to avoid overloading notation too much, but might be failing. When applying rules like FAILS-PRE/FAILS-POST the failed `require`/`ensure` assertion in $\Theta_{\text{pre/post}}$ must come from a precondition to an action that is called somewhere in p. There may be a better way to denote this syntactically.

2.4 Evaluation Rules

Note that the evaluation rules do not necessarily preserve state. Expressions can call actions and thus may have side effects, σ is not necessarily equal to σ' even when only evaluating an expression. Further, in a conditional expression, it is the desired behavior that all calls are invoked regardless of if their branch was taken — which is why we see the evalutation to $\langle v \rangle$ `if` $\langle v \rangle$ `else` $\langle v \rangle$ despite these side effects.

We read the declarations chronologically when building Γ. That is, read from beginning of program to the end and only use declarations up until that point. Similarly, build theory context Θ for the appropriate scope — usually an `action` definition or surrounding conditionals. Building these formally is likely a useful exercise, although for the purpose of describing operational semantics we can assume that these are handed to us well-formed.

2.5 Operational Semantics

Most of the rules in Figs. 3 to 5 speak for themselves. Some neat ones to highlight include FORALL-INST and EXISTS-INST, because of the finite instantiation across the types from earlier. NON-DET is also interesting, as the non-determinism is baked into the premise applying to multiple elements $t : \tau$.

3 Interpreter

I will zip and upload the code if you're interested. This was modeled after the previous homeworks.

- `ast.ml` contains the abstract syntax description of Ivy.
- `lexer.ml` describes how the lexer should split the program into tokens.
- `parser.mly` denotes how the parser gives meaning to the tokens.
- `repl.ml` contains the main read-evaluate-print loop.
- `ivy.ml` contains the interpreter and type checker for Ivy.

$$\frac{\Gamma \models \langle p, \sigma\rangle \to \langle p', \sigma'\rangle \qquad \sigma \models \Theta \qquad \sigma' \models \Theta}{\Gamma, \Theta \models \langle p, \sigma\rangle \to \langle p', \sigma'\rangle}\ \textbf{RESPECTS-AXIOMS}$$

$$\frac{\Gamma \models \langle p, \sigma\rangle \to \langle p', \sigma'\rangle \qquad \sigma \not\models \Theta_{\text{ax}}}{\Gamma, \Theta \models \langle p, \sigma\rangle \to \texttt{error}}\ \textbf{FAILS-AXIOMS-L}$$

$$\frac{\Gamma \models \langle p, \sigma\rangle \to \langle p', \sigma'\rangle \qquad \sigma \models \Theta_{\text{ax}} \qquad \sigma \not\models \Theta_{\text{pre/post}}}{\Gamma, \Theta \models \langle p, \sigma\rangle \to \langle \texttt{skip}, \sigma\rangle}\ \textbf{FAILS-PRE}$$

$$\frac{\Gamma \models \langle p, \sigma\rangle \to \langle p', \sigma'\rangle \qquad \sigma \models \Theta_{\text{ax}} \qquad \sigma' \not\models \Theta_{\text{pre/post}}}{\Gamma, \Theta \models \langle p, \sigma\rangle \to \langle \texttt{skip}, \sigma\rangle}\ \textbf{FAILS-POST}$$

Figure 2: Assertion Semantics

$$\frac{\Gamma \models \langle e, \sigma\rangle \to \langle e', \sigma'\rangle}{\Gamma \models \langle E[e], \sigma\rangle \to \langle E[e'], \sigma'\rangle}\ \textbf{EVAL} \qquad \frac{\oplus \in \{\wedge, \vee, \to, \Leftrightarrow\}}{\Gamma \models \langle v_1 \oplus v_2, \sigma\rangle \to \langle (\sigma(v_1) \oplus_{\texttt{bool}} \sigma(v_2)), \sigma\rangle}\ \textbf{BOOL-BINOP}$$

$$\frac{}{\Gamma \models \langle \neg v, \sigma\rangle \to \langle \neg_{\texttt{bool}} \sigma(v), \sigma\rangle}\ \textbf{NEG} \qquad \frac{}{\Gamma \models \langle \neg \texttt{true}, \sigma\rangle \to \langle \texttt{false}, \sigma\rangle}\ \textbf{T-LIT}$$

$$\frac{}{\Gamma \models \langle \neg \texttt{false}, \sigma\rangle \to \langle \texttt{true}, \sigma\rangle}\ \textbf{F-LIT} \qquad \frac{\Gamma \models x : \tau}{\Gamma \models \langle \forall x.e, \sigma\rangle \to \langle \bigwedge_{y \in \tau} e[x \mapsto y], \sigma\rangle}\ \textbf{FORALL-INST}$$

$$\frac{\Gamma \models x : \tau}{\Gamma \models \langle \exists x.e, \sigma\rangle \to \langle \bigvee_{y \in \tau} e[x \mapsto y], \sigma\rangle}\ \textbf{EXISTS-INST} \qquad \frac{}{\Gamma \models \langle v_1 \ \texttt{if true else}\ v_3, \sigma\rangle \to \langle \sigma(v_1), \sigma\rangle}\ \textbf{COND-T}$$

$$\frac{}{\Gamma \models \langle v_1 \ \texttt{if false else}\ v_3, \sigma\rangle \to \langle \sigma(v_2), \sigma\rangle}\ \textbf{COND-F} \qquad \frac{\Gamma, E[*] \models * : \tau \qquad \Gamma \models t : \tau}{\Gamma \models \langle *, \sigma\rangle \to \langle t, \sigma\rangle}\ \textbf{NON-DET}$$

Figure 3: Expression Semantics

$$\frac{}{\Gamma \models \langle x := v, \sigma\rangle \to \langle \texttt{skip}, \sigma[x \mapsto v]\rangle}\ \textbf{ASSIGN} \qquad \frac{}{\Gamma \models \langle \texttt{skip}; c_2, \sigma\rangle \to \langle c_2, \sigma\rangle}\ \textbf{SEQ}$$

$$\frac{}{\Gamma \models \langle \texttt{if true}\ \{c\}, \sigma\rangle \to \langle c, \sigma\rangle}\ \textbf{IF-T} \qquad \frac{}{\Gamma \models \langle \texttt{if false}\ \{c\}, \sigma\rangle \to \langle \texttt{skip}, \sigma\rangle}\ \textbf{IF-F}$$

$$\frac{}{\Gamma \models \langle \texttt{if true}\ \{c_1\}\ \texttt{else}\ \{c_2\}, \sigma\rangle \to \langle c_1, \sigma\rangle}\ \textbf{IF-ELSE-T} \qquad \frac{}{\Gamma \models \langle \texttt{if false}\ \{c_1\}\ \texttt{else}\ \{c_2\}, \sigma\rangle \to \langle c_2, \sigma\rangle}\ \textbf{IF-ELSE-F}$$

$$\frac{\Gamma \models \tau = \{y_i\}_{i=1}^n}{\Gamma \models \langle \texttt{for}\ x\ \texttt{in}\ \tau\ \{c\}, \sigma\rangle \to \langle c[x \mapsto y_1]; \cdots; c[x \mapsto y_n], \sigma\rangle}\ \textbf{FOR} \qquad \frac{}{\Gamma \models \langle \texttt{while}\ e\ \{c\}, \sigma\rangle \to \langle \texttt{if}\ e\ \{c\}; \texttt{while}\ e\ \{c\}, \sigma\rangle}\ \textbf{WHILE}$$

Figure 4: Command Semantics

$$\frac{\Gamma \vdash b_1 : \texttt{bool} \qquad \Gamma \vdash b_2 : \texttt{bool} \qquad \oplus \in \{\wedge, \vee, \rightarrow, \Leftrightarrow\}}{\Gamma \vdash b_1 \oplus b_2 : \texttt{bool}} \textbf{ BINOP-INTRO} \qquad \frac{\Gamma \vdash b : \texttt{bool}}{\Gamma \vdash \neg b : \texttt{bool}} \textbf{ NEG-INTRO}$$

$$\frac{\beta \in \{\texttt{true}, \texttt{false}\}}{\Gamma \vdash \beta : \texttt{bool}} \textbf{ BOOL-LIT} \qquad \frac{\Gamma \vdash e[x \mapsto y_i] : \texttt{bool} \qquad \tau = \{y_i\}_{i=1}^n}{\Gamma \vdash \forall (x : \tau).e : \texttt{bool}} \textbf{ FORALL-INTRO}$$

$$\frac{\Gamma \vdash e[x \mapsto y_i] : \texttt{bool} \qquad \tau = \{y_i\}_{i=1}^n}{\Gamma \vdash \exists (x : \tau).e : \texttt{bool}} \textbf{ EXISTS-INTRO} \qquad \frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash b : \texttt{bool}}{\Gamma \vdash e_1 \texttt{ if } b \texttt{ else } e_2 : \tau} \textbf{ COND-INTRO}$$

$$\frac{* \text{ inferred to be of type } \tau \text{ in } E}{\Gamma, E[*] \vdash * : \tau} \textbf{ NON-DET-INFER}$$

Figure 5: Typing Rules

$$\frac{x \in \Gamma \qquad \Gamma \vdash x : \tau \qquad \Gamma \vdash e : \tau \qquad \tau \in \Gamma}{\Gamma \vdash x := e : 1} \textbf{ T-ASSN} \qquad \frac{\Gamma \vdash c_1 : 1 \qquad \Gamma, c_1 \vdash c_2 : 1}{\Gamma \vdash c_1; c_2 : 1} \textbf{ T-SEQ}$$

$$\frac{\Gamma \vdash e : \texttt{bool} \qquad \Gamma \vdash c : 1}{\Gamma \vdash \texttt{if } e \, \{c\} : 1} \textbf{ T-IF} \qquad \frac{\Gamma \vdash e : \texttt{bool} \qquad \Gamma \vdash c_1 : 1 \qquad \Gamma \vdash c_2 : 1}{\Gamma \vdash \texttt{if } \ e \, \{c_1\} \texttt{ else } \{c_2\} : 1} \textbf{ T-IF-ELSE} \qquad \frac{\Gamma, x : \tau \vdash c : 1 \qquad \tau \in \Gamma}{\Gamma \vdash \texttt{for } x \texttt{ in } \tau \, \{c\} : 1} \textbf{ T-FOR}$$

$$\frac{\Gamma \vdash e : \texttt{bool} \qquad \Gamma \vdash c : 1}{\Gamma \vdash \texttt{while } e \, \{c\} : 1} \textbf{ T-WHILE} \qquad \frac{x \notin \Gamma \qquad \text{type of } x \text{ inferrable}}{\Gamma \vdash \texttt{var } x : 1} \textbf{ T-INFER-DEC} \qquad \frac{x \notin \Gamma \qquad \tau \in \Gamma}{\Gamma \vdash (\texttt{var } x : \tau) : 1} \textbf{ T-LVAR-DEC}$$

$$\frac{f \notin \Gamma \qquad \tau_i \in \Gamma}{\Gamma \vdash (\texttt{function } f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau_{n+1}) : 1} \textbf{ T-FUN-DEC}$$

$$\frac{\tau \in \Gamma \qquad x \notin \Gamma}{\Gamma \vdash (\texttt{individual } x : \tau)} \textbf{ T-GVAR-DEC}$$

Figure 6: Well-formed commands

Much of the implementation in `ivy.ml` followed from a combination of the 590 coursework, just with more declarations and stores to carry around.

4 Conclusion

Overall, this project takes a step toward useful Ivy semantics, and it was fun to do; however, there are still some gaps that I believe need to be addressed before calling this work complete or hoping to publish it somewhere.

1. Go back and prove that any concepts deliberately ignored were indeed redundant, and thus not needed for this formal analysis
2. Handle assertions and actions in greater detail. In particular, get these working in my interpreter. These are huge part of Ivy and are crucial to a complete semantics of it. I initally considered describing the actions via a translation in something resembling the simply-typed lambda calculus with a void type added. This way I could handle actions with or without return types. The overhead for this seemed too high and I prioritized getting a minimal working interpreter. Perhaps this wasn't the right choice, but in the future it is very feasible to finish this part up.
3. Escalate above empirical evidence on my hacky interpreter, and instead formalize these semantics — say, in Coq.

References

[1] Ivy Command Reference. `http://microsoft.github.io/ivy/commands.html`, 2022.

[2] The Ivy Language. `http://microsoft.github.io/ivy/language.html`, 2022.