A simple imperative language

Massimo Merro

4 October 2017

The language *While*

We provide the syntax of a simple imperative language by means of a BNF grammar containing:

$$
\begin{array}{llll}
\textit{Booleans} & b \in \mathbb{B} & = & \{\textit{true}, \textit{false}\} \\
\textit{Integers} & n \in \mathbb{N} & = & \{\ldots, -1, 0, 1, \ldots\} \\
\textit{Locations} & l \in \mathbb{L} & = & \{\mathrm{l}, \mathrm{l}_0, \mathrm{l}_1, \mathrm{l}_2, \ldots\} \\
\textit{Operations} & op & ::= & + \mid \geq \\
\textit{Expressions} & e \in \textit{Exp} & ::= & n \mid b \mid e \ op \ e \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \\
 & & & \mid l := e \mid\ !l \mid \textit{skip} \mid e; e \\
 & & & \mid \mathsf{while}\ e\ \mathsf{do}\ e
\end{array}
$$

Please, note the following:

- we consider abstract syntax; so our grammar defines syntactic trees
- integers are unbounded
- we have abstract locations; thus $!l$ means "the integer currently stored at location l" (for simplicity, we store only integers)
- untyped language, so have nonsensical expressions like $2 \geq \mathit{true}$
- don't have expression/command distinctions
- doesn't really matter what basic operations we have
- distinguish metavariables b, n, l, e, op from program locations $\mathrm{l}, \mathrm{l}_0, \ldots$

Some intuition

- assignment, "$l := e$" evaluates e and then stores the result in the location l
- conditional, taking a boolean and two expressions and yielding a expression "if e then e_1 else e_2"
- sequential composition, written "$e_1; e_2$", takes two commands (the semicolon here is an *operator* joining two commands into one and not just a piece of punctuation at the end of a command)
- do nothing, denoted by the constant "*skip*"
- loop constructor, which takes a boolean and a command and yields a command, written "while e do e_1".

Example program

A program in our language is given by a non-empty sequential composition of expressions $e_1; \ldots; e_n$

$$
\begin{array}{l}
\mathrm{l}_2 := 1; \\
\mathrm{l}_3 := 0; \\
\text{while } \neg(!\mathrm{l}_1 = !\mathrm{l}_2) \text{ do} \\
\qquad \mathrm{l}_2 := !\mathrm{l}_2 + 1; \\
\qquad \mathrm{l}_3 := !\mathrm{l}_3 + 1; \\
\mathrm{l}_1 := !\mathrm{l}_3
\end{array}
$$

How do we describe the behaviour of these programs?

How can we prescribe how these program should be executed?

Evaluating expressions

Value of expressions depend on current values in locations

- $!\mathrm{l}_1 + !\mathrm{l}_2 - 1$

In this case, the value depends on current values at locations l_1 and l_2.

Values stored at locations changes as program are executed

So, our operational semantics should take into considerations those changes!

- How do we evalute an expression $!l$?
- or what about an assignment $l := e$?

We need some more information about the state of the machine's *memory*.

Partial functions

$$f \ : \ A \rightharpoonup B$$

Meaning:

f returns an element of B for *some* elements of A

Convention:

- dom(f) is the set of elements in the domain of f, formally dom(f) $= \{a \in A : \exists b \in B \; s.t. \; f(a) = b\}$
- ran(f) is the set of elements in in the range of f, formally ran(f) $= \{b \in B : \exists a \in A \; s.t. \; f(a) = b\}$

So, $f(a)$ may not be defined for some a in A, that's why it's called partial! Furthermore, f could be undefined for all elements in A, i.e. a partial function can be empty, just $\{\}$.

Store

- In our language, Store is a set of *finite partial functions* from locations to integers
$$s : \mathbb{L} \rightharpoonup \mathbb{Z}$$
- For example : $\{\mathrm{l}_1 \mapsto 3, \mathrm{l}_2 \mapsto 6, \mathrm{l}_3 \mapsto 7\}$
- Updating: The store $s[\mathrm{l} \mapsto n]$ is defined by
$$s[\mathrm{l} \mapsto n](\mathrm{l}') = \begin{cases} n & \text{if } \mathrm{l} = \mathrm{l}' \\ s(\mathrm{l}') & \text{otherwise} \end{cases}$$
- Behaviour of our programs is relative to a *store*
- The store changes as the execution of a program proceeds

Transition systems

Operational semantics in terms of a transition system.

A transition system consists of

- a set *Config*, of configurations, and
- a binary relation $\rightarrow \subseteq \mathit{Config} \times \mathit{Config}$.

In particular,

- the elements of *Config* are often called *configurations* or *states*
- the relation $\rightarrow$ is called the *transition* or *reduction* relation
- we adopt an infix notation, so $c \rightarrow c'$ should be read as "configuration c can make a transition to the configuration c'
- complete execution of a program transforms an initial state into a terminal state.

A transition system is like an NFA$^{\epsilon}$ with an empty alphabet, except

- it can have infinitely many states
- we don't specify a start state or accepting states.

Operational semantics for our imperative language

Configurations are pairs $\langle e, s \rangle$ of an expression e and a store s. Our transition relation will have the form:

Judgements:

$$\langle e, s \rangle \rightarrow \langle e', s' \rangle$$

Meaning:

- starting from store s
- when evaluating expression e

one *step* of computation leads to

- store s'
- with expression e' remaining to be evalueted.

What is a step?

It depends...

What is a step?

Transitions are single computation steps. For example we will have:

$$\begin{array}{ll} \twoheadrightarrow & \langle \mathrm{l} := 2 + !\mathrm{l},\ \{\mathrm{l} \mapsto 3\}\rangle \\ \twoheadrightarrow & \langle \mathrm{l} := 2 + 3,\ \{\mathrm{l} \mapsto 3\}\rangle \\ \twoheadrightarrow & \langle \mathrm{l} := 5,\ \{\mathrm{l} \mapsto 3\}\rangle \\ \twoheadrightarrow & \langle \mathit{skip},\ \{\mathrm{l} \mapsto 5\}\rangle \\ \not\twoheadrightarrow & \end{array}$$

Here, $\not\twoheadrightarrow$ is a unary operator on *Config* defined by $c \not\twoheadrightarrow$ iff $\neg\exists c'.c \twoheadrightarrow c'$.
We want to keep on until we get to a value v, an expression in

$$\mathbb{V} = \mathbb{B} \cup \mathbb{Z} \cup \{\mathit{skip}\}$$

Say $\langle e, s\rangle$ is stuck or in deadlock if e is not a value and $\langle e, s\rangle \not\twoheadrightarrow$.
For example, $3 + \mathit{false}$ is stuck!

Transition system: basic operations

$$(\text{op} +)\ \frac{-}{\langle n_1 + n_2,\ s\rangle \twoheadrightarrow \langle n,\ s\rangle}\ n = \text{add}(n_1, n_2)$$

$$(\text{op} \geq)\ \frac{-}{\langle n_1 \geq n_2,\ s\rangle \twoheadrightarrow \langle b,\ s\rangle}\ b = \text{geq}(n_1, n_2)$$

$$(\text{op1})\ \frac{\langle e_1,\ s\rangle \twoheadrightarrow \langle e_1',\ s'\rangle}{\langle \mathrm{e}_1 \text{ op } \mathrm{e}_2,\ \mathrm{s}\rangle \twoheadrightarrow \langle \mathrm{e}_1' \text{ op } \mathrm{e}_2,\ \mathrm{s}'\rangle}$$

$$(\text{op2})\ \frac{\langle e_2,\ s\rangle \twoheadrightarrow \langle e_2',\ s'\rangle}{\langle v\ op\ e_2,\ \rangle \twoheadrightarrow \langle v\ op\ e_2',\ s'\rangle}$$

Observe that none of these transition rules introduces changes in the store.

Example

Suppose we want to find the sequence of transitions starting from the configuration $\langle (3+4)+(7+8),\, \emptyset\rangle$. Then,

$$(\text{op1})\ \frac{(\text{op }+)\ \dfrac{-}{\langle 3+4,\, \emptyset\rangle \rightarrow \langle 7,\, \emptyset\rangle}}{\langle (3+4)+(7+8),\, \emptyset\rangle \rightarrow \langle 7+(7+8),\, \emptyset\rangle}$$

$$(\text{op2})\ \frac{(\text{op }+)\ \dfrac{-}{\langle 7+8,\, \emptyset\rangle \rightarrow \langle 15,\, \emptyset\rangle}}{\langle 7+(7+8),\, \emptyset\rangle \rightarrow \langle 7+15,\, \emptyset\rangle}$$

$$(\text{op }+)\ \frac{-}{\langle 7+15,\, \emptyset\rangle \rightarrow \langle 22,\, \emptyset\rangle}$$

So, in three computation steps, $\langle (3+4)+(7+8),\, \emptyset\rangle \rightarrow\rightarrow\rightarrow \langle 22,\, \emptyset\rangle$.

Transition system: Dereferencing

What is the result of the evaluation of an expression $!l$ in a store s?

Inference rule:

$$(\text{deref}) \; \frac{-}{\langle !l,\, s\rangle \rightarrow \langle n,\, s\rangle} \quad \mathit{if}\ l \in \mathrm{dom}(s)\ \mathit{and}\ s(l) = n$$

Transition rules: Assignment

How to execute one step of command $l := e$, relative to a store s?

Intuition:

- Evaluate e relative to store s
- Update store s with resulting value

Inference rules:

$$(\text{assign1}) \ \frac{-}{\langle l := n, s\rangle \twoheadrightarrow \langle skip, s[l \mapsto n]\rangle} \quad \text{if } l \in \text{dom}(s)$$

$$(\text{assign2}) \ \frac{\langle e, s\rangle \twoheadrightarrow \langle e', s'\rangle}{\langle l := e, s\rangle \twoheadrightarrow \langle l := e', s'\rangle}$$

Transitions system: Conditional

How to execute one step of (if e then e_1 else e_2) relative to a store s?

Intuition:

- Evaluate e relative to store s
- if *true* start evaluating e_1
- if *false* start evaluating e_2

Inference rules:

$$(\text{If\_tt}) \ \frac{-}{\langle \text{if true then } e_1 \text{ else } e_2, s\rangle \twoheadrightarrow \langle e_1, s\rangle}$$

$$(\text{If\_ff}) \ \frac{-}{\langle \text{if false then } e_1 \text{ else } e_2, s\rangle \twoheadrightarrow \langle e_2, s\rangle}$$

$$(\text{If}) \ \frac{\langle e,\ s\rangle \twoheadrightarrow \langle e',\ s'\rangle}{\langle \text{if } e \text{ then } e_1 \text{ else } e_2,\ s\rangle \twoheadrightarrow \langle \text{if } e' \text{ then } e_1 \text{ else } e_2,\ s'\rangle}$$

Transition system: Sequential computation

How to execute one step of $(e_1; e_2)$ relative to store s?

Intuition:

- Execute one step of e_1 relative to state s
- If e_1 has terminated start executing e_2

skip indicates termination.

Inference rules:

$$(\text{Seq}) \quad \frac{\langle e_1,\ s\rangle \rightarrow \langle e_1',\ s'\rangle}{\langle e_1; e_2,\ s\rangle \rightarrow \langle e_1'; e_2,\ s'\rangle}$$

$$(\text{Seq.Skip}) \quad \frac{-}{\langle \mathit{skip}; e_2,\ s\rangle \rightarrow \langle e_2,\ s\rangle}$$

Transitions system: While

How to execute one step of while e do e_1 relative to store s?

Intuition:

- Evaluate e relative to s
- If *false* then terminate
- if *true* then execute one step of e_1, etc...

Inference rule:

$$(\text{While}) \ \frac{-}{\langle \text{while } e \text{ do } e_1,\ s\rangle \twoheadrightarrow \langle \text{if } e \text{ then } (e_1;\ \text{while } e \text{ do } e_1) \text{ else skip},\ s\rangle}$$

This is rewriting rule also called "unwinding", as it unfolds the while loop once: the semantics of while is given in terms of conditional and sequential composition.

Running programs

To run program P starting from a store s:

Find store s' such that

$$\langle P, s \rangle \twoheadrightarrow^* \langle v, s' \rangle$$

for $v \in \mathbb{V} = \mathbb{B} \cup \mathbb{Z} \cup \{skip\}$.
Configurations of the form $\langle v, s \rangle$ are said to be **terminal**.
Here, $\twoheadrightarrow^*$ denotes the reflexive and transitive closure of the reduction relation $\twoheadrightarrow$.

Example:

See McGusker notes at section 4.1.1.

Language properties

A number of interesting properties on the behaviour of programs:

Theorem 1 (Strong normalisation)

For every store s and every program P there exists some store s' such that $\langle P, s \rangle \twoheadrightarrow^* \langle v, s' \rangle$, with $\langle v, s \ \rangle$

Theorem 2 (Determinacy)

If $\langle e, s \rangle \twoheadrightarrow \langle e_1, s_1 \rangle$ and $\langle e, s \rangle \twoheadrightarrow \langle e_2, s_2 \rangle$ then $\langle e_1, s_1 \rangle = \langle e_2, s_2 \rangle$.

Do these properties hold in our language?
How can we prove them?

The meaning/semantics of programs

Let us consider again the the fragment of code seen at the beginning of this lecture:

$$
\begin{array}{l}
\mathrm{l}_2 := 1; \\
\mathrm{l}_3 := 0; \\
\text{while } \neg(!\mathrm{l}_1 = !\mathrm{l}_2) \text{ do} \\
\quad\quad \mathrm{l}_2 := !\mathrm{l}_2 + 1; \\
\quad\quad \mathrm{l}_3 := !\mathrm{l}_3 + 1; \\
\mathrm{l}_1 := !\mathrm{l}_3
\end{array}
$$

What does this program really do?

- Any program should transform an initial state into a terminal state
- But, for some initial states there may be no terminal state.

A semantic interpretation function

We can use our operational semantics to provide a formal semantics to the above program. Let

$$[\![-]\!] \, : \, Exp \rightarrow (Store \rightharpoonup Store)$$

where, given an arbitrary expression e, $[\![e]\!]$ is a partial function transforming an initial store s into a terminal store s'

Definition:

$$[\![e]\!](s) = \begin{cases} s' & \text{if } \langle e, s \rangle \twoheadrightarrow^* \langle v, s' \rangle \\ \text{undefined} & \text{otherwise} \end{cases}$$

Determinacy ensures that the function $[\![-]\!]$ is properly defined.

Application

So, if P is the program mentioned before:

$$
\begin{array}{l}
\mathrm{l}_2 := 1; \\
\mathrm{l}_3 := 0; \\
\text{while } \neg(!\mathrm{l}_1 = !\mathrm{l}_2) \text{ do} \\
\quad \mathrm{l}_2 := !\mathrm{l}_2 + 1; \\
\quad \mathrm{l}_3 := !\mathrm{l}_3 + 1; \\
\mathrm{l}_1 := !\mathrm{l}_3
\end{array}
$$

We can fully describe its behavior as follows:

$$
[\![P]\!](s)(l) = \begin{cases} s(\mathrm{l}_1) - 1 & \text{if } l \in \{\mathrm{l}_1, \mathrm{l}_3\} \text{ and } s(\mathrm{l}_1) > 0 \\ s(\mathrm{l}_1) & \text{if } l = \mathrm{l}_2 \text{ and } s(\mathrm{l}_1) > 0 \\ s(l) & \text{if } l \notin \{\mathrm{l}_1, \mathrm{l}_2, \mathrm{l}_3\} \text{ and } s(\mathrm{l}_1) > 0 \end{cases}
$$

Language design 1. Order of evaluation

For (e_1 op e_2) the rules of our operational semantics say that e_1 must be fully reduced to a value before we start reducing e_2. This evaluation strategy is called left-to-right. For example,

$$\langle (l := 1; 0) + (l := 2; 0),\ \{l \mapsto 0\}\rangle \ \rightarrow^5\ \langle 0, \{l \mapsto 2\}\rangle$$

Another possibility is to follow a right-to-left strategy by replacing rules (op1) and (op2) by

$$(\text{op1b})\ \frac{\langle e_2,\ s\rangle \rightarrow \langle e_2',\ s'\rangle}{\langle e_1 + e_2,\ s\rangle \rightarrow \langle e_1 + e_2',\ s'\rangle} \qquad (\text{op2b})\ \frac{\langle e_1,\ s\rangle \rightarrow \langle e_1',\ s'\rangle}{\langle e_1 + v,\ \rangle \rightarrow \langle e_1' + v,\ s'\rangle}$$

In a right-to-left evaluation strategy:

$$\langle (l := 1; 0) + (l := 2; 0),\ \{l \mapsto 0\}\rangle \ \rightarrow^5\ \langle 0, \{l \mapsto 1\}\rangle$$

If you allow both strategies in your semantics you loose Determinacy!

Language design 2. Assignment results

$$\text{(assign1)} \quad \frac{-}{\langle l := n, s\rangle \rightarrow \langle skip, s[l \mapsto n]\rangle} \quad \textit{if } l \in \text{dom}(s)$$

$$\text{(Seq.Skip)} \quad \frac{-}{\langle skip; e_2,\ s\rangle \rightarrow \langle e_2,\ s\rangle}$$

So

$$\langle(\mathrm{l} := 1; \mathrm{l} := 2),\ \{\mathrm{l} \mapsto 0\}\rangle \ \rightarrow^* \ \langle skip,\ \{\mathrm{l} \mapsto 2\}\rangle$$

However, in certain languages assignments result in expressions:

$$\text{(assign1b)} \quad \frac{-}{\langle l := n, s\rangle \rightarrow \langle n,\ s[l \mapsto n]\rangle} \quad \textit{if } l \in \text{dom}(s)$$

$$\text{(Seq.Skipb)} \quad \frac{-}{\langle v; e_2,\ s\rangle \rightarrow \langle e_2,\ s\rangle}$$

And

$$\langle(\mathrm{l} := 1; \mathrm{l} := 2),\ \{\mathrm{l} \mapsto 0\}\rangle \ \rightarrow^* \ \langle 2,\ \{\mathrm{l} \mapsto 2\}\rangle.$$

Language design 3. Store initialisation

Recall that

$$\text{(deref)}\ \frac{-}{\langle !l,\ s\rangle \twoheadrightarrow \langle n,\ s\rangle} \quad \textit{if } l \in \mathrm{dom}(s) \textit{ and } s(l) = n$$

$$\text{(assign1)}\ \frac{-}{\langle l := n, s\rangle \twoheadrightarrow \langle \mathit{skip}, s[l \mapsto n]\rangle} \quad \textit{if } l \in \mathrm{dom}(s)$$

Both require $l \in \mathrm{dom}(s)$, otherwise the expressions are stuck.
Instead, we could

1. implicitly initialise all locations to 0, or
2. allow assignment to an $l \notin \mathrm{dom}(s)$ to initialise that l.

Language design 4. Storable values

Recall stores s are finite partial functions from $\mathbb{L}$ to $\mathbb{Z}$, with rules:

$$\text{(deref)}\ \frac{-}{\langle !l,\ s\rangle \rightarrow \langle n, s\rangle} \quad \textit{if } l \in \mathrm{dom}(s) \textit{ and } s(l) = n$$

$$\text{(assign1)}\ \frac{-}{\langle l := n, s\rangle \rightarrow \langle \textit{skip},\ s[l \mapsto n]\rangle} \quad \textit{if } l \in \mathrm{dom}(s)$$

- We can store only integers: $\langle \mathrm{l} := \textit{true},\ s\rangle$ is stuck! (we will introduce a type system to rule out programs that could reach a stuck expression)
- Why not allow storage af any value? of locations? of programs?
- Notice also that store is statically defined
- Later on we will consider programs that can create new locations.

Expressiveness

Is our language expressive enough to write interesting programs?

- **yes**: it's Turing-powerful (try coding an arbitrary register machine in it)
- **no**: there is no support for features like functions, branching, objects, etc...

Is our language *too expressive* (i.e. can we write too many program in it)?

- **yes**: We would like to forbid programs like "$3 + true$" as early as possible, rather than let the program get stuck or give a runtime error. We'll do that by means of a type system.

Type systems

used for

- describing when programs make sense
- preventing certain kinds of errors
- structuring programs
- guiding language design
- providing information to compiler optimisers
- enforcing security properties
- etc etc...
- .... even to allow only polynomial-time computations.

In our small language, ideally, well-typed programs don't get stuck!

Type systems more formally

We will define a ternary relation

$$\Gamma \vdash e : T$$

read as "expression e has type T under assumptions Γ on the types of locations that may occur in e".

For example, according to the definition (coming up...):

$$\begin{array}{lcllll} \{\} & \vdash & \text{if true then 2 else } 3+4 & : & \text{int} & \\ l_1 : \text{intref} & \vdash & \text{if } !l_1 \geq 3 \text{ then } !l_1 \text{ else } 3 & : & \text{int} & \\ \{\} & \nvdash & 3 + \text{false} & : & T & \text{for any } T \\ \{\} & \nvdash & \text{if } \mathit{true} \text{ then } 3 \text{ else } \mathit{false} & : & T & \text{for any } T \end{array}$$

Note that the last program is ill-typed despite the fact that when you execute it you'll always get an int: type systems define approximations to the behaviour of programs, often quite crude!

However, it has to be so! We generally would like them to be decidable, so that compilation is guaranteed to terminate!!!

Types for the language While

Types of expressions:

$$T \quad ::= \quad \mathsf{int} \quad \big| \quad \mathsf{bool} \quad \big| \quad \mathsf{unit}$$

Types of locations:

$$T_{loc} \quad ::= \quad \mathsf{intref}$$

- Write T and $\mathrm{T}_{\mathrm{loc}}$ for the set of all terms of these grammars , ie $\mathrm{T} = \{\mathsf{int}, \mathsf{bool}, \mathsf{unit}\}$ and $\mathrm{T}_{\mathrm{loc}} = \{\mathsf{intref}\}$
- Let Γ range over $\mathrm{TypeEnv}$, the set of partial functions from $\mathbb{L}$ to $\mathrm{T}_{\mathrm{loc}}$
- Notations: write a Γ as $\mathrm{l}_1 : \mathsf{intref}, \ldots, \mathrm{l}_k : \mathsf{intref}$, instead of $\{\mathrm{l}_1 \mapsto \mathsf{intref}, \ldots, \mathrm{l}_k \mapsto \mathsf{intref}\}$
- For now, there is only one type in $\mathrm{T}_{\mathrm{loc}}$, so a Γ can be thought of as just a set of locations (later, $\mathrm{T}_{\mathrm{loc}}$ will be more interesting).

Defining the type judgement "$\Gamma \vdash e : T$" (1 of 3)

$$\text{(int)} \quad \frac{-}{\Gamma \vdash n : \mathsf{int}} \quad \text{for } n \in \mathbb{Z}$$

$$\text{(bool)} \quad \frac{-}{\Gamma \vdash b : \mathsf{bool}} \quad \text{for } b \in \{\mathit{true}, \mathit{false}\}$$

$$\text{(op }+\text{)} \quad \frac{\Gamma \vdash e_1 : \mathsf{int} \quad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1 + e_2 : \mathsf{int}} \qquad \text{(op }\geq\text{)} \quad \frac{\Gamma \vdash e_1 : \mathsf{int} \quad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1 \geq e_2 : \mathsf{bool}}$$

$$\text{(if)} \quad \frac{\Gamma : e_1 : \mathsf{bool} \quad \Gamma \vdash e_2 : T \quad \Gamma \vdash e_3 : T}{\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : T}$$

How to use it!

To show that

$$\{\} \vdash \text{if } \mathit{false} \text{ then } 2 \text{ else } 3 + 4 : \text{int}$$

we can give a type derivation like this:

$$\text{(if)}\ \dfrac{\text{(bool)}\ \dfrac{-}{\{\} \vdash \mathit{false} : \text{bool}} \quad \text{(int)}\ \dfrac{-}{\{\} \vdash 2 : \text{int}} \quad \nabla}{\{\} \vdash \text{if } \mathit{false} \text{ then } 2 \text{ else } 3 + 4 : \text{int}}$$

where ∇ is

$$\text{(op +)}\ \dfrac{\text{(int)}\ \dfrac{-}{\{\} \vdash 3 : \text{int}} \quad \text{(int)}\ \dfrac{-}{\{\} \vdash 4 : \text{int}}}{\{\} \vdash 3 + 4 : \text{int}}$$

Defining the type judgement "$\Gamma \vdash e : T$" (2 of 3)

$$\text{(assign)}\ \frac{\Gamma \vdash e : \mathsf{int}}{\Gamma \vdash l{:=}e : \mathsf{unit}}\ \textit{if}\ \Gamma(l) = \mathsf{intref}$$

$$\text{(deref)}\ \frac{-}{\Gamma \vdash\ !l : \mathsf{int}}\ \textit{if}\ \Gamma(l) = \mathsf{intref}$$

Defining the type judgement "$\Gamma \vdash e : T$" (3 of 3)

$$(\text{skip}) \quad \frac{-}{\Gamma \vdash \mathit{skip} : \text{unit}}$$

$$(\text{seq}) \quad \frac{\Gamma \vdash e_1 : \text{unit} \quad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1; e_2 : T}$$

Here, we are making an implicit, precise choice about the semantics of $e_1; e_2$. Can you see it?

$$(\text{while}) \quad \frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{unit}}{\Gamma \vdash \text{while } e_1 \text{ do } e_2 : \text{unit}}$$

Typing rules are syntax-directed: for each clause of the abstract syntax for expressions there is exactly one rule with a conclusion of that form.

Properties

Theorem 3 (Progress)

If $\Gamma \vdash e : T$ and $\mathrm{dom}(\Gamma) \subseteq \mathrm{dom}(s)$ then either e is a value or there exist e', s' such that $\langle e, s\rangle \twoheadrightarrow \langle e', s'\rangle$.

Theorem 4 (Type preservation)

If $\Gamma \vdash e : T$ and $\mathrm{dom}(\Gamma) \subseteq \mathrm{dom}(s)$ and $\langle e, s\rangle \twoheadrightarrow \langle e', s'\rangle$ then $\Gamma \vdash e' : T$ and $\mathrm{dom}(\Gamma) \subseteq \mathrm{dom}(s')$.

Merging them together we can assert that well-typed programs don't get stuck:

Theorem 5 (Safety)

If $\Gamma \vdash e : T$, $\mathrm{dom}(\Gamma) \subseteq \mathrm{dom}(s)$, and $\langle e, s\rangle \twoheadrightarrow^* \langle e', s'\rangle$ then either e' is a value or there exist e'', s'' such that $\langle e', s'\rangle \twoheadrightarrow \langle e'', s''\rangle$.

Type checking, typeability, and type inference

Type checking problem

Given a type system, a type environment Γ, an expression e and a type T, is $\Gamma \vdash e : T$ derivable?

Type inference problem

Given a type system, a type environment Γ and an expression e, find a type T such that the type judgement $\Gamma \vdash e : T$ is derivable, or show there is none.

The second problem is usually harder than the first one. Solving it usually results in providing a type inference algorithm: computing a type T for an expression e, given a type environment Γ (or failing, if there is none).

However, for our type system both problems are quite easy to solve.

More properties

Theorem 6 (Type inference)

Given Γ, e, one can find T such that $\Gamma \vdash e : T$, or show that there is none.

Theorem 7 (Decidability of type checking)

Given Γ, e, T, one can decide $\Gamma \vdash e : T$.

Theorem 8 (Uniqueness of typing)

If $\Gamma \vdash e : T$ and $\Gamma \vdash e : T'$ then $T = T'$.