Semantics of Programming Languages

Simon

April 13, 2025

Contents

1 First Imperative Language (L1)

1.1 L1 - Syntax

L1 has the following syntax:

Booleans	$b \in \mathbb{B}$	$= \{\mathbf{true}, \mathbf{false}\}$
Integers	$n \in \mathbb{Z}$	
Locations	$\ell \in \mathbb{L}$	$= \{l, l_0, l_1, l_2, \dots\}$
Operations	$op ::= + \mid \geq$	
Expressions		

$$\begin{aligned} e ::= & n \mid b \mid e_1 \; op \; e_2 \mid \mathbf{if} \; e_1 \mathbf{then} \; e_2 \; \mathbf{else} \; e_3 \mid \\ & \ell := e \mid !l \mid \\ & \mathbf{skip} \mid e_1; e_2 \mid \\ & \mathbf{while} \; e_1 \; \mathbf{do} \; e_2 \end{aligned}$$

1.2 Operational Semantics

In order to describe the behaviour of L1 programs we will use structural operational semantics to define various forms of automata.

A transition system consists of

- A set **Config**
- A binary relation $\longrightarrow \subseteq \mathbf{Config} \times \mathbf{Config}$

Notation:

- $\longrightarrow^*$ is the reflexive transitive closure of $\longrightarrow$, so $c \longrightarrow c'$ iff there exists $k \geq 0$ and $c_0, \dots, c_k$ such that $c = c_0 \longrightarrow c_1 \longrightarrow \cdots \longrightarrow c_k = c'$
- $\not\longrightarrow$ is a unary predicate (a subset of **Config**) defined by $c \not\longrightarrow$ iff $\neg\exists \; c'.c \longrightarrow c$

We say there are finite stores $s \in \mathbb{L} \rightharpoonup \mathbb{Z}$. And then define configurations to be pairs $\langle e, s\rangle$ of an expression e, and a store s, which means our transition relation will have the form:

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

We can continue transitioning until we reach a value:

$$\mathbb{V} = \mathbb{B} \;\cup\; \mathbb{Z} \;\cup\; \{\mathbf{skip}\}$$

$$v ::= b \mid n \mid \mathbf{skip}$$

Stuck
Call a configuration $c = \langle e, s\rangle$ stuck if $\quad e \notin \mathbb{V} \;\wedge\; \langle e, s\rangle \not\longrightarrow$

1.3 L1 Full Operational Semantics

$$\text{(op +)} \quad \langle n_1 + n_2, s\rangle \longrightarrow \langle n, s\rangle \qquad \text{if } n = n_1 + n_2$$

$$\text{(op} \geq\text{)} \quad \langle n_1 \geq n_2, s\rangle \longrightarrow \langle b, s\rangle \qquad \text{if } b = (n_1 \geq n_2)$$

$$\text{(op1)} \quad \frac{\langle e_1, s\rangle \longrightarrow \langle e_1', s'\rangle}{\langle e_1 \text{ op } e_2, s\rangle \longrightarrow \langle e_1' \text{ op } e_2, s'\rangle}$$

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

$$\text{(deref)} \quad \langle !\ell, s\rangle \longrightarrow \langle n, s\rangle \qquad \text{if } \ell \in \text{dom}(s) \text{ and } s(\ell) = n$$

$$\text{(assign1)} \quad \langle \ell := n, s\rangle \longrightarrow \langle\ \textbf{skip},\ s + \{\ell \mapsto n\}\rangle \qquad \text{if } \ell \in \text{dom}(s)$$

$$\text{(assign2)} \quad \frac{\langle e, s\rangle \longrightarrow \langle e', s'\rangle}{\langle \ell := e, s\rangle \longrightarrow \langle \ell := e', s'\rangle}$$

$$\text{(seq1)} \quad \langle\ \text{skip}\ ; e_2, s\rangle \longrightarrow \langle e_2, s\rangle$$

$$\text{(seq2)} \quad \frac{\langle e_1, s\rangle \longrightarrow \langle e_1', s'\rangle}{\langle e_1; e_2, s\rangle \longrightarrow \langle e_1'; e_2, s'\rangle}$$

$$\text{(if1)} \quad \langle \textbf{if true then } e_2 \textbf{ else } e_3,\ s\rangle \longrightarrow \langle e_2,\ s\rangle$$

$$\text{(if1)} \quad \langle \textbf{if false then } e_2 \textbf{ else } e_3,\ s\rangle \longrightarrow \langle e_2,\ s\rangle$$

$$\text{(if3)} \quad \frac{\langle e_1, s\rangle \longrightarrow \langle e_1', s'\rangle}{\langle\ \text{if } e_1 \text{ then } e_2 \text{ else } e_3, s\rangle \longrightarrow \langle\ \text{if } e_1' \text{ then } e_2 \text{ else } e_3, s'\rangle}$$

$$\text{(while)} \quad \langle \textbf{while } e_1 \textbf{ do } e_2, s\rangle \longrightarrow \langle \textbf{if } e_1 \textbf{ then}\,(e_2;\ \textbf{while } e_1 \textbf{ do } e_2)\ \textbf{else skip}, s\rangle$$

1.4 L1 Determinancy

Theorem 1.1: Determinancy

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

Lemma 1.1: Irreducibility of Values

For all $e \in L_1$, if e is a value then

$$\forall s.\neg\exists e', s' \,.\, \langle e, s\rangle \longrightarrow \langle e', s;\rangle$$

Proof:

By definition of the set of values e must be in the form of one of the following $n, b, \mathbf{skip}$, by considering the rules of the language there is no rule with the conclusion of the form $\langle e, s\rangle \longrightarrow \langle e', s'\rangle$ for any e of the form previously described.

1.4.1 Inversion

For inductive proofs we often need an inversion property, that given a tuple in one inductively defined relation, gives you a case analysis of the possible last rule used.

Lemma 1.2: Inversion for $\longrightarrow$

(take any occurences of (n_i, n) to be integers $(\mathbb{Z})$, (ℓ, l_i) to be labels $(\mathbb{L})$ and (b, b_i) to be booleans $(\mathbb{B})$)

If $\langle e, s\rangle \longrightarrow \langle \hat{e}, \hat{s}\rangle$ then either:

(op $+$)	$\exists n_1, n_2, n$	s.t. $e = n_1 + n_2$ and $\hat{e} = n$ and $n = n_1 + n_2$ and $\hat{s} = s$
(op $\geq$)	$\exists n_1, n_2, b$	s.t. $e = n_1 \geq n_2$ and $\hat{e} = b$ and $\hat{s} = s$ and $b = n_1 \geq n_2$
(op1)	$\exists e_1, e_2, op, e_1'$	s.t. $e = e_1 \; op \; e_2$ and $e = e_1' \; op \; e_2$ and $\langle e_1, s\rangle \longrightarrow \langle e_1', s'\rangle$
(op2)	$\exists n, e_2, op, e_2'$	s.t. $e = n \;\; op \; e_2$ and $e = n \;\; op \; e_2'$ and $\langle e_2, s\rangle \longrightarrow \langle e_2', s'\rangle$
(deref)	$\exists \ell, n$	s.t. $e = !\ell$, $\hat{e} = n$ and $\hat{s} = s$ and $\ell \in \mathrm{dom}(s)$ and $s(\ell) = n$
(assign1)	$\exists \ell, n$	s.t. $e = (\ell := n)$ and $\hat{e} = \mathbf{skip}$ and $\hat{s} = s + \{\ell \mapsto n\}$
(assign2)	$\exists \ell, e_1, e_1'$	s.t. $e = (\ell := e_1)$ and $\hat{e} = (\ell := e_1')$ and $\ell \in \mathrm{dom}(s)$ and $\langle e_1, s\rangle \longrightarrow \langle e_1', s'\rangle$
(if1)	$\exists e_2, e_3$	s.t. $e = \mathbf{if\ true\ then}\ e_2\ \mathbf{else}\ e_3$ and $\hat{e} = e_2$ and $\hat{s} = s$
(if2)	$\exists e_2, e_3$	s.t. $e = \mathbf{if\ false\ then}\ e_2\ \mathbf{else}\ e_3$ and $\hat{e} = e_3$ and $\hat{s} = s$
(if3)	$\exists e_1, e_2, e_3, e_1'$	s.t. $e = \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3$ and $\hat{e} = \mathbf{if}\ e_1'\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3$ and $\langle e_1, s\rangle \longrightarrow \langle e_2, \hat{s}\rangle$
(while)	$\exists e_1, e_2$	s.t. $e = \mathbf{while}\ e_1\ \mathbf{do}\ e_2$ and $\hat{e} = \mathbf{if}\ e_1\ \mathbf{then}\ (e_2; \mathbf{while}\ e_1\ \mathbf{do}\ e_2)\ \mathbf{else\ skip}$ and $\hat{s} = s$

Proof:
Let

$$\Phi(e, s, \hat{e}, \hat{s}) = \langle e, s\rangle \longrightarrow \langle \hat{e}, \hat{s}\rangle \implies (\text{op+}) \;\vee\; (\text{op} \geq) \;\vee\; \ldots \;\vee\; (\text{if3}) \;\vee\; (\text{while})$$

Assume the LHS of the implication that is there is a transition from $\langle e, s\rangle$ to $\langle \hat{e}, \hat{s}\rangle$ then we are required to prove that the transition is one of the transitions defined by the operational semantics of the L1 language.

Since we are assuming there is a transition we only need to consider rules that provide a transition and that $\langle e, s\rangle$ is of the form of the LHS of a transition rule.

Case 1: (e is of the form $n_1 + n_2$)

Take s to be arbitrary then since $n_1, n_2 \in \mathbb{Z}$ there is a unique n s.t. $n_1 + n_2 = n$ so we we can use the (op $+$) rule to get $\langle n_1 + n_2, s\rangle \longrightarrow \langle n, s\rangle$

Case 2: (e is of the form $n_1 \geq n_2$)

Take s to be arbitrary then since $n_1, n_2 \in \mathbb{Z}$ there is a b s.t. $n_1 \geq n_2 = b$ so we we can use the (op $\geq$) rule to get $\langle n_1 \geq n_2, s\rangle \longrightarrow \langle b, s\rangle$

Case 3: (e is of the form $n_1 \geq n_2$)

Take s to be arbitrary then since $n_1, n_2 \in \mathbb{Z}$ there is a b s.t. $n_1 \geq n_2 = b$ so we we can use the (op $\geq$) rule to get $\langle n_1 \geq n_2, s\rangle \longrightarrow \langle b, s\rangle$

Proof of Determinancy

Take

$$\Phi(e) \triangleq \forall s, e', e'', s', s''. \ \big(\langle e, s\rangle \longrightarrow \langle e', s'\rangle \wedge \langle e, s\rangle \longrightarrow \langle e'', s''\rangle\big) \implies \langle e', s'\rangle = \langle e'', s''\rangle$$

We are RTP that $\forall e \in L_1. \ \Phi(e)$.

Case $e \in \{\mathbf{skip}, b, n\}$

If e is a value there are no transition rules that have a conclusion of the from $\langle v, s\rangle \longrightarrow \langle .., \ ..\rangle$ which means the LHS is false for this form meaning $\Phi(e)$ when e is a value holds vacuously.

For the remaining cases of the form of e we will use structural induction, and proceed by assuming the LHS of the implication and use further case analysis to look at the type of the transition.

Case $e = \,!\ell$

Take arbitrary s, e', e'', s', s'' s.t. $\langle !\ell, s\rangle \longrightarrow \langle e', s'\rangle \wedge \langle !\ell, s\rangle \longrightarrow \langle e'', s''\rangle$, the only transition rule that has a conclusion with the lhs expression being of the form $!\ell$ is (deref), so both transitions instances of this rule which then means.

$$\begin{array}{ll} \ell \in \mathrm{dom}(s) & \ell \in \mathrm{dom}(s) \\ e' = s(\ell) & e'' = s(\ell) \\ s' = s & s'' = s \end{array}$$

So from this we have $s' = s = s''$, and since s is a store which is a partial function mapping from the set of labels to the set of integers then if $\ell \in \mathrm{dom}(s)$ then $e' = s(\ell) = e''$, which means $\langle e', s'\rangle = \langle e'', s''\rangle$, so Φ holds in this case of the form of e.

Case $e = \ (\ell \coloneqq e_1)$ suppose that $\Phi(e_1)$ then we are RTP $\Phi(\ell \coloneqq e_1)$

Take arbitrary s, e', e'', s', s'' s.t. $\langle \ell \coloneqq e_1, s\rangle \longrightarrow \langle e', s'\rangle \wedge \langle \ell \coloneqq e_1, s\rangle \longrightarrow \langle e'', s''\rangle$, then there are two forms for the transition they are the rules (assign1) and (assign2) wlog there are 3 cases we have for the possible permutations of the pairings:

Case $\langle \ell \coloneqq e_1, s\rangle \longrightarrow \langle e', s'\rangle$ is (assign1) and $\langle \ell \coloneqq e_1, s\rangle \longrightarrow \langle e'', s''\rangle$ is (assign1)

Then for both to be instances of (assign1) then e_1 must be a value, and more specifically an integer value n. That is $e = (\ell \coloneqq n)$, then looking at (assign1):

$$\begin{array}{ll} \ell \in \mathrm{dom}(s) & \ell \in \mathrm{dom}(s) \\ e' = \mathbf{skip} & e'' = \mathbf{skip} \\ s' = s + \{\ell \mapsto n\} & s'' = s + \{\ell \mapsto n\} \end{array}$$

$$\implies \langle e', s'\rangle = \langle e'', s''\rangle$$

Case $\langle \ell \coloneqq e_1, s\rangle \longrightarrow \langle e', s'\rangle$ is (assign2) and $\langle \ell \coloneqq e_1, s\rangle \longrightarrow \langle e'', s''\rangle$ is (assign2)

Then $\langle \ell \coloneqq e_1, s\rangle \longrightarrow \langle \ell \coloneqq e_1', s'\rangle$ with $\langle e_1, s\rangle \longrightarrow \langle e_1, s'\rangle$ and $\langle \ell \coloneqq e_1, s\rangle \longrightarrow \langle \ell \coloneqq e_1'', s'\rangle$ with $\langle e_1, s\rangle \longrightarrow \langle e_1'', s''\rangle$. By the inductive hypothesis, $\Phi(e_1)$ so we have that $\langle e_1', s'\rangle = \langle e_1'', s''\rangle$, which means that

$$\begin{aligned} &\implies \langle \ell \coloneqq e_1', s'\rangle = \langle \ell \coloneqq e_1'', s'\rangle \\ &\implies \langle e', s'\rangle = \langle e'', s''\rangle \end{aligned}$$

Case $\langle \ell \coloneqq e_1, s\rangle \longrightarrow \langle e', s'\rangle$ is (assign2) and $\langle \ell \coloneqq e_1, s\rangle \longrightarrow \langle e'', s''\rangle$ is (assign1)

Then from the first transition which is an instance of (assign2) we get $\exists e_1', s'. \ \langle e_1, s\rangle \longrightarrow \langle e_1', s'\rangle$, however we get from the second transition which is an instance of (assign1) that e_1 is value and from the Irreducibility of Values lemma we have $\nexists e_1', s'. \ \langle e_1, s\rangle \longrightarrow \langle e_1', s'\rangle$, which is a contradiction so the assumption is false which means the rhs holds vacuously.

Case $e = (e_1\ op\ e_2)$ suppose that $\Phi(e_1) \wedge \Phi(e_2)$ then we are RTP $\Phi(e_1\ op\ e_2)$

Take arbitrary s, e', e'', s', s'' s.t. $\langle e_1\ op\ e_2, s\rangle \longrightarrow \langle e', s'\rangle \wedge \langle e_1\ op\ e_2, s\rangle \longrightarrow \langle e'', s''\rangle$,

1.5 L1 Type System

Define a ternary relation $\vdash$, that with infix notation reads $\Gamma \vdash e : T$ as e has type T, under the assumptions Γ on the types of locations that may occur in e.

We write T and T_{loc} for the sets of all terms of these grammars.

And let Γ range over **TypeEnv**, the finite partial functions from locations $\mathbb{L}$ to T_{loc}.
(Notation: write Γ as $l_1 : \text{intref}, \ldots, l_k : \text{intref}$ instead of $\{l_1 \mapsto \text{intref}, \ldots, l_k \mapsto \text{intref}\}$)

1.6 L1 Collected Typing System

Types of expressions:

$$T \quad ::= \quad \text{int} \mid \text{bool} \mid \text{unit}$$

Types of locations:

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

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

$$\text{(bool)} \quad \Gamma \vdash b : \text{bool} \qquad \text{for } b \in \mathbb{B}$$

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

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

$$\text{(assign)}\ \frac{\Gamma(\ell) = \text{intref} \qquad \Gamma \vdash e : \text{int}}{\Gamma \vdash \ell := e : \text{unit}}$$

$$\text{(deref)}\ \frac{\Gamma(\ell) = \text{intref}}{\Gamma \vdash !\ell : \text{int}}$$

$$\text{(skip)} \quad \Gamma \vdash \mathbf{skip} : \text{unit}$$

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

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

2 Functions - L2

2.1 Concrete Syntax

By convention, application associates to the left, so $e_1\ e_2\ e_3$ denotes $(e_1\ e_2)\ e_3$ whereas type arrows associate to the right so $T_1 \to T_2 \to T_3$ denotes $T_1 \to (T_2 \to T_3)$.

A **fn** extends to the right as far as parantheses permit, so **fn** $x : \text{unit} \Rightarrow x; x$ denotes **fn** $x : \text{unit} \Rightarrow (x; x)$

- Variables are not locations ($\mathbb{L} \cap \mathbb{X} = \{\}$)
- Cannot abstract on locations. For example, (**fn** $\ell : \text{intref} \Rightarrow !\ell$) is not in the syntax

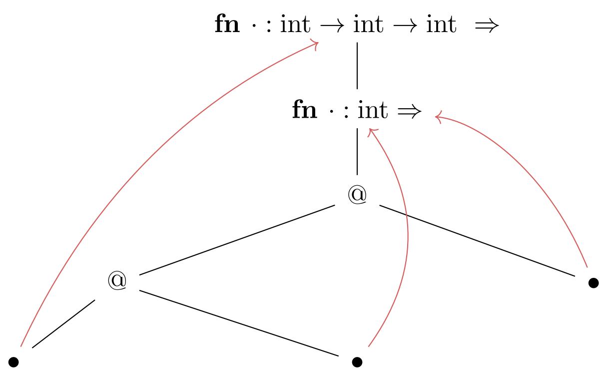
2.2 Alpha Conversion

In expressions **fn** $x : T \Rightarrow e$ the x is a binder

- Inside e, any x's (that themselves are not binder and are not inside another **fn** $x : T' \ldots$) mean the same thing - the formal paramater of this function.
- Outside this **fn** $x : T \Rightarrow e$ it does not matter which variable we used for the formal paramater

We will allow ourselves to at any time at all, in any expression replace the binding x and all occurences of x that are bound by that binder, by any other variable - so long as that does not change the binding graph.

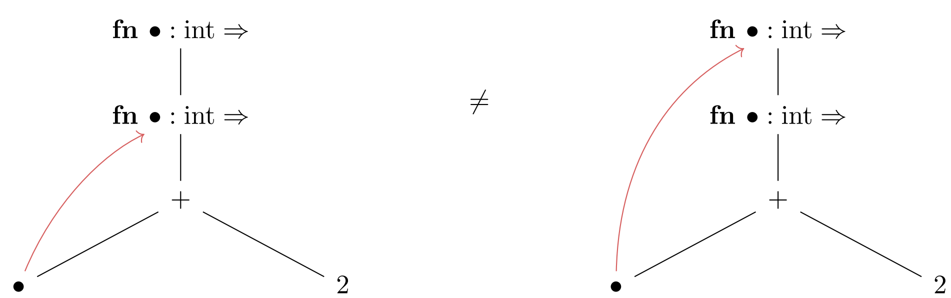
$$\textbf{fn}\ z : \text{int} \to \text{int} \to \text{int} \Rightarrow (\textbf{fn}\ y : \text{int} \Rightarrow z\ y\ y)$$



2.2.1 De Brujin Indices

Our implementation will use those pointers - known as De Brujin indices. Each occurence of a bound variable is represented by the number of **fn** $\cdot : T \Rightarrow$ nodes you have to count out to get to its binder.

$$\textbf{fn}\ \cdot : \text{int} \Rightarrow (\textbf{fn}\ \cdot : \text{int} \Rightarrow v_0 + 2) \qquad\qquad \textbf{fn}\ \cdot : \text{int} \Rightarrow (\textbf{fn}\ \cdot : \text{int} \Rightarrow v_1 + 2)$$



2.3 Free Variables

Say the free variables of an expression e are the set of variables x for which there is an occurence of x free in e

$$
\begin{array}{lcl}
\mathbf{fv}(n) & = & \{\} \\
\mathbf{fv}(b) & = & \{\} \\
\mathbf{fv}(!\ell) & = & \{\} \\
\mathbf{fv}(\mathbf{skip}) & = & \{\} \\
\mathbf{fv}(x) & = & \{x\} \\
\mathbf{fv}(\ell := e) & = & \mathbf{fv}(e) \\
\mathbf{fv}(\mathbf{fn}\ x : T \Rightarrow e) & = & \mathbf{fv}(e) - \{x\} \\
\mathbf{fv}(e_1 \ \ e_2) & = & \mathbf{fv}(e_1) \cup \mathbf{fv}(e_2) \\
\mathbf{fv}(e_1\ op\ e_2) & = & \mathbf{fv}(e_1) \cup \mathbf{fv}(e_2) \\
\mathbf{fv}(e_1; e_2) & = & \mathbf{fv}(e_1) \cup \mathbf{fv}(e_2) \\
\mathbf{fv}(\mathbf{while}\ e_1\ \mathbf{do}\ e_2) & = & \mathbf{fv}(e_1) \cup \mathbf{fv}(e_2) \\
\mathbf{fv}(\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3) & = & \mathbf{fv}(e_1) \cup \mathbf{fv}(e_2) \cup \mathbf{fv}(e_3)
\end{array}
$$

We say e is closed if $\mathbf{fv}(e) = \{\}$

If E is a set of expressions, write $\mathbf{fv}(E)$ for $\bigcup_{e\,\in\,E} \mathbf{fv}(e)$

2.3.1 Substitution

The semantics for functions will involve substituting actual paramaters for formal parameters.

Write $\{e/x\}e'$ (Can also use the notation from Computation Theory, $e'[e/x]$) for the result of substituting e for all free occurences of x in e'

$$
\begin{array}{lcll}
\{e/z\}x & = & \begin{cases} e & \text{if } x = z \\ x & \text{otherwise} \end{cases} & \\
\{e/z\}\big(\mathbf{fn}\ x : T \Rightarrow e_1\big) & = & \mathbf{fn}\ x : T \Rightarrow \big(\{e/z\}e_1\big) & \text{if} \quad x \neq z \qquad (^*) \\
 & & & \text{and } x \notin \mathbf{fv}(e)\ (^*) \\
\{e/z\}(e_1\ e_2) & = & \big(\{e/z\}e_1\big)\ \big(\{e/z\}e_2\big) & \\
\vdots & & \vdots &
\end{array}
$$

If $(^*)$ is not true, we first have to pick an alpha-variant of $\mathbf{fn}\ x : T \Rightarrow e_1$ to make it true

2.3.2 Simultaneous Substitution

A substitution σ is a finite partial function from variables to expressions.

Write σ as $\{e_1/x_1, \ldots, e_k/x_k\}$ instead of $\{x_1 \mapsto e_1, \ldots x_k \mapsto e_k\}$

Write $\mathrm{dom}(\sigma)$ for the set of variables in the domain of σ and $\mathrm{ran}(\sigma)$ for the set of expressions od σ, ie:

$$
\begin{array}{lcl}
\mathrm{dom}(\{e_1/x_1, \ldots, e_k/x_k\}) & = & \{x_1, \ldots, x_k\} \\
\mathrm{ran}(\{e_1/x_1, \ldots, e_k/x_k\}) & = & \{e_1, \ldots, e_k\}
\end{array}
$$

Define the application of simultaneous substitution to a term by:

$$
\begin{array}{lcll}
\sigma x & = & \begin{cases} \sigma x & \text{if } x \in \text{dom}(\sigma) \\ x & \text{otherwise} \end{cases} & \\
\sigma n & = & n & \\
\sigma(b) & = & b & \\
\sigma(\mathbf{skip}) & = & \mathbf{skip} & \\
\sigma(!\ell) & = & !\ell & \\
\sigma(\ell := e) & = & \ell := \sigma(e) & \\
\sigma\,(e_1\ e_2) & = & (\sigma\ e_1)\,(\sigma\ e_2) & \\
\sigma\,(e_1; e_2) & = & \sigma\,(e_1)\,;\ \sigma\,(e_2) & \\
\sigma\,(e_1\ \mathbf{op}\ e_2) & = & \sigma\,(e_1)\ \mathbf{op}\ \sigma\,(e_2) & \\
\sigma\,(\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3) & = & \mathbf{if}\ \sigma\,(e_1)\ \mathbf{then}\ \sigma\,(e_2)\ \mathbf{else}\ \sigma\,(e_3) & \\
\sigma\,(\mathbf{while}\ e_1\ \mathbf{do}\ e_2) & = & \mathbf{while}\ \sigma\,(e_1)\ \mathbf{do}\ \sigma\,(e_2) & \\
\sigma(\mathbf{fn}\ x : T \Rightarrow e) & = & \mathbf{fn}\ x : T \Rightarrow (\sigma\ e) & \text{if } x \notin \text{dom}(\sigma) \text{ and } x \notin \mathbf{fv}(\text{ran}(\sigma))\ (^*)
\end{array}
$$

Where $(^*)$ is similar to the non-simultaneous case, where we would choose an alpha-variant to make it true.

2.4 Function Behaviour

2.4.1 Call-by-Value (CBV), what we use

For this method we reduce the LHS of an application to a **fn**-term (which is now a value), then arguments to values, then replace all occurrences of the formal parameter in the **fn** term by that value

$$v ::= b \mid n \mid \mathbf{skip} \mid \mathbf{fn}\ x : T \Rightarrow e$$

$$\text{(app1)}\quad \frac{\langle e_1, s\rangle \longrightarrow \langle e_1', s'\rangle}{\langle e_1\ e_2, s\rangle \longrightarrow \langle e_1'\ e_2, s'\rangle}$$

$$\text{(app2)}\quad \frac{\langle e_2, s\rangle \longrightarrow \langle e_2', s'\rangle}{\langle v\ e_2, s\rangle \longrightarrow \langle v\ e_2', s'\rangle}$$

$$\text{(fn)}\quad \langle (\mathbf{fn}\ x : T \Rightarrow e)\ v, s\rangle \longrightarrow \langle \{v/x\}e, s\rangle$$

2.4.2 Call-by-Name (CBN)

For this method we reduce the LHS of an application to a **fn**-term (which is now a value), then replace all occurrences of the formal parameter in the **fn** term by the argument

$$v ::= b \mid n \mid \mathbf{skip} \mid \mathbf{fn}\ x : T \Rightarrow e$$

$$\text{(CBN-app)}\quad \frac{\langle e_1, s\rangle \longrightarrow \langle e_1', s'\rangle}{\langle e_1\ e_2, s\rangle \longrightarrow \langle e_1'\ e_2, s'\rangle}$$

$$\text{(fn)}\quad \langle (\mathbf{fn}\ x : T \Rightarrow e)\ e_2, s\rangle \longrightarrow \langle \{e_2/x\}e, s\rangle$$

2.4.3 Full Beta

Allow both left and right-hand sides of application to reduce. At any point where the LHS has reduced to a **fn**-term, replace all occurences of the formal parameter in the **fn**-term by the argument.

Allow reductions inside lambdas.

$$\text{(beta-app1)} \quad \frac{\langle e_1, s\rangle \longrightarrow \langle e_1', s'\rangle}{\langle e_1\ e_2, s\rangle \longrightarrow \langle e_1'\ e_2, s'\rangle}$$

$$\text{(beta-app2)} \quad \frac{\langle e_2, s\rangle \longrightarrow \langle e_2', s'\rangle}{\langle e_1\ e_2, s\rangle \longrightarrow \langle e_1\ e_2', s'\rangle}$$

$$\text{(beta-fn1)} \quad \langle(\textbf{fn}\ x : T \Rightarrow e)\ e_2, s\rangle \longrightarrow \langle\{e_2/x\}e, s\rangle$$

$$\text{(beta-fn2)} \quad \frac{\langle e, s\rangle \longrightarrow \langle e', s'\rangle}{\langle(\textbf{fn}\ x : T \Rightarrow e), s\rangle \longrightarrow \langle(\textbf{fn}\ x : T \Rightarrow e'), s\rangle}$$

This reduction relation includes the CBV and CBN relations, and also reduction inside lambdas.

We could also do **Normal-order reduction** as seen in computation theory, (leftmost, outermost variant of full beta).

We use **Call-By-Value** from now on

2.5 Local Definitions and Recursive Functions

For readability, we want to be able to name definitions, and to restrict their scope, so we add

$$\textbf{let val}\ x : T = e_1\ \textbf{in}\ e_2\ \textbf{end}$$

This x is a binder, binding any free occurences of x in e_2.

Can regard this as syntactic sugar:

$$\textbf{let val}\ x : T = e_1\ \textbf{in}\ e_2\ \textbf{end} \quad \rightsquigarrow \quad \big(\textbf{fn}\ x : T \Rightarrow e_2\big)e_1$$

Our alpha naming convention means that this is really a local definition - there is no way to refer to the locally-defined variable outside of the **let val**.
To allow for recursive definitions we could add something like:

$$\textbf{let val rec}\ x : T = e\ \textbf{in}\ e'\ \textbf{end}$$

Where x binds in both e and e'. But this leads to some weird things in a CBV language.

$$\textbf{let val rec}\ x : \text{int} = 3 :: x\ \textbf{in}\ x\ \textbf{end}$$

In a CBN language, it is reasonable to allow this kind of thing, as will only compute as much as needed. In a CBV language, would usually disallow, allowing recursive definitions only of functions. So we need to specialise the **let val rec** construct to only allow recursion at function types and only of function values.

$$\textbf{let val rec}\ x : T_1 \to T_2 = \big(\textbf{fn}\ y : T_1 \Rightarrow e_1\big)\ \textbf{in}\ e_2\ \textbf{end}$$

Here, the y binds in e_1; the x binds in $\big(\textbf{fn}\ y : T_1 \Rightarrow e_1\big)$ and e_2

2.6 Full L2 Operational Semantics

Additions to 1.3 L1 Full Operational Semantics

$$\text{(app1)} \quad \frac{\langle e_1, s\rangle \longrightarrow \langle e_1', s'\rangle}{\langle e_1\ e_2, s\rangle \longrightarrow \langle e_1'\ e_2, s'\rangle}$$

$$\text{(app2)} \quad \frac{\langle e_2, s\rangle \longrightarrow \langle e_2', s'\rangle}{\langle v\ e_2, s\rangle \longrightarrow \langle v\ e_2', s'\rangle}$$

$$\text{(fn)} \quad \langle (\mathbf{fn}\ x : T \Rightarrow e)\ v, s\rangle \longrightarrow \langle \{v/x\}e, s\rangle$$

$$\text{(let1)} \quad \frac{\langle e_1, s\rangle \longrightarrow \langle e_1', s'\rangle}{\langle \mathbf{let\ val}\ x : T = e_1\ \mathbf{in}\ e_2\ \mathbf{end}, s\rangle \longrightarrow \langle \mathbf{let\ val}\ x : T = e_1'\ \mathbf{in}\ e_2\ \mathbf{end}, s'\rangle}$$

$$\text{(let2)} \quad \langle \mathbf{let\ val}\ x : T = v\ \mathbf{in}\ e_2\ \mathbf{end}, s\rangle \longrightarrow \langle \{v/x\}e_2, s\rangle$$

$$\begin{aligned} \text{(letrecfn)} \quad & \langle \mathbf{let\ val\ rec}\ x : T_1 \to T_2 = (\mathbf{fn}\ y : T_1 \Rightarrow e_1)\ \mathbf{in}\ e_2\ \mathbf{end}, s\rangle \\ \longrightarrow \quad & \langle \{(\mathbf{fn}\ y : T_1 \Rightarrow \mathbf{let\ val\ rec}\ x : T_1 \to T_2 = (\mathbf{fn}\ y : T_1 \Rightarrow e_1)\ \mathbf{in}\ e_1\ \mathbf{end})/x\}e_2, s\rangle \end{aligned}$$

2.7 Function Typing

Before, Γ gave the types of store locations; it ranged over **TypeEnv** which was the set of all finite partial functions from locations $\mathbb{L}$ to T_{loc}.

Now, it must also give assumptions on the types of variables:
Type environments Γ are now pairs of a Γ_{loc} and Γ_{var} with the latter being a partial function from $\mathbb{X}$ to T.

We write

$$\mathrm{dom}(\Gamma) = \mathrm{dom}(\Gamma_{loc}) \quad \cup \quad \mathrm{dom}(\Gamma_{var})$$

If $x \notin \mathrm{dom}(\Gamma_{var})$, write $\Gamma, x : T$ for the pair of Γ_{loc} and the partial function which maps x to T but otherwise is like Γ_{var}.

Theorem 2.1: Progress

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

Theorem 2.2: Type Preservation

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

Theorem 2.3: Normalisation

In the sublanguage without while loops or store operations, if $\Gamma \vdash e : T$ and e closed then ther does not exist an infinite reduction sequence $\langle e, \{\}\rangle \longrightarrow \langle e_1, \{\}\rangle \longrightarrow \langle e_2, \{\}\rangle \longrightarrow \dots$

2.8 L2 Collected Typing System

Additions to 1.6 L1 Collected Typing System

Type environments Γ are now pairs of a Γ_{loc} and Γ_{var}

$$\text{(var)} \quad \Gamma \vdash x : T \quad \text{if } \Gamma(x) = T$$

$$\text{(fn)} \quad \frac{\Gamma, x : T \vdash e : T'}{\Gamma \vdash \textbf{fn}\ x : T \Rightarrow e : T \rightarrow T'}$$

$$\text{(app)} \quad \frac{\Gamma \vdash e_1 : T \rightarrow T' \quad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1 e_2 : T'}$$

$$\text{(let)} \quad \frac{\Gamma \vdash e_1 : T \quad \Gamma, x : T \vdash e_2 : T'}{\Gamma \vdash \textbf{let val}\ x : T = e_1\ \textbf{in}\ e_2\ \textbf{end} : T'}$$

$$\text{(let rec fn)} \quad \frac{\Gamma, x : T_1 \rightarrow T_2, y : T_1 \vdash e_1 : T_2 \qquad \Gamma, x : T_1 \rightarrow T_2 \vdash e_2 : T}{\Gamma \vdash\ \textbf{let val rec}\ x : T_1 \rightarrow T_2 = (\textbf{fn}\ y : T_1 \Rightarrow e_1)\ \textbf{in}\ e_2\ \textbf{end} : T}$$

3 Data - L3

3.1 Products and Sums

The **Product** type $T_1 * T_2$ lets you tuple together values of types T_1 and T_2.

A product may be constructed by simply using parentheses around two expressions seperated by a comma.

$$(e_1, e_2)$$

Individual elements may be projected out of the product using either #1 or #2 for projecting the first or second expression out of the product respectively. This may be seen as destructing the product type.

$$\langle \#1(v_1, v_2), s\rangle \longrightarrow \langle v_1, s\rangle$$

We however do not allow for $\#e\ e'$ as this cannot be typechecked.

Before we allow for data to be projected out of the product we must reduce the product down to a product of two values, (which itself is now a value). We do this with left to right reductions like the rest of the language.

$$\frac{\langle e, s\rangle \longrightarrow \langle e', s'\rangle}{\langle \#1\ e, s\rangle \longrightarrow \langle \#1\ e', s\rangle}$$

This will reduce down until the product is a value, meaning it only contains other values. (Similar for #2)

The **Sum** type $T_1 + T_2$ lets us form a disjoint union, with a value of the sum type either being being a value of type T_1 or a value of type T_2.

We construct a product type by injecting a value into the type, either inject left or inject right to create a value of type T_1 or T_2 respectively.

$$\textbf{inl}\ e : T \qquad\qquad \textbf{inr}\ e : T$$

We specify the type that the expression is being injected into to maintain the unique typing property. This specifiying the type is not the type inference but part of the semantics. If we did not specify then then we would have:

$$\{\} \vdash \textbf{inl}\ 3 : \text{int} + \text{int} \quad \wedge \quad \{\} \vdash \textbf{inl}\ 3 : \text{int} + \text{bool} \quad \wedge \quad \ldots$$

A compiler might use a type inference algorithm that can infer the type that it should be.

We can determine what value a sum type is by case splitting, this can also be seen as destructing the sum type.

$$\textbf{case}\ e : T\ \textbf{of}\ \textbf{inl}\ (x : T_1) \Rightarrow e_1 \mid \textbf{inr}\ (y : T_2) \Rightarrow e_2$$

Type	Constructors	Destructors
$T \to T$	$\textbf{fn}\ x : T \Rightarrow \_$	$\_\ e$
$T * T$	$(\_, \_)$	$\#1\ \_ \quad \vee \quad \#2\ \_$
$T + T$	$\textbf{inl}\ (\_) \quad \vee \quad \textbf{inr}\ (\_)$	**case**
bool	$\textbf{true} \quad \vee \quad \textbf{false}$	**if**

3.2 Datatypes and Records

Records are a generalisation of products. Labels $lab \in \mathbb{LAB}$ for a set $\mathbb{LAB} = \{p, q, \dots\}$. Each label has a corresponding expression associated with it, which can be projected out using the label.

$$\#q\ \{p : (x+2), q : (y+1), r : (5)\} \longrightarrow^* y + 1$$

3.3 Mutable Store

We are changing how the store works now from L1 and L2, in those languages we could only store integers, but we would like to store any value. We also cannot create any new locations during the runtime and thus they must all exist at the beginning. Functions cannot also abstact on locations.

We remove the specific intref type and replace the type with a T ref type, and remove the specific assign and deref rules in favour of assigning an expression to another, and derefencing an expression etc., making everything more general.

We make locations variables now, and the store s was a finite partial map from $\mathbb{L}$ to $\mathbb{Z}$ now we take stores to be the finite partial map from $\mathbb{L}$ to the set of all values.

3.4 Full L3 Operational Semantics

Additions to 2.6 Full L2 Operational Semantics

$$\text{(pair1)}\ \frac{\langle e_1, s\rangle \longrightarrow \langle e_1', s'\rangle}{\langle (e_1, e_2), s\rangle \longrightarrow \langle (e_1', e_2), s'\rangle}$$

$$\text{(pair2)}\ \frac{\langle e_2, s\rangle \longrightarrow \langle e_2', s'\rangle}{\langle (v, e_2), s\rangle \longrightarrow \langle (v, e_2'), s'\rangle}$$

$$\text{(proj1)}\ \langle \#1(v_1, v_2), s\rangle \longrightarrow \langle v_1, s'\rangle \qquad \text{(proj2)}\ \langle \#2(v_1, v_2), s\rangle \longrightarrow \langle v_2 s'\rangle$$

$$\text{(proj3)}\ \frac{\langle e, s\rangle \longrightarrow \langle e', s'\rangle}{\langle \#1\ \ e, s\rangle \longrightarrow \langle \#1\ \ e', s'\rangle} \qquad \text{(proj4)}\ \frac{\langle e, s\rangle \longrightarrow \langle e', s'\rangle}{\langle \#2\ \ e, s\rangle \longrightarrow \langle \#2\ \ e', s'\rangle}$$

$$\text{(inl)}\ \frac{\langle e, s\rangle \longrightarrow \langle e', s'\rangle}{\langle \textbf{inl}\ e : T, s\rangle \longrightarrow \langle \textbf{inl}\ e' : T, s'\rangle} \qquad \text{(inl)}\ \frac{\langle e, s\rangle \longrightarrow \langle e', s'\rangle}{\langle \textbf{inl}\ e : T, s\rangle \longrightarrow \langle \textbf{inl}\ e' : T, s'\rangle}$$

$$\text{(case1)}\ \frac{\langle e, s\rangle \longrightarrow \langle e', s'\rangle}{\begin{array}{r}\langle \textbf{case}\ e\ : T\ \textbf{of inl}\ (x : T_1) \Rightarrow e_1 \mid \textbf{inr}\ (y : T_2) \Rightarrow e_2, s\rangle \\ \longrightarrow \langle \textbf{case}\ e' : T\ \textbf{of inl}\ (x : T_1) \Rightarrow e_1 \mid \textbf{inr}\ (y : T_2) \Rightarrow e_2, s'\rangle\end{array}}$$

$$\text{(case2)}\ \langle \textbf{case inl}\ v : T\ \textbf{of inl}\ (x : T_1) \Rightarrow e_1 \mid \textbf{inr}\ (y : T_2) \Rightarrow e_2, s\rangle \longrightarrow \langle \{v/x\}e_1, s\rangle$$

$$\text{(case3)}\ \langle \textbf{case inr}\ v : T\ \textbf{of inl}\ (x : T_1) \Rightarrow e_1 \mid \textbf{inr}\ (y : T_2) \Rightarrow e_2, s\rangle \longrightarrow \langle \{v/y\}e_2, s\rangle$$

3.5 Type Checking The Store

For L1, our type properties used $\mathrm{dom}(\Gamma) \subseteq \mathrm{dom}(s)$ to express the condition 'All locations mentioned in Γ exist in the store s'.

Now we require more:

- For each $\ell \in \mathrm{dom}(s)$ need that $s(\ell)$ is typable
- $s(\ell)$ might contain some other locations

Definition 3.1: Well-typed Store

Let $\Gamma \vdash s$ if $\mathrm{dom}(\Gamma) = \mathrm{dom}(s)$ and if for all $\ell \in \mathrm{dom}(s)$, if $\Gamma(\ell) = T$ ref then $\Gamma \vdash s(\ell) : T$.

Theorem 3.1: Progress

If e closed and $\Gamma \vdash e : T$ and $\Gamma \vdash e : T$ and $\Gamma \vdash s$ then either e is a value or there exists e', s' such that $\langle e, s\rangle \longrightarrow \langle e', s'\rangle$.

Theorem 3.2: Type Preservation

If e closed and $\Gamma \vdash e : T$ and $\Gamma \vdash s$ and $\langle e, s\rangle \longrightarrow \langle e', s'\rangle$ then e' is closed and for some Γ' with disjoint domain to Γ we have $\Gamma, \Gamma' \vdash e' : T$ and $\Gamma, \Gamma' \vdash s$.

Theorem 3.3: Type Safety

If e closed and $\Gamma \vdash e : T$ and $\Gamma \vdash s$ and $\langle e, s\rangle \longrightarrow^* \langle e', s'\rangle$ then either e' is a value or there exists e'', s'', such that $\langle e', s'\rangle \longrightarrow \langle e'', s''\rangle$

3.6 Evaluation Contexts

Instead of having multiple individual rules we can create an evaluation context. This is especially beneficial for larger languages but not necessary for the L languages described here. Define evaluation contexts:

$$
\begin{aligned}
E \quad ::= \quad & \_\ \ op\ \ e \mid v\ \ op\ \ \_ \mid \textbf{if}\ \_\ \textbf{then}\ e\ \textbf{else}\ e \mid \\
& \_;\ e \mid \\
& \_\ \ e \mid v\ \ \_ \\
& \textbf{let val}\ x : T = \_\ \textbf{in}\ e_2\ \textbf{end} \mid \\
& (\_,\ e) \mid (v, \_) \mid \#1\ \_ \mid \#2\ \_ \mid \\
& \textbf{inl}\ \_\ \textbf{of inl}\ (x : T) \Rightarrow e \mid \textbf{inr}\ (x : T) \Rightarrow e \mid \\
& \{lab_1 = v_1, \ldots,\ lab_i = \_,\ \ldots,\ lab_k = e_k\} \mid \#lab\ \ \_ \mid \\
& \_ := e \mid v := \_ \mid !\_ \mid \mathrm{ref}\ \_
\end{aligned}
$$

And we now have the single context rule:

$$
\text{(eval)}\quad \frac{\langle e, s\rangle \longrightarrow \langle e', s'\rangle}{\langle E[e], s\rangle \longrightarrow \langle E[e'], s'\rangle}
$$

(Where $E[e]$ means that we substitute an expression e into on the holes _ in the above evaluation context)

We replace all the rules with premises with the above rule, we also add the computation rules as well ((op+), (op≥), (if1), (if2), ...)

4 Subtyping and Objects

The type system so far is very rigid we would like to introduce Subtype Polymorphism.

Any value of type $\{p : \text{int}, q : \text{int}\}$ can be used wherever a value of type $\{p : \text{int}\}$ is expected.
We introduce a subtyping relation between types, written $T <: T'$, read as T is a subtype of T'

$$\{p : \text{int}, q : \text{int}\} \;\; <: \;\; \{p : \text{int}\} \;\; <: \;\; \{\}$$

Introduce a subsumption rule

$$\text{(sub)}\ \frac{\Gamma \vdash e : T \qquad T <: T'}{\Gamma \vdash e : T'}$$

The Subtype Relation

$$\text{(s-refl)}\ \frac{}{T <: T}$$

$$\text{(s-trans)}\ \frac{T <: T' \qquad T' <: T''}{T <: T''}$$

$$\text{(s-record-width)}\ \ \{lab_1 : T_1, \ldots, lab_n : T_n, \ldots, lab_{n+k} : T_{n+k}\} <: \{lab_1 : T_1, \ldots, lab_n : T_n\}$$

$$\text{(s-record-depth)}\ \frac{T_1 <: T_1' \qquad \ldots \qquad T_n <: T_n'}{\{lab_1 : T_1, \ldots, lab_n : T_n\} <: \{lab_1 : T_1', \ldots, lab_n : T_n'\}}$$

$$\text{(s-record-depth)}\ \frac{\pi \text{ a permutation of } 1, \ldots, n}{\{lab_1 : T_1, \ldots, lab_n : T_n\} <: \{lab_{\pi(1)} : T_{\pi(1)}, \ldots, lab_{\pi(n)} : T_{\pi(n)}\}}$$

$$\text{(s-fn)}\ \frac{T_1' <: T_1 \qquad T_2 <: T_2'}{T_1 \to T_2 <: T_1' \to T_2'}$$

$$\text{(s-pair)}\ \frac{T_1 <: T_1' \qquad T_2 <: T_2'}{T_1 * T_2 <: T_1' * T_2'}$$

$$\text{(s-pair)}\ \frac{T_1 <: T_1' \qquad T_2 <: T_2'}{T_1 * T_2 <: T_1' * T_2'}$$

5 Concurrency

$$\begin{aligned} e \quad ::= \quad & n \mid b \mid e_1 \ op \ e_2 \mid \textbf{if } e_1 \textbf{then } e_2 \textbf{ else } e_3 \mid \\ & \ell := e \mid !l \mid \\ & () \mid e_1; e_2 \mid \\ & \textbf{while } e_1 \textbf{ do } e_2 \mid \\ & e_1 | e_2 \end{aligned}$$

$$\begin{aligned} T \quad & ::= \quad \text{int} \mid \text{bool} \mid \text{unit} \mid \text{proc} \\ T_{loc} \quad & ::= \quad \text{intref} \end{aligned}$$

$$\text{(thread)} \ \frac{\Gamma \vdash e : \text{unit}}{\Gamma \vdash e : \text{proc}}$$

$$\text{(parallel)} \ \frac{\Gamma \vdash e_1 : \text{proc} \qquad \Gamma \vdash e_2 : \text{proc}}{\Gamma \vdash e_1 | e_2 : \text{proc}}$$

$$\text{(parallel1)} \ \frac{\langle e_1, s \rangle \longrightarrow \langle e_2, s \rangle}{\langle e_1 | e_2, s \rangle \longrightarrow \langle e_1' | e_2, s \rangle}$$

$$\text{(parallel1)} \ \frac{\langle e_2, s \rangle \longrightarrow \langle e_2', s \rangle}{\langle e_1 | e_2, s \rangle \longrightarrow \langle e_1 | e_2', s \rangle}$$

5.1 Mutexes

Mutex names $m \in \mathbb{M} = \{m, m_1, \ldots, m_n\}$

Configurations $\langle e, s, M \rangle$ where $M : \mathbb{M} \longrightarrow \mathbb{B}$ is the mutex state

Expressions $e ::= \ldots \mid \textbf{lock } m \mid \textbf{unlock } m$

$$\text{(lock)} \ \frac{}{\Gamma \vdash \textbf{lock } m : \text{unit}} \qquad \text{(unlock)} \ \frac{}{\Gamma \vdash \textbf{unlock } m : \text{unit}}$$

$$\text{(lock)} \quad \langle \textbf{lock } m, s, M \rangle \longrightarrow \langle (), s, M + \{m \mapsto \textbf{true}\} \rangle \quad \text{if } \neg M(m)$$

$$\text{(unlock)} \quad \langle \textbf{unlock } m, s, M \rangle \longrightarrow \langle (), s, M + \{m \mapsto \textbf{false}\} \rangle$$

lock atomically checks the mutex is currently false and changes its state and lets the thread proceed.

All other rules need to carry the mutex state around and the changes from premises to its transition.

5.2 An Ordered 2PL Discipline, Informally

Fix an association between locations and mutexes. For simplicity make it 1:1 - ℓ with m, l_1 with m_1 etc.

Fix a lock acquisition order. For simplicity, make it $m, m_0, m_1, m_2, \ldots$

Require that each e_i

- acquires the lock m_j for each location l_j it uses, before it uses it
- acquires and releases each lock in a properly bracketed way
- does not acquire any lock after it's released any lock
- acquires locks in increasing order

Then any concurrent state should never deadlock and be serializable - any execution of it should be 'equivalent' to an execution of some permutaion that is a sequence of the expressions sequentially.

6 Semantic Equivalence

Take $\simeq$ to the semantic equivalence relation symbol. For $\simeq$ to be good it must obey

1. Programs that result in observably different values from some initial store are not equivalent.

$$\big(\exists s, s_1, s_2, v_1, v_2. \langle e_1, s\rangle \longrightarrow^* \langle v_1, s_1\rangle \wedge \langle e_2, s\rangle \longrightarrow^* \langle v_2, s_2\rangle \wedge v_1 \neq v_2\big) \implies e_1 \not\simeq e_2$$

2. Programs that terminate must not be equivalent to programs that do not
3. $\simeq$ must be an equivalence relation

$$e \simeq e \qquad e_1 \simeq e_2 \implies e_2 \simeq e_1 \qquad e_1 \simeq e_2 \simeq e_3 \implies e_1 \simeq e_3$$

4. $\simeq$ must be a congruence

$$\text{if } e_1 \simeq e_2 \text{ then for any context } C \text{ we must have } C[e_1] \simeq C[e_2]$$

5. $\simeq$ should relate as many programs as possible subject to the above

6.1 Semantic Equivalence for L1

Consider Typed L1 again.

Define $e_1 \simeq^T_\Gamma e_2$ to hold iff forall s such that $\mathrm{dom}(\Gamma) \subseteq \mathrm{dom}(s)$, we have $\Gamma \vdash e_1 : T$, $\Gamma \vdash e_2 : T$ and either:

$$\langle e_1, s\rangle \longrightarrow^\omega \quad \wedge \quad \langle e_2, s\rangle \longrightarrow^\omega$$

or

$$\exists v, s'.\ \langle e_1, s\rangle \longrightarrow^* \langle v, s'\rangle \quad \wedge \quad \langle e_2, s\rangle \longrightarrow^* \langle v, s'\rangle$$

With the L1 contexts being:

$$\begin{array}{rl} C & ::= \ \_ \ \ op \ \ e \mid e_1 \ \ op \ \ \_ \mid \\ & \quad \textbf{if } \_ \textbf{ then } e_2 \textbf{ else } e_3 \mid \textbf{if } e_1 \textbf{ then } \_ \textbf{ else } e_3 \mid \\ & \quad \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } \_ \mid \\ & \quad \ell := \_ \mid \\ & \quad \_;\ e_2 \mid e_1;\ \_ \\ & \quad \textbf{while } \_ \textbf{ do } e_2 \mid \textbf{while } e_1 \textbf{ do } \_ \end{array}$$

Say $\simeq^T_\Gamma$ has the congruence property if whenever $e_1 \simeq^T_\Gamma e_2$ we have, for all C and T', if $\Gamma \vdash C[e_1] : T'$ and $\Gamma \vdash C[e_2] : T'$ then $C[e_1] \simeq^{T'}_\Gamma C[e_2]$

6.2 Contextual Equivalence for L3

Consider typed L3 programs, $\Gamma \vdash e_1 : T$ and $\Gamma \vdash e_2 : T$.

We say they are contextually equivalent if, for every context C such that $\{\} \vdash C[e_1] : \mathrm{unit}$ and $\{\} \vdash C[e_2] : \mathrm{unit}$, we either have:

$$\langle C[e_1]\{\}\rangle \longrightarrow^\omega \quad \wedge \quad \langle C[e_2], \{\}\rangle \longrightarrow^\omega$$

or

$$\exists s_1, s_2. \langle C[e_1], \{\}\rangle \longrightarrow^* \langle \textbf{skip}, s_1\rangle \quad \wedge \quad \langle C[e_2], \{\}\rangle \longrightarrow^* \langle \textbf{skip}, s_2\rangle$$