Data

So far we have only looked at very simple basic data types – int, bool, and unit, and functions over them. We now explore more *structured data*, in as simple a form as possible, and revisit the semantics of *mutable store*.

Products, Sums, and Records

The two basic notions are the *product* and the *sum* type.

The product type $T_1 * T_2$ lets you tuple together values of types T_1 and T_2 – so for example a function that takes an integer and returns a pair of an integer and a boolean has type $\mathsf{int} \rightarrow (\mathsf{int} * \mathsf{bool})$. In C one has `structs`; in Java classes can have many fields.

The sum type $T_1 + T_2$ lets you form a disjoint union, with a value of the sum type either being a value of type T_1 *or* a value of type T_2. In C one has `unions`; in Java one might have many subclasses of a class

In most languages these appear in richer forms, e.g. with *labelled records* rather than simple products, or *labelled variants*, or ML *datatypes* with named *constructors*, rather than simple sums. We'll look at labelled records in detail, as a preliminary to the later lecture on subtyping.

Many languages don't allow structured data types to appear in arbitrary positions – e.g. the old C lack of support for functions that return structured values, inherited from close-to-the-metal early implementations. They might therefore have to have functions or methods that take a list of arguments, rather than a single argument that could be of product (or sum, or record) type.

Products

$$T \quad ::= \quad \ldots \mid T_1 * T_2$$

$$e \quad ::= \quad \ldots \mid (e_1, e_2) \mid \#1\ e \mid \#2\ e$$

Design choices:

- pairs, not arbitrary tuples – have $\mathsf{int} * (\mathsf{int} * \mathsf{int})$ and $(\mathsf{int} * \mathsf{int}) * \mathsf{int}$, but (a) they're different, and (b) we don't have $(\mathsf{int} * \mathsf{int} * \mathsf{int})$. In a full language you'd likely allow (b) (and still have it be a different type from the other two).
- have projections #1 and #2, not pattern matching $\mathbf{fn}\ (x, y) \Rightarrow e$. A full language should allow the latter, as it often makes for much more elegant code.
- don't have $\#e\ e'$ (couldn't typecheck!).

Products - typing

(pair) $$\frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : T_2}{\Gamma \vdash (e_1, e_2) : T_1 * T_2}$$

(proj1) $$\frac{\Gamma \vdash e : T_1 * T_2}{\Gamma \vdash \#1\ e : T_1}$$

(proj2) $$\frac{\Gamma \vdash e : T_1 * T_2}{\Gamma \vdash \#2\ e : T_2}$$

Products - reduction

$$v \quad ::= \quad \ldots \mid (v_1, v_2)$$

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

(pair2) $$\frac{\langle e_2, s\rangle \longrightarrow \langle e_2', s'\rangle}{\langle (v_1, e_2), s\rangle \longrightarrow \langle (v_1, e_2'), s'\rangle}$$

(proj1) $\langle \#1(v_1, v_2), s\rangle \longrightarrow \langle v_1, s\rangle$ (proj2) $\langle \#2(v_1, v_2), s\rangle \longrightarrow \langle v_2, s\rangle$

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

Again, have to choose evaluation strategy (CBV) and evaluation order (left-to-right, for consistency).

Sums (or Variants, or Tagged Unions)

$$T \quad ::= \quad \ldots \mid T_1 + T_2$$
$$e \quad ::= \quad \ldots \mid \textbf{inl}\ \ e : T \mid \textbf{inr}\ \ e : T \mid$$
$$\textbf{case}\ \ e\ \ \textbf{of}\ \ \textbf{inl}\ (x_1 : T_1) \Rightarrow e_1 \mid \textbf{inr}\ (x_2 : T_2) \Rightarrow e_2$$

Those xs are binders.

Here we diverge slightly from Moscow ML syntax - our $T_1 + T_2$ corresponds to the Moscow ML `(T1,T2) Sum` in the context of the declaration

```
datatype ('a,'b) Sum = inl of 'a | inr of 'b;
```

Sums - typing

(inl) $$\frac{\Gamma \vdash e : T_1}{\Gamma \vdash \textbf{inl}\ \ e : T_1 + T_2 : T_1 + T_2}$$

(inr) $$\frac{\Gamma \vdash e : T_2}{\Gamma \vdash \textbf{inr}\ \ e : T_1 + T_2 : T_1 + T_2}$$

(case) $$\frac{\begin{array}{c}\Gamma \vdash e : T_1 + T_2 \\ \Gamma, x : T_1 \vdash e_1 : T \\ \Gamma, y : T_2 \vdash e_2 : T\end{array}}{\Gamma \vdash \textbf{case}\ \ e\ \ \textbf{of}\ \ \textbf{inl}\ (x : T_1) \Rightarrow e_1 \mid \textbf{inr}\ (y : T_2) \Rightarrow e_2 : T}$$

Why do we have these irritating type annotations? To maintain the unique typing property, as otherwise

$$\textbf{inl}\ \ 3{:}\text{int} + \text{int}$$

and

$$\textbf{inl}\ \ 3{:}\text{int} + \text{bool}$$

You might:

- have a compiler use a type inference algorithm that can infer them.
- require every sum type in a program to be declared, each with different names for the constructors **inl** , **inr** (cf OCaml).
- ...

Sums - reduction

$$v \ ::= \ \ldots \mid \textbf{inl}\ \ v{:}T \mid \textbf{inr}\ \ v{:}T$$

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

(case1) $$\frac{\langle e, s\rangle \longrightarrow \langle e', s'\rangle}{\langle \textbf{case}\ \ e\ \ \textbf{of}\ \ \textbf{inl}\ (x{:}T_1) \Rightarrow e_1 \mid \textbf{inr}\ (y{:}T_2) \Rightarrow e_2, s\rangle \longrightarrow \langle \textbf{case}\ \ e'\ \ \textbf{of}\ \ \textbf{inl}\ (x{:}T_1) \Rightarrow e_1 \mid \textbf{inr}\ (y{:}T_2) \Rightarrow e_2, s'\rangle}$$

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

(inr) and (case3) like (inl) and (case2)

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

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

Constructors and Destructors

type	constructors	destructors
$T \to T$	**fn** $x{:}T \Rightarrow \_$	$\_\ e$
$T * T$	$(\_, \_)$	$\#1\ \_ \quad \#2\ \_$
$T + T$	**inl** $(\_)$ **inr** $(\_)$	**case**
bool	**true** **false**	**if**

The Curry-Howard Isomorphism

(var) $\Gamma, x{:}T \vdash x{:}T$	$\Gamma, P \vdash P$
(fn) $\dfrac{\Gamma, x{:}T \vdash e{:}T'}{\Gamma \vdash \textsf{fn}\ x{:}T \Rightarrow e : T \to T'}$	$\dfrac{\Gamma, P \vdash P'}{\Gamma \vdash P \to P'}$
(app) $\dfrac{\Gamma \vdash e_1{:}T \to T' \qquad \Gamma \vdash e_2{:}T}{\Gamma \vdash e_1\ e_2{:}T'}$	$\dfrac{\Gamma \vdash P \to P' \qquad \Gamma \vdash P}{\Gamma \vdash P'}$
(pair) $\dfrac{\Gamma \vdash e_1{:}T_1 \qquad \Gamma \vdash e_2{:}T_2}{\Gamma \vdash (e_1, e_2){:}T_1 * T_2}$	$\dfrac{\Gamma \vdash P_1 \qquad \Gamma \vdash P_2}{\Gamma \vdash P_1 \wedge P_2}$
(proj1) $\dfrac{\Gamma \vdash e{:}T_1 * T_2}{\Gamma \vdash \#1\ e{:}T_1}$ (proj2) $\dfrac{\Gamma \vdash e{:}T_1 * T_2}{\Gamma \vdash \#2\ e{:}T_2}$	$\dfrac{\Gamma \vdash P_1 \wedge P_2}{\Gamma \vdash P_1} \quad \dfrac{\Gamma \vdash P_1 \wedge P_2}{\Gamma \vdash P_2}$
(inl) $\dfrac{\Gamma \vdash e{:}T_1}{\Gamma \vdash \textsf{inl}\ e{:}T_1 + T_2{:}T_1 + T_2}$	$\dfrac{\Gamma \vdash P_1}{\Gamma \vdash P_1 \vee P_2}$
(inr), (case), (unit), (zero), etc.. – but not (letrec)	

ML Datatypes

Datatypes in ML generalise both sums and products, in a sense

```
datatype IntList = Null of unit
                 | Cons of Int * IntList
```

is (roughly!) like saying

```
IntList = unit + (Int * IntList)
```

Note (a) this involves recursion at the type level (e.g. types for binary trees), (b) it introduces constructors (`Null` and `Cons`) for each summand, and (c) it's *generative* - two different declarations of `IntList` will make different types. Making all that precise is beyond the scope of this course.

Records

A mild generalisation of products that'll be handy later.

Take field labels

Labels $lab \in \mathbb{LAB}$ for a set $\mathbb{LAB} = \{\mathrm{p}, \mathrm{q}, ...\}$

$$T ::= \ldots \mid \{lab_1{:}T_1, .., lab_k{:}T_k\}$$
$$e ::= \ldots \mid \{lab_1 = e_1, .., lab_k = e_k\} \mid \#lab\ e$$

(where in each record (type or expression) no lab occurs more than once)

Note:

- The condition on record formation means that our syntax is no longer 'free'. Formally, we should have a well-formedness judgment on types.
- Labels are not the same syntactic class as variables, so (**fn** x:$T \Rightarrow \{\mathrm{x} = 3\}$) is not an expression.
- Does the order of fields matter? Can you use reuse labels in different record types? The typing rules will fix an answer.
- In ML a pair (**true**, **fn** x:int $\Rightarrow$ x) is actually syntactic sugar for a record $\{1 =$ **true**$, 2 =$ **fn** x:int $\Rightarrow$ x$\}$.
- Note that $\#lab\ e$ is not an application, it just looks like one in the concrete syntax.
- Again we will choose a left-to-right evaluation order for consistency.

Records - typing

$$\text{(record)}\quad \frac{\Gamma \vdash e_1 : T_1 \quad .. \quad \Gamma \vdash e_k : T_k}{\Gamma \vdash \{lab_1 = e_1, .., lab_k = e_k\} : \{lab_1 : T_1, .., lab_k : T_k\}}$$

$$\text{(recordproj)}\quad \frac{\Gamma \vdash e : \{lab_1 : T_1, .., lab_k : T_k\}}{\Gamma \vdash \#lab_i\ e : T_i}$$

- Here the field order matters, so (**fn** x:{*foo*:int, *bar*:bool} ⇒ x){*bar* = **true**, *foo* = 17} does not typecheck. In ML, though, the order doesn't matter – so Moscow ML will accept strictly more programs in this syntax than this type system allows.
- Here and in Moscow ML can reuse labels, so {} ⊢ ({*foo* = 17}, {*foo* = **true**}):{*foo*:int}∗ {*foo*:bool} is legal, but in some languages (e.g. OCaml) you can't.

Records - reduction

$$v \quad ::= \quad ... \mid \{lab_1 = v_1, .., lab_k = v_k\}$$

$$\text{(record1)}\quad \frac{\langle e_i, s\rangle \longrightarrow \langle e_i', s'\rangle}{\langle \{lab_1 = v_1, .., lab_i = e_i, .., lab_k = e_k\}, s\rangle \longrightarrow \langle \{lab_1 = v_1, .., lab_i = e_i', .., lab_k = e_k\}, s'\rangle}$$

$$\text{(record2)}\quad \langle \#lab_i\ \{lab_1 = v_1, .., lab_k = v_k\}, s\rangle \longrightarrow \langle v_i, s\rangle$$

$$\text{(record3)}\quad \frac{\langle e, s\rangle \longrightarrow \langle e', s'\rangle}{\langle \#lab_i\ e, s\rangle \longrightarrow \langle \#lab_i\ e', s'\rangle}$$

Mutable Store

Mutable Store

Most languages have some kind of mutable store. Two main choices:

1 What we've got in L1 and L2:

$$e \quad ::= \quad ... \mid \ell := e \mid !\ell \mid x$$

- locations store mutable values
- variables refer to a previously-calculated value, immutably
- explicit dereferencing and assignment operators for locations

  fn x:int ⇒ $l := (!l) + \mathrm{x}$

2 The C-way (also Java etc).

- variables let you refer to a previously calculated value *and* let you overwrite that value with another.
- implicit dereferencing and assignment,

```
void foo(x:int) {
l = l + x
...}
```

- have some limited type machinery (const qualifiers) to limit mutability.

– pros and cons: ....

References

Staying with 1 here. But, those L1/L2 references are very limited:

- can only store ints - for uniformity, would like to store any value
- cannot create new locations (all must exist at beginning)
- cannot write functions that abstract on locations **fn** l:intref $\Rightarrow !l$

So, generalise.

$$
\begin{array}{lll}
T & ::= & \ldots \mid T \text{ ref} \\
T_{loc} & ::= & \sout{\text{intref}}\ T \text{ ref} \\
e & ::= & \ldots \mid \sout{\ell := e} \mid \sout{!\ell} \\
 & & \mid e_1 := e_2 \mid !e \mid \text{ref } e \mid \ell
\end{array}
$$

Have locations in the expression syntax, but that is just so we can express the intermediate states of computations – whole programs now should have no locations in at the start, but can create them with ref. They can have variables of T ref type, e.g.**fn** x:int ref $\Rightarrow !$x.

References - Typing

$$\text{(ref)}\quad \frac{\Gamma \vdash e : T}{\Gamma \vdash \text{ref } e : T \text{ ref}}$$

$$\text{(assign)}\quad \frac{\Gamma \vdash e_1 : T \text{ ref} \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1 := e_2 : \text{unit}}$$

$$\text{(deref)}\quad \frac{\Gamma \vdash e : T \text{ ref}}{\Gamma \vdash !e : T}$$

$$\text{(loc)}\quad \frac{\Gamma(\ell) = T \text{ ref}}{\Gamma \vdash \ell : T \text{ ref}}$$

References – Reduction

A location is a value:

$$v \quad ::= \quad \ldots \mid \ell$$

Stores s were finite partial maps from $\mathbb{L}$ to $\mathbb{Z}$. From now on, take them to be finite partial maps from $\mathbb{L}$ to the set of all values.

(ref1) $\langle \text{ ref } v, s\rangle \longrightarrow \langle \ell, s + \{\ell \mapsto v\}\rangle \quad \ell \notin \text{dom}(s)$

(ref2) $$\frac{\langle e, s\rangle \longrightarrow \langle e', s'\rangle}{\langle \text{ ref } e, s\rangle \longrightarrow \langle \text{ ref } e', s'\rangle}$$

(deref1) $\langle !\ell, s\rangle \longrightarrow \langle v, s\rangle \quad$ if $\ell \in \text{dom}(s)$ and $s(\ell) = v$

(deref2) $$\frac{\langle e, s\rangle \longrightarrow \langle e', s'\rangle}{\langle !e, s\rangle \longrightarrow \langle !e', s'\rangle}$$

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

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

(assign3) $$\frac{\langle e, s\rangle \longrightarrow \langle e', s'\rangle}{\langle e := e_2, s\rangle \longrightarrow \langle e' := e_2, s'\rangle}$$

- A ref *has* to do something at runtime – (ref 0, ref 0) should return a pair of two new locations, each containing 0, not a pair of one location repeated.
- Note the typing and this dynamics permit locations to contain locations, e.g. ref(ref 3).
- This semantics no longer has determinacy, for a technical reason – new locations are chosen arbitrarily. At the cost of some slight semantic complexity, we could regain determinacy by working 'up to alpha for locations'.
- What *is* the store:
  1. an array of bytes,
  2. an array of values, or
  3. a partial function from locations to values?

  We take the third, most abstract option. Within the language one cannot do arithmetic on locations (just as well!) (can in C, can't in Java) or test whether one is bigger than another (in presence of garbage collection, they may not stay that way). Might or might not even be able to test them for equality (can in ML, cannot in L3).
- This store just grows during computation – an implementation can garbage collect (in many fancy ways), but platonic memory is free.

  We *don't* have an explicit deallocation operation – if you do, you need a very baroque type system to prevent dangling pointers being dereferenced. We don't have uninitialised locations (cf. null pointers), so don't have to worry about dereferencing null.

Type-checking the store

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

Now need more: for each $\ell \in \text{dom}(s)$ need that $s(\ell)$ is typable. Moreover, $s(\ell)$ might contain some other locations...

Type-checking the store – Example

Consider

$$\begin{array}{rcl} e & = & \textbf{let val } \mathrm{x}{:}(\mathsf{int} \to \mathsf{int})\ \mathsf{ref} = \mathsf{ref}(\textbf{fn } \mathrm{z}{:}\mathsf{int} \Rightarrow \mathrm{z}) \textbf{ in} \\ & & (\mathrm{x} := (\textbf{fn } \mathrm{z}{:}\mathsf{int} \Rightarrow \textbf{if } \mathrm{z} \geq 1 \textbf{ then } \mathrm{z} + ((!\mathrm{x})\ (\mathrm{z} + -1)) \textbf{ else } 0); \\ & & (!\mathrm{x})\ 3) \textbf{ end} \end{array}$$

which has reductions

$$\begin{array}{l} \langle e, \{\}\rangle \longrightarrow^* \\ \langle e_1, \{l_1 \mapsto (\textbf{fn } \mathrm{z}{:}\mathsf{int} \Rightarrow \mathrm{z})\}\rangle \longrightarrow^* \\ \langle e_2, \{l_1 \mapsto (\textbf{fn } \mathrm{z}{:}\mathsf{int} \Rightarrow \textbf{if } \mathrm{z} \geq 1 \textbf{ then } \mathrm{z} + ((!l_1)\ (\mathrm{z} + -1)) \textbf{ else } 0)\}\rangle \\ \longrightarrow^* \langle 6, \ldots\rangle \end{array}$$

For reference, e_1 and e_2 are

$$\begin{array}{rcl} e_1 & = & l_1 := (\textbf{fn } \mathrm{z}{:}\mathsf{int} \Rightarrow \textbf{if } \mathrm{z} \geq 1 \textbf{ then } \mathrm{z} + ((!l_1)\ (\mathrm{z} + -1)) \textbf{ else } 0); \\ & & ((!l_1)\ 3) \\ e_2 & = & \textbf{skip}; ((!l_1)\ 3) \end{array}$$

Have made a recursive function by 'tying the knot by hand', not using **let val rec** .

To do this we needed to store function values – couldn't do this in L2, so this doesn't contradict the normalisation theorem we had there.

So, say $\Gamma \vdash s$ if $\forall\, \ell \in \text{dom}(s).\exists\, T.\Gamma(\ell) = T\ \ \mathsf{ref}\wedge \Gamma \vdash s(\ell){:}T$.

The statement of type preservation will then be:

Theorem 14 (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 for some Γ' With disjoint domain to Γ We have $\Gamma, \Gamma' \vdash e'{:}T$ and $\Gamma, \Gamma' \vdash s'$.*

Implementation

The collected definition so far is in the notes, called L3.

It is again a Moscow ML fragment (modulo the syntax for $T + T$), so you can run programs. The Moscow ML record typing is more liberal that that of L3, though.

Evaluation Contexts

We end this chapter by showing a slightly different style for defining operational semantics, collecting together many of the *context rules* into a single (eval) rule that uses a definition of a set of *evaluation contexts* to describe where in your program the next step of reduction can take place. This style becomes much more convenient for large languages, though for L1 and L2 there's not much advantage either way.

Evaluation Contexts

Define *evaluation contexts*

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

and have the single *context* rule

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

replacing the rules (all those with ≥ 1 premise) (op1), (op2), (seq2), (if3), (app1), (app2), (let1), (pair1), (pair2), (proj3), (proj4), (inl), (inr), (case1), (record1), (record3), (ref2), (deref2), (assign2), (assign3).

To (eval) we add all the *computation* rules (all the rest) (op $+$), (op $\geq$), (seq1), (if1), (if2), (while), (fn), (let2), (letrecfn), (proj1), (proj2), (case2), (case3), (record2), (ref1), (deref1), (assign1).

Theorem 15 *The two definitions of* $\longrightarrow$ *define the same relation.*

L3: Collected Definition

L3 Syntax

Booleans $b \in \mathbb{B} = \{\textbf{true}, \textbf{false}\}$
Integers $n \in \mathbb{Z} = \{..., -1, 0, 1, ...\}$
Locations $\ell \in \mathbb{L} = \{l, l_0, l_1, l_2, ...\}$
Variables $x \in \mathbb{X}$ for a set $\mathbb{X} = \{\mathrm{x}, \mathrm{y}, \mathrm{z}, ...\}$
Labels $lab \in \mathbb{LAB}$ for a set $\mathbb{LAB} = \{\mathrm{p}, \mathrm{q}, ...\}$

Operations $op ::= + \mid \geq$

Types:

$$T \quad ::= \quad \mathsf{int} \mid \mathsf{bool} \mid \mathsf{unit} \mid T_1 \to T_2 \mid T_1 * T_2 \mid T_1 + T_2 \mid \{lab_1{:}T_1, .., lab_k{:}T_k\} \mid T \ \ \mathsf{ref}$$

Expressions

$$\begin{array}{rcl} e & ::= & n \mid b \mid e_1 \ \ op \ \ e_2 \mid \textbf{if} \ \ e_1 \ \ \textbf{then} \ \ e_2 \ \ \textbf{else} \ \ e_3 \mid \\ & & e_1 := e_2 \mid !e \mid \ \mathsf{ref} \ e \mid \ell \mid \\ & & \textbf{skip} \mid e_1; e_2 \mid \\ & & \textbf{while} \ \ e_1 \ \ \textbf{do} \ \ e_2 \mid \\ & & \textbf{fn} \ \ x{:}T \Rightarrow e \mid e_1 \ e_2 \mid x \mid \\ & & \textbf{let} \ \ \textbf{val} \ \ x{:}T = e_1 \ \ \textbf{in} \ \ e_2 \ \ \textbf{end} \mid \\ & & \textbf{let} \ \ \textbf{val} \ \ \textbf{rec} \ \ x{:}T_1 \to T_2 = (\textbf{fn} \ \ y{:}T_1 \Rightarrow e_1) \ \textbf{in} \ \ e_2 \ \ \textbf{end} \mid \\ & & (e_1, e_2) \mid \#1 \ e \mid \#2 \ e \mid \\ & & \textbf{inl} \ \ e{:}T \mid \textbf{inr} \ \ e{:}T \mid \\ & & \textbf{case} \ \ e \ \ \textbf{of} \ \ \textbf{inl} \ (x_1{:}T_1) \Rightarrow e_1 \mid \textbf{inr} \ (x_2{:}T_2) \Rightarrow e_2 \mid \\ & & \{lab_1 = e_1, .., lab_k = e_k\} \mid \#lab \ e \end{array}$$

(where in each record (type or expression) no *lab* occurs more than once)

In expressions **fn** $x{:}T \Rightarrow e$ the x is a *binder*. In expressions **let** **val** $x{:}T = e_1$ **in** e_2 **end** the x is a binder. In expressions **let** **val** **rec** $x{:}T_1 \to T_2 = (\textbf{fn} \ \ y{:}T_1 \Rightarrow e_1)$ **in** e_2 **end** the y binds in e_1; the x binds in $(\textbf{fn} \ \ y{:}T \Rightarrow e_1)$ and in e_2. In **case** e **of** **inl** $(x_1{:}T_1) \Rightarrow e_1$ | **inr** $(x_2{:}T_2) \Rightarrow e_2$ the x_1 binds in e_1 and the x_2 binds in e_2.

L3 Semantics

Stores s were finite partial maps from $\mathbb{L}$ to $\mathbb{Z}$. From now on, take them to be finite partial maps from $\mathbb{L}$ to the set of all values.

Values $v ::= b \mid n \mid \textbf{skip} \mid \textbf{fn} \ \ x{:}T \Rightarrow e \mid (v_1, v_2) \mid \textbf{inl} \ \ v{:}T \mid \textbf{inr} \ \ v{:}T \mid \{lab_1 = v_1, .., lab_k = v_k\} \mid \ell$

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

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

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

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

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

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

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

(if2) $\langle \textbf{if}\ \textbf{false}\ \textbf{then}\ e_2\ \textbf{else}\ e_3, s\rangle \longrightarrow \langle e_3, s\rangle$

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

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

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

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

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

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

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

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

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

(pair2) $$\frac{\langle e_2, s\rangle \longrightarrow \langle e_2', s'\rangle}{\langle (v_1, e_2), s\rangle \longrightarrow \langle (v_1, e_2'), s'\rangle}$$

(proj1) $\langle \#1(v_1, v_2), s\rangle \longrightarrow \langle v_1, s\rangle$ (proj2) $\langle \#2(v_1, v_2), s\rangle \longrightarrow \langle v_2, s\rangle$

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

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

(case1) $$\frac{\langle e, s\rangle \longrightarrow \langle e', s'\rangle}{\begin{array}{l}\langle \textbf{case}\ e\ \textbf{of}\ \textbf{inl}\ (x{:}T_1) \Rightarrow e_1 \mid \textbf{inr}\ (y{:}T_2) \Rightarrow e_2, s\rangle \\ \longrightarrow \langle \textbf{case}\ e'\ \textbf{of}\ \textbf{inl}\ (x{:}T_1) \Rightarrow e_1 \mid \textbf{inr}\ (y{:}T_2) \Rightarrow e_2, s'\rangle\end{array}}$$

(case2) $\langle \textbf{case}\ \textbf{inl}\ v{:}T\ \textbf{of}\ \textbf{inl}\ (x{:}T_1) \Rightarrow e_1 \mid \textbf{inr}\ (y{:}T_2) \Rightarrow e_2, s\rangle$
$\longrightarrow \langle \{v/x\}e_1, s\rangle$
(inr) and (case3) like (inl) and (case2)

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

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

$$\text{(record1)}\quad \frac{\langle e_i, s\rangle \longrightarrow \langle e_i', s'\rangle}{\langle \{lab_1 = v_1, .., lab_i = e_i, .., lab_k = e_k\}, s\rangle \longrightarrow \langle \{lab_1 = v_1, .., lab_i = e_i', .., lab_k = e_k\}, s'\rangle}$$

$$\text{(record2)}\quad \langle \#lab_i\ \{lab_1 = v_1, .., lab_k = v_k\}, s\rangle \longrightarrow \langle v_i, s\rangle$$

$$\text{(record3)}\quad \frac{\langle e, s\rangle \longrightarrow \langle e', s'\rangle}{\langle \#lab_i\ e, s\rangle \longrightarrow \langle \#lab_i\ e', s'\rangle}$$

$$\text{(ref1)}\quad \langle\ \textsf{ref}\ v, s\rangle \longrightarrow \langle \ell, s + \{\ell \mapsto v\}\rangle \quad \ell \notin \text{dom}(s)$$

$$\text{(ref2)}\quad \frac{\langle e, s\rangle \longrightarrow \langle e', s'\rangle}{\langle\ \textsf{ref}\ e, s\rangle \longrightarrow \langle\ \textsf{ref}\ e', s'\rangle}$$

$$\text{(deref1)}\quad \langle !\ell, s\rangle \longrightarrow \langle v, s\rangle \quad \text{if } \ell \in \text{dom}(s) \text{ and } s(\ell) = v$$

$$\text{(deref2)}\quad \frac{\langle e, s\rangle \longrightarrow \langle e', s'\rangle}{\langle !e, s\rangle \longrightarrow \langle !e', s'\rangle}$$

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

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

$$\text{(assign3)}\quad \frac{\langle e, s\rangle \longrightarrow \langle e', s'\rangle}{\langle e := e_2, s\rangle \longrightarrow \langle e' := e_2, s'\rangle}$$

L3 Typing

Take $\Gamma \in$ TypeEnv2, the finite partial functions from $\mathbb{L} \cup \mathbb{X}$ to $\mathrm{T}_{\mathrm{loc}} \cup \mathrm{T}$ such that

$\forall\, \ell \in \text{dom}(\Gamma).\Gamma(\ell) \in \mathrm{T}_{\mathrm{loc}}$

$\forall\, x \in \text{dom}(\Gamma).\Gamma(x) \in \mathrm{T}$

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

$$\text{(bool)}\quad \Gamma \vdash b{:}\textsf{bool} \quad \text{for } b \in \{\textbf{true}, \textbf{false}\}$$

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

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

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

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

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

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

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

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

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

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

$$\text{(pair)}\quad \frac{\Gamma \vdash e_1:T_1 \qquad \Gamma \vdash e_2:T_2}{\Gamma \vdash (e_1, e_2):T_1 * T_2}$$

$$\text{(proj1)}\quad \frac{\Gamma \vdash e:T_1 * T_2}{\Gamma \vdash \#1\ e:T_1}$$

$$\text{(proj2)}\quad \frac{\Gamma \vdash e:T_1 * T_2}{\Gamma \vdash \#2\ e:T_2}$$

$$\text{(inl)}\quad \frac{\Gamma \vdash e:T_1}{\Gamma \vdash \textbf{inl}\ \ e:T_1 + T_2:T_1 + T_2}$$

$$\text{(inr)}\quad \frac{\Gamma \vdash e:T_2}{\Gamma \vdash \textbf{inr}\ \ e:T_1 + T_2:T_1 + T_2}$$

$$\text{(case)}\quad \frac{\Gamma \vdash e:T_1 + T_2 \quad \Gamma, x:T_1 \vdash e_1:T \quad \Gamma, y:T_2 \vdash e_2:T}{\Gamma \vdash \textbf{case}\ \ e\ \ \textbf{of}\ \ \textbf{inl}\ (x:T_1) \Rightarrow e_1 \mid \textbf{inr}\ (y:T_2) \Rightarrow e_2:T}$$

$$\text{(record)}\quad \frac{\Gamma \vdash e_1:T_1 \quad .. \quad \Gamma \vdash e_k:T_k}{\Gamma \vdash \{lab_1 = e_1, .., lab_k = e_k\}:\{lab_1:T_1, .., lab_k:T_k\}}$$

$$\text{(recordproj)}\quad \frac{\Gamma \vdash e:\{lab_1:T_1, .., lab_k:T_k\}}{\Gamma \vdash \#lab_i\ e:T_i}$$

$$\text{(ref)} \quad \frac{\Gamma \vdash e : T}{\Gamma \vdash \mathsf{ref}\ e : T\ \mathsf{ref}}$$

$$\text{(assign)} \quad \frac{\Gamma \vdash e_1 : T\ \mathsf{ref} \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1 := e_2 : \mathsf{unit}}$$

$$\text{(deref)} \quad \frac{\Gamma \vdash e : T\ \mathsf{ref}}{\Gamma \vdash !e : T}$$

$$\text{(loc)} \quad \frac{\Gamma(\ell) = T\ \mathsf{ref}}{\Gamma \vdash \ell : T\ \mathsf{ref}}$$