Semantics of Programming Languages

Computer Science Tripos, Part 1B

2019–20

Thursday 10 October	1	Thursday 31 October	7
Tuesday 15 October	2	Tuesday 5 November	8
Thursday 17 October	3	Thursday 7 November	9
Tuesday 22 October	4	Tuesday 12 November	10
Thursday 24 October	5	Thursday 14 November	11
Tuesday 29 October	6	Tuesday 19 November	12

- Science
- Engineering
- Craft
- Art
- Bodgery

FIG. 5. AN ILLUSTRATION OF WHAT EXPLOSION DID TO STAYS AND BRACES

Programming languages: basic engineering tools of our time

Semantics — What is it?

How to describe a programming language? Need to give:

- the *syntax* of programs; and
- their *semantics* (the meaning of programs, or how they behave).

Semantics — What is it?

How to describe a programming language? Need to give:

- the *syntax* of programs; and
- their *semantics* (the meaning of programs, or how they behave).

Styles of description:

- the language is defined by whatever some particular compiler does
- natural language ‘definitions’
- mathematically

Mathematical descriptions of syntax use formal grammars (eg BNF) – precise, concise, clear. In this course we’ll see how to work with mathematical definitions of semantics/behaviour.

What do we use semantics for?

1. to understand a particular language — what you can depend on as a programmer; what you must provide as a compiler writer
2. as a tool for language design:
   (a) for clean design
   (b) for expressing design choices, understanding language features and how they interact.
   (c) for proving properties of a language, eg type safety, decidability of type inference.
3. as a foundation for proving properties of particular programs

Design choices, from Micro to Macro

- basic values
- evaluation order
- what can be stored
- what can be abstracted over
- what is guaranteed at compile-time and run-time
- how effects are controlled
- how concurrency is supported
- how information hiding is enforceable
- how large-scale development and re-use are supported
- ...

Warmup

In C, if initially `x` has value `3`, what's the value of the following?

```
x++ + x++ + x++ + x++
```

C#

```
delegate int IntThunk();

class M {
  public static void Main() {
    IntThunk[] funcs = new IntThunk[11];
    for (int i = 0; i <= 10; i++)
    {
      funcs[i] = delegate() { return i; };
    }
    foreach (IntThunk f in funcs)
    {
      System.Console.WriteLine(f());
    }
  }
}
```

Output:

```
11
11
11
11
11
11
11
11
11
11
```

JavaScript

```
function bar(x) {
  return function() {
    var x = x;
    return x;
  };
}

var f = bar(200);

f()
```

Styles of Semantic Definitions

- Operational semantics
- Denotational semantics
- Axiomatic, or Logical, semantics

‘Toy’ languages

Real programming languages are large, with many features and, often, with redundant constructs – things that can be expressed in the rest of the language.

When trying to understand some particular combination of features it’s usual to define a small ‘toy’ language with just what you’re interested in, then scale up later. Even small languages can involve delicate design choices.

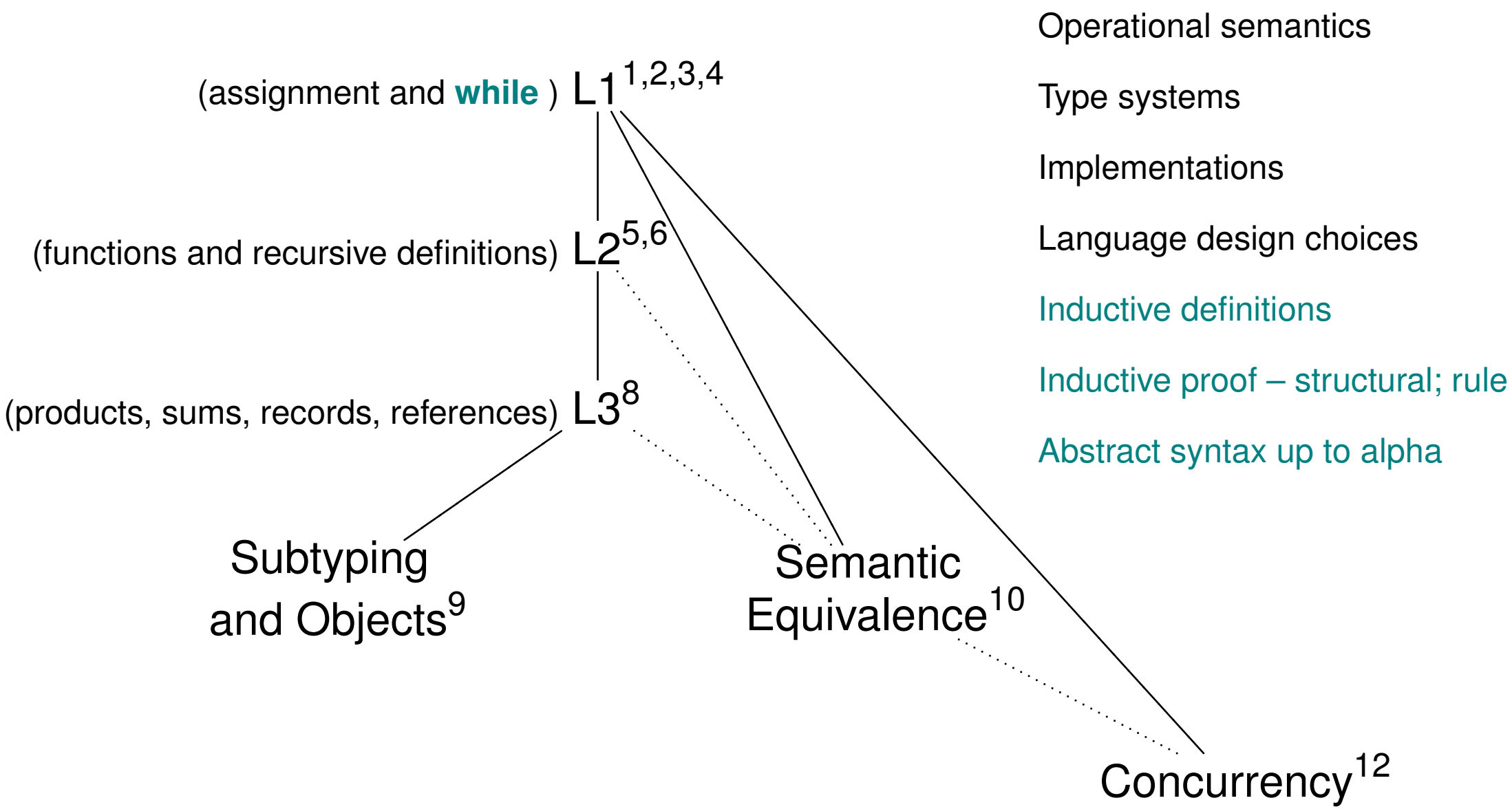
What's this course?

Core

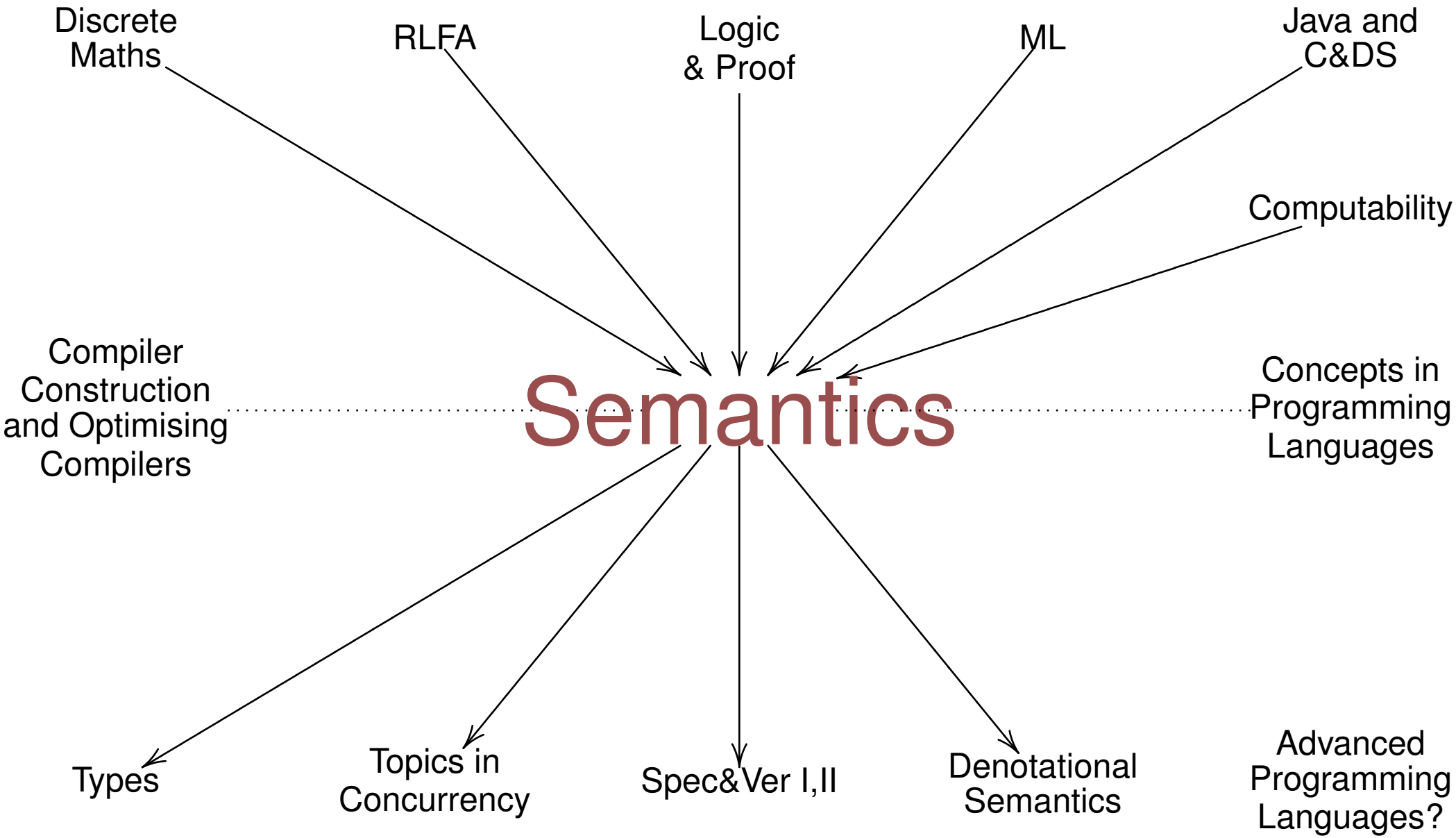
- operational semantics and typing for a tiny language
- technical tools (abstract syntax, inductive definitions, proof)
- design for functions, data and references

More advanced topics

- Subtyping and Objects
- Semantic Equivalence
- Concurrency


(assignment and **while**) L1[1,2,3,4]
(functions and recursive definitions) L2[5,6]
(products, sums, records, references) L3[8]
Subtyping and Objects[9]
Semantic Equivalence[10]
Concurrency[12]
Operational semantics
Type systems
Implementations
Language design choices
Inductive definitions
Inductive proof – structural; rule
Abstract syntax up to alpha

The Big Picture



Admin

- Please let me know of typos, and if it is too fast/too slow/too interesting/too dull (please complete the on-line feedback at the end)
- Exercises in the notes.
- Implementations on web.
- Books (Harper, Hennessy, Pierce, Winskel)

L1

L1 – Example

L1 is an imperative language with store locations (holding integers), conditionals, and **while** loops. For example, consider the program

$$
\begin{array}{l}
l_2 := 0; \\
\textbf{while}\ !l_1 \geq 1\ \ \textbf{do}\ (\\
\quad l_2 :=!l_2 + !l_1; \\
\quad l_1 :=!l_1 + -1)
\end{array}
$$

in the initial store $\{l_1 \mapsto 3, l_2 \mapsto 0\}$.

L1 – Syntax

Booleans $b \in \mathbb{B} = \{\textbf{true}, \textbf{false}\}$

Integers $n \in \mathbb{Z} = \{..., -1, 0, 1, ...\}$

Locations $\ell \in \mathbb{L} = \{l, l_0, l_1, l_2, ...\}$

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

Expressions

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

Write L_1 for the set of all expressions.

Transition systems

A *transition system* consists of

- a set Config, and
- a binary relation $\longrightarrow \subseteq \mathrm{Config} * \mathrm{Config}$.

The elements of Config are often called *configurations* or *states*. The relation $\longrightarrow$ is called the *transition* or *reduction* relation. We write $\longrightarrow$ infix, so $c \longrightarrow c'$ should be read as 'state c can make a transition to state c''.

L1 Semantics (1 of 4) – Configurations

Say *stores* s are finite partial functions from $\mathbb{L}$ to $\mathbb{Z}$. For example:

$$\{l_1 \mapsto 7,\ l_3 \mapsto 23\}$$

Take *configurations* to be pairs $\langle e, s \rangle$ of an expression e and a store s, so our transition relation will have the form

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

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

$$\begin{aligned}
& \langle l := 2 + !l, \quad \{l \mapsto 3\} \rangle \\
\longrightarrow \quad & \langle l := 2 + 3, \quad \{l \mapsto 3\} \rangle \\
\longrightarrow \quad & \langle l := 5, \quad \{l \mapsto 3\} \rangle \\
\longrightarrow \quad & \langle \textbf{skip}, \quad \{l \mapsto 5\} \rangle \\
\not\longrightarrow &
\end{aligned}$$

want to keep on until we get to a *value* v, an expression in

$$\mathbb{V} = \mathbb{B} \cup \mathbb{Z} \cup \{\textbf{skip}\}.$$

Say $\langle e, s \rangle$ is *stuck* if e is not a value and $\langle e, s \rangle \not\longrightarrow$. For example $2 + \textbf{true}$ will be stuck.

L1 Semantics (2 of 4) – Rules (basic operations)

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

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

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

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

Example

If we want to find the possible sequences of transitions of $\langle (2+3)+(6+7), \emptyset \rangle$ ... look for derivations of transitions.

(you might think the answer *should be* 18 – but we want to know what *this definition* says happens)

$$\text{(op1)}\ \frac{\text{(op }+\text{)}\ \dfrac{}{\langle 2+3, \emptyset \rangle \longrightarrow \langle 5, \emptyset \rangle}}{\langle (2+3)+(6+7), \emptyset \rangle \longrightarrow \langle 5+(6+7), \emptyset \rangle}$$

$$\text{(op2)}\ \frac{\text{(op }+\text{)}\ \dfrac{}{\langle 6+7, \emptyset \rangle \longrightarrow \langle 13, \emptyset \rangle}}{\langle 5+(6+7), \emptyset \rangle \longrightarrow \langle 5+13, \emptyset \rangle}$$

$$\text{(op }+\text{)}\ \frac{}{\langle 5+13, \emptyset \rangle \longrightarrow \langle 18, \emptyset \rangle}$$

L1 Semantics (3 of 4) – store and sequencing

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

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

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

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

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

Example

$$\begin{aligned}
\langle l := 3; !l, \{l \mapsto 0\}\rangle &\longrightarrow \langle \textbf{skip}; !l, \{l \mapsto 3\}\rangle \\
&\longrightarrow \langle !l, \{l \mapsto 3\}\rangle \\
&\longrightarrow \langle 3, \{l \mapsto 3\}\rangle
\end{aligned}$$

$$\langle l := 3; l :=!l, \{l \mapsto 0\}\rangle \longrightarrow ?$$

$$\langle 15+!l, \emptyset\rangle \longrightarrow ?$$

L1 Semantics (4 of 4) – The rest (conditionals and while)

(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)

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

Example

If

$$e = (l_2 := 0; \textbf{while } !l_1 \geq 1 \textbf{ do } (l_2 :=!l_2+!l_1; l_1 :=!l_1 + -1))$$

$$s = \{l_1 \mapsto 3, l_2 \mapsto 0\}$$

then

$$\langle e, s \rangle \longrightarrow^* ?$$

L1: Collected Definition

Syntax

Booleans $b \in \mathbb{B} = \{\textbf{true}, \textbf{false}\}$
Integers $n \in \mathbb{Z} = \{..., -1, 0, 1, ...\}$
Locations $\ell \in \mathbb{L} = \{l, l_0, l_1, l_2, ...\}$
Operations $op ::= + \mid \geq$
Expressions

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

Operational Semantics

Note that for each construct there are some *computation* rules, doing 'real work', and some *context* (or *congruence*) rules, allowing subcomputations and specifying their order.
Stores s are finite partial functions from $\mathbb{L}$ to $\mathbb{Z}$. *Values* v are expressions from the grammar $v ::= b \mid n \mid \textbf{skip}$.

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

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

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

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

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

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

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

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

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

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

Determinacy

Theorem 1 (L1 Determinacy) *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* $\langle e_1, s_1\rangle = \langle e_2, s_2\rangle$.

Proof – see later

L1 implementation

Many possible implementation strategies, including:

1. animate the rules — use unification to try to match rule conclusion left-hand-sides against a configuration; use backtracking search to find all possible transitions. Hand-coded, or in Prolog/LambdaProlog/Twelf.

2. write an interpreter working directly over the syntax of configurations. Coming up, in ML and Java.

3. compile to a stack-based virtual machine, and an interpreter for that. See Compiler Construction.

4. compile to assembly language, dealing with register allocation etc. etc. See Compiler Construction/Optimizing Compilers.

L1 implementation

Will implement an interpreter for L1, following the definition. Use mosml (Moscow ML) as the implementation language, as datatypes and pattern matching are good for this kind of thing.

First, must pick representations for locations, stores, and expressions:

```
type loc = string

type store = (loc * int) list
```

```
datatype oper = Plus | GTEQ


datatype expr =
    Integer of int
  | Boolean of bool
  | Op of expr * oper * expr
  | If of expr * expr * expr
  | Assign of loc * expr
  | Deref of loc
  | Skip
  | Seq of expr * expr
  | While of expr * expr
```

Store operations

Define auxiliary operations

```
lookup : store*loc -> int option
```

```
update : store*(loc*int) -> store option
```

which both return `NONE` if given a location that is not in the domain of the store. Recall that a value of type `T option` is either `NONE` or `SOME v` for a value `v` of `T`.

The single-step function

Now define the single-step function

```
reduce : expr*store -> (expr*store) option
```

which takes a configuration `(e,s)` and returns either

`NONE`, if $\langle e, s\rangle \not\longrightarrow$,

or `SOME (e',s')`, if it has a transition $\langle e, s\rangle \longrightarrow \langle e', s'\rangle$.

Note that if the semantics didn't define a deterministic transition system we'd have to be more elaborate.

(op $+$), (op $\geq$)

```
fun reduce (Integer n,s) = NONE
  | reduce (Boolean b,s) = NONE
  | reduce (Op (e1,opr,e2),s) =
    (case (e1,opr,e2) of
          (Integer n1, Plus, Integer n2) =>
            SOME(Integer (n1+n2), s)
        | (Integer n1, GTEQ, Integer n2) =>
            SOME(Boolean (n1 >= n2), s)
        | (e1,opr,e2) =>
            ...
```

(op1), (op2)

```
...
if (is_value e1) then
    case reduce (e2,s) of
        SOME (e2',s') =>
          SOME (Op(e1,opr,e2'),s')
      | NONE => NONE
else
    case reduce (e1,s) of
        SOME (e1',s') =>
          SOME(Op(e1',opr,e2),s')
      | NONE => NONE )
```

(assign1), (assign2)

```
| reduce (Assign (l,e),s) =
  (case e of
        Integer n =>
          (case update (s,(l,n)) of
              SOME s' => SOME(Skip, s')
            | NONE => NONE)
      | _ =>
          (case reduce (e,s) of
              SOME (e',s') =>
                SOME(Assign (l,e'), s')
            | NONE => NONE ) )
```

The many-step evaluation function

Now define the many-step evaluation function

```
evaluate: expr*store -> (expr*store) option
```

which takes a configuration `(e,s)` and returns the `(e',s')` such that $\langle e, s\rangle \longrightarrow^* \langle e', s'\rangle \not\longrightarrow$, if there is such, or does not return.

```
fun evaluate (e,s) =
  case reduce (e,s) of
    NONE => (e,s)
  | SOME (e',s') => evaluate (e',s')
```

Demo

The Java Implementation

Quite different code structure:

- the ML groups together all the parts of each algorithm, into the `reduce`, `infertype`, and `prettyprint` functions;
- the Java groups together everything to do with each clause of the abstract syntax, in the `IfThenElse`, `Assign`, etc. classes.

Language design 1. Order of evaluation

For $(e_1 \quad op \quad e_2)$, the rules above say e_1 should be fully reduced, to a value, before we start reducing e_2. For example:

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

For right-to-left evaluation, replace (op1) and (op2) by

(op1b) $$\frac{\langle e_2, s \rangle \longrightarrow \langle e_2', s' \rangle}{\langle e_1 \quad op \quad e_2, s \rangle \longrightarrow \langle e_1 \quad op \quad e_2', s' \rangle}$$

(op2b) $$\frac{\langle e_1, s \rangle \longrightarrow \langle e_1', s' \rangle}{\langle e_1 \quad op \quad v, s \rangle \longrightarrow \langle e_1' \quad op \quad v, s' \rangle}$$

In this language (call it L1b)

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

Language design 2. Assignment results

Recall

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

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

So

$$\begin{aligned} \langle l := 1; l := 2, \{l \mapsto 0\} \rangle &\longrightarrow \langle \textbf{skip}; l := 2, \{l \mapsto 1\} \rangle \\ &\longrightarrow^* \langle \textbf{skip}, \{l \mapsto 2\} \rangle \end{aligned}$$

We've chosen $\ell := n$ to result in skip, and $e_1; e_2$ to only progress if $e_1 = \textbf{skip}$, not for any value. Instead could have this:

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

(seq1') $\langle v; e_2, s \rangle \longrightarrow \langle e_2, s \rangle$

Language design 3. Store initialization

Recall that

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

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

both require $\ell \in \mathrm{dom}(s)$, otherwise the expressions are stuck.

Instead, could

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

Language design 4. Storable values

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

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

(assign1) $\langle \ell := n, s\rangle \longrightarrow \langle \mathbf{skip}, s + \{\ell \mapsto n\}\rangle$ if $\ell \in \mathrm{dom}(s)$

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

Can store only integers. $\langle l := \mathbf{true}, s\rangle$ is stuck.

Why not allow storage of any value? of locations? of programs?

Also, store is global. We will consider programs that can create new locations later.

Language design 5. Operators and basic values

Booleans are really not integers (unlike in C)

The L1 impl and semantics aren't quite in step.

Exercise: fix the implementation to match the semantics.

Exercise: fix the semantics to match the implementation.

Expressiveness

Is L1 expressive enough to write interesting programs?

- yes: it's Turing-powerful (try coding an arbitrary register machine in L1).
- no: there's no support for gadgets like functions, objects, lists, trees, modules,.....

Is L1 *too* expressive? (ie, can we write too many programs in it)

- yes: we'd like to forbid programs like $3 + \textbf{false}$ as early as possible, rather than let the program get stuck or give a runtime error. We'll do so with a *type system*.

L1 Typing

Type systems

used for

- describing when programs make sense
- preventing certain kinds of errors
- structuring programs
- guiding language design

Ideally, **well-typed programs don't get stuck**.

Run-time errors

Trapped errors. Cause execution to halt immediately. (E.g. jumping to an illegal address, raising a top-level exception, etc.) Innocuous?

Untrapped errors. May go unnoticed for a while and later cause arbitrary behaviour. (E.g. accessing data past the end of an array, security loopholes in Java abstract machines, etc.) Insidious!

Given a precise definition of what constitutes an untrapped run-time error, then a language is *safe* if all its syntactically legal programs cannot cause such errors.

Usually, safety is desirable. Moreover, we'd like as few trapped errors as possible.

Formal type systems

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):

$$\{\} \quad \vdash \quad \textbf{if true then } 2 \textbf{ else } 3+4 \quad : \quad \mathsf{int}$$

$$l_1 : \mathsf{intref} \quad \vdash \quad \textbf{if } !l_1 \geq 3 \textbf{ then } !l_1 \textbf{ else } 3 \quad : \quad \mathsf{int}$$

$$\{\} \quad \not\vdash \quad 3 + \textbf{false} \quad : \quad T \quad \text{for any } T$$

$$\{\} \quad \not\vdash \quad \textbf{if true then } 3 \textbf{ else false} \quad : \quad \mathsf{int}$$

Types for L1

Types of expressions:

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

Types of locations:

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

Write T and $\mathrm{T_{loc}}$ for the sets of all terms of these grammars.

Let Γ range over $\mathrm{TypeEnv}$, the finite partial functions from locations $\mathbb{L}$ to $\mathrm{T_{loc}}$. Notation: write a Γ as $l_1{:}\mathsf{intref}, \ldots, l_k{:}\mathsf{intref}$ instead of $\{l_1 \mapsto \mathsf{intref}, \ldots, l_k \mapsto \mathsf{intref}\}$.

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

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

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

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

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

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

Example

To show $\{\} \vdash \textbf{if}\ \ \textbf{false}\ \ \textbf{then}\ \ 2\ \ \textbf{else}\ \ 3+4 : \mathsf{int}$ we can give a type derivation like this:

$$\text{(if)}\ \frac{\text{(bool)}\ \dfrac{}{\{\} \vdash \textbf{false} : \mathsf{bool}} \qquad \text{(int)}\ \dfrac{}{\{\} \vdash 2 : \mathsf{int}} \qquad \nabla}{\{\} \vdash \textbf{if}\ \ \textbf{false}\ \ \textbf{then}\ \ 2\ \ \textbf{else}\ \ 3+4 : \mathsf{int}}$$

where ∇ is

Example

To show $\{\} \vdash \textbf{if}\ \ \textbf{false}\ \ \textbf{then}\ \ 2\ \ \textbf{else}\ \ 3+4 : \mathsf{int}$ we can give a type derivation like this:

$$\text{(if)}\ \frac{\text{(bool)}\ \dfrac{}{\{\} \vdash \textbf{false} : \mathsf{bool}} \qquad \text{(int)}\ \dfrac{}{\{\} \vdash 2 : \mathsf{int}} \qquad \nabla}{\{\} \vdash \textbf{if}\ \ \textbf{false}\ \ \textbf{then}\ \ 2\ \ \textbf{else}\ \ 3+4 : \mathsf{int}}$$

where ∇ is

$$\text{(op } +\text{)}\ \frac{\text{(int)}\ \dfrac{}{\{\} \vdash 3 : \mathsf{int}} \qquad \text{(int)}\ \dfrac{}{\{\} \vdash 4 : \mathsf{int}}}{\{\} \vdash 3+4 : \mathsf{int}}$$

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

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

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

Defining the type judgement $\boxed{\Gamma \vdash e : T}$ (3 of 3)

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

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

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

Properties

Theorem 2 (Progress) *If* $\Gamma \vdash e : T$ *and* $dom(\Gamma) \subseteq dom(s)$ *then either* e *is a value or there exist* e', s' *such that* $\langle e, s\rangle \longrightarrow \langle e', s'\rangle$.

Theorem 3 (Type Preservation) *If* $\Gamma \vdash e : T$ *and* $dom(\Gamma) \subseteq dom(s)$ *and* $\langle e, s\rangle \longrightarrow \langle e', s'\rangle$ *then* $\Gamma \vdash e' : T$ *and* $dom(\Gamma) \subseteq dom(s')$.

From these two we have that well-typed programs don't get stuck:

Theorem 4 (Safety) *If* $\Gamma \vdash e : T$, $dom(\Gamma) \subseteq dom(s)$, *and* $\langle e, s\rangle \longrightarrow^* \langle e', s'\rangle$ *then either* e' *is a value or there exist* e'', s'' *such that* $\langle e', s'\rangle \longrightarrow \langle e'', s''\rangle$.

Type checking, typeability, and type inference

Type checking problem for a type system: given Γ, e, T, is $\Gamma \vdash e{:}T$ derivable?

Type inference problem: given Γ and e, find T such that $\Gamma \vdash e{:}T$ is derivable, or show there is none.

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

For this type system, though, both are easy.

More Properties

Theorem 5 (Type inference) *Given* Γ, e*, one can find* T *such that* $\Gamma \vdash e : T$*, or show that there is none.*

Theorem 6 (Decidability of type checking) *Given* Γ, e, T*, one can decide* $\Gamma \vdash e : T$*.*

Also:

Theorem 7 (Uniqueness of typing) *If* $\Gamma \vdash e : T$ *and* $\Gamma \vdash e : T'$ *then* $T = T'$*.*

Type inference – Implementation

First must pick representations for types and for Γ's:

```
datatype type_L1 =
    int
  | unit
  | bool


datatype type_loc =
    intref


type typeEnv = (loc*type_loc) list
```

Now define the type inference function

```
infertype :  typeEnv -> expr -> type_L1 option
```

The Type Inference Algorithm

```
fun infertype gamma (Integer n) = SOME int
  | infertype gamma (Boolean b) = SOME bool
  | infertype gamma (Op (e1,opr,e2))
    = (case (infertype gamma e1, opr, infertype gamma e2) of
           (SOME int, Plus, SOME int) => SOME int
         | (SOME int, GTEQ, SOME int) => SOME bool
         | _ => NONE)
  | infertype gamma (If (e1,e2,e3))
    = (case (infertype gamma e1, infertype gamma e2, infertype gamma e3) of
           (SOME bool, SOME t2, SOME t3) =>
           if t2=t3 then SOME t2 else NONE
         | _ => NONE)
  | infertype gamma (Deref l)
    = (case lookup (gamma,l) of
           SOME intref => SOME int
         | NONE => NONE)
  | infertype gamma (Assign (l,e))
    = (case (lookup (gamma,l), infertype gamma e) of
           (SOME intref,SOME int) => SOME unit
         | _ => NONE)
  | infertype gamma (Skip) = SOME unit
  | infertype gamma (Seq (e1,e2))
    = (case (infertype gamma e1, infertype gamma e2) of
            (SOME unit, SOME t2) => SOME t2
          | _ => NONE )
  | infertype gamma (While (e1,e2))
    = (case (infertype gamma e1, infertype gamma e2) of
```

The Type Inference Algorithm – `If`

```
...
| infertype gamma (If (e1,e2,e3))
  = (case (infertype gamma e1,
           infertype gamma e2,
           infertype gamma e3) of
        (SOME bool, SOME t2, SOME t3) =>
          if t2=t3 then SOME t2 else NONE
      | _ => NONE)
```

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

The Type Inference Algorithm – `Deref`

```
...
| infertype gamma (Deref l)
  = (case lookup (gamma,l) of
        SOME intref => SOME int
      | NONE => NONE)
...
```

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

Demo

Executing L1 in Moscow ML

L1 is essentially a fragment of Moscow ML – given a typable L1 expression e and an initial store s, e can be executed in Moscow ML by wrapping it

```
let val skip = ()
    and l1 = ref n1
    and l2 = ref n2
    ..  .
    and lk = ref nk
in
  e
end;
```

where s is the store $\{l_1 \mapsto n_1, \ldots, l_k \mapsto n_k\}$ and all locations that occur in e are contained in $\{l_1, \ldots, l_k\}$.

Why *Not* Types?

- *"I can't write the code I want in this type system."*

  (the Pascal complaint) usually false for a modern typed language

- *"It's too tiresome to get the types right throughout development."*

  (the untyped-scripting-language complaint)

- *"Type annotations are too verbose."*

  type inference means you only have to write them where it's useful

- *"Type error messages are incomprehensible."*

  hmm. Sadly, sometimes true.

- *"I really can't write the code I want."*

Induction

We've stated several 'theorems', but how do we know they are true?

Intuition is often wrong – we need *proof*.

Use proof process also for strengthening our intuition about subtle language features, and for debugging definitions – it helps you examine all the various cases.

Most of our definitions are inductive. To prove things about them, we need the corresponding *induction principles*.

Three forms of induction

Prove facts about all natural numbers by *mathematical induction*.

Prove facts about all terms of a grammar (e.g. the L1 expressions) by *structural induction*.

Prove facts about all elements of a relation defined by rules (e.g. the L1 transition relation, or the L1 typing relation) by *rule induction*.

We shall see that all three boil down to induction over certain *trees*.

Principle of Mathematical Induction

For any property $\Phi(x)$ of natural numbers $x \in \mathbb{N} = \{0, 1, 2, ...\}$, to prove

$\forall\, x \in \mathbb{N}.\Phi(x)$

it's enough to prove

$\Phi(0)$ and $\forall\, x \in \mathbb{N}.\Phi(x) \Rightarrow \Phi(x+1)$.

i.e.

$$\boxed{\big(\Phi(0) \wedge (\forall\, x \in \mathbb{N}.\Phi(x) \Rightarrow \Phi(x+1))\big) \Rightarrow \forall\, x \in \mathbb{N}.\Phi(x)}$$

$$\boxed{\big(\Phi(0) \wedge (\forall\ x\ \in\ \mathbb{N}.\Phi(x) \Rightarrow \Phi(x+1))\big) \Rightarrow \forall\ x\ \in\ \mathbb{N}.\Phi(x)}$$

For example, to prove

Theorem 8 $1 + 2 + ... + x = 1/2 * x * (x + 1)$

use mathematical induction for

$\Phi(x) = (1 + 2 + ... + x = 1/2 * x * (x + 1))$

There's a model proof in the notes, as an example of good style. Writing a clear proof structure like this becomes essential when things get more complex – you have to *use* the formalism to help you get things right. Emulate it!

Abstract Syntax and Structural Induction

How to prove facts about all expressions, e.g. Determinacy for L1?

Theorem 1 (Determinacy) *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* $\langle e_1, s_1\rangle = \langle e_2, s_2\rangle$.

First, don't forget the elided universal quantifiers.

Theorem 1 (Determinacy) *For all* e, s, e_1, s_1, e_2, s_2*, 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* $\langle e_1, s_1\rangle = \langle e_2, s_2\rangle$.

Abstract Syntax

Then, have to pay attention to what an expression *is*.

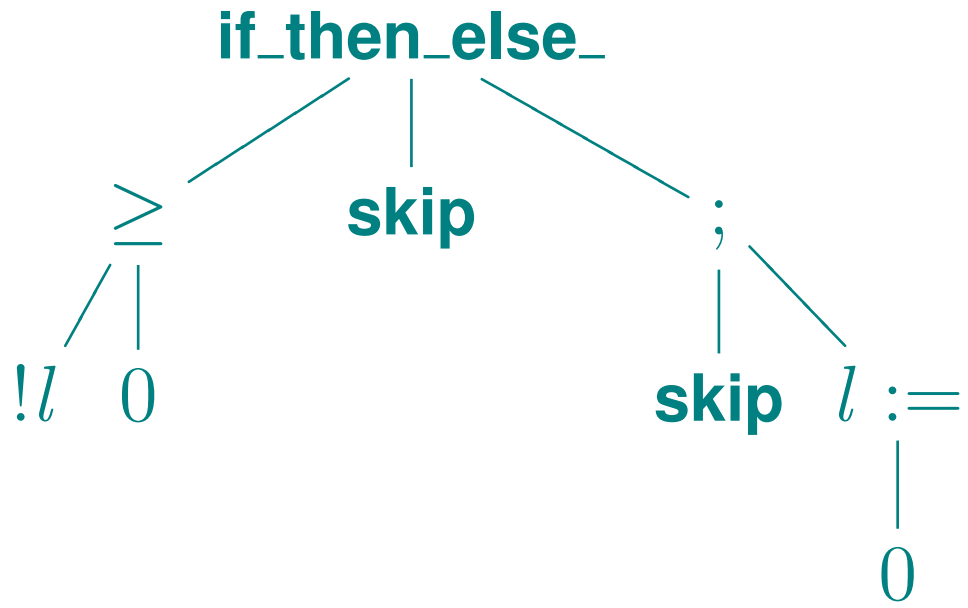
Recall we said:

$$
\begin{array}{rcl}
e & ::= & n \mid b \mid e \;\; op \;\; e \mid \textbf{if} \;\; e \;\; \textbf{then} \;\; e \;\; \textbf{else} \;\; e \mid \\
& & \ell := e \mid !\ell \mid \\
& & \textbf{skip} \mid e; e \mid \\
& & \textbf{while} \;\; e \;\; \textbf{do} \;\; e
\end{array}
$$

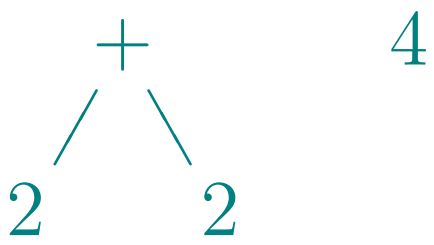
defining a set of expressions.

Q: Is an expression, e.g. **if** $!l \geq 0$ **then** **skip** **else** $(\textbf{skip}; l := 0)$:

1. a list of characters `[‘i’, ‘f’, ‘_’, ‘!’, ‘l’, ..]`;
2. a list of tokens `[ IF, DEREF, LOC "l", GTEQ, ..]`; or
3. an abstract syntax tree?

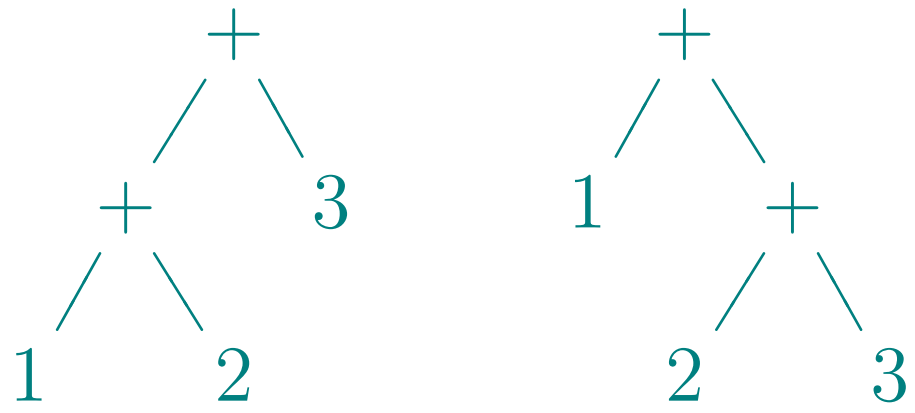


A: an abstract syntax tree. Hence: $2 + 2 \neq 4$



$1 + 2 + 3$ – ambiguous

$(1 + 2) + 3 \neq 1 + (2 + 3)$



Parentheses are only used for disambiguation – they are not part of the grammar. $1 + 2 = (1 + 2) = ((1 + 2)) = ((((1)))) + ((2)))$

Principle of Structural Induction (for abstract syntax)

For any property $\Phi(e)$ of expressions e, to prove

$$\forall\ e\ \in\ L_1.\Phi(e)$$

it's enough to prove for each tree constructor c (taking $k \geq 0$ arguments) that if Φ holds for the subtrees $e_1, .., e_k$ then Φ holds for the tree $c(e_1, .., e_k)$. i.e.

$$\big(\forall\ c.\forall\ e_1, .., e_k.(\Phi(e_1) \wedge ... \wedge \Phi(e_k)) \Rightarrow \Phi(c(e_1, .., e_k))\big) \Rightarrow \forall\ e.\Phi(e)$$

where the tree constructors (or node labels) c are n, **true**, **false**, $!l$, **skip**, $l :=$, **while_do_**, **if_then_else_**, etc.

In particular, for L1: to show $\forall\ e\ \in\ L_1.\Phi(e)$ it's enough to show:

nullary: $\Phi(\textbf{skip})$

$\forall\ b\ \in \{\textbf{true}, \textbf{false}\}.\Phi(b)$

$\forall\ n\ \in\ \mathbb{Z}.\Phi(n)$

$\forall\ \ell\ \in\ \mathbb{L}.\Phi(!\ell)$

unary: $\forall\ \ell\ \in\ \mathbb{L}.\forall\ e.\Phi(e) \Rightarrow \Phi(\ell := e)$

binary: $\forall\ \ op\ .\forall\ e_1, e_2.(\Phi(e_1) \wedge \Phi(e_2)) \Rightarrow \Phi(e_1\ \ op\ \ e_2)$

$\forall\ e_1, e_2.(\Phi(e_1) \wedge \Phi(e_2)) \Rightarrow \Phi(e_1; e_2)$

$\forall\ e_1, e_2.(\Phi(e_1) \wedge \Phi(e_2)) \Rightarrow \Phi(\textbf{while}\ \ e_1\ \ \textbf{do}\ \ e_2)$

ternary: $\forall\ e_1, e_2, e_3.(\Phi(e_1) \wedge \Phi(e_2) \wedge \Phi(e_3)) \Rightarrow \Phi(\textbf{if}\ \ e_1\ \ \textbf{then}\ \ e_2\ \ \textbf{else}$

(See how this comes directly from the grammar)

Proving Determinacy (Outline)

Theorem 1 (Determinacy) *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* $\langle e_1, s_1\rangle = \langle e_2, s_2\rangle$ *.*

Take

$$\Phi(e) \stackrel{\text{def}}{=} \forall\, s, e', s', e'', s''.$$
$$(\langle e, s\rangle \longrightarrow \langle e', s'\rangle \wedge \langle e, s\rangle \longrightarrow \langle e'', s''\rangle)$$
$$\Rightarrow \langle e', s'\rangle = \langle e'', s''\rangle$$

and show $\forall\ e\ \in\ L_1.\Phi(e)$ by structural induction.

$$\Phi(e) \stackrel{\text{def}}{=} \forall\, s, e', s', e'', s''.\ (\langle e, s\rangle \longrightarrow \langle e', s'\rangle \wedge \langle e, s\rangle \longrightarrow \langle e'', s''\rangle) \Rightarrow \langle e', s'\rangle = \langle e'', s''\rangle$$

nullary: $\Phi(\textbf{skip})$

$\forall\, b \in \{\textbf{true}, \textbf{false}\}.\Phi(b)$

$\forall\, n \in \mathbb{Z}.\Phi(n)$

$\forall\, \ell \in \mathbb{L}.\Phi(!\ell)$

unary: $\forall\, \ell \in \mathbb{L}.\forall\, e.\Phi(e) \Rightarrow \Phi(\ell := e)$

binary: $\forall\ op\ .\forall\, e_1, e_2.(\Phi(e_1) \wedge \Phi(e_2)) \Rightarrow \Phi(e_1\ \ op\ \ e_2)$

$\forall\, e_1, e_2.(\Phi(e_1) \wedge \Phi(e_2)) \Rightarrow \Phi(e_1; e_2)$

$\forall\, e_1, e_2.(\Phi(e_1) \wedge \Phi(e_2)) \Rightarrow \Phi(\textbf{while}\ \ e_1\ \ \textbf{do}\ \ e_2)$

ternary: $\forall\, e_1, e_2, e_3.(\Phi(e_1) \wedge \Phi(e_2) \wedge \Phi(e_3)) \Rightarrow \Phi(\textbf{if}\ \ e_1\ \ \textbf{then}\ \ e_2\ \ \textbf{else}\ \ e_3)$

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

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

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

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

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

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

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

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

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

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

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

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

(while)

$\langle \textbf{while } e_1 \textbf{ do } e_2, s\rangle \longrightarrow \langle \textbf{if } e_1 \textbf{ the}$

$$\Phi(e) \stackrel{\text{def}}{=} \forall\, s, e', s', e'', s''.\ (\langle e, s\rangle \longrightarrow \langle e', s'\rangle \wedge \langle e, s\rangle \longrightarrow \langle e'', s''\rangle) \Rightarrow \langle e', s'\rangle = \langle e'', s''\rangle$$

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

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

Lemma: Values don't reduce

Lemma 9 *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 defn e is a value if it is of one of the forms n, b, **skip**. By examination of the rules on slides ..., there is no rule with conclusion of the form $\langle e, s\rangle \longrightarrow \langle e', s'\rangle$ for e one of n, b, **skip**. □

Inversion

In proofs involving multiple inductive definitions one often needs an *inversion property*, that, given a tuple in one inductively defined relation, gives you a case analysis of the possible "last rule" used.

Lemma 10 (Inversion for $\longrightarrow$) *If* $\langle e, s\rangle \longrightarrow \langle \hat{e}, \hat{s}\rangle$ *then either*

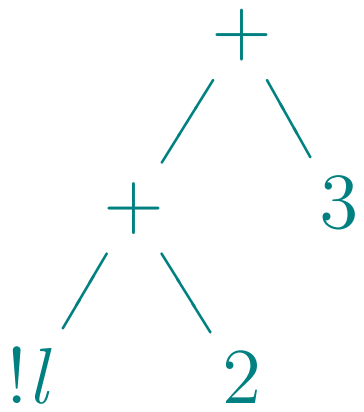
1. *(op* $+$*) there exists* n_1*,* n_2*, and* n *such that* $e = n_1 + n_2$*,* $\hat{e} = n$*,* $\hat{s} = s$*, and* $n = n_1 + n_2$ *(NB watch out for the two different* $+$*s), or*
2. *(op1) there exists* e_1*,* e_2*,* op *, and* e_1' *such that* $e = e_1 \;\; op \;\; e_2$*,* $\hat{e} = e_1' \;\; op \;\; e_2$*, and* $\langle e_1, s\rangle \longrightarrow \langle e_1', s'\rangle$*, or*
3. *...*

Lemma 11 (Inversion for $\vdash$) *If* $\Gamma \vdash e : T$ *then either*

1. *...*

All the determinacy proof details are in the notes.

Having proved those 9 things, consider an example $(!l + 2) + 3$. To see why $\Phi((!l + 2) + 3)$ holds:



Inductive Definitions and Rule Induction

How to prove facts about all elements of the L1 typing relation or the L1 reduction relation, e.g. Progress or Type Preservation?

Theorem 2 (Progress) *If* $\Gamma \vdash e : T$ *and* $dom(\Gamma) \subseteq dom(s)$ *then either* e *is a value or there exist* e', s' *such that* $\langle e, s \rangle \longrightarrow \langle e', s' \rangle$.

Theorem 3 (Type Preservation) *If* $\Gamma \vdash e : T$ *and* $dom(\Gamma) \subseteq dom(s)$ *and* $\langle e, s \rangle \longrightarrow \langle e', s' \rangle$ *then* $\Gamma \vdash e' : T$ *and* $dom(\Gamma) \subseteq dom(s')$.

What does $\langle e, s \rangle \longrightarrow \langle e', s' \rangle$ really mean?

Inductive Definitions

We defined the transition relation $\langle e, s\rangle \longrightarrow \langle e', s'\rangle$ and the typing relation $\Gamma \vdash e{:}T$ by giving some rules, eg

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

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

(op $+$) $\quad \dfrac{\Gamma \vdash e_1{:}\mathsf{int} \quad \Gamma \vdash e_2{:}\mathsf{int}}{\Gamma \vdash e_1 + e_2{:}\mathsf{int}}$

What did we actually mean?

These relations are just normal set-theoretic relations, written in infix notation.

For the transition relation:

- Start with $A = L_1 * \mathrm{store} * L_1 * \mathrm{store}$.
- Write $\longrightarrow \subseteq A$ infix, e.g. $\langle e, s\rangle \longrightarrow \langle e', s'\rangle$ instead of $(e, s, e', s') \in \longrightarrow$.

For the typing relation:

- Start with $A = \mathrm{TypeEnv} * L_1 * \mathrm{types}$.
- Write $\vdash \subseteq A$ mixfix, e.g. $\Gamma \vdash e{:}T$ instead of $(\Gamma, e, T) \in \vdash$.

For each rule we can construct the set of all concrete *rule instances*, taking all values of the metavariables that satisfy the side condition. For example, for (op $+$) and (op1) we take all values of n_1, n_2, s, n (satisfying $n = n_1 + n_2$) and of e_1, e_2, s, e_1', s'.

$$\text{(op}+\text{)}\ \frac{}{\langle 2+2, \{\}\rangle \longrightarrow \langle 4, \{\}\rangle}\ , \qquad \text{(op}+\text{)}\ \frac{}{\langle 2+3, \{\}\rangle \longrightarrow \langle 5, \{\}\rangle}\ , \ldots$$

$$\text{(op1)}\ \frac{\langle 2+2, \{\}\rangle \longrightarrow \langle 4, \{\}\rangle}{\langle (2+2)+3, \{\}\rangle \longrightarrow \langle 4+3, \{\}\rangle}\ , \qquad \text{(op1)}\ \frac{\langle 2+2, \{\}\rangle \longrightarrow \langle \textbf{false}, \{\}\rangle}{\langle (2+2)+3, \{\}\rangle \longrightarrow \langle \textbf{false}+3, \{\}\rangle}$$

Now a *derivation* of a transition $\langle e, s\rangle \longrightarrow \langle e', s'\rangle$ or typing judgment $\Gamma \vdash e{:}T$ is a finite tree such that each step *is* a concrete rule instance.

$$\dfrac{\dfrac{\dfrac{}{\langle 2+2, \{\}\rangle \longrightarrow \langle 4, \{\}\rangle}\,(\text{op}+)}{\langle (2+2)+3, \{\}\rangle \longrightarrow \langle 4+3, \{\}\rangle}\,(\text{op1})}{\langle (2+2)+3 \geq 5, \{\}\rangle \longrightarrow \langle 4+3 \geq 5, \{\}\rangle}\,(\text{op1})$$

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash !l{:}\mathsf{int}}\,(\text{deref}) \quad \dfrac{}{\Gamma \vdash 2{:}\mathsf{int}}\,(\text{int})}{\Gamma \vdash (!l+2){:}\mathsf{int}}\,(\text{op}+) \quad \dfrac{}{\Gamma \vdash 3{:}\mathsf{int}}\,(\text{int})}{\Gamma \vdash (!l+2)+3{:}\mathsf{int}}\,(\text{op}+)$$

and $\langle e, s\rangle \longrightarrow \langle e', s'\rangle$ is an element of the reduction relation (resp. $\Gamma \vdash e{:}T$ is an element of the transition relation) iff there is a derivation with that as the root node.

Principle of Rule Induction

For any property $\Phi(a)$ of elements a of A, and any set of rules which define a subset S_R of A, to prove

$$\forall\ a\ \in\ S_R.\Phi(a)$$

it's enough to prove that $\{a \mid \Phi(a)\}$ is closed under the rules, ie for each concrete rule instance

$$\frac{h_1 \quad .. \quad h_k}{c}$$

if $\Phi(h_1) \wedge \ldots \wedge \Phi(h_k)$ then $\Phi(c)$.

Principle of rule induction (a slight variant)

For any property $\Phi(a)$ of elements a of A, and any set of rules which inductively define the set S_R, to prove

$$\forall\ a\ \in\ S_R.\Phi(a)$$

it's enough to prove that

for each concrete rule instance

$$\frac{h_1 \quad .. \quad h_k}{c}$$

if $\Phi(h_1) \wedge ... \wedge \Phi(h_k) \wedge h_1\ \in\ S_R \wedge .. \wedge h_k\ \in\ S_R$ then $\Phi(c)$.

Proving Progress (Outline)

Theorem 2 (Progress) *If* $\Gamma \vdash e : T$ *and* $dom(\Gamma) \subseteq dom(s)$ *then either* e *is a value or there exist* e', s' *such that* $\langle e, s\rangle \longrightarrow \langle e', s'\rangle$.

Proof Take

$$\Phi(\Gamma, e, T) \stackrel{\text{def}}{=} \forall\, s.\ \mathsf{dom}(\Gamma) \subseteq \mathsf{dom}(s) \Rightarrow \mathsf{value}(e) \vee (\exists\, e', s'.\langle e, s\rangle \longrightarrow \langle e', s'\rangle)$$

We show that for all Γ, e, T, if $\Gamma \vdash e : T$ then $\Phi(\Gamma, e, T)$, by rule induction on the definition of $\vdash$.

Principle of Rule Induction (variant form): to prove $\Phi(a)$ for all a in the set S_R, it's enough to prove that for each concrete rule instance

$$\frac{h_1 \quad .. \quad h_k}{c}$$

if $\Phi(h_1) \wedge ... \wedge \Phi(h_k) \wedge h_1 \in S_R \wedge .. \wedge h_k \in S_R$ then $\Phi(c)$.

Instantiating to the L1 typing rules, have to show:

(int) $\forall\, \Gamma, n.\Phi(\Gamma, n, \mathsf{int})$

(deref) $\forall\, \Gamma, \ell.\Gamma(\ell) = \mathsf{intref} \Rightarrow \Phi(\Gamma, !\ell, \mathsf{int})$

(op $+$) $\forall\, \Gamma, e_1, e_2.(\Phi(\Gamma, e_1, \mathsf{int}) \wedge \Phi(\Gamma, e_2, \mathsf{int}) \wedge \Gamma \vdash e_1{:}\mathsf{int} \wedge \Gamma \vdash e_2{:}\mathsf{int})$

$\Rightarrow \Phi(\Gamma, e_1 + e_2, \mathsf{int})$

(seq) $\forall\, \Gamma, e_1, e_2, T.(\Phi(\Gamma, e_1, \mathsf{unit}) \wedge \Phi(\Gamma, e_2, T) \wedge \Gamma \vdash e_1{:}\mathsf{unit} \wedge \Gamma \vdash e_2{:}T)$

$\Rightarrow \Phi(\Gamma, e_1; e_2, T)$

etc.

Having proved those 10 things, consider an example $\Gamma \vdash (!l + 2) + 3$:int. To see why $\Phi(\Gamma,\ (!l + 2) + 3,\ \text{int})$ holds:

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash !l\text{:int}}\ \text{(deref)} \quad \dfrac{}{\Gamma \vdash 2\text{:int}}\ \text{(int)}}{\Gamma \vdash (!l + 2)\text{:int}}\ \text{(op } +\text{)} \quad \dfrac{}{\Gamma \vdash 3\text{:int}}\ \text{(int)}}{\Gamma \vdash (!l + 2) + 3\text{:int}}\ \text{(op } +\text{)}$$

Which Induction Principle to Use?

Which of these induction principles to use is a matter of convenience – you want to use an induction principle that matches the definitions you're working with.

Example Proofs

In the notes there are detailed example proofs for Determinacy (structural induction), Progress (rule induction on type derivations), and Type Preservation (rule induction on reduction derivations).

You should read them off-line, and do the exercises.

When is a proof a proof?

What's a proof?

Formal: a derivation in formal logic (e.g. a big natural deduction proof tree). Often far too verbose to deal with by hand (but can *machine-check* such things).

Informal but rigorous: an argument to persuade the reader that, if pushed, you could write a fully formal proof (the usual mathematical notion, e.g. those we just did). Have to learn by practice to see when they are rigorous.

Bogus: neither of the above.

Rant

clear structure matters!

Sometimes it seems hard or pointless to prove things because they seem ‘too obvious’....

1. proof lets you see (and explain) *why* they are obvious
2. sometimes the obvious facts are false...
3. sometimes the obvious facts are not obvious at all
4. sometimes a proof contains or suggests an algorithm that you need – eg, proofs that type inference is decidable (for fancier type systems)

Lemma: Values of integer type

Lemma 12 *for all* Γ, e, T*, if* $\Gamma \vdash e{:}T$*,* e *is a value and* $T = \mathsf{int}$ *then for some* $n \in \mathbb{Z}$ *we have* $e = n$*.*

Proving Progress

Theorem 2 (Progress) *If* $\Gamma \vdash e{:}T$ *and* $\mathit{dom}(\Gamma) \subseteq \mathit{dom}(s)$ *then either* e *is a value or there exist* e', s' *such that* $\langle e, s\rangle \longrightarrow \langle e', s'\rangle$.

Proof Take

$$\Phi(\Gamma, e, T) \stackrel{\text{def}}{=} \forall\, s.\ \mathsf{dom}(\Gamma) \subseteq \mathsf{dom}(s) \Rightarrow \mathsf{value}(e) \vee (\exists\, e', s'.\langle e, s\rangle \longrightarrow \langle e', s'\rangle)$$

We show that for all Γ, e, T, if $\Gamma \vdash e{:}T$ then $\Phi(\Gamma, e, T)$, by rule induction on the definition of $\vdash$.

Principle of Rule Induction (variant form): to prove $\Phi(a)$ for all a in the set S_R defined by the rules, it's enough to prove that for each rule instance

$$\frac{h_1 \quad .. \quad h_k}{c}$$

if $\Phi(h_1) \wedge ... \wedge \Phi(h_k) \wedge h_1 \in S_R \wedge .. \wedge h_k \in S_R$ then $\Phi(c)$.

Instantiating to the L1 typing rules, have to show:

(int) $\forall\, \Gamma, n.\Phi(\Gamma, n, \mathsf{int})$

(deref) $\forall\, \Gamma, \ell.\Gamma(\ell) = \mathsf{intref} \Rightarrow \Phi(\Gamma, !\ell, \mathsf{int})$

(op $+$) $\forall\, \Gamma, e_1, e_2.(\Phi(\Gamma, e_1, \mathsf{int}) \wedge \Phi(\Gamma, e_2, \mathsf{int}) \wedge \Gamma \vdash e_1 : \mathsf{int} \wedge \Gamma \vdash e_2 : \mathsf{int})$

$\Rightarrow \Phi(\Gamma, e_1 + e_2, \mathsf{int})$

(seq) $\forall\, \Gamma, e_1, e_2, T.(\Phi(\Gamma, e_1, \mathsf{unit}) \wedge \Phi(\Gamma, e_2, T) \wedge \Gamma \vdash e_1 : \mathsf{unit} \wedge \Gamma \vdash e_2 : T)$

$\Rightarrow \Phi(\Gamma, e_1; e_2, T)$

etc.

$$\Phi(\Gamma, e, T) \stackrel{\text{def}}{=} \forall\, s.\ \mathsf{dom}(\Gamma) \subseteq \mathsf{dom}(s) \Rightarrow \mathsf{value}(e) \vee (\exists\, e', s'.\langle e, s\rangle \longrightarrow \langle e', s'\rangle)$$

Case (op$+$). Recall the rule

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

Suppose $\Phi(\Gamma, e_1, \mathsf{int})$, $\Phi(\Gamma, e_2, \mathsf{int})$, $\Gamma \vdash e_1 : \mathsf{int}$, and $\Gamma \vdash e_2 : \mathsf{int}$.

We have to show $\Phi(\Gamma, e_1 + e_2, \mathsf{int})$.

Consider an arbitrary s. Assume $\mathsf{dom}(\Gamma) \subseteq \mathsf{dom}(s)$.

Now $e_1 + e_2$ is not a value, so we have to show

$\exists\, \langle e', s'\rangle.\ \langle e_1 + e_2, s\rangle \longrightarrow \langle e', s'\rangle$

Using $\Phi(\Gamma, e_1, \mathsf{int})$ and $\Phi(\Gamma, e_2, \mathsf{int})$ we have:

case e_1 reduces. Then $e_1 + e_2$ does, using (op1).

case e_1 is a value but e_2 reduces. Then $e_1 + e_2$ does, using (op2).

case Both e_1 and e_2 are values. Want to use:

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

Lemma 13 *for all* Γ, e, T*, if* $\Gamma \vdash e{:}T$*,* e *is a value and* $T = \mathsf{int}$ *then for some* $n \in \mathbb{Z}$ *we have* $e = n$*.*

We assumed (the variant rule induction principle) that $\Gamma \vdash e_1{:}\mathsf{int}$ and $\Gamma \vdash e_2{:}\mathsf{int}$, so using this Lemma have $e_1 = n_1$ and $e_2 = n_2$.

Then $e_1 + e_2$ reduces, using rule (op$+$).

All the other cases are in the notes.

Summarising Proof Techniques

Determinacy	structural induction for e
Progress	rule induction for $\Gamma \vdash e : T$
Type Preservation	rule induction for $\langle e, s\rangle \longrightarrow \langle e', s'\rangle$
Safety	mathematical induction on $\longrightarrow^k$
Uniqueness of typing	...
Decidability of typability	exhibiting an algorithm
Decidability of checking	corollary of other results

Functions – L2

Functions, Methods, Procedures...

```
fun addone x = x+1
```

```
public int addone(int x) {
  x+1
  }
```

```
<script type="text/vbscript">
function addone(x)
 addone = x+1
end function
</script>
```

C#

```
delegate int IntThunk();

class M {
  public static void Main() {
    IntThunk[] funcs = new IntThunk[11];
    for (int i = 0; i <= 10; i++)
    {
      funcs[i] = delegate() { return i; };
    }
    foreach (IntThunk f in funcs)
    {
      System.Console.WriteLine(f());
    }
  }
}
```

Functions – Examples

We will add expressions like these to L1.

$$(\textbf{fn}\ \ \mathrm{x{:}int} \Rightarrow \mathrm{x} + 1)$$

$$(\textbf{fn}\ \ \mathrm{x{:}int} \Rightarrow \mathrm{x} + 1)\ 7$$

$$(\textbf{fn}\ \ \mathrm{y{:}int} \Rightarrow (\textbf{fn}\ \ \mathrm{x{:}int} \Rightarrow \mathrm{x} + \mathrm{y}))$$

$$(\textbf{fn}\ \ \mathrm{y{:}int} \Rightarrow (\textbf{fn}\ \ \mathrm{x{:}int} \Rightarrow \mathrm{x} + \mathrm{y}))\ 1$$

$$(\textbf{fn}\ \ \mathrm{x{:}int} \to \mathrm{int} \Rightarrow (\textbf{fn}\ \ \mathrm{y{:}int} \Rightarrow \mathrm{x}\ (\mathrm{x}\ \mathrm{y})))$$

$$(\textbf{fn}\ \ \mathrm{x{:}int} \to \mathrm{int} \Rightarrow (\textbf{fn}\ \ \mathrm{y{:}int} \Rightarrow \mathrm{x}\ (\mathrm{x}\ \mathrm{y})))\ (\textbf{fn}\ \ \mathrm{x{:}int} \Rightarrow \mathrm{x} + 1)$$

$$\big((\textbf{fn}\ \ \mathrm{x{:}int} \to \mathrm{int} \Rightarrow (\textbf{fn}\ \ \mathrm{y{:}int} \Rightarrow \mathrm{x}\ (\mathrm{x}\ \mathrm{y})))\ (\textbf{fn}\ \ \mathrm{x{:}int} \Rightarrow \mathrm{x} + 1)\ \big)\ 7$$

Functions – Syntax

First, extend the L1 syntax:

Variables $x \in \mathbb{X}$ for a set $\mathbb{X} = \{\mathrm{x}, \mathrm{y}, \mathrm{z}, ...\}$

Expressions

$$e \quad ::= \quad ... \mid \textbf{fn}\ x{:}T \Rightarrow e \mid e_1\ e_2 \mid x$$

Types

$$\begin{aligned} T \quad &::= \quad \mathsf{int} \mid \mathsf{bool} \mid \mathsf{unit} \mid T_1 \to T_2 \\ T_{loc} \quad &::= \quad \mathsf{intref} \end{aligned}$$

Variable shadowing

$(\textbf{fn}\ \text{x:int} \Rightarrow (\textbf{fn}\ \text{x:int} \Rightarrow \text{x} + 1))$

```
class F {
  void m() {
    int y;
    {int y; ...  } // Static error
    ...
    {int y; ...  }
    ...
    }
}
```

Alpha conversion

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

- inside e, any x's (that aren't themselves binders and are not inside another **fn** $x{:}T' \Rightarrow \ldots$) mean the same thing – the formal parameter of this function.
- outside this **fn** $x{:}T \Rightarrow e$, it doesn't matter which variable we used for the formal parameter – in fact, we shouldn't be able to tell. For example, **fn** $\mathrm{x{:}int} \Rightarrow \mathrm{x} + 2$ should be the same as **fn** $\mathrm{y{:}int} \Rightarrow \mathrm{y} + 2$.

  cf $\int_0^1 x + x^2 dx = \int_0^1 y + y^2 dy$

Alpha conversion – free and bound occurrences

In a bit more detail (but still informally):

Say an occurrence of x in an expression e is *free* if it is not inside any $(\textbf{fn}\ \ x{:}T \Rightarrow \ldots)$. For example:

$$17$$

$$\mathrm{x} + \mathrm{y}$$

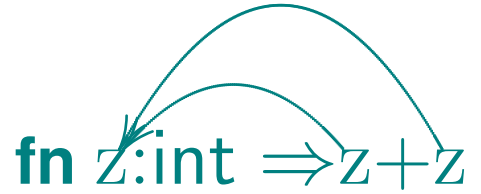
$$\textbf{fn}\ \ \mathrm{x{:}int} \Rightarrow \mathrm{x} + 2$$

$$\textbf{fn}\ \ \mathrm{x{:}int} \Rightarrow \mathrm{x} + \mathrm{z}$$

$$\textbf{if}\ \mathrm{y}\ \textbf{then}\ \ 2 + \mathrm{x}\ \textbf{else}\ ((\textbf{fn}\ \ \mathrm{x{:}int} \Rightarrow \mathrm{x} + 2)\mathrm{z})$$

All the other occurrences of x are *bound* by the closest enclosing $\textbf{fn}\ \ x{:}T \Rightarrow \ldots$.

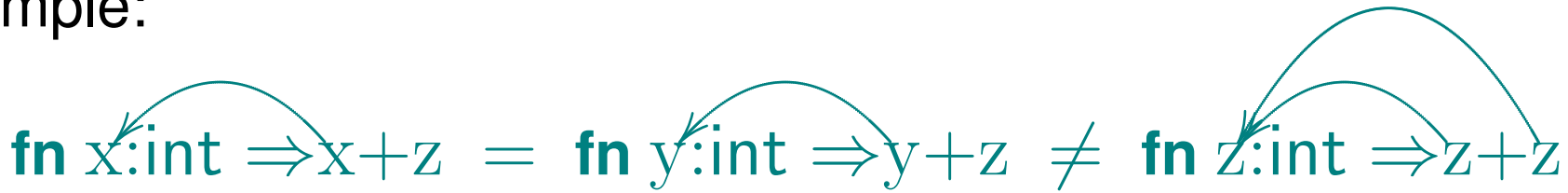
Alpha conversion – Binding examples



Alpha Conversion – The Convention

Convention: we will allow ourselves to *any time at all, in any expression* $\ldots(\textbf{fn}\ \ x{:}T \Rightarrow e)\ldots$, replace the binding x and all occurrences of x that are bound by that binder, by any other variable – so long as that doesn't change the binding graph.

For example:

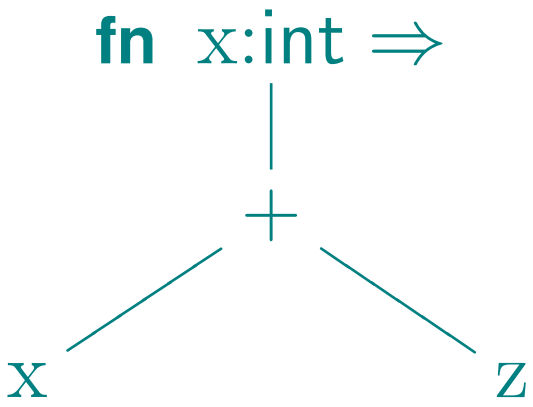


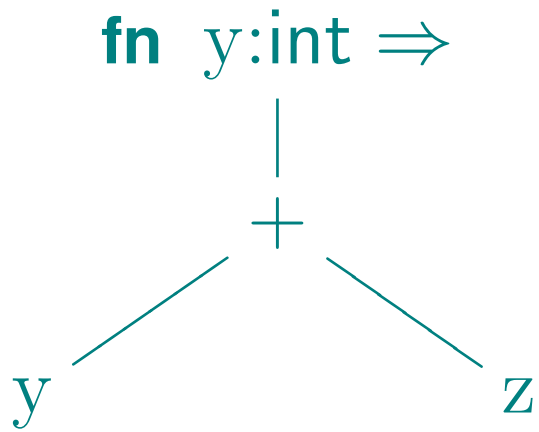
This is called 'working up to alpha conversion'. It amounts to regarding the syntax not as abstract syntax trees, but as abstract syntax trees with pointers...

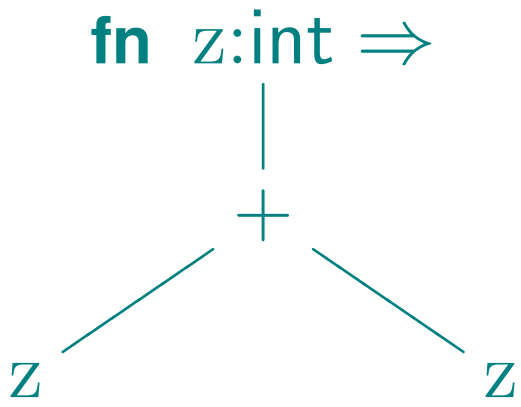
Abstract Syntax *up to Alpha Conversion*

fn $\mathrm{x}{:}\mathsf{int} \Rightarrow \mathrm{x} + \mathrm{z}$ $=$ **fn** $\mathrm{y}{:}\mathsf{int} \Rightarrow \mathrm{y} + \mathrm{z}$ $\neq$ **fn** $\mathrm{z}{:}\mathsf{int} \Rightarrow \mathrm{z} + \mathrm{z}$

Start with naive abstract syntax trees:

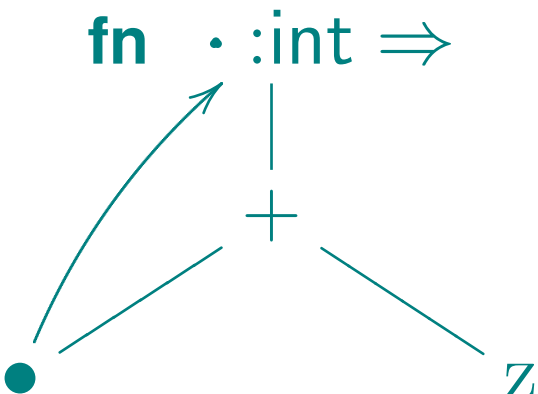


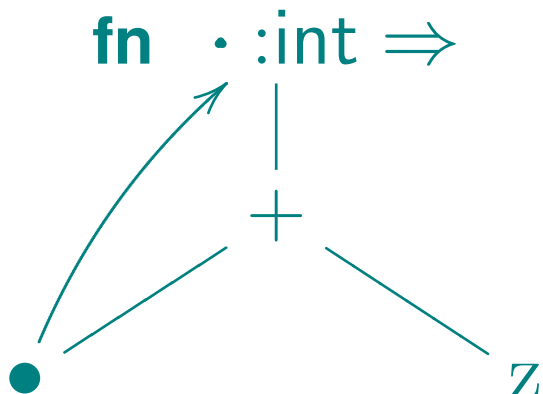


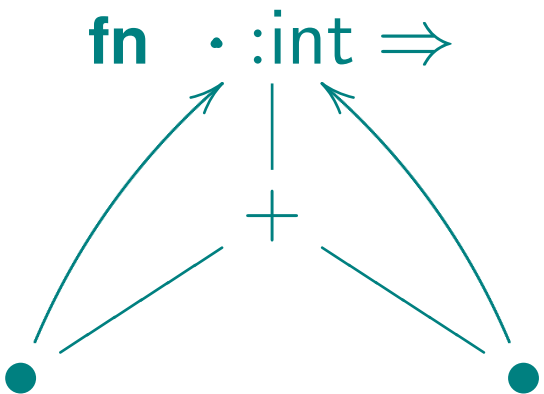


add pointers (from each x node to the closest enclosing **fn** $x{:}T \Rightarrow$ node);

remove names of binders and the occurrences they bind

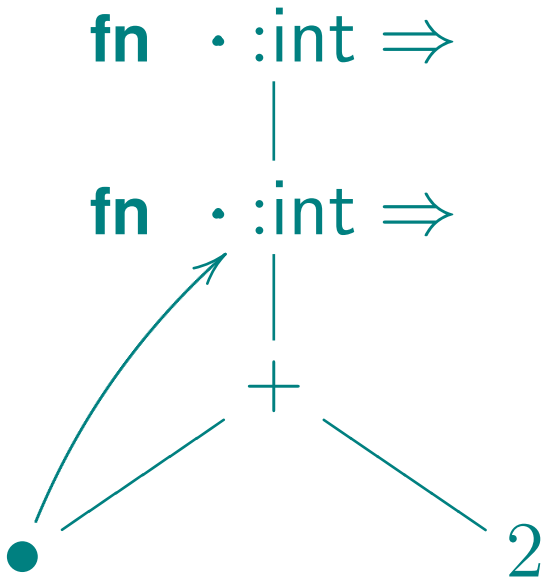


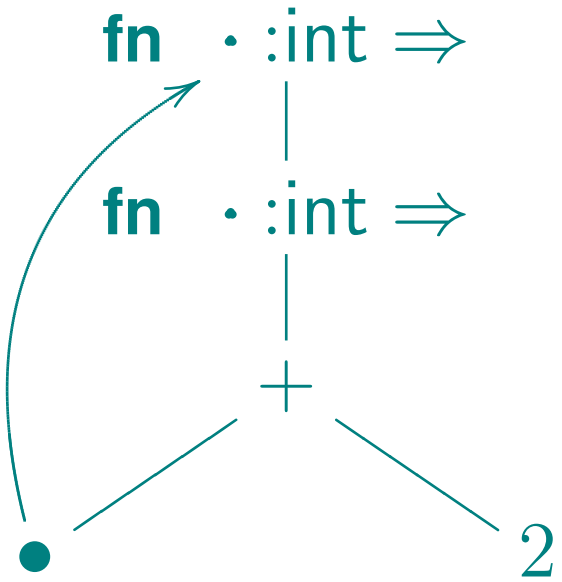




$$\textsf{fn}\ \ \mathrm{x{:}int} \Rightarrow (\textsf{fn}\ \ \mathrm{x{:}int} \Rightarrow \mathrm{x} + 2)$$

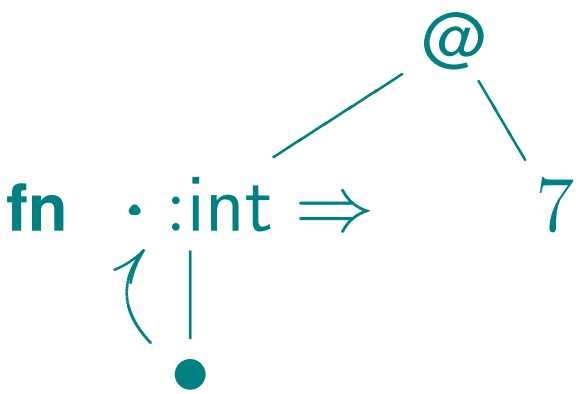
$$= \textsf{fn}\ \ \mathrm{y{:}int} \Rightarrow (\textsf{fn}\ \ \mathrm{z{:}int} \Rightarrow \mathrm{z} + 2) \quad \neq \quad \textsf{fn}\ \ \mathrm{z{:}int} \Rightarrow (\textsf{fn}\ \ \mathrm{y{:}int} \Rightarrow \mathrm{z} + 2)$$

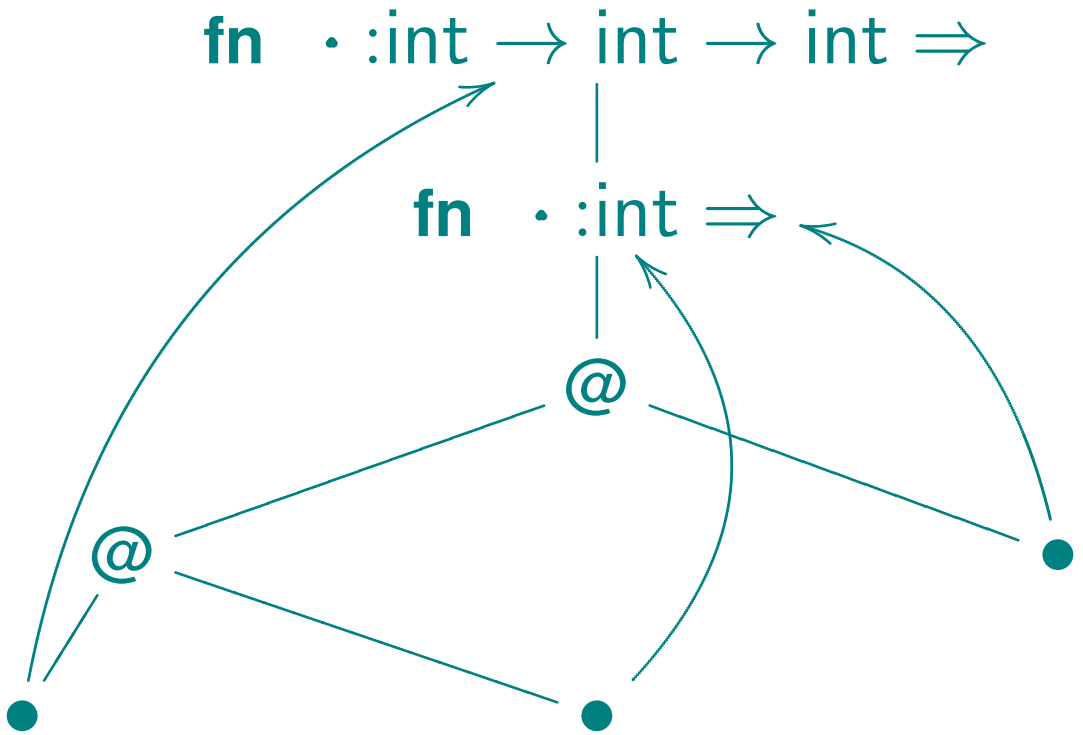




$(\textbf{fn}\ \ \mathrm{x{:}int} \Rightarrow \mathrm{x})\ 7$

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



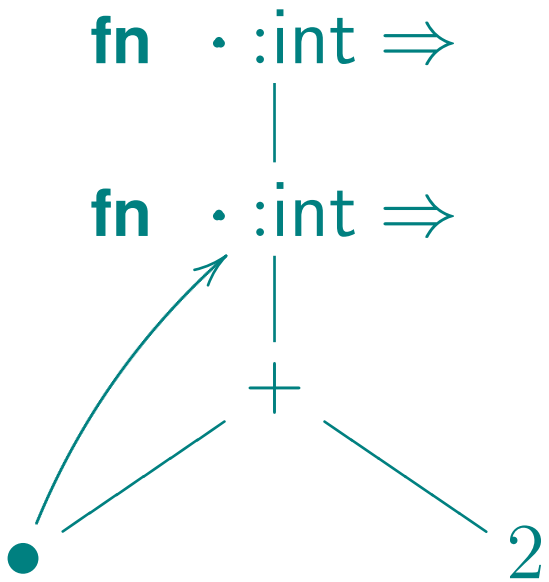


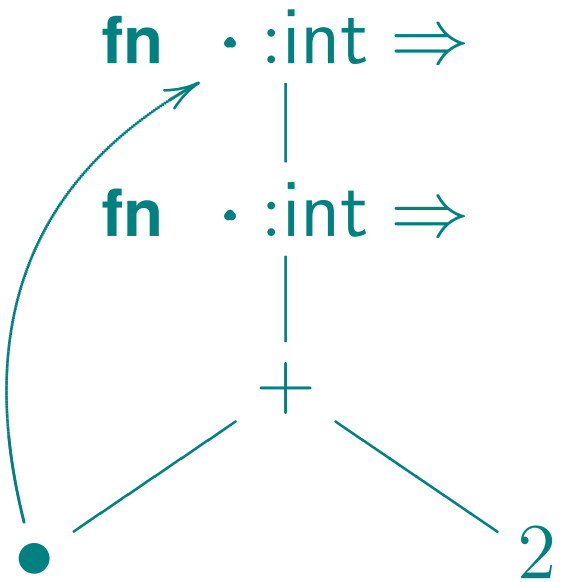
De Bruijn indices

Our implementation will use those pointers – known as *De Bruijn indices*.

Each occurrence of a bound variable is represented by the number of **fn** $\cdot : T \Rightarrow$ nodes you have to count out to to get to its binder.

$$\textbf{fn}\ \cdot : \text{int} \Rightarrow (\textbf{fn}\ \cdot : \text{int} \Rightarrow \text{v}_0 + 2) \quad \neq \quad \textbf{fn}\ \cdot : \text{int} \Rightarrow (\textbf{fn}\ \cdot : \text{int} \Rightarrow \text{v}_1 +$$





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.

$$\begin{aligned} \mathsf{fv}(x) &= \{x\} \\ \mathsf{fv}(e_1 \;\; op \;\; e_2) &= \mathsf{fv}(e_1) \cup \mathsf{fv}(e_2) \\ \mathsf{fv}(\textbf{fn} \;\; x{:}T \Rightarrow e) &= \mathsf{fv}(e) - \{x\} \end{aligned}$$

Say e is *closed* if $\mathsf{fv}(e) = \{\}$.

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

(note this definition is alpha-invariant - all our definitions should be)

Substitution – Examples

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

Write $\{e/x\}e'$ for the result of substituting e for all *free* occurrences of x in e'. For example

$$
\begin{array}{lcl}
\{3/\mathrm{x}\}(\mathrm{x} \geq \mathrm{x}) & = & (3 \geq 3) \\
\{3/\mathrm{x}\}((\textbf{fn}\ \ \mathrm{x{:}int} \Rightarrow \mathrm{x} + \mathrm{y})\mathrm{x}) & = & (\textbf{fn}\ \ \mathrm{x{:}int} \Rightarrow \mathrm{x} + \mathrm{y})3 \\
\{\mathrm{y} + 2/\mathrm{x}\}(\textbf{fn}\ \ \mathrm{y{:}int} \Rightarrow \mathrm{x} + \mathrm{y}) & = & \textbf{fn}\ \ \mathrm{z{:}int} \Rightarrow (\mathrm{y} + 2) + \mathrm{z}
\end{array}
$$

Substitution – Definition

Defining that:

$$\{e/z\}x = e \quad \text{if } x = z$$

$$= x \quad \text{otherwise}$$

$$\{e/z\}(\textbf{fn}\ x{:}T \Rightarrow e_1) = \textbf{fn}\ x{:}T \Rightarrow (\{e/z\}e_1) \quad \text{if } x \neq z \text{ (*) and } x \notin \mathsf{fv}(e) \text{ (*}$$

$$\{e/z\}(e_1\ e_2) = (\{e/z\}e_1)(\{e/z\}e_2)$$

$\ldots$

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

Substitution – Example Again

$$
\begin{aligned}
& \{y + 2/x\}(\textbf{fn}\ \ y{:}\mathsf{int} \Rightarrow x + y) \\
= \quad & \{y + 2/x\}(\textbf{fn}\ \ y'{:}\mathsf{int} \Rightarrow x + y') \text{ renaming} \\
= \quad & \textbf{fn}\ \ y'{:}\mathsf{int} \Rightarrow \{y + 2/x\}(x + y') \text{ as } y' \neq x \text{ and } y' \notin \mathsf{fv}(y + 2) \\
= \quad & \textbf{fn}\ \ y'{:}\mathsf{int} \Rightarrow \{y + 2/x\}x + \{y + 2/x\}y' \\
= \quad & \textbf{fn}\ \ y'{:}\mathsf{int} \Rightarrow (y + 2) + y'
\end{aligned}
$$

(could have chosen any other z instead of y', except y or x)

Simultaneous substitution

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

Notation: write a σ as $\{e_1/x_1, .., e_k/x_k\}$ instead of $\{x_1 \mapsto e_1, \ldots, x_k \mapsto e_k\}$ (for the function mapping x_1 to e_1 etc.)

A definition of $\sigma\ e$ is given in the notes.

Function Behaviour

Consider the expression

$$e = (\textbf{fn}\ \ \mathrm{x{:}unit} \Rightarrow (l := 1); \mathrm{x})\ (l := 2)$$

then

$$\langle e, \{l \mapsto 0\}\rangle \longrightarrow^* \langle \textbf{skip}, \{l \mapsto ???\}\rangle$$

Function Behaviour. Choice 1: Call-by-value

Informally: reduce left-hand-side of application to a **fn**-term; reduce argument to a value; then replace all occurrences of the formal parameter in the **fn**-term by that value.

$$e = (\textbf{fn}\ \ \mathrm{x{:}unit} \Rightarrow (l := 1); \mathrm{x})(l := 2)$$

$$\begin{aligned}
\langle e, \{l = 0\}\rangle &\longrightarrow \langle (\textbf{fn}\ \ \mathrm{x{:}unit} \Rightarrow (l := 1); \mathrm{x})\textbf{skip}, \{l = 2\}\rangle \\
&\longrightarrow \langle (l := 1); \textbf{skip}, \{l = 2\}\rangle \\
&\longrightarrow \langle \textbf{skip}; \textbf{skip}, \{l = 1\}\rangle \\
&\longrightarrow \langle \textbf{skip}, \{l = 1\}\rangle
\end{aligned}$$

L2 Call-by-value

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

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

L2 Call-by-value – reduction examples

$$
\begin{array}{ll}
 & \langle (\textbf{fn}\ \mathrm{x{:}int} \Rightarrow \textbf{fn}\ \mathrm{y{:}int} \Rightarrow \mathrm{x} + \mathrm{y})\ (3+4)\ 5\ ,\ s\rangle \\
= & \langle \big((\textbf{fn}\ \mathrm{x{:}int} \Rightarrow \textbf{fn}\ \mathrm{y{:}int} \Rightarrow \mathrm{x} + \mathrm{y})\ (3+4)\big)\ 5\ ,\ s\rangle \\
\longrightarrow & \langle \big((\textbf{fn}\ \mathrm{x{:}int} \Rightarrow \textbf{fn}\ \mathrm{y{:}int} \Rightarrow \mathrm{x} + \mathrm{y})\ 7\ \big)\ 5\ ,\ s\rangle \\
\longrightarrow & \langle \big(\{7/\mathrm{x}\}(\textbf{fn}\ \mathrm{y{:}int} \Rightarrow \mathrm{x} + \mathrm{y})\big)\ 5\ ,\ s\rangle \\
= & \langle \big((\textbf{fn}\ \mathrm{y{:}int} \Rightarrow 7 + \mathrm{y})\big)\ 5\ ,\ s\rangle \\
\longrightarrow & \langle 7+5\ ,\ s\rangle \\
\longrightarrow & \langle 12\ ,\ s\rangle
\end{array}
$$

$$(\textbf{fn}\ \mathrm{f{:}int} \to \mathrm{int} \Rightarrow \mathrm{f}\ 3)\ (\textbf{fn}\ \mathrm{x{:}int} \Rightarrow (1+2) + \mathrm{x})$$

Function Behaviour. Choice 2: Call-by-name

Informally: reduce left-hand-side of application to a **fn**-term; then replace all occurrences of the formal parameter in the **fn**-term by the argument.

$$e = (\textbf{fn}\ \ \mathrm{x{:}unit} \Rightarrow (l := 1); \mathrm{x})\ (l := 2)$$

$$\begin{aligned}
\langle e, \{l \mapsto 0\}\rangle &\longrightarrow \langle (l := 1); l := 2, \{l \mapsto 0\}\rangle \\
&\longrightarrow \langle \textbf{skip}\ \ ; l := 2, \{l \mapsto 1\}\rangle \\
&\longrightarrow \langle l := 2\ \ , \{l \mapsto 1\}\rangle \\
&\longrightarrow \langle \textbf{skip}\ \ , \{l \mapsto 2\}\rangle
\end{aligned}$$

L2 Call-by-name

(same typing rules as before)

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

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

Here, don't evaluate the argument at all if it isn't used

$$\begin{array}{lll} & \langle (\textbf{fn}\ \ \mathrm{x{:}unit} \Rightarrow \textbf{skip})(l := 2), \{l \mapsto 0\}\rangle \\ \longrightarrow & \langle \{l := 2/\mathrm{x}\}\textbf{skip} & , \{l \mapsto 0\}\rangle \\ = & \langle \textbf{skip} & , \{l \mapsto 0\}\rangle \end{array}$$

but if it is, end up evaluating it repeatedly.

Call-By-Need Example (Haskell)

```
let notdivby x y = y `mod` x /= 0
    enumFrom n = n :  (enumFrom (n+1))
    sieve (x:xs) =
      x :  sieve (filter (notdivby x) xs)
in
sieve (enumFrom 2)
==>
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,
 59,61,67,71,73,79,83,89,97,101,103,107,109,
 113,127,131,137,139,149,151,157,163,167,173,
 179,181,191,193,197,199,211,223,227,229,233,
 ^^Interrupted!
```

Purity

Function Behaviour. Choice 3: Full beta

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

$$(\textbf{fn}\ \ \mathrm{x{:}int} \Rightarrow 2+2) \longrightarrow (\textbf{fn}\ \ \mathrm{x{:}int} \Rightarrow 4)$$

L2 Beta

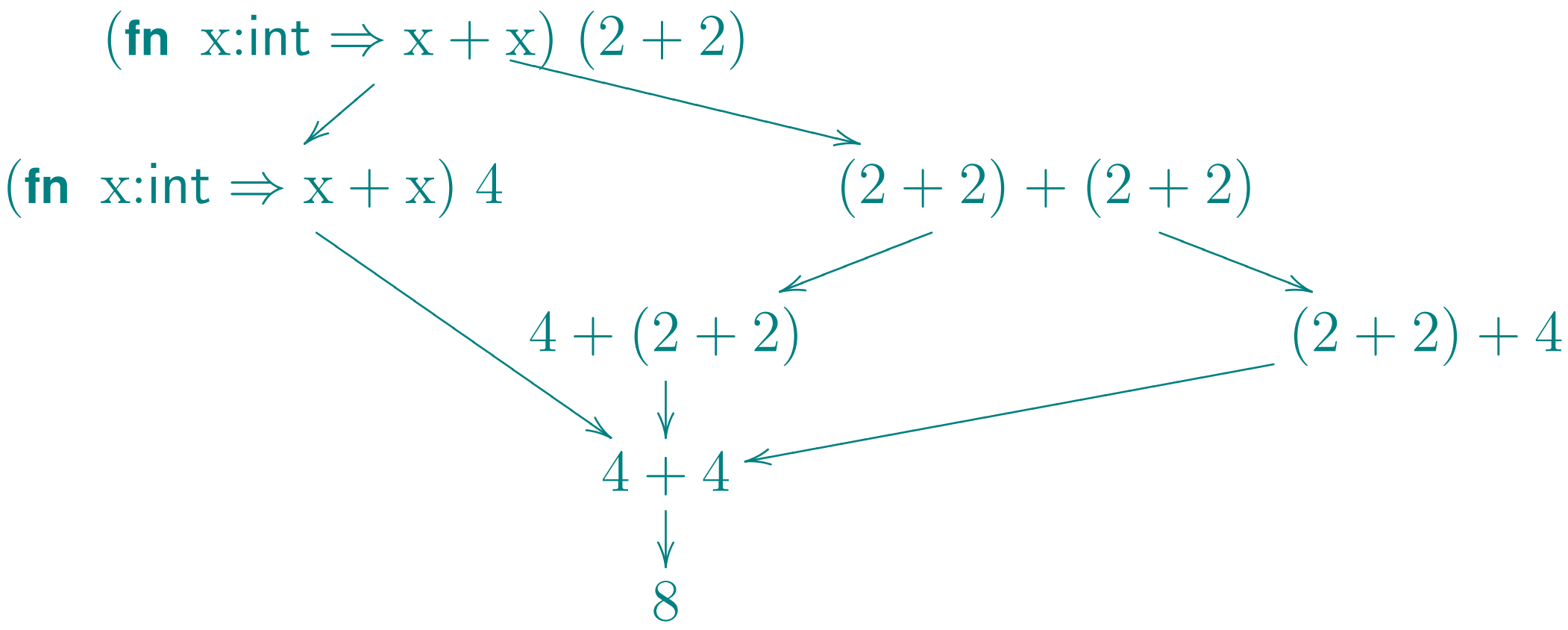
(beta-app1) $$\frac{\langle e_1, s\rangle \longrightarrow \langle e_1', s'\rangle}{\langle e_1\ e_2, s\rangle \longrightarrow \langle e_1'\ e_2, s'\rangle}$$

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

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

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

L2 Beta: Example



Function Behaviour. Choice 4: Normal-order reduction

Leftmost, outermost variant of full beta.

Back to CBV (from now on).

Typing functions (1)

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

Now, it must also give assumptions on the types of variables:

Type environments Γ are now pairs of a Γ_{loc} (a partial function from $\mathbb{L}$ to $\mathrm{T}_{\mathrm{loc}}$ as before) and a Γ_{var}, a partial function from $\mathbb{X}$ to T.

For example, we might have $\Gamma_{\mathsf{loc}} = l_1{:}\mathsf{intref}$ and $\Gamma_{\mathsf{var}} = \mathrm{x}{:}\mathsf{int},\ \mathrm{y}{:}\mathsf{bool} \to \mathsf{int}$.

Notation: we write $\mathsf{dom}(\Gamma)$ for the union of $\mathsf{dom}(\Gamma_{\mathsf{loc}})$ and $\mathsf{dom}(\Gamma_{\mathsf{var}})$. If $x \notin \mathsf{dom}(\Gamma_{\mathsf{var}})$, we 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}.

Typing functions (2)

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

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

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

Typing functions – Example

$$\dfrac{\dfrac{\dfrac{}{\mathrm{x{:}int} \vdash \mathrm{x{:}int}}\,(\mathrm{var}) \quad \dfrac{}{\mathrm{x{:}int} \vdash 2{:}\mathrm{int}}\,(\mathrm{int})}{\mathrm{x{:}int} \vdash \mathrm{x} + 2{:}\mathrm{int}}\,(\mathrm{op}+)}{\{\} \vdash (\mathbf{fn}\ \mathrm{x{:}int} \Rightarrow \mathrm{x} + 2){:}\mathrm{int} \to \mathrm{int}}\,(\mathrm{fn}) \quad \dfrac{}{\{\} \vdash 2{:}\mathrm{int}}\,(\mathrm{int})}{\{\} \vdash (\mathbf{fn}\ \mathrm{x{:}int} \Rightarrow \mathrm{x} + 2)\ 2{:}\mathrm{int}}\,(\mathrm{app})$$

Typing functions – Example

$$(\textbf{fn}\ \mathrm{x}{:}\mathsf{int} \to \mathsf{int} \Rightarrow \mathrm{x}((\textbf{fn}\ \mathrm{x}{:}\mathsf{int} \Rightarrow \mathrm{x})3))$$

Properties of Typing

We only consider executions of *closed* programs, with no free variables.

Theorem 14 (Progress) *If* e *closed and* $\Gamma \vdash e : T$ *and* $dom(\Gamma) \subseteq dom(s)$ *then either* e *is a value or there exist* e', s' *such that* $\langle e, s \rangle \longrightarrow \langle e', s' \rangle$.

Note there are now more stuck configurations, e.g. $((3)\ (4))$

Theorem 15 (Type Preservation) *If* e *closed and* $\Gamma \vdash e : T$ *and* $dom(\Gamma) \subseteq dom(s)$ *and* $\langle e, s \rangle \longrightarrow \langle e', s' \rangle$ *then* $\Gamma \vdash e' : T$ *and* e' *closed and* $dom(\Gamma) \subseteq dom(s')$.

Proving Type Preservation

Theorem 15 (Type Preservation) *If* e *closed and* $\Gamma \vdash e : T$ *and* $dom(\Gamma) \subseteq dom(s)$ *and* $\langle e, s\rangle \longrightarrow \langle e', s'\rangle$ *then* $\Gamma \vdash e' : T$ *and* e' *closed and* $dom(\Gamma) \subseteq dom(s')$.

Taking

$$\Phi(e, s, e', s') =$$
$$\forall\, \Gamma, T.$$
$$\Gamma \vdash e : T \wedge \ \mathsf{closed}(e) \wedge \mathsf{dom}(\Gamma) \subseteq \mathsf{dom}(s)$$
$$\Rightarrow$$
$$\Gamma \vdash e' : T \wedge \ \mathsf{closed}(e') \wedge \mathsf{dom}(\Gamma) \subseteq \mathsf{dom}(s')$$

we show $\forall\ e, s, e', s'.\langle e, s\rangle \longrightarrow \langle e', s'\rangle \Rightarrow \Phi(e, s, e', s')$ by rule induction.

To prove this one uses:

Lemma 16 (Substitution) *If* $\Gamma \vdash e : T$ *and* $\Gamma, x : T \vdash e' : T'$ *with* $x \notin \mathit{dom}(\Gamma)$ *then* $\Gamma \vdash \{e/x\}e' : T'$.

Normalization

Theorem 17 (Normalization) *In the sublanguage without while loops or store operations, if* $\Gamma \vdash e{:}T$ *and* e *closed then there does not exist an infinite reduction sequence* $\langle e, \{\}\rangle \longrightarrow \langle e_1, \{\}\rangle \longrightarrow \langle e_2, \{\}\rangle \longrightarrow \ldots$

Proof ? can't do a simple induction, as reduction can make terms grow.

See Pierce Ch.12 (the details are not in the scope of this course). □

Local definitions

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

$$e \quad ::= \quad \ldots \mid \textbf{let} \;\; \textbf{val} \;\; x{:}T = e_1 \;\; \textbf{in} \;\; e_2 \;\; \textbf{end}$$

this x is a binder, binding any free occurrences of x in e_2.

Can regard just as *syntactic sugar*:

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

Local definitions – derived typing and reduction rules (CBV)

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

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

(let1)

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

(let2)

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

Recursive definitions – first attempt

How about

$$\mathrm{x} = (\textbf{fn}\ \ \mathrm{y{:}int} \Rightarrow \textbf{if}\ \ \mathrm{y} \geq 1\ \ \textbf{then}\ \ \mathrm{y} + (\mathrm{x}\ (\mathrm{y} + -1))\ \textbf{else}\ \ 0)$$

where we use x within the definition of x? Think about evaluating $\mathrm{x}\ 3$.

Could add something like this:

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

(here the x binds in both e and e') then say

$$\begin{array}{l}\textbf{let}\ \ \textbf{val}\ \ \textbf{rec}\ \ \mathrm{x{:}int} \to \mathrm{int} = \\ \quad (\textbf{fn}\ \ \mathrm{y{:}int} \Rightarrow \textbf{if}\ \ \mathrm{y} \geq 1\ \ \textbf{then}\ \ \mathrm{y} + (\mathrm{x}(\mathrm{y} + -1))\ \textbf{else}\ \ 0) \\ \textbf{in}\ \ \mathrm{x}\ 3\ \ \textbf{end}\end{array}$$

But...

What about

let val rec $\mathrm{x} = (\mathrm{x}, \mathrm{x})$ **in** x **end** ?

Have some rather weird things, eg

let val rec $\mathrm{x{:}int\ list} = 3 :: \mathrm{x}$ **in** x **end**

does that terminate? if so, is it equal to

let val rec $\mathrm{x{:}int\ list} = 3 :: 3 :: \mathrm{x}$ **in** x **end** ? does

let val rec $\mathrm{x{:}int\ list} = 3 :: (\mathrm{x} + 1)$ **in** x **end** terminate?

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...

Recursive Functions

So, specialize the previous **let val rec** construct to

$$T \;=\; T_1 \to T_2 \qquad \text{recursion only at function types}$$

$$e \;=\; \textbf{fn}\ \ y{:}T_1 \Rightarrow e_1 \qquad \text{and only of function values}$$

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

(here the y binds in e_1; the x binds in $(\textbf{fn}\ \ y{:}T \Rightarrow e_1)$ and in e_2)

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

Concrete syntax: In ML can write **let fun** $f(x{:}T_1){:}T_2 = e_1$ **in** e_2 **end**,

or even **let fun** $f(x) = e_1$ **in** e_2 **end**, for

let val rec $f{:}T_1 \to T_2 =$ **fn** $x{:}T_1 \Rightarrow e_1$ **in** e_2 **end**.

Recursive Functions – Semantics

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

$\longrightarrow$

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

Recursive Functions – Minimization Example

Below, in the context of the **let val rec** , $\mathrm{x}\ f\ n$ finds the smallest $n' \geq n$ for which $f\ n'$ evaluates to some $m' \leq 0$.

```
let val rec x:(int → int) → int → int
  = fn f:int → int ⇒ fn z:int ⇒ if (f z) ≥ 1 then x f (z + 1) else z
in
  let val f:int → int
    = (fn z:int ⇒ if z ≥ 3 then (if 3 ≥ z then 0 else 1) else 1)
  in
    x f 0
  end
end
```

More Syntactic Sugar

Do we need $e_1; e_2$?

No: Could encode by $e_1; e_2 \;\leadsto\; (\textbf{fn}\;\; y{:}\mathsf{unit} \Rightarrow e_2)\, e_1$

Do we need **while** e_1 **do** e_2?

No: could encode by **while** e_1 **do** $e_2 \leadsto$

$$\begin{array}{l}
\quad \textbf{let}\;\; \textbf{val}\;\; \textbf{rec}\;\; \mathrm{w}{:}\mathsf{unit} \to \mathsf{unit} = \\
\qquad \textbf{fn}\;\; \mathrm{y}{:}\mathsf{unit} \Rightarrow \textbf{if}\;\; e_1\;\; \textbf{then}\, (e_2; (\mathrm{w}\;\; \textbf{skip}))\;\; \textbf{else}\;\; \textbf{skip} \\
\quad \textbf{in} \\
\qquad \mathrm{w}\;\; \textbf{skip} \\
\quad \textbf{end}
\end{array}$$

for fresh w and y not in $\mathsf{fv}(e_1) \cup \mathsf{fv}(e_2)$.

OTOH, Could we encode recursion in the language without?

We know at least that you can't in the language without **while** or store, as had normalisation theorem there and can write

let val rec x:int $\to$ int $=$ **fn** y:int $\Rightarrow \mathrm{x}(\mathrm{y}+1)$ **in** $\mathrm{x}\,0$ **end**

here.

Implementation

There is an implementation of L2 on the course web page.

See especially `Syntax.sml` and `Semantics.sml`. It uses a front end written with mosmllex and mosmlyac.

Implementation – Scope Resolution

```
datatype expr_raw = ...
  | Var_raw of string
  | Fn_raw of string * type_expr * expr_raw
  | App_raw of expr_raw * expr_raw
  | ...

datatype expr = ...
  | Var of int
  | Fn of type_expr * expr
  | App of expr * expr

resolve_scopes : expr_raw -> expr
```

Implementation – Substitution

`subst : expr -> int -> expr -> expr`

`subst e 0 e'` substitutes `e` for the outermost var in `e'`.

(the definition is only sensible if `e` is closed, but that's ok – we only evaluate whole programs. For a general definition, see [Pierce, Ch. 6])

```
fun subst e n (Var n1) = if n=n1 then e else Var n1
  | subst e n (Fn(t,e1)) = Fn(t,subst e (n+1) e1)
  | subst e n (App(e1,e2)) = App(subst e n e1,subst e n e2)
  | subst e n (Let(t,e1,e2))
    = Let (t,subst e n e1,subst e (n+1) e2)


  | subst e n (Letrecfn (tx,ty,e1,e2))
    = Letrecfn (tx,ty,subst e (n+2) e1,subst e (n+1) e2)
  | ...
```

Implementation – CBV reduction

```
reduce (App (e1,e2),s) = (case e1 of
    Fn (t,e) =>
    (if (is_value e2) then
       SOME (subst e2 0 e,s)
     else
       (case reduce (e2,s) of
          SOME(e2',s') => SOME(App (e1,e2'),s')
        | NONE => NONE))
  | _ => (case reduce (e1,s) of
            SOME (e1',s')=>SOME(App(e1',e2),s')
          | NONE => NONE ))
```

Implementation – Type Inference

```
type typeEnv
        = (loc*type_loc) list * type_expr list


inftype gamma (Var n) = nth (#2 gamma) n
inftype gamma (Fn (t,e))
= (case inftype (#1 gamma, t::(#2 gamma)) e of
      SOME t' => SOME (func(t,t') )
    | NONE => NONE )
inftype gamma (App (e1,e2))
= (case (inftype gamma e1, inftype gamma e2) c
      (SOME (func(t1,t1')), SOME t2) =>
          if t1=t2 then SOME t1' else NONE
```

Implementation – Closures

Naively implementing substitution is expensive. An efficient implementation would use *closures* instead – cf. Compiler Construction.

We could give a more concrete semantics, closer to implementation, in terms of closures, and then prove it corresponds to the original semantics...

(if you get that wrong, you end up with dynamic scoping, as in original LISP)

Aside: Small-step vs Big-step Semantics

Throughout this course we use *small-step* semantics, $\langle e, s\rangle \longrightarrow \langle e', s'\rangle$.

There is an alternative style, of *big-step* semantics $\langle e, s\rangle \Downarrow \langle v, s'\rangle$, for example

$$\frac{}{\langle n, s\rangle \Downarrow \langle n, s\rangle} \qquad \frac{\langle e_1, s\rangle \Downarrow \langle n_1, s'\rangle \quad \langle e_2, s'\rangle \Downarrow \langle n_2, s''\rangle}{\langle e_1 + e_2, s\rangle \Downarrow \langle n, s''\rangle \quad n = n_1 + n_2}$$

(see the notes from earlier courses by Andy Pitts).

For sequential languages, it doesn't make a major difference. When we come to add concurrency, small-step is more convenient.

Data – L3

Products

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

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

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) $$\frac{\langle e, s\rangle \longrightarrow \langle e', s'\rangle}{\langle \#2\ e, s\rangle \longrightarrow \langle \#2\ e', s'\rangle}$$

Sums (or Variants, or Tagged Unions)

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

Those xs are binders, treated up to alpha-equivalence.

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

Sums – type annotations

case e **of** **inl** $(x_1{:}T_1) \Rightarrow e_1 \mid$ **inr** $(x_2{:}T_2) \Rightarrow e_2$

Why do we have these type annotations?

To maintain the unique typing property. Otherwise

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

and

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

Sums – reduction

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

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

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

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

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**

Proofs as programs: The Curry-Howard correspondence

(var) $\Gamma, x{:}T \vdash x{:}T$ | $\Gamma, P \vdash P$

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

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

Proofs as programs: The Curry-Howard correspondence

(var) $\Gamma, x{:}T \vdash x{:}T$ $\qquad$ $\Gamma, P \vdash P$

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

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

(pair) $$\frac{\Gamma \vdash e_1{:}T_1 \qquad \Gamma \vdash e_2{:}T_2}{\Gamma \vdash (e_1, e_2){:}T_1 * T_2} \qquad \frac{\Gamma \vdash P_1 \quad \Gamma \vdash P_2}{\Gamma \vdash P_1 \wedge P_2}$$

(proj1) $\dfrac{\Gamma \vdash e{:}T_1 * T_2}{\Gamma \vdash \#1\ e{:}T_1}$ $\qquad$ (proj2) $\dfrac{\Gamma \vdash e{:}T_1 * T_2}{\Gamma \vdash \#2\ e{:}T_2}$ $\qquad$ $\dfrac{\Gamma \vdash P_1 \wedge P_2}{\Gamma \vdash P_1}$ $\qquad$ $\dfrac{\Gamma \vdash P_1 \wedge P_2}{\Gamma \vdash P_2}$

(inl) $$\frac{\Gamma \vdash e{:}T_1}{\Gamma \vdash \textbf{inl}\ \ e{:}T_1 + T_2{:}T_1 + T_2} \qquad \frac{\Gamma \vdash P_1}{\Gamma \vdash P_1 \vee P_2}$$

(inr), (case), (unit), (zero), etc.. – but not (letrec)

ML Datatypes

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

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

is (roughly!) like saying

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

Records

A generalization of products.

Take field labels

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

$$
\begin{array}{lcl}
T & ::= & ... \mid \{lab_1 : T_1, .., lab_k : T_k\} \\
e & ::= & ... \mid \{lab_1 = e_1, .., lab_k = e_k\} \mid \#lab\ e
\end{array}
$$

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

Records – typing

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

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

Records – reduction

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

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

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

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

Mutable Store

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

1 What we've got in L1 and L2:

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

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

  fn $\mathrm{x{:}int} \Rightarrow l := (!l) + \mathrm{x}$

2 In C and Java,

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

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

- have some limited type machinery to limit mutability.

– pros and cons: ....

References

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

References – Typing

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

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

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

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

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

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

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

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 need more: for each $\ell \in \mathrm{dom}(s)$ need that $s(\ell)$ is typable.

Moreover, $s(\ell)$ might contain some other locations...

Type-checking the store – Example

Consider

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

which has reductions

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

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

The statement of type preservation will then be:

Theorem 18 (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 $\Gamma, \Gamma' \vdash s'$.*

Definition 19 (Well-typed store) *Let* $\Gamma \vdash s$ *if* $dom(\Gamma) = dom(s)$ *and if for all* $\ell \in dom(s)$, *if* $\Gamma(\ell) = T$ ref *then* $\Gamma \vdash s(\ell) : T$.

Theorem 20 (Progress) *If* e *closed and* $\Gamma \vdash e : T$ *and* $\Gamma \vdash s$ *then either* e *is a value or there exist* e', s' *such that* $\langle e, s\rangle \longrightarrow \langle e', s'\rangle$.

Theorem 21 (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 22 (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 exist* e'', s'' *such that* $\langle e', s'\rangle \longrightarrow \langle e'', s''\rangle$.

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 than that of L3, though.

Evaluation Contexts

Define *evaluation contexts*

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

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).

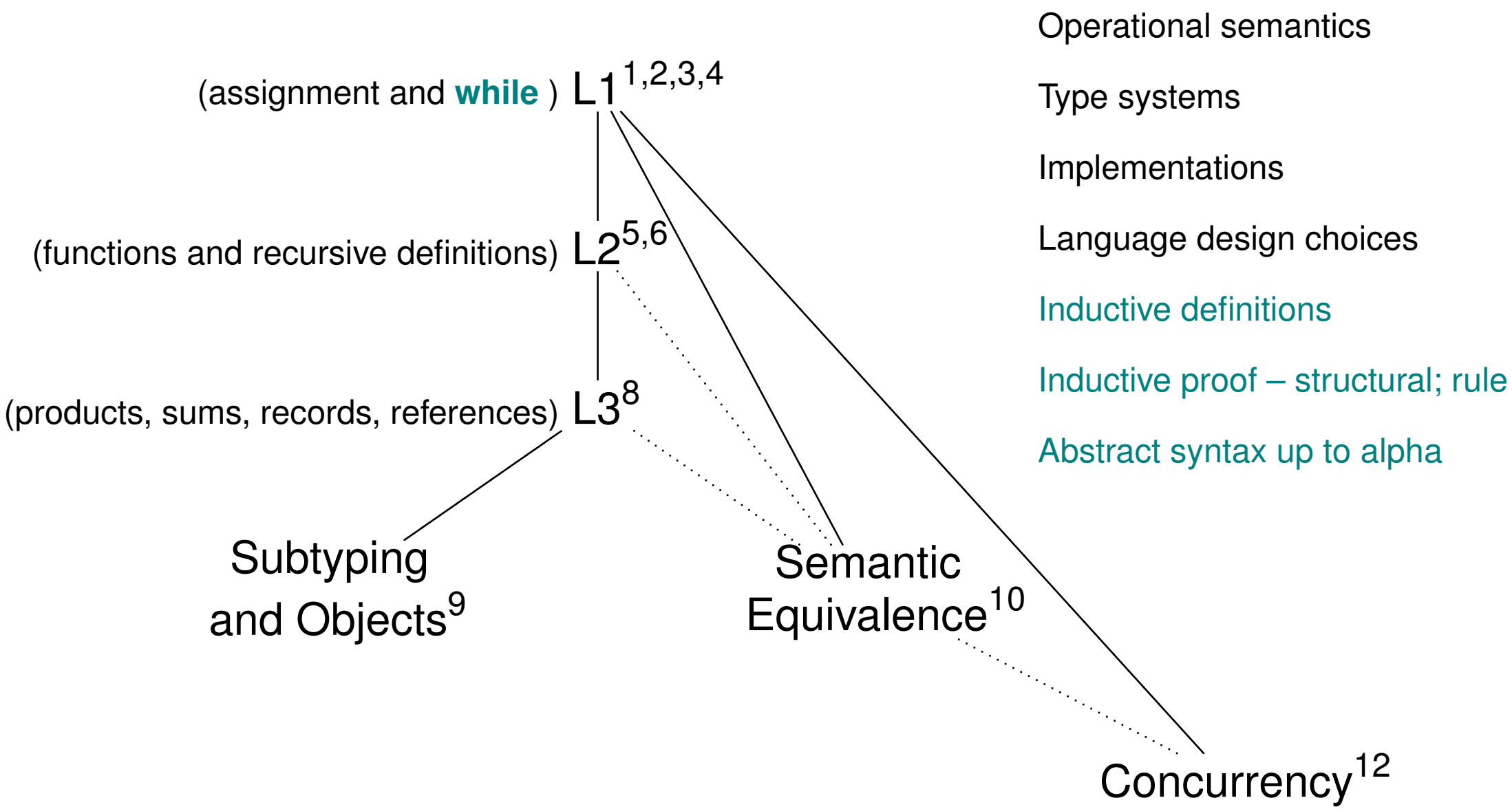
To (eval) we add all the *computation* rules (all the rest) (op $+$), (op $\geq$), (seq1), (if1), (if2), (while), (fn), (let2), (letrecfn), (proj1), (proj2), (case2), (case3), (record2), (ref1), (deref1), (assign1).

Theorem 23 *The two definitions of* $\longrightarrow$ *define the same relation.*

A Little History

Formal logic	1880–
Untyped lambda calculus	1930s
Simply-typed lambda calculus	1940s
Fortran	1950s
Curry-Howard, Algol 60, Algol 68, SECD machine (64)	1960s
Pascal, Polymorphism, ML, PLC	1970s
Structured Operational Semantics	1981–
Standard ML definition	1985
Haskell	1987
Subtyping	1980s
Module systems	1980–
Object calculus	1990–
Typed assembly and intermediate languages	1990–

And now? module systems, distribution, mobility, reasoning about objects, security, typed compilation,.....


(assignment and **while**) L1[1,2,3,4]
(functions and recursive definitions) L2[5,6]
(products, sums, records, references) L3[8]
Subtyping and Objects[9]
Semantic Equivalence[10]
Concurrency[12]
Operational semantics
Type systems
Implementations
Language design choices
Inductive definitions
Inductive proof – structural; rule
Abstract syntax up to alpha

Subtyping and Objects

Polymorphism

Ability to use expressions at many different types.

- Ad-hoc polymorphism (overloading).

  e.g. in Moscow ML the built-in $+$ can be used to add two integers or to add two reals. (see Haskell *type classes*)

- Parametric Polymorphism – as in ML. See the Part II Types course.

  can write a function that for any type α takes an argument of type α list and computes its length (parametric – uniform in whatever α is)

- Subtype polymorphism – as in various OO languages. See here.

  Dating back to the 1960s (Simula etc); formalized in 1980,1984,...

Subtyping – Motivation

Recall

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

so can't type

$$\not\vdash (\textbf{fn}\ \mathrm{x}{:}\{\mathrm{p}{:}\mathsf{int}\} \Rightarrow \#\mathrm{p}\ \mathrm{x})\ \{\mathrm{p} = 3, \mathrm{q} = 4\} : \mathsf{int}$$

even though we're giving the function a *better* argument, with more structure, than it needs.

Subsumption

'Better'? Any value of type $\{\mathrm{p{:}int}, \mathrm{q{:}int}\}$ can be used wherever a value of type $\{\mathrm{p{:}int}\}$ is expected. (*)

Introduce a *subtyping relation* between types, written $T <: T'$, read as T is a subtype of T' (a T is useful in more contexts than a T').

Will define it on the next slides, but it will include

$$\{\mathrm{p{:}int}, \mathrm{q{:}int}\} <: \{\mathrm{p{:}int}\} <: \{\}$$

Introduce a *subsumption rule*

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

allowing subtyping to be used, capturing (*).

Can then deduce $\{\mathrm{p} = 3, \mathrm{q} = 4\}{:}\{\mathrm{p{:}int}\}$, hence can type the example.

Example

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x{:}\{p{:}\mathsf{int}\} \vdash x{:}\{p{:}\mathsf{int}\}}\,(\text{var})}{x{:}\{p{:}\mathsf{int}\} \vdash \#p\ x{:}\mathsf{int}}\,(\text{record-proj})}{\{\} \vdash (\mathbf{fn}\ x{:}\{p{:}\mathsf{int}\} \Rightarrow \#p\ x){:}\{p{:}\mathsf{int}\} \to \mathsf{int}}\,(\text{fn}) \qquad \dfrac{\dfrac{\dfrac{}{\{\} \vdash 3{:}\mathsf{int}}\,(\text{var}) \quad \dfrac{}{\{\} \vdash 4{:}\mathsf{int}}\,(\text{var})}{\{\} \vdash \{p=3, q=4\}{:}\{p{:}\mathsf{int}, q{:}\mathsf{int}\}}\,(\text{record}) \quad (\bigstar)}{\{\} \vdash \{p=3, q=4\}{:}\{p{:}\mathsf{int}\}}\,(\text{sub})}{\{\} \vdash (\mathbf{fn}\ x{:}\{p{:}\mathsf{int}\} \Rightarrow \#p\ x)\{p=3, q=4\}{:}\mathsf{int}}\,(\text{app})$$

where $(\bigstar)$ is $\{p{:}\mathsf{int}, q{:}\mathsf{int}\} <: \{p{:}\mathsf{int}\}$

The Subtype Relation $\boxed{T <: T'}$

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

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

Subtyping – Records

Forgetting fields on the right:

$$\{lab_1 : T_1, .., lab_k : T_k, lab_{k+1} : T_{k+1}, .., lab_{k+k'} : T_{k+k'}\}$$

$$<: \qquad \text{(s-record-width)}$$

$$\{lab_1 : T_1, .., lab_k : T_k\}$$

Allowing subtyping within fields:

$$\text{(s-record-depth)} \quad \frac{T_1 <: T_1' \quad .. \quad T_k <: T_k'}{\{lab_1 : T_1, .., lab_k : T_k\} <: \{lab_1 : T_1', .., lab_k : T_k'\}}$$

Combining these:

$$\dfrac{\dfrac{}{\{\text{p:int}, \text{q:int}\} <: \{\text{p:int}\}}\ \text{(s-record-width)} \quad \dfrac{}{\{\text{r:int}\} <: \{\}}\ \text{(s-record-width)}}{\{\text{x:}\{\text{p:int}, \text{q:int}\}, \text{y:}\{\text{r:int}\}\} <: \{\text{x:}\{\text{p:int}\}, \text{y:}\{\}\}}\ \text{(s-record-depth)}$$

Allowing reordering of fields:

(s-record-order)

$$\frac{\pi \text{ a permutation of } 1, .., k}{\{lab_1 : T_1, .., lab_k : T_k\} <: \{lab_{\pi(1)} : T_{\pi(1)}, .., lab_{\pi(k)} : T_{\pi(k)}\}}$$

(the subtype order is *not* anti-symmetric – it is a preorder, not a partial order)

Subtyping – Functions

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

contravariant on the left of $\to$

covariant on the right of $\to$ (like (s-record-depth))

If $f : T_1 \to T_2$ then we can give f any argument which is a subtype of T_1; we can regard the result of f as any supertype of T_2. e.g., for

$$f = \textbf{fn}\ \mathrm{x}{:}\{\mathrm{p}{:}\mathsf{int}\} \Rightarrow \{\mathrm{p} = \#\mathrm{p}\ \mathrm{x}, \mathrm{q} = 28\}$$

we have

$$\{\} \vdash f{:}\{\mathrm{p}{:}\mathsf{int}\} \to \{\mathrm{p}{:}\mathsf{int}, \mathrm{q}{:}\mathsf{int}\}$$

$$\{\} \vdash f{:}\{\mathrm{p}{:}\mathsf{int}\} \to \{\mathrm{p}{:}\mathsf{int}\}$$

$$\{\} \vdash f{:}\{\mathrm{p}{:}\mathsf{int}, \mathrm{q}{:}\mathsf{int}\} \to \{\mathrm{p}{:}\mathsf{int}, \mathrm{q}{:}\mathsf{int}\}$$

$$\{\} \vdash f{:}\{\mathrm{p}{:}\mathsf{int}, \mathrm{q}{:}\mathsf{int}\} \to \{\mathrm{p}{:}\mathsf{int}\}$$

as

$$\{\mathrm{p}{:}\mathsf{int}, \mathrm{q}{:}\mathsf{int}\} <: \{\mathrm{p}{:}\mathsf{int}\}$$

On the other hand, for

$$\textbf{fn}\ \ \mathrm{x}{:}\{\mathrm{p}{:}\mathsf{int}, \mathrm{q}{:}\mathsf{int}\} \Rightarrow \{\mathrm{p} = (\#\mathrm{p}\ \mathrm{x}) + (\#\mathrm{q}\ \mathrm{x})\}$$

we have

$$\{\} \vdash f{:}\{\mathrm{p}{:}\mathsf{int}, \mathrm{q}{:}\mathsf{int}\} \to \{\mathrm{p}{:}\mathsf{int}\}$$

$$\{\} \nvdash f{:}\{\mathrm{p}{:}\mathsf{int}\} \to T \quad \text{for any } T$$

$$\{\} \nvdash f{:}T \to \{\mathrm{p}{:}\mathsf{int}, \mathrm{q}{:}\mathsf{int}\} \quad \text{for any } T$$

Subtyping – Products

Just like (s-record-depth)

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

Subtyping – Sums

Exercise.

Subtyping – References

Are either of these any good?

$$\frac{T <: T'}{T \ \text{ref} <: T' \ \text{ref}} \qquad \frac{T' <: T}{T \ \text{ref} <: T' \ \text{ref}}$$

No...

Semantics

No change (note that we've not changed the expression grammar).

Properties

Have Type Preservation and Progress.

Implementation

Type inference is more subtle, as the rules are no longer syntax-directed.

Getting a good runtime implementation is also tricky, especially with field re-ordering.

Subtyping – Down-casts

The subsumption rule (sub) permits up-casting at any point. How about down-casting? We could add

$$e \quad ::= \quad \ldots \mid (T)e$$

with typing rule

$$\frac{\Gamma \vdash e{:}T'}{\Gamma \vdash (T)e{:}T}$$

then you need a dynamic type-check...

This gives flexibility, but at the cost of many potential run-time errors. Many uses might be better handled by Parametric Polymorphism, aka Generics. (cf. work by Martin Odersky at EPFL, Lausanne, now in Java 1.5)

(Very Simple) Objects

$$
\begin{array}{l}
\textbf{let val } \mathrm{c}{:}\{\mathrm{get}{:}\mathsf{unit} \to \mathsf{int},\ \mathrm{inc}{:}\mathsf{unit} \to \mathsf{unit}\} = \\
\quad \textbf{let val } \mathrm{x}{:}\mathsf{int} \ \mathrm{ref} = \ \mathrm{ref}\ 0 \ \textbf{in} \\
\qquad \{\mathrm{get} = \textbf{fn } \mathrm{y}{:}\mathsf{unit} \Rightarrow !\mathrm{x}, \\
\qquad\ \mathrm{inc} = \textbf{fn } \mathrm{y}{:}\mathsf{unit} \Rightarrow \mathrm{x} := 1{+}!\mathrm{x}\} \\
\quad \textbf{end} \\
\textbf{in} \\
\quad (\#\mathrm{inc}\ \mathrm{c})(); (\#\mathrm{get}\ \mathrm{c})() \\
\textbf{end}
\end{array}
$$

$$
\mathit{Counter} = \{\mathrm{get}{:}\mathsf{unit} \to \mathsf{int},\ \mathrm{inc}{:}\mathsf{unit} \to \mathsf{unit}\}.
$$

Using Subtyping

```
let val c:{get:unit → int, inc:unit → unit, reset:unit → unit} =
    let val x:int ref = ref 0 in
        {get = fn y:unit ⇒!x,
         inc = fn y:unit ⇒ x := 1+!x,
         reset = fn y:unit ⇒ x := 0}
    end
in
   (#inc c)(); (#get c)()
end
```

$ResetCounter = \{\text{get:unit} \to \text{int}, \text{inc:unit} \to \text{unit}, \text{reset:unit} \to \text{unit}\}$

$<: Counter = \{\text{get:unit} \to \text{int}, \text{inc:unit} \to \text{unit}\}.$

Object Generators

```
let val newCounter:unit → {get:unit → int, inc:unit → unit} =
    fn y:unit ⇒
        let val x:int ref = ref 0 in
            {get = fn y:unit ⇒!x,
             inc = fn y:unit ⇒ x := 1+!x}
        end
in
   (#inc (newCounter ())) ()
end
```

and onwards to simple classes...

Reusing Method Code (Simple Classes)

Recall $Counter = \{\mathrm{get}{:}\mathsf{unit} \to \mathsf{int},\ \mathrm{inc}{:}\mathsf{unit} \to \mathsf{unit}\}$.

First, make the internal state into a record.

$$CounterRep = \{\text{p:int ref}\}.$$

$$\textbf{let val } \text{counterClass:}CounterRep \to Counter =$$
$$\quad \textbf{fn } \text{x:}CounterRep \Rightarrow$$
$$\quad\quad \{\text{get} = \textbf{fn } \text{y:unit} \Rightarrow !(\#\text{p x}),$$
$$\quad\quad\ \text{inc} = \textbf{fn } \text{y:unit} \Rightarrow (\#\text{p x}) := 1 + !(\#\text{p x})\}$$

$$\textbf{let val } \text{newCounter:unit} \to Counter =$$
$$\quad \textbf{fn } \text{y:unit} \Rightarrow$$
$$\quad\quad \textbf{let val } \text{x:}CounterRep = \{\text{p} = \textbf{ ref } 0\} \textbf{ in}$$
$$\quad\quad\quad \text{counterClass x}$$

Reusing Method Code (Simple Classes)

$$\begin{array}{l}\textbf{let val } \mathrm{resetCounterClass}{:}\mathit{CounterRep} \to \mathit{ResetCounter} = \\ \quad \textbf{fn } \mathrm{x}{:}\mathit{CounterRep} \Rightarrow \\ \quad\quad \textbf{let val } \mathrm{super} = \mathrm{counterClass\ x} \textbf{ in} \\ \quad\quad\quad \{\mathrm{get} = \#\mathrm{get\ super}, \\ \quad\quad\quad\ \mathrm{inc} = \#\mathrm{inc\ super}, \\ \quad\quad\quad\ \mathrm{reset} = \textbf{fn } \mathrm{y{:}unit} \Rightarrow (\#\mathrm{p\ x}) := 0\}\end{array}$$

$\mathit{CounterRep} = \{\mathrm{p{:}int\ ref}\}.$

$\mathit{Counter} = \{\mathrm{get{:}unit} \to \mathrm{int},\ \mathrm{inc{:}unit} \to \mathrm{unit}\}.$

$\mathit{ResetCounter} = \{\mathrm{get{:}unit} \to \mathrm{int},\ \mathrm{inc{:}unit} \to \mathrm{unit},\ \mathrm{reset{:}unit} \to \mathrm{unit}\}.$

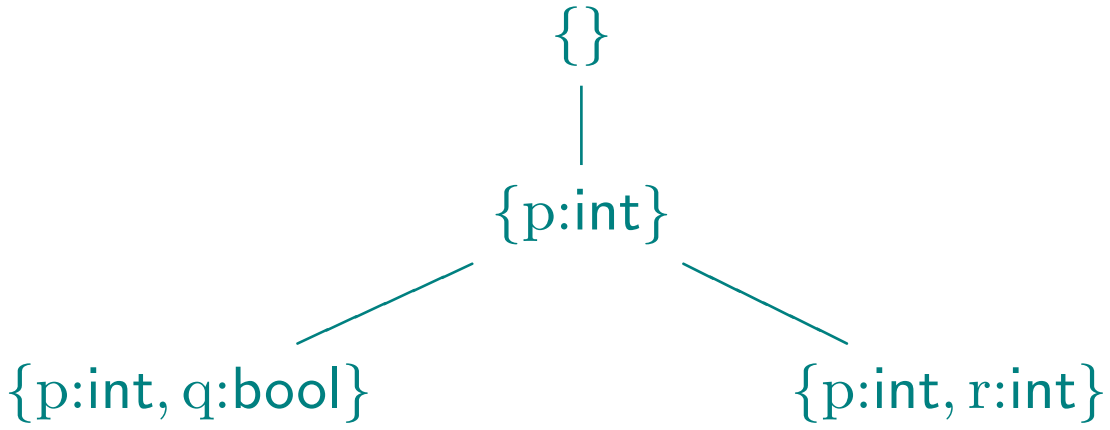
Reusing Method Code (Simple Classes)

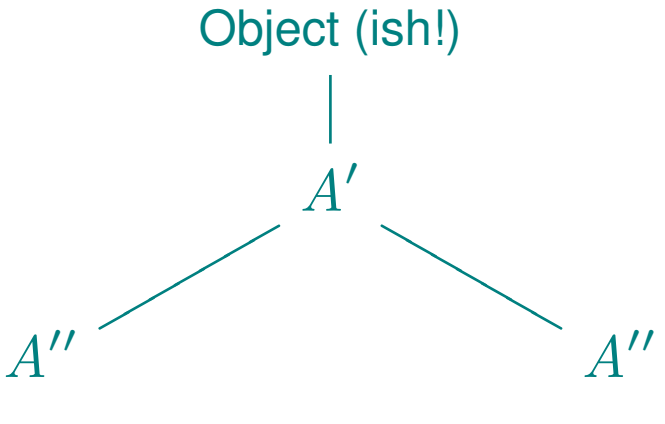
```
class Counter
  { protected int p;
     Counter() { this.p=0; }
     int get () { return this.p; }
     void inc () { this.p++ ; }
     };

class ResetCounter
  extends Counter
  { void reset () {this.p=0;}
     };
```

Subtyping – Structural vs Named

$$
\begin{aligned}
A' &= \{\} \text{ with } \{\text{p:int}\} \\
A'' &= A' \text{ with } \{\text{q:bool}\} \\
A''' &= A' \text{ with } \{\text{r:int}\}
\end{aligned}
$$





Concurrency

Our focus so far has been on semantics for *sequential* computation. But the world is not sequential...

- hardware is intrinsically parallel (fine-grain, across words, to coarse-grain, e.g. multiple execution units)
- multi-processor machines
- multi-threading (perhaps on a single processor)
- networked machines

Problems

- the state-spaces of our systems become *large*, with the *combinatorial explosion* – with n threads, each of which can be in 2 states, the system has 2^n states.
- the state-spaces become *complex*
- computation becomes *nondeterministic* (unless synchrony is imposed), as different threads operate at different speeds.
- parallel components competing for access to resources may *deadlock* or suffer *starvation*. Need *mutual exclusion* between components accessing a resource.

More Problems!

- *partial failure* (of some processes, of some machines in a network, of some persistent storage devices). Need *transactional mechanisms*.
- *communication between different environments* (with different local resources (e.g. different local stores, or libraries, or...)
- *partial version change*
- communication between administrative regions with *partial trust* (or, indeed, *no trust*); protection against mailicious attack.
- dealing with contingent complexity (embedded historical accidents; upwards-compatible deltas)

Theme: as for sequential languages, but much more so, it's a complicated world.

Aim of this lecture: just to give you a taste of how a little semantics can be used to express some of the fine distinctions. Primarily (1) to boost your intuition for informal reasoning, but also (2) this can support rigorous proof about really hairy crypto protocols, cache-coherency protocols, comms, database transactions,....

Going to define the simplest possible concurrent language, call it $L1_{|}$, and explore a few issues. You've seen most of them informally in C&DS.

Booleans $b \in \mathbb{B} = \{\textbf{true}, \textbf{false}\}$

Integers $n \in \mathbb{Z} = \{..., -1, 0, 1, ...\}$

Locations $\ell \in \mathbb{L} = \{l, l_0, l_1, l_2, ...\}$

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

Expressions

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

$$
\begin{array}{lll}
T & ::= & \mathsf{int} \mid \mathsf{bool} \mid \mathsf{unit} \mid \mathsf{proc} \\
T_{loc} & ::= & \mathsf{intref}
\end{array}
$$

Parallel Composition: Typing and Reduction

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

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

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

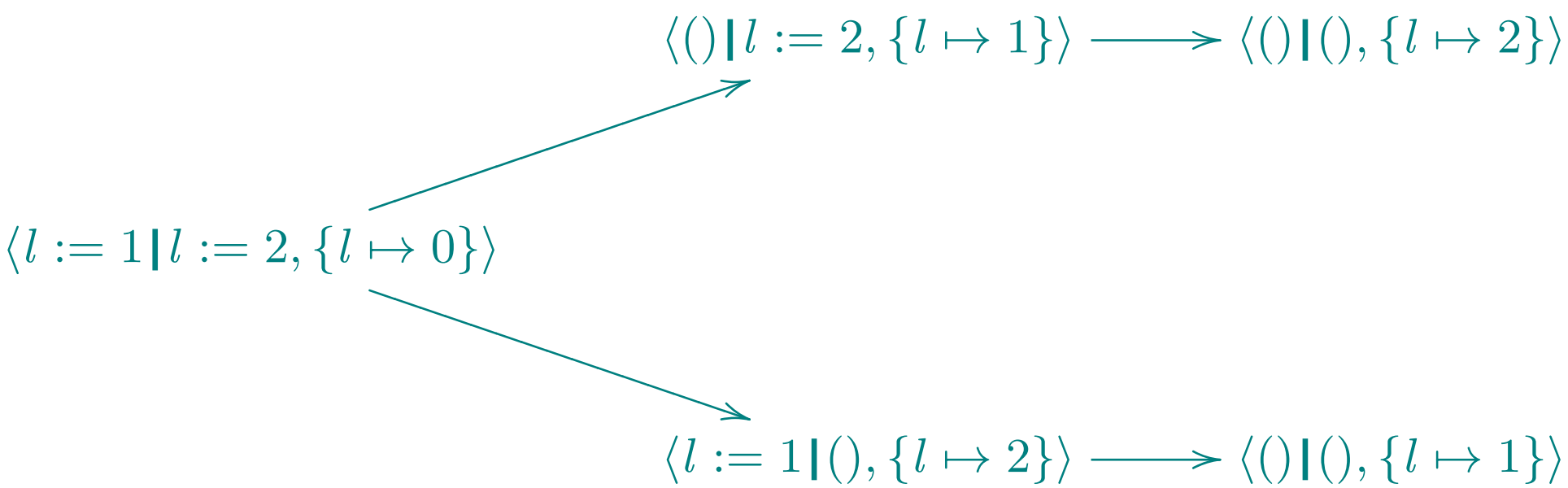
(parallel2) $$\frac{\langle e_2, s\rangle \longrightarrow \langle e_2', s'\rangle}{\langle e_1 \mid e_2, s\rangle \longrightarrow \langle e_1 \mid e_2', s'\rangle}$$

Parallel Composition: Design Choices

- threads don’t return a value
- threads don’t have an identity
- termination of a thread cannot be observed within the language
- threads aren’t partitioned into ‘processes’ or machines
- threads can’t be killed externally

Threads execute asynchronously – the semantics allows any interleaving of the reductions of the threads.

All threads can read and write the shared memory.



But, assignments and dereferencing are *atomic*. For example,

$$\langle l := 34987345908792384293 84 \mid l := 7, \{l \mapsto 0\}\rangle$$

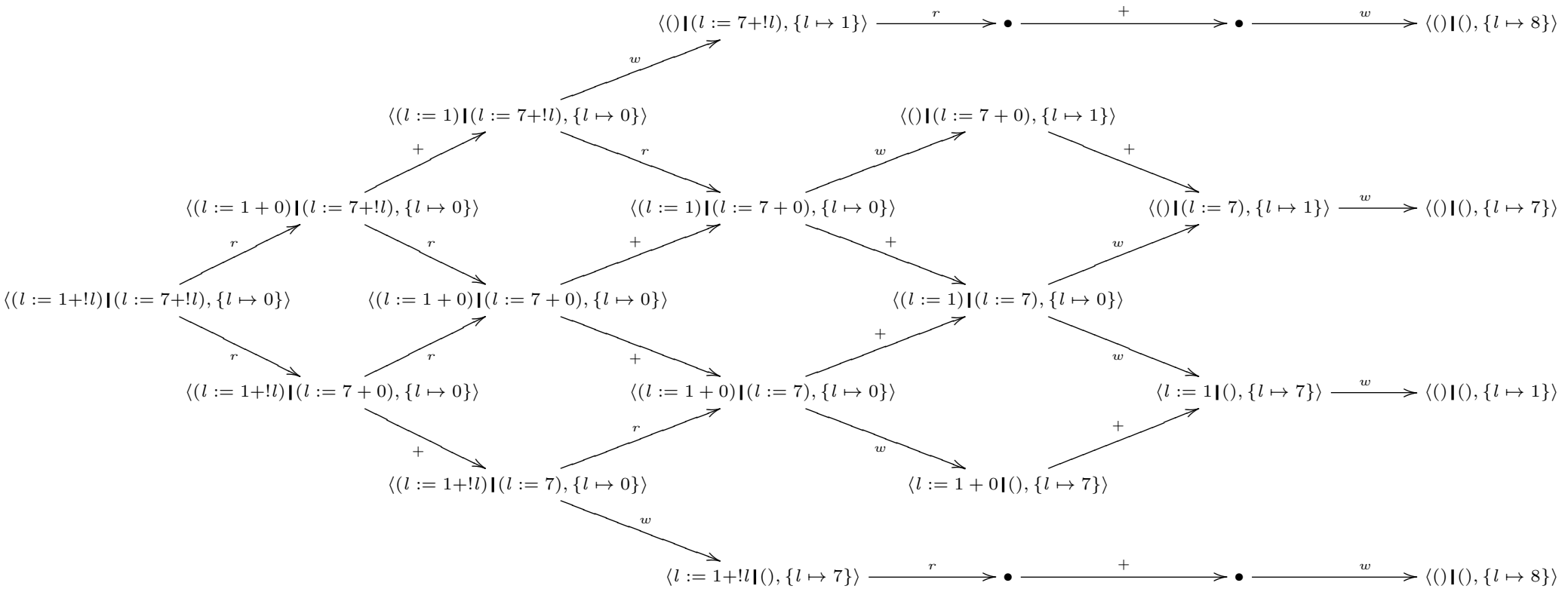
will reduce to a state with l either $34987345908792384293 84$ or 7, not something with the first word of one and the second word of the other.

Implement?

But but, in $(l := e) \mid e'$, the steps of evaluating e and e' can be interleaved.

Think of $(l := 1 + !l) \mid (l := 7 + !l)$ – there are races....

The behaviour of $(l := 1{+}!l) \mid (l := 7{+}!l)$ for the initial store $\{l \mapsto 0\}$:



Morals

- There is a combinatorial explosion.
- Drawing state-space diagrams only works for really tiny examples – we need better techniques for analysis.
- Almost certainly you (as the programmer) didn't want all those 3 outcomes to be possible – need better idioms or constructs for programming.

So, how do we get anything coherent done?

Need some way(s) to synchronize between threads, so can enforce *mutual exclusion* for shared data.

cf. Lamport's "Bakery" algorithm from Concurrent and Distributed Systems. Can you code that in $L1_|$? If not, what's the smallest extension required?

Usually, though, you can depend on built-in support from the scheduler, e.g. for *mutexes* and *condition variables* (or, at a lower level, `tas` or `cas`).

Adding Primitive Mutexes

Mutex names $m \in \mathbb{M} = \{\mathrm{m}, \mathrm{m}_1, ...\}$

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

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

(lock) $\dfrac{}{\Gamma \vdash \textbf{lock}\ \ m{:}\mathsf{unit}}$ (unlock) $\dfrac{}{\Gamma \vdash \textbf{unlock}\ \ m{:}\mathsf{unit}}$

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

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

Need to adapt all the other semantic rules to carry the mutex state M around. For example, replace

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

by

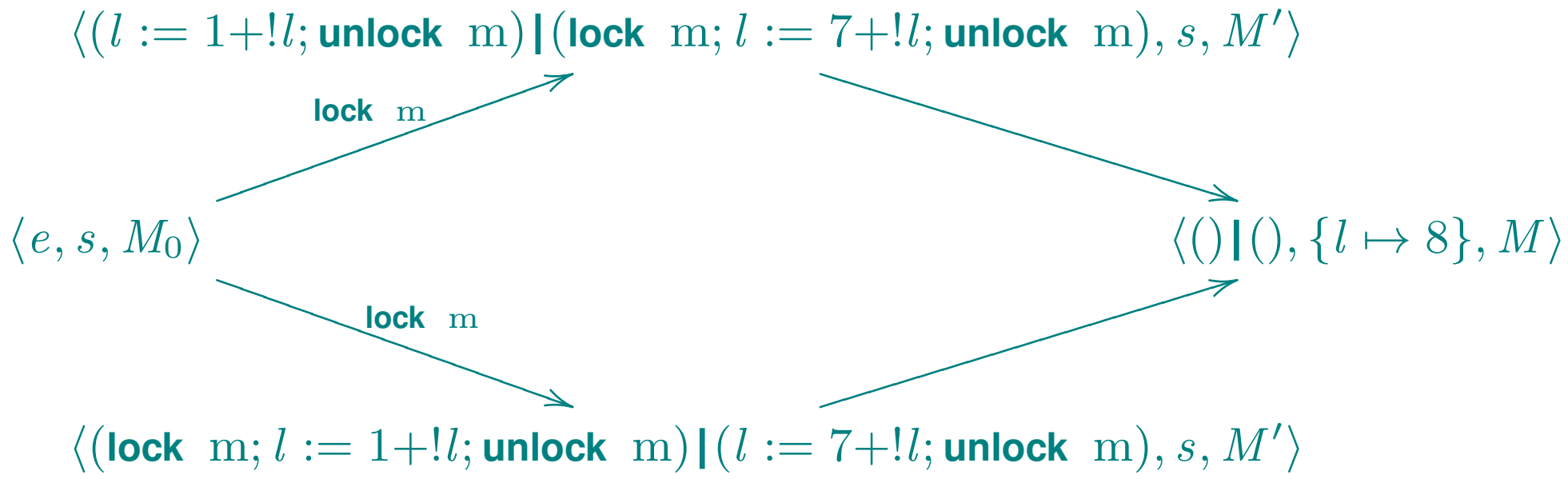
$$\text{(op2)} \quad \frac{\langle e_2, s, M\rangle \longrightarrow \langle e_2', s', M'\rangle}{\langle v \;\; op \;\; e_2, s, M\rangle \longrightarrow \langle v \;\; op \;\; e_2', s', M'\rangle}$$

Using a Mutex

Consider

$$e = (\textbf{lock}\ \ \mathrm{m}; l := 1{+}!l; \textbf{unlock}\ \ \mathrm{m}) \,|\, (\textbf{lock}\ \ \mathrm{m}; l := 7{+}!l; \textbf{unlock}\ \ \mathrm{m})$$

The behaviour of $\langle e, s, M \rangle$, with the initial store $s = \{l \mapsto 0\}$ and initial mutex state $M_0 = \lambda m \in \mathbb{M}.\textbf{false}$, is:



(where $M' = M_0 + \{\mathrm{m} \mapsto \textbf{true}\}$)

Using Several Mutexes

lock m can block (that's the point). Hence, you can *deadlock*.

$$
\begin{aligned}
e = \quad & (\textbf{lock}\ \mathrm{m}_1; \textbf{lock}\ \mathrm{m}_2; l_1 :=! l_2; \textbf{unlock}\ \mathrm{m}_1; \textbf{unlock}\ \mathrm{m}_2) \\
\mid \quad & (\textbf{lock}\ \mathrm{m}_2; \textbf{lock}\ \mathrm{m}_1; l_2 :=! l_1; \textbf{unlock}\ \mathrm{m}_1; \textbf{unlock}\ \mathrm{m}_2)
\end{aligned}
$$

Locking Disciplines

So, suppose we have several programs $e_1, \ldots, e_k$, all well-typed with $\Gamma \vdash e_i : \mathsf{unit}$, that we want to execute concurrently without 'interference' (whatever that is). Think of them as transaction bodies.

There are many possible locking disciplines. We'll focus on one, to see how it – and the properties it guarantees – can be made precise and proved.

An Ordered 2PL Discipline, Informally

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

Fix a lock acquisition order. For simplicity, make it $\mathrm{m}, \mathrm{m}_0, \mathrm{m}_1, \mathrm{m}_2, \ldots$.

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 (two-phase)
- acquires locks in increasing order

Then, informally, $(e_1 | \ldots | e_k)$ should (a) never deadlock, and (b) be *serializable* – any execution of it should be 'equivalent' to an execution of $e_{\pi(1)}; \ldots; e_{\pi(k)}$ for some permutation π.

Problem: Need a Thread-Local Semantics

Our existing semantics defines the behaviour only of global configurations $\langle e, s, M \rangle$. To state properties of subexpressions, e.g.

- e_i acquires the lock m_j for each location l_j it uses, before it uses it

which really means

- in any execution of $\langle (e_1 \mid \ldots \mid e_i \mid \ldots \mid e_k), s, M \rangle$, e_i acquires the lock m_j for each location l_j it uses, before it uses it

we need some notion of the behaviour of the thread e_i on its own

Solution: Thread local semantics

Instead of only defining the global $\langle e, s, M \rangle \longrightarrow \langle e', s', M' \rangle$, with rules

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

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

define a per-thread $e \xrightarrow{a} e'$ and use that to define $\langle e, s, M \rangle \longrightarrow \langle e', s', M' \rangle$, with rules like

(t-assign1) $\ell := n \xrightarrow{\ell := n} \textbf{skip}$

(t-parallel1) $$\frac{e_1 \xrightarrow{a} e_1'}{e_1 \,|\, e_2 \xrightarrow{a} e_1' \,|\, e_2}$$

(c-assign) $$\frac{e \xrightarrow{\ell := n} e' \quad \ell \in \mathrm{dom}(s)}{\langle e, s, M \rangle \longrightarrow \langle e', s + \{\ell \mapsto n\}, M \rangle}$$

Note the per-thread rules don't mention s or M. Instead, we record in the label a what interactions with the store or mutexes it has.

$$a \quad ::= \quad \tau \mid \ell := n \mid !\ell = n \mid \textbf{lock} \;\; m \mid \textbf{unlock} \;\; m$$

Conventionally, τ (tau), stands for "no interactions", so $e \xrightarrow{\tau} e'$ if e does an internal step, not involving the store or mutexes.

Theorem 24 (Coincidence of global and thread-local semantics) *The two definitions of* $\longrightarrow$ *agree exactly.*

Proof strategy: a couple of rule inductions.

Example of Thread-local transitions

For $e = (\textbf{lock}\ \ \mathrm{m}; (l := 1{+}!l; \textbf{unlock}\ \ \mathrm{m}))$ we have

$$
\begin{aligned}
e \quad &\xrightarrow{\textbf{lock}\ \ \mathrm{m}} \quad \textbf{skip}; (l := 1{+}!l; \textbf{unlock}\ \ \mathrm{m}) \\
&\xrightarrow{\tau} \quad (l := 1{+}!l; \textbf{unlock}\ \ \mathrm{m}) \\
&\xrightarrow{!l=n} \quad (l := 1 + n; \textbf{unlock}\ \ \mathrm{m}) \qquad \text{for any } n \in \mathbb{Z} \\
&\xrightarrow{\tau} \quad (l := n'; \textbf{unlock}\ \ \mathrm{m}) \qquad \text{for } n' = 1 + n \\
&\xrightarrow{l:=n'} \quad \textbf{skip}; \textbf{unlock}\ \ \mathrm{m} \\
&\xrightarrow{\tau} \quad \textbf{unlock}\ \ \mathrm{m} \\
&\xrightarrow{\textbf{unlock}\ \ \mathrm{m}} \quad \textbf{skip}
\end{aligned}
$$

Hence, using (t-parallel) and the (c-*) rules, for $s' = s + \{l \mapsto 1 + s(l)\}$,

$$\langle e \,|\, e', s, M_0\rangle \longrightarrow\longrightarrow\longrightarrow\longrightarrow\longrightarrow\longrightarrow\longrightarrow \langle \textbf{skip} \,|\, e', s', M_0\rangle$$

Global Semantics

(op $+$) $\langle n_1 + n_2, s, M\rangle \longrightarrow \langle n, s, M\rangle$ if $n = n_1 + n_2$

(op $\geq$) $\langle n_1 \geq n_2, s, M\rangle \longrightarrow \langle b, s, M\rangle$ if $b = (n_1 \geq n_2)$

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

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

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

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

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

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

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

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

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

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

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

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

(parallel2) $$\frac{\langle e_2, s, M\rangle \longrightarrow \langle e_2', s', M'\rangle}{\langle e_1 \,|\, e_2, s, M\rangle \longrightarrow \langle e_1 \,|\, e_2', s', M'\rangle}$$

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

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

Thread-Local Semantics

(t-op $+$) $n_1 + n_2 \xrightarrow{\tau} n$ if $n = n_1 + n_2$

(t-op $\geq$) $n_1 \geq n_2 \xrightarrow{\tau} b$ if $b = (n_1 \geq n_2)$

(t-op1) $$\frac{e_1 \xrightarrow{a} e_1'}{e_1 \;\; op \;\; e_2 \xrightarrow{a} e_1' \;\; op \;\; e_2}$$

(t-op2) $$\frac{e_2 \xrightarrow{a} e_2'}{v \;\; op \;\; e_2 \xrightarrow{a} v \;\; op \;\; e_2'}$$

(t-deref) $!\ell \xrightarrow{!\ell = n} n$

(t-assign1) $\ell := n \xrightarrow{\ell := n} \textbf{skip}$

(t-assign2) $$\frac{e \xrightarrow{a} e'}{\ell := e \xrightarrow{a} \ell := e'}$$

(t-seq1) $\textbf{skip}; e_2 \xrightarrow{\tau} e_2$

(t-seq2) $$\frac{e_1 \xrightarrow{a} e_1'}{e_1; e_2 \xrightarrow{a} e_1'; e_2}$$

(t-if1) $\textbf{if } \textbf{true } \textbf{then } e_2 \textbf{ else } e_3 \xrightarrow{\tau} e_2$

(t-if2) $\textbf{if } \textbf{false } \textbf{then } e_2 \textbf{ else } e_3 \xrightarrow{\tau} e_3$

(t-if3) $$\frac{e_1 \xrightarrow{a} e_1'}{\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \xrightarrow{a} \textbf{if } e_1' \textbf{ then } e_2 \textbf{ else } e_3}$$

(t-while)
$\textbf{while } e_1 \textbf{ do } e_2 \xrightarrow{\tau} \textbf{if } e_1 \textbf{ then } (e_2; \textbf{while } e_1 \textbf{ do } e_2) \textbf{ else skip}$

(t-parallel1) $$\frac{e_1 \xrightarrow{a} e_1'}{e_1 \,|\, e_2 \xrightarrow{a} e_1' \,|\, e_2}$$

(t-parallel2) $$\frac{e_2 \xrightarrow{a} e_2'}{e_1 \,|\, e_2 \xrightarrow{a} e_1 \,|\, e_2'}$$

(t-lock) $\textbf{lock } m \xrightarrow{\textbf{lock } m} ()$

(t-unlock) $\textbf{unlock } m \xrightarrow{\textbf{unlock } m} ()$

(c-tau) $$\frac{e \xrightarrow{\tau} e'}{\langle e, s, M\rangle \longrightarrow \langle e', s, M\rangle}$$

(c-assign) $$\frac{e \xrightarrow{\ell := n} e' \quad \ell \in \mathsf{dom}(s)}{\langle e, s, M\rangle \longrightarrow \langle e', s + \{\ell \mapsto n\}, M\rangle}$$

(c-lock) $$\frac{e \xrightarrow{\textbf{lock } m} e' \quad \neg M(m)}{\langle e, s, M\rangle \longrightarrow \langle e', s, M + \{m \mapsto \textbf{true}\}\rangle}$$

(c-deref) $$\frac{e \xrightarrow{!\ell = n} e' \quad \ell \in \mathsf{dom}(s) \wedge s(\ell) = n}{\langle e, s, M\rangle \longrightarrow \langle e', s, M\rangle}$$

(c-unlock) $$\frac{e \xrightarrow{\textbf{unlock } m} e'}{\langle e, s, M\rangle \longrightarrow \langle e', s, M + \{m \mapsto \textbf{false}\}\rangle}$$

Now can make the Ordered 2PL Discipline precise

Say e obeys the discipline if for any (finite or infinite)
$e \xrightarrow{a_1} e_1 \xrightarrow{a_2} e_2 \xrightarrow{a_3} \ldots$

- if a_i is $(l_j := n)$ or $(!l_j = n)$ then for some $k < i$ we have $a_k =$ **lock** m_j without an intervening **unlock** m_j.
- for each j, the subsequence of $a_1, a_2, \ldots$ with labels **lock** m_j and **unlock** m_j is a prefix of $((\textbf{lock}\ \mathrm{m}_j)(\textbf{unlock}\ \mathrm{m}_j))^*$. Moreover, if $\neg(e_k \xrightarrow{a})$ then the subsequence does not end in a **lock** m_j.
- if $a_i =$ **lock** m_j and $a_{i'} =$ **unlock** $\mathrm{m}_{j'}$ then $i < i'$
- if $a_i =$ **lock** m_j and $a_{i'} =$ **lock** $\mathrm{m}_{j'}$ and $i < i'$ then $j < j'$

... and make the guaranteed properties precise

Say $e_1, \ldots, e_k$ are *serializable* if for any initial store s, if $\langle (e_1 | \ldots | e_k), s, M_0 \rangle \longrightarrow^* \langle e', s', M' \rangle \not\longrightarrow$ then for some permutation π we have $\langle e_{\pi(1)}; \ldots; e_{\pi(k)}, s, M_0 \rangle \longrightarrow^* \langle e'', s', M' \rangle$.

Say they are *deadlock-free* if for any initial store s, if $\langle (e_1 | \ldots | e_k), s, M_0 \rangle \longrightarrow^* \langle e', s', M \rangle \not\longrightarrow$ then not $e' \xrightarrow{\textbf{lock}\ m} e''$, i.e. e' does not contain any blocked **lock** m subexpressions.

(Warning: there are many subtle variations of these properties!)

The Theorem

Conjecture 25 *If each e_i obeys the discipline, then $e_1, \ldots e_k$ are serializable and deadlock-free.*

(may be false!)

Proof strategy: Consider a (derivation of a) computation

$$\langle (e_1 | \ldots | e_k), s, M_0 \rangle \longrightarrow \langle \hat{e}_1, s_1, M_1 \rangle \longrightarrow \langle \hat{e}_2, s_2, M_2 \rangle \longrightarrow \ldots$$

We know each $\hat{e}_i$ is a corresponding parallel composition. Look at the points at which each e_i acquires its final lock. That defines a serialization order. In between times, consider commutativity of actions of the different e_i – the premises guarantee that many actions are semantically independent, and so can be permuted.

We've not discussed *fairness* – the semantics allows any interleaving between parallel components, not only fair ones.

Language Properties

(Obviously!) don't have Determinacy.

Still have Type Preservation.

Have Progress, but it has to be modified – a well-typed expression of type proc will reduce to some parallel composition of unit values.

Typing and type inference is scarcely changed.

(*very* fancy type systems can be used to enforce locking disciplines)

Semantic Equivalence

$$2 + 2 \stackrel{?}{\simeq} 4$$

In what sense are these two expressions the same?

They have different abstract syntax trees.

They have different reduction sequences.

But, you'd hope that in any program you could replace one by the other without affecting the result....

$$\int_0^{2+2} e^{\sin(x)} dx = \int_0^4 e^{\sin(x)} dx$$

How about $(l := 0; 4) \stackrel{?}{\simeq} (l := 1; 3 + !l)$

They will produce the same result (in any store), but you *cannot* replace one by the other in an arbitrary program context. For example:

$C[\_] = \_ + !l$

$$\begin{aligned} C[l := 0; 4] = \quad & (l := 0; 4) + !l \\ & \not\simeq \\ C[l := 1; 3 + !l] = \quad & (l := 1; 3 + !l) + !l \end{aligned}$$

On the other hand, consider

$(l :=!l + 1); (l :=!l - 1) \stackrel{?}{\simeq} (l :=!l)$

Those were all particular expressions – may want to know that some *general laws* are valid for all $e_1, e_2, \ldots$. How about these:

$$e_1; (e_2; e_3) \stackrel{?}{\simeq} (e_1; e_2); e_3$$

$$(\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3); e \stackrel{?}{\simeq} \textbf{if}\ e_1\ \textbf{then}\ e_2; e\ \textbf{else}\ e_3; e$$

$$e; (\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3) \stackrel{?}{\simeq} \textbf{if}\ e_1\ \textbf{then}\ e; e_2\ \textbf{else}\ e; e_3$$

$$e; (\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3) \stackrel{?}{\simeq} \textbf{if}\ e; e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3$$

$$\textbf{let val } \mathrm{x} = \text{ ref } 0 \textbf{ in fn } \mathrm{y}\text{:int} \Rightarrow (\mathrm{x} := !\mathrm{x} + \mathrm{y}); !\mathrm{x}$$

$$\underline{\overset{?}{\simeq}}$$

$$\textbf{let val } \mathrm{x} = \text{ ref } 0 \textbf{ in fn } \mathrm{y}\text{:int} \Rightarrow (\mathrm{x} := !\mathrm{x} - \mathrm{y}); (0 - !\mathrm{x})$$

Temporarily extend L3 with pointer equality

$$op \ ::= \ldots \mid =$$

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

$$\text{(op =)} \quad \langle \ell = \ell', s \rangle \longrightarrow \langle b, s \rangle \quad \text{if } b = (\ell = \ell')$$

$$
\begin{array}{l}
f = \quad \textbf{let val } \mathrm{x} = \text{ ref } 0 \textbf{ in} \\
\qquad \textbf{let val } \mathrm{y} = \text{ ref } 0 \textbf{ in} \\
\qquad \textbf{fn } \mathrm{z{:}int ref} \Rightarrow \textbf{if } \mathrm{z} = \mathrm{x} \textbf{ then } \mathrm{y} \textbf{ else } \mathrm{x} \\
\qquad \textbf{end end}
\end{array}
$$

$$
\begin{array}{l}
g = \quad \textbf{let val } \mathrm{x} = \text{ ref } 0 \textbf{ in} \\
\qquad \textbf{let val } \mathrm{y} = \text{ ref } 0 \textbf{ in} \\
\qquad \textbf{fn } \mathrm{z{:}int ref} \Rightarrow \textbf{if } \mathrm{z} = \mathrm{y} \textbf{ then } \mathrm{y} \textbf{ else } \mathrm{x} \\
\qquad \textbf{end end}
\end{array}
$$

$$f \stackrel{?}{\simeq} g$$

$$
\begin{array}{ll}
f = & \textbf{let val } \mathrm{x} = \text{ ref } 0 \textbf{ in} \\
& \textbf{let val } \mathrm{y} = \text{ ref } 0 \textbf{ in} \\
& \textbf{fn } \mathrm{z{:}int ref} \Rightarrow \textbf{if } \mathrm{z} = \mathrm{x} \textbf{ then } \mathrm{y} \textbf{ else } \mathrm{x} \\
g = & \textbf{let val } \mathrm{x} = \text{ ref } 0 \textbf{ in} \\
& \textbf{let val } \mathrm{y} = \text{ ref } 0 \textbf{ in} \\
& \textbf{fn } \mathrm{z{:}int ref} \Rightarrow \textbf{if } \mathrm{z} = \mathrm{y} \textbf{ then } \mathrm{y} \textbf{ else } \mathrm{x}
\end{array}
$$

$$f \stackrel{?}{\simeq} g \text{ ... no:}$$

Consider $C = t\ \_$, where

$$t = \textbf{fn } \mathrm{h{:}(int ref} \rightarrow \mathrm{int ref}) \Rightarrow$$

$$\textbf{let val } \mathrm{z} = \text{ ref } 0 \textbf{ in } \mathrm{h}\,(\mathrm{h}\,\mathrm{z}) = \mathrm{h}\,\mathrm{z}$$

$$\langle t\ f, \{\}\rangle \longrightarrow^* \langle \textbf{false}, \ldots\rangle$$

$$\langle t\ g, \{\}\rangle \longrightarrow^* \langle \textbf{true}, \ldots\rangle$$

With a ‘good’ notion of semantic equivalence, we might:

1. understand what a program *is*
2. prove that some particular expression (say an efficient algorithm) is equivalent to another (say a clear specification)
3. prove the soundness of general laws for equational reasoning about programs
4. prove some compiler optimizations are sound (source/IL/TAL)
5. understand the differences between languages

What does it mean for $\simeq$ to be 'good'?

1. programs that result in observably-different values (in some initial store) must not be equivalent

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

2. programs that terminate must not be equivalent to programs that don't

3. $\simeq$ must be an equivalence relation

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

4. $\simeq$ must be a congruence

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

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

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

(a) $\langle e_1, s \rangle \longrightarrow^\omega$ and $\langle e_2, s \rangle \longrightarrow^\omega$, or

(b) for some v, s' we have $\langle e_1, s \rangle \longrightarrow^* \langle v, s' \rangle$ and $\langle e_2, s \rangle \longrightarrow^* \langle v, s' \rangle$.

If $T = \mathsf{unit}$ then $C = \_; !l$.

If $T = \mathsf{bool}$ then $C = \textbf{if } \_ \textbf{ then } !l \textbf{ else } !l$.

If $T = \mathsf{int}$ then $C = l_1 := \_; !l$.

Congruence for Typed L1

The L1 contexts are:

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

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

Theorem 26 (Congruence for L1) $\simeq_{\Gamma}^{T}$ *has the congruence property.*

Proof Outline By case analysis, looking at each L1 context C in turn.

For each C (and for arbitrary e and s), consider the possible reduction sequences

$$\langle C[e], s\rangle \longrightarrow \langle e_1, s_1\rangle \longrightarrow \langle e_2, s_2\rangle \longrightarrow \ldots$$

For each such reduction sequence, deduce what behaviour of e was involved

$$\langle e, s\rangle \longrightarrow \langle \hat{e}_1, \hat{s}_1\rangle \longrightarrow \ldots$$

Using $e \simeq_{\Gamma}^{T} e'$ find a similar reduction sequence of e'.

Using the reduction rules construct a sequence of $C[e']$.

Theorem 26 (Congruence for L1) $\simeq_\Gamma^T$ *has the congruence property.*

By case analysis, looking at each L1 context in turn.

Case $C = (\ell := \_)$. Suppose $e \simeq_\Gamma^T e'$, $\Gamma \vdash \ell := e : T'$ and $\Gamma \vdash \ell := e' : T'$. By examining the typing rules $T = \mathsf{int}$ and $T' = \mathsf{unit}$.

To show $(\ell := e) \simeq_\Gamma^{T'} (\ell := e')$ we have to show for all s such that $\mathrm{dom}(\Gamma) \subseteq \mathrm{dom}(s)$, then $\Gamma \vdash \ell := e : T'$ ($\surd$), $\Gamma \vdash \ell := e' : T'$ ($\surd$), and either

1. $\langle \ell := e, s \rangle \longrightarrow^\omega$ and $\langle \ell := e', s \rangle \longrightarrow^\omega$, or
2. for some v, s' we have $\langle \ell := e, s \rangle \longrightarrow^* \langle v, s' \rangle$ and $\langle \ell := e', s \rangle \longrightarrow^* \langle v, s' \rangle$.

Consider the possible reduction sequences of a state $\langle \ell := e, s \rangle$. Either:

Case: $\langle \ell := e, s \rangle \longrightarrow^{\omega}$, i.e.

$\langle \ell := e, s \rangle \longrightarrow \langle e_1, s_1 \rangle \longrightarrow \langle e_2, s_2 \rangle \longrightarrow \ldots$

hence all these must be instances of (assign2), with

$\langle e, s \rangle \longrightarrow \langle \hat{e}_1, s_1 \rangle \longrightarrow \langle \hat{e}_2, s_2 \rangle \longrightarrow \ldots$

and $e_1 = (\ell := \hat{e}_1)$, $e_2 = (\ell := \hat{e}_2)$,...

Case: $\neg(\langle \ell := e, s \rangle \longrightarrow^{\omega})$, i.e.

$\langle \ell := e, s \rangle \longrightarrow \langle e_1, s_1 \rangle \longrightarrow \langle e_2, s_2 \rangle \ldots \longrightarrow \langle e_k, s_k \rangle \not\longrightarrow$

hence all these must be instances of (assign2) except the last, which must be an instance of (assign1), with

$\langle e, s \rangle \longrightarrow \langle \hat{e}_1, s_1 \rangle \longrightarrow \langle \hat{e}_2, s_2 \rangle \longrightarrow \ldots \longrightarrow \langle \hat{e}_{k-1}, s_{k-1} \rangle$

and $e_1 = (\ell := \hat{e}_1)$, $e_2 = (\ell := \hat{e}_2)$,..., $e_{k-1} = (\ell := \hat{e}_{k-1})$ and for some n we have $\hat{e}_{k-1} = n$, $e_k =$ **skip**, and $s_k = s_{k-1} + \{\ell \mapsto n\}$.

Now, if $\langle \ell := e, s \rangle \longrightarrow^{\omega}$ we have $\langle e, s \rangle \longrightarrow^{\omega}$, so by $e \simeq^{T}_{\Gamma} e'$ we have $\langle e', s \rangle \longrightarrow^{\omega}$, so (using (assign2)) we have $\langle \ell := e', s \rangle \longrightarrow^{\omega}$.

On the other hand, if $\neg(\langle \ell := e, s \rangle \longrightarrow^{\omega})$ then by the above there is some n and s_{k-1} such that $\langle e, s \rangle \longrightarrow^{*} \langle n, s_{k-1} \rangle$ and $\langle \ell := e, s \rangle \longrightarrow \langle \textbf{skip}, s_{k-1} + \{\ell \mapsto n\} \rangle$.

By $e \simeq^{T}_{\Gamma} e'$ we have $\langle e', s \rangle \longrightarrow^{*} \langle n, s_{k-1} \rangle$.

Then using (assign1)

$\langle \ell := e', s \rangle \longrightarrow^{*} \langle \ell := n, s_{k-1} \rangle \longrightarrow \langle \textbf{skip}, s_{k-1} + \{\ell \mapsto n\} \rangle = \langle e_k, s_k \rangle$ as required.

Back to the Examples

We defined $e_1 \simeq^T_\Gamma e_2$ iff for all s such that $\mathrm{dom}(\Gamma) \subseteq \mathrm{dom}(s)$, we have $\Gamma \vdash e_1 : T$, $\Gamma \vdash e_2 : T$, and either

1. $\langle e_1, s\rangle \longrightarrow^\omega$ and $\langle e_2, s\rangle \longrightarrow^\omega$, or
2. for some v, s' we have $\langle e_1, s\rangle \longrightarrow^* \langle v, s'\rangle$ and $\langle e_2, s\rangle \longrightarrow^* \langle v, s'\rangle$.

So:

$2 + 2 \simeq^{\mathsf{int}}_\Gamma 4$ for any Γ

$(l := 0; 4) \not\simeq^{\mathsf{int}}_\Gamma (l := 1; 3 + !l)$ for any Γ

$(l :=!l + 1); (l :=!l - 1) \simeq^{\mathsf{unit}}_\Gamma (l :=!l)$ for any Γ including $l{:}\mathsf{intref}$

And the general laws?

Conjecture 27 $e_1; (e_2; e_3) \simeq^T_\Gamma (e_1; e_2); e_3$ *for any* Γ, T, e_1, e_2 *and* e_3 *such that* $\Gamma \vdash e_1 : \mathsf{unit}$, $\Gamma \vdash e_2 : \mathsf{unit}$, *and* $\Gamma \vdash e_3 : T$

Conjecture 28

$((\textbf{\textit{if}}\ e_1\ \textbf{\textit{then}}\ e_2\ \textbf{\textit{else}}\ e_3); e) \simeq^T_\Gamma (\textbf{\textit{if}}\ e_1\ \textbf{\textit{then}}\ e_2; e\ \textbf{\textit{else}}\ e_3; e)$ *for any* Γ, T, e, e_1, e_2 *and* e_3 *such that* $\Gamma \vdash e_1 : \mathsf{bool}$, $\Gamma \vdash e_2 : \mathsf{unit}$, $\Gamma \vdash e_3 : \mathsf{unit}$, *and* $\Gamma \vdash e : T$

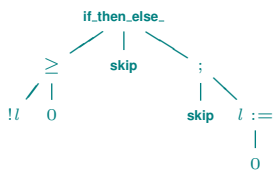
Conjecture 29

$(e; (\textbf{\textit{if}}\ e_1\ \textbf{\textit{then}}\ e_2\ \textbf{\textit{else}}\ e_3)) \simeq^T_\Gamma (\textbf{\textit{if}}\ e_1\ \textbf{\textit{then}}\ e; e_2\ \textbf{\textit{else}}\ e; e_3)$ *for any* Γ, T, e, e_1, e_2 *and* e_3 *such that* $\Gamma \vdash e : \mathsf{unit}$, $\Gamma \vdash e_1 : \mathsf{bool}$, $\Gamma \vdash e_2 : T$, *and* $\Gamma \vdash e_3 : T$

Q: Is a typed expression $\Gamma \vdash e : T$, e.g.

l:intref $\vdash$ **if** $!l \geq 0$ **then** **skip** **else** (**skip**; $l := 0$):unit:

1. a list of tokens `[ IF, DEREF, LOC "l", GTEQ, ..]`;

2. an abstract syntax tree 

;

3. the function taking store s to the reduction sequence
   $\langle e, s\rangle \longrightarrow \langle e_1, s_1\rangle \longrightarrow \langle e_2, s_2\rangle \longrightarrow \ldots$; or

4. - the equivalence class $\{e' \mid e \simeq^T_\Gamma e'\}$
   - the partial function $[\![e]\!]_\Gamma$ that takes any store s with $\mathrm{dom}(s) = \mathrm{dom}(\Gamma)$ and either is undefined, if $\langle e, s\rangle \longrightarrow^\omega$, or is $\langle v, s'\rangle$, if $\langle e, s\rangle \longrightarrow^* \langle v, s'\rangle$

Suppose $\Gamma \vdash e_1 : \mathsf{unit}$ and $\Gamma \vdash e_2 : \mathsf{unit}$.

When *is* $e_1; e_2 \simeq^{\mathsf{unit}}_{\Gamma} e_2; e_1$?

Contextual equivalence for L3

Definition 30 *Consider typed L3 programs,* $\Gamma \vdash e_1 : T$ *and* $\Gamma \vdash e_2 : T$*. We say that they are* contextually equivalent *if, for every context* C *such that* $\{\} \vdash C[e_1]$:unit *and* $\{\} \vdash C[e_2]$:unit*, we have either*

(a) $\langle C[e_1], \{\}\rangle \longrightarrow^{\omega}$ *and* $\langle C[e_2], \{\}\rangle \longrightarrow^{\omega}$*, or*

(b) for some s_1 *and* s_2 *we have* $\langle C[e_1], \{\}\rangle \longrightarrow^{*} \langle \textbf{\textit{skip}}, s_1\rangle$ *and* $\langle C[e_2], \{\}\rangle \longrightarrow^{*} \langle \textbf{\textit{skip}}, s_2\rangle$*.*

Low-level semantics

Can usefully apply semantics not just to high-level languages but to

- Intermediate Languages (e.g. Java Bytecode, MS IL, C−−)
- Assembly languages (esp. for use as a compilation target)
- C-like languages (cf. Cyclone)

By making these type-safe we can make more robust systems.

(see separate handout)

Epilogue

Lecture Feedback

Please do fill in the lecture feedback form – we need to know how the course could be improved / what should stay the same.

Good language design?

Need:

- precise definition of what the language is (so can communicate among the designers)
- technical properties (determinacy, decidability of type checking, etc.)
- pragmatic properties (usability in-the-large, implementability)

What can *you* use semantics for?

1. to understand a particular language — what you can depend on as a programmer; what you must provide as a compiler writer
2. as a tool for language design:
   (a) for clean design
   (b) for expressing design choices, understanding language features and how they interact.
   (c) for proving properties of a language, eg type safety, decidability of type inference.
3. as a foundation for proving properties of particular programs

The End