

# Semantics of Programming Languages

Computer Science Tripos, Part 1B

2014

Peter Sewell  
Computer Laboratory  
University of Cambridge

Lecture Theatre 1  
Tuesday / Thursday 10am  
4 February – 11 March 2014

Time-stamp: <2014-02-03 16:47:01 pes20>

©Peter Sewell 2014

©Sam Staton 2009–2013

©Peter Sewell 2003–2009

# Contents

<b>Syllabus</b>	<b>3</b>
<b>Learning Guide</b>	<b>4</b>
<b>Summary of Notation</b>	<b>5</b>
<b>1 Introduction</b>	<b>8</b>
<b>2 A First Imperative Language</b>	<b>12</b>
2.1 Operational Semantics . . . . .	13
2.2 Typing . . . . .	21
2.3 L1: Collected definition . . . . .	28
2.4 Exercises . . . . .	30
<b>3 Induction</b>	<b>31</b>
3.1 Abstract Syntax and Structural Induction . . . . .	33
3.2 Inductive Definitions and Rule Induction . . . . .	35
3.3 Example proofs . . . . .	38
3.4 Exercises . . . . .	45
<b>4 Functions</b>	<b>46</b>
4.1 Abstract syntax up to alpha conversion, and substitution . . . . .	48
4.2 Function Behaviour . . . . .	52
4.3 Function Typing . . . . .	56
4.4 Local Definitions and Recursive Functions . . . . .	58
4.5 Implementation . . . . .	61
4.6 L2: Collected Definition . . . . .	64
4.7 Exercises . . . . .	67
<b>5 Data</b>	<b>68</b>
5.1 Products and sums . . . . .	68
5.2 Datatypes and Records . . . . .	71
5.3 Mutable Store . . . . .	73
5.4 Evaluation Contexts . . . . .	76
5.5 L3: Collected definition . . . . .	77
5.6 Exercises . . . . .	81
<b>6 Subtyping and Objects</b>	<b>82</b>
6.1 Exercises . . . . .	88
<b>7 Semantic Equivalence</b>	<b>89</b>
7.1 Contextual equivalence . . . . .	94
7.2 Exercises . . . . .	94
<b>8 Concurrency</b>	<b>95</b>
8.1 Exercises . . . . .	105
<b>9 Epilogue</b>	<b>105</b>
<b>A Interpreter and type checker for L1 (ML)</b>	<b>107</b>
<b>B Interpreter and type checker for L1 (Java)</b>	<b>111</b>
<b>C How to do Proofs</b>	<b>117</b>
C.1 How to go about it . . . . .	117
C.2 And in More Detail... . . . .	119
C.2.1 Meet the Connectives . . . . .	119
C.2.2 Equivalences . . . . .	119
C.2.3 How to Prove a Formula . . . . .	120
C.2.4 How to Use a Formula . . . . .	122
C.3 An Example . . . . .	123
C.3.1 Proving the PL . . . . .	123
C.3.2 Using the PL . . . . .	124
C.4 Sequent Calculus Rules . . . . .	124

# Syllabus

*This course is a prerequisite for Types (Part II), Denotational Semantics (Part II), and Topics in Concurrency (Part II).*

## Aims

The aim of this course is to introduce the structural, operational approach to programming language semantics. It will show how to specify the meaning of typical programming language constructs, in the context of language design, and how to reason formally about semantic properties of programs.

## Lectures

- **Introduction.** Transition systems. The idea of structural operational semantics. Transition semantics of a simple imperative language. Language design options.
- **Types.** Introduction to formal type systems. Typing for the simple imperative language. Statements of desirable properties.
- **Induction.** Review of mathematical induction. Abstract syntax trees and structural induction. Rule-based inductive definitions and proofs. Proofs of type safety properties.
- **Functions.** Call-by-name and call-by-value function application, semantics and typing. Local recursive definitions.
- **Data.** Semantics and typing for products, sums, records, references.
- **Subtyping.** Record subtyping and simple object encoding.
- **Semantic equivalence.** Semantic equivalence of phrases in a simple imperative language, including the congruence property. Examples of equivalence and non-equivalence.
- **Concurrency.** Shared variable interleaving. Semantics for simple mutexes; a serializability property.

## Objectives

At the end of the course students should

- be familiar with rule-based presentations of the operational semantics and type systems for some simple imperative, functional and interactive program constructs
- be able to prove properties of an operational semantics using various forms of induction (mathematical, structural, and rule-based)
- be familiar with some operationally-based notions of semantic equivalence of program phrases and their basic properties

## Recommended reading

Hennessy, M. (1990). *The semantics of programming languages*. Wiley.

Out of print, but available on the web at

<http://www.scss.tcd.ie/Matthew.Hennessy/slexternal/reading.php>.

\* Pierce, B.C. (2002). *Types and programming languages*. MIT Press. ebook available at

<http://search.lib.cam.ac.uk/?itemid=|eresources|4228596>.

Winskel, G. (1993). *The formal semantics of programming languages*. MIT Press.

## Learning Guide

The books are all available in the Computer Laboratory Library. **Books:**

- Hennessy, M. (1990). *The Semantics of Programming Languages*. Wiley. Out of print.  
Introduces many of the key topics of the course. There's a copy on the web at <http://www.scss.tcd.ie/Matthew.Hennessy/slexternal/reading.php>.
- Pierce, B. C. (2002) *Types and Programming Languages*. MIT Press.  
This is a graduate-level text, covering a great deal of material on programming language semantics. The first half (through to Chapter 15) is relevant to this course, and some of the later material relevant to the Part II Types course. ebook available at <http://search.lib.cam.ac.uk/?itemid=|eresources|1472>
- Winskel, G. (1993). *The Formal Semantics of Programming Languages*. MIT Press.  
An introduction to both operational and denotational semantics.
- Harper, R. W (2012). *Practical Foundations for Programming Languages*. MIT Press.  
Also available from [www.cs.cmu.edu/~rwh/plbook/book.pdf](http://www.cs.cmu.edu/~rwh/plbook/book.pdf).

### Further reading:

- Plotkin, G. D.(1981). A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University.  
These notes first popularized the 'structural' approach to operational semantics. Although somewhat dated, they are still a mine of interesting examples. It is available at [http://homepages.inf.ed.ac.uk/gdp/publications/sos\\_jlap.pdf](http://homepages.inf.ed.ac.uk/gdp/publications/sos_jlap.pdf).
- Two essays in: Wand, I. and R. Milner (Eds) (1996), *Computing Tomorrow*, CUP:
  - Hoare, C. A. R.. *Algebra and Models*.
  - Milner, R. *Semantic Ideas in Computing*.Two accessible essays giving somewhat different perspectives on the semantics of computation and programming languages.
- Andrew Pitts lectured this course until 2002. The syllabus has changed, but you might enjoy his notes, still available at <http://www.cl.cam.ac.uk/teaching/2001/Semantics/>.
- Pierce, B. C. (ed) (2005) *Advanced Topics in Types and Programming Languages*. MIT Press.  
This is a collection of articles by experts on a range of programming-language semantics topics. Most of the details are beyond the scope of this course, but it gives a good overview of the state of the art. The contents are listed at <http://www.cis.upenn.edu/~bcpierce/attapl/>.

**Implementations:** Implementations of some of the languages are available on the course web page, accessible via <http://www.cl.cam.ac.uk/teaching/current>.

They are written in Moscow ML. This is installed on the Intel Lab machines. If you want to work with them on your own machine instead, there are Linux, Windows, and Mac versions of Moscow ML available at <http://www.itu.dk/~sestoft/mosml.html>.

**Exercises:** The notes contain various exercises, some related to the implementations. Those marked ★ should be straightforward checks that you are grasping the material; I suggest you attempt all of these. Exercises marked ★★ may need a little more thought – both proofs and some implementation-related; you should do most of them. Exercises marked ★★★ may need material beyond the notes, and/or be quite time-consuming. Below is a possible selection of exercises for supervisions.

1. §2.4: 1, 3, 4, 8, 9, 10, 11 (all these should be pretty quick); §3.4: 12, 15.
2. §4.7: 18, 19, 20, 21, 22; §5.6: 28; 2003.5.11.

3. §7.2: 37, §8.1: (39), 40; §6.1: 31, 32, 35; 2003.6.12, mock tripos from www.

**Tripos questions:** This version of the course was first given in 2002–2003. The questions since then are directly relevant, and there is an additional mock question on the course web page. The previous version of the course (by Andrew Pitts) used a slightly different form of operational semantics, ‘big-step’ instead of ‘small-step’ (see Page 63 of these notes), and different example languages, so the notation in most earlier questions may seem unfamiliar at first sight.

These questions use only small-step and should be accessible: 1998 Paper 6 Question 12, 1997 Paper 5 Question 12, and 1996 Paper 5 Question 12.

These questions use big-step, but apart from that should be ok: 2002 Paper 5 Question 9, 2002 Paper 6 Question 9, 2001 Paper 5 Question 9, 2000 Paper 5 Question 9, 1999 Paper 6 Question 9 (first two parts only), 1999 Paper 5 Question 9, 1998 Paper 5 Question 12, 1995 Paper 6 Question 12, 1994 Paper 7 Question 13, 1993 Paper 7 Question 10.

These questions depend on material which is no longer in this course (complete partial orders, continuations, or bisimulation – see the Part II Denotational Semantics and Topics in Concurrency courses): 2001 Paper 6 Question 9, 2000 Paper 6 Question 9, 1997 Paper 6 Question 12, 1996 Paper 6 Question 12, 1995 Paper 5 Question 12, 1994 Paper 8 Question 12, 1994 Paper 9 Question 12, 1993 Paper 8 Question 10, 1993 Paper 9 Question 10.

**Feedback:** Please do complete the on-line feedback form at the end of the course, and let me know during it if you discover errors in the notes or if the pace is too fast or slow. A list of corrections will be on the course web page.

**Acknowledgements (P. Sewell):** These notes are a modification of the notes that Sam Staton used for the course, 2010–2013.

**Acknowledgements (S. Staton):** These notes are a modification of the notes that Peter Sewell used for the course, 2003–2009.

**Acknowledgements (P. Sewell):** These notes draw, with thanks, on earlier courses by Andrew Pitts, on Benjamin Pierce’s book, and many other sources. Any errors are, of course, newly introduced by me.

## Summary of Notation

Each section is roughly in the order that notation is introduced. The grammars of the languages are not included here, but are in the Collected Definitions of L1, L2 and L3 later in this document.

## Logic and Set Theory

$\Phi \wedge \Phi'$	and
$\Phi \vee \Phi'$	or
$\Phi \Rightarrow \Phi'$	implies
$\neg \Phi$	not
$\forall x. \Phi(x)$	for all
$\exists x. \Phi(x)$	exists
$a \in A$	element of
$\{a_1, \dots, a_n\}$	the set with elements $a_1, \dots, a_n$
$A_1 \cup A_2$	union
$A_1 \cap A_2$	intersection
$A_1 \subseteq A_2$	subset or equal
$A_1 * A_2$	cartesian product (set of pairs)

## Finite partial functions

$\{a_1 \mapsto b_1, \dots, a_n \mapsto b_n\}$	finite partial function mapping each $a_i$ to $b_i$
$\text{dom}(s)$	set of elements in the domain of $s$
$f + \{a \mapsto b\}$	the finite partial function $f$ extended or overridden with $a$ maps to $b$
$\Gamma, x:T$	the finite partial function $\Gamma$ extended with $\{x \mapsto T\}$ – only used where $x$ not in $\text{dom}(\Gamma)$
$\Gamma, \Gamma'$	the finite partial function which is the union of $\Gamma$ and $\Gamma'$ – only used where they have disjoint domains
$\{l_1 \mapsto n_1, \dots, l_k \mapsto n_k\}$	an L1 or L2 store – the finite partial function mapping each $l_i$ to $n_i$
$\{l_1 \mapsto v_1, \dots, l_k \mapsto v_k\}$	an L3 store – the finite partial function mapping each $l_i$ to $v_i$
$l_1:\text{intref}, \dots, l_k:\text{intref}$	an L1 type environment – the finite partial function mapping each $l_i$ to $\text{intref}$
$\ell:\text{intref}, \dots, x:T, \dots$	an L2 type environment
$\ell:T_{loc}, \dots, x:T, \dots$	an L3 type environment
$\{e_1/x_1, \dots, e_k/x_k\}$	a substitution – the finite partial function $\{x_1 \mapsto e_1, \dots, x_k \mapsto e_k\}$ mapping $x_1$ to $e_1$ etc.

## Relations and auxiliary functions

$\langle e, s \rangle \longrightarrow \langle e', s' \rangle$	reduction (or transition) step
$\langle e, s \rangle \longrightarrow^* \langle e', s' \rangle$	reflexive transitive closure of $\longrightarrow$
$\langle e, s \rangle \longrightarrow^k \langle e', s' \rangle$	the $k$ -fold composition of $\longrightarrow$
$\langle e, s \rangle \longrightarrow^\omega$	has an infinite reduction sequence (a unary predicate)
$\langle e, s \rangle \not\longrightarrow$	cannot reduce (a unary predicate)
$\Gamma \vdash e:T$	in type environment $\Gamma$ , expression $e$ has type $T$
$\text{value}(e)$	$e$ is a value
$\text{fv}(e)$	the set of free variables of $e$
$\{e/x\}e'$	the expression resulting from substituting $e$ for $x$ in $e'$
$\sigma e$	the expression resulting from applying the substituting $\sigma$ to $e$
$\langle e, s \rangle \Downarrow \langle v, s' \rangle$	big-step evaluation
$\Gamma \vdash s$	store $s$ is well-typed with respect to type environment $\Gamma$
$T <: T'$	type $T$ is a subtype of type $T'$
$e \simeq e'$	semantic equivalence (informal)
$e \simeq_\Gamma^T e'$	semantic equivalence at type $T$ with respect to type environment $\Gamma$
$e \xrightarrow{a} e'$	single thread transition step, labelled with action $a$

### Particular sets

$\mathbb{B} = \{\mathbf{true}, \mathbf{false}\}$	the set of booleans
$\mathbb{L} = \{l, l_1, l_2, \dots\}$	the set of locations
$\mathbb{Z} = \{\dots, -1, 0, 1, \dots\}$	the set of integers
$\mathbb{N} = \{0, 1, \dots\}$	the set of natural numbers
$\mathbb{X} = \{x, y, \dots\}$	the set of L2 and L3 variables
$\mathbb{LAB} = \{p, q, \dots\}$	the set of record labels
$\mathbb{M} = \{m, m_0, m_1, \dots\}$	the set of mutex names
$\mathbb{T}$	the set of all types (in whichever language)
$\mathbb{T}_{loc}$	the set of all location types (in whichever language)
$L_1$	the set of all L1 expressions
TypeEnv	the set of all L1 type environments, finite partial functions from $\mathbb{L}$ to $\mathbb{Z}$
TypeEnv2	the set of all L2 type environments, the finite partial functions from $\mathbb{L} \cup \mathbb{X}$ to $\mathbb{T}_{loc} \cup \mathbb{T}$ such that $\forall \ell \in \text{dom}(\Gamma). \Gamma(\ell) \in \mathbb{T}_{loc}$ and $\forall x \in \text{dom}(\Gamma). \Gamma(x) \in \mathbb{T}$

### Metavariables

$b \in \mathbb{B}$	boolean
$n \in \mathbb{Z}$	integer
$\ell \in \mathbb{L}$	location
$op$	binary operation
$e, f$	expression (of whichever language)
$v$	value (of whichever language)
$s$	store (of whichever language)
$T \in \mathbb{T}$	type (of whichever language)
$T_{loc} \in \mathbb{T}_{loc}$	location type (of whichever language)
$\Gamma$	type environment (also, set of propositional assumptions)
$i, k, y$	natural numbers
$c$	configuration (or state), typically $\langle e, s \rangle$ with expression $e$ and store $s$
$\Phi$	formula
$c$	tree constructor
$R$	set of rules
$(H, c)$	a rule with hypotheses $H \subseteq A$ and conclusion $c \in A$ for some set $A$
$S_R$	a subset inductively defined by the set of rules $R$
$x \in \mathbb{X}$	variable
$\sigma$	substitution
$lab \in \mathbb{LAB}$	record label
$E$	evaluation context
$C$	arbitrary context
$\pi$	permutation of natural numbers
$m \in \mathbb{M}$	mutex name
$M$	state of all mutexes (a function $M: \mathbb{M} \rightarrow \mathbb{B}$ )
$a$	thread action

### Other

$\_$	hole in a context
$C[e]$	context $C$ with $e$ replacing the hole $\_$

# 1 Introduction

Slide 1

**Semantics of Programming Languages**

**Peter Sewell**

**1B, 12 lectures**

**2014**

In this course we will take a close look at programming languages. We will focus on how to define precisely what a programming language *is* – i.e., how the programs of the language behave, or, more generally, what their meaning, or *semantics*, is.

Slide 2

**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.

Many programming languages that you meet are described only in *natural language*, e.g. the English standards documents for C, Java, XML, etc. These are reasonably accessible (though often written in ‘standardsese’), but there are some major problems. It is very hard, if not impossible, to write really precise definitions in informal prose. The standards often end up being ambiguous or incomplete, or just too large and hard to understand. That leads to differing implementations and flaky systems, as the language implementors and users do not have a common understanding of what it is. More fundamentally, natural language standards obscure the real structure of languages – it’s all too easy to add a feature and a quick paragraph of text without thinking about how it interacts with the rest of the language.

Instead, as we shall see in this course, one can develop *mathematical* definitions of how programs behave, using logic and set theory (e.g. the definition of Standard ML, the .NET CLR, recent work on XQuery, etc.). These require a little more background to understand and use, but for many purposes they are a much better tool than informal standards.

Slide 3

**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 expressing design choices, understanding language features and how they interact.
  - (b) for proving properties of a language, eg type safety, decidability of type inference.
3. as a foundation for proving properties of particular programs



Semantics complements the study of language implementation (cf. *Compiler Construction* and *Optimising Compilers*). We need languages to be *both* clearly understandable, with precise definitions, *and* have good implementations.

This is true not just for the major programming languages, but also for intermediate languages (JVM, CLR), and the many, many scripting and command languages, that have often been invented on-the-fly without sufficient thought.

More broadly, while in this course we will look mostly at semantics for conventional programming languages, similar techniques can be used for hardware description languages, verification of distributed algorithms, security protocols, and so on – all manner of subtle systems for which relying on informal intuition alone leads to error. Some of these are explored in *Specification and Verification* and *Topics in Concurrency*.

Slide 4

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

Slide 5

```
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());  
        }  
    }  
}
```

Slide 6

```
JavaScript  
  
function bar(x) { return function() { var x =  
5; return x; }; }  
var f = bar(200);  
f()
```

Slide 7

```
JavaScript  
  
function bar(x) { return function() { var x =  
x; return x; }; }  
var f = bar(200);  
f()
```

Various different approaches have been used for expressing semantics.

### Styles of Semantic Definitions

Slide 8

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

Operational: define the meaning of a program in terms of the computation steps it takes in an idealized execution. Some definitions use *structural operational semantics*, in which the intermediate states are described using the language itself; others use *abstract machines*, which use more ad-hoc mathematical constructions.

Denotational: define the meaning of a program as elements of some abstract mathematical structure, e.g. regarding programming-language functions as certain mathematical functions. cf. the Denotational Semantics course.

Axiomatic or Logical: define the meaning of a program indirectly, by giving the axioms of a logic of program properties. cf. Specification and Verification.

### 'Toy' languages

Slide 9

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?

Slide 10

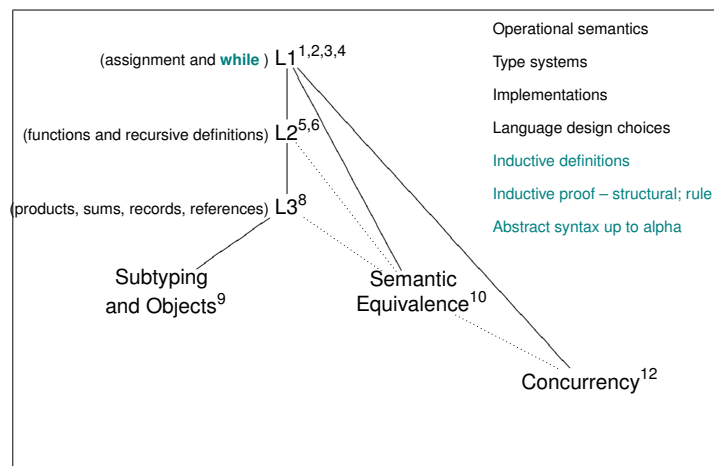
Core

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

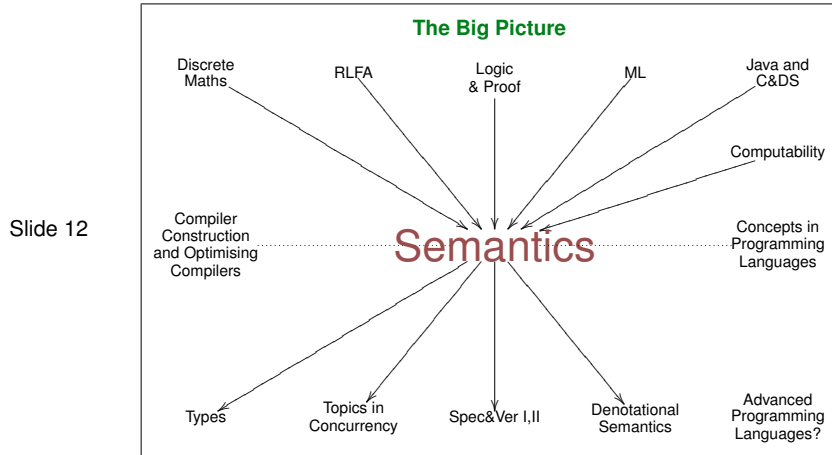
More advanced topics

- Subtyping and Objects
- Semantic Equivalence
- Concurrency

Slide 11



In the core we will develop enough techniques to deal with the semantics of a non-trivial small language, showing some language-design pitfalls and alternatives along the way. It will end up with the semantics of a decent fragment of ML. The second part will cover a selection of more advanced topics.



- Slide 13
- 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)
  - Not all previous Tripos questions are relevant (see the notes)
  - Exercises in the notes.
  - Implementations on web.
  - Books (Hennessy, Pierce, Winskel)

## 2 A First Imperative Language

Slide 14

**L1**

Slide 15

### L1 – Example

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

```
 $l_2 := 0;$   
while  $!l_1 \geq 1$  do (  
   $l_2 := !l_2 + !l_1;$   
   $l_1 := !l_1 + -1$ )
```

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

Slide 16

### L1 – Syntax

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

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

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

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

Expressions

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

Write  $L_1$  for the set of all expressions.

Points to note:

- we'll return later to *exactly* what the set  $L_1$  is when we talk about abstract syntax
- unbounded integers
- abstract locations – can't do pointer arithmetic on them
- untyped, so have nonsensical expressions like  $3 + \mathbf{true}$
- what kind of grammar is that (c.f. RLFA)?
- don't have expression/command distinction
- doesn't much matter what basic operators we have
- carefully distinguish metavariables  $b, n, \ell, op, e$  etc. from program locations  $l$  etc..

## 2.1 Operational Semantics

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

**Transition systems**

A *transition system* consists of

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

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  $c'$ ’.

Slide 17

To compare with the automata you saw in *Regular Languages and Finite Automata*: a transition system is like an NFA<sup>ε</sup> with an empty alphabet (so only ε transitions) except (a) it can have infinitely many states, and (b) we don’t specify a start state or accepting states. Sometimes one adds labels (e.g. to represent IO) but mostly we’ll just look at the values of terminated states, those that cannot do any transitions.

### Notation.

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

The particular transition systems we use for L1 are as follows.

**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

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

Slide 18

**Definition.** A *finite partial function*  $f$  from a set  $A$  to a set  $B$  is a set containing a finite number  $n \geq 0$  of pairs  $\{(a_1, b_1), \dots, (a_n, b_n)\}$ , often written  $\{a_1 \mapsto b_1, \dots, a_n \mapsto b_n\}$ , for which

- $\forall i \in \{1, \dots, n\}. a_i \in A$  (the domain is a subset of  $A$ )
- $\forall i \in \{1, \dots, n\}. b_i \in B$  (the range is a subset of  $B$ )
- $\forall i \in \{1, \dots, n\}, j \in \{1, \dots, n\}. i \neq j \Rightarrow a_i \neq a_j$  ( $f$  is functional, i.e. each element of  $A$  is mapped to at most one element of  $B$ )

For a partial function  $f$ , we write  $\text{dom}(f)$  for the set of elements in the domain of  $f$  (things that  $f$  maps to something) and  $\text{ran}(f)$  for the set of elements in the range of  $f$  (things that something is mapped to by  $f$ ). For example, for the store  $s$  above we have  $\text{dom}(s) = \{l_1, l_3\}$  and  $\text{ran}(s) = \{7, 23\}$ . Note that a finite partial function can be *empty*, just  $\{\}$ .

We write `store` for the set of all stores.

Slide 19

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

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

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

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

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

We could define the values in a different, but equivalent, style: Say *values*  $v$  are expressions from the grammar  $v ::= b \mid n \mid \mathbf{skip}$ .

Now define the behaviour for each construct of L1 by giving some rules that (together) define a transition relation  $\longrightarrow$ .

Slide 20

#### 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) 
$$\frac{\langle e_1, s \rangle \longrightarrow \langle e'_1, s' \rangle}{\langle e_1 \text{ op } e_2, s \rangle \longrightarrow \langle e'_1 \text{ op } e_2, s' \rangle}$$

(op2) 
$$\frac{\langle e_2, s \rangle \longrightarrow \langle e'_2, s' \rangle}{\langle v \text{ op } e_2, s \rangle \longrightarrow \langle v \text{ op } e'_2, s' \rangle}$$

How to read these? The rule (op +) says that for any instantiation of the metavariables  $n$ ,  $n_1$  and  $n_2$  (i.e. any choice of three integers), that satisfies the sidecondition, there is a transition from the instantiated configuration on the left to the one on the right.

We use a strict naming convention for metavariables:  $n$  can *only* be instantiated by integers, not by arbitrary expressions.

The rule (op1) says that for any instantiation of  $e_1$ ,  $e'_1$ ,  $e_2$ ,  $s$ ,  $s'$  (i.e. any three expressions and two stores), *if* a transition of the form above the line can be deduced *then* we can deduce the transition below the line.

Observe that – as you would expect – none of these first rules introduce changes in the store part of configurations.

### 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)

Slide 21

$$\begin{array}{l}
 \text{(op1)} \quad \frac{\text{(op +)} \quad \frac{}{\langle 2 + 3, \emptyset \rangle \longrightarrow \langle 5, \emptyset \rangle}}{\langle (2 + 3) + (6 + 7), \emptyset \rangle \longrightarrow \langle 5 + (6 + 7), \emptyset \rangle} \\
 \\
 \text{(op2)} \quad \frac{\text{(op +)} \quad \frac{}{\langle 6 + 7, \emptyset \rangle \longrightarrow \langle 13, \emptyset \rangle}}{\langle 5 + (6 + 7), \emptyset \rangle \longrightarrow \langle 5 + 13, \emptyset \rangle} \\
 \\
 \text{(op +)} \quad \frac{}{\langle 5 + 13, \emptyset \rangle \longrightarrow \langle 18, \emptyset \rangle}
 \end{array}$$

First transition: using (op1) with  $e_1 = 2 + 3$ ,  $e'_1 = 5$ ,  $e_2 = 6 + 7$ ,  $op = +$ ,  $s = \emptyset$ ,  $s' = \emptyset$ , and using (op +) with  $n_1 = 2$ ,  $n_2 = 3$ ,  $s = \emptyset$ . Note couldn't begin with (op2) as  $e_1 = 2 + 3$  is not a value, and couldn't use (op +) directly on  $(2 + 3) + (6 + 7)$  as  $2 + 3$  and  $6 + 7$  are not numbers from  $\mathbb{Z}$  – just expressions which might eventually evaluate to numbers (recall, by convention the  $n$  in the rules ranges over  $\mathbb{Z}$  only).

Second transition: using (op2) with  $e_1 = 5$ ,  $e_2 = 6 + 7$ ,  $e'_2 = 13$ ,  $op = +$ ,  $s = \emptyset$ ,  $s' = \emptyset$ , and using (op +) with  $n_1 = 6$ ,  $n_2 = 7$ ,  $s = \emptyset$ . Note that to use (op2) we needed that  $e_1 = 5$  is a value. We couldn't use (op1) as  $e_1 = 5$  does not have any transitions itself.

Third transition: using (op +) with  $n_1 = 5$ ,  $n_2 = 13$ ,  $s = \emptyset$ .

To find each transition we do something like *proof search* in natural deduction: starting with a state (at the bottom left), look for a rule and an instantiation of the metavariables in that rule that makes the left-hand-side of its conclusion match that state. Beware that in general there might be more than one rule and one instantiation that does this. If there isn't a derivation concluding in  $\langle e, s \rangle \longrightarrow \langle e', s' \rangle$  then there isn't such a transition.

### L1 Semantics (3 of 4) – store and sequencing

Slide 22

$$\begin{array}{l}
 \text{(deref)} \quad \langle !\ell, s \rangle \longrightarrow \langle n, s \rangle \quad \text{if } \ell \in \text{dom}(s) \text{ and } s(\ell) = n \\
 \\
 \text{(assign1)} \quad \langle \ell := n, s \rangle \longrightarrow \langle \text{skip}, s + \{\ell \mapsto n\} \rangle \quad \text{if } \ell \in \text{dom}(s) \\
 \\
 \text{(assign2)} \quad \frac{\langle e, s \rangle \longrightarrow \langle e', s' \rangle}{\langle \ell := e, s \rangle \longrightarrow \langle \ell := e', s' \rangle} \\
 \\
 \text{(seq1)} \quad \langle \text{skip}; e_2, s \rangle \longrightarrow \langle e_2, s \rangle \\
 \\
 \text{(seq2)} \quad \frac{\langle e_1, s \rangle \longrightarrow \langle e'_1, s' \rangle}{\langle e_1; e_2, s \rangle \longrightarrow \langle e'_1; e_2, s' \rangle}
 \end{array}$$

**Example**

Slide 23

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

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

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

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

Slide 24

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

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

(if3) 
$$\frac{\langle e_1, s \rangle \longrightarrow \langle e'_1, s' \rangle}{\langle \text{if } e_1 \text{ then } e_2 \text{ else } e_3, s \rangle \longrightarrow \langle \text{if } e'_1 \text{ then } e_2 \text{ else } e_3, s' \rangle}$$

(while)

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

**Example**

Slide 25

If

$$e = (b_2 := 0; \text{while } !l_1 \geq 1 \text{ do } (b_2 := !b_2 + !l_1; l_1 := !l_1 + -1))$$

$$s = \{l_1 \mapsto 3, b_2 \mapsto 0\}$$

then

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

That concludes our definition of L1. The full definition is collected on page 28.

**Determinacy**

Slide 26

**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

Note that top-level universal quantifiers are usually left out – the theorem really says “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  $\langle e_1, s_1 \rangle = \langle e_2, s_2 \rangle$ ”.



### 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.

Slide 27

### 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
```

Slide 28

We've chosen to represent locations as strings, so they pretty-print trivially. A lower-level implementation would use ML references.

In the semantics, a store is a finite partial function from locations to integers. In the implementation, we represent a store as a list of `loc*int` pairs containing, for each  $\ell$  in the domain of the store and mapped to  $n$ , exactly one element of the form  $(\ell, n)$ . The order of the list will not be important. This is not a very efficient implementation, but it is simple.

```
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
```

Slide 29

The expression and operation datatypes have essentially the same form as the abstract grammar. Note, though, that it does not exactly match the semantics, as that allowed arbitrary integers whereas here we use the bounded Moscow ML integers – so not every term of the abstract syntax is representable as an element of type `expr`, and the interpreter will fail with an overflow exception if `+` overflows.

**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`.

Slide 30

**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  $\langle e, s \rangle \not\rightarrow$ , or `SOME  $(e', s')$` , if it has a transition  $\langle e, s \rangle \rightarrow \langle e', s' \rangle$ .

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

Slide 31

(you might think it would be better ML style to use exceptions instead of these options; that would be fine).

**(op +), (op ≥)**

```
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) =>
        ...
```

Slide 32

Contrast this code with the semantic rules given earlier.

**(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 )
```

Slide 33

Note that the code depends on global properties of the semantics, including the fact that it defines a deterministic transition system, so the comments indicating that particular lines of code implement particular semantic rules are not the whole story.

Slide 34

```

(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 ) )

```

Slide 35

**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  $\langle e, s \rangle \xrightarrow{*} \langle e', s' \rangle \not\xrightarrow{*}$ , 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')

```

The full interpreter code is in Appendix A, and you can also download it from the course website, in the file `l1.ml`, together with a pretty-printer and the type-checker we will come to soon. For comparison, there is also a Java implementation in `l1.java`.

Slide 36

**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.

L1 is a simple language, but it nonetheless involves several language design choices.

Slide 37

**Language design 1. Order of evaluation**

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

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

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

(op1b) 
$$\frac{\langle e_2, s \rangle \longrightarrow \langle e'_2, s' \rangle}{\langle e_1 \text{ op } e_2, s \rangle \longrightarrow \langle e_1 \text{ op } e'_2, s' \rangle}$$

(op2b) 
$$\frac{\langle e_1, s \rangle \longrightarrow \langle e'_1, s' \rangle}{\langle e_1 \text{ op } v, s \rangle \longrightarrow \langle e'_1 \text{ op } v, s' \rangle}$$

In this language (call it L1b)

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

Left-to-right evaluation is arguably more intuitive than right-to-left.

One could also *underspecify*, taking both (op1) and (op1b) rules. That language doesn't have the Determinacy property.

Slide 38

**Language design 2. Assignment results**

Recall

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

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

So

$$\langle l := 1; l := 2, \{l \mapsto 0\} \rangle \longrightarrow \langle \text{skip}; l := 2, \{l \mapsto 1\} \rangle$$

$$\longrightarrow^* \langle \text{skip}, \{l \mapsto 2\} \rangle$$

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

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

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

Matter of taste? Another possibility: return the *old* value, e.g. in ANSI C signal handler installation.

Slide 39

**Language design 3. Store initialization**

Recall that

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

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

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

Instead, could

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

In the next section we will introduce a type system to rule out any program that could reach a stuck expression of these forms. (Would the two alternatives be a good idea?)

**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  $\ell \in \text{dom}(s)$  and  $s(\ell) = n$

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

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

Can store only integers.  $\langle l := \text{true}, s \rangle$  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  $3 + \text{false}$  as early as possible, rather than let the program get stuck or give a runtime error. We'll do so with a *type system*.

## 2.2 Typing

**L1 Typing**

Slide 44

**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.**

Type systems are also used to provide information to compiler optimizers; to enforce security properties, from simple absence of buffer overflows to sophisticated information-flow policies; and (in research languages) for many subtle properties, e.g. type systems that allow only polynomial-time computation. There are rich connections with logic, which we'll return to later.

Slide 45

**Formal type systems**

We will define a ternary relation  $\Gamma \vdash e : T$ , read as 'expression  $e$  has type  $T$ , under assumptions  $\Gamma$  on the types of locations that may occur in  $e$ '.

For example (according to the definition coming up):

$$\begin{aligned} \{\} &\vdash \text{if true then } 2 \text{ else } 3 + 4 && : \text{int} \\ l_1:\text{intref} &\vdash \text{if } !l_1 \geq 3 \text{ then } !l_1 \text{ else } 3 && : \text{int} \\ \{\} &\not\vdash 3 + \text{false} && : T \text{ for any } T \\ \{\} &\not\vdash \text{if true then } 3 \text{ else false} && : \text{int} \end{aligned}$$

Note that the last is excluded despite the fact that when you execute the program you will always get an `int` – type systems define *approximations* to the behaviour of programs, often quite crude – and this has to be so, as we generally would like them to be decidable, so that compilation is guaranteed to terminate.

Slide 46

**Types for L1**

Types of expressions:

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

Types of locations:

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

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

Let  $\Gamma$  range over `TypeEnv`, the finite partial functions from locations  $\mathbb{L}$  to  $\mathbb{T}_{loc}$ . Notation: write a  $\Gamma$  as  $l_1:\text{intref}, \dots, l_k:\text{intref}$  instead of  $\{l_1 \mapsto \text{intref}, \dots, l_k \mapsto \text{intref}\}$ .

- concretely,  $\mathbb{T} = \{\text{int}, \text{bool}, \text{unit}\}$  and  $\mathbb{T}_{loc} = \{\text{intref}\}$ .
- in this language, there is only one type in  $\mathbb{T}_{loc}$ , so a  $\Gamma$  can be thought of as just a set of locations. (Later,  $\mathbb{T}_{loc}$  will be more interesting.)

Slide 47

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

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

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

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

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

Note that in (if) the  $T$  is arbitrary, so long as both premises have the *same*  $T$ .

In some rules we arrange the premises vertically to save space, e.g.

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

but this is merely visual layout. Derivations using such a rule should be written as if it was in the horizontal form.

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

Slide 48

**Example**

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

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

where  $\nabla$  is

$$\text{(op +)} \quad \frac{\text{(int)} \quad \overline{\{\} \vdash 3:\text{int}} \quad \text{(int)} \quad \overline{\{\} \vdash 4:\text{int}}}{\{\} \vdash 3 + 4:\text{int}}$$

Slide 49

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

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

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

Here the  $\Gamma(\ell) = \text{intref}$  just means  $\ell \in \text{dom}(\Gamma)$ .

Slide 50

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

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

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

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

Note that the typing rules are *syntax-directed* – for each clause of the abstract syntax for expressions there is exactly one rule with a conclusion of that form.

Slide 51

**Properties**

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

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

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

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

(we'll discuss how to *prove* these results soon)

Semantic style: one could make an explicit definition of what configurations are runtime errors. Here, instead, those configurations are just stuck.

Slide 52

**Type checking, typeability, and type inference**

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

**Type inference problem:** given  $\Gamma$  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  $\Gamma$  (or failing, if there is none).

For this type system, though, both are easy.

Slide 53

**More Properties**

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

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

Also:

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



The file `l1.ml` contains also an implementation of a type inference algorithm for L1 – take a look.

Slide 54

**Type inference – Implementation**

First must pick representations for types and for  $\Gamma$ '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
```

In the semantics, type environments  $\Gamma$  are partial functions from locations to the singleton set `{intref}`. Here, just as we did for stores, we represent them as a list of `loc*type_loc` pairs containing, for each  $\ell$  in the domain of the type environment, exactly one element of the form `( $\ell$ , intref)`.

Slide 55

**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
      (SOME bool, SOME unit) -> SOME unit )
```

Slide 56

**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)

```

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

Slide 57

**The Type Inference Algorithm – Deref**

```

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

```

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

Again, the code depends on a uniqueness property (Theorem 7), without which we would have to have **infertype** return a **type\_L1** list of all the possible types.

Slide 58

**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  $\{l_1 \mapsto n_1, \dots, l_k \mapsto n_k\}$  and all locations that occur in  $e$  are contained in  $\{l_1, \dots, l_k\}$ .

(watch out for  $\sim 1$  and  $-1$ )

### 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."*

Slide 59

Some languages build the type system into the syntax. Original FORTRAN, BASIC etc. had typing built into variable names, with e.g. those beginning with I or J storing integers). Sometimes typing is built into the grammar, with e.g. separate grammatical classes of expressions and commands. As the type systems become more expressive, however, they quickly go beyond what can be captured in context-free grammars. They must then be separated from lexing and parsing, both conceptually and in implementations.

## 2.3 L1: Collected definition

### Syntax

Booleans  $b \in \mathbb{B} = \{\mathbf{true}, \mathbf{false}\}$   
 Integers  $n \in \mathbb{Z} = \{\dots, -1, 0, 1, \dots\}$   
 Locations  $\ell \in \mathbb{L} = \{\ell, \ell_0, \ell_1, \ell_2, \dots\}$

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

Expressions

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

### 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.

Say *stores*  $s$  are finite partial functions from  $\mathbb{L}$  to  $\mathbb{Z}$ . Say *values*  $v$  are expressions from the grammar  $v ::= b \mid n \mid \mathbf{skip}$ .

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

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

$$(\text{op1}) \quad \frac{\langle e_1, s \rangle \longrightarrow \langle e'_1, s' \rangle}{\langle e_1 \text{ op } e_2, s \rangle \longrightarrow \langle e'_1 \text{ op } e_2, s' \rangle}$$

$$(\text{op2}) \quad \frac{\langle e_2, s \rangle \longrightarrow \langle e'_2, s' \rangle}{\langle v \text{ op } e_2, s \rangle \longrightarrow \langle v \text{ op } e'_2, s' \rangle}$$

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

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

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

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

$$(\text{seq2}) \quad \frac{\langle e_1, s \rangle \longrightarrow \langle e'_1, s' \rangle}{\langle e_1; e_2, s \rangle \longrightarrow \langle e'_1; e_2, s' \rangle}$$

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

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

$$(\text{if3}) \quad \frac{\langle e_1, s \rangle \longrightarrow \langle e'_1, s' \rangle}{\langle \mathbf{if} \ e_1 \ \mathbf{then} \ e_2 \ \mathbf{else} \ e_3, s \rangle \longrightarrow \langle \mathbf{if} \ e'_1 \ \mathbf{then} \ e_2 \ \mathbf{else} \ e_3, s' \rangle}$$

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

## Typing

Types of expressions:

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

Types of locations:

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

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

Let  $\Gamma$  range over  $\text{TypeEnv}$ , the finite partial functions from locations  $\mathbb{L}$  to  $\mathbb{T}_{loc}$ .

$$\begin{aligned} & \text{(int)} \quad \Gamma \vdash n:\text{int} \quad \text{for } n \in \mathbb{Z} \\ & \text{(bool)} \quad \Gamma \vdash b:\text{bool} \quad \text{for } b \in \{\text{true}, \text{false}\} \\ & \text{(op } +) \quad \frac{\Gamma \vdash e_1:\text{int} \quad \Gamma \vdash e_2:\text{int}}{\Gamma \vdash e_1 + e_2:\text{int}} \quad \text{(op } \geq) \quad \frac{\Gamma \vdash e_1:\text{int} \quad \Gamma \vdash e_2:\text{int}}{\Gamma \vdash e_1 \geq e_2:\text{bool}} \\ & \text{(if)} \quad \frac{\Gamma \vdash e_1:\text{bool} \quad \Gamma \vdash e_2:T \quad \Gamma \vdash e_3:T}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3:T} \\ & \text{(assign)} \quad \frac{\Gamma(\ell) = \text{intref} \quad \Gamma \vdash e:\text{int}}{\Gamma \vdash \ell := e:\text{unit}} \\ & \text{(deref)} \quad \frac{\Gamma(\ell) = \text{intref}}{\Gamma \vdash !\ell:\text{int}} \\ & \text{(skip)} \quad \Gamma \vdash \text{skip}:\text{unit} \\ & \text{(seq)} \quad \frac{\Gamma \vdash e_1:\text{unit} \quad \Gamma \vdash e_2:T}{\Gamma \vdash e_1; e_2:T} \\ & \text{(while)} \quad \frac{\Gamma \vdash e_1:\text{bool} \quad \Gamma \vdash e_2:\text{unit}}{\Gamma \vdash \text{while } e_1 \text{ do } e_2:\text{unit}} \end{aligned}$$

## 2.4 Exercises

**Exercise 1** ★ Write a program to compute the factorial of the integer initially in location  $l_1$ . Take care to ensure that your program really is an expression in L1.

**Exercise 2** ★ Give full derivations of all the reduction steps of  $\langle (l_0 := 7); (l_1 := (!l_0 + 2)), \{l_0 \mapsto 0, l_1 \mapsto 0\} \rangle$

**Exercise 3** ★ Give full derivations of the first four reduction steps of the  $\langle e, s \rangle$  of the first L1 example on Slide 15.

**Exercise 4** ★ Adapt the implementation code to correspond to the two rules (op1b) and (op2b) on Slide 37. Give some test cases that distinguish between the original and the new semantics.

**Exercise 5** ★ Adapt the implementation code to correspond to the two rules (assign1') and (seq1') on Slide 38. Give some test cases that distinguish between the original and the new semantics.

**Exercise 6** ★★ Fix the L1 semantics to match the implementation, taking care with the representation of integers.

**Exercise 7** ★ Give a type derivation for  $(l_0 := 7); (l_1 := (!l_0 + 2))$  with  $\Gamma = l_0:\text{intref}, l_1:\text{intref}$ .

**Exercise 8** ★ Give a type derivation for  $e$  on Slide 25 with  $\Gamma = l_1:\text{intref}, l_2:\text{intref}, l_3:\text{intref}$ .

**Exercise 9** ★ Does Type Preservation hold for the variant language with rules (assign1') and (seq1')? on Slide 38? If not, give an example, and show how the type rules could be adjusted to make it true.

**Exercise 10** ★ Adapt the type inference implementation to match your revised type system from Exercise 9.

**Exercise 11** ★ Check whether mosml, the L1 implementation and the L1 semantics agree on the order of evaluation for operators and sequencing.

### 3 Induction

Key concepts in this chapter:

- Structural induction
- Rule induction

Slide 60

**Induction**

Slide 61

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*.

Slide 62

**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*.

Slide 63

**Principle of Mathematical Induction**

For any property  $\Phi(x)$  of natural numbers  $x \in \mathbb{N} = \{0, 1, 2, \dots\}$ , 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.  
 $(\Phi(0) \wedge (\forall x \in \mathbb{N}.\Phi(x) \Rightarrow \Phi(x + 1))) \Rightarrow \forall x \in \mathbb{N}.\Phi(x)$

(NB, the natural numbers include 0)

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

For example, to prove

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

use mathematical induction for

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

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!

Slide 64

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

I have annotated the proof to say what's going on.

**Proof** We prove  $\forall x. \Phi(x)$ , where <sup>(state  $\Phi$  explicitly)</sup>

$$\Phi(x) \stackrel{\text{def}}{=} (1 + 2 + \dots + x = 1/2 * x * (x + 1))$$

*(state the induction principle you're using)*

by mathematical induction.

*(Now show each conjunct of the premise of the induction principle)*

**Base case:** *(conjunct  $\Phi(0)$  )*

$\Phi(0)$  is <sup>(instantiate  $\Phi$ )</sup>  $(1 + \dots + 0 = 1/2 * 0 * (0 + 1))$ , which holds as both sides are equal to 0.

**Inductive step:** *(conjunct  $\forall x \in \mathbb{N}. \Phi(x) \Rightarrow \Phi(x + 1)$  )*

Consider an arbitrary  $k \in \mathbb{N}$  *(it's a universal ( $\forall$ ), so consider an arbitrary one).*

Suppose  $\Phi(k)$  *(to show the implication  $\Phi(k) \Rightarrow \Phi(k + 1)$ , assume the premise and try to show the conclusion).*

We have to show  $\Phi(k + 1)$ , i.e. *(state what we have to show explicitly)*

$$(1 + 2 + \dots + (k + 1)) = 1/2 * (k + 1) * ((k + 1) + 1)$$

Now, the left hand side is

$$\begin{aligned} (1 + 2 + \dots + (k + 1)) &= (1 + 2 + \dots + k) + (k + 1) && \text{(rearranging)} \\ &= (1/2 * k * (k + 1)) + (k + 1) && \text{(using } \Phi(k) \text{ )} \end{aligned}$$

*(say where you use the 'induction hypothesis' assumption  $\Phi(k)$  made above)*

and the right hand side is (rearranging)

$$\begin{aligned} 1/2 * (k + 1) * ((k + 1) + 1) &= 1/2 * (k * (k + 1) + (k + 1) * 1 + 1 * k + 1) \\ &= 1/2 * k * (k + 1) + 1/2 * ((k + 1) + k + 1) \\ &= 1/2 * k * (k + 1) + (k + 1) \end{aligned}$$

which is equal to the LHS.

□



### 3.1 Abstract Syntax and Structural Induction

Slide 65

**Abstract Syntax and Structural Induction**

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

**Theorem 1 (Determinacy)** *If  $\langle e, s \rangle \rightarrow \langle e_1, s_1 \rangle$  and  $\langle e, s \rangle \rightarrow \langle e_2, s_2 \rangle$  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  $\langle e, s \rangle \rightarrow \langle e_1, s_1 \rangle$  and  $\langle e, s \rangle \rightarrow \langle e_2, s_2 \rangle$  then  $\langle e_1, s_1 \rangle = \langle e_2, s_2 \rangle$ .*

Slide 66

**Abstract Syntax**

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

Recall we said:

$e ::= n \mid b \mid e \text{ op } e \mid \text{if } e \text{ then } e \text{ else } e \mid$   
 $\ell := e \mid \ell \mid$   
**skip**  $\mid e; e \mid$   
**while**  $e \text{ do } e$

defining a set of expressions.

Slide 67

Q: Is an expression, e.g. `if !l ≥ 0 then skip else (skip; l := 0)`:

- a list of characters [ 'i', 'f', ' ', '!', '≥', '0', ' ', 't', 'h', 'e', 'n', ' ', 's', 'k', 'i', 'p', ' ', 'e', 'l', 's', 'e', ' ', '(', 's', 'k', 'i', 'p', ';', 'l', ':', '=', '0', ')'];
- a list of tokens [ IF, Deref, LOC "l", GTEQ, ..]; or
- an abstract syntax tree?

Slide 68

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

$1 + 2 + 3$  – ambiguous

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

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

For semantics we don't want to be distracted by concrete syntax – it's easiest to work with abstract syntax trees, which for this grammar are finite trees, with ordered branches, labelled as follows:

- leaves (nullary nodes) labelled by  $\mathbb{B} \cup \mathbb{Z} \cup (\{!\} * \mathbb{L}) \cup \{\mathbf{skip}\} = \{\mathbf{true}, \mathbf{false}, \mathbf{skip}\} \cup \{\dots, -1, 0, 1, \dots\} \cup \{!l, !l_1, !l_2, \dots\}$ .
- unary nodes labelled by  $\{l :=, l_1 :=, l_2 :=, \dots\}$
- binary nodes labelled by  $\{+, \geq, ;, \mathbf{while\_do\_}\}$
- ternary nodes labelled by  $\{\mathbf{if\_then\_else\_}\}$

Abstract grammar *suggests* a concrete syntax – we write expressions as strings just for convenience, using parentheses to disambiguate where required and infix notation, but really mean trees.

**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  $\Phi$  holds for the subtrees  $e_1, \dots, e_k$  then  $\Phi$  holds for the tree  $c(e_1, \dots, e_k)$ . i.e.

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

where the tree constructors (or node labels)  $c$  are  $n$ , **true**, **false**, **!**, **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:  $\Phi(\mathbf{skip})$   
 $\forall b \in \{\mathbf{true}, \mathbf{false}\}. \Phi(b)$   
 $\forall n \in \mathbb{Z}. \Phi(n)$   
 $\forall \ell \in \mathbb{L}. \Phi(!\ell)$

Slide 70 unary:  $\forall \ell \in \mathbb{L}. \forall e. \Phi(e) \Rightarrow \Phi(\ell := e)$

binary:  $\forall op. \forall e_1, e_2. (\Phi(e_1) \wedge \Phi(e_2)) \Rightarrow \Phi(e_1 \text{ op } e_2)$   
 $\forall e_1, e_2. (\Phi(e_1) \wedge \Phi(e_2)) \Rightarrow \Phi(e_1; e_2)$   
 $\forall e_1, e_2. (\Phi(e_1) \wedge \Phi(e_2)) \Rightarrow \Phi(\mathbf{while } e_1 \mathbf{ do } e_2)$

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

(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.

To do that we need to verify all the premises of the principle of structural induction – the formulae in the second box below – for this  $\Phi$ .

Slide 72

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

nullary:  $\Phi(\text{skip})$   
 $\forall b \in \{\text{true}, \text{false}\}. \Phi(b)$   
 $\forall n \in \mathbb{Z}. \Phi(n)$   
 $\forall \ell \in \mathbb{L}. \Phi(!\ell)$

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

binary:  $\forall op. \forall e_1, e_2. (\Phi(e_1) \wedge \Phi(e_2)) \Rightarrow \Phi(e_1 \text{ op } e_2)$   
 $\forall e_1, e_2. (\Phi(e_1) \wedge \Phi(e_2)) \Rightarrow \Phi(e_1; e_2)$   
 $\forall e_1, e_2. (\Phi(e_1) \wedge \Phi(e_2)) \Rightarrow \Phi(\text{while } e_1 \text{ do } e_2)$

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

We will come back later to look at some of these details.

### 3.2 Inductive Definitions and Rule Induction

Slide 73

**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  $\text{dom}(\Gamma) \subseteq \text{dom}(s)$  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  $\text{dom}(\Gamma) \subseteq \text{dom}(s)$  and  $\langle e, s \rangle \longrightarrow \langle e', s' \rangle$  then  $\Gamma \vdash e': T$  and  $\text{dom}(\Gamma) \subseteq \text{dom}(s')$ .*

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

Slide 74

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

(op1) 
$$\frac{\langle e_1, s \rangle \longrightarrow \langle e'_1, s' \rangle}{\langle e_1 \text{ op } e_2, s \rangle \longrightarrow \langle e'_1 \text{ op } e_2, s' \rangle}$$

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

What did we actually mean?

Slide 75

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

For the transition relation:

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

Slide 76

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  $e_1, e_2, s, e'_1, s'$ .

$$\begin{array}{l} \text{(op+)} \frac{}{\langle 2 + 2, \{\} \rangle \longrightarrow \langle 4, \{\} \rangle}, \quad \text{(op+)} \frac{}{\langle 2 + 3, \{\} \rangle \longrightarrow \langle 5, \{\} \rangle}, \dots \\ \text{(op1)} \frac{\langle 2 + 2, \{\} \rangle \longrightarrow \langle 4, \{\} \rangle}{\langle (2 + 2) + 3, \{\} \rangle \longrightarrow \langle 4 + 3, \{\} \rangle}, \quad \text{(op1)} \frac{\langle 2 + 2, \{\} \rangle \longrightarrow \langle \text{false}, \{\} \rangle}{\langle (2 + 2) + 3, \{\} \rangle \longrightarrow \langle \text{false} + 3, \{\} \rangle} \end{array}$$

Note the last has a premise that is not itself derivable, but nonetheless this is a legitimate instance of (op1).

Slide 77

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.

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

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

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.

Now, to prove something about an inductively-defined set, we use rule induction.

Slide 78

### 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

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

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

For some proofs a slightly different principle is useful – this variant allows you to assume each of the  $h_i$  are themselves members of  $S_R$ .

Slide 79

**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

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

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

(This is just the original principle for the property  $(\Phi(a) \wedge a \in S_R)$ .)

Slide 80

**Proving Progress (Outline)**

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

**Proof Take**

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

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

Slide 81

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

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

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

Instantiating to the L1 typing rules, have to show:

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

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

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

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

etc.

Slide 82

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

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

### Which Induction Principle to Use?

Slide 83

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.

For completeness, observe the following:

Mathematical induction over  $\mathbb{N}$  is essentially the same as structural induction over  $n ::= \mathbf{zero} \mid \mathbf{succ}(n)$ .

Instead of using structural induction (for an arbitrary grammar), you could use mathematical induction on the *size* of terms.

Instead of using structural induction, you could use rule induction: supposing some fixed set of tree node labels (e.g. all the character strings), take  $A$  to be the set of all trees with those labels, and consider each clause of your grammar (e.g.  $e ::= \dots \mid e + e$ ) to be a rule

$$\frac{e \quad e}{e + e}$$

### 3.3 Example proofs

#### Example Proofs

Slide 84

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?

Slide 85

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.

Remember – the point is to use the mathematics to *help you think* about things that are too complex to keep in your head all at once: to keep track of all the cases etc. To do that, and to communicate with other people, it’s important to *write down* the reasoning and proof structure as clearly as possible. After you’ve done a proof you should give it to someone (your supervision partner first, perhaps) to see if they (a) can understand what you’ve said, and (b) if they believe it.

Slide 86

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)

**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  $\langle e_1, s_1 \rangle = \langle e_2, s_2 \rangle$ .*

**Proof** Take

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

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

**Cases skip, b, n.** For  $e$  of these forms there are no rules with a conclusion of the form  $\langle e, \dots \rangle \longrightarrow \langle \dots, \dots \rangle$  so the left hand side of the implication cannot hold, so the implication is true.

**Case !l.** Take arbitrary  $s, e', s', e'', s''$  such that  $\langle !l, s \rangle \longrightarrow \langle e', s' \rangle \wedge \langle !l, s \rangle \longrightarrow \langle e'', s'' \rangle$ .

The only rule which could be applicable is (deref), in which case, for those transitions to be instances of the rule we must have

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

so  $e' = e''$  and  $s' = s''$ .

**Case  $\ell := e$ .** Suppose  $\Phi(e)$  (then we have to show  $\Phi(\ell := e)$ ).

Take arbitrary  $s, e', s', e'', s''$  such that  $\langle \ell := e, s \rangle \longrightarrow \langle e', s' \rangle \wedge \langle \ell := e, s \rangle \longrightarrow \langle e'', s'' \rangle$ .

It's handy to have this lemma:

**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  $n, b, \text{skip}$ . 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  $n, b, \text{skip}$ .  $\square$

The only rules which could be applicable, for each of the two transitions, are (assign1) and (assign2).

**case  $\langle \ell := e, s \rangle \longrightarrow \langle e', s' \rangle$**  is an instance of (assign1). Then for some  $n$  we have  $e = n$  and  $\ell \in \text{dom}(s)$  and  $e' = \text{skip}$  and  $s' = s + \{\ell \mapsto n\}$ .

**case  $\langle \ell := n, s \rangle \longrightarrow \langle e'', s'' \rangle$**  is an instance of (assign1) (note we are using the fact that  $e = n$  here). Then  $e'' = \text{skip}$  and  $s'' = s + \{\ell \mapsto n\}$  so  $\langle e', s' \rangle = \langle e'', s'' \rangle$  as required.

**case  $\langle \ell := e, s \rangle \longrightarrow \langle e'', s'' \rangle$**  is an instance of (assign2). Then  $\langle n, s \rangle \longrightarrow \langle e'', s'' \rangle$ , which contradicts the lemma, so this case cannot arise.

**case  $\langle \ell := e, s \rangle \longrightarrow \langle e', s' \rangle$**  is an instance of (assign2). Then for some  $e'_1$  we have  $\langle e, s \rangle \longrightarrow \langle e'_1, s' \rangle$  (\*) and  $e' = (\ell := e'_1)$ .

**case  $\langle \ell := e, s \rangle \longrightarrow \langle e'', s'' \rangle$**  is an instance of (assign1). Then for some  $n$  we have  $e = n$ , which contradicts the lemma, so this case cannot arise.

**case  $\langle \ell := e, s \rangle \longrightarrow \langle e'', s'' \rangle$**  is an instance of (assign2). Then for some  $e'_1$  we have  $\langle e, s \rangle \longrightarrow \langle e'_1, s'' \rangle$  (\*\*) and  $e'' = (\ell := e'_1)$ . Now, by the induction hypothesis  $\Phi(e)$ , (\*) and (\*\*) we have  $\langle e'_1, s' \rangle = \langle e'_1, s'' \rangle$ , so  $\langle e', s' \rangle = \langle \ell := e'_1, s' \rangle = \langle \ell := e'_1, s'' \rangle = \langle e'', s'' \rangle$  as required.

**Case  $e_1 \text{ op } e_2$ .** Suppose  $\Phi(e_1)$  and  $\Phi(e_2)$ .

Take arbitrary  $s, e', s', e'', s''$  such that  $\langle e_1 \text{ op } e_2, s \rangle \longrightarrow \langle e', s' \rangle \wedge \langle e_1 \text{ op } e_2, s \rangle \longrightarrow \langle e'', s'' \rangle$ .

By examining the expressions in the left-hand-sides of the conclusions of the rules, and using the lemma above, the only possibilities are those below (you should check why this is so for yourself).

**case**  $op = +$  and  $\langle e_1 + e_2, s \rangle \longrightarrow \langle e', s' \rangle$  is an instance of (op+) and  $\langle e_1 + e_2, s \rangle \longrightarrow \langle e'', s'' \rangle$  is an instance of (op+).

Then for some  $n_1, n_2$  we have  $e_1 = n_1, e_2 = n_2, e' = n_3 = e''$  for  $n_3 = n_1 + n_2$ , and  $s' = s = s''$ .

**case**  $op = \geq$  and  $\langle e_1 \geq e_2, s \rangle \longrightarrow \langle e', s' \rangle$  is an instance of (op $\geq$ ) and  $\langle e_1 \geq e_2, s \rangle \longrightarrow \langle e'', s'' \rangle$  is an instance of (op $\geq$ ).

Then for some  $n_1, n_2$  we have  $e_1 = n_1, e_2 = n_2, e' = b = e''$  for  $b = (n_1 \geq n_2)$ , and  $s' = s = s''$ .

**case**  $\langle e_1 \ op \ e_2, s \rangle \longrightarrow \langle e', s' \rangle$  is an instance of (op1) and  $\langle e_1 \ op \ e_2, s \rangle \longrightarrow \langle e'', s'' \rangle$  is an instance of (op1).

Then for some  $e'_1$  and  $e''_1$  we have  $\langle e_1, s \rangle \longrightarrow \langle e'_1, s' \rangle$  (\*),  $\langle e_1, s \rangle \longrightarrow \langle e''_1, s'' \rangle$  (\*\*),  $e' = e'_1 \ op \ e_2$ , and  $e'' = e''_1 \ op \ e_2$ . Now, by the induction hypothesis  $\Phi(e_1)$ , (\*) and (\*\*) we have  $\langle e'_1, s' \rangle = \langle e''_1, s'' \rangle$ , so  $\langle e', s' \rangle = \langle e'_1 \ op \ e_2, s' \rangle = \langle e''_1 \ op \ e_2, s'' \rangle = \langle e'', s'' \rangle$  as required.

**case**  $\langle e_1 \ op \ e_2, s \rangle \longrightarrow \langle e', s' \rangle$  is an instance of (op2) and  $\langle e_1 \ op \ e_2, s \rangle \longrightarrow \langle e'', s'' \rangle$  is an instance of (op2).

Similar, save that we use the induction hypothesis  $\Phi(e_2)$ .

**Case**  $e_1; e_2$ . Suppose  $\Phi(e_1)$  and  $\Phi(e_2)$ .

Take arbitrary  $s, e', s', e'', s''$  such that  $\langle e_1; e_2, s \rangle \longrightarrow \langle e', s' \rangle \wedge \langle e_1; e_2, s \rangle \longrightarrow \langle e'', s'' \rangle$ .

By examining the expressions in the left-hand-sides of the conclusions of the rules, and using the lemma above, the only possibilities are those below.

**case**  $e_1 = \text{skip}$  and both transitions are instances of (seq1).

Then  $\langle e', s' \rangle = \langle e_2, s \rangle = \langle e'', s'' \rangle$ .

**case**  $e_1$  is not a value and both transitions are instances of (seq2). Then for some  $e'_1$  and  $e''_1$  we have  $\langle e_1, s \rangle \longrightarrow \langle e'_1, s' \rangle$  (\*),  $\langle e_1, s \rangle \longrightarrow \langle e''_1, s'' \rangle$  (\*\*),  $e' = e'_1; e_2$ , and  $e'' = e''_1; e_2$ .

Then by the induction hypothesis  $\Phi(e_1)$  we have  $\langle e'_1, s' \rangle = \langle e''_1, s'' \rangle$ , so  $\langle e', s' \rangle = \langle e'_1; e_2, s' \rangle = \langle e''_1; e_2, s'' \rangle = \langle e'', s'' \rangle$  as required.

**Case** **while**  $e_1$  **do**  $e_2$ . Suppose  $\Phi(e_1)$  and  $\Phi(e_2)$ .

Take arbitrary  $s, e', s', e'', s''$  such that  $\langle \text{while } e_1 \text{ do } e_2, s \rangle \longrightarrow \langle e', s' \rangle \wedge \langle \text{while } e_1 \text{ do } e_2, s \rangle \longrightarrow \langle e'', s'' \rangle$ .

By examining the expressions in the left-hand-sides of the conclusions of the rules both must be instances of (while), so  $\langle e', s' \rangle = \langle \text{if } e_1 \text{ then } (e_2; \text{while } e_1 \text{ do } e_2) \text{ else skip}, s \rangle = \langle e'', s'' \rangle$ .

**Case** **if**  $e_1$  **then**  $e_2$  **else**  $e_3$ . Suppose  $\Phi(e_1)$ ,  $\Phi(e_2)$  and  $\Phi(e_3)$ .

Take arbitrary  $s, e', s', e'', s''$  such that  $\langle \text{if } e_1 \text{ then } e_2 \text{ else } e_3, s \rangle \longrightarrow \langle e', s' \rangle \wedge \langle \text{if } e_1 \text{ then } e_2 \text{ else } e_3, s \rangle \longrightarrow \langle e'', s'' \rangle$ .

By examining the expressions in the left-hand-sides of the conclusions of the rules, and using the lemma above, the only possibilities are those below.

**case**  $e_1 = \text{true}$  and both transitions are instances of (if1).

**case**  $e_1 = \text{false}$  and both transitions are instances of (if2).



case  $e_1$  is not a value and both transitions are instances of (if3).

The first two cases are immediate; the last uses  $\Phi(e_1)$ .

□

(check we've done all the cases!)

(note that the level of written detail can vary, as here – if you and the reader agree – but you must do all the steps in your head. If in any doubt, write it down, as an aid to thought...!)

Slide 87

**Lemma: Values don't reduce**

**Lemma 10** 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  $n, b, \mathbf{skip}$ . 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  $n, b, \mathbf{skip}$ . □

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

**Proof** Take

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

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

**Case (int).** Recall the rule scheme

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

It has no premises, so we have to show that for all instances  $\Gamma, e, T$  of the conclusion we have  $\Phi(\Gamma, e, T)$ .

For any such instance, there must be an  $n \in \mathbb{Z}$  for which  $e = n$ .

Now  $\Phi$  is of the form  $\forall s. \text{dom}(\Gamma) \subseteq \text{dom}(s) \Rightarrow \dots$ , so consider an arbitrary  $s$  and assume  $\text{dom}(\Gamma) \subseteq \text{dom}(s)$ .

We have to show  $\text{value}(e) \vee (\exists e', s'. \langle e, s \rangle \longrightarrow \langle e', s' \rangle)$ . But the first disjunct is true as integers are values (according to the definition).

**Case (bool)** similar.

**Case (op+).** Recall the rule

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

We have to show that for all  $\Gamma, e_1, e_2$ , if  $\Phi(\Gamma, e_1, \text{int})$  and  $\Phi(\Gamma, e_2, \text{int})$  then  $\Phi(\Gamma, e_1 + e_2, \text{int})$ .

Suppose  $\Phi(\Gamma, e_1, \text{int})$  (\*),  $\Phi(\Gamma, e_2, \text{int})$  (\*\*),  $\Gamma \vdash e_1:\text{int}$  (\*\*\*), and  $\Gamma \vdash e_2:\text{int}$  (\*\*\*\*) (note that we're using the variant form of rule induction here).

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

We have to show  $\text{value}(e_1 + e_2) \vee (\exists e', s'. \langle e_1 + e_2, s \rangle \longrightarrow \langle e', s' \rangle)$ .

Now the first disjunct is false ( $e_1 + e_2$  is not a value), so we have to show the second, i.e.  $\exists \langle e', s' \rangle. \langle e_1 + e_2, s \rangle \longrightarrow \langle e', s' \rangle$ .

By (\*) one of the following holds.

**case**  $\exists e'_1, s'. \langle e_1, s \rangle \longrightarrow \langle e'_1, s' \rangle$ .

Then by (op1) we have  $\langle e_1 + e_2, s \rangle \longrightarrow \langle e'_1 + e_2, s' \rangle$ , so we are done.

**case**  $e_1$  is a value. By (\*\*) one of the following holds.

**case**  $\exists e'_2, s'. \langle e_2, s \rangle \longrightarrow \langle e'_2, s' \rangle$ .

Then by (op2)  $\langle e_1 + e_2, s \rangle \longrightarrow \langle e_1 + e'_2, s' \rangle$ , so we are done.

**case**  $e_2$  is a value.

(Now want to use (op+), but need to know that  $e_1$  and  $e_2$  are really integers. )

**Lemma 11** *for all  $\Gamma, e, T$ , if  $\Gamma \vdash e:T$ ,  $e$  is a value and  $T = \text{int}$  then for some  $n \in \mathbb{Z}$  we have  $e = n$ .*

**Proof** By rule induction. Take  $\Phi'(\Gamma, e, T) = ((\text{value}(e) \wedge T = \text{int}) \Rightarrow \exists n \in \mathbb{Z}. e = n)$ .

**Case (int).** ok

**Case** (bool),(skip). In instances of these rules the conclusion is a value but the type is not **int**, so ok.

**Case** otherwise. In instances of all other rules the conclusion is not a value, so ok.

(a rather trivial use of rule induction – we never needed to use the induction hypothesis, just to do case analysis of the last rule that might have been used in a derivation of  $\Gamma \vdash e:T$ ).  $\square$

Using the Lemma, (\*\*\*) and (\*\*\*\*) there exist  $n_1 \in \mathbb{Z}$  and  $n_2 \in \mathbb{Z}$  such that  $e_1 = n_1$  and  $e_2 = n_2$ . Then by (op+)  $\langle e_1 + e_2, s \rangle \longrightarrow \langle n, s \rangle$  where  $n = n_1 + n_2$ , so we are done.

**Case** (op  $\geq$ ). Similar to (op +).

**Case** (if). Recall the rule

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

Suppose  $\Phi(\Gamma, e_1, \mathbf{bool})$  (\*1),  $\Phi(\Gamma, e_2, T)$  (\*2),  $\Phi(\Gamma, e_3, T)$  (\*3),  $\Gamma \vdash e_1:\mathbf{bool}$  (\*4),  $\Gamma \vdash e_2:T$  (\*5) and  $\Gamma \vdash e_3:T$  (\*6).

Consider an arbitrary  $s$ . Assume  $\text{dom}(\Gamma) \subseteq \text{dom}(s)$ . Write  $e$  for **if**  $e_1$  **then**  $e_2$  **else**  $e_3$ .

This  $e$  is not a value, so we have to show  $\langle e, s \rangle$  has a transition.

**case**  $\exists e'_1, s'. \langle e_1, s \rangle \longrightarrow \langle e'_1, s' \rangle$ .

Then by (if3)  $\langle e, s \rangle \longrightarrow \langle \mathbf{if} \ e'_1 \ \mathbf{then} \ e_2 \ \mathbf{else} \ e_3, s \rangle$ , so we are done.

**case**  $e_1$  is a value.

(Now want to use (if1) or (if2), but need to know that  $e_1 \in \{\mathbf{true}, \mathbf{false}\}$ . Realize should have proved a stronger Lemma above).

**Lemma 12** For all  $\Gamma, e, T$ . if  $\Gamma \vdash e:T$  and  $e$  is a value, then  $T = \mathbf{int} \Rightarrow \exists n \in \mathbb{Z}. e = n$ ,  $T = \mathbf{bool} \Rightarrow \exists b \in \{\mathbf{true}, \mathbf{false}\}. e = b$ , and  $T = \mathbf{unit} \Rightarrow e = \mathbf{skip}$ .

**Proof** By rule induction – details omitted.  $\square$

Using the Lemma and (\*4) we have  $\exists b \in \{\mathbf{true}, \mathbf{false}\}. e_1 = b$ .

**case**  $b = \mathbf{true}$ . Use (if1).

**case**  $b = \mathbf{false}$ . Use (if2).

**Case** (deref). Recall the rule

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

(This is a leaf – it has no  $\Gamma \vdash e:T$  premises - so no  $\Phi$ s to assume).

Consider an arbitrary  $s$  with  $\text{dom}(\Gamma) \subseteq \text{dom}(s)$ .

By the condition  $\Gamma(\ell) = \mathbf{intref}$  we have  $\ell \in \text{dom}(\Gamma)$ , so  $\ell \in \text{dom}(s)$ , so there is some  $n$  with  $s(\ell) = n$ , so there is an instance of (deref)  $\langle !\ell, s \rangle \longrightarrow \langle n, s \rangle$ .

**Cases** (assign), (skip), (seq), (while). Left as an exercise.  $\square$

**Lemma: Values of integer type**

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

Slide 88

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

**Proof** First show the second part, using the following lemma.

**Lemma 14** *If  $\langle e, s \rangle \longrightarrow \langle e', s' \rangle$  then  $\text{dom}(s') = \text{dom}(s)$ .*

**Proof** Rule induction on derivations of  $\langle e, s \rangle \longrightarrow \langle e', s' \rangle$ . Take  $\Phi(e, s, e', s') = (\text{dom}(s) = \text{dom}(s'))$ .

All rules are immediate uses of the induction hypothesis except (assign1), for which we note that if  $\ell \in \text{dom}(s)$  then  $\text{dom}(s + (\ell \mapsto n)) = \text{dom}(s)$ .  $\square$

Now prove the first part, ie If  $\Gamma \vdash e:T$  and  $\text{dom}(\Gamma) \subseteq \text{dom}(s)$  and  $\langle e, s \rangle \longrightarrow \langle e', s' \rangle$  then  $\Gamma \vdash e':T$ .

Prove by rule induction on derivations of  $\langle e, s \rangle \longrightarrow \langle e', s' \rangle$ .

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

**Case** (op+). Recall

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

Take arbitrary  $\Gamma, T$ . Suppose  $\Gamma \vdash n_1 + n_2:T$  (\*) and  $\text{dom}(\Gamma) \subseteq \text{dom}(s)$ . The last rule in the derivation of (\*) must have been (op+), so must have  $T = \text{int}$ . Then can use (int) to derive  $\Gamma \vdash n:T$ .

**Case** (op  $\geq$ ). Similar.

**Case** (op1). Recall

$$\text{(op1)} \quad \frac{\langle e_1, s \rangle \longrightarrow \langle e'_1, s' \rangle}{\langle e_1 \text{ op } e_2, s \rangle \longrightarrow \langle e'_1 \text{ op } e_2, s' \rangle}$$

Suppose  $\Phi(e_1, s, e'_1, s')$  (\*) and  $\langle e_1, s \rangle \longrightarrow \langle e'_1, s' \rangle$ . Have to show  $\Phi(e_1 \text{ op } e_2, s, e'_1 \text{ op } e_2, s')$ . Take arbitrary  $\Gamma, T$ . Suppose  $\Gamma \vdash e_1 \text{ op } e_2:T$  and  $\text{dom}(\Gamma) \subseteq \text{dom}(s)$  (\*\*).

**case** op = +. The last rule in the derivation of  $\Gamma \vdash e_1 + e_2:T$  must have been (op+), so must have  $T = \text{int}$ ,  $\Gamma \vdash e_1:\text{int}$  (\*\*\*) and  $\Gamma \vdash e_2:\text{int}$  (\*\*\*\*). By the induction hypothesis (\*), (\*\*), and (\*\*\*) we have  $\Gamma \vdash e'_1:\text{int}$ . By the (op+) rule  $\Gamma \vdash e'_1 + e_2:T$ .

**case** op =  $\geq$ . Similar.

**Case** s (op2) (deref), (assign1), (assign2), (seq1), (seq2), (if1), (if2), (if3), (while). Left as exercises.  $\square$

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

**Proof** Hint: induction along  $\longrightarrow^*$  using the previous results.  $\square$

**Theorem 7 (Uniqueness of typing)** *If  $\Gamma \vdash e:T$  and  $\Gamma \vdash e:T'$  then  $T = T'$ .* The proof is left as Exercise 17.

**Theorem 5 (Decidability of typeability)** *Given  $\Gamma, e$ , one can decide  $\exists T. \Gamma \vdash e:T$ .*

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

**Proof** The implementation gives a type inference algorithm, which, if correct, and together with Uniqueness, implies both of these results.  $\square$

Slide 89

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

### 3.4 Exercises

You should be able to prove all the theorems about L1 independently. These exercises are to get you started.

**Exercise 12** ★ *Without looking at the proof in the notes, do the cases of the proof of Theorem 1 (Determinacy) for  $e_1 \text{ op } e_2$ ,  $e_1; e_2$ , **while**  $e_1$  **do**  $e_2$ , and **if**  $e_1$  **then**  $e_2$  **else**  $e_3$ .*

**Exercise 13** ★ *Try proving Determinacy for the language with nondeterministic order of evaluation for  $e_1 \text{ op } e_2$  (ie with both (op1) and (op1b) rules), which is not determinate. Explain where exactly the proof can't be carried through.*

**Exercise 14** ★ *Complete the proof of Theorem 2 (Progress).*

**Exercise 15** ★★ *Complete the proof of Theorem 3 (Type Preservation).*

**Exercise 16** ★★ *Give an alternate proof of Theorem 3 (Type Preservation) by rule induction over type derivations.*

**Exercise 17** ★★ *Prove Theorem 7 (Uniqueness of Typing).*

## 4 Functions

Slide 90

### Functions – L2

Slide 91

#### 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>
```

Slide 92

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());  
        }  
    }  
}
```

Most languages have some kind of function, method, or procedure – some way of abstracting a piece of code on a formal parameter so that you can use the code multiple times with different arguments, without having to duplicate the code in the source. The next two lectures explore the design space for functions, adding them to L1.

### Functions – Examples

We will add expressions like these to L1.

```
(fn x:int => x + 1)
(fn x:int => x + 1) 7
(fn y:int => (fn x:int => x + y))
(fn y:int => (fn x:int => x + y)) 1
(fn x:int -> int => (fn y:int => x (x y)))
(fn x:int -> int => (fn y:int => x (x y))) (fn x:int => x + 1)
((fn x:int -> int => (fn y:int => x (x y))) (fn x:int => x + 1)) 7
```

Slide 93

For simplicity, we'll deal with *anonymous* functions only. Functions will always take a single argument and return a single result — though either might itself be a function or a tuple.

### Functions – Syntax

First, extend the L1 syntax:

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

Expressions

$$e ::= \dots \mid \text{fn } x:T \Rightarrow e \mid e_1 e_2 \mid x$$

Types

$$T ::= \text{int} \mid \text{bool} \mid \text{unit} \mid T_1 \rightarrow T_2$$

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

Slide 94

**Concrete syntax.** By convention, application associates to the left, so  $e_1 e_2 e_3$  denotes  $(e_1 e_2) e_3$ , and type arrows associate to the right, so  $T_1 \rightarrow T_2 \rightarrow T_3$  denotes  $T_1 \rightarrow (T_2 \rightarrow T_3)$ . A **fn** extends to the right as far as parentheses permit, so **fn**  $x:\text{unit} \Rightarrow x; x$  denotes **fn**  $x:\text{unit} \Rightarrow (x; x)$ , not **(fn**  $x:\text{unit} \Rightarrow x); x$ . These conventions work well for functions that take several arguments, e.g. **fn**  $x:\text{unit} \Rightarrow \text{fn } y:\text{int} \Rightarrow x; y$  has type  $\text{unit} \rightarrow \text{int} \rightarrow \text{int}$ , and we can fully apply it simply by juxtaposing it with its two arguments **(fn**  $x:\text{unit} \Rightarrow \text{fn } y:\text{int} \Rightarrow x; y)$  **skip 15**.

- Variables are not locations ( $\mathbb{L} \cap \mathbb{X} = \{\}$ ), so  $x := 3$  is not in the syntax.
- You can't abstract on locations. For example, **(fn**  $l:\text{intref} \Rightarrow !l)$  is not in the syntax.
- The (non-meta) variables  $x, y, z$  are not the same as metavariables  $x, y, z$ . In the notes they are distinguished by font; in handwriting one just has to keep track in your head — not often a problem.
- These expressions look like lambda terms (and **fn**  $x:\text{int} \Rightarrow x$  could be written  $\lambda x:\text{int}.x$ ). But, (a) we're adding them to a rich language, not working with the pure lambda calculus (cf. *Foundations of Functional Programming*), and (b) we're going to explore several options for how they should behave.

**Type-directed language design.** This type grammar (and expression syntax) suggests the language will include higher-order functions — you can abstract on a variable of any type, including function types. If you only wanted first-order functions, you'd say

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

Note that first-order function types include types like  $\text{int} \rightarrow (\text{int} \rightarrow \text{int})$  and  $\text{int} \rightarrow (\text{int} \rightarrow (\text{int} \rightarrow \text{int}))$ , of functions that take an argument of base type and return a (first-order) function, e.g.

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

Some languages go further, forbidding partial application. We'll come back to this.

## 4.1 Abstract syntax up to alpha conversion, and substitution

In order to express the semantics for functions, we need some auxiliary definitions.

Slide 95

Variable shadowing

```

(fn x:int ⇒ (fn x:int ⇒ x + 1))

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

```

Variable shadowing is not allowed in Java. For large systems that would be a problem, eg in a language with nested function definitions, where you may wish to write a local function parameter without being aware of what is in the surrounding namespace.

Slide 96

Alpha conversion

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

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

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

Slide 97

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  $(\text{fn } x:T \Rightarrow \dots)$ . For example:

```

17
x + y
fn x:int ⇒ x + 2
fn x:int ⇒ x + z
if y then 2 + x else ((fn x:int ⇒ x + 2)z)

```

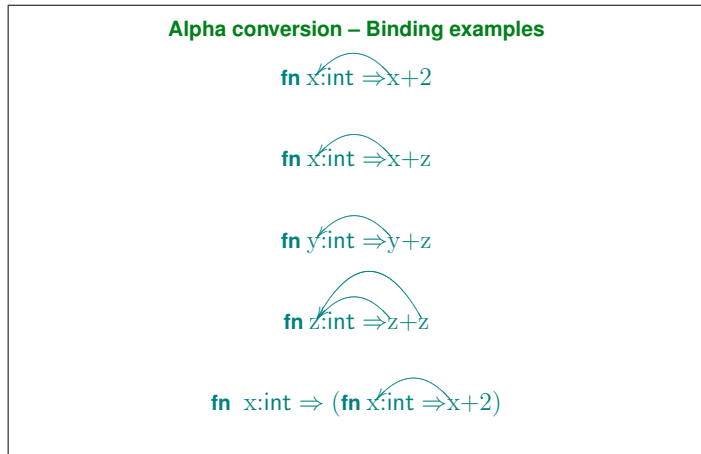
All the other occurrences of  $x$  are *bound* by the closest enclosing  $\text{fn } x:T \Rightarrow \dots$ .

Note that in  $\text{fn } x:\text{int} \Rightarrow 2$  the  $x$  is not an occurrence. Likewise, in  $\text{fn } x:\text{int} \Rightarrow x + 2$  the left  $x$  is not an occurrence; here the right  $x$  is an occurrence that is bound by the left  $x$ .

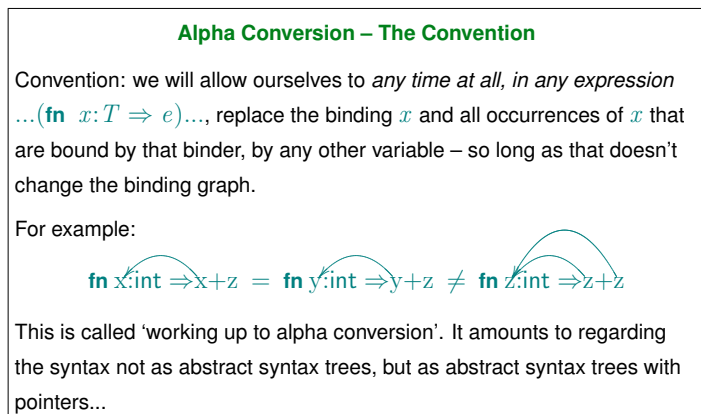


Sometimes it is handy to draw in the binding:

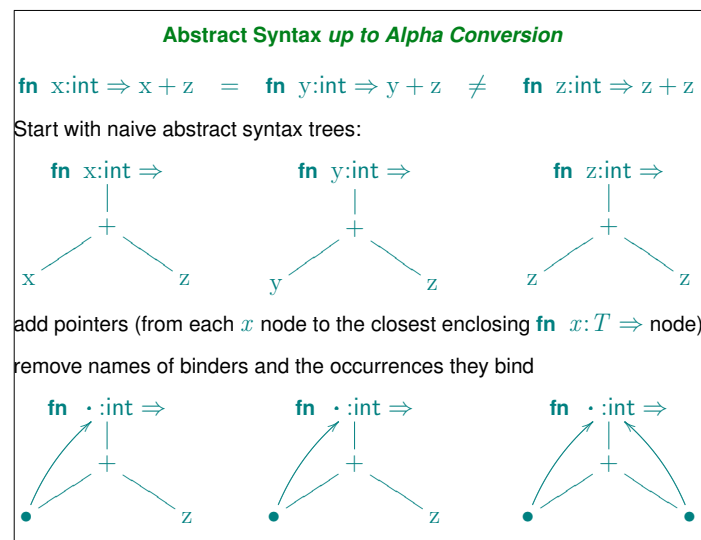
Slide 98



Slide 99



Slide 100



Slide 101

$\text{fn } x:\text{int} \Rightarrow (\text{fn } x:\text{int} \Rightarrow x + 2)$   
 $= \text{fn } y:\text{int} \Rightarrow (\text{fn } z:\text{int} \Rightarrow z + 2) \neq \text{fn } z:\text{int} \Rightarrow (\text{fn } y:\text{int} \Rightarrow z + 2)$

Slide 102

$(\text{fn } x:\text{int} \Rightarrow x) 7$      $\text{fn } z:\text{int} \rightarrow \text{int} \rightarrow \text{int} \Rightarrow (\text{fn } y:\text{int} \Rightarrow z y y)$

Slide 103

**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  $\text{fn } \cdot : T \Rightarrow$  nodes you have to count out to get to its binder.

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

Slide 104

**Free Variables**

Say the *free variables* of an expression  $e$  are the set of variables  $x$  for which there is an occurrence of  $x$  free in  $e$ .

$$\text{fv}(x) = \{x\}$$

$$\text{fv}(e_1 \text{ op } e_2) = \text{fv}(e_1) \cup \text{fv}(e_2)$$

$$\text{fv}(\text{fn } x:T \Rightarrow e) = \text{fv}(e) - \{x\}$$

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

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

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

For example

$$\begin{aligned} \text{fv}(x + y) &= \{x, y\} \\ \text{fv}(\mathbf{fn} \ x:\text{int} \Rightarrow x + y) &= \{y\} \\ \text{fv}(x + (\mathbf{fn} \ x:\text{int} \Rightarrow x + y)7) &= \{x, y\} \end{aligned}$$

Full definition of  $\text{fv}(e)$  is by recursion on the structure of  $e$ :

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

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

Slide 105

**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

$$\begin{aligned} \{3/x\}(x \geq x) &= (3 \geq 3) \\ \{3/x\}((\mathbf{fn} \ x:\text{int} \Rightarrow x + y)x) &= (\mathbf{fn} \ x:\text{int} \Rightarrow x + y)3 \\ \{y + 2/x\}(\mathbf{fn} \ y:\text{int} \Rightarrow x + y) &= \mathbf{fn} \ z:\text{int} \Rightarrow (y + 2) + z \end{aligned}$$

Note that substitution is a meta-operation – it's *not* part of the L2 expression grammar.

The notation used for substitution varies – people write  $\{3/x\}e$ , or  $[3/x]e$ , or  $e[3/x]$ , or  $\{x \leftarrow 3\}e$ , or...

Slide 106

**Substitution – Definition**

Defining that:

$$\begin{aligned} \{e/z\}x &= e && \text{if } x = z \\ &= x && \text{otherwise} \\ \{e/z\}(\mathbf{fn} \ x:T \Rightarrow e_1) &= \mathbf{fn} \ x:T \Rightarrow (\{e/z\}e_1) && \text{if } x \neq z \text{ (*)} \\ &&& \text{and } x \notin \text{fv}(e) \text{ (*)} \\ \{e/z\}(e_1 \ e_2) &= (\{e/z\}e_1)(\{e/z\}e_2) \\ \dots & \end{aligned}$$

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

Slide 107

**Substitution – Example Again**

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

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

Slide 108

**Simultaneous substitution**

A *substitution*  $\sigma$  is a finite partial function from variables to expressions.

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

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

Write  $\mathit{dom}(\sigma)$  for the set of variables in the domain of  $\sigma$ ;  $\mathit{ran}(\sigma)$  for the set of expressions in the range of  $\sigma$ , ie

$$\begin{aligned} \mathit{dom}(\{e_1/x_1, \dots, e_k/x_k\}) &= \{x_1, \dots, x_k\} \\ \mathit{ran}(\{e_1/x_1, \dots, e_k/x_k\}) &= \{e_1, \dots, e_k\} \end{aligned}$$

Define the application of simultaneous substitution to a term by:

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

## 4.2 Function Behaviour

Slide 109

**Function Behaviour**

Consider the expression

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

then

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

### 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.

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

Slide 110

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

This is a common design choice — ML, Java. It is a *strict* semantics – fully evaluating the argument to function before doing the application.

### L2 Call-by-value

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

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

Slide 111

### L2 Call-by-value – reduction examples

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

Slide 112

$(\mathbf{fn} \ f:\mathbf{int} \rightarrow \mathbf{int} \Rightarrow f \ 3) (\mathbf{fn} \ x:\mathbf{int} \Rightarrow (1 + 2) + x)$

- The rules for these constructs don't touch the store. In a *pure* functional language, configurations would just be expressions.
- A naive implementation of these rules would have to traverse  $e$  and copy  $v$  as many times as there are free occurrences of  $x$  in  $e$ . Real implementations don't do that, using *environments* instead of doing substitution. Environments are more efficient; substitutions are simpler to write down – so better for implementation and semantics respectively.

### 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.

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

Slide 113

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

This is the foundation of ‘lazy’ functional languages – e.g. Haskell

### L2 Call-by-name

(same typing rules as before)

$$\text{(CBN-app)} \quad \frac{\langle e_1, s \rangle \longrightarrow \langle e'_1, s' \rangle}{\langle e_1 \ e_2, s \rangle \longrightarrow \langle e'_1 \ e_2, s' \rangle}$$

Slide 114

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

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

$$\begin{aligned} &\langle (\mathbf{fn} \ x:\mathbf{unit} \Rightarrow \mathbf{skip})(l := 2), \{l \mapsto 0\} \rangle \\ &\longrightarrow \langle \{l := 2/x\}\mathbf{skip} \quad , \{l \mapsto 0\} \rangle \\ &= \langle \mathbf{skip} \quad , \{l \mapsto 0\} \rangle \end{aligned}$$

but if it is, end up evaluating it repeatedly.

### 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.

Slide 115

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

### L2 Beta

$$\text{(beta-app1)} \quad \frac{\langle e_1, s \rangle \longrightarrow \langle e'_1, s' \rangle}{\langle e_1 \ e_2, s \rangle \longrightarrow \langle e'_1 \ e_2, s' \rangle}$$

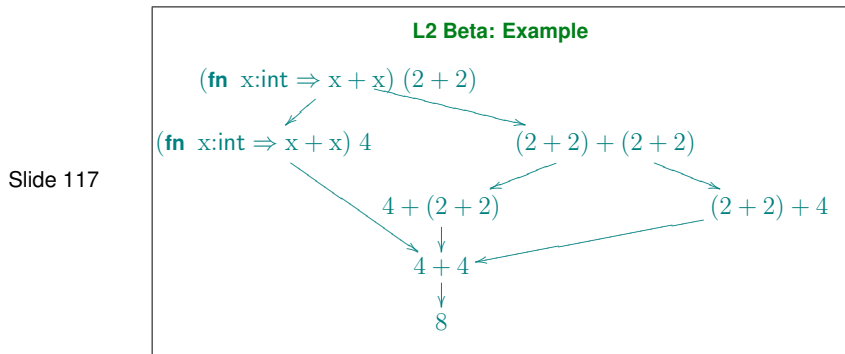
Slide 116

$$\text{(beta-app2)} \quad \frac{\langle e_2, s \rangle \longrightarrow \langle e'_2, s' \rangle}{\langle e_1 \ e_2, s \rangle \longrightarrow \langle e_1 \ e'_2, s' \rangle}$$

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

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

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



Slide 118

**Function Behaviour. Choice 4: Normal-order reduction**

Leftmost, outermost variant of full beta.

- What will `(fn x:unit => skip) (while true do skip)` do in the different semantics?
- What about `(fn x:unit => skip) (ℓ := !ℓ + 1)`?

Slide 119

**Purity**

Without strict, call-by-value semantics, it becomes hard to understand what order your code is going to be run in. Non-strict languages typically don't allow unrestricted side effects (our combination of store and CBN is *pretty odd*). Haskell encourages *pure* programming, without effects (store operations, IO, etc.) except where really necessary. Where they *are* necessary, it uses a fancy type system to give you some control of evaluation order.

For a pure language, Call-By-Name gives the same results as Call-By-Need, which is more efficient. The first time the argument evaluated we 'overwrite' all other copies by that value.

Slide 120

**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!

```

Slide 121

Back to CBV (from now on).

### 4.3 Function Typing

**Typing functions (1)**

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

Now, it must also give assumptions on the types of variables: e.g.  
 $l_1:\text{intref}, x:\text{int}, y:\text{bool} \rightarrow \text{int}$ .

Slide 122 Type environments,  $\Gamma \in \text{TypeEnv2}$ , are finite partial functions from  $\mathbb{L} \cup \mathbb{X}$  to  $\mathbb{T}_{\text{loc}} \cup \mathbb{T}$  such that

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

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

Notation: if  $x \notin \text{dom}(\Gamma)$ , write  $\Gamma, x:T$  for the partial function which maps  $x$  to  $T$  but otherwise is like  $\Gamma$ .

**Typing functions (2)**

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

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

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

Slide 123

**Typing functions – Example**

$$\frac{\frac{\frac{\overline{x:\text{int} \vdash x:\text{int}} \text{ (var)}}{x:\text{int} \vdash x + 2:\text{int}} \text{ (op+)}}{\{\} \vdash (\text{fn } x:\text{int} \Rightarrow x + 2):\text{int} \rightarrow \text{int}} \text{ (fn)}}{\{\} \vdash (\text{fn } x:\text{int} \Rightarrow x + 2) 2:\text{int}} \text{ (app)}$$

Slide 124

- Note that sometimes you need the alpha convention, e.g. to type

**fn**  $x:\text{int} \Rightarrow x + (\text{fn } x:\text{bool} \Rightarrow \text{if } x \text{ then } 3 \text{ else } 4)\text{true}$

It's a good idea to start out with all binders different from each other and from all free variables. It would be a bad idea to prohibit variable shadowing like this in source programs.

- In ML you have *parametrically polymorphic* functions, e.g.  $(\text{fn } x:\alpha \Rightarrow x):\alpha \rightarrow \alpha$ , but we won't talk about them here – that's in Part II *Types*.

Another example:

$$\frac{\frac{\frac{\overline{l:\text{intref}, x:\text{unit} \vdash 1:\text{int}} \text{ (int)}}{l:\text{intref}, x:\text{unit} \vdash (l := 1):\text{unit}} \text{ (asn)}}{l:\text{intref}, x:\text{unit} \vdash (l := 1); x:\text{unit}} \text{ (seq)}}{\frac{\overline{l:\text{intref} \vdash 2:\text{int}} \text{ (int)}}{l:\text{intref} \vdash (l := 2):\text{unit}} \text{ (asn)}}{l:\text{intref} \vdash (\text{fn } x:\text{unit} \Rightarrow (l := 1); x) (l := 2):\text{unit}} \text{ (app)}$$



### Properties of Typing

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

**Theorem 15 (Progress)** *If  $e$  closed and  $\Gamma \vdash e:T$  and  $\text{dom}(\Gamma) \subseteq \text{dom}(s)$  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 16 (Type Preservation)** *If  $e$  closed and  $\Gamma \vdash e:T$  and  $\text{dom}(\Gamma) \subseteq \text{dom}(s)$  and  $\langle e, s \rangle \longrightarrow \langle e', s' \rangle$  then  $\Gamma \vdash e':T$  and  $e'$  closed and  $\text{dom}(\Gamma) \subseteq \text{dom}(s')$ .*

Slide 125

### Proving Type Preservation

**Theorem 16 (Type Preservation)** *If  $e$  closed and  $\Gamma \vdash e:T$  and  $\text{dom}(\Gamma) \subseteq \text{dom}(s)$  and  $\langle e, s \rangle \longrightarrow \langle e', s' \rangle$  then  $\Gamma \vdash e':T$  and  $e'$  closed and  $\text{dom}(\Gamma) \subseteq \text{dom}(s')$ .*

Taking

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

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.

Slide 126

To prove this one uses:

**Lemma 17 (Substitution)** *If  $\Gamma \vdash e:T$  and  $\Gamma, x:T \vdash e':T'$  with  $x \notin \text{dom}(\Gamma)$  then  $\Gamma \vdash \{e/x\}e':T'$ .*

Slide 127

Determinacy and type inference properties also hold.

### Normalization

**Theorem 18 (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  $\langle e, \{\} \rangle \longrightarrow \langle e_1, \{\} \rangle \longrightarrow \langle e_2, \{\} \rangle \longrightarrow \dots$*

**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).  $\square$

Slide 128

## 4.4 Local Definitions and Recursive Functions

Slide 129

**Local definitions**

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

$$e ::= \dots \mid \mathbf{let\ val\ } x:T = e_1 \mathbf{\ in\ } e_2 \mathbf{\ end}$$

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

Can regard just as *syntactic sugar*:

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

Slide 130

**Local definitions – derived typing and reduction rules (CBV)**

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

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

(let1)

$$\frac{\langle e_1, s \rangle \longrightarrow \langle e'_1, s' \rangle}{\langle \mathbf{let\ val\ } x:T = e_1 \mathbf{\ in\ } e_2 \mathbf{\ end}, s \rangle \longrightarrow \langle \mathbf{let\ val\ } x:T = e'_1 \mathbf{\ in\ } e_2 \mathbf{\ end}, s' \rangle}$$

(let2)

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

Our alpha convention means this really is a local definition – there is no way to refer to the locally-defined variable outside the **let val** .

$$x + \mathbf{let\ val\ } x:\text{int} = x \mathbf{\ in\ } (x + 2) \mathbf{\ end} = x + \mathbf{let\ val\ } y:\text{int} = x \mathbf{\ in\ } (y + 2) \mathbf{\ end}$$

Slide 131

**Recursive definitions – first attempt**

How about

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

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

Could add something like this:

$$e ::= \dots \mid \mathbf{let\ val\ rec\ } x:T = e \mathbf{\ in\ } e' \mathbf{\ end}$$

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

$$\mathbf{let\ val\ rec\ } x:\text{int} \rightarrow \text{int} =$$

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

$$\mathbf{in\ } x\ 3 \mathbf{\ end}$$

Slide 132

**But...**

What about  
**let val rec**  $x = (x, x)$  **in**  $x$  **end** ?

Have some rather weird things, eg  
**let val rec**  $x:\text{int}$   $\text{list} = 3 :: x$  **in**  $x$  **end**

does that terminate? if so, is it equal to  
**let val rec**  $x:\text{int}$   $\text{list} = 3 :: 3 :: x$  **in**  $x$  **end** ? does  
**let val rec**  $x:\text{int}$   $\text{list} = 3 :: (x + 1)$  **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...

Slide 133

**Recursive Functions**

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

$$T = T_1 \rightarrow T_2 \quad \text{recursion only at function types}$$

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

$$e ::= \dots \mid \text{let val rec } x:T_1 \rightarrow T_2 = (\text{fn } y:T_1 \Rightarrow e_1) \text{ in } e_2 \text{ end}$$

(here the  $y$  binds in  $e_1$ ; the  $x$  binds in  $(\text{fn } y:T_1 \Rightarrow e_1)$  and in  $e_2$ )

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

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**  $f:T_1 \rightarrow T_2 = \text{fn } x:T_1 \Rightarrow e_1$  **in**  $e_2$  **end**.

Slide 134

**Recursive Functions – Semantics**

$$(\text{letrecfn}) \quad \text{let val rec } x:T_1 \rightarrow T_2 = (\text{fn } y:T_1 \Rightarrow e_1) \text{ in } e_2 \text{ end}$$

$$\rightarrow$$

$$\{(\text{fn } y:T_1 \Rightarrow \text{let val rec } x:T_1 \rightarrow T_2 = (\text{fn } y:T_1 \Rightarrow e_1) \text{ in } e_1 \text{ end})/x\}e_2$$

For example:

```

let val rec x:int → int =
  (fn y:int ⇒ if y ≥ 1 then y + (x(y + -1)) else 0)
in
  x 3
end
→ (letrecfn)
(fn y:int ⇒
  let val rec x:int → int =
    (fn y:int ⇒ if y ≥ 1 then y + (x(y + -1)) else 0)
  in
    if y ≥ 1 then y + (x(y + -1)) else 0
  end ) 3
  ) 3
→ (app)
let val rec x:int → int =
  (fn y:int ⇒ if y ≥ 1 then y + (x(y + -1)) else 0)
in
  if 3 ≥ 1 then 3 + (x(3 + -1)) else 0
end
→ (letrecfn)
if 3 ≥ 1 then
  3 + ((fn y:int ⇒
    let val rec x:int → int =
      (fn y:int ⇒ if y ≥ 1 then y + (x(y + -1)) else 0)
    in
      if y ≥ 1 then y + (x(y + -1)) else 0
    end )(3 + -1))
else
  0
→ ...

```

Slide 135

**Recursive Functions – Minimization Example**

Below, in the context of the **let val rec**,  $x f n$  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

```

As a test case, we apply it to the function  $(fn z:int ⇒ if z ≥ 3 then (if 3 ≥ z then 0 else 1) else 1)$ , which is 0 for argument 3 and 1 elsewhere.

Slide 136

**More Syntactic Sugar**

Do we need  $e_1; e_2$ ?

No: Could encode by  $e_1; e_2 \rightsquigarrow (\text{fn } y:\text{unit} \Rightarrow e_2)e_1$

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

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

```
let val rec w:unit → unit =
  fn y:unit ⇒ if e1 then (e2; (w skip)) else skip
in
  w skip
end
```

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

In each case typing is the same. Reduction is ‘essentially’ the same — we will be able to make this precise when we study contextual equivalence.

## 4.5 Implementation

Slide 137

**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`.

The implementation lets you type in L2 expressions and initial stores and watch them resolve, type-check, and reduce.

Slide 138

**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
```

(it raises an exception if the expression has any free variables)

Slide 139

### 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)
  | ...
```

If  $e'$  represents a closed term  $\text{fn } x:T \Rightarrow e'_1$  then  $e' = \text{Fn}(t,e1')$  for  $t$  and  $e1'$  representing  $T$  and  $e'_1$ . If also  $e$  represents a closed term  $e$  then  $\text{subst } e \ 0 \ e1'$  represents  $\{e/x\}e'_1$ .

Slide 140

### 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 ))
```

Slide 141

### 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) of
  (SOME (func(t1,t1')), SOME t2) =>
    if t1=t2 then SOME t1' else NONE
  | _ => NONE )
```

### Implementation – Closures

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

Slide 142

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

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

Slide 143

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

(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.

## 4.6 L2: Collected Definition

### Syntax

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

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

Locations  $\ell \in \mathbb{L} = \{\ell, \ell_0, \ell_1, \ell_2, \dots\}$

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

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

Types

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

Expressions

$$\begin{aligned} e &::= n \mid b \mid e_1 \ op \ e_2 \mid \mathbf{if} \ e_1 \ \mathbf{then} \ e_2 \ \mathbf{else} \ e_3 \mid \\ &\ell := e \mid !\ell \mid \\ &\mathbf{skip} \mid e_1; e_2 \mid \\ &\mathbf{while} \ e_1 \ \mathbf{do} \ e_2 \mid \\ &\mathbf{fn} \ x:T \Rightarrow e \mid e_1 \ e_2 \mid x \mid \\ &\mathbf{let} \ \mathbf{val} \ x:T = e_1 \ \mathbf{in} \ e_2 \ \mathbf{end} \mid \\ &\mathbf{let} \ \mathbf{val} \ \mathbf{rec} \ x:T_1 \rightarrow T_2 = (\mathbf{fn} \ y:T_1 \Rightarrow e_1) \ \mathbf{in} \ e_2 \ \mathbf{end} \end{aligned}$$

In expressions  $\mathbf{fn} \ x:T \Rightarrow e$  the  $x$  is a *binder*. In expressions  $\mathbf{let} \ \mathbf{val} \ x:T = e_1 \ \mathbf{in} \ e_2 \ \mathbf{end}$  the  $x$  is a binder. In expressions  $\mathbf{let} \ \mathbf{val} \ \mathbf{rec} \ x:T_1 \rightarrow T_2 = (\mathbf{fn} \ y:T_1 \Rightarrow e_1) \ \mathbf{in} \ e_2 \ \mathbf{end}$  the  $y$  binds in  $e_1$ ; the  $x$  binds in  $(\mathbf{fn} \ y:T \Rightarrow e_1)$  and in  $e_2$ .

### Operational Semantics

Say *stores*  $s$  are finite partial functions from  $\mathbb{L}$  to  $\mathbb{Z}$ . Values  $v ::= b \mid n \mid \mathbf{skip} \mid \mathbf{fn} \ x:T \Rightarrow e$

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

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

$$(\text{op1}) \quad \frac{\langle e_1, s \rangle \longrightarrow \langle e'_1, s' \rangle}{\langle e_1 \ op \ e_2, s \rangle \longrightarrow \langle e'_1 \ op \ e_2, s' \rangle}$$

$$(\text{op2}) \quad \frac{\langle e_2, s \rangle \longrightarrow \langle e'_2, s' \rangle}{\langle v \ op \ e_2, s \rangle \longrightarrow \langle v \ op \ e'_2, s' \rangle}$$

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

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

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

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

$$(\text{seq2}) \quad \frac{\langle e_1, s \rangle \longrightarrow \langle e'_1, s' \rangle}{\langle e_1; e_2, s \rangle \longrightarrow \langle e'_1; e_2, s' \rangle}$$



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

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

$$(if3) \quad \frac{\langle e_1, s \rangle \longrightarrow \langle e'_1, s' \rangle}{\langle \mathbf{if\ } e_1 \ \mathbf{then\ } e_2 \ \mathbf{else\ } e_3, s \rangle \longrightarrow \langle \mathbf{if\ } e'_1 \ \mathbf{then\ } e_2 \ \mathbf{else\ } e_3, s' \rangle}$$

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

$$(app1) \quad \frac{\langle e_1, s \rangle \longrightarrow \langle e'_1, s' \rangle}{\langle e_1 \ e_2, s \rangle \longrightarrow \langle e'_1 \ e_2, s' \rangle}$$

$$(app2) \quad \frac{\langle e_2, s \rangle \longrightarrow \langle e'_2, s' \rangle}{\langle v \ e_2, s \rangle \longrightarrow \langle v \ e'_2, s' \rangle}$$

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

$$(let1) \quad \frac{\langle e_1, s \rangle \longrightarrow \langle e'_1, s' \rangle}{\langle \mathbf{let\ val\ } x:T = e_1 \ \mathbf{in\ } e_2 \ \mathbf{end}, s \rangle \longrightarrow \langle \mathbf{let\ val\ } x:T = e'_1 \ \mathbf{in\ } e_2 \ \mathbf{end}, s' \rangle}$$

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

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

## Typing

Type environments,  $\Gamma \in \text{TypeEnv2}$ , are finite partial functions from  $\mathbb{L} \cup \mathbb{X}$  to  $\mathbb{T}_{\text{loc}} \cup \mathbb{T}$  such that

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

## 4.7 Exercises

**Exercise 18** ★ *What are the free variables of the following?*

1.  $x + ((\text{fn } y:\text{int} \Rightarrow z) 2)$
2.  $x + (\text{fn } y:\text{int} \Rightarrow z)$
3.  $\text{fn } y:\text{int} \Rightarrow \text{fn } y:\text{int} \Rightarrow \text{fn } y:\text{int} \Rightarrow y$
4.  $!l_0$
5.  $\text{while } !l_0 \geq y \text{ do } l_0 := x$

*Draw their abstract syntax trees (up to alpha equivalence).*

**Exercise 19** ★ *What are the results of the following substitutions?*

1.  $\{\text{fn } x:\text{int} \Rightarrow y/z\} \text{fn } y:\text{int} \Rightarrow z y$
2.  $\{\text{fn } x:\text{int} \Rightarrow x/x\} \text{fn } y:\text{int} \Rightarrow x y$
3.  $\{\text{fn } x:\text{int} \Rightarrow x/x\} \text{fn } x:\text{int} \Rightarrow x x$

**Exercise 20** ★ *Give typing derivations, or show why no derivation exists, for:*

1.  $\text{if } 6 \text{ then } 7 \text{ else } 8$
2.  $\text{fn } x:\text{int} \Rightarrow x + (\text{fn } x:\text{bool} \Rightarrow \text{if } x \text{ then } 3 \text{ else } 4)\text{true}$

**Exercise 21** ★★ *Give a grammar for types, and typing rules for functions and application, that allow only first-order functions and prohibit partial applications (see page 47).*

**Exercise 22** ★★ *Write a function of type  $\text{unit} \rightarrow \text{bool}$  that, when applied to **skip**, returns **true** in the CBV semantics and **false** in the CBN semantics. Is it possible to do it without using the store?*

**Exercise 23** ★★ *Prove Lemma 17 (Substitution).*

**Exercise 24** ★★ *Prove Theorem 16 (Type Preservation).*

**Exercise 25** ★★ *Adapt the L2 implementation to CBN functions. Think of a few good test cases and check them in the new and old code.*

**Exercise 26** ★★★ *Re-implement the L2 interpreter to use closures instead of substitution.*

## 5 Data

Slide 144

### Data – L3

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

### 5.1 Products and sums

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

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

The sum type  $T_1 + T_2$  lets you form a disjoint union, with a value of the sum type either being a value of type  $T_1$  or a value of type  $T_2$ . In C one has `unions`; in Java one might have many subclasses of a class (see the `11.java` representation of the L1 abstract syntax, for example).

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

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

Slide 145

#### Products

$$T ::= \dots \mid T_1 * T_2$$
$$e ::= \dots \mid (e_1, e_2) \mid \#1 e \mid \#2 e$$

Design choices:

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

Slide 146

**Products – typing**

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

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

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

Slide 147

**Products – reduction**

$v ::= \dots \mid (v_1, v_2)$

$$\text{(pair1)} \quad \frac{\langle e_1, s \rangle \rightarrow \langle e'_1, s' \rangle}{\langle (e_1, e_2), s \rangle \rightarrow \langle (e'_1, e_2), s' \rangle}$$

$$\text{(pair2)} \quad \frac{\langle e_2, s \rangle \rightarrow \langle e'_2, s' \rangle}{\langle (v_1, e_2), s \rangle \rightarrow \langle (v_1, e'_2), s' \rangle}$$

$$\text{(proj1)} \quad \langle \#1(v_1, v_2), s \rangle \rightarrow \langle v_1, s \rangle \quad \text{(proj2)} \quad \langle \#2(v_1, v_2), s \rangle \rightarrow \langle v_2, s \rangle$$

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

We have chosen left-to-right evaluation order for consistency.

Slide 148

**Sums (or Variants, or Tagged Unions)**

$T ::= \dots \mid T_1 + T_2$

$e ::= \dots \mid \text{inl } e : T \mid \text{inr } e : T \mid$

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

Those  $x$ s are binders, treated up to alpha-equivalence.

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

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

Slide 149

**Sums – typing**

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

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

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

Slide 150

**Sums – type annotations**

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

Why do we have these type annotations?

To maintain the unique typing property. Otherwise

**inl** 3:int + int

and

**inl** 3:int + bool

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

Slide 151

**Sums – reduction**

$v ::= \dots \mid \mathbf{inl} \ v:T \mid \mathbf{inr} \ v:T$

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

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

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

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

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

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

Slide 152

**Constructors and Destructors**

type	constructors	destructors
$T \rightarrow T$	<b>fn</b> $x:T \Rightarrow \_$	$\_ \ e$
$T * T$	$(\_, \_)$	$\#1 \ \_ \ \#2 \ \_$
$T + T$	<b>inl</b> $(\_) \ \ \mathbf{inr} \ (\_)$	<b>case</b>
bool	<b>true</b> <b>false</b>	<b>if</b>

Slide 153

Proofs as programs: The Curry-Howard correspondence	
(var) $\Gamma, x:T \vdash x:T$	$\Gamma, P \vdash P$
(fn) $\frac{\Gamma, x:T \vdash e:T'}{\Gamma \vdash \mathbf{fn} \ x:T \Rightarrow e : T \rightarrow T'}$	$\frac{\Gamma, P \vdash P'}{\Gamma \vdash P \rightarrow P'}$
(app) $\frac{\Gamma \vdash e_1:T \rightarrow T' \quad \Gamma \vdash e_2:T}{\Gamma \vdash e_1 \ e_2:T'}$	$\frac{\Gamma \vdash P \rightarrow P' \quad \Gamma \vdash P}{\Gamma \vdash P'}$

Slide 154

Proofs as programs: The Curry-Howard correspondence	
(var) $\Gamma, x:T \vdash x:T$	$\Gamma, P \vdash P$
(fn) $\frac{\Gamma, x:T \vdash e:T'}{\Gamma \vdash \mathbf{fn} \ x:T \Rightarrow e : T \rightarrow T'}$	$\frac{\Gamma, P \vdash P'}{\Gamma \vdash P \rightarrow P'}$
(app) $\frac{\Gamma \vdash e_1:T \rightarrow T' \quad \Gamma \vdash e_2:T}{\Gamma \vdash e_1 \ e_2:T'}$	$\frac{\Gamma \vdash P \rightarrow P' \quad \Gamma \vdash P}{\Gamma \vdash P'}$
(pair) $\frac{\Gamma \vdash e_1:T_1 \quad \Gamma \vdash e_2:T_2}{\Gamma \vdash (e_1, e_2):T_1 * T_2}$	$\frac{\Gamma \vdash P_1 \quad \Gamma \vdash P_2}{\Gamma \vdash P_1 \wedge P_2}$
(proj1) $\frac{\Gamma \vdash e:T_1 * T_2}{\Gamma \vdash \#1 \ e:T_1}$ (proj2) $\frac{\Gamma \vdash e:T_1 * T_2}{\Gamma \vdash \#2 \ e:T_2}$	$\frac{\Gamma \vdash P_1 \wedge P_2}{\Gamma \vdash P_1} \quad \frac{\Gamma \vdash P_1 \wedge P_2}{\Gamma \vdash P_2}$
(inl) $\frac{\Gamma \vdash e:T_1}{\Gamma \vdash \mathbf{inl} \ e:T_1 + T_2:T_1 + T_2}$	$\frac{\Gamma \vdash P_1}{\Gamma \vdash P_1 \vee P_2}$
(inr), (case), (unit), (zero), etc... – but not (letrec)	

The typing rules for a pure language correspond to the rules for a natural deduction calculus.

## 5.2 Datatypes and Records

Slide 155

ML Datatypes	
Datatypes in ML generalize both sums and products, in a sense	
<pre>datatype IntList = Null of unit                   Cons of Int * IntList</pre>	
is (roughly!) like saying	
<pre>IntList = unit + (Int * IntList)</pre>	

In L3 you cannot define `IntList`. It involves recursion at the type level (e.g. types for binary trees). Making this precise is beyond the scope of this course.

Slide 156

**Records**

A generalization of products.

Take field labels  
 Labels  $lab \in \mathbb{L}AB$  for a set  $\mathbb{L}AB = \{p, q, \dots\}$

$$T ::= \dots \mid \{lab_1:T_1, \dots, lab_k:T_k\}$$

$$e ::= \dots \mid \{lab_1 = e_1, \dots, lab_k = e_k\} \mid \#lab e$$

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

Note:

- Labels are not the same syntactic class as variables, so  $(\mathbf{fn} x:T \Rightarrow \{x = 3\})$  is not an expression.
- In ML a pair  $(\mathbf{true}, \mathbf{fn} x:\mathbf{int} \Rightarrow x)$  is syntactic sugar for a record  $\{1 = \mathbf{true}, 2 = \mathbf{fn} x:\mathbf{int} \Rightarrow x\}$ .
- Note that  $\#lab e$  is not an application, it just looks like one in the concrete syntax.
- Again we will choose a left-to-right evaluation order for consistency.

Slide 157

**Records – typing**

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

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

- Here the field order matters, so  $(\mathbf{fn} x:\{\ell_1:\mathbf{int}, \ell_2:\mathbf{bool}\} \Rightarrow x)\{\ell_2 = \mathbf{true}, \ell_1 = 17\}$  does not typecheck.
- Here you can reuse labels, so  $\{\} \vdash (\{\ell_1 = 17\}, \{\ell_1 = \mathbf{true}\}):\{\ell_1:\mathbf{int}\} * \{\ell_1:\mathbf{bool}\}$  is legal, but in some languages (e.g. OCaml) you can't.

Slide 158

**Records – reduction**

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

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

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

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



## 5.3 Mutable Store

Slide 159

**Mutable Store**

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

1 What we've got in L1 and L2:

$$e ::= \dots \mid \ell := e \mid !\ell \mid x$$

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

```
fn x:int => l := (!l) + x
```

Slide 160

2 In C and Java,

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

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

- implicit dereferencing, `l = l + x`
- have some limited type machinery to limit mutability.

– pros and cons: ....

We are staying with option 1 here. But we will now overcome some limitations of references in L1/L2:

- can only store `ints` – we would like to store any value
- cannot create new locations (all must exist at beginning)
- cannot write functions that abstract on locations `fn l:intref =>!l`

Slide 161

**References**

$$T ::= \dots \mid T \text{ ref}$$

$$T_{loc} ::= \text{intref } T \text{ ref}$$

$$e ::= \dots \mid \ell := e \mid !\ell$$

$$\mid e_1 := e_2 \mid !e \mid \text{ref } e \mid \ell$$

We are now allowing variables of `T ref` type, e.g. `fn x:int ref =>!x`. Whole programs should now have no locations at the start. They should create new locations with `ref`.

**References – Typing**

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

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

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

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

Slide 162

**References – Reduction**

A location is a value:

$$v ::= \dots \mid \ell$$

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

Slide 163

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

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

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

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

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

Slide 164

- A `ref` has to do something at runtime – (`ref 0`, `ref 0`) should return a pair of two new locations, each containing `0`, not a pair of one location repeated.
- Note the typing and this dynamics permit locations to contain locations, e.g. `ref( ref 3)`.
- This semantics no longer has determinacy, for a technical reason – new locations are chosen arbitrarily. At the cost of some slight semantic complexity, we could regain determinacy by working ‘up to alpha for locations’.
- Within the language you cannot do arithmetic on locations (can in C, can’t in Java) or test whether one is bigger than another. In L3 you cannot even test locations for equality (in ML you can).
- This store just grows during computation – an implementation can garbage collect.

We *don't* have an explicit deallocation operation – if you do, you need a very baroque type system to prevent dangling pointers being dereferenced.

**Type-checking the store**

Slide 165

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

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

**Type-checking the store – Example**

Slide 166

Consider

```
e = let val x:(int → int) ref = ref(fn z:int ⇒ z) in
      (x := (fn z:int ⇒ if z ≥ 1 then z + (!!x) (z + -1) else 0));
      (!!x) 3 end
```

which has reductions

```
⟨e, {}⟩ →*
⟨e1, {l1 ↦ (fn z:int ⇒ z)}⟩ →*
⟨e2, {l1 ↦ (fn z:int ⇒ if z ≥ 1 then z + (!!l1) (z + -1) else 0)}⟩
→* ⟨6, ...⟩
```

For reference,  $e_1$  and  $e_2$  are

$$e_1 = l_1 := (\text{fn } z:\text{int} \Rightarrow \text{if } z \geq 1 \text{ then } z + (!!l_1) (z + -1) \text{ else } 0);$$

$$e_2 = \text{skip}; (!!l_1) 3$$

Have made a recursive function by ‘tying the knot by hand’, not using **let val rec**. To do this we needed to store function values. We couldn’t do this in L2, so this doesn’t contradict the normalization theorem we had there.

**Definition 19 (Well-typed store)** Let  $\Gamma \vdash s$  if  $\text{dom}(\Gamma) = \text{dom}(s)$  and if for all  $\ell \in \text{dom}(s)$ , if  $\Gamma(\ell) = T \text{ ref}$  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  $\langle e, s \rangle \longrightarrow \langle e', s' \rangle$ .

Slide 167

**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  $\Gamma'$  with disjoint domain to  $\Gamma$  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**

Slide 168

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.

## 5.4 Evaluation Contexts

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

Slide 169

**Evaluation Contexts**

Define *evaluation contexts*

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

Slide 170

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  $\geq 1$  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.*

Slide 171

**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,.....

## 5.5 L3: Collected definition

### L3 syntax

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

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

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

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

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

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

Types:

$$T ::= \text{int} \mid \text{bool} \mid \text{unit} \mid T_1 \rightarrow T_2 \mid T_1 * T_2 \mid T_1 + T_2 \mid \{lab_1:T_1, \dots, lab_k:T_k\} \mid T \text{ ref}$$

Expressions

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

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

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

### L3 semantics

Stores  $s$  are finite partial maps from  $\mathbb{L}$  to the set of all values.

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

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

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

$$(\text{op1}) \quad \frac{\langle e_1, s \rangle \longrightarrow \langle e'_1, s' \rangle}{\langle e_1 \text{ op } e_2, s \rangle \longrightarrow \langle e'_1 \text{ op } e_2, s' \rangle}$$

$$(\text{op2}) \quad \frac{\langle e_2, s \rangle \longrightarrow \langle e'_2, s' \rangle}{\langle v \text{ op } e_2, s \rangle \longrightarrow \langle v \text{ op } e'_2, s' \rangle}$$

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

$$(\text{seq2}) \quad \frac{\langle e_1, s \rangle \longrightarrow \langle e'_1, s' \rangle}{\langle e_1; e_2, s \rangle \longrightarrow \langle e'_1; e_2, s' \rangle}$$

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

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

$$(if3) \quad \frac{\langle e_1, s \rangle \longrightarrow \langle e'_1, s' \rangle}{\langle \mathbf{if\ } e_1 \ \mathbf{then\ } e_2 \ \mathbf{else\ } e_3, s \rangle \longrightarrow \langle \mathbf{if\ } e'_1 \ \mathbf{then\ } e_2 \ \mathbf{else\ } e_3, s' \rangle}$$

(while)

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

$$(app1) \quad \frac{\langle e_1, s \rangle \longrightarrow \langle e'_1, s' \rangle}{\langle e_1 \ e_2, s \rangle \longrightarrow \langle e'_1 \ e_2, s' \rangle}$$

$$(app2) \quad \frac{\langle e_2, s \rangle \longrightarrow \langle e'_2, s' \rangle}{\langle v \ e_2, s \rangle \longrightarrow \langle v \ e'_2, s' \rangle}$$

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

(let1)

$$\frac{\langle e_1, s \rangle \longrightarrow \langle e'_1, s' \rangle}{\langle \mathbf{let\ val\ } x:T = e_1 \ \mathbf{in\ } e_2 \ \mathbf{end}, s \rangle \longrightarrow \langle \mathbf{let\ val\ } x:T = e'_1 \ \mathbf{in\ } e_2 \ \mathbf{end}, s' \rangle}$$

(let2)

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

(letrecfn)  $\mathbf{let\ val\ rec\ } x:T_1 \rightarrow T_2 = (\mathbf{fn\ } y:T_1 \Rightarrow e_1) \ \mathbf{in\ } e_2 \ \mathbf{end}$

$$\longrightarrow \langle \mathbf{fn\ } y:T_1 \Rightarrow \mathbf{let\ val\ rec\ } x:T_1 \rightarrow T_2 = (\mathbf{fn\ } y:T_1 \Rightarrow e_1) \ \mathbf{in\ } e_1 \ \mathbf{end} / x \rangle e_2$$

$$(pair1) \quad \frac{\langle e_1, s \rangle \longrightarrow \langle e'_1, s' \rangle}{\langle (e_1, e_2), s \rangle \longrightarrow \langle (e'_1, e_2), s' \rangle}$$

$$(pair2) \quad \frac{\langle e_2, s \rangle \longrightarrow \langle e'_2, s' \rangle}{\langle (v_1, e_2), s \rangle \longrightarrow \langle (v_1, e'_2), s' \rangle}$$

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

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

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

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

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

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

$$\begin{array}{c}
(\text{inr}) \quad \frac{\langle e, s \rangle \longrightarrow \langle e', s' \rangle}{\langle \mathbf{inr} \ e: T, s \rangle \longrightarrow \langle \mathbf{inr} \ e': T, s' \rangle} \\
\\
(\text{case3}) \quad \langle \mathbf{case} \ \mathbf{inr} \ v: T \ \mathbf{of} \ \mathbf{inl} \ (x: T_1) \Rightarrow e_1 \mid \mathbf{inr} \ (y: T_2) \Rightarrow e_2, s \rangle \\
\longrightarrow \langle \{v/y\}e_2, s \rangle \\
\\
(\text{record1}) \quad \frac{\langle e_i, s \rangle \longrightarrow \langle e'_i, s' \rangle}{\langle \{lab_1 = v_1, \dots, lab_i = e_i, \dots, lab_k = e_k\}, s \rangle \\
\longrightarrow \langle \{lab_1 = v_1, \dots, lab_i = e'_i, \dots, lab_k = e_k\}, s' \rangle} \\
\\
(\text{record2}) \quad \langle \#lab_i \ \{lab_1 = v_1, \dots, lab_k = v_k\}, s \rangle \longrightarrow \langle v_i, s \rangle \\
\\
(\text{record3}) \quad \frac{\langle e, s \rangle \longrightarrow \langle e', s' \rangle}{\langle \#lab_i \ e, s \rangle \longrightarrow \langle \#lab_i \ e', s' \rangle} \\
\\
(\text{ref1}) \quad \langle \mathbf{ref} \ v, s \rangle \longrightarrow \langle \ell, s + \{\ell \mapsto v\} \rangle \quad \ell \notin \text{dom}(s) \\
\\
(\text{ref2}) \quad \frac{\langle e, s \rangle \longrightarrow \langle e', s' \rangle}{\langle \mathbf{ref} \ e, s \rangle \longrightarrow \langle \mathbf{ref} \ e', s' \rangle} \\
\\
(\text{deref1}) \quad \langle !\ell, s \rangle \longrightarrow \langle v, s \rangle \quad \text{if } \ell \in \text{dom}(s) \text{ and } s(\ell) = v \\
\\
(\text{deref2}) \quad \frac{\langle e, s \rangle \longrightarrow \langle e', s' \rangle}{\langle !e, s \rangle \longrightarrow \langle !e', s' \rangle} \\
\\
(\text{assign1}) \quad \langle \ell := v, s \rangle \longrightarrow \langle \mathbf{skip}, s + \{\ell \mapsto v\} \rangle \quad \text{if } \ell \in \text{dom}(s) \\
\\
(\text{assign2}) \quad \frac{\langle e, s \rangle \longrightarrow \langle e', s' \rangle}{\langle \ell := e, s \rangle \longrightarrow \langle \ell := e', s' \rangle} \\
\\
(\text{assign3}) \quad \frac{\langle e, s \rangle \longrightarrow \langle e', s' \rangle}{\langle e := e_2, s \rangle \longrightarrow \langle e' := e_2, s' \rangle}
\end{array}$$

### L3 Typing

Type environments,  $\Gamma \in \text{TypeEnv2}$ , are finite partial functions from  $\mathbb{L} \cup \mathbb{X}$  to  $\text{T}_{\text{loc}} \cup \text{T}$  such that

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

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

$$\begin{array}{c}
(\text{int}) \quad \Gamma \vdash n: \text{int} \quad \text{for } n \in \mathbb{Z} \\
\\
(\text{bool}) \quad \Gamma \vdash b: \text{bool} \quad \text{for } b \in \{\mathbf{true}, \mathbf{false}\} \\
\\
(\text{op } +) \quad \frac{\Gamma \vdash e_1: \text{int} \quad \Gamma \vdash e_2: \text{int}}{\Gamma \vdash e_1 + e_2: \text{int}} \quad (\text{op } \geq) \quad \frac{\Gamma \vdash e_1: \text{int} \quad \Gamma \vdash e_2: \text{int}}{\Gamma \vdash e_1 \geq e_2: \text{bool}} \\
\\
(\text{if}) \quad \frac{\Gamma \vdash e_1: \text{bool} \quad \Gamma \vdash e_2: T \quad \Gamma \vdash e_3: T}{\Gamma \vdash \mathbf{if} \ e_1 \ \mathbf{then} \ e_2 \ \mathbf{else} \ e_3: T}
\end{array}$$

$$\begin{array}{c}
\text{(skip)} \quad \Gamma \vdash \mathbf{skip}:\mathbf{unit} \\
\\
\text{(seq)} \quad \frac{\Gamma \vdash e_1:\mathbf{unit} \quad \Gamma \vdash e_2:T}{\Gamma \vdash e_1; e_2:T} \\
\\
\text{(while)} \quad \frac{\Gamma \vdash e_1:\mathbf{bool} \quad \Gamma \vdash e_2:\mathbf{unit}}{\Gamma \vdash \mathbf{while} \ e_1 \ \mathbf{do} \ e_2:\mathbf{unit}} \\
\\
\text{(var)} \quad \Gamma \vdash x:T \quad \text{if } \Gamma(x) = T \\
\\
\text{(fn)} \quad \frac{\Gamma, x:T \vdash e:T'}{\Gamma \vdash \mathbf{fn} \ x:T \Rightarrow e : T \rightarrow T'} \\
\\
\text{(app)} \quad \frac{\Gamma \vdash e_1:T \rightarrow T' \quad \Gamma \vdash e_2:T}{\Gamma \vdash e_1 \ e_2:T'} \\
\\
\text{(let)} \quad \frac{\Gamma \vdash e_1:T \quad \Gamma, x:T \vdash e_2:T'}{\Gamma \vdash \mathbf{let} \ \mathbf{val} \ x:T = e_1 \ \mathbf{in} \ e_2 \ \mathbf{end}:T'} \\
\\
\text{(let rec fn)} \quad \frac{\Gamma, x:T_1 \rightarrow T_2, y:T_1 \vdash e_1:T_2 \quad \Gamma, x:T_1 \rightarrow T_2 \vdash e_2:T}{\Gamma \vdash \mathbf{let} \ \mathbf{val} \ \mathbf{rec} \ x:T_1 \rightarrow T_2 = (\mathbf{fn} \ y:T_1 \Rightarrow e_1) \ \mathbf{in} \ e_2 \ \mathbf{end}:T} \\
\\
\text{(pair)} \quad \frac{\Gamma \vdash e_1:T_1 \quad \Gamma \vdash e_2:T_2}{\Gamma \vdash (e_1, e_2):T_1 * T_2} \\
\\
\text{(proj1)} \quad \frac{\Gamma \vdash e:T_1 * T_2}{\Gamma \vdash \#1 \ e:T_1} \\
\\
\text{(proj2)} \quad \frac{\Gamma \vdash e:T_1 * T_2}{\Gamma \vdash \#2 \ e:T_2} \\
\\
\text{(inl)} \quad \frac{\Gamma \vdash e:T_1}{\Gamma \vdash \mathbf{inl} \ e:T_1 + T_2:T_1 + T_2} \\
\\
\text{(inr)} \quad \frac{\Gamma \vdash e:T_2}{\Gamma \vdash \mathbf{inr} \ e:T_1 + T_2:T_1 + T_2} \\
\\
\text{(case)} \quad \frac{\Gamma \vdash e:T_1 + T_2 \quad \Gamma, x:T_1 \vdash e_1:T \quad \Gamma, y:T_2 \vdash e_2:T}{\Gamma \vdash \mathbf{case} \ e \ \mathbf{of} \ \mathbf{inl} \ (x:T_1) \Rightarrow e_1 \ | \ \mathbf{inr} \ (y:T_2) \Rightarrow e_2:T} \\
\\
\text{(record)} \quad \frac{\Gamma \vdash e_1:T_1 \quad \dots \quad \Gamma \vdash e_k:T_k}{\Gamma \vdash \{lab_1 = e_1, \dots, lab_k = e_k\}:\{lab_1:T_1, \dots, lab_k:T_k\}} \\
\\
\text{(recordproj)} \quad \frac{\Gamma \vdash e:\{lab_1:T_1, \dots, lab_k:T_k\}}{\Gamma \vdash \#lab_i \ e:T_i}
\end{array}$$



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

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

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

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

## 5.6 Exercises

**Exercise 27** ★★ *Prove Theorem 14: Type Preservation for L3.*

**Exercise 28** ★★ *Labelled variant types are a generalization of sum types, just as records are a generalization of products. Design abstract syntax, type rules and evaluation rules for labelled variants, analogously to the way in which records generalise products.*

**Exercise 29** ★★ *Design type rules and evaluation rules for ML-style exceptions. Start with exceptions that do not carry any values. Hint 1: take care with nested handlers within recursive functions. Hint 2: you might want to express your semantics using evaluation contexts.*

**Exercise 30** ★★★ *Extend the L2 implementation to cover all of L3.*

## 6 Subtyping and Objects

Slide 172

### Subtyping and Objects

Our type systems so far would all be annoying to use, as they're quite rigid (Pascal-like). There is little support for code reuse, so you would have to have different sorting code for, e.g., `int` lists and `int * int` lists.

Slide 173

#### 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  $\alpha$  takes an argument of type  $\alpha$  list and computes its length (parametric – uniform in whatever  $\alpha$  is)
- Subtype polymorphism – as in various OO languages. See here.  
Dating back to the 1960s (Simula etc); formalized in 1980, 1984, ...

Slide 174

#### Subtyping – Motivation

Recall

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

so can't type

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

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

### Subsumption

'Better'? Any value of type  $\{p:\text{int}, q:\text{int}\}$  can be used wherever a value of type  $\{p:\text{int}\}$  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

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

Introduce a *subsumption rule*

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

allowing subtyping to be used, capturing (\*).

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

Slide 175

### Example

$$\frac{\frac{\frac{\overline{x:\{p:\text{int}\}} \vdash x:\{p:\text{int}\}}{x:\{p:\text{int}\} \vdash \#p x:\text{int}} \text{(record-proj)}}{\{\} \vdash (\text{fn } x:\{p:\text{int}\} \Rightarrow \#p x):\{p:\text{int}\} \rightarrow \text{int}} \text{(fn)}}{\{\} \vdash (\text{fn } x:\{p:\text{int}\} \Rightarrow \#p x)\{p = 3, q = 4\}:\text{int}} \text{(app)}}{\frac{\frac{\overline{\{\}} \vdash 3:\text{int}} \quad \overline{\{\}} \vdash 4:\text{int}}{\{\} \vdash \{p = 3, q = 4\}:\{p:\text{int}, q:\text{int}\}} \text{(record)}}{\{\} \vdash \{p = 3, q = 4\}:\{p:\text{int}\}} \text{(★)}} \text{(sub)}} \text{(sub)}$$

where (★) is  $\{p:\text{int}, q:\text{int}\} <: \{p:\text{int}\}$

Slide 176

Now, we define the subtyping relation.

### The Subtype Relation $T <: T'$

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

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

Slide 177

### Subtyping – Records

Forgetting fields on the right:

$$\frac{\{lab_1:T_1, \dots, lab_k:T_k, lab_{k+1}:T_{k+1}, \dots, lab_{k+k'}:T_{k+k'}\}}{\{lab_1:T_1, \dots, lab_k:T_k\}} <: \text{(s-record-width)}$$

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

Allowing subtyping within fields:

$$\text{(s-record-depth)} \quad \frac{T_1 <: T'_1 \quad \dots \quad T_k <: T'_k}{\{lab_1:T_1, \dots, lab_k:T_k\} <: \{lab_1:T'_1, \dots, lab_k:T'_k\}}$$

Combining these:

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

Slide 178

Another example:

$$\frac{\frac{\overline{\{x:\{p:\text{int}, q:\text{int}\}, y:\{r:\text{int}\}\} <: \{x:\{p:\text{int}, q:\text{int}\}\}} \text{ (s-rec-w)} \quad \frac{\overline{\{p:\text{int}, q:\text{int}\} <: \{p:\text{int}\}} \text{ (s-rec-d)}}{\overline{\{x:\{p:\text{int}, q:\text{int}\}\} <: \{x:\{p:\text{int}\}\}} \text{ (s-trans)}}}{\overline{\{x:\{p:\text{int}, q:\text{int}\}, y:\{r:\text{int}\}\} <: \{x:\{p:\text{int}\}\}}}$$

Slide 179

Allowing reordering of fields:

(s-record-order)

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

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

Slide 180

### Subtyping – Functions

$$\text{(s-fn)} \quad \frac{T'_1 <: T_1 \quad T_2 <: T'_2}{T_1 \rightarrow T_2 <: T'_1 \rightarrow T'_2}$$

*contravariant* on the left of  $\rightarrow$

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

Slide 181

If  $f:T_1 \rightarrow T_2$  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

$$f = \mathbf{fn} \ x:\{p:\text{int}\} \Rightarrow \{p = \#p \ x, q = 28\}$$

we have

$$\{\} \vdash f:\{p:\text{int}\} \rightarrow \{p:\text{int}, q:\text{int}\}$$

$$\{\} \vdash f:\{p:\text{int}\} \rightarrow \{p:\text{int}\}$$

$$\{\} \vdash f:\{p:\text{int}, q:\text{int}\} \rightarrow \{p:\text{int}, q:\text{int}\}$$

$$\{\} \vdash f:\{p:\text{int}, q:\text{int}\} \rightarrow \{p:\text{int}\}$$

as

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

Slide 182

On the other hand, for

$$\mathbf{fn} \ x:\{p:\text{int}, q:\text{int}\} \Rightarrow \{p = (\#p \ x) + (\#q \ x)\}$$

we have

$$\{\} \vdash f:\{p:\text{int}, q:\text{int}\} \rightarrow \{p:\text{int}\}$$

$$\{\} \not\vdash f:\{p:\text{int}\} \rightarrow T \text{ for any } T$$

$$\{\} \not\vdash f:T \rightarrow \{p:\text{int}, q:\text{int}\} \text{ for any } T$$

**Subtyping – Products**

Just like (s-record-depth)

(s-pair)  $\frac{T_1 <: T'_1 \quad T_2 <: T'_2}{T_1 * T_2 <: T'_1 * T'_2}$

**Subtyping – Sums**

Exercise.

**Subtyping – References**

Are either of these any good?

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

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

$e ::= \dots \mid (T)e$

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)

The following development is taken from [Pierce, Chapter 18], where you can find more details (including a treatment of self and a direct semantics for a 'featherweight' fragment of Java).

Slide 187

### (Very Simple) Objects

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

*Counter* = {get:unit → int, inc:unit → unit}.

Slide 188

### 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* = {get:unit → int, inc:unit → unit, reset:unit → unit}

<: *Counter* = {get:unit → int, inc:unit → unit}.

Slide 189

### 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...

Slide 190

### Reusing Method Code (Simple Classes)

Recall `Counter = {get:unit → int, inc:unit → unit}`.

First, make the internal state into a record.

`CounterRep = {p:int ref}`.

```
let val counterClass:CounterRep → Counter =  
  fn x:CounterRep ⇒  
    {get = fn y:unit ⇒!(#p x),  
     inc = fn y:unit ⇒ (#p x) := 1+!(#p x)}
```

```
let val newCounter:unit → Counter =  
  fn y:unit ⇒  
    let val x:CounterRep = {p = ref 0} in  
      counterClass x
```

Slide 191

### Reusing Method Code (Simple Classes)

```
let val resetCounterClass:CounterRep → ResetCounter =
```

```
  fn x:CounterRep ⇒  
    let val super = counterClass x in  
      {get = #get super,  
       inc = #inc super,  
       reset = fn y:unit ⇒ (#p x) := 0}
```

`CounterRep = {p:int ref}`.

`Counter = {get:unit → int, inc:unit → unit}`.

`ResetCounter = {get:unit → int, inc:unit → unit, reset:unit → unit}`.

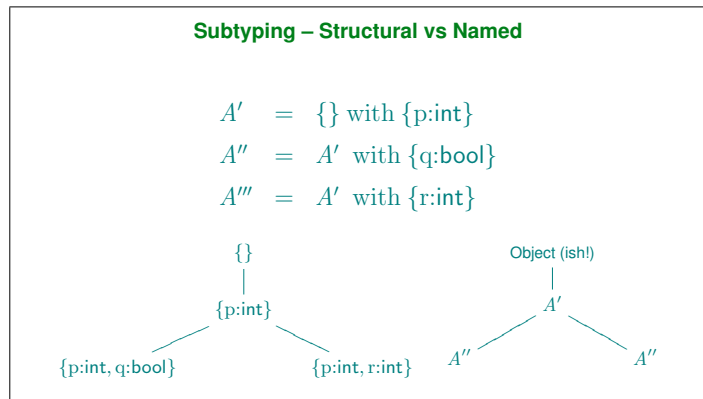
Slide 192

### 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;}  
  };
```

Slide 193



## 6.1 Exercises

**Exercise 31** ★ For each of the following, either give a type derivation or explain why it is untypable.

1.  $\{\} \vdash \{p = \{p = \{p = \{p = 3\}\}\}\} : \{p:\{\}\}$
2.  $\{\} \vdash \mathbf{fn} \ x:\{p:\text{bool}, q:\{p:\text{int}, q:\text{bool}\}\} \Rightarrow \#q \#p \ x : ?$
3.  $\{\} \vdash \mathbf{fn} \ f:\{p:\text{int}\} \rightarrow \text{int} \Rightarrow (f \ \{q = 3\}) + (f \ \{p = 4\}) : ?$
4.  $\{\} \vdash \mathbf{fn} \ f:\{p:\text{int}\} \rightarrow \text{int} \Rightarrow (f \ \{q = 3, p = 2\}) + (f \ \{p = 4\}) : ?$

**Exercise 32** ★ For each of the two bogus  $T$  ref subtype rules on Slide 184, give an example program that is typable with that rule but gets stuck at runtime.

**Exercise 33** ★★ What should the subtype rules for sums  $T + T'$  be?

**Exercise 34** ★★ ...and for **let** and **let rec** ?

**Exercise 35** ★★ Prove a Progress Theorem for  $L3$  with subtyping.



## 7 Semantic Equivalence

Slide 194

### Semantic Equivalence

Slide 195

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

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....

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

Slide 196

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:

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

$$C[l := 0; 4] = (l := 0; 4) + !l$$

$$\neq$$

$$C[l := 1; 3+!l] = (l := 1; 3+!l) + !l$$

On the other hand, consider

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

Slide 197

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

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

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

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

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

Slide 198

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

$$\stackrel{?}{\simeq}$$

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

Slide 199

Temporarily extend L3 with pointer equality

$op ::= \dots \mid =$

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

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

Slide 200

$$f = \text{let val } x = \text{ref } 0 \text{ in} \\ \text{let val } y = \text{ref } 0 \text{ in} \\ \text{fn } z : \text{int ref} \Rightarrow \text{if } z = x \text{ then } y \text{ else } x \\ \text{end end}$$

$$g = \text{let val } x = \text{ref } 0 \text{ in} \\ \text{let val } y = \text{ref } 0 \text{ in} \\ \text{fn } z : \text{int ref} \Rightarrow \text{if } z = y \text{ then } y \text{ else } x \\ \text{end end}$$

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

The last two examples are taken from A.M. Pitts, Operational Semantics and Program Equivalence. In: G. Barthe, P. Dybjer and J. Saraiva (Eds), Applied Semantics. Lecture Notes in Computer Science, Tutorial, Volume 2395 (Springer-Verlag, 2002), pages 378-412. <http://www.cl.cam.ac.uk/~amp12/papers/opespe/opespe-lncs.pdf>

Slide 201

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'?

- Slide 202
1. programs that result in observably-different values (in some initial store) must not be equivalent  

$$(\exists s, s_1, s_2, v_1, v_2. \langle e_1, s \rangle \longrightarrow^* \langle v_1, s_1 \rangle \wedge \langle e_2, s \rangle \longrightarrow^* \langle v_2, s_2 \rangle \wedge v_1 \neq v_2) \Rightarrow e_1 \not\approx e_2$$
  2. programs that terminate must not be equivalent to programs that don't
  3.  $\simeq$  must be an equivalence relation  

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

- Slide 203
- Consider Typed L1 again.
- Define  $e_1 \simeq_{\Gamma}^T e_2$  to hold iff for all  $s$  such that  $\text{dom}(\Gamma) \subseteq \text{dom}(s)$ , 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  $\langle e_2, s \rangle \longrightarrow^* \langle v, s' \rangle$ .

In this definition, part (b), we require that  $e_1$  and  $e_2$  result in the same value and moreover the same store. This is because, if we were to equate two programs  $e_1$  and  $e_2$  that result in *different* stores — say  $s_1(l) \neq s_2(l)$  — then we could distinguish them using the following contexts, and the semantic equivalence would not be a congruence.

- Slide 204
- If  $T = \text{unit}$  then  $C = \_ ; !l$ .
- If  $T = \text{bool}$  then  $C = \text{if } \_ \text{ then } !l \text{ else } !l$ .
- If  $T = \text{int}$  then  $C = l_1 := \_ ; !l$ .

### Congruence for Typed L1

- Slide 205
- The L1 contexts are:
- $$C ::= \_ \text{ op } e_2 \mid e_1 \text{ op } \_ \mid$$
- $$\text{if } \_ \text{ then } e_2 \text{ else } e_3 \mid \text{if } e_1 \text{ then } \_ \text{ else } e_3 \mid$$
- $$\text{if } e_1 \text{ then } e_2 \text{ else } \_ \mid$$
- $$l := \_ \mid$$
- $$\_ ; e_2 \mid e_1 ; \_ \mid$$
- $$\text{while } \_ \text{ do } e_2 \mid \text{while } e_1 \text{ do } \_$$
- 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  $C[e_1] \simeq_{\Gamma}^{T'} C[e_2]$ .

Slide 206

**Theorem 24 (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

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

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

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

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

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

**Theorem 24 (Congruence for L1)**  $\simeq_{\Gamma}^T$  has the congruence property.

**Proof** By case analysis, looking at each L1 context in turn. We give only one case here, leaving the others for the reader.

**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 we have  $T = \text{int}$  and  $T' = \text{unit}$ .

To show  $\ell := e \simeq_{\Gamma}^T \ell := e'$  we have to show for all  $s$  such that  $\text{dom}(\Gamma) \subseteq \text{dom}(s)$ , then  $\Gamma \vdash \ell := e : T'(\checkmark)$ ,  $\Gamma \vdash \ell := e' : T'(\checkmark)$ , 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  $\langle \ell := e, s \rangle$ . Recall that (by examining the reduction rules), if  $\langle \ell := e, s \rangle \longrightarrow \langle e_1, s_1 \rangle$  then either that is an instance of (assign1), with  $\exists n. e = n \wedge \ell \in \text{dom}(s) \wedge e_1 = \text{skip} \wedge s' = s + \{\ell \mapsto n\}$ , or it is an instance of (assign2), with  $\exists \hat{e}_1. \langle e, s \rangle \longrightarrow \langle \hat{e}_1, s_1 \rangle \wedge e_1 = (\ell := \hat{e}_1)$ . We know also that  $\langle \text{skip}, s \rangle$  does not reduce.

Now (using Determinacy), for any  $e$  and  $s$  we have either

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

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

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

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

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

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

$$\langle \ell := e, s \rangle \longrightarrow \langle e_1, s_1 \rangle \longrightarrow \langle e_2, s_2 \rangle \dots \longrightarrow \langle e_k, s_k \rangle \not\rightarrow$$

hence all these must be instances of (assign2) except the last, which must be an instance of (assign1), with

$$\langle e, s \rangle \longrightarrow \langle \hat{e}_1, s_1 \rangle \longrightarrow \langle \hat{e}_2, s_2 \rangle \longrightarrow \dots \longrightarrow \langle \hat{e}_{k-1}, s_{k-1} \rangle$$

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  $\hat{e}_{k-1} = n$ ,  $e_k = \text{skip}$ , and  $s_k = s_{k-1} + \{\ell \mapsto n\}$ .

(the other possibility, of zero or more (assign1) reductions ending in a stuck state, is excluded by Theorems 2 and 3 (type preservation and progress))

Now, if  $\langle \ell := e, s \rangle \longrightarrow^{\omega}$ , by the above there is an infinite reduction sequence for  $\langle e, s \rangle$ , so by  $e \simeq_{\Gamma}^T e'$  there is an infinite reduction sequence of  $\langle e', s \rangle$ , so (using (assign2)) there is an infinite reduction sequence of  $\langle \ell := e', s \rangle$ .

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  $\langle \ell := e, s \rangle \longrightarrow \langle \mathbf{skip}, s_{k-1} + \{\ell \mapsto n\} \rangle$ . By  $e \simeq_\Gamma^T e'$  we have  $\langle e', s \rangle \longrightarrow^* \langle n, s_{k-1} \rangle$ . Then using (assign1)  $\langle \ell := e', s \rangle \longrightarrow^* \langle \ell := n, s_{k-1} \rangle \longrightarrow \langle \mathbf{skip}, s_{k-1} + \{\ell \mapsto n\} \rangle = \langle e_k, s_k \rangle$  as required.

□

**Back to the Examples**

We defined  $e_1 \simeq_\Gamma^T e_2$  iff for all  $s$  such that  $\text{dom}(\Gamma) \subseteq \text{dom}(s)$ , 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:

$2 + 2 \simeq_\Gamma^{\text{int}} 4$  for any  $\Gamma$

$(l := 0; 4) \not\simeq_\Gamma^{\text{int}} (l := 1; 3+!l)$  for any  $\Gamma$

$(l := !l + 1); (l := !l - 1) \simeq_\Gamma^{\text{unit}} (l := !l)$  for any  $\Gamma$  including  $l:\text{intref}$

Slide 207

And the general laws?

**Conjecture 25**  $e_1; (e_2; e_3) \simeq_\Gamma^T (e_1; e_2); e_3$  for any  $\Gamma, T, e_1, e_2$  and  $e_3$  such that  $\Gamma \vdash e_1:\text{unit}, \Gamma \vdash e_2:\text{unit}$ , and  $\Gamma \vdash e_3:T$


**Conjecture 26**  
 $((\text{if } e_1 \text{ then } e_2 \text{ else } e_3); e) \simeq_\Gamma^T (\text{if } e_1 \text{ then } e_2; e \text{ else } e_3; e)$  for any  $\Gamma, T, e, e_1, e_2$  and  $e_3$  such that  $\Gamma \vdash e_1:\text{bool}, \Gamma \vdash e_2:\text{unit}, \Gamma \vdash e_3:\text{unit}$ , and  $\Gamma \vdash e:T$

**Conjecture 27**  
 $(e; (\text{if } e_1 \text{ then } e_2 \text{ else } e_3)) \simeq_\Gamma^T (\text{if } e_1 \text{ then } e; e_2 \text{ else } e; e_3)$  for any  $\Gamma, T, e, e_1, e_2$  and  $e_3$  such that  $\Gamma \vdash e:\text{unit}, \Gamma \vdash e_1:\text{bool}, \Gamma \vdash e_2:T$ , and  $\Gamma \vdash e_3:T$

Slide 208

Q: Is a typed expression  $\Gamma \vdash e:T$ , e.g.

$l:\text{intref} \vdash \text{if } !l \geq 0 \text{ then skip else } (\mathbf{skip}; l := 0):\text{unit}$ :

1. a list of tokens [ IF, DEREf, LOC "1", GTEQ, ... ];
2. an abstract syntax tree ;
3. the function taking store  $s$  to the reduction sequence  $\langle e, s \rangle \longrightarrow \langle e_1, s_1 \rangle \longrightarrow \langle e_2, s_2 \rangle \longrightarrow \dots$ ; or
4. • the equivalence class  $\{e' \mid e \simeq_\Gamma^T e'\}$   
 • the partial function  $\llbracket e \rrbracket_\Gamma$  that takes any store  $s$  with  $\text{dom}(s) = \text{dom}(\Gamma)$  and either is undefined, if  $\langle e, s \rangle \longrightarrow^\omega$ , or is  $\langle v, s' \rangle$ , if  $\langle e, s \rangle \longrightarrow^* \langle v, s' \rangle$

Slide 209

(the Determinacy theorem tells us that this is a definition of a function).

Slide 210

Suppose  $\Gamma \vdash e_1:\text{unit}$  and  $\Gamma \vdash e_2:\text{unit}$ .

When is  $e_1; e_2 \simeq_\Gamma^{\text{unit}} e_2; e_1$  ?

A sufficient condition: they don't mention any locations (but not necessary... e.g. if  $e_1$  does but  $e_2$  doesn't)

## 7.1 Contextual equivalence

The definition of semantic equivalence works fine for L1. However, when we come to L2 and L3, the simple notion does not give a congruence.

Here is a basic definition of an equivalence for L3.

Slide 211

**Contextual equivalence for L3**

**Definition 28** 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  $\{\} \vdash C[e_1]:\text{unit}$  and  $\{\} \vdash C[e_2]:\text{unit}$ , 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  $\langle C[e_1], \{\} \rangle \longrightarrow^* \langle \text{skip}, s_1 \rangle$  and  $\langle C[e_2], \{\} \rangle \longrightarrow^* \langle \text{skip}, s_2 \rangle$ .

Notice that contextual equivalence is a congruence by definition.

Contextual equivalence is undecidable in general. An important research topic is finding techniques for proving contextual equivalence.

## 7.2 Exercises

**Exercise 36** ★★ Prove some of the other cases of the Congruence theorem for semantic equivalence in L1.

**Exercise 37** ★★ Prove that if  $\Gamma_1 \vdash e_1:\text{unit}$  and  $\Gamma_2 \vdash e_2:\text{unit}$  in L1, and  $\Gamma_1$  is disjoint from  $\Gamma_2$ , then  $e_1; e_2 \simeq_{\Gamma}^{\text{unit}} e_2; e_1$  where  $\Gamma = \Gamma_1 \cup \Gamma_2$

**Exercise 38** ★★ Prove that the programs  $l:\text{int ref} \vdash l := 0:\text{unit}$  and  $l:\text{int ref} \vdash l := 1:\text{unit}$ , considered as L3 programs, are not contextually equivalent. Hint: find a context that will diverge for one of them, but not for the other.

## 8 Concurrency

Slide 212

### Concurrency

Slide 213

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

Slide 214

#### 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.

Slide 215

#### 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 malicious attack.
- dealing with contingent complexity (embedded historical accidents; upwards-compatible deltas)

Slide 216

**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.

Slide 217

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

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

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

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

Expressions

$e ::= n \mid b \mid e_1 \ op \ e_2 \mid \text{if } e_1 \ \text{then } e_2 \ \text{else } e_3 \mid$

$\ell := e \mid !\ell \mid$

**skip**  $\mid e_1; e_2 \mid$

**while**  $e_1 \ \text{do } e_2 \mid$

$e_1 \mid e_2$

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

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

Slide 218

### Parallel Composition: Typing and Reduction

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

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

(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)  $\frac{\langle e_2, s \rangle \longrightarrow \langle e'_2, s' \rangle}{\langle e_1 \mid e_2, s \rangle \longrightarrow \langle e_1 \mid e'_2, s' \rangle}$

Slide 219

### Parallel Composition: Design Choices

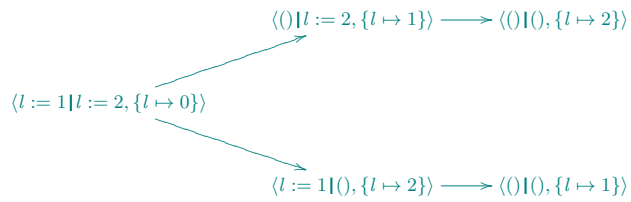
- threads don't return a value
- threads don't have an identity
- termination of a thread cannot be observed within the language
- threads aren't partitioned into 'processes' or machines
- threads can't be killed externally



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

All threads can read and write the shared memory.

Slide 220



NB from here on, we are using  $()$  instead of **skip** — that's the ML syntax.

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

$\langle l := 3498734590879238429384 \mid l := 7, \{l \mapsto 0\} \rangle$

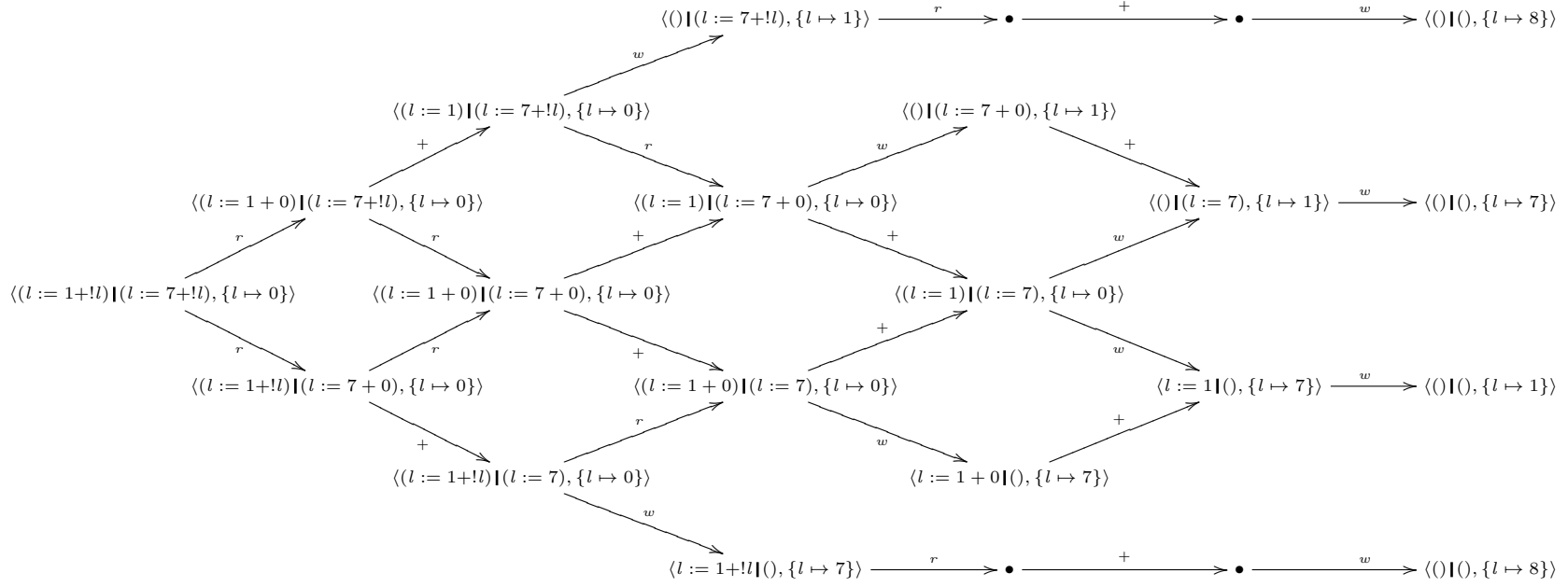
will reduce to a state with  $l$  either 3498734590879238429384 or 7, not something with the first word of one and the second word of the other.

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....

Slide 221



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

Note that the labels  $+$ ,  $w$  and  $r$  in the picture are just informal hints as to how those transitions were derived – they are not actually part of the reduction relation.

Some of the nondeterministic choices “don’t matter”, as you can get back to the same state. Others do...

Slide 222

**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.

Slide 223

**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`).

See this – in the library – for a good discussion of mutexes and condition variables: A. Birrell, J. Guttag, J. Horning, and R. Levin. *Thread synchronization: a Formal Specification*. In G. Nelson, editor, System Programming with Modula-3, chapter 5, pages 119-129. Prentice-Hall, 1991.

See N. Lynch. *Distributed Algorithms* for other mutual exclusion algorithms (and much else besides).

Consider simple mutexes, with commands to lock an unlocked mutex and to unlock a locked mutex (and do nothing for an unlock of an unlocked mutex).

Slide 224

**Adding Primitive Mutexes**

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

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

Expressions  $e ::= \dots \mid \text{lock } m \mid \text{unlock } m$

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

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

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

Note that (lock) *atomically* (a) checks the mutex is currently false, (b) changes its state, and (c) lets the thread proceed.

Also, there is no record of which thread is holding a locked mutex.

Slide 225

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

$$(op2) \frac{\langle e_2, s \rangle \longrightarrow \langle e'_2, s' \rangle}{\langle v \text{ op } e_2, s \rangle \longrightarrow \langle v \text{ op } e'_2, s' \rangle}$$

by

$$(op2) \frac{\langle e_2, s, M \rangle \longrightarrow \langle e'_2, s', M' \rangle}{\langle v \text{ op } e_2, s, M \rangle \longrightarrow \langle v \text{ op } e'_2, s', M' \rangle}$$

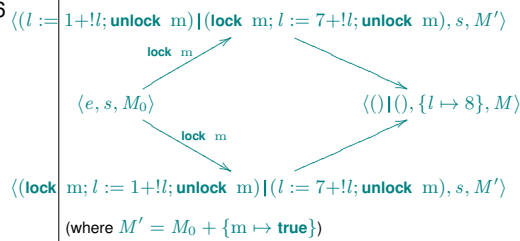
Slide 226

### Using a Mutex

Consider

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

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



In all the intervening states (until the first **unlock** ) the second **lock** can't proceed.

Look back to behaviour of the program without mutexes. We've essentially cut down to the top and bottom paths (and also added some extra reductions for **lock** , **unlock** , and ;).

In this example,  $l := 1+!l$  and  $l := 7+!l$  commute, so we end up in the same final state whichever got the lock first. In general, that won't be the case.

Slide 227

### Using Several Mutexes

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

$e = (\text{lock } m_1; \text{lock } m_2; l_1 := !l_2; \text{unlock } m_1; \text{unlock } m_2) | (\text{lock } m_2; \text{lock } m_1; l_2 := !l_1; \text{unlock } m_1; \text{unlock } m_2)$

Slide 228

### Locking Disciplines

So, suppose we have several programs  $e_1, \dots, e_k$ , all well-typed with  $\Gamma \vdash e_i : \text{unit}$ , 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.

Slide 229

### 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  $m, m_0, m_1, m_2, \dots$

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

These are *semantic properties* again. In general, it won't be computable whether they hold. For simple  $e_i$ , though, it's often obvious. Further, one can construct syntactic disciplines that are checkable and are sufficient to guarantee these.

Slide 230

### 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  $\langle (e_1 | \dots | e_i | \dots | e_k), s, M \rangle$ ,  $e_i$  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

Slide 231

### Solution: Thread local semantics

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

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

$$\text{(parallel1)} \quad \frac{\langle e_1, s, M \rangle \longrightarrow \langle e'_1, s', M' \rangle}{\langle e_1 | e_2, s, M \rangle \longrightarrow \langle e'_1 | e_2, s', M' \rangle}$$

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

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

$$\text{(t-parallel1)} \quad \frac{e_1 \xrightarrow{a} e'_1}{e_1 | e_2 \xrightarrow{a} e'_1 | e_2}$$

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

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.

$$a ::= \tau \mid \ell := n \mid !\ell = n \mid \mathbf{lock} \ m \mid \mathbf{unlock} \ m$$

Slide 232

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

**Theorem 29 (Coincidence of global and thread-local semantics)** *The two definitions of  $\longrightarrow$  agree exactly.*

**Proof strategy:** a couple of rule inductions.

The full thread local semantics are on the next page.

### Example of Thread-local transitions

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

$$\begin{array}{l} e \xrightarrow{\mathbf{lock} \ m} \mathbf{skip}; (l := 1 + !l; \mathbf{unlock} \ m) \\ \xrightarrow{\tau} (l := 1 + !l; \mathbf{unlock} \ m) \\ \xrightarrow{!\ell = n} (l := 1 + n; \mathbf{unlock} \ m) \quad \text{for any } n \in \mathbb{Z} \\ \xrightarrow{\tau} (l := n'; \mathbf{unlock} \ m) \quad \text{for } n' = 1 + n \\ \xrightarrow{l := n'} \mathbf{skip}; \mathbf{unlock} \ m \\ \xrightarrow{\tau} \mathbf{unlock} \ m \\ \xrightarrow{\mathbf{unlock} \ m} \mathbf{skip} \end{array}$$

Slide 233

Hence, using (t-parallel) and the (c-\*) rules, for  $s' = s + \{l \mapsto 1 + s(l)\}$ ,  $\langle e \mid e', s, M_0 \rangle \longrightarrow \langle \mathbf{skip} \mid e', s', M_0 \rangle$

(need  $l \in \text{dom}(s)$  also)

One often uses similar labelled transitions in defining *communication* between threads (or machines), and also in working with observational equivalences for concurrent languages (cf. *bisimulation*) – to come in *Topics in Concurrency*.

### Global Semantics

$$\begin{aligned}
(\text{op } +) \quad & \langle n_1 + n_2, s, M \rangle \longrightarrow \langle n, s, M \rangle \quad \text{if } n = n_1 + n_2 \\
(\text{op } \geq) \quad & \langle n_1 \geq n_2, s, M \rangle \longrightarrow \langle b, s, M \rangle \quad \text{if } b = (n_1 \geq n_2) \\
(\text{op1}) \quad & \frac{\langle e_1, s, M \rangle \longrightarrow \langle e'_1, s', M' \rangle}{\langle e_1 \text{ op } e_2, s, M \rangle \longrightarrow \langle e'_1 \text{ op } e_2, s', M' \rangle} \\
(\text{op2}) \quad & \frac{\langle e_2, s, M \rangle \longrightarrow \langle e'_2, s', M' \rangle}{\langle v \text{ op } e_2, s, M \rangle \longrightarrow \langle v \text{ op } e'_2, s', M' \rangle} \\
(\text{deref}) \quad & \langle !\ell, s, M \rangle \longrightarrow \langle n, s, M \rangle \quad \text{if } \ell \in \text{dom}(s) \text{ and } s(\ell) = n \\
(\text{assign1}) \quad & \langle \ell := n, s, M \rangle \longrightarrow \langle \text{skip}, s + \{\ell \mapsto n\}, M \rangle \quad \text{if } \ell \in \text{dom}(s) \\
(\text{assign2}) \quad & \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} \\
(\text{seq1}) \quad & \langle \text{skip}; e_2, s, M \rangle \longrightarrow \langle e_2, s, M \rangle \\
(\text{seq2}) \quad & \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} \\
(\text{if1}) \quad & \langle \text{if true then } e_2 \text{ else } e_3, s, M \rangle \longrightarrow \langle e_2, s, M \rangle \\
(\text{if2}) \quad & \langle \text{if false then } e_2 \text{ else } e_3, s, M \rangle \longrightarrow \langle e_3, s, M \rangle \\
(\text{if3}) \quad & \frac{\langle e_1, s, M \rangle \longrightarrow \langle e'_1, s', M' \rangle}{\langle \text{if } e_1 \text{ then } e_2 \text{ else } e_3, s, M \rangle \longrightarrow \langle \text{if } e'_1 \text{ then } e_2 \text{ else } e_3, s', M' \rangle} \\
(\text{while}) \quad & \langle \text{while } e_1 \text{ do } e_2, s, M \rangle \longrightarrow \langle \text{if } e_1 \text{ then } (e_2; \text{while } e_1 \text{ do } e_2) \text{ else skip}, s, M \rangle \\
(\text{parallel1}) \quad & \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} \\
(\text{parallel2}) \quad & \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} \\
(\text{lock}) \quad & \langle \text{lock } m, s, M \rangle \longrightarrow \langle (), s, M + \{m \mapsto \text{true}\} \rangle \text{ if } \neg M(m) \\
(\text{unlock}) \quad & \langle \text{unlock } m, s, M \rangle \longrightarrow \langle (), s, M + \{m \mapsto \text{false}\} \rangle
\end{aligned}$$

### Thread-Local Semantics

$$\begin{aligned}
(\text{t-op } +) \quad & n_1 + n_2 \xrightarrow{\tau} n \quad \text{if } n = n_1 + n_2 \\
(\text{t-op } \geq) \quad & n_1 \geq n_2 \xrightarrow{\tau} b \quad \text{if } b = (n_1 \geq n_2) \\
(\text{t-op1}) \quad & \frac{e_1 \xrightarrow{a} e'_1}{e_1 \text{ op } e_2 \xrightarrow{a} e'_1 \text{ op } e_2} \\
(\text{t-op2}) \quad & \frac{e_2 \xrightarrow{a} e'_2}{v \text{ op } e_2 \xrightarrow{a} v \text{ op } e'_2} \\
(\text{t-deref}) \quad & !\ell \xrightarrow{! \ell = n} n \\
(\text{t-assign1}) \quad & \ell := n \xrightarrow{\ell := n} \text{skip} \\
(\text{t-assign2}) \quad & \frac{e \xrightarrow{a} e'}{\ell := e \xrightarrow{a} \ell := e'} \\
(\text{t-seq1}) \quad & \text{skip}; e_2 \xrightarrow{\tau} e_2 \\
(\text{t-seq2}) \quad & \frac{e_1 \xrightarrow{a} e'_1}{e_1; e_2 \xrightarrow{a} e'_1; e_2} \\
(\text{t-if1}) \quad & \text{if true then } e_2 \text{ else } e_3 \xrightarrow{\tau} e_2 \\
(\text{t-if2}) \quad & \text{if false then } e_2 \text{ else } e_3 \xrightarrow{\tau} e_3 \\
(\text{t-if3}) \quad & \frac{e_1 \xrightarrow{a} e'_1}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \xrightarrow{a} \text{if } e'_1 \text{ then } e_2 \text{ else } e_3} \\
(\text{t-while}) \quad & \text{while } e_1 \text{ do } e_2 \xrightarrow{\tau} \text{if } e_1 \text{ then } (e_2; \text{while } e_1 \text{ do } e_2) \text{ else skip} \\
(\text{t-parallel1}) \quad & \frac{e_1 \xrightarrow{a} e'_1}{e_1 | e_2 \xrightarrow{a} e'_1 | e_2} \\
(\text{t-parallel2}) \quad & \frac{e_2 \xrightarrow{a} e'_2}{e_1 | e_2 \xrightarrow{a} e_1 | e'_2} \\
(\text{t-lock}) \quad & \text{lock } m \xrightarrow{\text{lock } m} () \\
(\text{t-unlock}) \quad & \text{unlock } m \xrightarrow{\text{unlock } m} ()
\end{aligned}$$

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

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

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

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

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

### Now can make the Ordered 2PL Discipline precise

Say  $e$  obeys the discipline if for any (finite or infinite)

$$e \xrightarrow{a_1} e_1 \xrightarrow{a_2} e_2 \xrightarrow{a_3} \dots$$

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

Slide 234

### ... and make the guaranteed properties precise

Say  $e_1, \dots, e_k$  are *serializable* if for any initial store  $s$ , if

$$\langle (e_1 | \dots | e_k), s, M_0 \rangle \longrightarrow^* \langle e', s', M' \rangle \not\rightarrow$$

then for some permutation  $\pi$  we have  $\langle e_{\pi(1)}; \dots; e_{\pi(k)}, s, M_0 \rangle \longrightarrow^* \langle e'', s', M' \rangle$ .

Say they are *deadlock-free* if for any initial store  $s$ , if

$$\langle (e_1 | \dots | e_k), s, M_0 \rangle \longrightarrow^* \langle e', s', M \rangle \not\rightarrow$$

then not  $e' \xrightarrow{\mathbf{lock} \ m} e''$ , i.e.  $e'$  does not contain any blocked  $\mathbf{lock} \ m$  subexpressions.

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

Slide 235

### The Theorem

**Conjecture 30** If each  $e_i$  obeys the discipline, then  $e_1, \dots, e_k$  are serializable and deadlock-free.

(may be false!)

**Proof strategy:** Consider a (derivation of a) computation

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

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.

Slide 236

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

Slide 237

### 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)

Slide 238



## 8.1 Exercises

**Exercise 39** ★★ *Are the mutexes specified here similar to those described in C&DS?*

**Exercise 40** ★★ *Can you show all the conditions for O2PL are necessary, by giving for each an example that satisfies all the others and either is not serialisable or deadlocks?*

**Exercise 41** ★★★★★ *Prove the Conjecture about it.*

**Exercise 42** ★★★★★ *Write a semantics for an extension of L1 with threads that are more like Unix threads (e.g. with thread ids, fork, etc..). Include some of the various ways Unix threads can exchange information.*

## 9 Epilogue

Slide 239

### Epilogue

Slide 240

#### 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.

Slide 241

#### 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)

Slide 242

#### 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 expressing design choices, understanding language features and how they interact.
  - (b) for proving properties of a language, eg type safety, decidability of type inference.
3. as a foundation for proving properties of particular programs



## A Interpreter and type checker for L1 (ML)

Here is an interpreter and type checker for L1. You can download the source code from the course website.

```
(* 2002-11-08 -- Time-stamp: <2006-10-25 09:22:33 pes20>    --SML-*-* *)
(* Peter Sewell                                           *)
```

```
(* This file contains an interpreter, pretty-printer and type-checker
   for the language L1. To make it go, copy it into a working
   directory, ensure Moscow ML is available, and type
```

```
    mosml -P full l1.ml
```

```
That will give you a MoscowML top level in which these definitions
are present. You can then type
```

```
doit ();
```

```
to show the reduction sequence of < l1:=3;!l1 , {l1=0 } >, and
```

```
doit2 ();
```

```
to run the type-checker on the same simple example; you can try
other examples analogously. This file doesn't have a parser for
l1, so you'll have to enter the abstract syntax directly, eg
```

```
prettyreduce (Seq( Assign ("l1",Integer 3), Deref "l1"), [("l1",0)]);
```

```
This has been tested with Moscow ML version 2.00 (June 2000), but
should work with any other implementation of Standard ML.    *)
```

```
(* *****
(* the abstract syntax *)
(* *****
```

```
type loc = string
```

```
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
```

```
(* *****
(* an interpreter for the semantics *)
(* *****
```

```
fun is_value (Integer n) = true
  | is_value (Boolean b) = true
  | is_value (Skip) = true
  | is_value _ = false
```

```

(* In the semantics, a store is a finite partial function from
locations to integers. In the implementation, we represent a store
as a list of loc*int pairs containing, for each l in the domain of
the store, exactly one element of the form (l,n). The operations

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

both return NONE if given a location that is not in the domain of
the store. This is not a very efficient implementation, but it is
simple. *)

type store = (loc * int) list

fun lookup ( [], l ) = NONE
  | lookup ( (l',n')::pairs, l) =
    if l=l' then SOME n' else lookup (pairs,l)

fun update' front [] (l,n) = NONE
  | update' front ((l',n')::pairs) (l,n) =
    if l=l' then
      SOME(front @ ((l,n)::pairs) )
    else
      update' ((l',n')::front) pairs (l,n)

fun update (s, (l,n)) = update' [] s (l,n)

(* now define the single-step function

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

which takes a configuration (e,s) and returns either NONE, if it has
no transitions, or SOME (e',s'), if it has a transition (e,s) -->
(e',s').

Note that the code depends on global properties of the semantics,
including the fact that it defines a deterministic transition
system, so the comments indicating that particular lines of code
implement particular semantic rules are not the whole story. *)

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)    (*op + *)
    | (Integer n1, GTEQ, Integer n2) => SOME(Boolean (n1 >= n2), s) (*op >=*)
    | (e1,opr,e2) => (
      if (is_value e1) then (
        case reduce (e2,s) of
          SOME (e2',s') => SOME (Op(e1,opr,e2'),s')    (* (op2) *)
        | NONE => NONE )
      else (
        case reduce (e1,s) of
          SOME (e1',s') => SOME(Op(e1',opr,e2),s')    (* (op1) *)
        | NONE => NONE ) ) )
  | reduce (If (e1,e2,e3),s) =
    (case e1 of
      Boolean(true) => SOME(e2,s)    (* (if1) *)
    | Boolean(false) => SOME(e3,s)  (* (if2) *)
    | _ => (case reduce (e1,s) of
      SOME(e1',s') => SOME(If(e1',e2,e3),s')    (* (if3) *)

```

```

        | NONE => NONE ))
| reduce (Deref l,s) =
  (case lookup (s,l) of
    SOME n => SOME(Integer n,s)          (* (deref) *)
    | NONE => NONE )
| reduce (Assign (l,e),s) =
  (case e of
    Integer n => (case update (s,(l,n)) of
      SOME s' => SOME(Skip, s')          (* (assign1) *)
      | NONE => NONE)
    | _ => (case reduce (e,s) of
      SOME (e',s') => SOME(Assign (l,e'), s')  (* (assign2) *)
      | NONE => NONE ) )
| reduce (While (e1,e2),s) = SOME( If(e1,Seq(e2,While(e1,e2)),Skip),s) (* (while) *)
| reduce (Skip,s) = NONE
| reduce (Seq (e1,e2),s) =
  (case e1 of
    Skip => SOME(e2,s)                    (* (seq1) *)
    | _ => ( case reduce (e1,s) of
      SOME (e1',s') => SOME(Seq (e1',e2), s')  (* (seq2) *)
      | NONE => NONE ) )

(* now define the many-step evaluation function

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

  which takes a configuration (e,s) and returns the unique (e',s')
  such that (e,s) -->* (e',s') -/-. *)

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

(* ***** *)
(* typing *)
(* ***** *)

(* types *)

datatype type_L1 =
  int
  | unit
  | bool

datatype type_loc =
  intref

type typeEnv = (loc*type_loc) list

(* in the semantics, type environments gamma are partial functions
from locations to the singleton set {intref}. Here, just as we did for
stores, we represent them as a list of loc*type_loc pairs containing,
for each l in the domain of the type environment, exactly one element
of the form (l,intref). *)

(* ***** *)
(* type inference *)
(* ***** *)

```

```

(* infertype : typeEnv -> expr -> type_L1 option *)

(* again, we depend on a uniqueness property, without which we would
have to have infertype return a type_L1 list of all the possible types *)

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
        (SOME bool, SOME unit) => SOME unit
        | _ => NONE )

```

## B Interpreter and type checker for L1 (Java)

Here is an interpreter and type checker for L1, written in Java by Matthew Parkinson.

Note the different code organization between the ML and Java versions: the ML has a datatype with a constructor for each clause of the abstract syntax grammar, and `reduce` and `inferType` function definitions that each have a case for each of those constructors; the Java has a subclass of `Expression` for each clause of the abstract syntax, each of which defines `smallStep` and `typecheck` methods.

```
public class L1 {

    public static void main(String [] args) {
        Location l1 = new Location ("l1");
        Location l2 = new Location ("l2");
        Location l3 = new Location ("l3");
        State s1 = new State()
            .add(l1,new Int(1))
            .add(l2,new Int(5))
            .add(l3,new Int(0));

        Environment env = new Environment()
            .add(l1).add(l2).add(l3);

        Expression e =
            new Seq(new While(new GTeq(new Deref(l2),new Deref(l1)),
                new Seq(new Assign(l3, new Plus(new Deref(l1),new Deref(l3))),
                    new Assign(l1,new Plus(new Deref(l1),new Int(1))))),
                ),
            new Deref(l3))
        ;
    try{
        //Type check
        Type t= e.typeCheck(env);
        System.out.println("Program has type: " + t);

        //Evaluate program
        System.out.println(e + "\n \n");
        while(!(e instanceof Value) ){
            e = e.smallStep(s1);
            //Display each step of reduction
            System.out.println(e + "\n \n");
        }
        //Give some output
        System.out.println("Program has type: " + t);
        System.out.println("Result has type: " + e.typeCheck(env));
        System.out.println("Result: " + e);
        System.out.println("Terminating State: " + s1);
    } catch (TypeError te) {
        System.out.println("Error:\n" + te);
        System.out.println("From code:\n" + e);
    } catch (CanNotReduce cnr) {
        System.out.println("Caught Following exception" + cnr);
        System.out.println("While trying to execute:\n " + e);
        System.out.println("In state: \n " + s1);
    }
}

}

class Location {
    String name;
}
```

```

    Location(String n) {
        this.name = n;
    }
    public String toString() {return name;}
}

class State {
    java.util.HashMap store = new java.util.HashMap();

    //Used for setting the initial store for testing not used by
    //semantics of L1
    State add(Location l, Value v) {
        store.put(l,v);
        return this;
    }

    void update(Location l, Value v) throws CanNotReduce {
        if(store.containsKey(l)) {
            if(v instanceof Int) {
                store.put(l,v);
            }
            else throw new CanNotReduce("Can only store integers");
        }
        else throw new CanNotReduce("Unknown location!");
    }

    Value lookup(Location l) throws CanNotReduce {
        if(store.containsKey(l)) {
            return (Int)store.get(l);
        }
        else throw new CanNotReduce("Unknown location!");
    }
    public String toString() {
        String ret = "[";
        java.util.Iterator iter = store.entrySet().iterator();
        while(iter.hasNext()) {
            java.util.Map.Entry e = (java.util.Map.Entry)iter.next();
            ret += "(" + e.getKey() + " |-> " + e.getValue() + ")";
            if(iter.hasNext()) ret += ", ";
        }
        return ret + "]";
    }
}

class Environment {
    java.util.HashSet env = new java.util.HashSet();

    //Used to initially setup environment, not used by type checker.
    Environment add(Location l) {
        env.add(l); return this;
    }

    boolean contains(Location l) {
        return env.contains(l);
    }
}

class Type {
    int type;
    Type(int t) {type = t;}
    public static final Type BOOL = new Type(1);
    public static final Type INT = new Type(2);
    public static final Type UNIT = new Type(3);
}

```



```

    public String toString() {
        switch(type) {
            case 1: return "BOOL";
            case 2: return "INT";
            case 3: return "UNIT";
        }
        return "???" ;
    }
}

abstract class Expression {
    abstract Expression smallStep(State state) throws CanNotReduce;
    abstract Type typeCheck(Environment env) throws TypeError;
}

abstract class Value extends Expression {
    final Expression smallStep(State state) throws CanNotReduce{
        throw new CanNotReduce("I'm a value");
    }
}

class CanNotReduce extends Exception{
    CanNotReduce(String reason) {super(reason);}
}

class TypeError extends Exception { TypeError(String reason) {super(reason);}}

class Bool extends Value {
    boolean value;

    Bool(boolean b) {
        value = b;
    }

    public String toString() {
        return value ? "TRUE" : "FALSE";
    }

    Type typeCheck(Environment env) throws TypeError {
        return Type.BOOL;
    }
}

class Int extends Value {
    int value;
    Int(int i) {
        value = i;
    }
    public String toString(){return ""+ value;}

    Type typeCheck(Environment env) throws TypeError {
        return Type.INT;
    }
}

class Skip extends Value {
    public String toString(){return "SKIP";}
    Type typeCheck(Environment env) throws TypeError {
        return Type.UNIT;
    }
}

```

```

class Seq extends Expression {
    Expression exp1,exp2;
    Seq(Expression e1, Expression e2) {
        exp1 = e1;
        exp2 = e2;
    }

    Expression smallStep(State state) throws CanNotReduce {
        if(exp1 instanceof Skip) {
            return exp2;
        } else {
            return new Seq(exp1.smallStep(state),exp2);
        }
    }
    public String toString() {return exp1 + "; " + exp2;}

    Type typeCheck(Environment env) throws TypeError {
        if(exp1.typeCheck(env) == Type.UNIT) {
            return exp2.typeCheck(env);
        }
        else throw new TypeError("Not a unit before ';'");
    }
}

class GTeq extends Expression {
    Expression exp1, exp2;
    GTeq(Expression e1,Expression e2) {
        exp1 = e1;
        exp2 = e2;
    }

    Expression smallStep(State state) throws CanNotReduce {
        if(!( exp1 instanceof Value)) {
            return new GTeq(exp1.smallStep(state),exp2);
        } else if (!( exp2 instanceof Value)) {
            return new GTeq(exp1, exp2.smallStep(state));
        } else {
            if( exp1 instanceof Int && exp2 instanceof Int ) {
                return new Bool(((Int)exp1).value >= ((Int)exp2).value);
            }
            else throw new CanNotReduce("Operands are not both integers.");
        }
    }
    public String toString(){return exp1 + " >= " + exp2;}

    Type typeCheck(Environment env) throws TypeError {
        if(exp1.typeCheck(env) == Type.INT && exp2.typeCheck(env) == Type.INT) {
            return Type.BOOL;
        }
        else throw new TypeError("Arguments not both integers.");
    }
}

class Plus extends Expression {
    Expression exp1, exp2;
    Plus(Expression e1,Expression e2) {
        exp1 = e1;
        exp2 = e2;
    }

    Expression smallStep(State state) throws CanNotReduce {

```

```

        if(!( exp1 instanceof Value)) {
            return new Plus(exp1.smallStep(state),exp2);
        } else if (!( exp2 instanceof Value)) {
            return new Plus(exp1, exp2.smallStep(state));
        } else {
            if( exp1 instanceof Int && exp2 instanceof Int ) {
                return new Int(((Int)exp1).value + ((Int)exp2).value);
            }
            else throw new CanNotReduce("Operands are not both integers.");
        }
    }
    public String toString(){return exp1 + " + " + exp2;}

    Type typeCheck(Environment env) throws TypeError {
        if(exp1.typeCheck(env) == Type.INT && exp2.typeCheck(env) == Type.INT) {
            return Type.INT;
        }
        else throw new TypeError("Arguments not both integers.");
    }
}

class IfThenElse extends Expression {
    Expression exp1,exp2,exp3;

    IfThenElse (Expression e1, Expression e2,Expression e3) {
        exp1 = e1;
        exp2 = e2;
        exp3 = e3;
    }

    Expression smallStep(State state) throws CanNotReduce {
        if(exp1 instanceof Value) {
            if(exp1 instanceof Bool) {
                if(((Bool)exp1).value)
                    return exp2;
                else
                    return exp3;
            }
            else throw new CanNotReduce("Not a boolean in test.");
        }
        else {
            return new IfThenElse(exp1.smallStep(state),exp2,exp3);
        }
    }
    public String toString() {return "IF " + exp1 + " THEN " + exp2 + " ELSE " + exp3;}

    Type typeCheck(Environment env) throws TypeError {
        if(exp1.typeCheck(env) == Type.BOOL) {
            Type t = exp2.typeCheck(env);
            if(exp3.typeCheck(env) == t)
                return t;
            else throw new TypeError("If branches not the same type.");
        }
        else throw new TypeError("If test is not bool.");
    }
}

class Assign extends Expression {
    Location l;
    Expression exp1;
}

```

```

Assign(Location l, Expression exp1) {
    this.l = l;
    this.exp1 = exp1;
}

Expression smallStep(State state) throws CanNotReduce{
    if(exp1 instanceof Value) {
        state.update(l, (Value)exp1);
        return new Skip();
    }
    else {
        return new Assign(l, exp1.smallStep(state));
    }
}
public String toString() {return l + " = " + exp1;}

Type typeCheck(Environment env) throws TypeError {
    if(env.contains(l) && exp1.typeCheck(env) == Type.INT) {
        return Type.UNIT;
    }
    else throw new TypeError("Invalid assignment");
}
}

class Deref extends Expression {
    Location l;

    Deref(Location l) {
        this.l = l;
    }

    Expression smallStep(State state) throws CanNotReduce {
        return state.lookup(l);
    }
    public String toString() {return "!" + l;}

    Type typeCheck(Environment env) throws TypeError {
        if(env.contains(l)) return Type.INT;
        else throw new TypeError("Location not known about!");
    }
}

class While extends Expression {
    Expression exp1, exp2;

    While(Expression e1, Expression e2) {
        exp1 = e1;
        exp2 = e2;
    }

    Expression smallStep(State state) throws CanNotReduce {
        return new IfThenElse(exp1, new Seq(exp2, this), new Skip());
    }

    public String toString(){return "WHILE " + exp1 + " DO {" + exp2 + "}}";

    Type typeCheck(Environment env) throws TypeError {
        if(exp1.typeCheck(env) == Type.BOOL && exp2.typeCheck(env) == Type.UNIT)
            return Type.UNIT;
        else throw new TypeError("Error in while loop");
    }
}
}

```

## C How to do Proofs

The purpose of this handout is give a general guide as to how to prove theorems. This should give you some help in answering questions that begin with “Show that the following is true ...”. It is based on notes by Myra VanInwegen, with additional text added by Peter Sewell in §C.1. Many thanks to Myra for making her original notes available.

The focus here is on doing *informal but rigorous proofs*. These are rather different from the *formal proofs*, in Natural Deduction or Sequent Calculus, that were introduced in the Logic and Proof course. Formal proofs are derivations in one of those proof systems – they are in a completely well-defined form, but are often far too verbose to deal with by hand (although they can be machine-checked). Informal proofs, on the other hand, are the usual mathematical notion of proof: written arguments to persuade the reader that you could, if pushed, write a fully formal proof.

This is important for two reasons. Most obviously, you should learn how to do these proofs. More subtly, but more importantly, only by working with the mathematical definitions in some way can you develop a good intuition for what they mean — trying to do some proofs is the best way of understanding the definitions.

### C.1 How to go about it

Proofs differ, but for many of those you meet the following steps should be helpful.

1. Make sure the statement of the conjecture is precisely defined. In particular, make sure you understand any strange notation, and find the definitions of all the auxiliary gadgets involved (e.g. definitions of any typing or reduction relations mentioned in the statement, or any other predicates or functions).
2. Try to understand at an intuitive level what the conjecture is saying – verbalize out loud the basic point. For example, for a Type Preservation conjecture, the basic point might be something like “if a well-typed configuration reduces, the result is still well-typed (with the same type)”.
3. Try to understand intuitively why it is true (or false...). Identify what the most interesting cases might be — the cases that you think are most likely to be suspicious, or hard to prove. Sometimes it’s good to start with the easy cases (if the setting is unfamiliar to you); sometimes it’s good to start with the hard cases (to find any interesting problems as soon as possible).
4. Think of a good basic strategy. This might be:
  - (a) simple logic manipulations;
  - (b) collecting together earlier results, again by simple logic; or
  - (c) some kind of induction.
5. Try it! (remembering you might have to backtrack if you discover you picked a strategy that doesn’t work well for this conjecture). This might involve any of the following:
  - (a) Expanding definitions, inlining them. Sometimes you can just blindly expand all definitions, but more often it’s important to expand only the definitions which you want to work with the internal structure of — otherwise things just get too verbose.
  - (b) Making abbreviations — defining a new variable to stand for some complex gadget you’re working with, saying e.g.

where  $e = (\text{let } x:\text{int} = 7+2 \text{ in } x+x)$

Take care with choosing variable names.

- (c) Doing *equational reasoning*, e.g.

$e = e1$  by ...  
 $= e2$  by ...  
 $= e3$  as ...

Here the  $e$  might be any mathematical object — arithmetic expressions, or expressions of some grammar, or formulae. Some handy equations over formulae are given in §C.2.2.

- (d) Proving a formula based on its structure. For example, to prove a formula  $\forall x \in S.P(x)$  you would often assume you have an arbitrary  $x$  and then try to prove  $P(x)$ .

Take an arbitrary  $x \in S$ .  
We now have to show  $P(x)$ :

This is covered in detail in §C.2.3. Much proof is of this form, automatically driven by the structure of the formula.

- (e) Using an assumption you’ve made above.

- (f) Induction. As covered in the 1B Semantics notes, there are various kinds of induction you might want to use: mathematical induction over the natural numbers, structural induction over the elements of some grammar, or rule induction over the rules defining some relation (especially a reduction or typing relation). For each, you should:

- i. Decide (and state!) what kind of induction you’re using. This may need some thought and experience, and you might have to backtrack.
- ii. Remind yourself what the induction principle is exactly.
- iii. Decide on the induction hypothesis you’re going to use, writing down a predicate  $\Phi$  which is such that the conclusion of the induction principle implies the thing you’re trying to prove. Again, this might need some thought. Take care with the quantifiers here — it’s suspicious if your definition of  $\Phi$  has any globally-free variables...
- iv. Go through each of the premises of the induction principle and prove each one (using any of these techniques as appropriate). Many of those premises will be implications, e.g.  $\forall x \in \mathbb{N}.\Phi(x) \Rightarrow \Phi(x+1)$ , for which you can do a proof based on the structure of the formula — taking an arbitrary  $x$ , assuming  $\Phi(x)$ , and trying to prove  $\Phi(x+1)$ . Usually at some point in the latter you’d make use of the assumption  $\Phi(x)$ .

6. In all of the above, remember: the point of doing a proof on paper is to *use* the formalism to *help you think* — to help you cover all cases, precisely — and also to *communicate with the reader*. For both, you need to write clearly:

- (a) Use enough words! “Assume”, “We have to show”, “By such-and-such we know”, “Hence”,...
- (b) Don’t use random squiggles. It’s good to have formulae properly nested within text, with and no “ $\Rightarrow$ ” or “ $\therefore$ ” between lines of text.

7. If it hasn’t worked yet... either

- (a) you’ve make some local mistake, e.g. mis-instantiated something, or used the same variable for two different things, or not noticed that you have a definition you should have expanded or an assumption you should have used. Fix it and continue.

- (b) you've discovered that the conjecture is really false. Usually at this point it's a good idea to construct a counterexample that is as simple as possible, and to check carefully that it really is a counterexample.
  - (c) you need to try a different strategy — often, to use a different induction principle or to strengthen your induction hypothesis.
  - (d) you didn't really understand intuitively what the conjecture is saying, or what the definitions it uses mean. Go back to them again.
8. If it has worked: read through it, skeptically, and check. Maybe you'll need to *re-write* it to make it comprehensible: proof *discovery* is not the same as proof *exposition*. See the example proofs in the Semantics notes.
  9. Finally, give it to someone else, as skeptical and careful as you can find, to see if they believe it — to see if they believe that *what you've written down is a proof*, not that they believe that *the conjecture is true*.

## C.2 And in More Detail...

First, I'll explain informal proof intuitively, giving a couple of examples. Then I'll explain how this intuition is reflected in the sequent rules from Logic and Proof.

In the following, I'll call any logic statement a formula. In general, what we'll be trying to do is *prove* a formula, using a collection of formulas that we know to be true or are assuming to be true. There's a big difference between *using* a formula and *proving* a formula. In fact, what you do is in many ways opposite. So, I'll start by explaining how to *prove* a formula.

### C.2.1 Meet the Connectives

Here are the logical connectives and a very brief description of what each means.

$P \wedge Q$	$P$ and $Q$ are both true
$P \vee Q$	$P$ is true, or $Q$ is true, or both are true
$\neg P$	$P$ is not true ( $P$ is false)
$P \Rightarrow Q$	if $P$ is true then $Q$ is true
$P \Leftrightarrow Q$	$P$ is true exactly when $Q$ is true
$\forall x \in S.P(x)$	for all $x$ in $S$ , $P$ is true of $x$
$\exists x \in S.P(x)$	there exists an $x$ in $S$ such that $P$ holds of $x$

### C.2.2 Equivalences

These are formulas that mean the same thing, and this is indicated by a  $\simeq$  between them. The fact that they are equivalent to each other is justified by the truth tables of the connectives.

definition of $\Rightarrow$	$P \Rightarrow Q \simeq \neg P \vee Q$
definition of $\Leftrightarrow$	$P \Leftrightarrow Q \simeq (P \Rightarrow Q) \wedge (Q \Rightarrow P)$
definition of $\neg$	$\neg P \simeq P \Rightarrow \text{false}$
de Morgan's Laws	$\neg(P \wedge Q) \simeq \neg P \vee \neg Q$ $\neg(P \vee Q) \simeq \neg P \wedge \neg Q$
extension to quantifiers	$\neg(\forall x.P(x)) \simeq \exists x.\neg P(x)$ $\neg(\exists x.P(x)) \simeq \forall x.\neg P(x)$
distributive laws	$P \vee (Q \wedge R) \simeq (P \vee Q) \wedge (P \vee R)$ $P \wedge (Q \vee R) \simeq (P \wedge Q) \vee (P \wedge R)$
coalescing quantifiers	$(\forall x.P(x)) \wedge (\forall x.Q(x)) \simeq \forall x.(P(x) \wedge Q(x))$ $(\exists x.P(x)) \vee (\exists x.Q(x)) \simeq \exists x.(P(x) \vee Q(x))$
these ones apply if $x$ is not free in $Q$	$(\forall x.P(x)) \wedge Q \simeq (\forall x.P(x) \wedge Q)$ $(\forall x.P(x)) \vee Q \simeq (\forall x.P(x) \vee Q)$ $(\exists x.P(x)) \wedge Q \simeq (\exists x.P(x) \wedge Q)$ $(\exists x.P(x)) \vee Q \simeq (\exists x.P(x) \vee Q)$

### C.2.3 How to Prove a Formula

For each of the logical connectives, I'll explain how to handle them.

$\boxed{\forall x \in S.P(x)}$  This means “For all  $x$  in  $S$ ,  $P$  is true of  $x$ .” Such a formula is called a universally quantified formula. The goal is to prove that the property  $P$ , which has some  $x$ s somewhere in it, is true no matter what value in  $S$   $x$  takes on. Often the “ $\in S$ ” is left out. For example, in a discussion of lists, you might be asked to prove  $\forall l.\text{length } l > 0 \Rightarrow \exists x.\text{member}(x, l)$ . Obviously,  $l$  is a list, even if it isn't explicitly stated as such.

There are several choices as to how to prove a formula beginning with  $\forall x$ . The standard thing to do is to just prove  $P(x)$ , not assuming anything about  $x$ . Thus, in doing the proof you sort of just mentally strip off the  $\forall x$ . What you would write when doing this is “Let  $x$  be any  $S$ ”. However, there are some subtleties—if you're already using an  $x$  for something else, you can't use the same  $x$ , because then you *would* be assuming something about  $x$ , namely that it equals the  $x$  you're already using. In this case, you need to use alpha-conversion<sup>1</sup> to change the formula you want to prove to  $\forall y \in S.P(y)$ , where  $y$  is some variable you're not already using, and then prove  $P(y)$ . What you could write in this case is “Since  $x$  is already in use, we'll prove the property of  $y$ ”.

An alternative is induction, if  $S$  is a set that is defined with a structural definition. Many objects you're likely to be proving properties of are defined with a structural definition. This includes natural numbers, lists, trees, and terms of a computer language. Sometimes you can use induction over the natural numbers to prove things about other objects, such as graphs, by inducting over the number of nodes (or edges) in a graph.

You use induction when you see that during the course of the proof you would need to use the property  $P$  for the subparts of  $x$  in order to prove it for  $x$ . This usually ends up being the case if  $P$  involves functions defined recursively (i.e., the return value for the function depends on the function value on the subparts of the argument).

A special case of induction is case analysis. It's basically induction where you don't use the inductive hypothesis: you just prove the property for each possible form that  $x$  could have. Case analysis can be used to prove the theorem about lists above.

A final possibility (which you can use for all formulas, not just for universally quantified ones) is to assume the contrary, and then derive a contradiction.

<sup>1</sup>Alpha-equivalence says that the name of a bound variable doesn't matter, so you can change it at will (this is called alpha-conversion). You'll get to know the exact meaning of this soon enough so I won't explain this here.



$\boxed{\exists x \in S.P(x)}$  This says “There exists an  $x$  in  $S$  such that  $P$  holds of  $x$ .” Such a formula is called an existentially quantified formula. The main way to prove this is to figure out what  $x$  has to be (that is, to find a concrete representation of it), and then prove that  $P$  holds of that value. Sometimes you can’t give a completely specified value, since the value you pick for  $x$  has to depend on the values of other things you have floating around. For example, say you want to prove

$$\forall x, y \in \mathbb{R}. x < y \wedge \sin x < 0 \wedge \sin y > 0 \Rightarrow \exists z. x < z \wedge z < y \wedge \sin z = 0$$

where  $\mathbb{R}$  is the set of real numbers. By the time you get to dealing with the  $\exists z. x < z \wedge z < y \wedge \sin z = 0$ , you will have already assumed that  $x$  and  $y$  were any real numbers. Thus the value you choose for  $z$  has to depend on whatever  $x$  and  $y$  are.

An alternative way to prove  $\exists x \in S.P(x)$  is, of course, to assume that no such  $x$  exists, and derive a contradiction.

To summarize what I’ve gone over so far: to *prove* a universally quantified formula, you must prove it for a generic variable, one that you haven’t used before. To prove an existentially quantified formula, you get to choose a value that you want to prove the property of.

$\boxed{P \Rightarrow Q}$  This says “If  $P$  is true, then  $Q$  is true”. Such a formula is called an implication, and it is often pronounced “ $P$  implies  $Q$ ”. The part before the  $\Rightarrow$  sign (here  $P$ ) is called the antecedent, and the part after the  $\Rightarrow$  sign (here  $Q$ ) is called the consequent.  $P \Rightarrow Q$  is equivalent to  $\neg P \vee Q$ , and so if  $P$  is false, or if  $Q$  is true, then  $P \Rightarrow Q$  is true.

The standard way to prove this is to assume  $P$ , then use it to help you prove  $Q$ . Note that I said that you will be *using*  $P$ . Thus you will need to follow the rules in Section C.2.4 to deal with the logical connectives in  $P$ .

Other ways to prove  $P \Rightarrow Q$  involve the fact that it is equivalent to  $\neg P \vee Q$ . Thus, you can prove  $\neg P$  without bothering with  $Q$ , or you can just prove  $Q$  without bothering with  $P$ . To reason by contradiction you assume that  $P$  is true and that  $Q$  is not true, and derive a contradiction.

Another alternative is to prove the contrapositive:  $\neg Q \Rightarrow \neg P$ , which is equivalent to it.

$\boxed{P \Leftrightarrow Q}$  This says “ $P$  is true if and only if  $Q$  is true”. The phrase “if and only if” is usually abbreviated “iff”. Basically, this means that  $P$  and  $Q$  are either both true, or both false.

Iff is usually used in two main ways: one is where the equivalence is due to one formula being a definition of another. For example,  $A \subseteq B \Leftrightarrow (\forall x. x \in A \Rightarrow x \in B)$  is the standard definition of subset. For these iff statements, you don’t have to prove them. The other use of iff is to state the equivalence of two different things. For example, you could define an SML function **fact**:

```
fun fact 0 = 1
  | fact n = n * fact (n - 1)
```

Since in SML whole numbers are integers (both positive and negative) you may be asked to prove: **fact**  $x$  terminates  $\Leftrightarrow x \geq 0$ . The standard way to do this is us the equivalence  $P \Leftrightarrow Q$  is equivalent to  $P \Rightarrow Q \wedge Q \Rightarrow P$ . And so you’d prove that (**fact**  $x$  terminates  $\Rightarrow x \geq 0$ )  $\wedge$  ( $x \geq 0 \Rightarrow$  **fact**  $x$  terminates).

$\boxed{\neg P}$  This says “ $P$  is not true”. It is equivalent to  $P \Rightarrow \text{false}$ , thus this is one of the ways you prove it: you assume that  $P$  is true, and derive a contradiction (that is, you prove false). Here’s an example of this, which you’ll run into later this year: the undecidability of the halting problem can be rephrased as  $\neg \exists x \in RM. x \text{ solves the halting problem}$ , where  $RM$  is the set of register machines. The proof of this in your Computation Theory notes follows exactly the pattern I described—it assumes there is such a machine and derives a contradiction.

The other major way to prove  $\neg P$  is to figure out what the negation of  $P$  is, using equivalences like De Morgan's Law, and then prove that. For example, to prove  $\neg \forall x \in \mathcal{N}. \exists y \in \mathcal{N}. x = y^2$ , where  $\mathcal{N}$  is the set of natural numbers, you could push in the negation to get:  $\exists x \in \mathcal{N}. \forall y \in \mathcal{N}. x \neq y^2$ , and then you could prove that.

$\boxed{P \wedge Q}$  This says “ $P$  is true and  $Q$  is true”. Such a formula is called a conjunction. To prove this, you have to prove  $P$ , and you have to prove  $Q$ .

$\boxed{P \vee Q}$  This says “ $P$  is true or  $Q$  is true”. This is *inclusive* or: if  $P$  and  $Q$  are both true, then  $P \vee Q$  is still true. Such a formula is called a disjunction. To prove this, you can prove  $P$  or you can prove  $Q$ . You have to choose which one to prove. For example, if you need to prove  $(5 \bmod 2 = 0) \vee (5 \bmod 2 = 1)$ , then you'll choose the second one and prove that.

However, as with existentials, the choice of which one to prove will often depend on the values of other things, like universally quantified variables. For example, when you are studying the theory of programming languages (you will get a bit of this in Semantics), you might be asked to prove

$$\forall P \in ML. \quad P \text{ is properly typed} \Rightarrow \\ \text{(the evaluation of } P \text{ runs forever)} \vee (P \text{ evaluates to a value)}$$

where  $ML$  is the set of all ML programs. You don't know in advance which of these will be the case, since some programs do run forever, and some do evaluate to a value. Generally, the best way to prove the disjunction in this case (when you don't know in advance which will hold) is to use the equivalence with implication. For example, you can use the fact that  $P \vee Q$  is equivalent to  $\neg P \Rightarrow Q$ , then assume  $\neg P$ , then use this to prove  $Q$ . For example, your best bet to proving this programming languages theorem is to assume that the evaluation of  $P$  doesn't run forever, and use this to prove that  $P$  evaluates to a value.

### C.2.4 How to Use a Formula

You often end up using a formula to prove other formulas. You can use a formula if someone has already proved that it's true, or you are assuming it because it was in an implication, namely, the  $A$  in  $A \Rightarrow B$ . For each logical connective, I'll tell you how to use it.

$\boxed{\forall x \in S.P(x)}$  This formula says that something is true of *all* elements of  $S$ . Thus, when you use it, you can pick any value at all to use instead of  $x$  (call it  $v$ ), and then you can use  $P(v)$ .

$\boxed{\exists x \in S.P(x)}$  This formula says that there is some  $x$  that satisfies  $P$ . However, you do not know what it is, so you can not assume anything about it. The usual approach is to just say that the thing that is being said to exist is just  $x$ , and use the fact that  $P$  holds of  $x$  to prove something else. However, if you're already using an  $x$  for something else, you have to pick another variable to represent the thing that exists.

To summarize this: to *use* a universally quantified formula, you can choose any value, and use that the formula holds for that variable. To use an existentially quantified formula, you must not assume anything about the value that is said to exist, so you just use a variable (one that you haven't used before) to represent it. Note that this is more or less opposite of what you do when you prove a universally or existentially quantified formula.

$\boxed{\neg P}$  Usually, the main use of this formula is to prove the negation of something else. An example is the use of reduction to prove the unsolvability of various problems in the Computation Theory (you'll learn all about this in Lent term). You want to prove  $\neg Q$ , where  $Q$  states that a certain problem (Problem 1) is decidable (in other words, you want to prove that Problem 1 is not decidable). You know  $\neg P$ , where  $P$  states that another problem (Problem 2) is decidable (i.e.  $\neg P$  says that Problem 2 is not decidable). What you

do basically is this. You first prove  $Q \Rightarrow P$ , which says that if Problem 1 is decidable, then so is Problem 2. Since  $Q \Rightarrow P \simeq \neg P \Rightarrow \neg Q$ , you have now proved  $\neg P \Rightarrow \neg Q$ . You already know  $\neg P$ , so you use modus ponens<sup>2</sup> to get that  $\neg Q$ .

$P \Rightarrow Q$  The main way to use this is that you prove  $P$ , and then you use modus ponens to get  $Q$ , which you can then use.

$P \Leftrightarrow Q$  The main use of this is to replace an occurrence of  $P$  in a formula with  $Q$ , and vice versa.

$P \wedge Q$  Here you can use both  $P$  and  $Q$ . Note, you're not *required* to use both of them, but they are both true and are waiting to be used by you if you need them.

$P \vee Q$  Here, you know that one of  $P$  or  $Q$  is true, but you do not know which one. To use this to prove something else, you have to do a split: first you prove the thing using  $P$ , then you prove it using  $Q$ .

Note that in each of the above, there is again a difference in the way you use a formula, versus the way you prove it. They are in a way almost opposites. For example, in proving  $P \wedge Q$ , you have to prove both  $P$  and  $Q$ , but when you are using the formula, you don't have to use both of them.

### C.3 An Example

There are several exercises in the Semantics notes that ask you to prove something. Here, we'll go back to Regular Languages and Finite Automata. (If they've faded, it's time to remind yourself of them.) The Pumping Lemma for regular sets (PL for short) is an astonishingly good example of the use of quantifiers. We'll go over the proof and use of the PL, paying special attention to the logic of what's happening.

#### C.3.1 Proving the PL

My favorite book on regular languages, finite automata, and their friends is the Hopcroft and Ullman book *Introduction to Automata Theory, Languages, and Computation*. You should locate this book in your college library, and if it isn't there, insist that your DoS order it for you.

In the *Automata Theory* book, the Pumping Lemma is stated as: "Let  $L$  be a regular set. Then there is a constant  $n$  such that if  $z$  is any word in  $L$ , and  $|z| \geq n$ , we may write  $z = uvw$  in such a way that  $|uw| \leq n$ ,  $|v| \geq 1$ , and for all  $i \geq 0$ ,  $uv^i w$  is in  $L$ ." The Pumping Lemma is, in my experience, one of the most difficult things about learning automata theory. It is difficult because people don't know what to do with all those logical connectives. Let's write it as a logical formula.

$$\begin{aligned} \forall L \in \text{RegularLanguages.} \\ \exists n. \forall z \in L. |z| \geq n \Rightarrow \\ \exists uvw. z = uvw \wedge |uw| \leq n \wedge |v| \geq 1 \wedge \\ \forall i \geq 0. uv^i w \in L \end{aligned}$$

Complicated, eh? Well, let's prove it, using the facts that Hopcroft and Ullman have established in the chapters previous to the one with the PL. I'll give the proof and put in square brackets comments about what I'm doing.

Let  $L$  be any regular language. [Here I'm dealing with the  $\forall L \in \text{RegularLanguages}$  by stating that I'm not assuming anything about  $L$ .] Let  $M$  be a minimal-state deterministic

<sup>2</sup>Modus ponens says that if  $A \Rightarrow B$  and  $A$  are both true, then  $B$  is true.

finite state machine accepting  $L$ . [Here I'm *using* a fact that Hopcroft and Ullman have already proved about the equivalence of regular languages and finite automata.] Let  $n$  be the number of states in this finite state machine. [I'm dealing with the  $\exists n$  by giving a very specific value of what it will be, based on the arbitrary  $L$ .] Let  $z$  be any word in  $L$ . [Thus I deal with  $\forall z \in L$ .] Assume that  $|z| \geq n$ . [Thus I'm taking care of the  $\Rightarrow$  by assuming the antecedent.]

Say  $z$  is written  $a_1a_2 \dots a_m$ , where  $m \geq n$ . Consider the states that  $M$  is in during the processing of the first  $n$  symbols of  $z$ ,  $a_1a_2 \dots a_n$ . There are  $n + 1$  of these states. Since there are only  $n$  states in  $M$ , there must be a duplicate. Say that after symbols  $a_j$  and  $a_k$  we are in the same state, state  $s$  (i.e. there's a loop from this state that the machine goes through as it accepts  $z$ ), and say that  $j < k$ . Now, let  $u = a_1a_2 \dots a_j$ . This represents the part of the string that gets you to state  $s$  the first time. Let  $v = a_{j+1} \dots a_k$ . This represents the loop that takes you from  $s$  and back to it again. Let  $w = a_{k+1} \dots a_m$ , the rest of word  $z$ . [We have chosen definite values for  $u$ ,  $v$ , and  $w$ .] Then clearly  $z = uvw$ , since  $u$ ,  $v$ , and  $w$  are just different sections of  $z$ .  $|uv| \leq n$  since  $u$  and  $v$  occur within the first  $n$  symbols of  $z$ .  $|v| \geq 1$  since  $j < k$ . [Note that we're dealing with the formulas connected with  $\wedge$  by proving each of them.]

Now, let  $i$  be a natural number (i.e.  $\geq 0$ ). [This deals with  $\forall i \geq 0$ .] Then  $uv^i w \in L$ . [Finally our conclusion, but we have to explain why this is true.] This is because we can repeat the loop from  $s$  to  $s$  (represented by  $v$ ) as many times as we like, and the resulting word will still be accepted by  $M$ .

### C.3.2 Using the PL

Now we use the PL to prove that a language is not regular. This is a rewording of Example 3.1 from Hopcroft and Ullman. I'll show that  $L = \{0^i \mid i \text{ is an integer, } i \geq 1\}$  is not regular. Note that  $L$  consists of all strings of 0's whose length is a perfect square. I will *use* the PL. I want to prove that  $L$  is not regular. I'll assume the negation (i.e., that  $L$  is regular) and derive a contradiction. So here we go. Remember that what I'm emphasizing here is not the finite automata stuff itself, but how to use a complicated theorem to prove something else.

Assume  $L$  is regular. We will use the PL to get a contradiction. Since  $L$  is regular, the PL applies to it. [We note that we're using the  $\forall$  part of the PL for this particular  $L$ .] Let  $n$  be as described in the PL. [This takes care of using the  $\exists n$ . Note that we *are not* assuming anything about its actual value, just that it's a natural number.] Let  $z = 0^{n^2}$ . [Since the PL says that something is true of *all*  $z$ s, we can choose the one we want to use it for.] So by the PL there exist  $u$ ,  $v$ , and  $w$  such that  $z = uvw$ ,  $|uv| \leq n$ ,  $|v| \geq 1$ . [Note that we don't assume anything about what the  $u$ ,  $v$ , and  $w$  actually are; the only thing we know about them is what the PL tells us about them. This is where people trying to use the PL usually screw up.] The PL then says that for any  $i$ , then  $uv^i w \in L$ . Well, then  $uv^2 w \in L$ . [This is using the  $\forall i \geq 0$  bit.] However,  $n^2 < |uv^2 w| \leq n^2 + n$ , since  $1 \leq |v| \leq n$ . But  $n^2 + n < (n + 1)^2$ . Thus  $|uv^2 w|$  lies properly between  $n^2$  and  $(n + 1)^2$  and is thus not a perfect square. Thus  $uv^2 w$  is not in  $L$ . This is a contradiction. Thus our assumption (that  $L$  was regular) was incorrect. Thus  $L$  is not a regular language.

## C.4 Sequent Calculus Rules

In this section, I will show how the intuitive approach to things that I've described above is reflected in the sequent calculus rules. A sequent is  $\Gamma \vdash \Delta$ , where  $\Gamma$  and  $\Delta$  are sets of

formulas.<sup>3</sup> Technically, this means that

$$A_1 \wedge A_2 \wedge \dots \wedge A_n \Rightarrow B_1 \vee B_2 \vee \dots \vee B_m \quad (1)$$

where  $A_1, A_2, \dots, A_n$  are the formulas in  $\Gamma$ , and  $B_1, B_2, \dots, B_m$  are the formulas in  $\Delta$ . Less formally, this means “using the formulas in  $\Gamma$  we can prove that one of the formula in  $\Delta$  is true.” This is just the intuition I described above about using vs proving formulas, except that I only talked about proving that one formula is true, rather than proving that one of several formulas is true. In order to handle the  $\vee$  connective, there can be any number of formulas on the right hand side of the  $\vdash$ .

For each logic connective,<sup>4</sup> I’ll give the rules for it, and explain how it relates to the intuitive way of using or proving formulas. For each connective there are at least two rules for it: one for the left side of the  $\vdash$ , and one for the right side. This corresponds to having different ways to treat a formula depending on whether you’re using it (for formulas on the left hand side of the  $\vdash$ ) or proving it (for formulas on the right side of the  $\vdash$ ).

It’s easiest to understand these rules from the bottom up. The conclusion of the rule (the sequent below the horizontal line) is what we want to prove. The hypotheses of the rule (the sequents above the horizontal line) are how we go about proving it. We’ll have to use more rules, adding to the top, to build up the proof of the hypothesis, but this at least tells us how to get going.

You can stop when the formula you have on the top is a *basic sequent*. This is  $\Gamma \vdash \Delta$  where there’s at least one formula (say  $P$ ) that’s in both  $\Gamma$  and  $\Delta$ . You can see why this is the basic true formula: it says that if  $P$  and the other formulas in  $\Gamma$  are true, then  $P$  or one of the other formula in  $\Delta$  is true.

In building proofs from these rules, there are several ways that you end up with formulas to the left of the  $\vdash$ , where you can use them rather than proving them. One is that you’ve already proved it before. This is shown with the cut rule:

$$\frac{\Gamma \vdash \Delta, P \quad P, \Gamma \vdash \Delta}{\Gamma \vdash \Delta} \text{ (cut)}$$

The  $\Delta, P$  in the first sequent in the hypotheses means that to the right of the  $\vdash$  we have the set consisting of the formula  $P$  plus all the formulas in  $\Delta$ , i.e., if all formulas in  $\Gamma$  are true, then  $P$  or one of the formulas in  $\Delta$  is true. Similarly  $P, \Gamma$  to the left of the  $\vdash$  in the second sequent means the set consisting of the formula  $P$  plus all the formulas in  $\Gamma$ .

We read this rule from the bottom up to make sense of it. Say we want to prove one of the formulas in  $\Delta$  from the formulas in  $\Gamma$ , and we want to make use of a formula  $P$  that we’ve already proved. The fact that we’ve proved  $P$  is shown by the left hypothesis (of course, unless the left hypothesis is itself a basic sequent, then in a completed proof there will be more lines on top of the left hypothesis, showing the actual proof of the sequent). The fact that we are allowed to use  $P$  in the proof of  $\Delta$  is shown in the right hand hypothesis. We continue to build the proof up from there, using  $P$ .

Some other ways of getting formulas to the left of the  $\vdash$  are shown in the rules ( $\neg r$ ) and ( $\Rightarrow r$ ) below.

$\forall x \in S.P(x)$  The two rules for universally quantified formulas are:

$$\frac{P(v), \Gamma \vdash \Delta}{\forall x.P(x), \Gamma \vdash \Delta} \text{ (\forall l)} \quad \frac{\Gamma \vdash \Delta, P(x)}{\Gamma \vdash \Delta, \forall x.P(x)} \text{ (\forall r)}$$

<sup>3</sup>In your Logic and Proof notes, the symbol that divides  $\Gamma$  from  $\Delta$  is  $\Rightarrow$ . However, that conflicts with the use of  $\Rightarrow$  as implication. Thus I will use  $\vdash$ . You will see something similar in Semantics, where it separates assumptions (of the types of variables) from something that they allow you to prove.

<sup>4</sup>I won’t mention iff here: as  $P \Leftrightarrow Q$  is equivalent to  $P \Rightarrow Q \wedge Q \Rightarrow P$ , we don’t need separate rules for it.

In the  $(\forall r)$  rule,  $x$  must not be free in the conclusion.

Now, what's going on here? In the  $(\forall l)$  rule, the  $\forall x.P(x)$  is on the left side of the  $\vdash$ . Thus, we are using it (along with some other formula, those in  $\Gamma$ ) to prove something ( $\Delta$ ). According to the intuition above, in order to *use*  $\forall x.P(x)$ , you can use it with any value, where  $v$  is used to represent that value. In the hypothesis, you see the formula  $P(v)$  to the left of the  $\vdash$ . This is just  $P$  with  $v$  substituted for  $x$ . The use of this corresponds exactly to using the fact that  $P$  is true of any value whatsoever, since we are using it with  $v$ , which is any value of our choice.

In the  $(\forall r)$  rule, the  $\forall x.P(x)$  is on the right side of the  $\vdash$ . Thus, we are proving it. Thus, we need to prove it for a generic  $x$ . This is why the  $\forall x$  is gone in the hypothesis. The  $x$  is still sitting somewhere in the  $P$ , but we're just using it as a plain variable, not assuming anything about it. And this explains the side condition too: "In the  $(\forall r)$  rule,  $x$  must not be free in the conclusion." If  $x$  is not free in the conclusion, this means that  $x$  is not free in the formulas in  $\Gamma$  or  $\Delta$ . That means the only place the  $x$  occurs free in the hypothesis is in  $P$  itself. This corresponds exactly with the requirement that we're proving that  $P$  is true of a generic  $x$ : if  $x$  were free in  $\Gamma$  or  $\Delta$ , we *would* be assuming something about  $x$ , namely that value of  $x$  is the same as the  $x$  used in those formulas.

Note that induction is not mentioned in the rules. This is because the sequent calculus used here just deals with pure logic. In more complicated presentations of logic, it is explained how to define new types via structural induction, and from there you get mechanisms to allow you to do induction.

$\exists x \in S.P(x)$  The two rules for existentially quantified formulas are:

$$\frac{P(x), \Gamma \vdash \Delta}{\exists x.P(x), \Gamma \vdash \Delta} (\exists l) \quad \frac{\Gamma \vdash \Delta, P(v)}{\Gamma \vdash \Delta, \exists x.P(x)} (\exists r)$$

In the  $(\exists l)$  rule,  $x$  must not be free in the conclusion.

In  $(\exists l)$ , we are using  $\exists x.P(x)$ . Thus we cannot assume anything about the value that the formula says exists, so we just use it as  $x$  in the hypothesis. The side condition about  $x$  not being free in the conclusions comes from the requirement not to assume anything about  $x$  (since we don't know what it is). If  $x$  isn't free in the conclusion, then it's not free in  $\Gamma$  or  $\Delta$ . If it were free in  $\Gamma$  or  $\Delta$ , then we would be assuming that the  $x$  used there is the same as the  $x$  we're assuming exists, and this isn't allowed.

In  $(\exists r)$ , we are proving  $\exists x.P(x)$ . Thus we must pick a particular value (call it  $v$ ) and prove  $P$  for that value. The value  $v$  is allowed to contain variables that are free in  $\Gamma$  or  $\Delta$ , since you can set it to anything you want.

$\neg P$  The rules for negation are:

$$\frac{\Gamma \vdash \Delta, P}{\neg P, \Gamma \vdash \Delta} (\neg l) \quad \frac{P, \Gamma \vdash \Delta}{\Gamma \vdash \Delta, \neg P} (\neg r)$$

Let's start with the right rule first. I said that the way to prove  $\neg P$  is to assume  $P$  and derive a contradiction. If  $\Delta$  is the empty set, then this is exactly what this rule says: If there are no formulas to the right hand side of the  $\vdash$ , then this means that the formulas in  $\Gamma$  are inconsistent (that means, they cannot all be true at the same time). This means that you have derived a contradiction. So if  $\Delta$  is the empty set, the hypothesis of the rule says that, assuming  $P$ , you have obtained a contradiction. Thus, if you are absolutely certain about all your other hypotheses, then you can be sure that  $P$  is not true. The best way to understand the rule if  $\Delta$  is not empty is to write out the meaning of the sequents in terms of the meaning of the sequent given by Equation 1 and work out the equivalence of the top and bottom of the rule using the equivalences in your Logic and Proof notes.

The easiest way to understand  $(\neg l)$  is again by using equivalences.

$P \Rightarrow Q$  The two rules for implication are:

$$\frac{\Gamma \vdash \Delta, P \quad Q, \Gamma \vdash \Delta}{P \Rightarrow Q, \Gamma \vdash \Delta} (\Rightarrow l) \quad \frac{P, \Gamma \vdash \Delta, Q}{\Gamma \vdash \Delta, P \Rightarrow Q} (\Rightarrow r)$$

The rule  $(\Rightarrow l)$  is easily understood using the intuitive explanation of how to use  $P \Rightarrow Q$  given above. First, we have to prove  $P$ . This is the left hypothesis. Then we can use  $Q$ , which is what the right hypothesis says.

The right rule  $(\Rightarrow r)$  is also easily understood. In order to prove  $P \Rightarrow Q$ , we assume  $P$ , then use this to prove  $Q$ . This is exactly what the hypothesis says.

$P \wedge Q$  The rules for conjunction are:

$$\frac{P, Q, \Gamma \vdash \Delta}{P \wedge Q, \Gamma \vdash \Delta} (\wedge l) \quad \frac{\Gamma \vdash \Delta, P \quad \Gamma \vdash \Delta, Q}{\Gamma \vdash \Delta, P \wedge Q} (\wedge r)$$

Both of these rules are easily explained by the intuition above. The left rule  $(\wedge l)$  says that when you use  $P \wedge Q$ , you can use  $P$  and  $Q$ . The right rule says that to prove  $P \wedge Q$  you must prove  $P$ , and you must prove  $Q$ . You may wonder why we need separate hypotheses for the two different proofs. We can't just put  $P, Q$  to the right of the  $\vdash$  in a single hypothesis, because that would mean that we're proving one of the other of them (see the meaning of the sequent given in Equation 1). So we need separate hypotheses to make sure that each of  $P$  and  $Q$  has actually been proved.

$P \vee Q$  The rules for disjunction are:

$$\frac{P, \Gamma \vdash \Delta \quad Q, \Gamma \vdash \Delta}{P \vee Q, \Gamma \vdash \Delta} (\vee l) \quad \frac{\Gamma \vdash \Delta, P, Q}{\Gamma \vdash \Delta, P \vee Q} (\vee r)$$

These are also easily understood by the intuitive explanations above. The left rule says that to prove something (namely, one of the formulas in  $\Delta$ ) using  $P \vee Q$ , you need to prove it using  $P$ , then prove it using  $Q$ . The right rule says that in order to prove  $P \vee Q$ , you can prove one or the other. The hypothesis says that you can prove one or the other, because in order to show a sequent  $\Gamma \vdash \Delta$  true, you only need to show that *one* of the formulas in  $\Delta$  is true.