Semantics of Programming Languages

Neel Krishnaswami
University of Cambridge
Department of Computer Science and Technology
Michaelmas 2021

*Notes originally by Peter Sewell and Sam Staton
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syllabus</td>
<td>3</td>
</tr>
<tr>
<td>Learning Guide</td>
<td>4</td>
</tr>
<tr>
<td>Summary of Notation</td>
<td>5</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>8</td>
</tr>
<tr>
<td>2 A First Imperative Language</td>
<td>14</td>
</tr>
<tr>
<td>2.1 Operational Semantics</td>
<td>15</td>
</tr>
<tr>
<td>2.2 Typing</td>
<td>23</td>
</tr>
<tr>
<td>2.3 L1: Collected definition</td>
<td>30</td>
</tr>
<tr>
<td>2.4 Exercises</td>
<td>32</td>
</tr>
<tr>
<td>3 Induction</td>
<td>33</td>
</tr>
<tr>
<td>3.1 Abstract Syntax and Structural Induction</td>
<td>35</td>
</tr>
<tr>
<td>3.2 Inductive Definitions and Rule Induction</td>
<td>41</td>
</tr>
<tr>
<td>3.3 Example proofs</td>
<td>44</td>
</tr>
<tr>
<td>3.4 Exercises</td>
<td>49</td>
</tr>
<tr>
<td>4 Functions</td>
<td>50</td>
</tr>
<tr>
<td>4.1 Abstract syntax up to alpha conversion, and substitution</td>
<td>52</td>
</tr>
<tr>
<td>4.2 Function Behaviour</td>
<td>56</td>
</tr>
<tr>
<td>4.3 Function Typing</td>
<td>60</td>
</tr>
<tr>
<td>4.4 Local Definitions and Recursive Functions</td>
<td>62</td>
</tr>
<tr>
<td>4.5 Implementation</td>
<td>65</td>
</tr>
<tr>
<td>4.6 L2: Collected Definition</td>
<td>68</td>
</tr>
<tr>
<td>4.7 Exercises</td>
<td>71</td>
</tr>
<tr>
<td>5 Data</td>
<td>72</td>
</tr>
<tr>
<td>5.1 Products and sums</td>
<td>72</td>
</tr>
<tr>
<td>5.2 Datatypes and Records</td>
<td>75</td>
</tr>
<tr>
<td>5.3 Mutable Store</td>
<td>77</td>
</tr>
<tr>
<td>5.4 Evaluation Contexts</td>
<td>80</td>
</tr>
<tr>
<td>5.5 L3: Collected definition</td>
<td>81</td>
</tr>
<tr>
<td>5.6 Exercises</td>
<td>85</td>
</tr>
<tr>
<td>6 Subtyping and Objects</td>
<td>86</td>
</tr>
<tr>
<td>6.1 Exercises</td>
<td>92</td>
</tr>
<tr>
<td>7 Concurrency</td>
<td>93</td>
</tr>
<tr>
<td>7.1 Exercises</td>
<td>103</td>
</tr>
<tr>
<td>8 Semantic Equivalence</td>
<td>104</td>
</tr>
<tr>
<td>8.1 Contextual equivalence</td>
<td>109</td>
</tr>
<tr>
<td>8.2 Exercises</td>
<td>109</td>
</tr>
<tr>
<td>9 Epilogue</td>
<td>109</td>
</tr>
<tr>
<td>A Interpreter and type checker for L1 (ML)</td>
<td>112</td>
</tr>
<tr>
<td>B Interpreter and type checker for L1 (Java)</td>
<td>116</td>
</tr>
<tr>
<td>C How to do Proofs</td>
<td>122</td>
</tr>
<tr>
<td>C.1 How to go about it</td>
<td>122</td>
</tr>
<tr>
<td>C.2 And in More Detail</td>
<td>124</td>
</tr>
<tr>
<td>C.2.1 Meet the Connectives</td>
<td>124</td>
</tr>
<tr>
<td>C.2.2 Equivalences</td>
<td>124</td>
</tr>
<tr>
<td>C.2.3 How to Prove a Formula</td>
<td>125</td>
</tr>
<tr>
<td>C.2.4 How to Use a Formula</td>
<td>127</td>
</tr>
<tr>
<td>C.3 An Example</td>
<td>128</td>
</tr>
<tr>
<td>C.3.1 Proving the PL</td>
<td>128</td>
</tr>
<tr>
<td>C.3.2 Using the PL</td>
<td>129</td>
</tr>
<tr>
<td>C.4 Sequent Calculus Rules</td>
<td>129</td>
</tr>
</tbody>
</table>
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.
- **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


Out of print, but available on the web at


Learning Guide

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

  Introduces many of the key topics of the course. There’s a copy on the web at https://www.scss.tcd.ie/Matthew.Hennessy/splexternal2015/resources/sembookWiley.pdf

  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

  An introduction to both operational and denotational semantics.


**Further reading:**

  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.

  - 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/.

  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/atapl/.

**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 (Page 32): 1, 3, 4, 8, 9, 10, 11 (all these should be pretty quick); §3.4 (Page 49): 12, 13.5, 15.
2. §4.7 (Page 71): 18, 19, 20, 21, 22, 23; §5.6 (Page 85): 28; 2003.5.11.

3. §7.1 (Page 103): 37, (39), 40; §6.1 (Page 92): 31, 32, 35; 2003.6.12, further tripos questions

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 67 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.


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 (N. Krishnaswami): These notes are a modification of the notes that Peter Sewell used for the course, 2014–2018. Thanks also to Jean Pichon for comments.

Acknowledgements (P. Sewell): These notes are a modification of the notes that Sam Staton used for the course, 2010–2013. Thanks also to Jean Pichon for comments.

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

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

Finite partial functions

- $\{a_i \mapsto b_i\}$: 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 $\{x \mapsto T\}$
- $\Gamma, \Gamma'$: the finite partial function which is the union of $\Gamma$ and $\Gamma'$
- $\{l_i \mapsto n_i, ..., l_k \mapsto n_k\}$: an L1 or L2 store – the finite partial function mapping each $l_i$ to $n_i$
- $\{l_i \mapsto v_i, ..., l_k \mapsto v_k\}$: an L3 store – the finite partial function mapping each $l_i$ to $v_i$
- $\ell\text{-intref}, ..., x : T, ...$: an L2 type environment
- $\ell\text{-L3}, ..., x : T, ...$: an L3 type environment
- $\{e_1/x_1, ..., e_k/x_k\}$: a substitution – the finite partial function $\{x_1 \mapsto e_1, ..., x_k \mapsto e_k\}$ mapping $x_1$ to $e_1$ etc.

Relations and auxiliary functions

- $\langle e, s \rangle \rightarrow \langle e', s' \rangle$: reduction (or transition) step
- $\langle e, s \rangle \rightarrow^* \langle e', s' \rangle$: reflexive transitive closure of $\rightarrow$
- $\langle e, s \rangle \rightarrow^k \langle e', s' \rangle$: the $k$-fold composition of $\rightarrow$
- $\langle e, s \rangle \rightarrow^\omega$: has an infinite reduction sequence (a unary predicate)
- $\langle e, s \rangle \not\rightarrow$: 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
- $\{e/x\}e'$: the expression resulting from substituting $e$ for $x$ in $e'$
- $\sigma e$: the expression resulting from applying the substituting $\sigma$ to $e$
- $\langle e, s \rangle \downarrow \langle v, s' \rangle$: store $s$ is well-typed with respect to type environment $\Gamma$
- $T <: T'$: type $T$ is a subtype of type $T'$
- $e \approx e'$: semantic equivalence (informal)
- $e \simeq_\Gamma e'$: semantic equivalence at type $T$ with respect to type environment $\Gamma$
- $e \rightarrow a e'$: single thread transition step, labelled with action $a$
Particular sets

- $\mathbb{B} = \{\text{true, false}\}$ the set of booleans
- $\mathbb{L} = \{l_1, l_2, \ldots\}$ the set of locations
- $\mathbb{Z} = \{-1, 0, 1, \ldots\}$ the set of integers
- $\mathbb{N} = \{0, 1, \ldots\}$ the set of natural numbers
- $\mathbb{X} = \{x, y, \ldots\}$ the set of L2 and L3 variables
- $\mathbb{LAB} = \{p, q, \ldots\}$ the set of record labels
- $\mathbb{M} = \{m, m_0, m_1, \ldots\}$ the set of mutex names
- $\mathbb{T}$ the set of all types (in whichever language)
- $\mathbb{T}_{\text{loc}}$ the set of all location types (in whichever language)
- $\mathbb{L}_1$ the set of all L1 expressions
- $\mathbb{TypeEnv}$ the set of all L1 type environments, finite partial functions from $\mathbb{L}$ to $\mathbb{T}_{\text{loc}}$
- $\mathbb{TypeEnv2}$ the set of all L2 type environments, the pairs of a finite partial function from $\mathbb{L}$ to $\mathbb{T}_{\text{loc}}$ and a finite partial function from $\mathbb{X}$ to $\mathbb{T}$

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)
- $T_{\text{loc}} \in \mathbb{T}_{\text{loc}}$ 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
- $\_C[e]$ hole in a context $C$ with $e$ replacing the hole $\_$
1 Introduction

<table>
<thead>
<tr>
<th>Date</th>
<th>Lecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thursday 10 October</td>
<td>1</td>
</tr>
<tr>
<td>Tuesday 15 October</td>
<td>2</td>
</tr>
<tr>
<td>Thursday 17 October</td>
<td>3</td>
</tr>
<tr>
<td>Tuesday 22 October</td>
<td>4</td>
</tr>
<tr>
<td>Thursday 24 October</td>
<td>5</td>
</tr>
<tr>
<td>Tuesday 29 October</td>
<td>6</td>
</tr>
<tr>
<td>Thursday 31 October</td>
<td>7</td>
</tr>
<tr>
<td>Tuesday 5 November</td>
<td>8</td>
</tr>
<tr>
<td>Thursday 7 November</td>
<td>9</td>
</tr>
<tr>
<td>Tuesday 12 November</td>
<td>10</td>
</tr>
<tr>
<td>Thursday 14 November</td>
<td>11</td>
</tr>
<tr>
<td>Tuesday 19 November</td>
<td>12</td>
</tr>
</tbody>
</table>

- Science
- Engineering
- Craft
- Art
- Bodgery
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.

Semantics — What is it?
How to describe a programming language? Need to give:

- the syntax of programs; and
- their semantics (the meaning of programs, or how they behave).
Semantics — What is it?

How to describe a programming language? Need to give:

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

Styles of description:

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

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

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.

What do we use semantics for?

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

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

```
x++ + x++ + x++ + x++
```
Various different approaches have been used for expressing semantics.

**Styles of Semantic Definitions**
- 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*

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

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

What’s this course?

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

**More advanced topics**
- Subtyping and Objects
- Semantic Equivalence
- Concurrency
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.
L1 is an imperative language with store locations (holding integers), conditionals, and while loops. For example, consider the program

\[
L_1 := 0; \\
\text{while } L_1 \geq 1 \text{ do } ( \\
L_2 := L_2 + L_1; \\
L_1 := L_1 + (-1))
\]

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

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 \( 3 + \text{true} \)
- 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 \( b, n, \ell, \alpha, b, e \) 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 \( \text{Config} \), and
• a binary relation \( \rightarrow \subseteq \text{Config} \times \text{Config} \).

The elements of \( \text{Config} \) are often called configurations or states. The relation \( \rightarrow \) is called the transition or reduction relation. We write \( \rightarrow \) infix, so \( c \rightarrow c' \) should be read as ‘state \( c \) can make a transition to state \( c' \)’.

To compare with the automata you saw in Regular Languages and Finite Automata: a transition system is like an NFA \( \varepsilon \) with an empty alphabet (so only \( \varepsilon \) 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.

• \( \rightarrow^* \) is the reflexive transitive closure of \( \rightarrow \), so \( c \rightarrow^* c' \) iff there exist \( k \geq 0 \) and \( c_0, \ldots, c_k \) such that \( c = c_0 \rightarrow c_1 \rightarrow \ldots \rightarrow c_k = c' \).
• \( \not\rightarrow \) is a unary predicate (a subset of \( \text{Config} \)) defined by \( c \not\rightarrow \) iff \( \exists c'. c \not\rightarrow c' \).

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 \( L \) 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

\[
\langle e, s \rangle \rightarrow \langle e', s' \rangle
\]

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 \( \{(a_1, b_1), \ldots, (a_n, b_n)\} \), often written \( \{a_1 \mapsto b_1, \ldots, a_n \mapsto b_n\} \), for which

• \( \forall i \in \{1, \ldots, n\}. a_i \in A \) (the domain is a subset of \( A \))
• \( \forall i \in \{1, \ldots, n\}. b_i \in B \) (the range is a subset of \( B \))
• \( \forall i, j \in \{1, \ldots, n\}. i \neq j \Rightarrow a_i \neq a_j \) (\( f \) 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.
Transitions are single computation steps. For example we will have:

\[ \langle l := 2 + 1, \{l \mapsto 3\} \rangle \]
\[ \rightarrow \langle l := 2 + 3, \{l \mapsto 3\} \rangle \]
\[ \rightarrow \langle l := 5, \{l \mapsto 3\} \rangle \]
\[ \rightarrow \langle \text{skip}, \{l \mapsto 5\} \rangle \]
\[ \not\rightarrow \]

want to keep on until we get to a value \( v \), an expression in

\[ V = \mathbb{B} \cup \mathbb{Z} \cup \{\text{skip}\} \]

Say \( \langle e, s \rangle \) is \textit{stuck} if \( e \) is not a value and \( \langle e, s \rangle \not\rightarrow \). For example

\[ 2 + \text{true} \]

will be stuck.

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

Now define the behaviour for each construct of \( L_1 \) by giving some rules that (together) define a transition relation \( \rightarrow \).

How to read these? The rule \( \text{(op +)} \) 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 \textit{only} be instantiated by integers, not by arbitrary expressions.

The rule \( \text{(op1)} \) says that for any instantiation of \( e_1, e'_1, e_2, s, s' \) (i.e. any three expressions and two stores), \textit{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.
If we want to find the possible sequences of transitions of \((2 + 3) + (6 + 7), \emptyset\) ... look for derivations of transitions. 
(you might think the answer should be 18 – but we want to know what this definition says happens)

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 \(\langle e, s \rangle \rightarrow \langle e', s' \rangle\) then there isn’t such a transition.

L1 Semantics (3 of 4) – store and sequencing

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

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

(assign2) \(\langle e, s \rangle \rightarrow \langle e', s' \rangle\)

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

(seq2) \(\langle e_1, s \rangle \rightarrow \langle e'_1, s' \rangle\)

\(\langle e_1; e_2, s \rangle \rightarrow \langle e'_1; e'_2, s' \rangle\)
Example

\[
\begin{align*}
(l := 3; l \rightarrow 0} & \rightarrow (\text{skip}; l \rightarrow 3)} \\
& \rightarrow (l \rightarrow 3) \\
& \rightarrow (3, l \rightarrow 3)
\end{align*}
\]

\[
(l := 3; l := l, \{l \rightarrow 0}\} \rightarrow ?
\]

\[
(15+l, \emptyset) \rightarrow ?
\]

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

(1) \(\text{if } \text{true } \text{then } e_2 \text{ else } e_3, s \rightarrow e_2, s\)

(2) \(\text{if } \text{false } \text{then } e_2 \text{ else } e_3, s \rightarrow e_3, s\)

(3) \(e_1, s \rightarrow e_1', s'\)

([if]

\[
\text{if } e_1 \text{ then } e_2 \text{ else } e_3, s \rightarrow \text{if } e_1' \text{ then } e_2 \text{ else } e_3, s'
\]

(while)

\[
\text{while } e_1 \text{ do } e_2, s \rightarrow \text{if } e_1 \text{ then } (e_2; \text{while } e_1 \text{ do } e_2) \text{ else skip, } s
\]

That concludes our definition of L1. The full definition is collected on page 30.

Determinacy

Theorem 1 (L1 Determinacy) If \((e, s) \rightarrow (e_1, s_1)\) and \((e, s) \rightarrow (e_2, s_2)\) then \((e_1, s_1) = (e_2, s_2)\).

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 \((e, s) \rightarrow (e_1, s_1)\) and \((e, s) \rightarrow (e_2, s_2)\) then \((e_1, s_1) = (e_2, s_2)\)."
L1 implementation

Many possible implementation strategies, including:

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

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

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

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

L1 implementation

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

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

type loc = string

type store = (loc * int) list

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

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

\[
\text{lookup} : \text{store} \times \text{loc} \to \text{int option}
\]

\[
\text{update} : \text{store} \times (\text{loc} \times \text{int}) \to \text{store option}
\]

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

The single-step function

Now define the single-step function

\[
\text{reduce} : \text{expr} \times \text{store} \to (\text{expr} \times \text{store}) \text{ option}
\]

which takes a configuration \((e, s)\) and returns either \text{NONE}, if \(e, s \not\rightarrow\), or \text{SOME } (e', s'), if it has a transition \(e, s \rightarrow (e', s')\).

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

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

Contrast this code with the semantic rules given earlier.

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.

\[
\text{reduce (Assign (l,e), s)} =
\begin{cases}
\text{case e of} & \\
\text{Integer n =>} & \\
\text{(case update (s, (l,n)) of} & \\
\text{SOME s' => SOME(Skip, s')} & \\
\text{| NONE => NONE)} & \\
\text{| | =>} & \\
\text{(case reduce (e,s) of} & \\
\text{SOME (e',s') =>} & \\
\text{SOME(Assign (l,e'), s')} & \\
\text{| NONE => NONE ) )}
\end{cases}
\]

The many-step evaluation function

Now define the many-step evaluation function

\[\text{evaluate: expr*store -> (expr*store) option}\]

which takes a configuration \((e, s)\) and returns the \((e', s')\) such that

\[\langle e, s \rangle \xrightarrow{\tau} \langle e', s' \rangle\], if there is such, or does not return.

\[
\text{fun evaluate (e,s) =}
\begin{cases}
\text{case reduce (e,s) of} & \\
\text{NONE => (e,s)} & \\
\text{| SOME (e',s') => evaluate (e',s')}
\end{cases}
\]

The full interpreter code is in Appendix A, and you can also download it from the course website, in the file \texttt{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 \texttt{l1.java}.

The Java Implementation

Quite different code structure:

- the ML groups together all the parts of each algorithm, into the \texttt{reduce, infertype, and prettyprint} functions;
- the Java groups together everything to do with each clause of the abstract syntax, in the \texttt{IfThenElse, Assign, etc. classes.}
L1 is a simple language, but it nonetheless involves several language design choices.

**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:

\[
((l := 1; 0) + (l := 2; 0), \{l \mapsto 0\}) \xrightarrow{\Delta} (0, \{l \mapsto 0\})
\]

For right-to-left evaluation, replace \((\text{op1})\) and \((\text{op2})\) by

\[
(\text{op1b}) \\
\frac{\langle e_2, s \rangle \rightarrow \langle e'_2, s' \rangle}{\langle e_1 \text{ op } e_2, s \rangle \rightarrow \langle e_1 \text{ op } e'_2, s' \rangle}
\]

\[
(\text{op2b}) \\
\frac{\langle e_1, s \rangle \rightarrow \langle e'_1, s' \rangle}{\langle e_1 \text{ op } v, s \rangle \rightarrow \langle e'_1 \text{ op } v, s' \rangle}
\]

In this language (call it L1b)

\[
((l := 1; 0) + (l := 2; 0), \{l \mapsto 0\}) \xrightarrow{\Delta} (0, \{l \mapsto 1\})
\]

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

One could also *underspecify*, taking both \((\text{op1})\) and \((\text{op1b})\) rules. That language doesn’t have the Determinacy property.

**Language design 2. Assignment results**

Recall

\[
(\text{assign1}) \quad \langle \ell := n, s \rangle \rightarrow \langle \text{skip}, s + \{\ell \mapsto n\} \rangle \quad \text{if } \ell \in \text{dom}(s)
\]

\[
(\text{seq1}) \quad \langle \text{skip}; e_2, s \rangle \rightarrow \langle e_2, s \rangle
\]

So

\[
(l := 1; l := 2, \{l \mapsto 0\}) \xrightarrow{\Delta} \langle \text{skip}; l := 2, \{l \mapsto 1\} \rangle \\
\xrightarrow{\star} \langle \text{skip}, \{l \mapsto 2\} \rangle
\]

We’ve chosen \(\ell := n\) 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:

\[
(\text{assign1'}) \quad \langle \ell := n, s \rangle \rightarrow \langle n, s + \{\ell \mapsto n\} \rangle \quad \text{if } \ell \in \text{dom}(s)
\]

\[
(\text{seq1'}) \quad \langle v; e_2, s \rangle \rightarrow \langle e_2, s \rangle
\]

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

**Language design 3. Store initialization**

Recall that

\[
(\text{deref}) \quad \langle \ell, s \rangle \rightarrow \langle n, s \rangle \quad \text{if } \ell \in \text{dom}(s) \text{ and } s(\ell) = n
\]

\[
(\text{assign1}) \quad \langle \ell := n, s \rangle \rightarrow \langle \text{skip}, s + \{\ell \mapsto n\} \rangle \quad \text{if } \ell \in \text{dom}(s)
\]

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

Instead, could

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

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?)
Type systems are 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.

**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):

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

$\ell_1: \text{intref} \vdash \text{if \ !\ell_1 \geq \ 3 \ then \ !\ell_1 \ else \ 3} : \text{int}$

$\{\} \vdash \text{3 + false} : T$ for any $T$

$\{\} \vdash \text{if \ true \ then \ 3 \ else \ false} : \text{int}$

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.

**Types for L1**

Types of expressions:

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

Types of locations:

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

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

Let $\Gamma$ range over $\text{TypeEnv}$, the finite partial functions from locations $L$ to $T_{\text{loc}}$. Notation: write a $\Gamma$ as $\ell_1: \text{intref}, ..., \ell_k: \text{intref}$ instead of $\{\ell_1 \mapsto \text{intref}, ..., \ell_k \mapsto \text{intref}\}$.

- concretely, $T = \{\text{int, bool, unit}\}$ and $T_{\text{loc}} = \{\text{intref}\}$.
- in this language, there is only one type in $T_{\text{loc}}$, so a $\Gamma$ can be thought of as just a set of locations. (Later, $T_{\text{loc}}$ will be more interesting.)
Defining the type judgement \[ \Gamma \vdash e : T \] (1 of 3)

- (int) \[ \Gamma \vdash n : \text{int} \text{ for } n \in \mathbb{Z} \]
- (bool) \[ \Gamma \vdash b : \text{bool} \text{ for } b \in \{\text{true, false}\} \]
- (op \(+\)) \[ \Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \quad \Gamma \vdash e_1 + e_2 : \text{int} \]
- (op \(\geq\)) \[ \Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \quad \Gamma \vdash e_1 \geq e_2 : \text{bool} \]
- (if) \[ \Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : T \quad \Gamma \vdash e_3 : T \quad \Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : T \]

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.

- (op \(+\)) \[
\frac{
\Gamma \vdash e_1 : \text{int} \\
\Gamma \vdash e_2 : \text{int}
}{
\Gamma \vdash e_1 + e_2 : \text{int}
}
\]

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

- (op \(+\)) \[
\frac{
\Gamma \vdash e_1 : \text{int} \\
\Gamma \vdash e_2 : \text{int}
}{
\Gamma \vdash e_1 + e_2 : \text{int}
}
\]

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

- (bool) \[ \{\} \vdash \text{false} : \text{bool} \]
- (int) \[ \{\} \vdash 2 : \text{int} \]
- \[ \{\} \vdash \text{if false then 2 else 3 + 4} : \text{int} \]

where \( \nabla \) is

- (int) \[ \{\} \vdash 3 : \text{int} \]
- (int) \[ \{\} \vdash 4 : \text{int} \]
- \[ \{\} \vdash 3 + 4 : \text{int} \]

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

\[ \Gamma \vdash e : T \]

(s/**/kip) \[ \Gamma \vdash \text{skip} : \text{unit} \]

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

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

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.

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.

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.

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 \texttt{l1.ml} contains also an implementation of a type inference algorithm for L1 – take
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 $\{\text{intref}\}$. 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 $(\ell, \text{intref})$.

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
  | => NONE)
| infertype gamma (While (e1, e2)) = (case (infertype gamma e1, infertype gamma e2) of
  (SOME bool, SOME unit) => SOME unit
  | => NONE)
The Type Inference Algorithm – If

\[ \text{infertype gamma (If (e1,e2,e3))} = \begin{cases} \text{infertype gamma e1,} \\ \text{infertype gamma e2,} \\ \text{infertype gamma e3) of} \\ \text{(SOME bool, SOME t2, SOME t3) =>} \\ \text{if t2=t3 then SOME t2 else NONE} \\ \text{else => NONE) } \]

\[ \Gamma \vdash e_1 : \text{bool} \]
\[ \Gamma \vdash e_2 : T \]
\[ \Gamma \vdash e_3 : T \]
\[ \Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : T \]

The Type Inference Algorithm – Deref

\[ \text{infertype gamma (Deref l)} = \begin{cases} \text{lookup (gamma,l) of} \\ \text{SOME intref => SOME int} \\ \text{NONE => NONE) } \]

\[ \text{(deref) } \Gamma(\ell) = \text{intref} \]
\[ \Gamma \vdash \text{!}\ell : \text{int} \]

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

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

\[ \text{let val skip = ()} \]
\[ \text{and l1 = ref n1} \]
\[ \text{and l2 = ref n2} \]
\[ \text{...} \]
\[ \text{and lk = ref nk} \]
\[ \text{in} \]
\[ \text{e} \]
\[ \text{end;} \]

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

(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.”

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 $b \in B = \{\text{true}, \text{false}\}$
Integers $n \in \mathbb{Z} = \{\ldots, -1, 0, 1, \ldots\}$
Locations $\ell \in L = \{l, l_0, l_1, l_2, \ldots\}$

Operations $op ::= + | \geq$

Expressions

$$e ::= n \mid b \mid e_1 \ op \ e_2 \mid \text{if } e_1 \ \text{then } e_2 \ \text{else } e_3$$

Operations

Expressions

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 $L$ to $\mathbb{Z}$. Say values $v$ are expressions from the grammar $v ::= b \mid n \mid \text{skip}$.

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

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

$$(\text{op1}) \langle e_1, s \rangle \rightarrow \langle e'_1, s' \rangle$$

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

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

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

$$(\text{assign2}) \langle e, s \rangle \rightarrow \langle e', s' \rangle$$

$$\langle \ell := e, s \rangle \rightarrow \langle \ell := e', s' \rangle$$

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

$$(\text{seq2}) \langle e_1, s \rangle \rightarrow \langle e'_1, s' \rangle$$

$$\langle e_1; e_2, s \rangle \rightarrow \langle e'_1; e_2, s' \rangle$$

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

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

$$(\text{if3}) \langle e_1, s \rangle \rightarrow \langle e'_1, s' \rangle$$

$$\langle \text{if } e_1 \ \text{then } e_2 \ \text{else } e_3, s \rangle \rightarrow \langle \text{if } e'_1 \ \text{then } e_2 \ \text{else } e_3, s' \rangle$$

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

30
Typing

Types of expressions:

\[ T ::= \text{int} \mid \text{bool} \mid \text{unit} \]

Types of locations:

\[ T_{\text{loc}} ::= \text{intref} \]

Write \( T \) and \( T_{\text{loc}} \) for the sets of all terms of these grammars.

Let \( \Gamma \) range over \( \text{TypeEnv} \), the finite partial functions from locations \( L \) to \( T_{\text{loc}} \).

\[
\begin{align*}
\text{(int)} & \quad \Gamma \vdash n : \text{int} \quad \text{for } n \in \mathbb{Z} \\
\text{(bool)} & \quad \Gamma \vdash b : \text{bool} \quad \text{for } b \in \{\text{true}, \text{false}\} \\
\text{(op +)} & \quad \Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \quad \Gamma \vdash e_1 + e_2 : \text{int} \\
\text{(op ≥)} & \quad \Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \quad \Gamma \vdash e_1 \geq e_2 : \text{bool} \\
\text{(if)} & \quad \Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : T \quad \Gamma \vdash e_3 : T \quad \Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : T \\
\text{(assign)} & \quad \Gamma(\ell) = \text{intref} \quad \Gamma \vdash e : \text{int} \quad \Gamma \vdash \ell := e : \text{unit} \\
\text{(deref)} & \quad \Gamma(\ell) = \text{intref} \quad \Gamma \vdash !\ell : \text{int} \\
\text{(skip)} & \quad \Gamma \vdash \text{skip} : \text{unit} \\
\text{(seq)} & \quad \Gamma \vdash e_1 : \text{unit} \quad \Gamma \vdash e_2 : T \quad \Gamma \vdash e_1 ; e_2 : T \\
\text{(while)} & \quad \Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{unit} \quad \Gamma \vdash \text{while } e_1 \text{ do } e_2 : \text{unit}
\end{align*}
\]
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 \( L_1 \).

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

Exercise 3 ★ Give full derivations of the first four reduction steps of the \( \langle e, s \rangle \) of the first \( L_1 \) example on Slide 22.

Exercise 4 ★ Adapt the implementation code to correspond to the two rules (op1b) and (op2b) on Slide 44. 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 45. Give some test cases that distinguish between the original and the new semantics.

Exercise 6 ★★ Fix the \( L_1 \) 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 32 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 45? 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 \( L_1 \) implementation and the \( L_1 \) semantics agree on the order of evaluation for operators and sequencing.
3 Induction

Key concepts in this chapter:
- Structural induction
- Rule induction

Induction

We’ve stated several ‘theorems’, but how do we know they are true?
Intuition is often wrong – we need proof.
Use proof process also for strengthening our intuition about subtle language features, and for debugging definitions – it helps you examine all the various cases.
Most of our definitions are inductive. To prove things about them, we need the corresponding induction principles.

Three forms of induction
Prove facts about all natural numbers by mathematical induction.
Prove facts about all terms of a grammar (e.g. the L1 expressions) by structural induction.
Prove facts about all elements of a relation defined by rules (e.g. the L1 transition relation, or the L1 typing relation) by rule induction.
We shall see that all three boil down to induction over certain trees.

Principle of Mathematical Induction
For any property \( \Phi(x) \) of natural numbers \( x \in \mathbb{N} = \{0, 1, 2, \ldots\} \), to prove
\[
\forall x \in \mathbb{N}. \Phi(x)
\]
it’s enough to prove
\[
\Phi(0) \text{ and } \forall x \in \mathbb{N}. \Phi(x) \Rightarrow \Phi(x + 1),
\]
i.e.
\[
\left( \Phi(0) \land (\forall x \in \mathbb{N}. \Phi(x) \Rightarrow \Phi(x + 1)) \right) \Rightarrow \forall x \in \mathbb{N}. \Phi(x).
\]

(NB, the natural numbers include 0)
For example, to prove

\[ (\Phi(0) \land (\forall x \in \mathbb{N}. \Phi(x) \Rightarrow \Phi(x+1))) \Rightarrow \forall x \in \mathbb{N}. \Phi(x) \]

**Theorem 8** \[1 + 2 + \ldots + x = 1/2 \ast x \ast (x + 1)\]

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

**Proof** We prove \( \forall x. \Phi(x) \), where

\[ \Phi(x) \overset{\text{def}}{=} (1 + 2 + \ldots + x = 1/2 \ast x \ast (x + 1)) \]

by mathematical induction.

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

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

\( \Phi(0) \) is \((1 + \ldots + 0 = 1/2 \ast 0 \ast (0 + 1))\), 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)*

\[ (1 + 2 + \ldots + (k + 1)) = 1/2 \ast (k + 1) \ast ((k + 1) + 1) \]

Now, the left hand side is

\[ (1 + 2 + \ldots + (k + 1)) = (1 + 2 + \ldots + k) + (k + 1) \]

(rearranging)

\[ = (1/2 \ast k \ast (k + 1)) + (k + 1) \]

(using \( \Phi(k) \))

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

and the right hand side is (rearranging)

\[ 1/2 \ast (k + 1) \ast ((k + 1) + 1) = 1/2 \ast (k \ast (k + 1) + (k + 1) + 1 \ast (k + 1)) = 1/2 \ast k \ast (k + 1) + 1/2 \ast ((k + 1) + k + 1) = 1/2 \ast k \ast (k + 1) + (k + 1) \]

which is equal to the LHS.
Abstract Syntax and Structural Induction

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

**Theorem 1 (Determinacy)**

If $e, s \rightarrow (e_1, s_1)$ and $e, s \rightarrow (e_2, s_2)$ then $(e_1, s_1) = (e_2, s_2)$.

First, don’t forget the elided universal quantifiers.

**Theorem 1 (Determinacy)**

For all $e, s, e_1, s_1, e_2, s_2$, if $e, s \rightarrow (e_1, s_1)$ and $e, s \rightarrow (e_2, s_2)$ then $(e_1, s_1) = (e_2, s_2)$.

Then, have to pay attention to what an expression is. Recall we said:

- $e ::= n \mid b \mid e \ op \ e \mid if \ e \ then \ e \ else \ e$
- $\ell ::= e \mid !\ell$
- skip $\mid e; e$
- while $e \ do \ e$

defining a set of expressions.

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

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

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

Parentheses are only used for disambiguation – they are not part of the grammar. $1 + 2 = (1 + 2) = (((1 + 2))) = (((1 + (2 + 2))))}$
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 $\mathbb{B} \cup \mathbb{Z} \cup \{(!) \ast \mathbb{L}\} \cup \{\text{skip}\} = \{\text{true, false, skip}\} \cup \{-1, 0, 1, \ldots\} \cup \{!, l_1, l_2, \ldots\}$.
- unary nodes labelled by $\{l :=, l_1 :=, l_2 :=, \ldots\}$
- binary nodes labelled by $\{+, \geq, ;, \text{while_do}_1\}$
- ternary nodes labelled by $\{\text{if_then_else}_1\}$

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.

<table>
<thead>
<tr>
<th>Principle of Structural Induction (for abstract syntax)</th>
</tr>
</thead>
<tbody>
<tr>
<td>For any property $\Phi(e)$ of expressions $e$, to prove</td>
</tr>
<tr>
<td>$\forall e \in L_1, \Phi(e)$</td>
</tr>
<tr>
<td>it’s enough to prove for each tree constructor $c$ (taking $k \geq 0$ arguments)</td>
</tr>
<tr>
<td>that if $\Phi$ holds for the subtrees $e_1, \ldots, e_k$ then $\Phi$ holds for the tree</td>
</tr>
<tr>
<td>$c(e_1, \ldots, e_k)$, i.e.</td>
</tr>
<tr>
<td>${\forall c, \forall e_1, \ldots, e_k. (\Phi(e_1) \land \cdots \land \Phi(e_k)) \Rightarrow \Phi(c(e_1, \ldots, e_k))} \Rightarrow \forall e. \Phi(e)$</td>
</tr>
<tr>
<td>where the tree constructors (or node labels) $c$ are $n$, true, false, $!$, skip,</td>
</tr>
<tr>
<td>$l :=, \text{while_do}_1$, if_then_else$_1$, etc.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>nullary: $\Phi(\text{skip})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\forall b \in {\text{true, false}}. \Phi(b)$</td>
</tr>
<tr>
<td>$\forall n \in \mathbb{Z}. \Phi(n)$</td>
</tr>
<tr>
<td>$\forall \ell \in \mathbb{L}. \Phi(\ell)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>unary: $\Phi(c)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\forall \ell \in \mathbb{L}. \exists e. \Phi(e) \Rightarrow \Phi(\ell := e)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>binary: $\Phi(c)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\forall \ast, \forall e_1, e_2. (\Phi(e_1) \land \Phi(e_2)) \Rightarrow \Phi(e_1 \ast e_2)$</td>
</tr>
<tr>
<td>$\forall e_1, e_2. (\Phi(e_1) \land \Phi(e_2)) \Rightarrow \Phi(\text{while}\ e_1 \text{ do } e_2)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ternary: $\Phi(c)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\forall e_1, e_2, e_3. (\Phi(e_1) \land \Phi(e_2) \land \Phi(e_3)) \Rightarrow \Phi(\text{if}\ e_1 \text{ then } e_2 \text{ else } e_3)$</td>
</tr>
</tbody>
</table>

(See how this comes directly from the grammar)

---

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 $\Phi$.  

---

<table>
<thead>
<tr>
<th>Proving Determinacy (Outline)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Theorem 1 (Determinacy)</strong> If $\langle e, s \rangle \rightarrow \langle e_1, s_1 \rangle$ and</td>
</tr>
<tr>
<td>$\langle e, s \rangle \rightarrow \langle e_2, s_2 \rangle$ then $\langle e_1, s_1 \rangle = \langle e_2, s_2 \rangle$ .</td>
</tr>
</tbody>
</table>

Take |
| $\Phi(e) \overset{\text{def}}{=} \forall s, e', s', e'', s''. |
| $\langle (e, s) \rightarrow \langle e', s' \rangle \land (e, s) \rightarrow \langle e'', s'' \rangle \Rightarrow \langle e', s' \rangle = \langle e'', s'' \rangle \rangle$ |

and show $\forall e \in L_1, \Phi(e)$ by structural induction.
Lemma 9: Values don’t reduce

Lemma 9: For all $e \in L_1$, if $e$ is a value then
\[ \forall a. \exists e', s'. \langle e, s \rangle \rightarrow \langle e', s' \rangle. \]

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

\[ \square \]
Inversion

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

Lemma 10 (Inversion for \( \rightarrow \)) if \( (e, s) \rightarrow (\hat{e}, \hat{s}) \) then either

1. \((\text{op} +)\) there exists \(n_1, n_2,\) and \(n\) such that \(e = n_1 + n_2, \hat{e} = n,\)
   \(\hat{s} = s,\) and \(n = n_1 + n_2\) (NB watch out for the two different \(+\) signs), or

2. \((\text{op1})\) there exists \(e_1, e_2,\) \(\text{op}\), and \(e'_1\) such that \(e = e_1 \text{ op } e_2,\)
   \(\hat{e} = e'_1 \text{ op } e_2,\) and \((e_1, s) \rightarrow (e'_1, s'),\) or

3. ...

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

1. ...

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

\[
\begin{array}{c}
+ \\
+ \ 3 \\
(!l \ 2)
\end{array}
\]
Theorem 1 (Determinacy) If $(e, s) \rightarrow (e_1, s_1)$ and $(e, s) \rightarrow (e_2, s_2)$ then $(e_1, s_1) = (e_2, s_2)$.

Proof Take

$$\Phi(e) \overset{\text{def}}{=} \forall s, e', s', e'', s''.(\langle e, s \rangle \rightarrow \langle e', s' \rangle \land \langle e, s \rangle \rightarrow \langle e'', s'' \rangle) \Rightarrow \langle e', s' \rangle = \langle e'', s'' \rangle$$

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

Cases $\text{skip}, b, n$. For $e$ of these forms there are no rules with a conclusion of the form $(e, ...) \rightarrow (..., ...)$ so the left hand side of the implication cannot hold, so the implication is true.

Case $\ell$. Take arbitrary $s, e', s', e'', s''$ such that $(\ell, s) \rightarrow (e', s') \land (\ell, s) \rightarrow (e'', s'')$.

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

$$\ell \in \text{dom}(s) \quad e' = s(\ell) \quad e'' = s(\ell) \quad s' = s \quad s'' = s$$

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 $(\ell := e, s) \rightarrow (e', s') \land (\ell := e, s) \rightarrow (e'', s'')$.

It’s handy to have this lemma:

**Lemma 12** For all $e \in L_1$, if $e$ is a value then $\forall s. \exists e'. s'(e, s) \rightarrow (e', s')$.

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 $(e, s) \rightarrow (e', s')$ for $e$ one of $n, b, \text{skip}$. \qed

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

case $(\ell := e, s) \rightarrow (e', s')$ 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 $(\ell := n, s) \rightarrow (e', s'')$ 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 $(e', s') = (e'', s'')$ as required.

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

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

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

case $(\ell := e, s) \rightarrow (e'', s'')$ is an instance of (assign2). Then for some $e'_2$ we have $(e, s) \rightarrow (e'_2, s'')(**)$ and $e'' = (\ell := e'_2)$. Now, by the induction hypothesis $\Phi(e)$, (*) and (**) we have $(e'_1, s') = (e'_2, s'')$, so $(e', s') = (\ell := e'_1, s') = (\ell := e'_2, s'') = (e'', s'')$ 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 $(e_1 \text{ op } e_2, s) \rightarrow (e', s') \land (e_1 \text{ op } e_2, s) \rightarrow (e'', s'')$. 

39
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 \( \langle e_1 + e_2, s \rangle \rightarrow \langle e', s' \rangle \) is an instance of \((op+)\) and \( \langle e_1 + e_2, s \rangle \rightarrow \langle e'', s'' \rangle \) 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 \( \langle e_1 \geq e_2, s \rangle \rightarrow \langle e', s' \rangle \) is an instance of \((op\geq)\) and \( \langle e_1 \geq e_2, s \rangle \rightarrow \langle e'', s'' \rangle \) 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 \( b = (n_1 + n_2) \), and \( s' = s = s'' \).

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

Then for some \( e_1' \) and \( e_1'' \) we have \( \langle e_1, s \rangle \rightarrow \langle e_1', s' \rangle \) \((*)\), \( \langle e_1, s \rangle \rightarrow \langle e_1'', s'' \rangle \) \((**)\), \( e' = e_1' \ op \ e_2 \), 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 \( \langle e', s' \rangle = \langle e_1' \ op \ e_2, s' \rangle = \langle e_1'' \ op \ e_2, s'' \rangle \) as required.

**Case** \( \langle e_1 \ op \ e_2, s \rangle \rightarrow \langle e', s' \rangle \) is an instance of \((op2)\) and \( \langle e_1 \ op \ e_2, s \rangle \rightarrow \langle e'', s'' \rangle \) 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 \( \langle e_1; e_2, s \rangle \rightarrow \langle e', s' \rangle \land \langle e_1; e_2, s \rangle \rightarrow \langle e'', s'' \rangle \).

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 \( \langle e_1, s \rangle \rightarrow \langle e_1', s' \rangle \) \((*)\), \( \langle e_1, s \rangle \rightarrow \langle e_1'', s'' \rangle \) \((**)\), \( e' = e_1' \land e_2 \), and \( e'' = e_1'' \land e_2 \).

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

**Case** \( \text{while } e_1 \ do \ e_2 \). Suppose \( \Phi(e_1) \) and \( \Phi(e_2) \).

Take arbitrary \( s, e', s', e'', s'' \) such that \( \langle \text{while } e_1 \ do \ e_2, s \rangle \rightarrow \langle e', s' \rangle \land \langle \text{while } e_1 \ do \ e_2, s \rangle \rightarrow \langle e'', s'' \rangle \).

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

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

Take arbitrary \( s, e', s', e'', s'' \) such that \( \langle \text{if } e_1 \ then \ e_2 \ else \ e_3, s \rangle \rightarrow \langle e', s' \rangle \land \langle \text{if } e_1 \ then \ e_2 \ else \ e_3, s \rangle \rightarrow \langle e'', s'' \rangle \).

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...!)

3.2 Inductive Definitions and Rule Induction

<table>
<thead>
<tr>
<th>Slide 86</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inductive Definitions and Rule Induction</strong></td>
</tr>
<tr>
<td>How to prove facts about all elements of the L1 typing relation or the L1 reduction relation, e.g. Progress or Type Preservation?</td>
</tr>
<tr>
<td><strong>Theorem 2 (Progress)</strong></td>
</tr>
<tr>
<td><strong>Theorem 3 (Type Preservation)</strong></td>
</tr>
<tr>
<td>What does $(e, s) \rightarrow (e', s')$ really mean?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Slide 87</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inductive Definitions</strong></td>
</tr>
<tr>
<td>We defined the transition relation $(e, s) \rightarrow (e', s')$ and the typing relation $\Gamma \vdash e : T$ by giving some rules, eg</td>
</tr>
<tr>
<td>[(\text{op }+) \quad (n_1 + n_2, s) \rightarrow (n, s) \quad \text{if } n = n_1 + n_2]</td>
</tr>
<tr>
<td>[(\text{op1}) \quad (e_1, s) \rightarrow (e'_1, s') \rightarrow (e_1 \text{ op } e_2, s) \rightarrow (e_1 \text{ op } e_2, s')]</td>
</tr>
<tr>
<td>[(\text{op }+) \quad \Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \quad \Gamma \vdash e_1 + e_2 : \text{int}]</td>
</tr>
<tr>
<td>What did we actually mean?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Slide 88</th>
</tr>
</thead>
<tbody>
<tr>
<td>These relations are just normal set-theoretic relations, written in infix notation.</td>
</tr>
<tr>
<td>For the transition relation:</td>
</tr>
<tr>
<td>• Start with $A = L_1 \ast \text{store} \ast L_1 \ast \text{store}$.</td>
</tr>
<tr>
<td>• Write $\rightarrow \subseteq A$ infix, e.g. $(e, s) \rightarrow (e', s')$ instead of $(e, s, e', s') \in \rightarrow$.</td>
</tr>
<tr>
<td>For the typing relation:</td>
</tr>
<tr>
<td>• Start with $A = \text{TypeEnv} \ast L_1 \ast \text{types}$.</td>
</tr>
<tr>
<td>• Write $\vdash \subseteq A$ mixfix, e.g. $\Gamma \vdash e : T$ instead of $(\Gamma, e, T) \in \vdash$.</td>
</tr>
</tbody>
</table>
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 \((\text{op} +)\) and \((\text{op}1)\) we take all values of \(n_1, n_2, s, n\) (satisfying \(n = n_1 + n_2\)) and of \(e_1, e_2, s, e'_1, e'_2\).

Note the last has a premise that is not itself derivable, but nonetheless this is a legitimate instance of \((\text{op}1)\).

Now a derivation of a transition \(\langle e, s \rangle \rightarrow \langle e', s' \rangle\) or typing judgment \(\Gamma \vdash e : T\) is a finite tree such that each step is a concrete rule instance.

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

**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 \(\{a | \Phi(a)\}\) is closed under the rules, ie for each concrete rule instance

\[h_1 \ldots h_k\]

if \(\Phi(h_1) \land \ldots \land \Phi(h_k)\) then \(\Phi(c)\).

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

$$\frac{\Phi(h_1) \land \cdots \land \Phi(h_k) \land h_1 \in S_R \land \cdots \land h_k \in S_R}{\Phi(c)}$$

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

**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 $(e, s) \rightarrow (e', s')$.

**Proof**

Take $\Phi(\Gamma, e, T) \overset{\text{def}}{=} \forall s. \text{dom}(\Gamma) \subseteq \text{dom}(s) \Rightarrow \text{value}(e) \lor (\exists e', s'. (e, s) \rightarrow (e', s'))$

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 $\vdash$.

**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

$$\frac{\Phi(h_1) \land \cdots \land \Phi(h_k) \land h_1 \in S_R \land \cdots \land h_k \in S_R}{\Phi(c)}$$

Instantiating to the L1 typing rules, have to show:

- **(int)** $\forall \Gamma, n. \Phi(\Gamma, n, \text{int})$
- **(deref)** $\forall \Gamma, \ell. \Phi(\Gamma) = \text{intref} \Rightarrow \Phi(\Gamma, \ell, \text{int})$
- **(op +)** $\forall \Gamma, e_1, e_2. (\Phi(\Gamma, e_1, \text{int}) \land \Phi(\Gamma, e_2, \text{int}) \land \Gamma \vdash e_1 : \text{int} \land \Gamma \vdash e_2 : \text{int})$
  $\Rightarrow \Phi(\Gamma, e_1 + e_2, \text{int})$
- **(seq)** $\forall \Gamma, e_1, e_2, T. (\Phi(\Gamma, e_1, \text{unit}) \land \Phi(\Gamma, e_2, T) \land \Gamma \vdash e_1 : \text{unit} \land \Gamma \vdash e_2 : T)$
  $\Rightarrow \Phi(\Gamma, e_1; e_2, T)$
- etc.

Having proved those 10 things, consider an example

$\Gamma \vdash (\ell + 2) + 3\text{int}$. To see why $\Phi(\Gamma, (\ell + 2) + 3, \text{int})$ holds:

$$\frac{\Gamma \vdash \ell : \text{int}}{\Gamma \vdash (\ell + 2) : \text{int}} \quad \frac{\Gamma \vdash (\ell + 2) : \text{int}}{\Gamma \vdash (\ell + 2) + 3\text{int}}$$

$$\frac{\Gamma \vdash (\ell + 2) + 3\text{int}}{\Gamma \vdash (\ell + 2) + 3\text{int}}$$
Which Induction Principle to Use?

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

For completeness, observe the following:

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

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. \( e ::= \ldots | e + e \)) to be a rule

\[
\frac{e}{e + e}
\]

3.3 Example proofs

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

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

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.

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—
   e.g. proofs that type inference is decidable (for fancier type systems)
**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 $\langle e, s \rangle \rightarrow (e', s')$.

**Proof** Take

$$
\Phi(\Gamma, e, T) \equiv \forall s. \text{dom}(\Gamma) \subseteq \text{dom}(s) \Rightarrow \text{value}(e) \lor (\exists e', s'. \langle e, s \rangle \rightarrow (e', s'))
$$

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

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

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 $\forall s. \text{dom}(\Gamma) \subseteq \text{dom}(s) \Rightarrow ...$, so consider an arbitrary $s$ and assume $\text{dom}(\Gamma) \subseteq \text{dom}(s)$.

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

**Case** (bool) similar.

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

$$
\begin{align*}
\text{(op +)} & \quad \Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \\
& \quad \Gamma \vdash e_1 + e_2 : \text{int}
\end{align*}
$$

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 $\text{value}(e_1 + e_2) \lor (\exists e', s'. \langle e_1 + e_2, s \rangle \rightarrow (e', s'))$.

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

By (*) one of the following holds.

**case** $\exists e_1', s'. \langle e_1, s \rangle \rightarrow (e_1', s')$.

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

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

**case** $\exists e_2', s'. \langle e_2, s \rangle \rightarrow (e_2', s')$.

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

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

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

**Lemma 13** 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 $\Phi'(\Gamma, e, T) = ((\text{value}(e) \land T = \text{int}) \Rightarrow \exists n \in \mathbb{Z}. e = n)$.

**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 $\Gamma \vdash e : T$).

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+) $(e_1 + e_2, s) \rightarrow (n, s)$ where $n = n_1 + n_2$, so we are done.

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

Case (if). Recall the rule

\begin{align*}
\Gamma \vdash e_1 : \text{bool} \\
\Gamma \vdash e_2 : T \\
\Gamma \vdash e_3 : T
\end{align*}

\[(\text{if}) \quad \Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : T\]

Suppose $\Phi(\Gamma, e_1, \text{bool})$ (*1), $\Phi(\Gamma, e_2, T)$ (*2), $\Phi(\Gamma, e_3, T)$ (*3), $\Gamma \vdash e_1 : \text{bool}$ (*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 $\text{if } e_1 \text{ then } e_2 \text{ else } e_3$.

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

Case $\exists e_1', s'. \langle e_1, s \rangle \rightarrow \langle e_1', s' \rangle$.

Then by (if3) $\langle e, s \rangle \rightarrow (\text{if } e_1' \text{ then } e_2 \text{ else } e_3, s)$, so we are done.

Case $e_1$ is a value.

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

**Lemma 14** For all $\Gamma, e, T$. if $\Gamma \vdash e : T$ and $e$ is a value, then $T = \text{int} \Rightarrow \exists n \in \mathbb{Z}, e = n, T = \text{bool} \Rightarrow \exists b \in \{\text{true, false}\}. e = b, \text{ and } T = \text{unit} \Rightarrow e = \text{skip}$.

**Proof** By rule induction – details omitted.

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

Case $b = \text{true}$. Use (if1).

Case $b = \text{false}$. Use (if2).

Case (deref). Recall the rule

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

(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 $\Gamma(\ell) = \text{intref}$ 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) $\langle \ell, s \rangle \rightarrow \langle n, s \rangle$.

Cases (assign), (skip), (seq), (while). Left as an exercise.
Theorem 3 (Type Preservation) If $\Gamma \vdash e : T$ and $\text{dom}(\Gamma) \subseteq \text{dom}(s)$ and $(e, s) \rightarrow (e', s')$ then $\Gamma \vdash e' : T$ and $\text{dom}(\Gamma) \subseteq \text{dom}(s')$.

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

Lemma 16 If $(e, s) \rightarrow (e', s')$ then $\text{dom}(s') = \text{dom}(s)$.

Proof Rule induction on derivations of $(e, s) \rightarrow (e', s')$. 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 $\text{dom}(s + (\ell \mapsto n)) = \text{dom}(s)$.

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

Proof by rule induction on derivations of $(e, s) \rightarrow (e', s')$. Take $\Phi(e, s, e', s') = \forall \Gamma, T . (\Gamma \vdash e : T \land \text{dom}(\Gamma) \subseteq \text{dom}(s)) \Rightarrow \Gamma \vdash e' : T$.

Case (op+). Recall

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

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

$$(\text{op1}) \quad (e_1, s) \rightarrow (e_1', s') \quad (e_1 \text{ op } e_2, s) \rightarrow (e_1' \text{ op } e_2, s')$$

Suppose $\Phi(e_1, s, e_1', s')$ (*) and $(e_1, s) \rightarrow (e_1', s')$. 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 $\text{dom}(\Gamma) \subseteq \text{dom}(s)$ (**).

\text{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 $\Gamma \vdash e_1' + e_2 : T$.

\text{case op $\geq$}. Similar.

\text{Case s (op2)} (deref), (assign1), (assign2), (seq1), (seq2), (if1), (if2), (if3), (while).

Left as exercises.

\text{□}

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

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

\text{□}

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.

\text{□}
### Proving Progress

**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 $(e, s) \rightarrow (e', s')$.

**Proof**

Take

$$\Phi(\Gamma, e, T) \overset{\text{def}}{=} \forall s. \text{dom}(\Gamma) \subseteq \text{dom}(s) \Rightarrow \text{value}(e) \lor (\exists e', s'. (e, s) \rightarrow (e', s'))$$

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 $\vdash$.

---

**Principle of Rule Induction (variant form):** to prove $\Phi(a)$ for all $a$ in the set $S_R$ defined by the rules, it’s enough to prove that for each rule instance $h_1 \ldots h_k$ if $\Phi(h_1) \land \ldots \land \Phi(h_k) \land h_1 \in S_R \land \ldots \land h_k \in S_R$ then $\Phi(e)$.

Instantiating to the L1 typing rules, have to show:

- $\text{int}$
  - $\forall \Gamma, n. \Phi(\Gamma, n, \text{int})$
  - $\text{deref}$
  - $\forall \Gamma, \ell. \Gamma(\ell) \rightarrow \text{intref} \Rightarrow \Phi(\Gamma, \ell, \text{int})$
- $\text{op} +$
  - $\forall \Gamma, e_1, e_2. (\Phi(\Gamma, e_1, \text{int}) \land \Phi(\Gamma, e_2, \text{int}) \land \Gamma \vdash e_2 : \text{int} \land \Gamma \vdash e_1 : \text{int})$
  - $\Rightarrow \Phi(\Gamma, e_1 + e_2, \text{int})$
- $\text{seq}$
  - $\forall \Gamma, e_1, e_2, T. (\Phi(\Gamma, e_1, \text{unit}) \land \Phi(\Gamma, e_2, T) \land \Gamma \vdash e_1 : \text{unit} \land \Gamma \vdash e_2 : T)$
  - $\Rightarrow \Phi(\Gamma, e_1; e_2, T)$
- etc.

---

**Case** ($\text{op} +$).

Recall the rule

$$\Gamma \vdash e_1 : \text{int}$$

$$\Gamma \vdash e_2 : \text{int}$$

$$\Gamma \vdash e_1 + e_2 : \text{int}$$

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

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

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

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

$$\exists (e', s'). (e_1 + e_2, s) \rightarrow (e', s').$$
Using $\Phi(\Gamma, e_1, \text{int})$ and $\Phi(\Gamma, e_2, \text{int})$ we have:

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

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

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

\[
\text{(op +)} : \langle n_1 + n_2, s \rangle \rightarrow \langle n, s \rangle \quad \text{if } n = n_1 + n_2
\]

**Lemma 17** 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 $e = n$.

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

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

---

**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 $e_1 \ op \ e_2, e_1; e_2, \text{while} \ e_1 \ do \ e_2,$ and if $e_1$ then $e_2$ else $e_3$.

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

**Exercise 13.5 ★** Flesh out the statements of Inversion for the operational semantics and type system. Prove them by rule induction (this needs a trivial use of rule induction, without relying on the induction hypothesis).

**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).
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.
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.
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.

#### Variable shadowing

```java
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, e.g., 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.

#### Alpha conversion

In expressions `fn x:T ⇒ e` the `x` is a **binder**.

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

\[
\int_0^1 x + x^2 dx = \int_0^1 y + y^2 dy
\]

#### Alpha conversion – free and bound occurrences

In a bit more detail (but still informally):

Say an occurrence of `x` in an expression `e` is **free** if it is not inside any `(fn x:T ⇒ ...)`. 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)x)`

All the other occurrences of `x` are **bound** by the closest enclosing `fn x:T ⇒ ...`

Note that in `fn x:int ⇒ 2` the `x` is not an occurrence. Likewise, in `fn x:int ⇒ x + 2` the left `x` is not an occurrence; here the right `x` is an occurrence that is bound by the left `x`.  

52
Sometimes it is handy to draw in the binding:

### Alpha conversion – Binding examples

\[
\begin{align*}
\text{fn } x: \text{int} & \Rightarrow x + 2 \\
\text{fn } x: \text{int} & \Rightarrow x + z \\
\text{fn } y: \text{int} & \Rightarrow y + z \\
\text{fn } z: \text{int} & \Rightarrow z + z \\
\text{fn } x: \text{int} & \Rightarrow (\text{fn } x: \text{int} \Rightarrow x + 2)
\end{align*}
\]

### Alpha Conversion – The Convention

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

For example:

\[
\begin{align*}
\text{fn } x: \text{int} & \Rightarrow x + z = \text{fn } y: \text{int} \Rightarrow y + z \\
\text{fn } z: \text{int} & \Rightarrow z + z
\end{align*}
\]

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

### Abstract Syntax up to Alpha Conversion

\[
\begin{align*}
\text{fn } x: \text{int} & \Rightarrow x + z = \text{fn } y: \text{int} \Rightarrow y + z \neq \text{fn } z: \text{int} \Rightarrow z + z
\end{align*}
\]

Start with naive abstract syntax trees:

---

Slide 116

---

add pointers (from each \[ x \] node to the closest enclosing \[ \text{fn } x: T \Rightarrow \] node);
remove names of binders and the occurrences they bind

---

Slide 117

---
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 nodes you have to count out to get to its binder.

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 $e$.

\[
\begin{align*}
fv(x) &= \{x\} \\
fv(e_1 \ op \ e_2) &= fv(e_1) \cup fv(e_2) \\
fv(fn \ x:T \Rightarrow e) &= fv(e) - \{x\}
\end{align*}
\]

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

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

\[
\begin{align*}
fv(x + y) &= \{x,y\} \\
fv(fn \ x:int \Rightarrow x + y) &= \{y\} \\
fv(x + (fn \ x:int \Rightarrow x + y)) &= \{x,y\}
\end{align*}
\]

Full definition of \(fv(e)\) is by recursion on the structure of \(e\):

\[
\begin{align*}
fv(x) &= \{x\} \\
fv(fn \ x:T \Rightarrow e) &= fv(e) - \{x\} \\
fv(e_1 \ op \ e_2) &= fv(e_1) \cup fv(e_2) \\
fv(if \ e_1 \ then \ e_2 \ else \ e_3) &= fv(e_1) \cup fv(e_2) \cup fv(e_3) \\
fv(b) &= \{\} \\
fv(skip) &= \{\} \\
fv(\ell := e) &= fv(e) \\
fv(!\ell) &= \{\} \\
fv(e_1; e_2) &= fv(e_1) \cup fv(e_2) \\
fv(while \ e_1 \ do \ e_2) &= fv(e_1) \cup fv(e_2)
\end{align*}
\]

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

**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

\[
\begin{align*}
\{3/x\}(x \geq x) &= (3 \geq 3) \\
\{3/x\}(fn \ x:int \Rightarrow x + y) &= (fn \ x:int \Rightarrow x + y)3 \\
\{y + 2/x\}(fn \ y:int \Rightarrow x + y) &= fn \ z:int \Rightarrow (y + 2) + z
\end{align*}
\]

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...

**Substitution – Definition**

Defining that:

\[
\begin{align*}
\{e/z\}x &= e \quad \text{if } x = z \\
\{e/z\}x &= x \quad \text{otherwise}
\end{align*}
\]

\[
\begin{align*}
\{e/z\}(fn \ x:T \Rightarrow e_1) &= fn \ x:T \Rightarrow (\{e/z\}e_1) \quad \text{if } x \neq z (*) \\
\text{and } x \notin fv(e) (*)
\end{align*}
\]

\[
\begin{align*}
\{e/z\}(e_1 \ e_2) &= (\{e/z\}e_1)(\{e/z\}e_2)
\end{align*}
\]

... if (*) is not true, we first have to pick an alpha-variant of \(fn \ x:T \Rightarrow e_1\) to make it so (always can)
Substitution – Example Again

\[
\{y + 2/x\}(\text{fn } y:\text{int} \Rightarrow x + y) = \{y + 2/x\}(\text{fn } y':\text{int} \Rightarrow x + y') \text{ renaming}
\]

\[
= \text{fn } y':\text{int} \Rightarrow \{y + 2/x\}(x + y') \text{ as } y' \neq x \text{ and } y' \notin \text{fv}(y + 2)
\]

\[
= \text{fn } y':\text{int} \Rightarrow \{y + 2/x\}x + \{y + 2/x\}y'
\]

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

Simultaneous substitution

A substitution \(\sigma\) is a finite partial function from variables to expressions.

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

A definition of \(\sigma\) is given in the notes.

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

\[
\text{dom}\{e_1/x_1, \ldots, e_k/x_k\} = \{x_1, \ldots, x_k\}
\]

\[
\text{ran}\{e_1/x_1, \ldots, e_k/x_k\} = \{e_1, \ldots, e_k\}
\]

Define the application of simultaneous substitution to a term by:

\[
\sigma x = \begin{cases} \sigma(x) & \text{if } x \in \text{dom}(\sigma) \\ x & \text{otherwise} \end{cases}
\]

\[
\sigma(\text{fn } x:T \Rightarrow e) = \begin{cases} \text{fn } x:T \Rightarrow (\sigma e) & \text{if } x \notin \text{dom}(\sigma) \text{ and } x \notin \text{fv}(\text{ran}(\sigma)) (*) \end{cases}
\]

\[
\sigma(e_1 e_2) = (\sigma e_1)(\sigma e_2)
\]

\[
\sigma n = n
\]

\[
\sigma(e_1 \text{ op } e_2) = \sigma(e_1) \text{ op } \sigma(e_2)
\]

\[
\sigma(\text{if } e_1 \text{ then } e_2 \text{ else } e_3) = \text{if } \sigma(e_1) \text{ then } \sigma(e_2) \text{ else } \sigma(e_3)
\]

\[
\sigma(b) = b
\]

\[
\sigma(\text{skip}) = \text{skip}
\]

\[
\sigma(\ell := e) = \ell := \sigma(e)
\]

\[
\sigma(!\ell) = !\ell
\]

\[
\sigma(e_1; e_2) = \sigma(e_1); \sigma(e_2)
\]

\[
\sigma(\text{while } e_1 \text{ do } e_2) = \text{while } \sigma(e_1) \text{ do } \sigma(e_2)
\]

4.2 Function Behaviour

Function Behaviour

Consider the expression

\[
e = (\text{fn } x:\text{unit} \Rightarrow (i := 1); x) \ (i := 2)
\]

then

\[
\{e, \{i := 0\}\} \rightarrow^* \{\text{skip}, \{i := ???\}\}
\]
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.

\[
e = (fn \ x:unit \Rightarrow (l := 1); x)(l := 2)
\]

\[
\{e, \{l = 0\}\} \rightarrow (fn \ x:unit \Rightarrow (l := 1); x)\{l = 2\}
\]
\[
\rightarrow (l := 1); skip, \{l = 2\}
\]
\[
\rightarrow (skip; skip), \{l = 1\}
\]
\[
\rightarrow (skip), \{l = 1\}
\]

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 \( v ::= b \mid n \mid skip \mid fn \ x:T \Rightarrow e \)

(app1) \[
\begin{align*}
\langle e_1, s \rangle & \rightarrow \langle e'_1, s' \rangle \\
\langle e_1, e_2, s \rangle & \rightarrow \langle e'_1, e'_2, s' \rangle
\end{align*}
\]

(app2) \[
\begin{align*}
\langle e_2, s \rangle & \rightarrow \langle e'_2, s' \rangle \\
v, e_2, s & \rightarrow v, e'_2, s'
\end{align*}
\]

(fn) \[
\langle (fn \ x:T \Rightarrow e), s \rangle \rightarrow \langle \{v/x\} e, s \rangle
\]

L2 Call-by-value – reduction examples

\[
(fn \ x:int \Rightarrow fn \ y:int \Rightarrow x + y) (3 + 4), s
\]
\[
= (fn \ x:int \Rightarrow fn \ y:int \Rightarrow x + y) (3 + 4), s
\]
\[
\rightarrow (fn \ x:int \Rightarrow fn \ y:int \Rightarrow x + y) (7), s
\]
\[
\rightarrow (7 + 5), s
\]
\[
\rightarrow (12), s
\]

\[
(fn \ f:int \Rightarrow int \Rightarrow f \ 3) \ (fn \ x:int \Rightarrow (1 + 2) + x)
\]

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

\[
e = (\texttt{fn } x: \texttt{unit} \Rightarrow (l := 1); x) (l := 2)
\]

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

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.
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 $\text{fn}$-term, replace all occurrences of the formal parameter in the $\text{fn}$-term by the argument. Allow reduction inside lambdas.

$$(\text{fn } x : \text{int} \Rightarrow 2 + 2) \rightarrow (\text{fn } x : \text{int} \Rightarrow 4)$$

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

Function Behaviour. Choice 4: Normal-order reduction

Leftmost, outermost variant of full beta.

- What will $(\text{fn } x : \text{unit} \Rightarrow \text{skip})$ (while true do skip) do in the different semantics?
- What about $(\text{fn } x : \text{unit} \Rightarrow \text{skip})$ ($\ell := \ell + 1$)?

Back to CBV (from now on).
4.3 Function Typing

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

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

Type environments \( \Gamma \) are now pairs of a \( \Gamma_{\text{loc}} \) (a partial function from \( L \) to \( T_{\text{loc}} \) as before) and a \( \Gamma_{\text{var}} \), a partial function from \( X \) to \( T \).

For example, we might have \( \Gamma_{\text{loc}} = l_1 : \text{intref} \) and \( \Gamma_{\text{var}} = x : \text{int}, y : \text{bool} \rightarrow \text{int} \).

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

For typing functions:

\[
\begin{align*}
\text{(fn)} & \quad \Gamma, x : T \vdash e : T' \\
& \quad \Gamma \vdash \text{fn}\ x : T \Rightarrow e : T' \rightarrow T'
\end{align*}
\]

\[
\begin{align*}
\text{(app)} & \quad \Gamma \vdash e_1 : T \rightarrow T' \\
& \quad \Gamma \vdash e_2 : T \\
& \quad \Gamma \vdash e_1\ e_2 : T'
\end{align*}
\]

Typing functions – Example

\[
\begin{align*}
\text{(var)} & \quad \Gamma \vdash x : T & \text{if } \Gamma(x) = T \\
\text{(int)} & \quad \Gamma \vdash x + 2 : \text{int} & \text{if } \Gamma(x) = \text{int} \\
\text{(op +)} & \quad \Gamma \vdash x + (\text{fn}\ x : \text{bool} \Rightarrow \text{if}\ x\ \text{then}\ 3\ \text{else}\ 4) : \text{int} & \text{if } x \in \text{dom}(\Gamma_{\text{var}})
\end{align*}
\]

\[
\begin{align*}
\text{(app)} & \quad \Gamma \vdash (\text{fn}\ x : \text{int} \Rightarrow x)\ \text{if}\ x\ \text{then}\ 3\ \text{else}\ 4)\ (2) : \text{int} & \text{if } x \in \text{dom}(\Gamma_{\text{var}})
\end{align*}
\]

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

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

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 \textit{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 \textit{Types}.
Another example:

\[
\begin{array}{lll}
\text{(int)} & \text{L:intref, x:unit ⊢ l:int} & \text{(asn)} \\
\text{(asn)} & \text{L:intref, x:unit ⊢ (l := 1):unit} & \text{(seq)} \\
\text{(seq)} & \text{L:intref, x:unit ⊢ x:unit} & \text{(fn)} \\
\text{(fn)} & \text{L:intref ⊢ (fn x:unit ⇒ (l := 1); x):unit → unit} & \text{(int)} \\
\text{(app)} & \text{L:intref ⊢ (fn x:unit ⇒ (l := 1); x) (l := 2):unit} & \\
\end{array}
\]

Properties of Typing

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

Theorem 18 (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 \( \langle e, s \rangle \rightarrow \langle e', s' \rangle \).

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

Theorem 19 (Type Preservation) If \( e \) closed and \( \Gamma \vdash e : T \) and \( \text{dom}(\Gamma) \subseteq \text{dom}(s) \) and \( h e, s i \rightarrow h e' \), \( s' i \) then \( \Gamma \vdash e' : T \) and \( e' \) closed and \( \text{dom}(\Gamma) \subseteq \text{dom}(s') \).

Proving Type Preservation

Theorem 19 (Type Preservation) If \( e \) closed and \( \Gamma \vdash e : T \) and \( \text{dom}(\Gamma) \subseteq \text{dom}(s) \) and \( \langle e, s \rangle \rightarrow \langle e', s' \rangle \) then \( \Gamma \vdash e' : T \) and \( e' \) closed and \( \text{dom}(\Gamma) \subseteq \text{dom}(s') \).

Taking

\[
\Phi(e, s, e', s') = \\
\forall \Gamma, T. \\
\Gamma \vdash e : T \land \text{closed}(e) \land \text{dom}(\Gamma) \subseteq \text{dom}(s) \\
\Rightarrow \\
\Gamma \vdash e' : T \land \text{closed}(e') \land \text{dom}(\Gamma) \subseteq \text{dom}(s')
\]

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

To prove this one uses:

Lemma 20 (Substitution) If \( \Gamma \vdash e : T \) and \( \Gamma, x:T \vdash e':T' \) with \( x \notin \text{dom}(\Gamma) \) then \( \Gamma \vdash \{ e/x \} e' : T' \).

Determinacy and type inference properties also hold.

Normalization

Theorem 21 (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 \( \langle e, \{} \rangle \rightarrow \langle e_1, \{} \rangle \rightarrow \langle e_2, \{} \rangle \rightarrow ... \)

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).
4.4 Local Definitions and Recursive Functions

Local definitions

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

```
let val x:T = e1 in e2 end
```

This `x` is a binder, binding any free occurrences of `x` in `e2`.

Can regard just as syntactic sugar:

```
let val x:T = e1 in e2 end
(fn x:T => e2)e1
```

Local definitions – derived typing and reduction rules (CBV)

```
let val T = e1 in e2 end
(fn x:T => e2)e1
```

\[\Gamma \vdash e_1 : T \quad \Gamma, x:T \vdash e_2 : T' \quad \Gamma \vdash \text{let } x:T = e_1 \text{ in } e_2 \text{ end}: T'\]

(\text{let1})

\[\langle e_1, s \rangle \rightarrow \langle e_1', s' \rangle\]

\[\langle \text{let } x:T = e_1 \text{ in } e_2 \text{ end}, s \rangle \rightarrow \langle \text{let } x:T = e_1' \text{ in } e_2 \text{ end}, s' \rangle\]

(\text{let2})

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 + let val x:int = x in (x + 2) end = x + let val y:int = x in (y + 2) end
```

Recursive definitions – first attempt

How about

```
x = (fn y:int => if y \geq 1 then y + (x(y + 1)) else 0)
```

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

Could add something like this:

```
e ::= \ldots \mid \text{let val } rec \ x:T = e \text{ in } e' \text{ end}
```

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

```
let val rec x:int => int =
(fn y:int => if y \geq 1 then y + (x(y + 1)) else 0)
in x 3 end
```

62
But...

What about

\[
\text{let val rec } \ x = (x, x) \ \text{in } x \text{ end}
\]

Have some rather weird things, eg

\[
\text{let val rec } \ x : \text{int list} = 3 :: x \ \text{in } x \text{ end}
\]

does that terminate? if so, is it equal to

\[
\text{let val rec } \ x : \text{int list} = 3 :: 3 :: x \ \text{in } x \text{ end}
\]

\[
\text{does that terminate?}
\]

In a CBN language, it is reasonable to allow this kind of thing, as will only compute as much as needed. In a CBV language, would usually disallow, allowing recursive definitions only of functions...

### Recursive Functions

So, specialize the previous let val rec construct to

\[
\begin{align*}
T &= T_1 \rightarrow T_2 & \text{recursion only at function types} \\
e &= \text{fn } y : T_1 \Rightarrow e_1 & \text{and only of function values}
\end{align*}
\]

\[
e ::= \ldots | \text{let val rec } x : T_1 \rightarrow T_2 = (\text{fn } y : T_1 \Rightarrow e_1) \ \text{in } e_2 \ \text{end}
\]

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

\[
\Gamma, x : T_1 \rightarrow T_2, y : T_1 \vdash e_1 : T_2 \\
\Gamma, x : T_1 \rightarrow T_2 \vdash e_2 : T
\]

\[
\Gamma \vdash \text{let rec fn } x : T_1 \rightarrow T_2 = (\text{fn } y : T_1 \Rightarrow e_1) \ \text{in } e_2 \ \text{end} : T
\]

Concrete syntax: In ML can write let fun \(f(x : T_1) : T_2 = e_1 \ \text{in } e_2 \ \text{end}\), or even let fun \(f(x) = e_1 \ \text{in } e_2 \ \text{end}\), for

\[
\text{let val rec } f : T_1 \rightarrow T_2 = \text{fn } x : T_1 \Rightarrow e_1 \ \text{in } e_2 \ \text{end}.
\]

### Recursive Functions – Semantics

\[
\begin{align*}
&((\text{fn } y : T_1 \Rightarrow \text{let rec } x : T_1 \rightarrow T_2 = (\text{fn } y : T_1 \Rightarrow e_1) \ \text{in } e_2 \ \text{end})/x) e_2, s)
\end{align*}
\]
For example:

```plaintext
let val rec \( x: \text{int} \rightarrow \text{int} = \)
        (fn y:int \Rightarrow if \ y \geq 1 \ then \ y + (x(y + 1)) \ else \ 0) \in
    x 3 \end

\rightarrow (\text{letrecfn})
(fn y:int \Rightarrow
    let val rec \( x: \text{int} \rightarrow \text{int} = \)
    (fn y:int \Rightarrow if \ y \geq 1 \ then \ y + (x(y + 1)) \ else \ 0) \in
        if \ y \geq 1 \ then \ y + (x(y + 1)) \ else \ 0
    end ) 3
\rightarrow (\text{app})
let val rec \( x: \text{int} \rightarrow \text{int} = \)
        (fn y:int \Rightarrow if \ y \geq 1 \ then \ y + (x(y + 1)) \ else \ 0) \in
    if \ 3 \geq 1 \ then \ 3 + (x(3 + 1)) \ else \ 0
end
\rightarrow (\text{letrecfn})
if \ 3 \geq 1 \ then
    3 + ((fn y:int \Rightarrow
        let val rec \( x: \text{int} \rightarrow \text{int} = \)
        (fn y:int \Rightarrow if \ y \geq 1 \ then \ y + (x(y + 1)) \ else \ 0) \in
            if \ y \geq 1 \ then \ y + (x(y + 1)) \ else \ 0
        end ) (3 + 1))
else
    0
\rightarrow ...
```

As a test case, we apply it to the function (fn z:int \Rightarrow if \ z \geq 3 then (if \ 3 \geq z then 0 else 1) else 1), which is 0 for argument 3 and 1 elsewhere.
More Syntactic Sugar

Do we need $e_1; e_2$?
No: Could encode by $e_1; e_2 \leadsto (\text{fn } y:\text{unit} \Rightarrow e_2)e_1$

Do we need while $e_1$ do $e_2$?
No: could encode by while $e_1$ do $e_2 \leadsto$

$$\text{let val rec } w:\text{unit} \rightarrow \text{unit } =$$
$$\text{fn } y:\text{unit} \Rightarrow \text{if } e_1 \text{ then } (e_2; (w \text{ skip})) \text{ else skip in }$$
$$w \text{ skip end}$$

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

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 mosmlex and mosmlyac.

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

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)
Implementation – Substitution

\[ \text{subst : expr -> int -> expr -> expr} \]

\[ \text{subst e 0 e'} \] 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])

\[
\begin{align*}
\text{fun subst e n (Var n1) = if n=n1 then e else Var n1} \\
& | \text{subst e n (Fn(t,e1)) = Fn(t,subst e (n+1) e1)} \\
& | \text{subst e n (App(e1,e2)) = App(subst e n e1,subst e n e2)} \\
& | \text{subst e n (Let(t,e1,e2))} \\
& = \text{Let (t,subst e n e1,subst e (n+1) e2)} \\
& | \text{subst e n (Letrecfn (tx,ty,e1,e2))} \\
& = \text{letrecfn (tx,ty,subst e (n+2) e1,subst e (n+1) e2)} \\
& | \ldots
\end{align*}
\]

If \( e' \) represents a closed term \( \text{fn} \ x: T \Rightarrow e'_1 \) then \( e' = \text{Fn}(t,e_1) \) for \( t \) and \( e_1' \) representing \( T \) and \( e'_1 \). If also \( e \) represents a closed term \( e \) then \( \text{subst e 0 e1'} \) represents \( \{e/x\}e'_1 \).

Implementation – CBV reduction

\[
\text{reduce (App (e1,e2),s) = (case e1 of} \\
\text{Fn (t,e) => (if (is_value e2) then} \\
\text{SOME (subst e2 0 e,s)} \quad \text{else} \\
\text{(case reduce (e2,s) of} \\
\text{SOME(e2',s') => SOME(App (e1,e2'),s') } \\
\text{| NONE => NONE))} \\
\text{| _ => (case reduce (e1,s) of} \\
\text{SOME (e1',s')=>SOME(App(e1',e2),s')} \quad \text{| NONE => NONE )}}
\]

Implementation – Type Inference

\[
\begin{align*}
\text{type typeEnv} &= (loc * type_loc) list * type_expr list \\
\text{inftype gamma (Var n) = nth (#2 gamma) n} \\
\text{inftype gamma (Fn (t,e))} &= (case inftype (#1 gamma, t::(#2 gamma)) e of} \\
& \quad \text{SOME t' => SOME (func(t,t'))} \\
& \quad \text{| NONE => NONE } \\
\text{inftype gamma (App (e1,e2))} &= (case (inftype gamma e1, inftype gamma e2) of} \\
& \quad (SOME (func(t1,t1')), SOME t2) => \text{if t1=t2 then SOME t1' else NONE } \\
& \quad | _ => \text{NONE )}
\end{align*}
\]
Implementation – Closures

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

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

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

Aside: Small-step vs Big-step Semantics

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

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

$\langle n, s \rangle \Downarrow \langle n, s \rangle$  $\langle n_1, s' \rangle$  $\langle e_2, s' \rangle \Downarrow \langle n_2, s'' \rangle$

$n = n_1 + n_2$

(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 $b \in B = \{true, false\}$
Integers $n \in Z = \{\ldots, -1, 0, 1, \ldots\}$
Locations $l \in L = \{l_0, l_1, l_2, \ldots\}$
Variables $x \in X$ for a set $X = \{x, y, z, \ldots\}$

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

Types

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

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

Expressions

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

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

Operational Semantics

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

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

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

$$(op1) \quad \langle e_1, s \rangle \rightarrow \langle e'_1, s' \rangle$$

$$(op2) \quad \langle e_2, s \rangle \rightarrow \langle e'_2, s' \rangle$$

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

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

$$(\text{assign2}) \quad \langle e, s \rangle \rightarrow \langle e', s' \rangle$$

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

$$(\text{seq2}) \quad \langle e_1, s \rangle \rightarrow \langle e'_1, s' \rangle$$

$$\langle e_1; e_2, s \rangle \rightarrow \langle e'_1; e'_2, s' \rangle$$

68
Typing

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

- \( \text{(int)} \) \( \Gamma \vdash n : \text{int} \) for \( n \in \mathbb{Z} \)
- \( \text{(bool)} \) \( \Gamma \vdash b : \text{bool} \) for \( b \in \{ \text{true}, \text{false} \} \)
- \( \text{(op +)} \) \( \Gamma \vdash e_1 : \text{int} \) \( \Gamma \vdash e_2 : \text{int} \) \( \Gamma \vdash e_1 + e_2 : \text{int} \)
- \( \text{(op ≥)} \) \( \Gamma \vdash e_1 : \text{int} \) \( \Gamma \vdash e_2 : \text{int} \) \( \Gamma \vdash e_1 \geq e_2 : \text{bool} \)
- \( \text{(if)} \) \( \Gamma \vdash e_1 : \text{bool} \) \( \Gamma \vdash e_2 : \mathbb{T} \) \( \Gamma \vdash e_3 : \mathbb{T} \)
  - \( \Gamma \vdash \text{if } e_1 \quad \text{then } e_2 \quad \text{else } e_3 : \mathbb{T} \)
- \( \text{(assign)} \) \( \Gamma(\ell) = \text{intref} \) \( \Gamma \vdash e : \text{int} \)
  - \( \Gamma \vdash \ell := e : \text{unit} \)
- \( \text{(deref)} \) \( \Gamma(\ell) = \text{intref} \)
  - \( \Gamma \vdash !\ell : \text{int} \)
(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) \[ \frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{unit}}{\Gamma \vdash \text{while } e_1 \text{ do } e_2 : \text{unit}} \]

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

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

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

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

(let rec fn) \[ \frac{\Gamma, x : T_1 \rightarrow T_2, y : T_1 \vdash e_1 : T_2 \quad \Gamma, x : T_1 \rightarrow T_2 \vdash e_2 : T}{\Gamma \vdash \text{let val rec } x : T_1 \rightarrow T_2 = (\text{fn } y : T_1 \Rightarrow e_1) \text{ in } e_2 \text{ end} : T} \]
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. \( !0 \)
5. \( \text{while } !0 \geq y \ \text{do } !0 := x \)

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. \( \text{fn } x: \text{int} \Rightarrow y \)
2. \( \text{fn } x: \text{int} \Rightarrow (\text{fn } x: \text{bool} \Rightarrow \text{if } x \ \text{then } 3 \ \text{else } 4) \true \)

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 51).

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

Exercise 23 ★★ Prove Lemma 20 (Substitution).

Exercise 24 ★★ Prove Theorem 19 (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

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 $T_1 \times T_2$ 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 $\text{int} \rightarrow (\text{int} \times \text{bool})$. 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 l1.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 163

Design choices:

• pairs, not arbitrary tuples – have $\text{int} \times (\text{int} \times \text{int})$ and $(\text{int} \times \text{int}) \times \text{int}$, but (a) they’re different, and (b) we don’t have $(\text{int} \times \text{int}) \times \text{int}$. 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 $\text{fn} (x, y) \Rightarrow e$. A full language should allow the latter, as it often makes for much more elegant code.

• don’t have $#e e'$ (couldn’t typecheck!).
```

72
We have chosen left-to-right evaluation order for consistency.

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;
```
Sums – type annotations

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

Why do we have these type annotations?

To maintain the unique typing property. Otherwise

\[
\text{inl} : \text{int} + \text{int}
\]

and

\[
\text{inl} : \text{int} + \text{bool}
\]

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 `\text{inl}`, `\text{inr}` (cf OCaml).

Sums – reduction

\[
v ::= \ldots \mid \text{inl} \ v : T \mid \text{inr} \ v : T
\]

\[
\begin{align*}
\text{(inl)} & \quad \langle e, s \rangle \rightarrow \langle e', s' \rangle \quad \langle \text{inl} \ e : T, s \rangle \rightarrow \langle \text{inl} \ e' : T, s' \rangle \\
\text{(case1)} & \quad \langle \text{case } e \text{ of } \text{inl} \ (x : T_1) \Rightarrow e_1 \mid \text{inr} \ (y : T_2) \Rightarrow e_2, s \rangle \\
& \quad \rightarrow \langle \text{case } e' \text{ of } \text{inl} \ (x : T_1) \Rightarrow e_1 \mid \text{inr} \ (y : T_2) \Rightarrow e_2, s' \rangle \\
\text{(case2)} & \quad \langle \text{case } \text{inl} \ v : T \text{ of } \text{inl} \ (x : T_1) \Rightarrow e_1 \mid \text{inr} \ (y : T_2) \Rightarrow e_2, s \rangle \\
& \quad \rightarrow \langle \{v/x\} e_1, s \rangle \\
\text{(inr) and (case3) like (inl) and (case2)}
\end{align*}
\]

\[
\begin{align*}
\text{(inr)} & \quad \langle e, s \rangle \rightarrow \langle e', s' \rangle \\
& \quad \langle \text{inr} \ e : T, s \rangle \rightarrow \langle \text{inr} \ e' : T, s' \rangle \\
\text{(case3)} & \quad \langle \text{case } \text{inr} \ v : T \text{ of } \text{inl} \ (x : T_1) \Rightarrow e_1 \mid \text{inr} \ (y : T_2) \Rightarrow e_2, s \rangle \\
& \quad \rightarrow \langle \{v/y\} e_2, s \rangle
\end{align*}
\]

Constructors and Destructors

<table>
<thead>
<tr>
<th>type</th>
<th>constructors</th>
<th>destructors</th>
</tr>
</thead>
<tbody>
<tr>
<td>(T \rightarrow T)</td>
<td>(\text{fn} \ x : T \Rightarrow _)</td>
<td>(_ \ e)</td>
</tr>
<tr>
<td>(T \ast T)</td>
<td>(_ \ast _)</td>
<td>#1 _ #2 _</td>
</tr>
<tr>
<td>(T + T)</td>
<td>(\text{inl} \ (_)) (\text{inr} \ (_))</td>
<td>\text{case}</td>
</tr>
<tr>
<td>\text{bool}</td>
<td>\text{true} \text{false}</td>
<td>\text{if}</td>
</tr>
</tbody>
</table>
The typing rules for a pure language correspond to the rules for a natural deduction calculus.

### 5.2 Datatypes and Records

**ML Datatypes**

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

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

is (roughly!) like saying

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

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.
A generalization of products.

Take field labels 

\[ \text{Labels } lab \in \text{LAB } \text{for a set } \text{LAB } = \{p, q, \ldots\} \]

\[ T ::= \ldots \mid \{lab_1: T_1, \ldots, lab_k: T_k\} \]

\[ e ::= \ldots \mid \{lab_1 = e_1, \ldots, lab_k = e_k\} \mid \#\text{lab } e \]

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

Note:

- Labels are not the same syntactic class as variables, so \((\text{fn } x:T \Rightarrow \{x = 3\})\) is not an expression.
- In ML a pair \((\text{true, fn } x:\text{int} \Rightarrow x)\) is syntactic sugar for a record \(\{1 = \text{true}, 2 = \text{fn } x:\text{int} \Rightarrow x\}\).
- Note that \(#\text{lab } e\) 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.

Here the field order matters, so \((\text{fn } x:(\ell_1:\text{int}, \ell_2:\text{bool}) \Rightarrow x)(\ell_2 = \text{true}, \ell_1 = 17)\) does not typecheck.

- Here you can reuse labels, so \(\{\} \vdash \{(\ell_1 = 17), \{\ell_1 = \text{true}\} : \{\ell_1 : \text{int}\} * \{\ell_1 : \text{bool}\}\) is legal, but in some languages (e.g. OCaml) you can’t.

\[ v ::= \ldots \mid \{lab_1 = v_1, \ldots, lab_k = v_k\} \]

\[ \langle e, s \rangle \rightarrow \langle e', s' \rangle \]

\[ \langle \{lab_1 = v_1, \ldots, lab_k = v_k\}, s \rangle \rightarrow \langle \{lab_1 = v_1, \ldots, lab_k = v_k\}, s' \rangle \]

\[ \langle \#\text{lab } \{lab_1 = v_1, \ldots, lab_k = v_k\}, s \rangle \rightarrow \langle v, s \rangle \]

\[ \langle \#\text{lab } e, s \rangle \rightarrow \langle \#\text{lab } e', s' \rangle \]
5.3 Mutable Store

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

1 What we’ve got in L1 and L2:

\[
e ::= ... | \ell ::= e | !\ell | x
\]

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

\[
\text{fn } x:\text{int } \Rightarrow \ell ::= (!\ell) + x
\]

2 In C and Java,

- variables let you refer to a previously calculated value \textit{and} let you overwrite that value with another.

\[
\text{void foo}(x:\text{int}) \{
  l = l + x
  ...
\}
\]

- implicit dereferencing,

- 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 \textit{ints} – we would like to store any value
- cannot create new locations (all must exist at beginning)
- cannot write functions that abstract on locations \textit{fn } l:\text{intref } \Rightarrow !l

We are now allowing variables of \textit{T ref} type, e.g. \textit{fn } x:\text{int ref } \Rightarrow !x. Whole programs should now have no locations at the start. They should create new locations with \textit{ref}.
References – Typing

\[ \frac{\Gamma ⊢ e : T}{\Gamma ⊢ \text{ref } e : T} \]

\[ \frac{\Gamma ⊢ e_1 : \text{ref } T \quad \Gamma ⊢ e_2 : T}{\Gamma ⊢ e_1 \equiv e_2 : \text{unit}} \]

\[ \frac{\Gamma ⊢ e : T \quad \Gamma ⊢ \text{ref } e : T}{\Gamma \vdash \ell : T \quad \text{ref}} \]

Slide 180

References – Reduction

A location is a value:

\[ v ::= ... \mid \ell \]

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.

\[ (\text{ref1}) \quad \{\text{ref } v, s\} \rightarrow \{\ell, s + \{\ell \mapsto v\}\} \quad \ell \notin \text{dom}(s) \]

\[ (\text{ref2}) \quad \{e, s\} \rightarrow \{e', s'\} \quad (\text{ref } e, s) \rightarrow \{\text{ref } e', s'\} \]

\[ (\text{deref1}) \quad \{\ell, s\} \rightarrow \{v, s\} \quad \text{if } \ell \in \text{dom}(s) \text{ and } s(\ell) = v \]

\[ (\text{deref2}) \quad \{e, s\} \rightarrow \{e', s'\} \quad \{!e, s\} \rightarrow \{!e', s'\} \]

\[ (\text{assign1}) \quad \{\ell := v, s\} \rightarrow \{\text{skip}, s + \{\ell \mapsto v\}\} \quad \text{if } \ell \in \text{dom}(s) \]

\[ (\text{assign2}) \quad \{e, s\} \rightarrow \{e', s'\} \quad \{!e, s\} \rightarrow \{!e', s'\} \]

\[ (\text{assign3}) \quad \{e, s\} \rightarrow \{e', s'\} \quad \{e := e_2, s\} \rightarrow \{e' := e_2, s'\} \]

\[ \text{• A } \text{ref has to do something at runtime} - (\text{ref 0, ref 0}) \text{ should return a pair of two new locations, each containing 0, not a pair of one location repeated.} \]

\[ \text{• Note the typing and this dynamics permit locations to contain locations, e.g. ref(ref 3).} \]

\[ \text{• This semantics no longer has determinacy, for a technical reason - new locations are chosen arbitrarily. At the cost of some slight semantic complexity, we could regain determinacy by working 'up to alpha for locations'}. \]

\[ \text{• Within the language you cannot do arithmetic on locations (can in C, can't in Java) or test whether one is bigger than another. In L3 you cannot even test locations for equality (in ML you can).} \]

\[ \text{• This store just grows during computation - an implementation can garbage collect.} \]
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

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

Consider

\[
e = \text{let val } x : (\text{int} \rightarrow \text{int}) \text{ ref } = \text{ref}(\text{fn } z : \text{int} \Rightarrow z) \text{ in } \\
(x := (\text{fn } z : \text{int} \Rightarrow \text{if } z \geq 1 \text{ then } z + ((\!x) (z + -1)) \text{ else } 0)); \\
(\!x) 3)
\]

which has reductions

\[
\langle e, \{} \rangle \rightarrow^* \\
\langle e_1, \{ \ell_1 \mapsto (\text{fn } z : \text{int} \Rightarrow z) \} \rangle \rightarrow^* \\
\langle e_2, \{ \ell_1 \mapsto (\text{fn } z : \text{int} \Rightarrow \text{if } z \geq 1 \text{ then } z + ((\!\ell_1) (z + -1)) \text{ else } 0) \} \rangle \rightarrow^* (6, \ldots)
\]

For reference, $e_1$ and $e_2$ are

\[
e_1 = \ell_1 := (\text{fn } z : \text{int} \Rightarrow \text{if } z \geq 1 \text{ then } z + ((\!\ell_1) (z + -1)) \text{ else } 0); \\
(\!\ell_1) 3
\]

\[
e_2 = \text{skip}; (\!\ell_1) 3
\]

Have made a recursive function by ‘tying the knot by hand’, not using \textbf{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 22 (Well-typed store)

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

### Theorem 23 (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 $\langle e, s \rangle \rightarrow \langle e', s' \rangle$.

### Theorem 24 (Type Preservation)

If $e$ closed and $\Gamma \vdash e : T$ and $\Gamma \vdash s$ and $\langle e, s \rangle \rightarrow^* \langle e', s' \rangle$ 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 25 (Type Safety)

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

### Implementation

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

Define evaluation contexts

\[ E ::= \_ \ | \ op \ e \ | \ v \ op \ _ \ | \ if \ then \ e \ else \ e \ | \ _ \ e \ | \ _ \ v \ _ \ | \ let \ val \ x:T = \_ \ in \ e_2 \ end \ | \ \langle \_, \_ \rangle \ | \ \langle \v, \_ \rangle \ | \ \langle \#1, \_ \rangle \ | \ \langle \#2, \_ \rangle \ | \ \langle \inl, \_ \rangle \ | \ \langle \inr, \_ \rangle \ | \ \langle \case \ of \ \inl \ (x: T) \Rightarrow \_ \ | \ \inr \ (x: T) \Rightarrow \_ \ | \ \{ \text{lab}_1 = v_1, \ldots, \text{lab}_k = v_k \} | \ # \text{lab}_i \ | \ _ ::= v \ | \ v ::= \_ \ | \ ref \_ \]

and have the single context rule

\[ (\text{eval}) \quad \langle e, s \rangle \rightarrow \langle e', s' \rangle \]

replacing the rules (all those with \( \geq 1 \) premise) \((\text{op1}), (\text{op2}), (\text{seq2}), (\text{if3}), (\text{app1}), (\text{app2}), (\text{let1}), (\text{pair1}), (\text{pair2}), (\text{proj3}), (\text{proj4}), (\text{nil}), (\text{inr}), (\text{case1}), (\text{record1}), (\text{record3}), (\text{ref2}), (\text{deref2}), (\text{assign2}), (\text{assign3}). \]

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

\textbf{Theorem 26} The two definitions of \( \rightarrow \) define the same relation.

\[\begin{array}{c}
\text{Formal logic} & 1880- \\
\text{Untyped lambda calculus} & 1930s \\
\text{Simply-typed lambda calculus} & 1940s \\
\text{Fortran} & 1950s \\
\text{Curry-Howard, Algol 60, Algol 68, SECD machine (64)} & 1960s \\
\text{Pascal, Polymorphism, ML, PL/1, PL/2} & 1970s \\
\text{Structured Operational Semantics} & 1981- \\
\text{Standard ML definition} & 1985 \\
\text{Haskell} & 1990- \\
\text{Subtyping} & 1980s \\
\text{Module systems} & 1980- \\
\text{Object calculus} & 1990- \\
\text{Typed assembly and intermediate languages} & 1990- \\
\end{array}\]

And now? module systems, distribution, mobility, reasoning about objects, security, typed compilation......
5.5 L3: Collected definition

L3 syntax

Booleans \( b \in \mathcal{B} = \{ \text{true}, \text{false} \} \)
Integers \( n \in \mathcal{Z} = \{-\ldots,-1,0,1,\ldots\} \)
Locations \( \ell \in \mathcal{L} = \{ l_0, l_1, l_2, \ldots \} \)
Variables \( x \in \mathcal{X} \) for a set \( \mathcal{X} = \{ x, y, z, \ldots \} \)
Labels \( \text{lab} \in \text{LAB} \) for a set \( \text{LAB} = \{ p, q, \ldots \} \)

Operations \( \text{op} ::= + \geq \)

Types:

\[
T ::= \text{int} | \text{bool} | \text{unit} | T_1 \rightarrow T_2 | T_1 \times T_2 | T_1 + T_2 | \{ \text{lab}_1:T_1, \ldots, \text{lab}_k:T_k \} \]

Expressions

\[
e ::= n | b | e_1 \text{ op } e_2 | \text{if } e_1 \text{ then } e_2 \text{ else } e_3 |
\]

\[
\begin{align*}
\text{skip} & : e_1; e_2 |
\text{while } e_1 \text{ do } e_2 | \\
\text{let val } x:T & = e_1 \text{ in } e_2 \text{ end} | \\
\text{let val rec } x:T_1 & \rightarrow T_2 = (\text{fn } y:T_1 \Rightarrow e_1) \text{ in } e_2 \text{ end} | \\
\text{inl } e:T & \text{ inr } e:T | \\
\text{case } e \text{ of } \text{inl } (x_1:T_1) & \Rightarrow e_1 | \text{inr } (x_2:T_2) \Rightarrow e_2 | \\
\{ \text{lab}_1 = e_1, \ldots, \text{lab}_k = e_k \} & | \# \text{lab } e
\end{align*}
\]

(where in each record (type or expression) no \( \text{lab} \) occurs more than once)

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

L3 semantics

Stores \( s \) are finite partial maps from \( \mathcal{L} \) to the set of all values.

Values \( v ::= b | n | \text{skip} | \text{fn } x:T \Rightarrow e | (n_1, v_2) | \text{inl } v:T | \text{inr } v:T | \{ \text{lab}_1 = v_1, \ldots, \text{lab}_k = v_k \} | \ell \)

\[
\begin{align*}
(\text{op } +) & \quad \langle n_1 + n_2, s \rangle \rightarrow \langle n, s \rangle \quad \text{if } n = n_1 + n_2 \\
(\text{op } \geq) & \quad \langle n_1 \geq n_2, s \rangle \rightarrow \langle b, s \rangle \quad \text{if } b = (n_1 \geq n_2) \\
(\text{op}_1) & \quad \langle e_1, s \rangle \rightarrow \langle e_1', s' \rangle \\
(\text{op}_2) & \quad \langle e_2, s \rangle \rightarrow \langle e_2', s' \rangle \\
(\text{seq}_1) & \quad \langle \text{skip} ; e_2, s \rangle \rightarrow \langle e_2, s \rangle \\
(\text{seq}_2) & \quad \langle e_1, e_2, s \rangle \rightarrow \langle e_1', e_2', s' \rangle
\end{align*}
\]

81
(if1) \( \langle \text{if true then } e_2 \text{ else } e_3, s \rangle \rightarrow \langle e_2, s \rangle \)

(If2) \( \langle \text{if false then } e_2 \text{ else } e_3, s \rangle \rightarrow \langle e_3, s \rangle \)

(If3) \( \langle e_1, s \rangle \rightarrow \langle e'_1, s' \rangle \)

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

(app1) \( \langle e_1, s \rangle \rightarrow \langle e'_1, s' \rangle \) \( \langle e_1, e_2, s \rangle \rightarrow \langle e'_1, e'_2, s' \rangle \)

(app2) \( \langle e_2, s \rangle \rightarrow \langle e'_2, s' \rangle \) \( \langle v, e_2, s \rangle \rightarrow \langle v, e'_2, s' \rangle \)

(fn) \( \langle \text{fn } x:T \Rightarrow e \text{ in } s \rangle \rightarrow \langle \text{\{v/x\} } e, s \rangle \)

(let1) \( \langle e_1, s \rangle \rightarrow \langle e'_1, s' \rangle \)

\( \langle \text{let val } x:T = e_1 \text{ in } e_2, s \rangle \rightarrow \langle \text{let val } x:T = e'_1 \text{ in } e_2, s' \rangle \)

(let2) \( \langle \text{let val } x:T = v \text{ in } e_2, s \rangle \rightarrow \langle \{v/x\} e_2, s \rangle \)

(letrecfn) \( \langle \text{let val rec } x:T_1 \rightarrow T_2 = (\text{fn } y:T_1 \Rightarrow e_1) \text{ in } e_2, s \rangle \rightarrow \langle \{\text{fn } y:T_1 \Rightarrow \text{let val rec } x:T_1 \rightarrow T_2 = (\text{fn } y:T_1 \Rightarrow e_1) \text{ in } e_1, s \}/x \} e_2, s \rangle \)

(pair1) \( \langle e_1, e_2, s \rangle \rightarrow \langle \{e'_1, e'_2, s' \} \)

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

(proj1) \( \langle \#(v_1, e_2), s \rangle \rightarrow \langle v_1, s \rangle \) \( \langle \#(v_2, e_2), s \rangle \rightarrow \langle e_2, s \rangle \)

(proj2) \( \langle \#(v_1, e_2), s \rangle \rightarrow \langle \{v_1, e'_2, s' \} \)

(proj3) \( \langle \#(e, s) \rightarrow \langle \#(e', s') \)

(proj4) \( \langle \#(e, s) \rightarrow \langle \#(e', s') \)

(inl) \( \langle \text{inl } e:T, s \rangle \rightarrow \langle \text{inl } e:T, s' \rangle \)

(case1) \( \langle \text{case } e \text{ of } \text{inl } (x:T_1) \Rightarrow e_1 \text{ } \text{inr } (y:T_2) \Rightarrow e_2, s \rangle \rightarrow \langle \text{case } e' \text{ of } \text{inl } (x:T_1) \Rightarrow e_1 \text{ } \text{inr } (y:T_2) \Rightarrow e_2, s' \rangle \)

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

(inr) and (case3) like (inl) and (case2)
We use an abbreviated notation to access and manipulate such pairs, writing
\[ \text{(inr)} \quad \langle e, s \rangle \rightarrow \langle e', s' \rangle \]
and
\[ \text{(inr)} \quad e : T, s \rightarrow \langle \text{inr} \quad e' : T, s' \rangle \]

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

\[ \text{(record1)} \quad \langle \{ \text{lab}_1 = v_1, \ldots, \text{lab}_i = e_i, \ldots, \text{lab}_k = e_k \}, s \rangle \rightarrow \langle \{ \text{lab}_1 = v_1, \ldots, \text{lab}_i = e'_i, \ldots, \text{lab}_k = e_k \}, s' \rangle \]

\[ \text{(record2)} \quad \langle \# \text{lab}_i \{ \text{lab}_1 = v_1, \ldots, \text{lab}_k = e_k \}, s \rangle \rightarrow \langle u, s \rangle \]

\[ \text{(record3)} \quad \langle \# \text{lab}_i \ e, s \rangle \rightarrow \langle \# \text{lab}_i \ e', s' \rangle \]

\[ \text{(ref1)} \quad \langle \text{ref} \quad v, s \rangle \rightarrow \langle \ell, s + \{ \ell \rightarrow v \} \rangle \quad \ell \notin \text{dom}(s) \]

\[ \text{(ref2)} \quad \langle \text{ref} \quad e, s \rangle \rightarrow \langle \text{ref} \quad e', s' \rangle \]

\[ \text{(deref1)} \quad \langle \ell, s \rangle \rightarrow \langle v, s \rangle \quad \text{if} \ \ell \in \text{dom}(s) \ \text{and} \ s(\ell) = v \]

\[ \text{(deref2)} \quad \langle e, s \rangle \rightarrow \langle e', s' \rangle \]

\[ \langle e, s \rangle \rightarrow \langle e' \rangle \]

\[ \langle e := v, s \rangle \rightarrow \langle \text{skip}, s + \{ \ell \rightarrow v \} \rangle \quad \text{if} \ \ell \in \text{dom}(s) \]

\[ \langle e, s \rangle \rightarrow \langle e', s' \rangle \]

\[ \langle e := e, s \rangle \rightarrow \langle e := e', s' \rangle \]

\[ \langle e, s \rangle \rightarrow \langle e', s' \rangle \]

\[ \langle e := e_2, s \rangle \rightarrow \langle e' := e_2, s' \rangle \]

---

**L3 Typing**

Type environments, \( \Gamma \in \text{TypeEnv2} \), are pairs of a finite partial function \( \Gamma_{\text{loc}} \) from \( L \) to \( T_{\text{loc}} \) and a finite partial function \( \Gamma_{\text{var}} \) from \( X \) to \( T \).

We use an abbreviated notation to access and manipulate such pairs, writing \( \Gamma(\ell) \) for \( \Gamma_{\text{loc}}(\ell) \) and \( \Gamma(x) \) for \( \Gamma_{\text{var}}(x) \), and similarly writing just \( \Gamma, x : T \) for the pair of \( \Gamma_{\text{loc}} \) and \( \Gamma_{\text{var}}, x : T \).

\[
\begin{align*}
\text{(int)} \quad & \Gamma \vdash \text{n : int} \quad \text{for} \ n \in \mathbb{Z} \\
\text{(bool)} \quad & \Gamma \vdash \text{b : bool} \quad \text{for} \ b \in \{ \text{true, false} \} \\
\text{(op +)} \quad & \Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \quad \Gamma \vdash e_1 + e_2 : \text{int} \\
\text{(op ≥)} \quad & \Gamma \vdash e_1 + e_2 : \text{int} \quad \Gamma \vdash e_1 \geq e_2 : \text{bool} \\
\text{(if)} \quad & \Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : T \quad \Gamma \vdash e_3 : T \\
\Gamma \vdash \text{if} \ e_1 \ \text{then} \ e_2 \ \text{else} \ e_3 : T
\end{align*}
\]
(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) \( \frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{unit}}{\Gamma \vdash \text{while } e_1 \text{ do } e_2 : \text{unit}} \)

(var) \( \frac{\Gamma \vdash x : T \quad \text{if } \Gamma(x) = T}{\Gamma \vdash e_1 ; e_2 : T} \)

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

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

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

(let rec fn) \( \frac{\Gamma, x : T_1 \rightarrow T_2, y : T_1 \vdash e_1 : T_2 \quad \Gamma, x : T_1 \rightarrow T_2 \vdash e_2 : T \quad \Gamma \vdash \text{let val rec } x_1 : T_1 \rightarrow T_2 = (\text{fn } y : T_1 \Rightarrow e_1) \text{ in } e_2 \text{ end} : T}{\Gamma \vdash e_1 : T'} \)

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

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

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

(inl) \( \frac{\Gamma \vdash e : T_1}{\Gamma \vdash \text{inl } e : T_1 \ast T_2 ; T_1 + T_2} \)

(inr) \( \frac{\Gamma \vdash e : T_2}{\Gamma \vdash \text{inr } e : T_1 \ast T_2 ; T_1 + T_2} \)

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

(record) \( \frac{\Gamma \vdash e_1 : T_1 \quad \ldots \quad \Gamma \vdash e_k : T_k}{\Gamma \vdash \{ \text{lab}_1 = e_1, \ldots, \text{lab}_k = e_k \} : \{ \text{lab}_1 : T_1, \ldots, \text{lab}_k : T_k \}} \)

(recordproj) \( \frac{\Gamma \vdash e : \{ \text{lab}_1 : T_1, \ldots, \text{lab}_k : T_k \}}{\Gamma \vdash \#\text{lab}_1 e : T_1} \)
\[ \frac{\Gamma \vdash e : T}{\Gamma \vdash \text{ref } e : T \text{ ref}} \]

\[ \frac{\Gamma \vdash e_1 : T \text{ ref} \quad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1 := e_2 : \text{unit}} \]

\[ \frac{\Gamma \vdash e : T \text{ ref}}{\Gamma \vdash !e : T} \]

\[ \frac{\Gamma(\ell) = T \text{ ref}}{\Gamma \vdash \ell : T \text{ ref}} \]

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

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.

### 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, ...

### Subtyping – Motivation

Recall

\[
\Gamma \vdash e_1 : T \rightarrow T'
\]

\[
\begin{array}{c}
\Gamma \vdash e_2 : T' \\
\Gamma \vdash e_1, e_2 : T''
\end{array}
\]

so can’t type

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

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

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

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

Will define it on the next slides, but it will include
\[ \{ p:int, q:int \} <: \{ p:int \} <: \{ \} \]

Introduce a subsumption rule
\[ \frac{e:T \Rightarrow T'}{\Gamma \vdash e:T'} \]
allowing subtyping to be used, capturing (*).

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

---

Example

\[ x: \{ p:int \} \vdash x: \{ p:int \} \]
\[ \text{(var)} \]

\[ x: \{ p:int \} \vdash \#p x: \{ p:int \} \]
\[ \text{(record-proj)} \]

\[ \{\} \vdash (\text{fn} x: \{ p:int \} \Rightarrow \#p x): \{ p:int \} \rightarrow \text{int} \]
\[ \text{(fn)} \]

\[ \{\} \vdash 3: \text{int} \]
\[ \text{(var)} \]

\[ \{\} \vdash 4: \text{int} \]
\[ \text{(var)} \]

\[ \{\} \vdash \{ p = 3, q = 4 \}: \{ p:int, q:int \} \]
\[ \text{(record)} \]

\[ \{\} \vdash \{ p = 3, q = 4 \}: \{ p:int \} \]
\[ \text{(sub)} \]

\[ \{\} \vdash (\text{fn} x: \{ p:int \} \Rightarrow \#p x) \{ p = 3, q = 4 \}: \text{int} \]
\[ \text{(app)} \]

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

Now, we define the subtype relation.

---

The Subtype Relation
\( T <: T' \)

(s-refl) \( T <: T \)

(s-trans) \( T <: T' \Rightarrow T' <: T'' \Rightarrow T <: T'' \)

---

Subtyping – Records

Forgetting fields on the right:
\[ \{ lab_1:T_{11}, \ldots, lab_k:T_{k1}, lab_{k+1}:T_{k+11}, \ldots, lab_{k+k}:T_{k+k1} \} \]
\( <: \)
\( \{ lab_1:T_{11}, \ldots, lab_k:T_{k1} \} \)
\( \text{(s-record-width)} \)

Allowing subtyping within fields:
\( T_k <: T'_k \Rightarrow T_k <: T'_k \)
\( \{ lab_1:T_{11}, \ldots, lab_k:T_{k1} \} <: \{ lab_1:T'_{11}, \ldots, lab_k:T'_{k1} \} \)
\( \text{(s-record-depth)} \)

Combining these:
\( \{ p:int, q:int \} <: \{ p:int \} \)
\( \{ x: \{ p:int \}, y: \{ r:int \} \} <: \{ x: \{ p:int \}, y: \{ \} \} \)
\( \text{(s-record-width)} \)
\( \text{(s-record-width)} \)
\( \text{(s-record-depth)} \)
Another example:

\[
\{x:\{p:\text{int}, q:\text{int}\}\}, y:\{x:\{p:\text{int}, q:\text{int}\}\}\} \lll \{x:\{p:\text{int}, q:\text{int}\}\} \lll \{x:\{p:\text{int}\}\}
\]

Allowing reordering of fields:

\[
\pi \text{ a permutation of } 1, \ldots, k
\]

\[
\{\text{lab}_1:T_1, \ldots, \text{lab}_k:T_k\}\lll \{\text{lab}_{\pi(1)}:T_{\pi(1)}, \ldots, \text{lab}_{\pi(k)}:T_{\pi(k)}\}
\]

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

Subtyping – Functions

\[
\begin{align*}
T'_1 & \lll T_1 & \quad & T'_2 & \lll T_2 \\
T_1 & \rightarrow T_2 & \lll & T'_1 & \rightarrow T'_2
\end{align*}
\]

contravariant on the left of \(\rightarrow\)

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

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

\[
f = \text{fn } x:\{p:\text{int}\} \Rightarrow \{p = \#p x, q = 28\}
\]

we have

\[
\begin{align*}
\{} & \vdash f:\{p:\text{int}\} \rightarrow \{\text{int}, q:\text{int}\} \\
\{} & \vdash f:\{\text{int}\} \rightarrow \{\text{int}\} \\
\{} & \vdash f:\{\text{int}, q:\text{int}\} \rightarrow \{\text{int}, q:\text{int}\} \\
\{} & \vdash f:\{\text{int}, q:\text{int}\} \rightarrow \{\text{int}\}
\end{align*}
\]

as

\[
\{\text{int}, q:\text{int}\} \lll \{\text{int}\}
\]

On the other hand, for

\[
\text{fn } x:\{\text{int}, q:\text{int}\} \Rightarrow \{p = (\#p x) + (\#q x)\}
\]

we have

\[
\begin{align*}
\{} & \vdash f:\{\text{int}, q:\text{int}\} \rightarrow \{\text{int}\} \\
\{} & \not\vdash f:\{\text{int}\} \rightarrow T \quad \text{for any } T \\
\{} & \not\vdash f:T \rightarrow \{\text{int}, q:\text{int}\} \quad \text{for any } T
\end{align*}
\]
Subtyping – Products

Just like (s-record-depth)

\[
\begin{align*}
T_1 {:} T'_1 & \quad T_2 {:} T'_2 \\
T_1 \ast T_2 & \quad T'_1 \ast T'_2
\end{align*}
\]

Subtyping – Sums

Exercise.

Subtyping – References

Are either of these any good?

\[
\begin{align*}
T & {:} T' \\
T \text{ ref} & {:} T' \text{ ref} \\
T' & {:} T
\end{align*}
\]

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

\[
e ::= \ldots \mid (T)e
\]

with typing rule

\[
\Gamma \vdash e : T' \\
\Gamma \vdash (T)e : T
\]

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).
(Very Simple) Objects

```ocaml
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}.
```

Using Subtyping

```ocaml
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}.
```

Object Generators

```ocaml
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...
Recall \( \text{Counter} = \{ \text{get:} \text{unit} \rightarrow \text{int}, \text{inc:} \text{unit} \rightarrow \text{unit} \} \).

First, make the internal state into a record.
\( \text{CounterRep} = \{ \text{p: int ref} \} \).

\[
\begin{align*}
\text{let val } & \quad \text{counterClass:} \text{CounterRep} \rightarrow \text{Counter} = \\
& \quad \text{fn } x: \text{CounterRep} \Rightarrow \\
& \quad \{ \text{get = fn } y: \text{unit} \Rightarrow !(\#p x), \\
& \quad \text{inc = fn } y: \text{unit} \Rightarrow (\#p x) := 1+!(\#p x) \}\n\end{align*}
\]

\[
\begin{align*}
\text{let val } & \quad \text{newCounter:} \text{unit} \rightarrow \text{Counter} = \\
& \quad \text{fn } y: \text{unit} \Rightarrow \\
& \quad \text{let val } x: \text{CounterRep} = \{ \text{p = ref 0} \} \text{ in } \\
& \quad \text{counterClass } x
\end{align*}
\]

\[
\begin{align*}
\text{let val } & \quad \text{resetCounterClass:} \text{CounterRep} \rightarrow \text{ResetCounter} = \\
& \quad \text{fn } x: \text{CounterRep} \Rightarrow \\
& \quad \text{let val } \quad \text{super = counterClass } x \text{ in } \\
& \quad \{ \text{get = #get super,} \\
& \quad \text{inc = #inc super,} \\
& \quad \text{reset = fn } y: \text{unit} \Rightarrow (\#p x) := 0 \}\n\end{align*}
\]

\( \text{CounterRep} = \{ \text{p: int ref} \} \).
\( \text{Counter} = \{ \text{get:} \text{unit} \rightarrow \text{int}, \text{inc:} \text{unit} \rightarrow \text{unit} \} \).
\( \text{ResetCounter} = \{ \text{get:} \text{unit} \rightarrow \text{int}, \text{inc:} \text{unit} \rightarrow \text{unit}, \text{reset:} \text{unit} \rightarrow \text{unit} \} \).

\[
\begin{align*}
\text{class Counter} & \quad \{ \text{protected int } \text{p}; \\
& \quad \text{Counter}() \{ \text{this.p=0; } \} \\
& \quad \text{int get }() \{ \text{return this.p; } \} \\
& \quad \text{void inc }() \{ \text{this.p++; } \} \\
& \quad \}\n\end{align*}
\]

\[
\begin{align*}
\text{class ResetCounter} & \quad \{ \text{extends Counter} \\
& \quad \text{extends Counter} \\
& \quad \{ \text{void reset }() \{ \text{this.p=0; } \} \\
& \quad \}\n\end{align*}
\]
6.1 Exercises

Exercise 31 ★For each of the following, either give a type derivation or explain why it is untypable.

1. \(\{\}\ \vdash \{p = \{p = \{p = 3\}\}\}:\{p:\}\}\)
2. \(\{\}\ \vdash \text{fn } x:\{p:\text{bool}, q:\{p:\text{int}, q:\text{bool}\}\} \Rightarrow \#q \#p x : ?\)
3. \(\{\}\ \vdash \text{fn } f:\{p:\text{int}\} \rightarrow \text{int} \Rightarrow (f \{q = 3\}) + (f \{p = 4\}) : ?\)
4. \(\{\}\ \vdash \text{fn } f:\{p:\text{int}\} \rightarrow \text{int} \Rightarrow (f \{q = 3, p = 2\}) + (f \{p = 4\}) : ?\)

Exercise 32 ★For each of the two bogus \(T\:\text{ref}\) subtype rules on Slide 202, 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 \texttt{let} and \texttt{let rec} ?

Exercise 35 ★★Prove a Progress Theorem for L3 with subtyping.
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

- 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.

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

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

Going to define the simplest possible concurrent language, call it L1, and explore a few issues. You’ve seen most of them informally in C&DS.

Booleans \( b \in B = \{ \text{true, false} \} \)
Integers \( n \in Z = \{ ..., -1, 0, 1, ... \} \)
Locations \( \ell \in L = \{ l_0, l_1, l_2, ... \} \)
Operations \( op ::= + | \geq \)

Expressions
\[
\begin{align*}
e & ::= n | b | e_1 \ op e_2 | \text{if } e_1 \ \text{then } e_2 \ \text{else } e_3 \\
\ell & ::= e | \ell \\
\text{skip} & ::= e_1; e_2 \\
\text{while } e_1 \ \text{do } e_2 & \\
\end{align*}
\]

Types \( T ::= \text{int} | \text{bool} | \text{unit} | \text{proc} \)
Locations \( T_{loc} ::= \text{intref} \)

Parallel Composition: Typing and Reduction

\[
\begin{align*}
\text{(thread)} & \quad \Gamma \vdash e : \text{unit} \quad \Gamma \vdash e : \text{proc} \\
\text{(parallel)} & \quad \Gamma \vdash e_1 : \text{proc} \quad \Gamma \vdash e_2 : \text{proc} \\
& \quad \Gamma \vdash e_1 e_2 : \text{proc} \\
\text{(parallel1)} & \quad \langle e_1, s \rangle \longrightarrow \langle e'_1, s' \rangle \\
& \quad \langle e_1 e_2, s \rangle \longrightarrow \langle e'_1 e_2, s' \rangle \\
\text{(parallel2)} & \quad \langle e_2, s \rangle \longrightarrow \langle e'_2, s' \rangle \\
& \quad \langle e_1 e_2, s \rangle \longrightarrow \langle e_1 e'_2, s' \rangle \\
\end{align*}
\]

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.

\[
\begin{align*}
\langle i \rangle l := 2, \{ l \rightarrow 1 \} & \longrightarrow \langle i \rangle \langle \rangle, \{ l \rightarrow 2 \} \\
\{ l := 1 \} l := 2, \{ l \rightarrow 0 \} & \quad \text{(initial state)} \\
\{ l := 1 \} \langle \rangle, \{ l \rightarrow 2 \} & \longrightarrow \langle i \rangle \langle \rangle, \{ l \rightarrow 1 \}
\end{align*}
\]

NB from here on, we are using () instead of skip — that’s the ML syntax.

But, assignments and dereferencing are atomic. For example,

\[
\langle l := 3498734590879238429384 \mid l := 7, \{ l \rightarrow 0 \} \rangle
\]

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 \( \langle l := e \rangle \mid e' \), the steps of evaluating \( e \) and \( e' \) can be interleaved.

Think of \( \langle l := 1+!l \rangle \mid (l := 7+!l) \) – there are races....
The behaviour of

\[
(l := 1 + !l) (l := 7 + !l)
\]

for the initial store \( \{ l \rightarrow 0 \} \):

```
\[
(0 \leftarrow t) (0 \leftarrow t) (0 \leftarrow t)
\]
```

The behaviour of the initial store for \( (i + 2 = i) \) and \( (i + 1 = i) \):
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...

<table>
<thead>
<tr>
<th>Slide 222</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Morals</strong></td>
</tr>
<tr>
<td>• There is a combinatorial explosion.</td>
</tr>
<tr>
<td>• Drawing state-space diagrams only works for really tiny examples – we need better techniques for analysis.</td>
</tr>
<tr>
<td>• Almost certainly you (as the programmer) didn’t want all those 3 outcomes to be possible – need better idioms or constructs for programming.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Slide 223</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>So, how do we get anything coherent done?</strong></td>
</tr>
<tr>
<td>Need some way(s) to synchronize between threads, so can enforce mutual exclusion for shared data.</td>
</tr>
<tr>
<td>cf. Lamport’s “Bakery” algorithm from Concurrent and Distributed Systems. Can you code that in L1? If not, what’s the smallest extension required?</td>
</tr>
<tr>
<td>Usually, though, you can depend on built-in support from the scheduler, e.g. for mutexes and condition variables (or, at a lower level, <code>t a s</code> or <code>c a s</code>).</td>
</tr>
</tbody>
</table>


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).

<table>
<thead>
<tr>
<th>Slide 224</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Adding Primitive Mutexes</strong></td>
</tr>
<tr>
<td>Mutex names $m \in M = {m, m_1, \ldots}$</td>
</tr>
<tr>
<td>Configurations $\langle e, s, M \rangle$ where $M: \mathbb{M} \rightarrow \mathbb{B}$ is the mutex state</td>
</tr>
<tr>
<td>Expressions $e ::= \ldots \mid \text{lock} \ m \mid \text{unlock} \ m$</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{(lock)} & \quad \frac{\Gamma \vdash \text{lock} \ m : \text{unit}}{\Gamma \vdash \text{lock} \ m : \text{unit}} \\
\text{(unlock)} & \quad \frac{\Gamma \vdash \text{unlock} \ m : \text{unit}}{\Gamma \vdash \text{unlock} \ m : \text{unit}} \\
\text{(lock)} & \quad \frac{\langle \text{lock} \ m, s, M \rangle \rightarrow \langle(), s, M + \{m \mapsto \text{true}\}\rangle \text{ if } \neg M(m)}{\Gamma \vdash \text{lock} \ m, s, M : \text{unit}} \\
\text{(unlock)} & \quad \frac{\langle \text{unlock} \ m, s, M \rangle \rightarrow \langle(), s, M + \{m \mapsto \text{false}\}\rangle}{\Gamma \vdash \text{unlock} \ m, s, M : \text{unit}}
\end{align*}
\]

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

\[
(e_2, s) \rightarrow (e'_2, s')
\]

by

\[
(e_2, s, M) \rightarrow (e'_2, s', M')
\]

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 \((e, s, M)\), with the initial store \( s = \{l \mapsto 0\} \) and initial mutex state \( M_0 = \lambda m \in M. \text{false} \), 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.

Using Several Mutexes

\[
\text{lock } m \text{ can block (that’s the point). Hence, you can deadlock.}
\]

Locking Disciplines

So, suppose we have several programs \( e_1, \ldots, e_k \), 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.
An Ordered 2PL Discipline, Informally

Fix an association between locations and mutexes. For simplicity, make it 1:1 – associate \( l_i \) with \( m_i \), \( l_1 \) with \( m_1 \), etc.

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

Require that each \( e_i \)

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

Then, informally, \((e_1 \ldots e_k)\) should (a) never deadlock, and (b) be \textbf{serializable} – any execution of it should be ‘equivalent’ to an execution of \( e_{\pi(1)} \ldots e_{\pi(k)} \) for some permutation \( \pi \).

These are \textbf{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.

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 \( \langle e_1 \ldots e_i \ldots e_k, s, M \rangle \), \( e_i \) acquires the lock \( m_j \) for each location \( l_j \) it uses, before it uses it

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

Solution: Thread local semantics

Instead of only defining the global \( \langle e, s, M \rangle \rightarrow \langle e', s', M' \rangle \), with rules

\[
\text{assign1}: \quad \ell := n \rightarrow \langle \text{skip}, s + \{\ell \rightarrow n\}, M \rangle \quad \text{if } \ell \in \text{dom}(s)
\]

\[
\text{parallel1}: \quad \langle e_1, s, M \rangle \rightarrow \langle e'_1, s', M' \rangle \quad \langle e_2, s, M \rangle \rightarrow \langle e'_2, s', M' \rangle \rightarrow \langle e'_1, e'_2, s', M' \rangle
\]

define a per-thread \( e \xrightarrow{n} e' \) and use that to define

\( \langle e, s, M \rangle \rightarrow \langle e', s', M' \rangle \), with rules like

\[
\text{t-assn1}: \quad \ell := n \xrightarrow{c} \text{skip}
\]

\[
\text{t-parallel1}: \quad \frac{e_1 \xrightarrow{c_1} e'_1}{e_1[e_2 \rightarrow e'_2]} \quad \frac{e_2 \xrightarrow{c_2} e'_2}{e_1[e_2 \rightarrow e'_2]}
\]

\[
\text{c-assn}: \quad e \xrightarrow{\ell = n} e' \quad \ell \in \text{dom}(s)
\]

\[
\langle e, s, M \rangle \rightarrow \langle e', s + \{\ell \rightarrow n\}, M \rangle
\]
Note the per-thread rules don’t mention $s$ or $M$. Instead, we record in the label $\ell$ what interactions with the store or mutexes it has.

$$a ::= \tau \mid \ell := n \mid \ell = n \mid \text{lock } m \mid \text{unlock } m$$

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 27 (Coincidence of global and thread-local semantics)** The two definitions of $\xrightarrow{}$ 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 $e = (\text{lock } m; (l := 1+!l; \text{unlock } m))$ we have

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

Hence, using (t-parallel) and the (c-*) rules, for $s' = s + \{l \mapsto 1 + s(l)\}$, $\langle e|e', s, M_0 \rangle \xrightarrow{} \text{skip}|e', s', M_0 \rangle$

(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

\[
\text{Global Semantics}
\]

\[
\begin{align*}
\text{(op +)} & \quad \langle n_1 + n_2, s, M \rangle \rightarrow \langle n, s, M \rangle & \text{if } n = n_1 + n_2 \\
\text{(op ≥)} & \quad \langle n_1 \geq n_2, s, M \rangle \rightarrow \langle b, s, M \rangle & \text{if } b = (n_1 \geq n_2) \\
\text{(op1)} & \quad \langle e_1, s, M \rangle \rightarrow \langle e'_1, s', M' \rangle & \quad \langle e_1 \text{ op } e_2, s, M \rangle \rightarrow \langle e'_1 \text{ op } e_2, s', M' \rangle \\
\text{(op2)} & \quad \langle e_2, s, M \rangle \rightarrow \langle e'_2, s', M' \rangle & \quad \langle v \text{ op } e_2, s, M \rangle \rightarrow \langle v \text{ op } e_2, s', M' \rangle \\
\text{(deref)} & \quad \langle M, s, M \rangle \rightarrow \langle n, s, M \rangle & \text{if } \ell \in \text{dom}(s) \text{ and } s(\ell) = n \\
\text{(assign1)} & \quad \langle \ell := n, s, M \rangle \rightarrow \langle \text{skip}, s + \{\ell \mapsto n\}, M \rangle & \text{if } \ell \in \text{dom}(s) \\
\text{(assign2)} & \quad \langle e, s, M \rangle \rightarrow \langle e', s', M' \rangle & \quad \langle \ell := e, s, M \rangle \rightarrow \langle \ell := e', s', M' \rangle \\
\text{(seq1)} & \quad \langle \text{skip}, e_2, s, M \rangle \rightarrow \langle e_2, s, M \rangle \\
\text{(seq2)} & \quad \langle e_1, s, M \rangle \rightarrow \langle e'_1, s', M' \rangle & \quad \langle e_1; e_2, s, M \rangle \rightarrow \langle e'_1; e_2, s', M' \rangle \\
\text{(if1)} & \quad \langle \text{if } \text{true } \text{then } e_2 \text{ else } e_3, s, M \rangle \rightarrow \langle e_2, s, M \rangle \\
\text{(if2)} & \quad \langle \text{if } \text{false } \text{then } e_2 \text{ else } e_3, s, M \rangle \rightarrow \langle e_3, s, M \rangle \\
\text{(if3)} & \quad \langle e_1, s, M \rangle \rightarrow \langle e'_1, s', M' \rangle & \quad \langle \text{if } e_1 \text{ then } e_2 \text{ else } e_3, s, M \rangle \rightarrow \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 \rightarrow \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 \langle e_1; e_2, s, M \rangle \rightarrow \langle e'_1; e_2, s', M' \rangle & \quad \langle e_1; e_2, s, M \rangle \rightarrow \langle e'_1; e_2, s', M' \rangle \\
\text{(parallel2)} & \quad \langle e_2, s, M \rangle \rightarrow \langle e'_2, s', M' \rangle & \quad \langle e_1; e_2, s, M \rangle \rightarrow \langle e'_1; e_2, s', M' \rangle \\
\text{(lock)} & \quad \langle \text{lock } m, s, M \rangle \rightarrow \langle (\ell), s, M + \{m \mapsto \text{true}\} \rangle & \text{if } \neg M(m) \\
\text{(unlock)} & \quad \langle \text{unlock } m, s, M \rangle \rightarrow \langle (\ell), s, M + \{m \mapsto \text{false}\} \rangle \\
\text{(c-tau)} & \quad \langle e, s, M \rangle \rightarrow \langle e', s, M \rangle \\
\text{(c-assign)} & \quad \langle e, s, M \rangle \rightarrow \langle e', s + \{\ell \mapsto n\}, M \rangle \\
\text{(c-deref)} & \quad \langle e, s, M \rangle \rightarrow \langle e', s, M \rangle & \quad \langle e, s, M \rangle \rightarrow \langle e', s, M \rangle \\
\text{(c-lock)} & \quad \langle e, s, M \rangle \rightarrow \langle e', s, M + \{m \mapsto \text{true}\} \rangle \\
\text{(c-unlock)} & \quad \langle e, s, M \rangle \rightarrow \langle e', s, M + \{m \mapsto \text{false}\} \rangle
\end{align*}
\]

Thread-Local Semantics

\[
\text{Thread-Local Semantics}
\]

\[
\begin{align*}
\text{(t-op +)} & \quad n_1 + n_2 \rightarrow n & \text{if } n = n_1 + n_2 \\
\text{(t-op ≥)} & \quad n_1 \geq n_2 \rightarrow b & \text{if } b = (n_1 \geq n_2) \\
\text{(t-op1)} & \quad e_1 \text{ op } e_2 \rightarrow e'_1 \text{ op } e_2 \\
\text{(t-op2)} & \quad e_2 \text{ op } e_2 \rightarrow e'_2 \text{ op } e_2 \\
\text{(t-deref)} & \quad \ell \text{ = } n \rightarrow n \\
\text{(t-assign1)} & \quad \ell := n \rightarrow \text{skip} \\
\text{(t-assign2)} & \quad e \rightarrow e' \\
\text{(t-seq1)} & \quad \text{skip}, e_2 \rightarrow e_2 \\
\text{(t-seq2)} & \quad e_1 \rightarrow e'_1; e_2 \\
\text{(t-if1)} & \quad \text{if } \text{true } \text{then } e_2 \text{ else } e_3 \rightarrow e_2 \\
\text{(t-if2)} & \quad \text{if } \text{false } \text{then } e_2 \text{ else } e_3 \rightarrow e_3 \\
\text{(t-if3)} & \quad e_1 \rightarrow e'_1 \\
\text{(t-while)} & \quad \text{while } e_1 \text{ do } e_2 \rightarrow \text{if } e_1 \text{ then } (e_2; \text{while } e_1 \text{ do } e_2) \text{ else skip} \\
\text{(t-parallel1)} & \quad e_1 \text{ op } e_2 \rightarrow e'_1 \text{ op } e_2 \\
\text{(t-parallel2)} & \quad e_2 \rightarrow e'_2 \\
\text{(t-lock)} & \quad \text{lock } m \rightarrow \text{true} \\
\text{(t-unlock)} & \quad \text{unlock } m \rightarrow \text{false}
\end{align*}
\]
Now can make the Ordered 2PL Discipline precise

Say \( e \) obeys the discipline if for any (finite or infinite)

\[
e \xrightarrow{a_1} e_1 \xrightarrow{a_2} e_2 \xrightarrow{a_3} \ldots
\]

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

... and make the guaranteed properties precise

Say \( e_1, \ldots, e_k \) are serializable if for any initial store \( s \), if

\[
((e_1 \ldots e_k), s, M_0) \rightarrow^* (e', s', M') \not\rightarrow
\]

then for some permutation \( \pi \) we have \( (e_{\pi(1)} \ldots e_{\pi(k)}, s, M_0) \rightarrow^* (e'', s', M') \).

Say they are deadlock-free if for any initial store \( s \), if

\[
((e_1 \ldots e_k), s, M_0) \rightarrow^* (e', s', M) \not\rightarrow
\]

then not \( e' \text{ lock } m \rightarrow e'' \), i.e. \( e' \) does not contain any blocked \text{lock } m subexpressions.

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

Theorem

Conjecture 28 If each \( e_i \) obeys the discipline, then \( e_1, \ldots, e_k \) are serializable and deadlock-free.

(may be false!)

Proof strategy: Consider a (derivation of a) computation

\[
((e_1 \ldots e_k), s, M_0) \rightarrow (\tilde{e}_1, s_1, M_1) \rightarrow (\tilde{e}_2, s_2, M_2) \rightarrow \ldots
\]

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

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

Language Properties

(Obviously!) don’t have Determinacy.

Still have Type Preservation.

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

Typing and type inference is scarcely changed.

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

Exercise 36 ★★ Are the mutexes specified here similar to those described in CEDS?

Exercise 37 ★★ 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 38 ★★★★ Prove the Conjecture about it.

Exercise 39 ★★★ 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.
8 Semantic Equivalence

Slide 239

Semantic Equivalence

Slide 240

In what sense are these two expressions the same?
They have different abstract syntax trees.
They have different reduction sequences.
But, you’d hope that in any program you could replace one by the other without affecting the result...

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

How about \( (l := 0; 4) \simeq (l := 1; 3+1l) \)?
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[\_] = +1l

\[ C[l := 0; 4] = (l := 0; 4)+1l \]
\[ \not\simeq \]
\[ C[l := 1; 3+1l] = (l := 1; 3+1l)+1l \]

On the other hand, consider

\( (l :=1l + 1); (l :=1l - 1) \simeq (l :=1l) \)

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

\( e_1; (e_2; e_3) \simeq (e_1; e_2); e_3 \)
\( (if \ e_1 \ then \ e_2 \ else \ e_3); e \simeq if \ e_1 \ then \ e_2; e \ else \ e_3; e \)
\( e; (if \ e_1 \ then \ e_2 \ else \ e_3) \simeq if \ e_1 \ then \ e; e_2 \ else \ e; e_3 \)
\( e; (if \ e_1 \ then \ e_2 \ else \ e_3) \simeq if \ e; e_1 \ then \ e_2 \ else \ e_3 \)

\( \text{let val } x = \text{ ref } 0 \text{ in } \text{ fn } y: \text{ int } \Rightarrow (x := !x + y); !x \)
\( \simeq \)
\( \text{let val } x = \text{ ref } 0 \text{ in } \text{ fn } y: \text{ int } \Rightarrow (x := !x - y); (0-!x) \)
Temporarily extend L3 with pointer equality

\[ op ::= ... \]

\[ \Gamma \vdash e_1 : \text{T ref} \]

\[ \Gamma \vdash e_2 : \text{T ref} \]

\[ \Gamma \vdash e_1 = e_2 : \text{bool} \]

\[ (op \equiv) \quad \langle \ell = \ell', s \rangle \rightarrow \langle b, s \rangle \quad \text{if } b = (\ell = \ell') \]

---

\[ f = \quad \text{let val } x = \text{ref 0 in} \]
\[ \quad \text{let val } y = \text{ref 0 in} \]
\[ \quad \text{fn } z : \text{int ref } \Rightarrow \text{if } z = x \text{ then } y \text{ else } x \]
\[ \quad \text{end end} \]

\[ g = \quad \text{let val } x = \text{ref 0 in} \]
\[ \quad \text{let val } y = \text{ref 0 in} \]
\[ \quad \text{fn } z : \text{int ref } \Rightarrow \text{if } z = y \text{ then } y \text{ else } x \]
\[ \quad \text{end end} \]

\[ f \equiv g \]

---


---

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

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

1. programs that result in observably-different values (in some initial store) must not be equivalent
   \[ (\exists s, s_1, s_2, v_1, v_2, e_1, s \rightarrow^* (e_1, s) \wedge (e_2, s) \rightarrow^* (v_2, s_2) \wedge v_1 \neq v_2) \Rightarrow e_1 \neq e_2 \]

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

3. \( \simeq \) must be an equivalence relation
   \[ e \simeq e, e_1 \simeq e_2 \Rightarrow e_2 \simeq e_1, e_1 \simeq e_2 \simeq e_3 \Rightarrow e_1 \simeq e_3 \]

4. \( \simeq \) must be a congruence
   if \( e_1 \simeq e_2 \) then for any context \( C \) we must have \( C[e_1] \simeq C[e_2] \)

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

Semantic Equivalence for L1

Consider Typed L1 again.

Define \( e_1 \simeq^T \Gamma e_2 \) to hold iff 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) \( (e_1, s) \rightarrow^\omega \) and \( (e_2, s) \rightarrow^\omega \), or

(b) for some \( v, s' \) we have \( (e_1, s) \rightarrow^* (v, s') \) and \( (e_2, s) \rightarrow^* (v, s') \).

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.

If \( T = \text{unit} \) then \( C = \_! \).
If \( T = \text{bool} \) then \( C = \text{if} \_ \text{then} ! \text{else} ! \).
If \( T = \text{int} \) then \( C = l := \_! \).

Congruence for Typed L1

The L1 contexts are:

\[
C ::= \_ \mid \text{op } e_2 \mid e_1 \text{ op } \_
\]
\[
\text{if } \_ \text{then } e_2 \text{ else } e_3 \mid \text{if } e_1 \text{ then } \_ \text{else } e_3 \mid \\
\text{if } e_1 \text{ then } e_2 \text{ else } \_
\]
\[
\ell ::= \_
\]
\[
\_ : e_2 \mid e_1 : \_
\]
\[
\text{while } \_ \text{do } e_2 \mid \text{while } e_1 \text{ do } \_
\]

Say \( \simeq^T \Gamma \) has the congruence property if whenever \( e_1 \simeq^T \Gamma e_2 \) we have, for all \( C \) and \( T' \), if \( \Gamma \vdash C[e_1]; T' \) and \( \Gamma \vdash C[e_2]; T' \) then \( C[e_1] \simeq^T \Gamma C[e_2] \).
By case analysis, looking at each L1 context in turn. We give only one case here, leaving the others for the reader.

Consider the possible reduction sequences of a state \( \langle \ell := e, s \rangle \). For each such reduction sequence, deduce what behaviour of \( e \) was involved.

There are two possibilities:

1. There is an infinite reduction sequence with \( e \rightarrow^* s \)

2. For some \( v, s' \), we have \( \langle \ell := e, s \rangle \rightarrow^* \langle v, s' \rangle \)

Now (using Determinacy), for any \( e \) and \( s \) we have either

**Case:** \( \langle \ell := e, s \rangle \rightarrow^* s \), i.e.

\[ \langle \ell := e, s \rangle \rightarrow \langle e_1, s_1 \rangle \rightarrow \langle e_2, s_2 \rangle \rightarrow \ldots \]

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

\[ \langle e, s \rangle \rightarrow \langle e_1, s_1 \rangle \rightarrow \langle e_2, s_2 \rangle \rightarrow \ldots \]

and \( e_1 = (\ell := e_1), e_2 = (\ell := e_2) \),...

**Case:** \( \neg(\langle \ell := e, s \rangle \rightarrow^* s) \), i.e.

\[ \langle \ell := e, s \rangle \rightarrow \langle e_1, s_1 \rangle \rightarrow \langle e_2, s_2 \rangle \rightarrow \ldots \rightarrow \langle e_k, s_k \rangle \]

hence all these must be instances of (assign2) except the last, which must be an instance of (assign1), with

\[ \langle e, s \rangle \rightarrow \langle e_1, s_1 \rangle \rightarrow \langle e_2, s_2 \rangle \rightarrow \ldots \rightarrow \langle e_{k-1}, s_{k-1} \rangle \]

and \( e_1 = (\ell := e_1), e_2 = (\ell := e_2), \ldots, e_{k-1} = (\ell := e_{k-1}) \) and for some \( n \) we have \( e_{k-1} = n, e_k = \text{skip} \), and \( s_k = s_{k-1} + \{\ell \mapsto n\} \).

Now, if \( \langle \ell := e, s \rangle \rightarrow^* s \), by the above there is an infinite reduction sequence for \( \langle e, s \rangle \), so by \( e \sim_{T_1} e' \) 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 \( \neg ((\ell := e, s) \rightarrow^{\omega}) \) then by the above there is some \( n \) and \( s_{k-1} \) such that \( (e, s) \rightarrow^* (n, s_{k-1}) \) and \( (\ell := e, s) \rightarrow^* (\text{skip}, s_{k-1} + \{ \ell \mapsto n \}) \). By \( e \models^\mathcal{F} \! e' \) we have \( (e', s) \rightarrow^* (n, s_{k-1}) \). Then using (assign1) \( (\ell := e', s) \rightarrow^* (\ell := n, s_{k-1}) \rightarrow^* (\text{skip}, s_{k-1} + \{ \ell \mapsto n \} = (e_k, s_k) \) as required.

\[
\ell := \ell + 1; \text{if } (\ell := \ell + 1) \rangle \models^\text{int} (\ell := \ell + 1) \models^\text{int} (\ell := \ell + 1) \models^\text{int} (\ell := \ell + 1)
\]

And the general laws?

**Conjecture 30** \( e_1; (e_2; e_3) \models^\mathcal{F} \! (e_1; e_2; e_3) \) for any \( e_1, e_2 \) and \( e_3 \) such that \( \Gamma \models e_1 : \text{unit}, \Gamma \models e_2 : \text{unit} \), and \( \Gamma \models e_3 : T \).

**Conjecture 31** \( (\text{if } e_1 \text{ then } e_2 \text{ else } e_3) ; e \models^\mathcal{F} \! (\text{if } e_1 \text{ then } e_2 \text{ else } e_3 ; e) \) for any \( e, e_1, e_2, e_3 \) such that \( \Gamma \models e_1 : \text{bool}, \Gamma \models e_2 : \text{unit}, \Gamma \models e_3 : T \).

**Conjecture 32** \( (e ; (\text{if } e_1 \text{ then } e_2 \text{ else } e_3)) \models^\mathcal{F} \! (e ; (\text{if } e_1 \text{ then } e_2 \text{ else } e_3 ; e)) \) for any \( e_1, e_2, e_3 \) such that \( \Gamma \models e : \text{unit}, \Gamma \models e_1 : \text{bool}, \Gamma \models e_2 : T \), and \( \Gamma \models e_3 : T \).

**Q:** Is a typed expression \( \Gamma \models e : T \), e.g.

\[
\text{intref} \models \text{if } ! ! \geq 0 \text{ then skip else (skip; } l := 0 : \text{unit}):
\]

1. a list of tokens \( \{ \text{IF, DEREF, LOC "1", GTEQ, } \ldots \} \);
2. an abstract syntax tree

\[
\vdash \frac{T}{T}
\]

3. the function taking store \( s \) to the reduction sequence

\[ (e, s) \rightarrow (e_1, s_1) \rightarrow (e_2, s_2) \rightarrow \ldots; \text{or} \]

4. the equivalence class \( \{ e' \mid e \models^\mathcal{F} \! e' \} \)

- the partial function \( [e]_1 \) that takes any store \( s \) with \( \text{dom}(s) = \text{dom}(\Gamma) \) and either is undefined, if \( (e, s) \rightarrow^{\omega} \), or is \( (v, s'), \) if \( (e, s) \rightarrow^* (v, s') \)

(the Determinacy theorem tells us that this is a definition of a function).

**Slide 254**

**Suppose** \( \Gamma \models e_1 : \text{unit} \) and \( \Gamma \models e_2 : \text{unit} \).

When is \( e_1 ; e_2 \models^\text{int} \! e_2 ; e_1 \)?

A sufficient condition: they don’t mention any locations (but not necessary... e.g. if \( e_1 \) does but \( e_2 \) doesn’t)
8.1 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.

**Definition 33** 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) $\langle C[e_1], \{} \rangle \rightarrow^* \omega$ and $\langle C[e_2], \{} \rangle \rightarrow^*$, or

(b) for some $s_1$ and $s_2$ we have $\langle C[e_1], \{} \rangle \rightarrow^* \langle \text{skip}, s_1 \rangle$ and $\langle C[e_2], \{} \rangle \rightarrow^* \langle \text{skip}, s_2 \rangle$.

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.

8.2 Exercises

**Exercise 40 ★★** Prove some of the other cases of the Congruence theorem for semantic equivalence in L1.

**Exercise 41 ★★** 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 $e_1; e_2 \bowtie^*_\Gamma e_2; e_1$ where $\Gamma = \Gamma_1 \cup \Gamma_2$.

**Exercise 42 ★★** Prove that the programs $l : \text{int} \vdash l := 0 : \text{unit}$ and $l : \text{int} \vdash l := 1 : \text{unit}$, considered as L3 programs, are not contextually equivalent. **Hint:** find a context that will diverge for one of them, but not for the other.

9 Epilogue
What can you use semantics for?

1. to understand a particular language — what you can depend on as a programmer; what you must provide as a compiler writer

2. as a tool for language design:
   (a) for clean design
   (b) for expressing design choices, understanding language features and how they interact.
   (c) for proving properties of a language, eg type safety, decidability of type inference.

3. as a foundation for proving properties of particular programs
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.

(* 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. *)

val 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 *)
(* *****************)
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.

```java
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("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\n" + 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 equation reasoning, e.g.

\[
\begin{align*}
e &= e_1 \text{ by } \ldots \\
  &= e_2 \text{ by } \ldots \\
  &= e_3 \text{ as } \ldots
\end{align*}
\]

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 “⇒” or “∵” 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.

\[ P \land Q \quad P \text{ and } Q \text{ are both true} \]
\[ P \lor Q \quad P \text{ is true, or } Q \text{ is true, or both are true} \]
\[ \neg P \quad P \text{ is not true} \]
\[ P \Rightarrow Q \quad \text{if } P \text{ is true then } Q \text{ is true} \]
\[ P \iff Q \quad P \text{ is true exactly when } Q \text{ is true} \]
\[ \forall x \in S.P(x) \quad \text{for all } x \text{ in } S, P \text{ holds of } x \]
\[ \exists x \in S.P(x) \quad \text{there exists an } x \text{ in } S \text{ such that } P \text{ holds of } x \]

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.
### C.2.3 How to Prove a Formula

For each of the logical connectives, I’ll explain how to handle them.

- **∀x ∈ 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 xs somewhere in it, is true no matter what value in S x takes on. Often the “∈ S” is left out. For example, in a discussion of lists, you might be asked to prove ∀.length l > 0 ⇒ ∃x. 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 ∀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 ∀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-conversion¹ to change the formula you want to prove to ∀y ∈ 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.

---

¹Alpha-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.
Another alternative is to prove the contrapositive: \( \neg P \Rightarrow \neg Q \), which is equivalent to it.

To reason by contradiction you assume that \( \neg P \) is not true, and derive a contradiction (that is, you prove \( \neg P \Rightarrow \neg Q \)).

Here’s an example of this, which you’ll run into later this year: the undecidability of the halting problem can be rephrased as \( \neg \exists x \in RM. \ x \ solves \ the \ halting \ problem \), 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.

\[ \exists x \in S.P(x) \] 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, you want to prove

\[ \forall x, y \in \mathbb{R}. x < y \land \sin x < 0 \land \sin y > 0 \Rightarrow \exists z. x < z \land z < y \land \sin z = 0 \]

where \( \mathbb{R} \) is the set of real numbers. By the time you get to dealing with the \( \exists z. x < z \land z < y \land \sin z = 0 \), 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.

\[ P \Rightarrow Q \] 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 \( \neg P \lor Q \), 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 \( \neg P \lor Q \). 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.

\[ P \iff Q \] 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.

If is usually used in two main ways: one is where the equivalence is due to one formula being a definition of another. For example, \( A \subseteq B \iff (\forall x. x \in A \Rightarrow x \in B) \) 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 \( \text{fact} \):

\[
\text{fun fact } 0 = 1 \\
| \text{fact } n = n * \text{fact } (n - 1)
\]

Since in SML whole numbers are integers (both positive and negative) you may be asked to prove: \( \text{fact } x \) terminates \( \Leftrightarrow x \geq 0 \). The standard way to do this is us the equivalence \( P \iff Q \) is equivalent to \( P \Rightarrow Q \land Q \Rightarrow P \). And so you’d prove that \( (\text{fact } x \text{ terminates } \Rightarrow x \geq 0) \land (x \geq 0 \Rightarrow \text{fact } x \text{ terminates}) \).

\( \neg P \) 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 \( \neg \exists x \in RM. \ x \ solves \ the \ halting \ problem \), 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.

\[ P \land Q \] 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 \( Q \).

\[ P \lor Q \] This says “\( P \) is true or \( Q \) is true”. This is inclusive or: if \( P \) and \( Q \) are both true, then \( P \lor Q \) 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 \( (5 \mod 2 = 0) \lor (5 \mod 2 = 1) \), 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

\[
\forall P \in ML. \quad P \text{ is properly typed } \Rightarrow \quad (\text{the evaluation of } P \text{ runs forever}) \lor (P \text{ evaluates to a value})
\]

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 \( P \lor Q \) 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.

\[ \forall x \in S. P(x) \] 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 \( P(v) \).

\[ \exists x \in S. P(x) \] 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 it 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 exists, 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.

\[ \neg P \] 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 \( Q \Rightarrow P \sim \neg P \Rightarrow \neg Q \), you have now proved \( \neg P \Rightarrow \neg Q \). You already know \( \neg P \), so you use modus ponens\(^2\) 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.

\[ P \land Q \] 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.

\[ P \lor Q \] 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, verses the way you prove it. They are in a way almost opposites. For example, in proving \( P \land Q \), 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 \( |uv| \leq n \), \( |v| \geq 1 \), and for all \( i \geq 0 \), \( uv^iw \) 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.

\[
\forall L \in \text{RegularLanguages}. \\
\exists n. \forall z \in L. |z| \geq n \Rightarrow \\
\exists u v w. z = uvw \land |uv| \leq n \land |v| \geq 1 \land \\
\forall i \geq 0. uv^iw \in L
\]

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

\(^2\)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 \( a_1a_2\ldots a_m \), where \( m \geq n \). Consider the states that \( M \) is in during the processing of the first \( n \) symbols of \( z \), \( a_1a_2\ldots a_n \). 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 \( u = a_1a_2\ldots a_j \). This represents the part of the string that gets you to state \( s \) the first time. Let \( v = a_{j+1}\ldots a_k \). This represents the loop that takes you from \( s \) and back to it again. Let \( w = a_{k+1}\ldots a_m \), 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 \( \land \) 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 \( L = \{0^i | i \text{ is an integer, } i \geq 1 \} \) 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 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 \( z \), 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 \( uv^{i+1} w \in L \). [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. Technically, this means that

\[ A_1 \land A_2 \land \ldots A_n \Rightarrow B_1 \lor B_2 \lor \ldots B_m \]  

where \( A_1, A_2, \ldots A_n \) are the formulas in \( \Gamma \), and \( B_1, B_2, \ldots B_n \) 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 \( \lor \) connective, there can be any number of formulas on the right hand side of the \( \Rightarrow \).

For each logic connective, 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 \( \Rightarrow \), 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 \( \Rightarrow \)) or proving it (for formulas on the right side of the \( \Rightarrow \)).

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 \( \Gamma \Rightarrow \Delta \) 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 \( \Rightarrow \), 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:

\[
\frac{\Gamma \Rightarrow \Delta, P \quad \Gamma \Rightarrow P, \Gamma \Rightarrow \Delta}{\Gamma \Rightarrow \Delta} \text{ (cut)}
\]

The \( \Delta, P \) in the first sequent in the hypotheses means that to the right of the \( \Rightarrow \) 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 \( \Rightarrow \) 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 \( \Rightarrow \) are shown in the rules (\( \neg \Rightarrow \)) and (\( \Rightarrow \Rightarrow \)) below.

\[
\forall x \in S.P(x) \quad \text{The two rules for universally quantified formulas are:}
\]

\[
\frac{P(v), \Gamma \Rightarrow \Delta}{\forall x, P(x), \Gamma \Rightarrow \Delta} \text{ (\( \forall \))}
\]

\[
\frac{\Gamma \Rightarrow \Delta, P(x)}{\Gamma \Rightarrow \Delta, \forall x.P(x)} \text{ (\( \forall \))}
\]

\[
3\text{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 \( \Rightarrow \). You will see something similar in Semantics, where it separates assumptions (of the types of variables) from something that they allow you to prove.}
\]

\[
4\text{I won’t mention \( \neg \Rightarrow \) here: as \( P \Rightarrow Q \) is equivalent to \( P \Rightarrow Q \land Q \Rightarrow P \), 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:

\[
\begin{align*}
P(x), \Gamma \vdash \Delta & \quad \frac{\Gamma \vdash \exists x.P(x), \Gamma \vdash \Delta}{\exists x \in S.P(x)} (\exists l) \\
& \quad \frac{\Gamma \vdash \Delta, \exists x.P(x)}{\Gamma \vdash \Delta, \exists x.P(x) (\exists r)}
\end{align*}
\]

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:

\[
\begin{align*}
\Gamma \vdash \Delta, P & \quad \frac{\Gamma, \neg P, \Gamma \vdash \Delta \quad \neg l}{P, \Gamma \vdash \Delta} \\
\neg P, \Gamma & \vdash \Delta \quad \frac{P, \Gamma \vdash \Delta, \neg P, \Gamma \vdash \Delta \quad \neg r}{\Gamma \vdash \Delta, \neg P}
\end{align*}
\]

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.
The two rules for implication are:

\[ \frac{\Gamma \vdash \Delta, P, Q \vdash \Delta}{P \Rightarrow Q, \Gamma \vdash \Delta} (\Rightarrow l) \quad \frac{P, \Gamma \vdash \Delta, Q}{\Gamma \vdash \Delta, Q \Rightarrow P} (\Rightarrow r) \]

The rule (\Rightarrow l) 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.

The rules for conjunction are:

\[ \frac{P, Q, \Gamma \vdash \Delta}{P \land Q, \Gamma \vdash \Delta} (\land l) \quad \frac{\Gamma \vdash \Delta, P \quad \Gamma \vdash \Delta, Q}{\Gamma \vdash \Delta, P \land Q} (\land r) \]

Both of these rules are easily explained by the intuition above. The left rule (\land l) says that when you use \( P \land Q \), you can use \( P \) and \( Q \). The right rule says that to prove \( P \land Q \) 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.

The rules for disjunction are:

\[ \frac{P, \Gamma \vdash \Delta \quad Q, \Gamma \vdash \Delta}{P \lor Q, \Gamma \vdash \Delta} (\lor l) \quad \frac{\Gamma \vdash \Delta, P \quad \Gamma \vdash \Delta, Q}{\Gamma \vdash \Delta, P \lor Q} (\lor r) \]

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 \( P \lor Q \), you need to prove it using \( P \), then prove it using \( Q \). The right rule says that in order to prove \( P \lor Q \), 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.