Semantics of Programming Languages

Computer Science Tripos, Part 1B

2004–5

Peter Sewell

Computer Laboratory

University of Cambridge

Time-stamp: <2005-01-08 14:37:57 pes20>

Contents

Syllabus

Lecturer: Dr P.M. Sewell

No. of lectures: 12

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. [1 lecture]
- **Types.** Introduction to formal type systems. Typing for the simple imperative language. Statements of desirable properties. [1 lecture]
- **Induction.** Review of mathematical induction. Abstract syntax trees and structural induction. Rule-based inductive definitions and proofs. Proofs of type safety properties. [2 lectures]
- **Functions.** Call-by-name and call-by-value function application, semantics and typing. Local recursive definitions. [2 lectures]
- **Data.** Semantics and typing for products, sums, records, references. [1 lecture]
- **Semantic equivalence.** Semantic equivalence of phrases in a simple imperative language, including the congruence property. Examples of equivalence and non-equivalence. [1 lecture]
- **Concurrency.** Shared variable interleaving. Semantics for simple mutexes; a serializability property. [1 lecture]
- **Subtyping.** Record subtyping and simple object encoding. [1 lecture]
- **Low-level semantics.** Monomorphic typed assembly language. [1 lecture]

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 books

Hennessy, M. (1990). *The semantics of programming languages.* Wiley. Out of print, but available on the web at `http://www.cogs.susx.ac.uk/users/matthewh/semnotes.ps.gz`
* Pierce, B. C. (2002). *Types and programming languages.* MIT Press.
Winskel, G. (1993). *The formal semantics of programming languages.* MIT Press.

Learning Guide

Books:

- Hennessy, M. (1990). *The Semantics of Programming Languages.* Wiley. Out of print, but available on the web at `http://www.cogs.susx.ac.uk/users/matthewh/semnotes.ps.gz`.

  Introduces many of the key topics of the course.

- Pierce, B. C. (2002) *Types and Programming Languages.* MIT Press.

  This is a graduate-level text, covering a great deal of material on programming language semantics. The first half (through to Chapter 15) is relevant to this course, and some of the later material relevant to the Part II Types course.

- Winskel, G. (1993). *The Formal Semantics of Programming Languages.* MIT Press.

  An introduction to both operational and denotational semantics; recommended for the Part II Denotational Semantics course.

Further reading:

- Plotkin, G. D.(1981). A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University.

  These notes first popularised the 'structural' approach to operational semantics—the approach emphasised in this course—but couched solely in terms of transition relations ('small-step' semantics), rather than evaluation relations ('big-step', 'natural', or 'relational' semantics). Although somewhat dated and hard to get hold of (the Computer Laboratory Library has a copy), they are still a mine of interesting examples.

- The two essays:
  Hoare, C. A. R.. Algebra and Models.
  Milner, R. Semantic Ideas in Computing.
  In: Wand, I. and R. Milner (Eds) (1996). *Computing Tomorrow.* CUP.

  Two accessible essays giving somewhat different perspectives on the semantics of computation and programming languages.

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

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.dina.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 most of these. Exercises marked ★★ may need a little more thought – both proofs and some implementation-related; you should do some of each. Exercises marked ★★★ may need material beyond the notes, and/or be quite time-consuming.

Tripos questions: This is a new course, first given in 2002–2003. The 2003 and 2004 questions (Paper 5 Question 11 and Paper 6 Question 12, and Paper 5 Question 11 and Paper 6 Question 12 respectively) are directly relevant, and there is an additional mock question on the course web page. The previous course used a slightly different form of operational semantics, 'big-step' instead of 'small-step' (see Page 76 of these notes), and different example languages, so the notation in most earlier questions may seem unfamiliar at first sight.

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

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

These questions depend on material which is no longer in this course (complete partial orders, continuations, or bisimulation – see the Part II Denotational Semantics and Topics in Concurrency courses): 2001 Paper 6

Question 9, 2000 Paper 6 Question 9, 1997 Paper 6 Question 12, 1996 Paper 6 Question 12, 1995 Paper 5 Question 12, 1994 Paper 8 Question 12, 1994 Paper 9 Question 12, 1993 Paper 8 Question 10, 1993 Paper 9 Question 10.

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

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

Summary of Notation

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

Logic and Set Theory

$\Phi \wedge \Phi'$	and
$\Phi \vee \Phi'$	or
$\Phi \Rightarrow \Phi'$	implies
$\neg\ \Phi$	not
$\forall\ x.\Phi(x)$	for all
$\exists\ x.\Phi(x)$	exists
$a\ \in\ A$	element of
$\{a_1, ..., a_n\}$	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

Finite Partial Functions

$\{a_1 \mapsto b_1, ..., a_n \mapsto b_n\}$	finite partial function mapping each a_i to b_i
$\mathrm{dom}(s)$	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 Γ extended with $\{x \mapsto T\}$ – only used where x not in $\mathrm{dom}(\Gamma)$
Γ, Γ'	the finite partial function which is the union of Γ and Γ – only used where they have disjoint domains
$\{l_1 \mapsto n_1, ..., l_k \mapsto n_k\}$	an L1 or L2 store – the finite partial function mapping each l_i to n_i
$\{l_1 \mapsto v_1, ..., l_k \mapsto v_k\}$	an L3 store – the finite partial function mapping each l_i to v_i
$l_1{:}\mathsf{intref}, ..., l_k{:}\mathsf{intref}$	an L1 type environment – the finite partial function mapping each l_i to intref
$\ell{:}\mathsf{intref}, ..., x{:}T, ...$	an L2 type environment
$\ell{:}T_{loc}, ..., x{:}T, ...$	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

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

Particular sets

$\mathbb{B} = \{\textsf{\textbf{true}}, \textsf{\textbf{false}}\}$	the set of booleans
$\mathbb{L} = \{l, l_1, l_2, ...\}$	the set of locations
$\mathbb{Z} = \{.., -1, 0, 1, ...\}$	the set of integers
$\mathbb{N} = \{0, 1, ...\}$	the set of natural numbers
$\mathbb{X} = \{\mathrm{x}, \mathrm{y}, ...\}$	the set of L2 and L3 variables
$\mathbb{LAB} = \{\mathrm{p}, \mathrm{q}, ...\}$	the set of record labels
$\mathbb{M} = \{\mathrm{m}, \mathrm{m}_0, \mathrm{m}_1, ...\}$	the set of mutex names
T	the set of all types (in whichever language)
$\mathrm{T}_{\mathrm{loc}}$	the set of all location types (in whichever language)
L_1	the set of all L1 expressions
TypeEnv	the set of all L1 type environments, finite partial functions from $\mathbb{L}$ to $\mathbb{Z}$
TypeEnv2	the set of all L2 type environments, the finite partial functions from $\mathbb{L} \cup \mathbb{X}$ to $\mathrm{T}_{\mathrm{loc}} \cup \mathrm{T}$ such that $\forall\, \ell \in \mathrm{dom}(\Gamma).\Gamma(\ell) \in \mathrm{T}_{\mathrm{loc}}$ and $\forall\, x \in \mathrm{dom}(\Gamma).\Gamma(x) \in \mathrm{T}$
$\mathbb{A}$	thread actions

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)
$T \in \mathrm{T}$	type (of whichever language)
$T_{loc} \in \mathrm{T}_{\mathrm{loc}}$	location type (of whichever language)
Γ	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
Φ	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
σ	substitution
$lab \in \mathbb{LAB}$	record label
E	evaluation context
C	arbitrary context
π	permutation of natural numbers
$m \in \mathbb{M}$	mutex name
M	state of all mutexes (a function $M{:}\mathbb{M} \longrightarrow \mathbb{B}$)
a	thread action, for $a \in \mathbb{A}$

Other

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

1 Introduction

Semantics of Programming Languages

Peter Sewell

1B, 12 lectures

2004–5

In this course we will take a close look at programming languages. We will focus on how one can 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).

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

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

This is true not just for the major programming languages, but also for intermediate languages (JVM, CLR), and the many, many scripting and command languages, that have often been invented on-the-fly without sufficient thought. How many of you will do language design? lots!

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.*

Warmup

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

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

x++ + ++x
```

from Micro to Macro

- simple evaluation order
- what can be stored
- evaluation strategy (call-by-value, call-by-name)
- what can be abstracted over; what can be passed around
- what can/should type systems guarantee at compile-time
- ...

Various different approaches have been used for expressing semantics.

Styles of Semantic Definitions

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

...Static and dynamic semantics...

Operational: define the meaning of a program in terms of the computation steps it takes in an idealised 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.

All these are *dynamic* semantics, describing behaviour in one way or another. In contrast the *static* semantics of a language describes its compile-time typechecking.

'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.

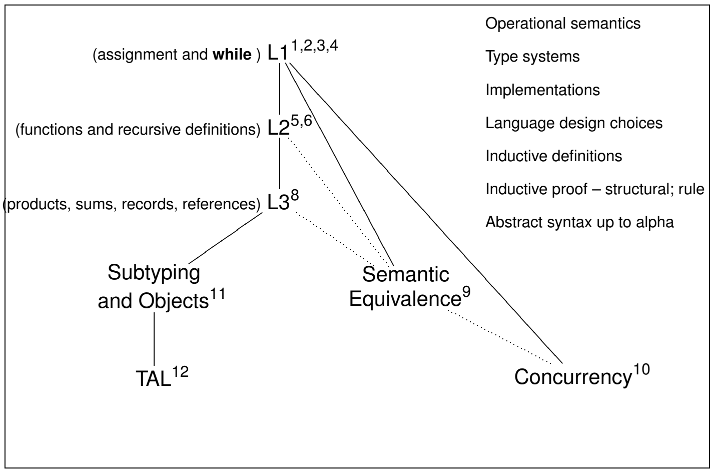
What's this course?

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

- Semantic Equivalence
- Concurrency
- Subtyping and Objects
- Low-level Semantics (Typed Assembly Language)


(assignment and **while**) L1[1,2,3,4]
(functions and recursive definitions) L2[5,6]
(products, sums, records, references) L3[8]
Subtyping and Objects[11]
Semantic Equivalence[9]
TAL[12]
Concurrency[10]

Operational semantics

Type systems

Implementations

Language design choices

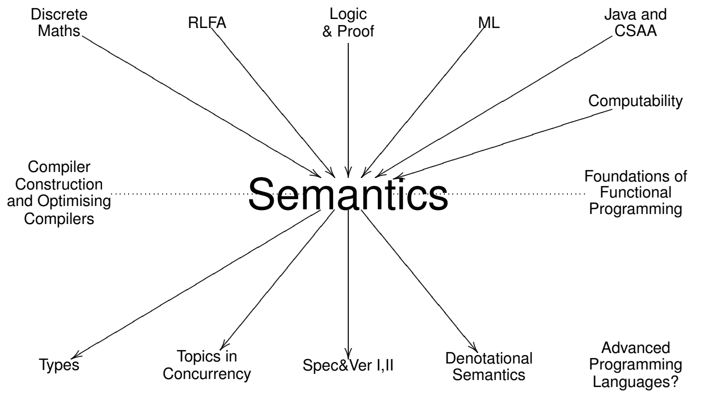
Inductive definitions

Inductive proof – structural; rule

Abstract syntax up to alpha

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.

The Big Picture (a Semantic-O-Centric view...)



Admin

- This is a fairly new course, so:
  - ˇ please let me know of typos, and if it is too fast/too slow/too interesting/too dull (please complete the on-line feedback at the end)
  - ˇ not all previous Tripos questions are relevant (see the notes)
- Exercises in the notes.
- Implementations on web.
- Books (Hennessy, Pierce, Winskel)

2 A First Imperative Language

L1

L1 – Example

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

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

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

L1 – Syntax

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

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

Expressions

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

Write L_1 for the set of all expressions.

Points to note:

- we'll return later to *exactly* what the set L_1 is when we talk about abstract syntax
- unbounded integers

- abstract locations – can't do pointer arithmetic on them
- untyped, so have nonsensical expressions like $3 +$ **true**
- what kind of grammar is that?
- don't have expression/command distinction
- doesn't much matter what basic operators we have
- carefully distinguish metavariables $b, n, \ell,\ op\ , e$ etc. from program locations l etc..

2.1 Operational Semantics

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

Transition systems

A *transition system* consists of

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

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

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 ε transitions) except (a) it can have infinitely many states, and (b) we don't specify a start state or accepting states. Sometimes one adds labels (e.g. to represent IO) but mostly we'll just look at the values of terminated states, those that cannot do any transitions.

Some handy auxilary notation:

- $\longrightarrow^*$ is the reflexive transitive closure of $\longrightarrow$, so $c \longrightarrow^* c'$ iff there exist $k \geq 0$ and $c_0, .., c_k$ such that $c = c_0 \longrightarrow c_1 ... \longrightarrow c_k = c'$.
- $\not\longrightarrow$ is a unary predicate (a subset of Config) defined by $c \not\longrightarrow$ iff $\neg\ \exists\ c'.c \longrightarrow c'$.
- The transition relation is *deterministic* if for all states c there is at most one c' such that $c \longrightarrow c'$, ie if $\forall\ c.\forall\ c', c''.(c \longrightarrow c' \wedge c \longrightarrow c'') \implies c' = c''$.

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

L1 Semantics (1 of 4) – Configurations

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

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

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

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

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

- $\forall\ i\ \in \{1, .., n\}.a_i\ \in\ A$ (the domain is a subset of$A$)
- $\forall\ i\ \in \{1, .., n\}.b_i\ \in\ B$ (the range is a subset of B)
- $\forall\ i\ \in \{1, .., n\}, j\ \in \{1, .., n\}.i \neq j \Rightarrow a_i \neq a_j$ (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 dom(f) for the set of elements in the domain of f (things that f maps to something) and ran(f) for the set of elements in the range of f (things that something is mapped to by f). For example, for the s above we have $\mathrm{dom}(s) = \{l_1, l_3\}$ and $\mathrm{ran}(s) = \{7, 23\}$. 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:

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

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

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

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

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

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

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

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

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

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

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

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

We use a strict naming convention for metavariables: n can *only* be instantiated by integers, not by arbitrary expressions, cabbages, or what-have-you.

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

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

Example

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

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

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

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

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

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

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

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

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

L1 Semantics (3 of 4) – store and sequencing

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

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

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

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

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

Example

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

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

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

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

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

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

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

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

Example

If

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

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

then

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

Determinacy

Theorem 1 (L1 Determinacy) *If* $\langle e, s\rangle \longrightarrow \langle e_1, s_1\rangle$ *and* $\langle e, s\rangle \longrightarrow \langle e_2, s_2\rangle$ *then* $\langle e_1, s_1\rangle = \langle e_2, s_2\rangle$.

Proof - see later

Note that top-level universal quantifiers are usually left out – the theorem really says "For all e, s, e_1, s_1, e_2, s_2, if $\langle e, s\rangle \longrightarrow \langle e_1, s_1\rangle$ and $\langle e, s\rangle \longrightarrow \langle e_2, s_2\rangle$ then $\langle e_1, s_1\rangle = \langle e_2, s_2\rangle$".

L1 Implementation

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, rather arbitrarily (really, so they pretty-print trivially). A lower-level implementation would use ML references or, even lower, machine pointers.

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 and mapped to n, exactly one element of the form `(l,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

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

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

The single-step function

Now define the single-step function

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

which takes a configuration `(e,s)` and returns either
`NONE`, if $\langle e, s\rangle \not\longrightarrow$,
or `SOME (e',s')`, if it has a transition $\langle e, s\rangle \longrightarrow \langle e', s'\rangle$.

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

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

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

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

Contrast this code with the semantic rules given earlier.

(op1), (op2)

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

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.

(assign1), (assign2)

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

The many-step evaluation function

Now define the many-step evaluation function

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

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

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

Demo

The full interpreter code is available on the web, in the file `l1.ml`, together with a pretty-printer and the type-checker we will come to soon. You should make it go...

```
(* 2002-11-08 -- Time-stamp: <2004-01-03 16:17:04 pes20>    -*-SML-*- *)
(* Peter Sewell                                                       *)

(* This file contains an interpreter, pretty-printer and type-checker
   for the language L1.  To make it go, copy it into a working
   directory, ensure Moscow ML is available, and type

       mosml -P full l1.ml

   That will give you a MoscowML top level in which these definitions
   are present.  You can then type

      doit ();

   to show the reduction sequence of < l1:=3;!l1 , l1=0  >, and

      doit2 ();

   to run the type-checker on the same simple example; you can try
   other examples analogously.  This file doesn't have a parser for
   l1, so you'll have to enter the abstract syntax directly, eg

      prettyreduce (Seq( Assign ("l1",Integer 3), Deref "l1"), [("l1",0)]);

   This has been tested with Moscow ML version 2.00 (June 2000), but
   should work with any other implementation of Standard ML.   *)


(* *********************)
(* the abstract syntax *)
(* *********************)

type loc = string

datatype oper = Plus | GTEQ

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


(* **********************************)
(* an interpreter for the semantics *)
(* **********************************)

fun is_value (Integer n) = true
  | is_value (Boolean b) = true
  | is_value (Skip) = true
  | is_value _ = false

  (* In the semantics, a store is a finite partial function from
  locations to integers.  In the implementation, we represent a store
  as a list of loc*int pairs containing, for each l in the domain of
  the store, exactly one element of the form (l,n).  The operations

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

  both return NONE if given a location that is not in the domain of
  the store.  This is not a very efficient implementation, but it is
  simple. *)

type store = (loc * int) list

fun lookup ( [], l ) = NONE
  | lookup ( (l',n')::pairs, l) =
    if l=l' then SOME n' else lookup (pairs,l)

fun update'  front [] (l,n) = NONE
 |  update'  front ((l',n')::pairs) (l,n) =
    if l=l' then
        SOME(front @ ((l,n)::pairs) )
    else
        update' ((l',n')::front) pairs (l,n)

fun update (s, (l,n)) = update' [] s (l,n)


  (* now define the single-step function

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

  which takes a configuration (e,s) and returns either NONE, if it has
  no transitions, or SOME (e',s'), if it has a transition (e,s) -->
  (e',s').

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

fun reduce (Integer n,s) = NONE
  | reduce (Boolean b,s) = NONE
  | reduce (Op (e1,opr,e2),s) =
    (case (e1,opr,e2) of
         (Integer n1, Plus, Integer n2) => SOME(Integer (n1+n2), s)   (*op + *)
       | (Integer n1, GTEQ, Integer n2) => SOME(Boolean (n1 >= n2), s)(*op >=*)
       | (e1,opr,e2) => (
         if (is_value e1) then (
             case reduce (e2,s) of
                 SOME (e2',s') => SOME (Op(e1,opr,e2'),s')     (* (op2) *)
               | NONE => NONE )
         else (
             case reduce (e1,s) of
                 SOME (e1',s') => SOME(Op(e1',opr,e2),s')      (* (op1) *)
               | NONE => NONE ) ) )
  | reduce (If (e1,e2,e3),s) =
    (case e1 of
         Boolean(true) => SOME(e2,s)                           (* (if1) *)
       | Boolean(false) => SOME(e3,s)                          (* (if2) *)
       | _ => (case reduce (e1,s) of
                   SOME(e1',s') => SOME(If(e1',e2,e3),s')      (* (if3) *)
                 | NONE => NONE ))
  | reduce (Deref l,s) =
    (case lookup  (s,l) of
          SOME n => SOME(Integer n,s)                          (* (deref) *)
        | NONE => NONE )
  | reduce (Assign (l,e),s) =
    (case e of
         Integer n => (case update (s,(l,n)) of
                           SOME s' => SOME(Skip, s')           (* (assign1) *)
                         | NONE => NONE)
       | _ => (case reduce (e,s) of
                   SOME (e',s') => SOME(Assign (l,e'), s')     (* (assign2) *)
                 | NONE => NONE  ) )
  | reduce (While (e1,e2),s) = SOME( If(e1,Seq(e2,While(e1,e2)),Skip),s) (* (while) *)
  | reduce (Skip,s) = NONE
  | reduce (Seq (e1,e2),s) =
    (case e1 of
        Skip => SOME(e2,s)                                     (* (seq1) *)
      | _ => ( case reduce (e1,s) of
                 SOME (e1',s') => SOME(Seq (e1',e2), s')       (* (seq2) *)
               | NONE => NONE ) )


  (* now define the many-step evaluation function

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

  which takes a configuration (e,s) and returns the unique (e',s')
  such that   (e,s) -->* (e',s') -/->.  *)

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

The Java Implementation

Quite different code structure:

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

For comparison, here is a Java implementation – with thanks to Matthew Parkinson. This includes code for type inference (the ML code for which is on Page 33) and printy-printing (in `l1.ml` but not shown above).

Note the different code organisation between the ML and Java versions: the ML has a datatype with a constructor for each clause of the abstract syntax grammar, and `reduce` and `infertype` function definitions that each have a case for each of those constructors; the Java has a subclass of `Expression` for each clause of the abstract syntax, each of which defines `smallStep` and `typecheck` methods.

```
public class L1 {

    public static void main(String [] args) {
        Location l1 = new Location ("l1");
        Location l2 = new Location ("l2");
        Location l3 = new Location ("l3");
        State s1 = new State()
            .add(l1,new Int(1))
            .add(l2,new Int(5))
            .add(l3,new Int(0));

        Environment env = new Environment()
            .add(l1).add(l2).add(l3);

        Expression e =
            new Seq(new While(new GTeq(new Deref(l2),new Deref(l1)),
                      new Seq(new Assign(l3, new Plus(new Deref(l1),new Deref(l3))),
                              new Assign(l1,new Plus(new Deref(l1),new Int(1))))

                              ),
                    new Deref(l3))
                    ;
        try{
            //Type check
            Type t= e.typeCheck(env);
            System.out.println("Program has type: " + t);

            //Evaluate program
            System.out.println(e + "\n \n");
            while(!(e instanceof Value) ){
                e = e.smallStep(s1);
                //Display each step of reduction
                System.out.println(e + "\n \n");
            }
            //Give some output
            System.out.println("Program has type: " + t);
            System.out.println("Result has type: " + e.typeCheck(env));
            System.out.println("Result: " + e);
            System.out.println("Terminating State: " + s1);
        } catch (TypeError te) {
            System.out.println("Error:\n" + te);
            System.out.println("From code:\n" + e);
        } catch (CanNotReduce cnr) {
            System.out.println("Caught Following exception" + cnr);
            System.out.println("While trying to execute:\n " + e);
            System.out.println("In state: \n " + s1);
        }
    }
}

class Location {
    String name;
    Location(String n) {
        this.name = n;
    }
    public String toString() {return name;}
}

class State {
    java.util.HashMap store = new java.util.HashMap();

    //Used for setting the initial store for testing not used by
    //semantics of L1
    State add(Location l, Value v) {
        store.put(l,v);
        return this;
    }

    void update(Location l, Value v) throws CanNotReduce {
        if(store.containsKey(l)) {
            if(v instanceof Int) {
                store.put(l,v);
            }
            else throw new CanNotReduce("Can only store integers");
        }
        else throw new CanNotReduce("Unknown location!");
    }

    Value lookup(Location l) throws CanNotReduce {
        if(store.containsKey(l)) {
            return (Int)store.get(l);
        }
        else throw new CanNotReduce("Unknown location!");
    }
    public String toString() {
        String ret = "[";
        java.util.Iterator iter = store.entrySet().iterator();
        while(iter.hasNext()) {
            java.util.Map.Entry e = (java.util.Map.Entry)iter.next();
            ret += "(" + e.getKey() + " |-> " + e.getValue() + ")";
            if(iter.hasNext()) ret +=", ";
        }
        return ret + "]";
    }
}

class Environment {
    java.util.HashSet env = new java.util.HashSet();

    //Used to initially setup environment, not used by type checker.
    Environment add(Location l) {
        env.add(l); return this;
    }

    boolean contains(Location l) {
        return env.contains(l);
    }
}
class Type {
    int type;
    Type(int t) {type = t;}
    public static final Type BOOL = new Type(1);
    public static final Type INT = new Type(2);
    public static final Type UNIT = new Type(3);
    public String toString() {
        switch(type) {
        case 1: return "BOOL";
        case 2: return "INT";
        case 3: return "UNIT";
        }
        return "???";
    }
}


abstract class Expression {
    abstract Expression smallStep(State state) throws CanNotReduce;
    abstract Type typeCheck(Environment env) throws TypeError;
}

abstract class Value extends Expression {
    final Expression smallStep(State state) throws CanNotReduce{
        throw new CanNotReduce("I'm a value");
    }
}

class CanNotReduce extends Exception{
    CanNotReduce(String reason) {super(reason);}
}

class TypeError extends Exception { TypeError(String reason) {super(reason);}}

class Bool extends Value {
    boolean value;

    Bool(boolean b) {
        value = b;
    }

    public String toString() {
        return value ? "TRUE" : "FALSE";
    }

    Type typeCheck(Environment env) throws TypeError {
        return Type.BOOL;
    }
}

class Int extends Value {
    int value;
    Int(int i) {
        value = i;
    }
    public String toString(){return ""+ value;}

    Type typeCheck(Environment env) throws TypeError {
        return Type.INT;
    }
}

class Skip extends Value {
    public String toString(){return "SKIP";}
    Type typeCheck(Environment env) throws TypeError {
        return Type.UNIT;
    }
}

class Seq extends Expression {
    Expression exp1,exp2;
    Seq(Expression e1, Expression e2) {
        exp1 = e1;
        exp2 = e2;
    }

    Expression smallStep(State state) throws CanNotReduce {
        if(exp1 instanceof Skip) {
            return exp2;
        } else {
            return new Seq(exp1.smallStep(state),exp2);
        }
    }
    public String toString() {return exp1 + "; " + exp2;}

    Type typeCheck(Environment env) throws TypeError {
        if(exp1.typeCheck(env) == Type.UNIT) {
            return exp2.typeCheck(env);
        }
        else throw new TypeError("Not a unit before ';'.");
    }
}

class GTeq extends Expression {
    Expression exp1, exp2;
    GTeq(Expression e1,Expression e2) {
        exp1 = e1;
        exp2 = e2;
    }

    Expression smallStep(State state) throws CanNotReduce {
        if(!( exp1 instanceof Value)) {
            return new GTeq(exp1.smallStep(state),exp2);
        } else if (!( exp2 instanceof Value)) {
            return new GTeq(exp1, exp2.smallStep(state));
        } else {
            if( exp1 instanceof Int && exp2 instanceof Int ) {
                return new Bool(((Int)exp1).value >= ((Int)exp2).value);
            }
            else throw new CanNotReduce("Operands are not both integers.");
        }
    }
    public String toString(){return exp1 + " >= " + exp2;}

    Type typeCheck(Environment env) throws TypeError {
        if(exp1.typeCheck(env) == Type.INT && exp2.typeCheck(env) == Type.INT) {
            return Type.BOOL;
        }
        else throw new TypeError("Arguments not both integers.");
    }
}

class Plus extends Expression {
    Expression exp1, exp2;
    Plus(Expression e1,Expression e2) {
        exp1 = e1;
        exp2 = e2;
    }

    Expression smallStep(State state) throws CanNotReduce {
        if(!( exp1 instanceof Value)) {
            return new Plus(exp1.smallStep(state),exp2);
        } else if (!( exp2 instanceof Value)) {
            return new Plus(exp1, exp2.smallStep(state));
        } else {
            if( exp1 instanceof Int && exp2 instanceof Int ) {
                return new Int(((Int)exp1).value + ((Int)exp2).value);
            }
            else throw new CanNotReduce("Operands are not both integers.");
        }
    }
    public String toString(){return exp1 + " + " + exp2;}

    Type typeCheck(Environment env) throws TypeError {
        if(exp1.typeCheck(env) == Type.INT && exp2.typeCheck(env) == Type.INT) {
            return Type.INT;
        }
        else throw new TypeError("Arguments not both integers.");
    }
}


class IfThenElse extends Expression {
    Expression exp1,exp2,exp3;

    IfThenElse (Expression e1, Expression e2,Expression e3) {
        exp1 = e1;
        exp2 = e2;
        exp3 = e3;
    }

    Expression smallStep(State state) throws CanNotReduce {
        if(exp1 instanceof Value) {
            if(exp1 instanceof Bool) {
                if(((Bool)exp1).value)
                    return exp2;
                else
                    return exp3;
            }
            else throw new CanNotReduce("Not a boolean in test.");
        }
        else {
            return new IfThenElse(exp1.smallStep(state),exp2,exp3);
        }
    }
    public String toString() {return "IF " + exp1 + " THEN " + exp2 + " ELSE " + exp3;}

    Type typeCheck(Environment env) throws TypeError {
        if(exp1.typeCheck(env) == Type.BOOL) {
            Type t = exp2.typeCheck(env);
            if(exp3.typeCheck(env) == t)
                return t;
            else throw new TypeError("If branchs 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");
    }
}
```

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

Language design 1. Order of evaluation

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

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

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

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

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

In this language (call it L1b)

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

For programmers whose first language has left-to-right reading order, left-to-right evaluation is arguably more intuitive than right-to-left. Nonetheless, some languages are right-to-left for efficiency reasons (e.g. OCaml bytecode).

It is important to have the *same* order for all operations, otherwise we certainly have a counter-intuitive language.

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

Sometimes ordering really is not always guaranteed, say for two writes $l := 1; l := 2$. In L1 it is defined, but if we were talking about a setting with a cache (either processors, or disk block writes, or something) we might have to do something additional to force ordering. Similarly if you have concurrency $l := 1 \mid l := 2$. Work on redesigning the Java Memory Model by Doug Lea and Bill Pugh, which involves this kind of question, can be found at `http://www.cs.umd.edu/~pugh/java/memoryModel/`.

One could also underspecify in a language definition but require each implementation to use a consistent order, or require each implementation to use a consistent order for each operator occurrence in the program source code. A great encouragement to the bugs...

Language design 2. Assignment results

Recall

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

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

So

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

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

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

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

Matter of taste?

Another possiblity: return the *old* value, e.g. in ANSI C signal handler installation `signal(n,h)`. Atomicity?

Language design 3. Store initialisation

Recall that

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

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

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

Instead, could

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

These would both be bad design decisions, liable to lead to ghastly bugs, with locations initialised on some code path but not others. Option 1 would be particularly awkward in a richer language where values other than integers can be stored, where there may not be any sensible value to default-initialise to.

Looking ahead, any reasonable type system will rule out, at compile-time, any program that could reach a stuck expression of these forms.

Language design 4. Storable values

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

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

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

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

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

This is annoying – unmotivated *irregularity* – why not allow storage of any value? of locations? of expressions???

Also, store is global....leading to ghastly programming in big code. Will revisit later.

Language design 5. Operators and basic values

Booleans are really not integers (pace C)

How many operators? Obviously want more than just $+$ and $\geq$. But this is semantically dull - in a full language would add in many, in standard libraries.

(beware, it's not completely dull - eg floating point specs! Even the L1 impl and semantics aren't in step.).

Exercise: fix the implementation to match the semantics.

Exercise: fix the semantics to match the implementation.

Expressiveness

Is L1 expressive enough to write interesting programs?

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

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

- yes: we'd like to forbid programs like $3 +$ **false** as early as possible, not wait for a runtime error (which might occur only on some execution paths). We'll do so with a *type system*.

2.2 Typing

L1 Typing

Type Systems

used for

- preventing certain kinds of errors
- structuring programs
- guiding language design

Type systems are also used to provide information to compiler optimisers; 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.

Run-time errors

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

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

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

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

We cannot expect to exclude *all* trapped errors, eg arith overflows, or out-of-memory errors, but certainly want to exclude all untrapped errors.

So, how to do so? Can use runtime checks and compile-time checks – want compile-time where possible.

Formal type systems

Divide programs into the good and the bad...

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

For example (according to the definition coming up):

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

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:

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

Types of locations:

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

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

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

- concretely, $\mathrm{T} = \{\mathsf{int}, \mathsf{bool}, \mathsf{unit}\}$ and $\mathrm{T_{loc}} = \{\mathsf{intref}\}$.
- in this (very small!) language, there is only one type in $\mathrm{T_{loc}}$, so a Γ is (up to isomorphism) just a set of locations. Later, $\mathrm{T_{loc}}$ will be more interesting...
- our semantics only let you store integers, so we have stratified types into T and $\mathrm{T_{loc}}$. If you wanted to store other values, you'd say

$$\begin{array}{lcl}
T & ::= & \mathsf{int} \mid \mathsf{bool} \mid \mathsf{unit} \\
T_{loc} & ::= & T\ \ \mathsf{ref}
\end{array}$$

If you wanted to be able to manipulate references as first-class objects, the typing would be

$$\begin{array}{lcl}
T & ::= & \mathsf{int} \mid \mathsf{bool} \mid \mathsf{unit} \mid T\ \ \mathsf{ref} \\
T_{loc} & ::= & T\ \ \mathsf{ref}
\end{array}$$

and there would be consequent changes (what exactly?) to the syntax and the semantics. This is our first sight of an important theme: type-system-directed language design.

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

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

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

$$\text{(op }+\text{)}\ \frac{\begin{array}{c}\Gamma \vdash e_1{:}\mathsf{int}\\ \Gamma \vdash e_2{:}\mathsf{int}\end{array}}{\Gamma \vdash e_1 + e_2{:}\mathsf{int}} \qquad \text{(op }\geq\text{)}\ \frac{\begin{array}{c}\Gamma \vdash e_1{:}\mathsf{int}\\ \Gamma \vdash e_2{:}\mathsf{int}\end{array}}{\Gamma \vdash e_1 \geq e_2{:}\mathsf{bool}}$$

$$\text{(if)}\ \frac{\begin{array}{c}\Gamma \vdash e_1{:}\mathsf{bool}\\ \Gamma \vdash e_2{:}T\\ \Gamma \vdash e_3{:}T\end{array}}{\Gamma \vdash \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3{:}T}$$

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

Example

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

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

where ∇ is

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

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

$$\text{(assign)}\ \frac{\begin{array}{c}\Gamma(\ell) = \mathsf{intref}\\ \Gamma \vdash e{:}\mathsf{int}\end{array}}{\Gamma \vdash \ell := e{:}\mathsf{unit}}$$

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

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

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

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

$$\text{(seq)}\ \frac{\begin{array}{c}\Gamma \vdash e_1{:}\mathsf{unit}\\ \Gamma \vdash e_2{:}T\end{array}}{\Gamma \vdash e_1; e_2{:}T}$$

$$\text{(while)}\ \frac{\begin{array}{c}\Gamma \vdash e_1{:}\mathsf{bool}\\ \Gamma \vdash e_2{:}\mathsf{unit}\end{array}}{\Gamma \vdash \textbf{while}\ e_1\ \textbf{do}\ e_2{:}\mathsf{unit}}$$

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 dom$(\Gamma) \subseteq$ dom(s) then either e is a value or there exist e', s' such that $\langle e, s\rangle \longrightarrow \langle e', s'\rangle$.*

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

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

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

(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.

For L1 we don't need to type the range of the store, as by definition all stored things are integers.

Type checking, typeability, and type inference

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

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

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

For this type system, though, both are easy.

More Properties

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

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

Also:

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

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

Type inference - Implementation

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

```
datatype type_L1 =
     int
   | unit
   | bool

datatype type_loc =
     intref

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

Now define the type inference function

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

In the semantics, type environments Γ 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)`.

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 )
```

ahem.

The Type Inference Algorithm – `If`

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

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

The Type Inference Algorithm – `Deref`

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

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

Again, the code depends on a uniqueness property (Theorem 7), without which we would have to have `infertype` return a `type_L1 list` of all the possible types.

Demo

Executing L1 in Moscow ML

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

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

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

(watch out for ~`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."*

  Garbage collection? Marshalling? Multi-stage computation?

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 one has typing built into the grammar, with e.g. separate grammatical classes of expressions and commands. As the type systems become more expressive, however, they quickly go beyond what can be captured in context-free grammars. They must then be separated from lexing and parsing, both conceptually and in implementations.

2.3 L1: Collected Definition

Syntax

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

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

Expressions

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

Operational Semantics

Note that for each construct there are some *computation* rules, doing 'real work', and some *context* (or *congruence*) rules, allowing subcomputations and specifying their order.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Typing

Types of expressions:

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

Types of locations:

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

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

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

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

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

$$\text{(op +)} \quad \frac{\begin{array}{c}\Gamma \vdash e_1{:}\mathsf{int} \\ \Gamma \vdash e_2{:}\mathsf{int}\end{array}}{\Gamma \vdash e_1 + e_2{:}\mathsf{int}} \qquad \text{(op} \geq\text{)} \quad \frac{\begin{array}{c}\Gamma \vdash e_1{:}\mathsf{int} \\ \Gamma \vdash e_2{:}\mathsf{int}\end{array}}{\Gamma \vdash e_1 \geq e_2{:}\mathsf{bool}}$$

$$\text{(if)} \quad \frac{\begin{array}{c}\Gamma \vdash e_1{:}\mathsf{bool} \\ \Gamma \vdash e_2{:}T \\ \Gamma \vdash e_3{:}T\end{array}}{\Gamma \vdash \textbf{if} \;\; e_1 \;\; \textbf{then} \;\; e_2 \;\; \textbf{else} \;\; e_3{:}T}$$

$$\text{(assign)} \quad \frac{\begin{array}{c}\Gamma(\ell) = \mathsf{intref} \\ \Gamma \vdash e{:}\mathsf{int}\end{array}}{\Gamma \vdash \ell := e{:}\mathsf{unit}}$$

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

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

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

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

2.4 Exercises

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

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

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

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

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

Exercise 7 ★★*Fix the L1 semantics to match the implementation, taking care with the representation of integers.*

Exercise 8 ★*Give a type derivation for $(l_0 := 7); (l_1 := (!l_0 + 2))$ with $\Gamma = l_0{:}\mathsf{intref}, l_1{:}\mathsf{intref}$.*

Exercise 9 ★*Give a type derivation for the e on Page 15 with $\Gamma = l_1{:}\mathsf{intref}, l_2{:}\mathsf{intref}, l_3{:}\mathsf{intref}$.*

Exercise 10 ★*Does Type Preservation hold for the variant language with rules (assign1') and (seq1')? If not, give an example, and show how the type rules could be adjusted to make it true.*

Exercise 11 ★*Adapt the type inference implementation to match your revised type system from Exercise 10.*

Exercise 12 ★*Check whether mosml, the L1 implementation and the L1 agree on the order of evaluation for operators and sequencing.*

Exercise 13 ★ *(just for fun) Adapt the implementation to output derivation trees, in ASCII, (or to show where proof search gets stuck) for $\longrightarrow$ or $\vdash$.*

3 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 – so to prove things about them, we need the corresponding *induction principles*.

Three forms of induction

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

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

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

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

Principle of Mathematical Induction

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

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

it's enough to prove

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

i.e.

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

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

For example, to prove

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

use mathematical induction for
$\Phi(x) = (1 + 2 + ... + x = 1/2 * x * (x + 1))$

There's a model proof in the notes, *(annotated to say what's going on)*, as an example of good style. Writing a clear proof structure like this becomes essential when things get more complex – you have to *use* the formalism to help you get things right. Emulate it! *(but without the annotations!)*

(NB, the natural numbers include 0)

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

Proof We prove $\forall\ x.\Phi(x)$, where *(state Φ explicitly)*

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

by mathematical induction. *(state the induction principle you're using)*

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

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

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

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 (∀), 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)*

$$(1 + 2 + ... + (k + 1)) = 1/2 * (k + 1) * ((k + 1) + 1)$$

Now, the left hand side is

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

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

and the right hand side is

$$\begin{array}{rcll} 1/2 * (k + 1) * ((k + 1) + 1) & = & 1/2 * (k * (k + 1) + (k + 1) * 1 + 1 * k + 1) & \text{(rearranging)} \\ & = & 1/2 * k * (k + 1) + 1/2 * ((k + 1) + k + 1) & \text{(rearranging)} \\ & = & 1/2 * k * (k + 1) + (k + 1) & \text{(rearranging)} \end{array}$$

which is equal to the LHS.

□

Complete Induction

For reference we recall here the principle of *complete induction*, which is equivalent to the principle of mathematical induction (anything you can prove with one, you could prove with the other) but is sometimes more convenient:

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

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

it's enough to prove

$\forall\ k\ \in\ \mathbb{N}.(\forall\ y\ \in\ \mathbb{N}.y < k \Rightarrow \Phi(y)) \Rightarrow \Phi(k)$.

3.1 Abstract Syntax and Structural Induction

Abstract Syntax and Structural Induction

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

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

First, don't forget the elided universal quantifiers.

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

Abstract Syntax

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

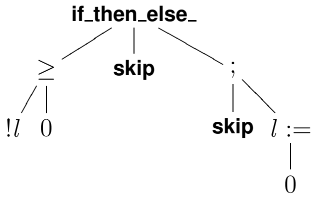
Recall we said:

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

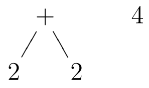
defining a set of expressions.

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

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

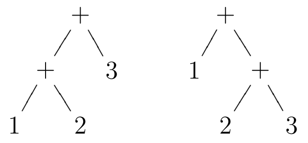


A: an abstract syntax tree. Hence: $2 + 2 \neq 4$



$1 + 2 + 3$ – ambiguous

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



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

All those are (sometimes) useful ways of looking at expressions (for lexing and parsing you start with (1) and

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

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

Abstract grammar *suggests* a concrete syntax – we write expressions as strings just for convenience, using parentheses to disambiguate where required and infix/mixfix notation, but really mean trees. Arguments about exactly what concrete syntax a language should have – beloved amongst computer scientists everywhere – do not belong in a semantics course.

Just as for natural numbers to prove $\forall\ x\ \in\ \mathbb{N}.\Phi(x)$ it was enough to prove $\Phi(0)$ and all the implications $\Phi(x) \Rightarrow \Phi(x+1)$ (for arbitrary $x\ \in\ \mathbb{N}$), here to prove $\forall\ e\ \in\ L_1.\Phi(e)$ it is enough to prove $\Phi(c)$ for each nullary tree constructor c and all the implications $(\Phi(e_1) \wedge ... \wedge \Phi(e_k)) \Rightarrow \Phi(c(e_1, .., e_k))$ for each tree constructor of arity $k \geq 1$ (and for arbitrary $e_1\ \in L_1, .., e_k\ \in\ L_1$).

Principle of Structural Induction (for abstract syntax)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

(See how this comes directly from the grammar)

If you think of the natural numbers as the abstract syntax trees of the grammar $n ::= \textbf{zero}\ \ |\ \textbf{succ}\ (n)$ then Structural Induction for that grammar is exactly the same as the Principal of Mathematical Induction.

Proving Determinacy (Outline)

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

Take

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

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

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

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

nullary:	$\Phi(\textbf{skip})$
	$\forall\ b\ \in \{\textbf{true}, \textbf{false}\}.\Phi(b)$
	$\forall\ n\ \in\ \mathbb{Z}.\Phi(n)$
	$\forall\ \ell\ \in\ \mathbb{L}.\Phi(!\ell)$
unary:	$\forall\ \ell\ \in\ \mathbb{L}.\forall\ e.\Phi(e) \Rightarrow \Phi(\ell := e)$
binary:	$\forall\ \ op\ .\forall\ e_1, e_2.(\Phi(e_1) \wedge \Phi(e_2)) \Rightarrow \Phi(e_1\ \ op\ \ e_2)$
	$\forall\ e_1, e_2.(\Phi(e_1) \wedge \Phi(e_2)) \Rightarrow \Phi(e_1; e_2)$
	$\forall\ e_1, e_2.(\Phi(e_1) \wedge \Phi(e_2)) \Rightarrow \Phi(\textbf{while}\ \ e_1\ \ \textbf{do}\ \ e_2)$
ternary:	$\forall\ e_1, e_2, e_3.(\Phi(e_1) \wedge \Phi(e_2) \wedge \Phi(e_3)) \Rightarrow \Phi(\textbf{if}\ \ e_1\ \ \textbf{then}\ \ e_2\ \ \textbf{else}\ \ e_3)$

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

3.2 Inductive Definitions and Rule Induction

Inductive Definitions and Rule Induction

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

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

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

Have to pay attention to what the elements of these relations really are...

Inductive Definitions

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

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

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

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

What did we actually mean?

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

For the transition relation:

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

For the typing relation:

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

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

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

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

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

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

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

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

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

Now, to prove something about an inductively-defined set...

Principle of Rule Induction

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

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

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

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

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

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

Principle of rule induction (a slight variant)

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

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

it's enough to prove that

for each concrete rule instance

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

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

Proving Progress (Outline)

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

Proof Take

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

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

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

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

if $\Phi(h_1) \wedge ... \wedge \Phi(h_k) \wedge h_1\ \in\ S_R \wedge .. \wedge h_k\ \in\ S_R$ then $\Phi(c)$.
Instantiating to the L1 typing rules, have to show:

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

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

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

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

etc.

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

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

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 equivalent to complete induction over $\mathbb{N}$.

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

Instead of using structural induction (for an arbitrary grammar), you could use complete 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.$e ::= \ldots \mid e + e$) to be a rule

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

3.3 Example Proofs

Example Proofs

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

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

When is a proof a proof?

What's a proof?

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

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

Bogus: neither of the above.

□

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

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

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

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

Proof Take

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

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

Cases skip, b, n. For e of these forms there are no rules with a conclusion of the form $\langle e, ...\rangle \longrightarrow \langle .., ..\rangle$ 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 $\langle !\ell, s\rangle \longrightarrow \langle e', s'\rangle \wedge \langle !\ell, s\rangle \longrightarrow \langle e'', s''\rangle$.

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

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

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

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

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

It's handy to have this lemma:

Lemma 1 *For all $e\ \in\ L_1$, if e is a value then $\forall\ s.\neg\ \exists e', s'.\langle e, s\rangle \longrightarrow \langle e', s'\rangle$.*

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

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

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

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

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

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

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

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

Case $e_1\ \ op\ \ e_2$. Suppose $\Phi(e_1)$ and $\Phi(e_2)$.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

case $e_1 =$ **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...!)

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

Proof Take

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

We show that for all Γ, 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

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

It has no premises, so we have to show that for all instances Γ, 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 Φ 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 $\text{value}(e) \vee (\exists\, e', s'.\langle e, s\rangle \longrightarrow \langle e', s'\rangle)$. But the first disjunct is true as integers are values (according to the definition).

Case (bool) similar.

Case (op+). Recall the rule

$$\text{(op +)} \quad \frac{\begin{array}{c}\Gamma \vdash e_1{:}\mathsf{int}\\ \Gamma \vdash e_2{:}\mathsf{int}\end{array}}{\Gamma \vdash e_1 + e_2{:}\mathsf{int}}$$

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

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

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

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

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

By (*) one of the following holds.

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

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

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

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

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

case e_2 is a value.

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

Lemma 2 *for all* Γ, e, T, *if* $\Gamma \vdash e{:}T$, e *is a value and* $T = \mathsf{int}$ *then for some* $n \in \mathbb{Z}$ *we have* $e = n$.

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

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+) $\langle e_1 + e_2, s\rangle \longrightarrow \langle n, s\rangle$ where $n = n_1 + n_2$, so we are done.

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

Case (if). Recall the rule

$$\text{(if)}\ \frac{\begin{array}{c}\Gamma \vdash e_1{:}\mathsf{bool}\\ \Gamma \vdash e_2{:}T\\ \Gamma \vdash e_3{:}T\end{array}}{\Gamma \vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3{:}T}$$

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

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

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

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

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

case e_1 is a value.

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

Lemma 3 *For all* Γ, e, T. *if* $\Gamma \vdash e{:}T$ *and* e *is a value, then* $T = \mathsf{int} \Rightarrow \exists\, n \in \mathbb{Z}.e = n$, $T = \mathsf{bool} \Rightarrow \exists\, b \in \{\mathbf{true}, \mathbf{false}\}.e = b$, *and* $T = \mathsf{unit} \Rightarrow e = \mathbf{skip}$.

Proof By rule induction – details omitted. □

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

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

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

Case (deref). Recall the rule

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

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

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

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

Cases (assign), (skip), (seq), (while). Left as an exercise.

□

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

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

> **Lemma 4** *If* $\langle e, s\rangle \longrightarrow \langle e', s'\rangle$ *then* $dom(s') = dom(s)$.
>
> **Proof** Rule induction on derivations of $\langle e, s\rangle \longrightarrow \langle e', s'\rangle$. Take $\Phi(e, s, e', s') = (\mathrm{dom}(s) = \mathrm{dom}(s'))$.
>
> All rules are immediate uses of the induction hypothesis except (assign1), for which we note that if $\ell \in \mathrm{dom}(s)$ then $\mathrm{dom}(s + (\ell \mapsto n)) = \mathrm{dom}(s)$. □

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

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

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

Case (op+). Recall

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

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

Case (op $\geq$). Similar.

Case (op1). Recall

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

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

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

case $op =\geq$. Similar.

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

□

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

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

Theorem 7 (Uniqueness of typing) *If* $\Gamma \vdash e{:}T$ *and* $\Gamma \vdash e{:}T'$ *then* $T = T'$. The proof is left as Exercise 18.

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

Theorem 6 (Decidability of type checking) *Given* Γ, 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. □

Proving Progress

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

Proof Take

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

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

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

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

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

Instantiating to the L1 typing rules, have to show:

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

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

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

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

etc.

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

Case (op$+$). Recall the rule

$$\text{(op }+\text{)}\quad \frac{\begin{array}{c}\Gamma \vdash e_1{:}\mathsf{int} \\ \Gamma \vdash e_2{:}\mathsf{int}\end{array}}{\Gamma \vdash e_1 + e_2{:}\mathsf{int}}$$

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

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

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

Now $e_1 + e_2$ is not a value, so we have to show
$\exists\langle e', s'\rangle.\langle e_1 + e_2, s\rangle \longrightarrow \langle e', s'\rangle$.

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

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

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

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

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

Lemma 5 *for all* Γ, e, T, *if* $\Gamma \vdash e{:}T$, e *is a value and* $T = \mathsf{int}$ *then for some* $n \in \mathbb{Z}$ *we have* $e = n$.

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

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

All the other cases are in the notes.

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

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

Proving Determinacy

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

Take

$$\begin{aligned}\Phi(e) \ \overset{\text{def}}{=}\ & \forall\, s, e', s', e'', s''.\\ & \quad (\langle e, s\rangle \longrightarrow \langle e', s'\rangle \wedge \langle e, s\rangle \longrightarrow \langle e'', s''\rangle)\\ & \quad\quad \Rightarrow \langle e', s'\rangle = \langle e'', s''\rangle\end{aligned}$$

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

Principle of Structural Induction: to prove $\Phi(e)$ for all expressions e of L1, it's enough to prove for each tree constructor c that if Φ holds for the subtrees $e_1, .., e_k$ then Φ holds for the tree $c(e_1, .., e_k)$.
Instantiating to the L1 grammar, have to show:

nullary:	$\Phi(\textbf{skip})$
	$\forall\, b \in \{\textbf{true}, \textbf{false}\}.\Phi(b)$
	$\forall\, n \in \mathbb{Z}.\Phi(n)$
	$\forall\, \ell \in \mathbb{L}.\Phi(!\ell)$
unary:	$\forall\, \ell \in \mathbb{L}.\forall\, e.\Phi(e) \Rightarrow \Phi(\ell := e)$
binary:	$\forall\ op\ .\forall\, e_1, e_2.(\Phi(e_1) \wedge \Phi(e_2)) \Rightarrow \Phi(e_1\ \ op\ \ e_2)$
	$\forall\, e_1, e_2.(\Phi(e_1) \wedge \Phi(e_2)) \Rightarrow \Phi(e_1; e_2)$
	$\forall\, e_1, e_2.(\Phi(e_1) \wedge \Phi(e_2)) \Rightarrow \Phi(\textbf{while}\ \ e_1\ \ \textbf{do}\ \ e_2)$
ternary:	$\forall\, e_1, e_2, e_3.(\Phi(e_1) \wedge \Phi(e_2) \wedge \Phi(e_3)) \Rightarrow \Phi(\textbf{if}\ \ e_1\ \ \textbf{then}\ \ e_2\ \ \textbf{else}\ \ e_3)$

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

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

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

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

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

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

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

(seq1) $\langle$**skip**$; e_2, s\rangle \longrightarrow \langle e_2, s\rangle$

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

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

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

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

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

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

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

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

Lemma: Values don't reduce

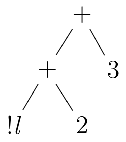
It's handy to have this lemma:

Lemma 6 *For all $e\ \in\ L_1$, if e is a value then*
$\forall\, s.\neg\,\exists e', s'.\langle e, s\rangle \longrightarrow \langle e', s'\rangle$.

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

All the other cases are in the notes.

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



Summarising Proof Techniques

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

3.4 Inductive Definitions, More Formally (optional)

Here we will be more precise about inductive definitions and rule induction. Following this may give you a sharper understanding, but it is not itself examinable. To make an *inductive definition* of a particular subset of a set A, take a set R of some concrete rule instances, each of which is a pair (H, c) where H is a finite subset of A (the hypotheses) and c is an element of A (the conclusion).

Consider finite trees labelled by elements of A for which every step is in R, eg

$$\dfrac{\dfrac{}{a_1} \qquad \dfrac{\dfrac{}{a_3}}{a_2}}{a_0}$$

where $(\{\}, a_1)$, $(\{\}, a_3)$, $(\{a_3\}, a_2)$, and $(\{a_1, a_2\}, a_0)$ all elements of R.

The subset S_R of A *inductively defined* by the rule intances R is the set of $a \in A$ such that there is such a proof with root node labelled by a.

For the definition of the transition relation:

- Start with $A = \text{expr} * \text{store} * \text{expr} * \text{store}$
- We define $\longrightarrow \subseteq A$ (write infix, e.g.$\langle e, s\rangle \longrightarrow \langle e', s'\rangle$ instead of $(e, s, e', s') \in \longrightarrow$).
- The rule instances R are the concrete rule instances of the transition rules.

For the definition of the typing relation:

- Start with $A = \text{TypeEnv} * \text{expr} * \text{types}$.
- We define $\vdash \subseteq A$ (write mixfix, e.g.$\Gamma \vdash e{:}T$ instead of $(\Gamma, e, T) \in \vdash$).
- The rule instances are the concrete rule instances of the typing rules.

Instead of talking informally about derivations as finite trees, we can regard S_R as a *least fixed point.* Given rules R, define $F_R{:}P\ A \rightarrow P\ A$ by

$$F_R(S) = \{c \mid \exists\, H.(H, c) \in R \wedge H \subseteq S\}$$

($F_R(S)$ is the set of all things you can derive in exactly one step from things in S)

$$\begin{array}{lcl} S_R^0 & = & \{\} \\ S_R^{k+1} & = & F_R(S_R^k) \\ S_R^\omega & = & \bigcap_{k \in \mathbb{N}} {S_R}^k \end{array}$$

Theorem 9 $S_R = S_R^\omega$.

Say a subset $S \subseteq A$ is *closed under rules* R if $\forall (H, c) \in R.(H \subseteq S) \Rightarrow c \in S$, ie, if $F_R(S) \subseteq S$.

Theorem 10 $S_R = \bigcap \{S \mid S \subseteq A \wedge F_R(S) \subseteq S\}$

This says 'the subset S_R of A inductively defined by R is the smallest set closed under the rules R'. It is the intersection of all of them, so smaller than (or equal to) any of them.

Now, to prove something about an inductively-defined set...

To see why rule induction is sound, using this definition: Saying $\{a \mid \Phi(a)\}$ closed under the rules means exactly $F_R(\{a \mid \Phi(a)\}) \subseteq \{a \mid \Phi(a)\}$, so by Theorem 10 we have $S_R \subseteq \{a \mid \Phi(a)\}$, i.e.$\forall\ a\ \in\ S_R.a\ \in \{a' \mid \Phi(a')\}$, i.e.$\forall\ a\ \in\ S_R.\Phi(a)$.

3.5 Exercises

Exercise 14 ★*Without looking at the proof in the notes, do the cases of the proof of Theorem 1 (Determinacy) for $e_1\ \ op\ \ e_2$, $e_1; e_2$,* **while** e_1 **do** e_2, *and* **if** e_1 **then** e_2 **else** e_3.

Exercise 15 ★*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.*

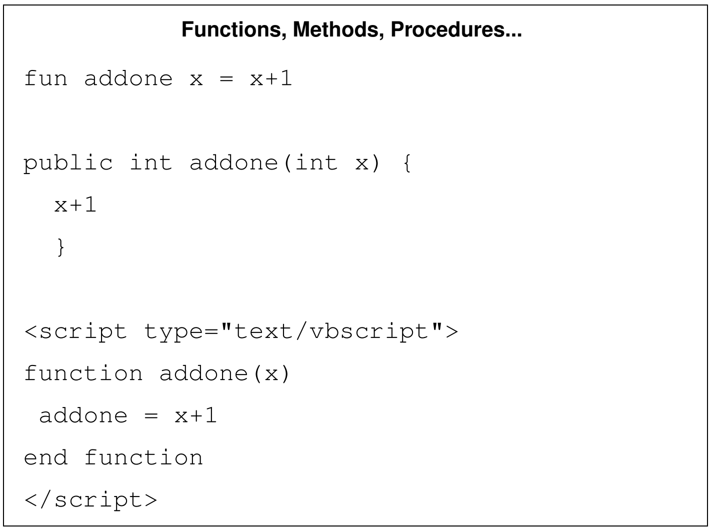
Exercise 16 ★*Complete the proof of Theorem 2 (Progress).*

Exercise 17 ★★*Complete the proof of Theorem 3 (Type Preservation).*

Exercise 18 ★★*Prove Theorem 7 (Uniqueness of Typing).*

4 Functions

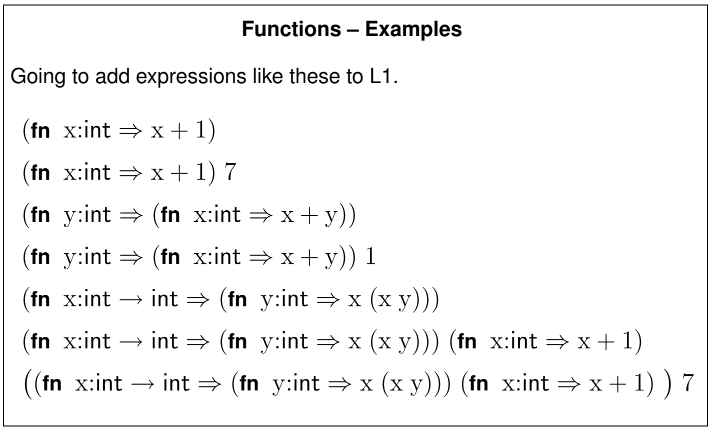
Functions – L2



Functions, Methods, Procedures...

```
fun addone x = x+1


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


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

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



Functions – Examples

Going to add expressions like these to L1.

$(\mathbf{fn}\ x{:}\mathsf{int} \Rightarrow x + 1)$

$(\mathbf{fn}\ x{:}\mathsf{int} \Rightarrow x + 1)\ 7$

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

$(\mathbf{fn}\ y{:}\mathsf{int} \Rightarrow (\mathbf{fn}\ x{:}\mathsf{int} \Rightarrow x + y))\ 1$

$(\mathbf{fn}\ x{:}\mathsf{int} \rightarrow \mathsf{int} \Rightarrow (\mathbf{fn}\ y{:}\mathsf{int} \Rightarrow x\ (x\ y)))$

$(\mathbf{fn}\ x{:}\mathsf{int} \rightarrow \mathsf{int} \Rightarrow (\mathbf{fn}\ y{:}\mathsf{int} \Rightarrow x\ (x\ y)))\ (\mathbf{fn}\ x{:}\mathsf{int} \Rightarrow x + 1)$

$\big((\mathbf{fn}\ x{:}\mathsf{int} \rightarrow \mathsf{int} \Rightarrow (\mathbf{fn}\ y{:}\mathsf{int} \Rightarrow x\ (x\ y)))\ (\mathbf{fn}\ x{:}\mathsf{int} \Rightarrow x + 1)\ \big)\ 7$

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

Functions – Syntax

First, extend the L1 syntax:

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

Expressions

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

Types

$$\begin{array}{lll} T & ::= & \textsf{int} \mid \textsf{bool} \mid \textsf{unit} \mid T_1 \to T_2 \\ T_{loc} & ::= & \textsf{intref} \end{array}$$

- Concrete syntax: by convention, application associates to the left, so $e_1\ e_2\ e_3$ denotes $(e_1\ e_2)\ e_3$, and type arrows associate to the right, so $T_1 \to T_2 \to T_3$ denotes $T_1 \to (T_2 \to T_3)$. A **fn** extends to the right as far as parentheses permit, so **fn** x:unit $\Rightarrow$ x; x denotes **fn** x:unit $\Rightarrow$ (x; x), not (**fn** x:unit $\Rightarrow$ x); x. These conventions work well for functions that take several arguments, e.g.**fn** x:unit $\Rightarrow$ **fn** y:int $\Rightarrow$ x; y has type unit $\to$ int $\to$ int, and we can fully apply it simply by juxtaposing it with its two arguments (**fn** x:unit $\Rightarrow$ **fn** y:int $\Rightarrow$ x; y) **skip** 15.
- Variables are not locations ($\mathbb{L} \cap \mathbb{X} = \{\}$), so x := 3 is not in the syntax.
- The (non-meta) variables x, y, z are not the same as metavariables x, y, z. In the notes they are distinguished by font; in handwriting one just have to keep track in your head – not often a problem.
- These expressions look like lambda terms (**fn** x:int $\Rightarrow$ x could be written λx:int.x). But, (a) we're adding them to a rich language, not working with the pure lambda calculus (cf. *Foundations of Functional Programming*), and (b) we're going to explore several options for how they should behave.
- Returning to the idea of type-directed language design: this type grammar (and expression syntax) suggests the language will include higher-order functions – you can abstract on a variable of any type, including function types. If you only wanted first-order functions, you'd say

  $$\begin{array}{lll} A & ::= & \textsf{int} \mid \textsf{bool} \mid \textsf{unit} \\ T & ::= & A \mid A \to T \\ T_{loc} & ::= & \textsf{intref} \end{array}$$

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

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

  Some languages go further, forbidding partial application – and thereby avoiding the need for heap-allocated closures in the implementation. We'll come back to this.

4.1 Function Preliminaries: Abstract Syntax up to Alpha Conversion, and Substitution

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

Variable shadowing

(**fn** x:int ⇒ (**fn** x:int ⇒ $\mathrm{x}+1$))

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

The second is not allowed in Java. For large systems that would be a problem, eg in a language with nested function definitions, where you may wish to write a local function parameter without being aware of what is in the surrounding namespace. There are other issues to do with class namespaces.

Alpha conversion

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

- inside e, any x's (that aren't themselves binders and are not inside another **fn** $x{:}T' \Rightarrow$ ...) mean the same thing – the formal parameter of this function.
- outside this **fn** $x{:}T \Rightarrow e$, it doesn't matter which variable we used for the formal parameter – in fact, we shouldn't be able to tell. For example, **fn** x:int ⇒ $\mathrm{x}+2$ should be the same as **fn** y:int ⇒ $\mathrm{y}+2$.
  cf $\int_0^1 x + x^2 dx = \int_0^1 y + y^2 dy$

Alpha conversion – free and bound occurrences

In a bit more detail (but still informally):

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

17

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

fn x:int ⇒ $\mathrm{x}+2$

fn x:int ⇒ $\mathrm{x}+\mathrm{z}$

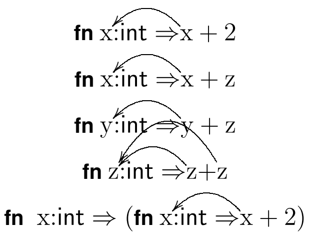
if y **then** $2+\mathrm{x}$ **else** ((**fn** x:int ⇒ $\mathrm{x}+2$)z)

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

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

Sometimes it is handy to draw in the binding:

Alpha conversion – Binding examples



fn x:int ⇒x + 2

fn x:int ⇒x + z

fn y:int ⇒y + z

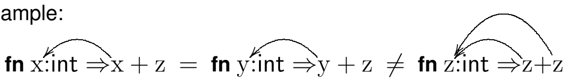
fn z:int ⇒z+z

fn x:int ⇒ (**fn** x:int ⇒x + 2)

Alpha Conversion – The Convention

Convention: we will allow ourselves to *any time at all, in any expression* ...(**fn** x:T ⇒ e)..., 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:



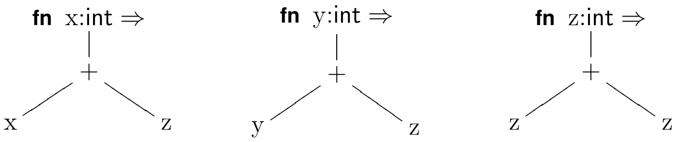
fn x:int ⇒x + z = **fn** y:int ⇒y + z ≠ **fn** z:int ⇒z+z

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

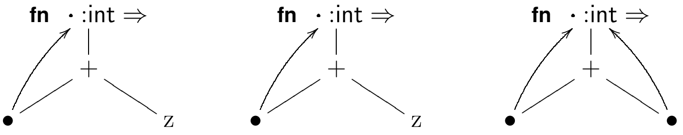
Abstract Syntax *up to Alpha Conversion*

fn x:int ⇒ x + z = **fn** y:int ⇒ y + z ≠ **fn** z:int ⇒ z + z

Start with naive abstract syntax trees:

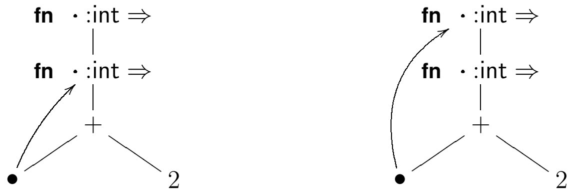


add pointers (from each x node to the closest enclosing **fn** x:T ⇒ node); remove names of binders and the occurrences they bind

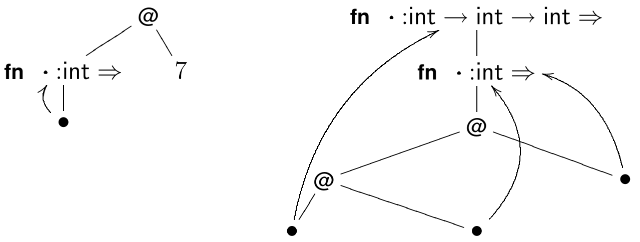


fn x:int ⇒ (**fn** x:int ⇒ x + 2)

= **fn** y:int ⇒ (**fn** z:int ⇒ z + 2) ≠ **fn** z:int ⇒ (**fn** y:int ⇒ z + 2)



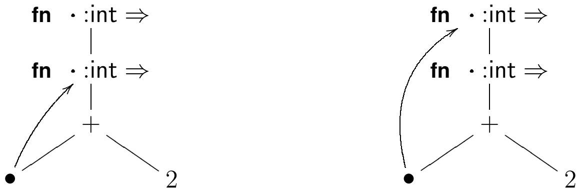
$(\textbf{fn}\ \mathrm{x}{:}\mathsf{int} \Rightarrow \mathrm{x})\ 7 \qquad \textbf{fn}\ \mathrm{z}{:}\mathsf{int} \rightarrow \mathsf{int} \rightarrow \mathsf{int} \Rightarrow (\textbf{fn}\ \mathrm{y}{:}\mathsf{int} \Rightarrow \mathrm{z}\ \mathrm{y}\ \mathrm{y})$



De Bruijn Indices

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

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



Free Variables

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

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

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

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

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

For example

$$\begin{array}{lcl}
\mathrm{fv}(\mathrm{x}+\mathrm{y}) & = & \{\mathrm{x},\mathrm{y}\} \\
\mathrm{fv}(\textbf{fn}\ \ \mathrm{x}{:}\mathsf{int} \Rightarrow \mathrm{x}+\mathrm{y}) & = & \{\mathrm{y}\} \\
\mathrm{fv}(\mathrm{x}+(\textbf{fn}\ \ \mathrm{x}{:}\mathsf{int} \Rightarrow \mathrm{x}+\mathrm{y})7) & = & \{\mathrm{x},\mathrm{y}\}
\end{array}$$

Full definition of $\mathrm{fv}(e)$:

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

(for an example of a definition that is *not* alpha-invariant, consider

$$\begin{array}{lcl}
\mathrm{bv}(x) & = & \{\} \\
\mathrm{bv}(\textbf{fn}\ \ x{:}T \Rightarrow e) & = & \{x\} \cup \mathrm{bv}(e) \\
\mathrm{bv}(e_1\ e_2) & = & \mathrm{bv}(e_1) \cup \mathrm{bv}(e_2) \\
\ldots & &
\end{array}$$

This is fine for concrete terms, but we're working up to alpha conversion, so $(\textbf{fn}\ \ \mathrm{x{:}int} \Rightarrow 2) = (\textbf{fn}\ \ \mathrm{y{:}int} \Rightarrow 2)$ but $\mathrm{bv}(\textbf{fn}\ \ \mathrm{x{:}int} \Rightarrow 2) = \{\mathrm{x}\} \neq \{\mathrm{y}\} = \mathrm{bv}(\textbf{fn}\ \ \mathrm{y{:}int} \Rightarrow 2)$. Argh! Can see from looking back at the abstract syntax trees up to alpha conversion that they just don't have this information in, anyway.)

The semantics for functions will involve substituting actual parameters for formal parameters. That's a bit delicate in a world with binding...

Substitution – Examples

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

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

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

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:

$$\begin{array}{lcll}
\{e/z\}x & = & e & \text{if } x = z \\
 & = & x & \text{otherwise} \\
\{e/z\}(\textbf{fn}\ \ x{:}T \Rightarrow e_1) & = & \textbf{fn}\ \ x{:}T \Rightarrow (\{e/z\}e_1) & \text{if } x \neq z\ (*) \\
 & & & \text{and } x \notin \mathrm{fv}(e)\ (*) \\
\{e/z\}(e_1\ e_2) & = & (\{e/z\}e_1)(\{e/z\}e_2) & \\
\ldots & & &
\end{array}$$

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

Substitution – Example Again

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

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

Substitution – Simultaneous

Generalising to simultaneous substitution: Say a *substitution* σ is a finite partial function from variables to expressions.

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

Define $\sigma\ e$ in the notes.

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

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

Define the application of a substitution to a term by:

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

4.2 Function Typing

Typing functions (1)

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

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

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

$\forall\ \ell\ \in \text{dom}(\Gamma).\Gamma(\ell) \in\ \mathrm{T_{loc}}$

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

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

Typing functions (2)

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

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

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

Typing functions – Example

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

- The syntax is explicitly typed, so don't need to 'guess' a T in the **fn** rule.
- Recall that variables of these types are quite different from locations – you can't assign to variables; you can't abstract on locations. For example, (**fn** l:intref $\Rightarrow !l$) is not in the syntax.
- Note that sometimes you need the alpha convention, e.g. to type

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

  It's a good idea to start out with all binders different from each other and from all free variables. It would be a bad idea to prohibit variable shadowing like this in source programs.
- In ML you have *parametrically polymorphic* functions, but we won't talk about them here – that's in Part II *Types*.
- Note that these functions are not recursive (as you can see in the syntax: there's no way in the body of **fn** $x{:}T \Rightarrow e$ to refer to the function as a whole).
- With our notational convention for $\Gamma, x{:}T$, we could rewrite the (var) rule as $\Gamma, x{:}T \vdash x{:}T$. By the convention, x is not in the domain of Γ, and $\Gamma + \{x \mapsto T\}$ is a perfectly good partial function.

Another example:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{l{:}\mathsf{intref}, \mathrm{x}{:}\mathsf{unit} \vdash 1{:}\mathsf{int}}\,(\mathrm{int})}{l{:}\mathsf{intref}, \mathrm{x}{:}\mathsf{unit} \vdash (l := 1){:}\mathsf{unit}}\,(\mathrm{assign}) \quad \dfrac{}{l{:}\mathsf{intref}, \mathrm{x}{:}\mathsf{unit} \vdash \mathrm{x}{:}\mathsf{unit}}\,(\mathrm{var})}{l{:}\mathsf{intref}, \mathrm{x}{:}\mathsf{unit} \vdash (l := 1); \mathrm{x}{:}\mathsf{unit}}\,(\mathrm{seq})}{l{:}\mathsf{intref} \vdash (\mathbf{fn}\ \mathrm{x}{:}\mathsf{unit} \Rightarrow (l := 1); \mathrm{x}){:}\mathsf{unit} \to \mathsf{unit}}\,(\mathrm{fn}) \quad \dfrac{\dfrac{}{l{:}\mathsf{intref} \vdash 2{:}\mathsf{int}}\,(\mathrm{int})}{l{:}\mathsf{intref} \vdash (l := 2){:}\mathsf{unit}}\,(\mathrm{assign})}{l{:}\mathsf{intref} \vdash (\mathbf{fn}\ \mathrm{x}{:}\mathsf{unit} \Rightarrow (l := 1); \mathrm{x})\ (l := 2){:}\mathsf{unit}}\,(\mathrm{app})$$

4.3 Function Behaviour

Function Behaviour

Consider the expression

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

then

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

Function Behaviour. Choice 1: Call-by-value

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

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

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

This is most common design choice - ML, Java,...

L2 Call-by-value

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

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

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

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

- This is a *strict* semantics – fully evaluating the argument to function before doing the application.
- One could evaluate $e_1\ e_2$ right-to-left instead or left-to-right. That would be perverse – better design is to match the evaluation order for operators etc.

L2 Call-by-value – reduction examples

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

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

- The syntax has explicit types and the semantics involves syntax, so types appear in semantics – but they are not used in any interesting way, so an implementation could erase them before execution. Not all languages have this property.
- The rules for these constructs, and those in the next few lectures, don't touch the store, but we need to include it in the rules in order to get the sequencing of side-effects right. In a *pure* functional language, configurations would just be expressions.
- A naive implementation of these rules would have to traverse e and copy v as many times as there are free occurrences of x in e. Real implementations don't do that, using *environments* instead of doing

substitution. Environments are more efficient; substitutions are simpler to write down – so better for implementation and semantics respectively.

Function Behaviour. Choice 2: Call-by-name

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

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

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

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

L2 Call-by-name

(same typing rules as before)

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

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

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

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

but if it is, end up evaluating it repeatedly.

Haskell uses a refined variant – call-by-need – in which the first time the argument evaluated we ‘overwrite’ all other copies by that value.

That lets you do some very nice programming, e.g. with potentially-infinite datastructures.

Call-By-Need Example (Haskell)

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

But, it becomes rather hard to understand what order your code is going to be run in! Hence, non-strict languages typically don’t allow unrestricted side effects (our combination of store and CBN is *pretty odd*).

Instead, 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.

Purity

Note that Call-by-Value and Call-by-Name are distinguishable even if there is no store – consider applying a function to a non-terminating argument, eg (**fn** x:unit ⇒ **skip**) (**while true do skip**).

Call-by-Name and Call-by-Need are not distinguishable except by performance properties – but those really matter.

Function Behaviour. Choice 3: Full beta

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

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

L2 Beta

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

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

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

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

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

L2 Beta: Example

$$\begin{tikzcd}
& (\textbf{fn}\ \text{x:int} \Rightarrow \text{x}+\text{x})\ (2+2) \arrow[dl] \arrow[dr] & & \\
(\textbf{fn}\ \text{x:int} \Rightarrow \text{x}+\text{x})\ 4 \arrow[ddr] & & (2+2)+(2+2) \arrow[dl] \arrow[dr] & \\
& 4+(2+2) \arrow[d] & & (2+2)+4 \arrow[dll] \\
& 4+4 \arrow[d] & & \\
& 8 & &
\end{tikzcd}$$

This ain't much good for a programming language... why? (if you've got any non-terminating computation Ω, then $(\lambda x.y)\ \Omega$ might terminate or not, depending on the implementation) (in pure lambda you do have *confluence*, which saves you – at least mathematically)

Function Behaviour. Choice 4: Normal-order reduction

Leftmost, outermost variant of full beta.

Back to CBV (from now on).

Properties of Typing

As before, but only interested in executing *closed* programs.

Theorem 11 (Progress) *If e closed and $\Gamma \vdash e{:}T$ and dom$(\Gamma) \subseteq$ dom(s) then either e is a value or there exist e', s' such that $\langle e, s\rangle \longrightarrow \langle e', s'\rangle$.*

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

Theorem 12 (Type Preservation) *If e closed and $\Gamma \vdash e{:}T$ and dom$(\Gamma) \subseteq$ dom(s) and $\langle e, s\rangle \longrightarrow \langle e', s'\rangle$ then $\Gamma \vdash e'{:}T$ and e' closed and dom$(\Gamma) \subseteq$ dom(s').*

Proving Type Preservation

Theorem 12 (Type Preservation) *If e closed and $\Gamma \vdash e{:}T$ and dom$(\Gamma) \subseteq$ dom(s) and $\langle e, s\rangle \longrightarrow \langle e', s'\rangle$ then $\Gamma \vdash e'{:}T$ and e' closed and dom$(\Gamma) \subseteq$ dom(s').*

Taking

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

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

To prove this one uses:

Lemma 7 (Substitution) *If $\Gamma \vdash e{:}T$ and $\Gamma, x{:}T \vdash e'{:}T'$ with $x \notin$ dom(Γ) then $\Gamma \vdash \{e/x\}e'{:}T'$.*

Determinacy and type inference properties also hold.

Normalisation

Theorem 13 (Normalisation) *In the sublanguage without while loops or store operations, if $\Gamma \vdash e{:}T$ and e closed then there does not exist an infinite reduction sequence $\langle e, \{\}\rangle \longrightarrow \langle e_1, \{\}\rangle \longrightarrow \langle e_2, \{\}\rangle \longrightarrow ...$*

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:

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

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

Can regard just as *syntactic sugar*:

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

Local definitions – derived typing and reduction rules (CBV)

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

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

(let1)

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

(let2)

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

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

$$\mathrm{x} + \textbf{let val } \mathrm{x}{:}\mathsf{int} = \mathrm{x} \textbf{ in } (\mathrm{x}+2) \textbf{ end} \quad = \quad \mathrm{x} + \textbf{let val } \mathrm{y}{:}\mathsf{int} = \mathrm{x} \textbf{ in } (\mathrm{y}+2) \textbf{ end}$$

Recursive definitions – first attempt

How about

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

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

Could add something like this:

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

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

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

But...

What about

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

Have some rather weird things, eg

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

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

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

Recursive Functions

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

$$\begin{array}{rcll} T & = & T_1 \rightarrow T_2 & \text{recursion only at function types} \\ e & = & \textbf{fn } y{:}T_1 \Rightarrow e_1 & \text{and only of function values} \end{array}$$

e ::= ... | **let val rec** $x{:}T_1 \rightarrow T_2 = ($**fn** $y{:}T_1 \Rightarrow e_1)$ **in** e_2 **end**

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

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

Concrete syntax: In ML can write **let fun** $f(x{:}T_1){:}T_2 = e_1$ **in** e_2 **end**, or even **let fun** $f(x) = e_1$ **in** e_2 **end**, for
let val rec $f{:}T_1 \rightarrow T_2 =$ **fn** $x{:}T_1 \Rightarrow e_1$ **in** e_2 **end**.

Recursive Functions – Semantics

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

$\longrightarrow$

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

(sometimes use **fix**:$((T_1 \rightarrow T_2) \rightarrow (T_1 \rightarrow T_2)) \rightarrow (T_1 \rightarrow T_2)$ – cf. the Y combinator, in *Foundations of Functional Programming*)

For example:

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

Recursive Functions – Minimisation Example

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

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

As a test case, we apply it to the function (**fn** z:int ⇒ **if** $z \geq 3$ **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 $e_1; e_2 \rightsquigarrow (\textbf{fn}\ y{:}\text{unit} \Rightarrow e_2)e_1$

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

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

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

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

In each case typing is the same (more precisely?); reduction is 'essentially' the same. What does that mean? More later, on contextual equivalence.

OTOH, Could we encode recursion in the language without?

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

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

here.

4.5 Implementation

Implementation

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

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

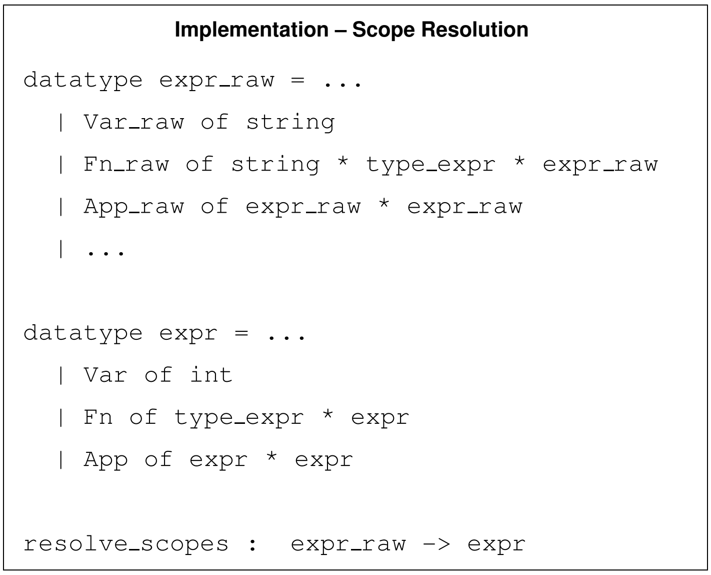
Also, as before, L2 expressions can be executed directly in a Moscow ML context.

The README file says:

```
(* 2002-11-08 -- Time-stamp: <2003-04-25 17:28:25 pes20>    *)
(* Peter Sewell                                              *)

This directory contains an interpreter, pretty-printer and
type-checker for the language L2.

To make it go, copy it into a working directory, ensure Moscow ML
is available (including mosmllex and mosmlyac), and type

  make
  mosml
  load "Main";

It prompts you for an L2 expression (terminated by RETURN, no terminating
semicolons) and then for an initial store.  For the latter, if you
just press RETURN you get a default store in which all the locations
mentioned in your expression are mapped to 0.
```

```
Watch out for the parsing - it is not quite the same as (eg) mosml, so
you need to parenthesise more.

The source files are:

  Main.sml         the top-level loop
  Syntax.sml       datatypes for raw and de-bruijn expressions
  Lexer.lex        the lexer (input to mosmllex)
  Parser.grm       the grammar (input to mosmlyac)
  Semantics.sml    scope resolution, the interpreter, and the typechecker
  PrettyPrint.sml  pretty-printing code

  Examples.l2      some handy examples for cut-and-pasting into the
                     top-level loop

of these, you're most likely to want to look at, and change, Semantics.sml.
You should first also look at Syntax.sml.
```

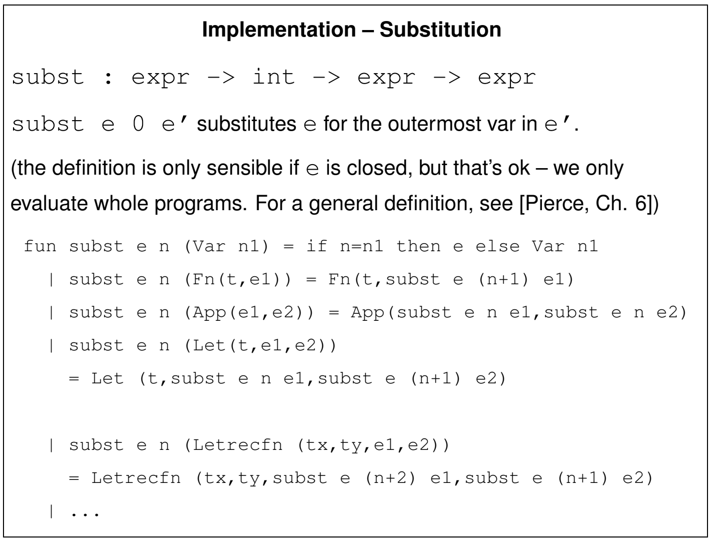
The implementation lets you type in L2 expressions and initial stores and watch them resolve, type-check, and reduce.



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

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

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

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

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

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

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

Implementation – CBV reduction

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

Implementation – Type Inference

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

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

Implementation – Closures

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

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

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

Aside: Small-step vs Big-step Semantics

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

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

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

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

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

4.6 L2: Collected Definition

Syntax

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

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

Types

$$\begin{array}{lll} T & ::= & \textsf{int} \mid \textsf{bool} \mid \textsf{unit} \mid T_1 \to T_2 \\ T_{loc} & ::= & \textsf{intref} \end{array}$$

Expressions

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

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

Operational Semantics

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Typing

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

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

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

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

$$\text{(op +)}\ \frac{\begin{array}{c}\Gamma \vdash e_1{:}\mathsf{int}\\ \Gamma \vdash e_2{:}\mathsf{int}\end{array}}{\Gamma \vdash e_1 + e_2{:}\mathsf{int}} \qquad \text{(op} \geq\text{)}\ \frac{\begin{array}{c}\Gamma \vdash e_1{:}\mathsf{int}\\ \Gamma \vdash e_2{:}\mathsf{int}\end{array}}{\Gamma \vdash e_1 \geq e_2{:}\mathsf{bool}}$$

$$\text{(if)}\ \frac{\begin{array}{c}\Gamma \vdash e_1{:}\mathsf{bool}\\ \Gamma \vdash e_2{:}T\\ \Gamma \vdash e_3{:}T\end{array}}{\Gamma \vdash \textbf{if}\ \ e_1\ \ \textbf{then}\ \ e_2\ \ \textbf{else}\ \ e_3{:}T}$$

$$\text{(assign)}\ \frac{\begin{array}{c}\Gamma(\ell) = \mathsf{intref}\\ \Gamma \vdash e{:}\mathsf{int}\end{array}}{\Gamma \vdash \ell := e{:}\mathsf{unit}}$$

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

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

$$\text{(seq)}\ \frac{\begin{array}{c}\Gamma \vdash e_1{:}\mathsf{unit}\\ \Gamma \vdash e_2{:}T\end{array}}{\Gamma \vdash e_1; e_2{:}T}$$

$$\text{(while)}\ \frac{\begin{array}{c}\Gamma \vdash e_1{:}\mathsf{bool}\\ \Gamma \vdash e_2{:}\mathsf{unit}\end{array}}{\Gamma \vdash \textbf{while}\ \ e_1\ \ \textbf{do}\ \ e_2{:}\mathsf{unit}}$$

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

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

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

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

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

4.7 Exercises

Exercise 19 ★*What are the free variables of the following?*

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

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

Exercise 20 ★*What are the following?*

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

Exercise 21 ★*Give typing derivations, or show why no derivation exists, for:*

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

Exercise 22 ★★*Give a grammar for types, and typing rules for functions and application, that allow only first-order functions and prohibit partial applications.*

Exercise 23 ★★*Write a function of type* $\mathsf{unit} \to \mathsf{bool}$ *that, when applied to* $()$*, returns* **true** *in the CBV semantics and* **false** *in the CBN semantics. Can you do it without using the store?*

Exercise 24 ★★*Prove Lemma 7 (Substitution).*

Exercise 25 ★★*Prove Theorem 12 (Type Preservation).*

Exercise 26 ★★*Adapt the L2 implementation to CBN functions. Think of a few good test cases and check them in the new and old code.*

Exercise 27 ★★★*Re-implement the L2 interpreter to use closures instead of substitution.*

5 Data

Data – L3

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

5.1 Products, Sums, and Records

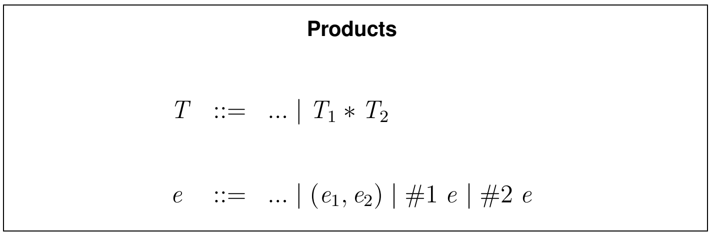
The two basic notions are the *product* and the *sum* type.

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

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



Products

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

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

Design choices:

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

Products - typing

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

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

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

Products - reduction

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

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

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

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

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

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

Sums (or Variants, or Tagged Unions)

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

Those xs are binders.

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

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

Sums - typing

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

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

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

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

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

and

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

You might:

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

Sums - reduction

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

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

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

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

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

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

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

Constructors and Destructors

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

The Curry-Howard Isomorphism

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

ML Datatypes

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

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

is (roughly!) like saying

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

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

Records

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

Take field labels

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

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

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

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

Note:

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

Records - typing

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

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

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

Records - reduction

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

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

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

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

5.2 Mutable Store

Mutable Store

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

1 What we've got in L1 and L2:

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

- locations store mutable values
- variables refer to a previously-calculated value, immutably
- explicit dereferencing and assignment operators for locations
  $\textbf{fn}\ \mathrm{x}{:}\mathsf{int} \Rightarrow l := (!l) + \mathrm{x}$

2 The C-way (also Java etc).

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

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

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

– pros and cons: ....

References

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

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

So, generalise.

T ::= ... | T ref

T_{loc} ::= ~~intref~~ T ref

e ::= ... | ~~$\ell := e$~~ | ~~$!\ell$~~

| $e_1 := e_2$ | $!e$ | ref e | ℓ

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

References - Typing

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

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

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

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

References – Reduction

A location is a value:

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

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.

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

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

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

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

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

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

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

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

  1. an array of bytes,
  2. an array of values, or
  3. a partial function from locations to values?

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

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

Type-checking the store

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

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

Type-checking the store – Example

Consider

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

which has reductions

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

For reference, e_1 and e_2 are

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

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

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

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

The statement of type preservation will then be:

Theorem 14 (Type Preservation) *If e closed and $\Gamma \vdash e{:}T$ and $\Gamma \vdash s$ and $\langle e, s\rangle \longrightarrow \langle e', s'\rangle$ then for some Γ' with disjoint domain to Γ we have $\Gamma, \Gamma' \vdash e'{:}T$ and $\Gamma, \Gamma' \vdash s'$.*

Implementation

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

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

5.3 Evaluation Contexts

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

Evaluation Contexts

Define *evaluation contexts*

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

and have the single *context* rule

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

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

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

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

A Little (Oversimplified!) History

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

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

5.4 L3: Collected Definition

L3 Syntax

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

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

Types:

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

Expressions

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

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

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

L3 Semantics

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$$\text{(record1)}\quad \frac{\langle e_i, s\rangle \longrightarrow \langle e_i', s'\rangle}{\begin{array}{l}\langle \{\mathit{lab}_1 = v_1, .., \mathit{lab}_i = e_i, .., \mathit{lab}_k = e_k\}, s\rangle \\ \longrightarrow \langle \{\mathit{lab}_1 = v_1, .., \mathit{lab}_i = e_i', .., \mathit{lab}_k = e_k\}, s'\rangle\end{array}}$$

$$\text{(record2)}\quad \langle \#\mathit{lab}_i\ \{\mathit{lab}_1 = v_1, .., \mathit{lab}_k = v_k\}, s\rangle \longrightarrow \langle v_i, s\rangle$$

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

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

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

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

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

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

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

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

L3 Typing

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

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

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

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

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

$$\text{(op }+\text{)}\quad \frac{\begin{array}{c}\Gamma \vdash e_1{:}\mathsf{int} \\ \Gamma \vdash e_2{:}\mathsf{int}\end{array}}{\Gamma \vdash e_1 + e_2{:}\mathsf{int}} \qquad \text{(op }\geq\text{)}\quad \frac{\begin{array}{c}\Gamma \vdash e_1{:}\mathsf{int} \\ \Gamma \vdash e_2{:}\mathsf{int}\end{array}}{\Gamma \vdash e_1 \geq e_2{:}\mathsf{bool}}$$

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

$$\text{(assign)}\quad \frac{\begin{array}{c}\Gamma(\ell) = \mathsf{intref}\\ \Gamma \vdash e{:}\mathsf{int}\end{array}}{\Gamma \vdash \ell := e{:}\mathsf{unit}}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

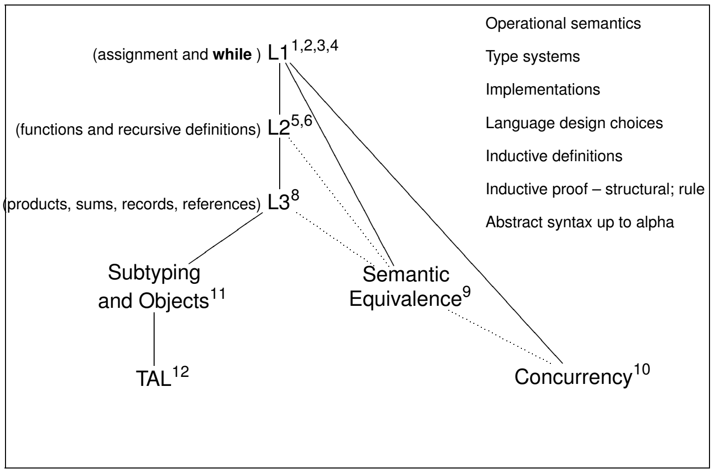
$$\text{(loc)}\quad \frac{\Gamma(\ell) = T\ \ \mathsf{ref}}{\Gamma \vdash \ell{:}T\ \ \mathsf{ref}}$$

5.5 Exercises

Exercise 28 ★★*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.*


(assignment and **while**) L1[1,2,3,4]
(functions and recursive definitions) L2[5,6]
(products, sums, records, references) L3[8]
Subtyping and Objects[11]
TAL[12]
Semantic Equivalence[9]
Concurrency[10]
Operational semantics
Type systems
Implementations
Language design choices
Inductive definitions
Inductive proof – structural; rule
Abstract syntax up to alpha

6 Semantic Equivalence

Semantic Equivalence

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

In what sense are these two expressions the same?

They have different abstract syntax trees.

They have different reduction sequences.

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

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

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

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

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

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

On the other hand, consider

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

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

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

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

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

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

$$\begin{array}{l} \textbf{let val } \mathrm{x} = \mathrm{ref}\ 0 \textbf{ in fn } \mathrm{y{:}int} \Rightarrow (\mathrm{x} :=!\mathrm{x} + \mathrm{y}); !\mathrm{x} \\ \quad\overset{?}{\simeq} \\ \textbf{let val } \mathrm{x} = \mathrm{ref}\ 0 \textbf{ in fn } \mathrm{y{:}int} \Rightarrow (\mathrm{x} :=!\mathrm{x} - \mathrm{y}); (0-!\mathrm{x}) \end{array}$$

Temporarily extend L3 with pointer equality

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

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

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

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

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

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

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

$f \overset{?}{\simeq} g$ ... no:

Consider $C = t\ \_$, where

$$\begin{aligned}
t = \ & \textbf{fn } \mathrm{h}{:}(\mathsf{int\ ref} \rightarrow \mathsf{int\ ref}) \Rightarrow\\
& \quad \textbf{let val } \mathrm{z} = \ \mathsf{ref}\ 0\ \textbf{in } \mathrm{h}\ (\mathrm{h}\ \mathrm{z}) = \mathrm{h}\ \mathrm{z}
\end{aligned}$$

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

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

The last two examples are taken from A.M. Pitts, Operational Semantics and Program Equivalence. In: G. Barthe, P. Dybjer and J. Saraiva (Eds), Applied Semantics. Lecture Notes in Computer Science, Tutorial, Volume 2395 (Springer-Verlag, 2002), pages 378-412. (Revised version of lectures at the International Summer School On Applied Semantics, APPSEM 2000, Caminha, Minho, Portugal, 9-15 September 2000.) `ftp://ftp.cl.cam.ac.uk/papers/amp12/opespe-lncs.pdf`

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

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

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

1. programs that result in observably-different values (in some initial store) must not be equivalent
   $(\exists\ s, s_1, s_2, v_1, v_2 . \langle e_1, s\rangle \longrightarrow \langle v_1, s_1\rangle \wedge \langle e_2, s\rangle \longrightarrow \langle v_2, s_2\rangle$
   $\wedge\ v_1 \neq v_2) \Rightarrow e_1 \not\simeq e_2$
2. programs that terminate must not be equivalent to programs that don’t
3. $\simeq$ must be an equivalence relation
   $e \simeq e, \quad e_1 \simeq e_2 \Rightarrow e_2 \simeq e_1, \quad e_1 \simeq e_2 \simeq e_3 \implies e_1 \simeq e_3$
4. $\simeq$ must be a congruence
   if $e_1 \simeq e_2$ then for any context C we must have $C[e_1] \simeq C[e_2]$
5. $\simeq$ should relate as many programs as possible subject to the above.

Semantic Equivalence for L1

Consider Typed L1 again.

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

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

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

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

If $T =$ bool then $C =$ **if** _ **then** $!l$ **else** $!l$.

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

Congruence for Typed L1

The L1 contexts are:

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

Say $\simeq^T_\Gamma$ 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 $C[e_1] \simeq^{T'}_\Gamma C[e_2]$.

Theorem 16 (Congruence for L1) $\simeq_\Gamma^T$ *has the congruence property.*

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

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

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

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

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

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

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

Theorem 16 (Congruence for L1) $\simeq_\Gamma^T$ *has the congruence property.*

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

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

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

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

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

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

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

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

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

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

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

$\langle \ell := e, s\rangle \longrightarrow \langle e_1, s_1\rangle \longrightarrow \langle e_2, s_2\rangle ... \longrightarrow \langle e_k, s_k\rangle \not\longrightarrow$

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

$\langle e, s\rangle \longrightarrow \langle \hat{e}_1, s_1\rangle \longrightarrow \langle \hat{e}_2, s_2\rangle \longrightarrow ... \longrightarrow \langle \hat{e}_{k-1}, s_{k-1}\rangle$

and $e_1 = (\ell := \hat{e}_1)$, $e_2 = (\ell := \hat{e}_2)$,..., $e_{k-1} = (\ell := \hat{e}_{k-1})$ and for some n we have $\hat{e}_{k-1} = n$, $e_k =$ **skip**, and $s_k = s_{k-1} + \{\ell \mapsto n\}$.

(the other possibility, of zero or more (assign1) reductions ending in a stuck state, is excluded by Theorems 2 and 3 (type preservation and progress))

Now, if $\langle \ell := e, s \rangle \longrightarrow^{\omega}$ we have $\langle e, s \rangle \longrightarrow^{\omega}$, so by $e \ \simeq^{T}_{\Gamma} \ e'$ we have $\langle e', s \rangle \longrightarrow^{\omega}$, so (using (assign2)) we have $\langle \ell := e', s \rangle \longrightarrow^{\omega}$.

On the other hand, if $\neg(\langle \ell := e, s \rangle \longrightarrow^{\omega})$ then by the above there is some n and s_{k-1} such that $\langle e, s \rangle \longrightarrow^{*} \langle n, s_{k-1} \rangle$ and $\langle \ell := e, s \rangle \longrightarrow \langle \textbf{skip}, s_{k-1} + \{\ell \mapsto n\} \rangle$.

By $e \ \simeq^{T}_{\Gamma} \ e'$ we have $\langle e', s \rangle \longrightarrow^{*} \langle n, s_{k-1} \rangle$.

Then using (assign1)

$\langle \ell := e', s \rangle \longrightarrow^{*} \langle \ell := n, s_{k-1} \rangle \longrightarrow \langle \textbf{skip}, s_{k-1} + \{\ell \mapsto n\} \rangle =$ $\langle e_k, s_k \rangle$ as required.

Theorem 16 (Congruence for L1) $\simeq^{T}_{\Gamma}$ *has the congruence property.*

Proof By case analysis, looking at each L1 context in turn. We give only one case here, leaving the others for the reader.

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

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

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

Consider the possible reduction sequences of a state $\langle \ell := e, s \rangle$. Recall that (by examining the reduction rules), if $\langle \ell := e, s \rangle \longrightarrow \langle e_1, s_1 \rangle$ then either that is an instance of (assign1), with $\exists\ n.e = n \wedge \ell \ \in \mathrm{dom}(s) \wedge e_1 = \textbf{skip} \wedge s' = s + \{\ell \mapsto n\}$, or it is an instance of (assign2), with $\exists\ \hat{e}_1.\langle e, s \rangle \longrightarrow \langle \hat{e}_1, s_1 \rangle \wedge e_1 = (\ell := \hat{e}_1)$. We know also that $\langle \textbf{skip}, s \rangle$ does not reduce.

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

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

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

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

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

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

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

$\langle \ell := e, s \rangle \longrightarrow \langle e_1, s_1 \rangle \longrightarrow \langle e_2, s_2 \rangle ... \longrightarrow \langle e_k, s_k \rangle \not\longrightarrow$

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

$\langle e, s \rangle \longrightarrow \langle \hat{e}_1, s_1 \rangle \longrightarrow \langle \hat{e}_2, s_2 \rangle \longrightarrow ... \longrightarrow \langle \hat{e}_{k-1}, s_{k-1} \rangle$

and $e_1 = (\ell := \hat{e}_1)$, $e_2 = (\ell := \hat{e}_2)$,..., $e_{k-1} = (\ell := \hat{e}_{k-1})$ and for some n we have $\hat{e}_{k-1} = n$, $e_k = \textbf{skip}$, and $s_k = s_{k-1} + \{\ell \mapsto n\}$.

(the other possibility, of zero or more (assign1) reductions ending in a stuck state, is excluded by Theorems 2 and 3 (type preservation and progress))

Now, if $\langle \ell := e, s \rangle \longrightarrow^{\omega}$, by the above there is an infinite reduction sequence for $\langle e, s \rangle$, so by $e \ \simeq^{T}_{\Gamma} \ e'$ there is an infinite reduction sequence of $\langle e', s \rangle$, so (using (assign2)) there is an infinite reduction sequence of $\langle \ell := e', s \rangle$.

On the other hand, if $\neg(\langle \ell := e, s \rangle \longrightarrow^{\omega})$ then by the above there is some n and s_{k-1} such that $\langle e, s \rangle \longrightarrow^{*} \langle n, s_{k-1} \rangle$ and $\langle \ell := e, s \rangle \longrightarrow \langle \textbf{skip}, s_{k-1} + \{\ell \mapsto n\} \rangle$. By $e \ \simeq^{T}_{\Gamma} \ e'$ we have $\langle e', s \rangle \longrightarrow^{*} \langle n, s_{k-1} \rangle$. Then using (assign1) $\langle \ell := e', s \rangle \longrightarrow^{*} \langle \ell := n, s_{k-1} \rangle \longrightarrow \langle \textbf{skip}, s_{k-1} + \{\ell \mapsto n\} = \langle e_k, s_k \rangle$ as required.

□

Back to the Examples

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

1. $\langle e_1, s\rangle \longrightarrow^\omega$ and $\langle e_2, s\rangle \longrightarrow^\omega$, or
2. for some v, s' we have $\langle e_1, s\rangle \longrightarrow^* \langle v, s'\rangle$ and $\langle e_2, s\rangle \longrightarrow^* \langle v, s'\rangle$.

So:

$2 + 2 \simeq^{\text{int}}_\Gamma 4$ for any Γ

$(l := 0; 4) \not\simeq^{\text{int}}_\Gamma (l := 1; 3{+}!l)$ for any Γ

$(l :=!l + 1); (l :=!l - 1) \simeq^{\text{unit}}_\Gamma (l :=!l)$ for any Γ including l:intref

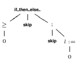
And the general laws?

Conjecture 1 $e_1; (e_2; e_3) \simeq^T_\Gamma (e_1; e_2); e_3$ *for any* Γ, T, e_1, e_2 *and* e_3 *such that* $\Gamma \vdash e_1$:unit, $\Gamma \vdash e_2$:unit, *and* $\Gamma \vdash e_3{:}T$

Conjecture 2
$((\textbf{\textit{if}}\ e_1\ \textbf{\textit{then}}\ e_2\ \textbf{\textit{else}}\ e_3); e) \simeq^T_\Gamma (\textbf{\textit{if}}\ e_1\ \textbf{\textit{then}}\ e_2; e\ \textbf{\textit{else}}\ e_3; e)$ *for any* Γ, T, e, e_1, e_2 *and* e_3 *such that* $\Gamma \vdash e_1$:bool, $\Gamma \vdash e_2$:unit, $\Gamma \vdash e_3$:unit, *and* $\Gamma \vdash e{:}T$

Conjecture 3
$(e; (\textbf{\textit{if}}\ e_1\ \textbf{\textit{then}}\ e_2\ \textbf{\textit{else}}\ e_3)) \simeq^T_\Gamma (\textbf{\textit{if}}\ e_1\ \textbf{\textit{then}}\ e; e_2\ \textbf{\textit{else}}\ e; e_3)$ *for any* Γ, T, e, e_1, e_2 *and* e_3 *such that* $\Gamma \vdash e$:unit, $\Gamma \vdash e_1$:bool, $\Gamma \vdash e_2{:}T$, *and* $\Gamma \vdash e_3{:}T$

Q: Is a typed expression $\Gamma \vdash e{:}T$, e.g.
l:intref $\vdash$ **if** $!l \geq 0$ **then skip else** (**skip**; $l := 0$):unit:

1. a list of characters `[’i’, ’f’, ’_’, ’!’, ’l’, ..];`
2. a list of tokens `[ IF, DEREF, LOC "l", GTEQ, ..];`
3. an abstract syntax tree 

;
4. the function taking store s to the reduction sequence $\langle e, s\rangle \longrightarrow \langle e_1, s_1\rangle \longrightarrow \langle e_2, s_2\rangle \longrightarrow$ ...; or
5. - the equivalence class $\{e' \mid e \simeq^T_\Gamma e'\}$
   - the partial function $[\![e]\!]_\Gamma$ that takes any store s with dom(s) = dom(Γ) and either is undefined, if $\langle e, s\rangle \longrightarrow^\omega$, or is $\langle v, s'\rangle$, if $\langle e, s\rangle \longrightarrow^* \langle v, s'\rangle$

(the Determinacy theorem tells us that this is a definition of a function).

Suppose $\Gamma \vdash e_1$:unit and $\Gamma \vdash e_2$:unit.

When *is* $e_1; e_2 \simeq^{\text{unit}}_\Gamma e_2; e_1$?

A sufficient condition: they don’t mention any locations (but not necessary... e.g. if e_1 does but e_2 doesn’t)

A weaker sufficient condition: they don’t mention any of the same locations. (but not necessary... e.g. if they both just read l)

An even weaker sufficient condition: we can regard each expression as a partial function over stores with domain $\mathrm{dom}(\Gamma)$. Say $[\![e_i]\!]_\Gamma$ is the function that takes a store s with $\mathrm{dom}(s) = \mathrm{dom}(\Gamma)$ and either is undefined, if $\langle e_i, s\rangle \longrightarrow^\omega$, or is s', if $\langle e_i, s\rangle \longrightarrow^* \langle (), s'\rangle$ (the Determinacy theorem tells us that this is a definition of a function).

For each location ℓ in $\mathrm{dom}(\Gamma)$, say e_i *semantically depends on* ℓ if there exists s, n such that $[\![e_i]\!]_\Gamma(s) \neq [\![e_i]\!]_\Gamma(s + \{\ell \mapsto n\}))$. (note this is much weaker than "e_i contains an dereference of ℓ")

Say e_i *semantically affects* ℓ if there exists s such that $s(\ell) \neq [\![e_i]\!]_\Gamma(s)(\ell)$. (note this is much weaker than "e_i contains an assignment to ℓ")

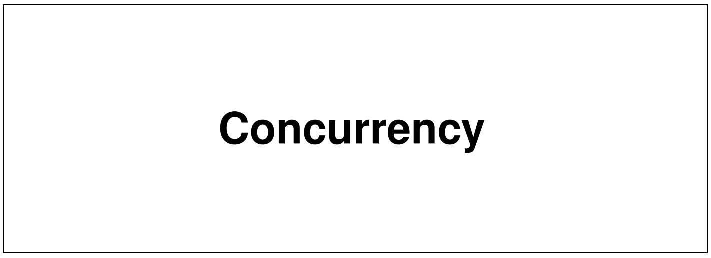
Now $e_1; e_2 \simeq^{\mathsf{unit}}_\Gamma e_2; e_1$ if there is no ℓ that is depended on by one e_i and affected by the other.

(sill not necessary...?)

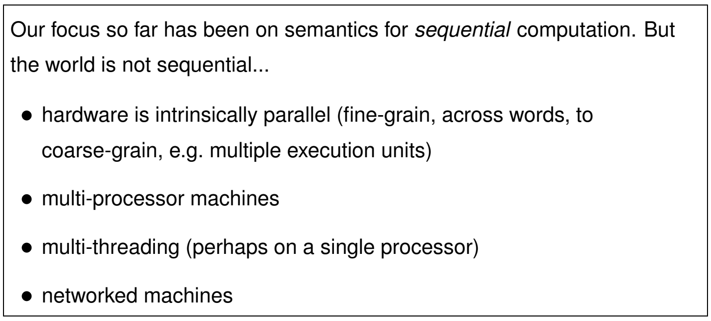
6.1 Exercises

Exercise 31 ★★*Prove some of the other cases of the Congruence theorem.*

7 Concurrency

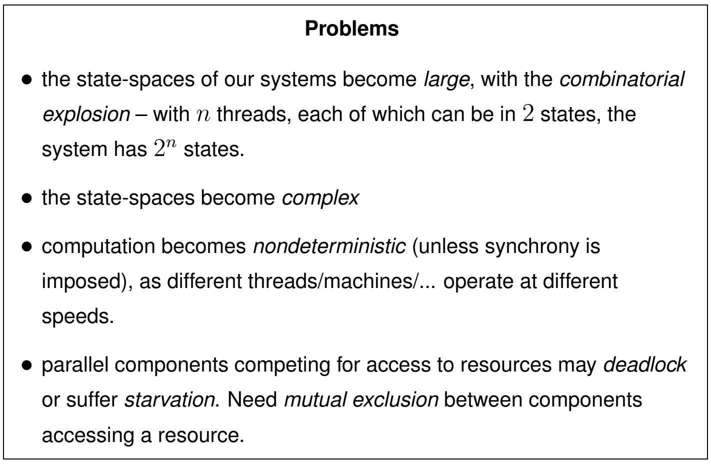


Concurrency



Our focus so far has been on semantics for *sequential* computation. But the world is not sequential...

- hardware is intrinsically parallel (fine-grain, across words, to coarse-grain, e.g. multiple execution units)
- multi-processor machines
- multi-threading (perhaps on a single processor)
- networked machines



Problems

- the state-spaces of our systems become *large*, with the *combinatorial explosion* – with n threads, each of which can be in 2 states, the system has 2^n states.
- the state-spaces become *complex*
- computation becomes *nondeterministic* (unless synchrony is imposed), as different threads/machines/... operate at different speeds.
- parallel components competing for access to resources may *deadlock* or suffer *starvation*. Need *mutual exclusion* between components accessing a resource.

More Problems!

- *partial failure* (of some processes, of some machines in a network, of some persistent storage devices). Need *transactional mechanisms*.
- *communication between different environments* (with different local resources (e.g. different local stores, or libraries, or...)
- *partial version change*
- communication between administrative regions with *partial trust* (or, indeed, *no trust*); protection against mailicious attack.
- dealing with contingent complexity (embedded historical accidents; upwards-compatible deltas)

Theme: as for sequential languages, but much more so, it's a complicated world.

Aim of this lecture: just to give you a taste of how a little semantics can be used to express some of the fine distinctions. Primarily (1) to boost your intuition for informal reasoning, but also (2) this can support rigorous proof about really hairy crypto protocols, cache-coherency protocols, comms, database transactions,....

Going to define the simplest possible (well, almost) concurrent language, call it L1$_{|}$, and explore a few issues. You've seen most of them informally in CSAA.

Booleans $b \in \mathbb{B} = \{\textbf{true}, \textbf{false}\}$
Integers $n \in \mathbb{Z} = \{..., -1, 0, 1, ...\}$
Locations $\ell \in \mathbb{L} = \{l, l_0, l_1, l_2, ...\}$

Operations $op ::= + \mid \geq$

Expressions

$$\begin{array}{rcl} e & ::= & n \mid b \mid e_1 \;\; op \;\; e_2 \mid \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \mid \\ & & \ell := e \mid !\ell \mid \\ & & \textbf{skip} \mid e_1 ; e_2 \mid \\ & & \textbf{while } e_1 \textbf{ do } e_2 \mid \\ & & e_1 \,|\, e_2 \end{array}$$

$$\begin{array}{rcl} T & ::= & \mathsf{int} \mid \mathsf{bool} \mid \mathsf{unit} \mid \mathsf{proc} \\ T_{loc} & ::= & \mathsf{intref} \end{array}$$

Parallel Composition: Typing and Reduction

(thread) $\dfrac{\Gamma \vdash e{:}\mathsf{unit}}{\Gamma \vdash e{:}\mathsf{proc}}$

(parallel) $\dfrac{\Gamma \vdash e_1{:}\mathsf{proc} \qquad \Gamma \vdash e_2{:}\mathsf{proc}}{\Gamma \vdash e_1 \mathbin{|} e_2{:}\mathsf{proc}}$

(parallel1) $\dfrac{\langle e_1, s\rangle \longrightarrow \langle e_1', s'\rangle}{\langle e_1 \mathbin{|} e_2, s\rangle \longrightarrow \langle e_1' \mathbin{|} e_2, s'\rangle}$

(parallel2) $\dfrac{\langle e_2, s\rangle \longrightarrow \langle e_2', s'\rangle}{\langle e_1 \mathbin{|} e_2, s\rangle \longrightarrow \langle e_1 \mathbin{|} e_2', s'\rangle}$

Parallel Composition: Design Choices

- threads don't return a value
- threads don't have an identity
- termination of a thread cannot be observed within the language
- threads aren't partitioned into 'processes' or machines
- threads can't be killed externally

Threads execute asynchronously – the semantics allows any interleaving of the reductions of the threads.

All threads can read and write the shared memory.

$$\begin{tikzcd}
 & \langle () \mathbin{|} l := 2, \{l \mapsto 1\}\rangle \arrow[r] & \langle () \mathbin{|} (), \{l \mapsto 2\}\rangle \\
\langle l := 1 \mathbin{|} l := 2, \{l \mapsto 0\}\rangle \arrow[ur] \arrow[dr] & & \\
 & \langle l := 1 \mathbin{|} (), \{l \mapsto 2\}\rangle \arrow[r] & \langle () \mathbin{|} (), \{l \mapsto 1\}\rangle
\end{tikzcd}$$

But, assignments and dereferencing are *atomic*. For example,

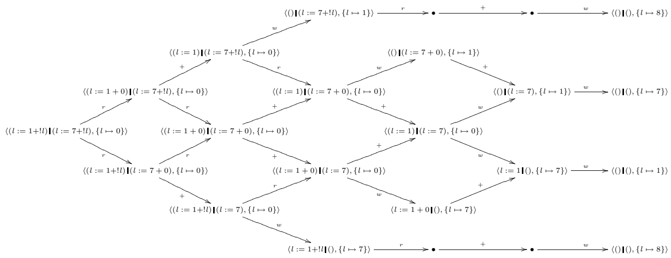
$\langle l := 3498734590879238429384 \mid l := 7, \{l \mapsto 0\}\rangle$

will reduce to a state with l either 3498734590879238429384 or 7, not something with the first word of one and the second word of the other. Implement?

But but, in $(l := e) \mathbin{|} e'$, the steps of evaluating e and e' can be interleaved.

Think of $(l := 1{+}!l) \mathbin{|} (l := 7{+}!l)$ – there are races....

The behaviour of $(l := 1+!l) \parallel (l := 7+!l)$ for the initial store $\{l \mapsto 0\}$:


$\langle () \parallel (l := 7+!l), \{l \mapsto 1\}\rangle$
r
+
w
$\langle () \parallel (), \{l \mapsto 8\}\rangle$
w
$\langle (l := 1) \parallel (l := 7+!l), \{l \mapsto 0\}\rangle$
$\langle () \parallel (l := 7 + 0), \{l \mapsto 1\}\rangle$
+
r
w
+
$\langle (l := 1 + 0) \parallel (l := 7+!l), \{l \mapsto 0\}\rangle$
$\langle (l := 1) \parallel (l := 7 + 0), \{l \mapsto 0\}\rangle$
$\langle () \parallel (l := 7), \{l \mapsto 1\}\rangle$
w
$\langle () \parallel (), \{l \mapsto 7\}\rangle$
r
r
+
+
w
$\langle (l := 1+!l) \parallel (l := 7+!l), \{l \mapsto 0\}\rangle$
$\langle (l := 1 + 0) \parallel (l := 7 + 0), \{l \mapsto 0\}\rangle$
$\langle (l := 1) \parallel (l := 7), \{l \mapsto 0\}\rangle$
r
r
+
+
w
$\langle (l := 1+!l) \parallel (l := 7 + 0), \{l \mapsto 0\}\rangle$
$\langle (l := 1 + 0) \parallel (l := 7), \{l \mapsto 0\}\rangle$
$\langle l := 1 \parallel (), \{l \mapsto 7\}\rangle$
w
$\langle () \parallel (), \{l \mapsto 1\}\rangle$
+
r
w
+
$\langle (l := 1+!l) \parallel (l := 7), \{l \mapsto 0\}\rangle$
$\langle l := 1 + 0 \parallel (), \{l \mapsto 7\}\rangle$
w
$\langle l := 1+!l \parallel (), \{l \mapsto 7\}\rangle$
r
+
w
$\langle () \parallel (), \{l \mapsto 8\}\rangle$

Note that the labels $+$, w and r in this 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...

Morals

- There really is a combinatorial explosion – and you don't want to be standing next to it...
- Drawing state-space diagrams only works for really tiny examples – we need better techniques for analysis.
- Almost certainly you (as the programmer) didn't want all those 3 outcomes to be possible – need better idioms or constructs for programming.

So, how do we get anything coherent done?

Need some way(s) to synchronise between threads, so can enforce *mutual exclusion* for shared data.

cf. Lamport's "Bakery" algorithm from Concurrent Systems and Applications. Can you code that in L1$_l$? If not, what's the smallest extension required?

Usually, though, you can depend on built-in support from the scheduler, e.g. for *mutexes* and *condition variables* (or, at a lower level, `tas` or `cas`).

See this – in the library – for a good discussion of mutexes and condition variables: A. Birrell, J. Guttag, J. Horning, and R. Levin. *Thread synchronization: a Formal Specification.* In G. Nelson, editor, System Programming with Modula-3, chapter 5, pages 119-129. Prentice-Hall, 1991.

See N. Lynch. *Distributed Algorithms* for other mutual exclusion algorithms (and much else besides).

Consider simple mutexes, with commands to lock an unlocked mutex and to unlock a locked mutex (and do nothing for an unlock of an unlocked mutex).

Adding Primitive Mutexes

Mutex names $m \in \mathbb{M} = \{\mathrm{m}, \mathrm{m}_1, ...\}$

Configurations $\langle e, s, M\rangle$ where $M{:}\mathbb{M} \to \mathbb{B}$ is the mutex state

Expressions $e ::= ...\mid$ **lock** $m \mid$ **unlock** m

$$\text{(lock)}\ \frac{}{\Gamma \vdash \textbf{lock}\ \ m{:}\text{unit}} \qquad \text{(unlock)}\ \frac{}{\Gamma \vdash \textbf{unlock}\ \ m{:}\text{unit}}$$

$$\text{(lock)}\quad \langle \textbf{lock}\ \ m, s, M\rangle \longrightarrow \langle (), s, M + \{m \mapsto \textbf{true}\}\rangle \text{ if } \neg M(m)$$

$$\text{(unlock)}\quad \langle \textbf{unlock}\ \ m, s, M\rangle \longrightarrow \langle (), s, M + \{m \mapsto \textbf{false}\}\rangle$$

Note that (lock) *atomically* (a) checks the mutex is currently false, (b) changes its state, and (c) lets the thread proceed.

Also, there is no record of which thread is holding a locked mutex.

Need to adapt all the other semantic rules to carry the mutex state M around. For example, replace

$$\text{(op2)}\ \frac{\langle e_2, s\rangle \longrightarrow \langle e_2', s'\rangle}{\langle v\ \ op\ \ e_2, s\rangle \longrightarrow \langle v\ \ op\ \ e_2', s'\rangle}$$

by

$$\text{(op2)}\ \frac{\langle e_2, s, M\rangle \longrightarrow \langle e_2', s', M'\rangle}{\langle v\ \ op\ \ e_2, s, M\rangle \longrightarrow \langle v\ \ op\ \ e_2', s', M'\rangle}$$

(note, the M and s must behave the same wrt evaluation order).

Using a Mutex

Consider

$e = (\textbf{lock}\ \ \mathrm{m}; l := 1{+}!l; \textbf{unlock}\ \ \mathrm{m}) \,|\, (\textbf{lock}\ \ \mathrm{m}; l := 7{+}!l; \textbf{unlock}\ \ \mathrm{m})$

The behaviour of $\langle e, s, M\rangle$, with the initial store $s = \{l \mapsto 0\}$ and initial mutex state $M_0 = \lambda m \in \mathbb{M}.\textbf{false}$, is:

$$\begin{tikzcd}
 & \langle (l := 1{+}!l; \textbf{unlock}\ \mathrm{m}) \,|\, (\textbf{lock}\ \mathrm{m}; l := 7{+}!l; \textbf{unlock}\ \mathrm{m}), s, M'\rangle \arrow[dr, "12"] & \\
\langle e, s, M_0\rangle \arrow[ur, "\textbf{lock}\ \mathrm{m}"] \arrow[dr, "\textbf{lock}\ \mathrm{m}"'] & & \langle () \,|\, (), \{l \mapsto 8\}, M\rangle \\
 & \langle (\textbf{lock}\ \mathrm{m}; l := 1{+}!l; \textbf{unlock}\ \mathrm{m}) \,|\, (l := 7{+}!l; \textbf{unlock}\ \mathrm{m}), s, M'\rangle \arrow[ur, "12"'] &
\end{tikzcd}$$

(where $M' = M_0 + \{\mathrm{m} \mapsto \textbf{true}\}$)

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.

On the downside, we've also lost any performance benefits of concurrency (for this program that's fine, but in general there's some other computation that wouldn't conflict and so could be done in parallel).

Using Several Mutexes

lock m can block (that's the point). Hence, you can *deadlock*.

$$\begin{array}{rl} e = & (\textbf{lock}\ \mathrm{m}_1; \textbf{lock}\ \mathrm{m}_2; l_1 := !l_2; \textbf{unlock}\ \mathrm{m}_1; \textbf{unlock}\ \mathrm{m}_2) \\ \mid & (\textbf{lock}\ \mathrm{m}_2; \textbf{lock}\ \mathrm{m}_1; l_2 := !l_1; \textbf{unlock}\ \mathrm{m}_1; \textbf{unlock}\ \mathrm{m}_2) \end{array}$$

Locking Disciplines

So, suppose we have several programs $e_1, ..., e_k$, all well-typed with $\Gamma \vdash e_i{:}\mathtt{unit}$, that we want to execute concurrently without 'interference' (whatever that is). Think of them as transaction bodies.

There are many possible locking disciplines. We'll focus on one, to see how it – and the properties it guarantees – can be made precise and proved.

An Ordered 2PL Discipline, Informally

Fix an association between locations and mutexes. For simplicity, make it 1:1 – associate l with m, l_1 with m_1, etc.

Fix a lock acquisition order. For simplicity, make it $\mathrm{m}, \mathrm{m}_0, \mathrm{m}_1, \mathrm{m}_2, ...$

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, $(e_1 \mid ... \mid e_k)$ should (a) never deadlock, and (b) be *serialisable* – any execution of it should be 'equivalent' to an execution of $e_{\pi(1)}; ...; e_{\pi(k)}$ for some permutation π.

These are *semantic properties* again. In general, it won't be computable whether they hold. For simple e_i, though, it's often obvious. Further, one can construct syntactic disciplines that are checkable and are sufficient to guarantee these.

See *Transactional Information Systems*, Gerhard Weikum and Gottfried Vossen, for much more detail on locking disciplines etc. (albeit not from a programming-language semantics perspective).

Problem: Need a Thread-Local Semantics

Our existing semantics defines the behaviour only of global configurations $\langle e, s, M\rangle$. To state properties of subexpressions, e.g.

- e_i acquires the lock m_j for each location l_j it uses, before it uses it

which really means

- in any execution of $\langle (e_1 \mid ... \mid e_i \mid ... \mid e_k), s, M\rangle$, e_i acquires the lock m_j for each location l_j it uses, before it uses it

we need some notion of the behaviour of the thread e_i on its own

Solution: Write One Down

Instead of only defining the global $\langle e, s, M\rangle \longrightarrow \langle e', s', M'\rangle$, with rules

$$\text{(assign1)}\quad \langle \ell := n, s, M\rangle \longrightarrow \langle \textbf{skip}, s + \{\ell \mapsto n\}, M\rangle \quad \text{if } \ell \in \mathsf{dom}(s)$$

$$\text{(parallel1)}\quad \frac{\langle e_1, s, M\rangle \longrightarrow \langle e_1', s', M'\rangle}{\langle e_1 \mathbin{|} e_2, s, M\rangle \longrightarrow \langle e_1' \mathbin{|} e_2, s', M'\rangle}$$

define a per-thread $e \xrightarrow{a} e'$ and use that to define $\langle e, s, M\rangle \longrightarrow \langle e', s', M'\rangle$, with rules like

$$\text{(t-assign1)}\quad \ell := n \xrightarrow{\ell:=n} \textbf{skip}$$

$$\text{(t-parallel1)}\quad \frac{e_1 \xrightarrow{a} e_1'}{e_1 \mathbin{|} e_2 \xrightarrow{a} e_1' \mathbin{|} e_2}$$

$$\text{(c-assign)}\quad \frac{e \xrightarrow{\ell:=n} e' \quad \ell \in \mathsf{dom}(s)}{\langle e, s, M\rangle \longrightarrow \langle e', s + \{\ell \mapsto n\}, M\rangle}$$

Note the per-thread rules don't mention s or M. Instead, we record in the label a what interactions with the store or mutexes it has.

$$a \quad ::= \quad \tau \mid \ell := n \mid !\ell = n \mid \textbf{lock}\ \ m \mid \textbf{unlock}\ \ m$$

Conventionally, τ (tau), stands for "no interactions", so $e \xrightarrow{\tau} e'$ if e does an internal step, not involving the store or mutexes.

Theorem 17 (Coincidence of global and thread-local semantics) *The two definitions of $\longrightarrow$ agree exactly.*

Proof strategy: a couple of rule inductions.

Write $\mathbb{A}$ for the set of all actions (explicitly, $\mathbb{A} = \{\tau\} \cup \{\ell := n \mid \ell \in \mathbb{L} \wedge n \in \mathbb{N}\} \cup \{!\ell = n \mid \ell \in \mathbb{L} \wedge n \in \mathbb{N}\} \cup \{\textbf{lock}\ \ m \mid m \in \mathbb{M}\} \cup \{\textbf{unlock}\ \ m \mid m \in M\}$).

Global Semantics

$$\text{(op +)}\quad \langle n_1 + n_2, s, M\rangle \longrightarrow \langle n, s, M\rangle \quad \text{if } n = n_1 + n_2$$

$$\text{(op }\geq\text{)}\quad \langle n_1 \geq n_2, s, M\rangle \longrightarrow \langle b, s, M\rangle \quad \text{if } b = (n_1 \geq n_2)$$

$$\text{(op1)}\quad \frac{\langle e_1, s, M\rangle \longrightarrow \langle e_1', s', M'\rangle}{\langle e_1\ \ op\ \ e_2, s, M\rangle \longrightarrow \langle e_1'\ \ op\ \ e_2, s', M'\rangle}$$

$$\text{(op2)}\quad \frac{\langle e_2, s, M\rangle \longrightarrow \langle e_2', s', M'\rangle}{\langle v\ \ op\ \ e_2, s, M\rangle \longrightarrow \langle v\ \ op\ \ e_2', s', M'\rangle}$$

$$\text{(deref)}\quad \langle !\ell, s, M\rangle \longrightarrow \langle n, s, M\rangle \quad \text{if } \ell \in \mathsf{dom}(s) \text{ and } s(\ell) = n$$

$$\text{(assign1)}\quad \langle \ell := n, s, M\rangle \longrightarrow \langle \textbf{skip}, s + \{\ell \mapsto n\}, M\rangle \quad \text{if } \ell \in \mathsf{dom}(s)$$

$$\text{(assign2)}\quad \frac{\langle e, s, M\rangle \longrightarrow \langle e', s', M'\rangle}{\langle \ell := e, s, M\rangle \longrightarrow \langle \ell := e', s', M'\rangle}$$

$$\text{(seq1)}\quad \langle \textbf{skip}; e_2, s, M\rangle \longrightarrow \langle e_2, s, M\rangle$$

$$\text{(seq2)}\quad \frac{\langle e_1, s, M\rangle \longrightarrow \langle e_1', s', M'\rangle}{\langle e_1; e_2, s, M\rangle \longrightarrow \langle e_1'; e_2, s', M'\rangle}$$

$$\text{(if1)}\quad \langle \textbf{if}\ \ \textbf{true}\ \ \textbf{then}\ \ e_2\ \ \textbf{else}\ \ e_3, s, M\rangle \longrightarrow \langle e_2, s, M\rangle$$

$$\text{(if2)}\quad \langle \textbf{if}\ \ \textbf{false}\ \ \textbf{then}\ \ e_2\ \ \textbf{else}\ \ e_3, s, M\rangle \longrightarrow \langle e_3, s, M\rangle$$

$$\text{(if3)}\quad \frac{\langle e_1, s, M\rangle \longrightarrow \langle e_1', s', M'\rangle}{\langle \textbf{if}\ \ e_1\ \ \textbf{then}\ \ e_2\ \ \textbf{else}\ \ e_3, s, M\rangle \longrightarrow \langle \textbf{if}\ \ e_1'\ \ \textbf{then}\ \ e_2\ \ \textbf{else}\ \ e_3, s', M'\rangle}$$

(while)

$$\langle \textbf{while}\ \ e_1\ \ \textbf{do}\ \ e_2, s, M\rangle \longrightarrow \langle \textbf{if}\ \ e_1\ \ \textbf{then}\ (e_2; \textbf{while}\ \ e_1\ \ \textbf{do}\ \ e_2)\ \textbf{else}\ \ \textbf{skip}, s, M\rangle$$

$$\text{(parallel1)}\quad \frac{\langle e_1, s, M\rangle \longrightarrow \langle e_1', s', M'\rangle}{\langle e_1 \mathbin{|} e_2, s, M\rangle \longrightarrow \langle e_1' \mathbin{|} e_2, s', M'\rangle}$$

$$\text{(parallel2)}\quad \frac{\langle e_2, s, M\rangle \longrightarrow \langle e_2', s', M'\rangle}{\langle e_1 \mathbin{|} e_2, s, M\rangle \longrightarrow \langle e_1 \mathbin{|} e_2', s', M'\rangle}$$

$$\text{(lock)}\quad \langle \textbf{lock}\ \ m, s, M\rangle \longrightarrow \langle (), s, M + \{m \mapsto \textbf{true}\}\rangle \text{ if } \neg M(m)$$

$$\text{(unlock)}\quad \langle \textbf{unlock}\ \ m, s, M\rangle \longrightarrow \langle (), s, M + \{m \mapsto \textbf{false}\}\rangle$$

Thread-Local Semantics

$$\text{(t-op +)}\quad n_1 + n_2 \xrightarrow{\tau} n \quad \text{if } n = n_1 + n_2$$

$$\text{(t-op }\geq\text{)}\quad n_1 \geq n_2 \xrightarrow{\tau} b \quad \text{if } b = (n_1 \geq n_2)$$

$$\text{(t-op1)}\quad \frac{e_1 \xrightarrow{a} e_1'}{e_1\ \ op\ \ e_2 \xrightarrow{a} e_1'\ \ op\ \ e_2}$$

$$\text{(t-op2)}\quad \frac{e_2 \xrightarrow{a} e_2'}{v\ \ op\ \ e_2 \xrightarrow{a} v\ \ op\ \ e_2'}$$

$$\text{(t-deref)}\quad !\ell \xrightarrow{!\ell=n} n$$

$$\text{(t-assign1)}\quad \ell := n \xrightarrow{\ell:=n} \textbf{skip}$$

$$\text{(t-assign2)}\quad \frac{e \xrightarrow{a} e'}{\ell := e \xrightarrow{a} \ell := e'}$$

$$\text{(t-seq1)}\quad \textbf{skip}; e_2 \xrightarrow{\tau} e_2$$

$$\text{(t-seq2)}\quad \frac{e_1 \xrightarrow{a} e_1'}{e_1; e_2 \xrightarrow{a} e_1'; e_2}$$

$$\text{(t-if1)}\quad \textbf{if}\ \ \textbf{true}\ \ \textbf{then}\ \ e_2\ \ \textbf{else}\ \ e_3 \xrightarrow{\tau} e_2$$

$$\text{(t-if2)}\quad \textbf{if}\ \ \textbf{false}\ \ \textbf{then}\ \ e_2\ \ \textbf{else}\ \ e_3 \xrightarrow{\tau} e_3$$

$$\text{(t-if3)}\quad \frac{e_1 \xrightarrow{a} e_1'}{\textbf{if}\ \ e_1\ \ \textbf{then}\ \ e_2\ \ \textbf{else}\ \ e_3 \xrightarrow{a} \textbf{if}\ \ e_1'\ \ \textbf{then}\ \ e_2\ \ \textbf{else}\ \ e_3}$$

(t-while)

$$\textbf{while}\ \ e_1\ \ \textbf{do}\ \ e_2 \xrightarrow{\tau} \textbf{if}\ \ e_1\ \ \textbf{then}\ (e_2; \textbf{while}\ \ e_1\ \ \textbf{do}\ \ e_2)\ \textbf{else}\ \ \textbf{skip}$$

$$\text{(t-parallel1)}\quad \frac{e_1 \xrightarrow{a} e_1'}{e_1 \mathbin{|} e_2 \xrightarrow{a} e_1' \mathbin{|} e_2}$$

$$\text{(t-parallel2)}\quad \frac{e_2 \xrightarrow{a} e_2'}{e_1 \mathbin{|} e_2 \xrightarrow{a} e_1 \mathbin{|} e_2'}$$

$$\text{(t-lock)}\quad \textbf{lock}\ \ m \xrightarrow{\textbf{lock}\ \ m} ()$$

$$\text{(t-unlock)}\quad \textbf{unlock}\ \ m \xrightarrow{\textbf{unlock}\ \ m} ()$$

$$\text{(c-tau)}\quad \frac{e \xrightarrow{\tau} e'}{\langle e, s, M\rangle \longrightarrow \langle e', s, M\rangle}$$

$$\text{(c-assign)}\quad \frac{e \xrightarrow{\ell:=n} e' \quad \ell \in \mathsf{dom}(s)}{\langle e, s, M\rangle \longrightarrow \langle e', s + \{\ell \mapsto n\}, M\rangle} \qquad \text{(c-lock)}\quad \frac{e \xrightarrow{\textbf{lock}\ \ m} e' \quad \neg M(m)}{\langle e, s, M\rangle \longrightarrow \langle e', s, M + \{m \mapsto \textbf{true}\}\rangle}$$

$$\text{(c-deref)}\quad \frac{e \xrightarrow{!\ell=n} e' \quad \ell \in \mathsf{dom}(s) \wedge s(\ell) = n}{\langle e, s, M\rangle \longrightarrow \langle e', s, M\rangle} \qquad \text{(c-unlock)}\quad \frac{e \xrightarrow{\textbf{unlock}\ \ m} e'}{\langle e, s, M\rangle \longrightarrow \langle e', s, M + \{m \mapsto \textbf{false}\}\rangle}$$

Example of Thread-local transitions

For $e = (\textbf{lock}\ \mathrm{m}; (l := 1+!l; \textbf{unlock}\ \mathrm{m}))$ we have

$$\begin{array}{lll}
e & \xrightarrow{\textbf{lock}\ \mathrm{m}} & \textbf{skip}; (l := 1+!l; \textbf{unlock}\ \mathrm{m}) \\
 & \xrightarrow{\tau} & (l := 1+!l; \textbf{unlock}\ \mathrm{m}) \\
 & \xrightarrow{!l=n} & (l := 1+n; \textbf{unlock}\ \mathrm{m}) \quad \text{for any } n \in \mathbb{Z} \\
 & \xrightarrow{\tau} & (l := n'; \textbf{unlock}\ \mathrm{m}) \quad \text{for } n' = 1+n \\
 & \xrightarrow{l:=n'} & \textbf{skip}; \textbf{unlock}\ \mathrm{m} \\
 & \xrightarrow{\tau} & \textbf{unlock}\ \mathrm{m} \\
 & \xrightarrow{\textbf{unlock}\ \mathrm{m}} & \textbf{skip}
\end{array}$$

Hence, using (t-parallel) and the (c-*) rules, for $s' = s + \{l \mapsto 1 + s(l)\}$,
$\langle e \,|\, e', s, M_0\rangle \longrightarrow\longrightarrow\longrightarrow\longrightarrow\longrightarrow\longrightarrow\longrightarrow \langle \textbf{skip} \,|\, e', s', M_0\rangle$

(need $l \in \mathrm{dom}(s)$ 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*.

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 $a_k = \textbf{lock}\ m_j$ without an intervening $\textbf{unlock}\ m_j$.
- for each j, the subsequence of $a_1, a_2, \ldots$ with labels $\textbf{lock}\ \mathrm{m}_j$ and $\textbf{unlock}\ \mathrm{m}_j$ is a prefix of $((\textbf{lock}\ \mathrm{m}_j)(\textbf{unlock}\ \mathrm{m}_j))^*$. Moreover, if $\neg(e_k \xrightarrow{a})$ then the subsequence does not end in a $\textbf{lock}\ \mathrm{m}_j$.
- if $a_i = \textbf{lock}\ \mathrm{m}_j$ and $a_{i'} = \textbf{unlock}\ \mathrm{m}_{j'}$ then $i < i'$
- if $a_i = \textbf{lock}\ \mathrm{m}_j$ and $a_{i'} = \textbf{lock}\ \mathrm{m}_{j'}$ and $i < i'$ then $j < j'$

... and make the guaranteed properties precise

Say $e_1, \ldots, e_k$ are *serialisable* if for any initial store s, if
$\langle (e_1 | \ldots | e_k), s, M_0\rangle \longrightarrow^* \langle e, s', M'\rangle \not\longrightarrow$ then for some permutation π we have $\langle e_{\pi(1)}; \ldots; e_{\pi(k)}, s, M_0\rangle \longrightarrow^* \langle e', s', M'\rangle$.

Say they are *deadlock-free* if for any initial store s, if
$\langle (e_1 | \ldots | e_k), s, M_0\rangle \longrightarrow^* \langle e, s', M\rangle \not\longrightarrow$ then not $e \xrightarrow{\textbf{lock}\ m} e'$,
i.e. e does not contain any blocked $\textbf{lock}\ m$ subexpressions.

(Warning: there are many subtle variations of these properties!)

The Theorem

Conjecture 4 *If each e_i obeys the discipline, then $e_1, ...e_k$ are serialisable and deadlock-free.*

(may be false!)

Proof strategy: Consider a (derivation of a) computation

$\langle (e_1 | ... | e_k), s, M_0 \rangle \longrightarrow \langle \hat{e}_1, s_1, M_1 \rangle \longrightarrow \langle \hat{e}_2, s_2, M_2 \rangle \longrightarrow ...$

We know each $\hat{e}_i$ is a corresponding parallel composition. Look at the points at which each e_i acquires its final lock. That defines a serialisation order. In between times, consider commutativity of actions of the different e_i – the premises guarantee that many actions are semantically independent, and so can be permuted.

We've not discussed *fairness* – the semantics allows any interleaving between parallel components, not only fair ones.

Language Properties

(Obviously!) don't have Determinacy.

Still have Type Preservation.

Have Progress, but it has to be modified – a well-typed expression of type proc will reduce to some parallel composition of unit values.

Typing and type inference is scarcely changed.

(*very* fancy type systems can be used to enforce locking disciplines)

7.1 Exercises

Exercise 32 ★★*Are the mutexes specified here similar to those described in CSAA?*

Exercise 33 ★★*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 34 ★★★★*Prove the Conjecture about it.*

Exercise 35 ★★★*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 Subtyping and Objects

Subtyping and Objects

Our type systems so far would all be annoying to use, as they're quite rigid (Pascal-like). There is no support for code reuse (except for functions), 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 α takes an argument of type α list and computes its length (parametric - uniform in whatever α is)

- Subtype polymorphism – as in various OO languages. See here.

  Dating back to the 1960s (Simula etc); formalised in 1980,1984,...

Subtyping – Motivation

Recall

$$\text{(app)}\ \frac{\Gamma \vdash e_1{:}T \to T' \qquad \Gamma \vdash e_2{:}T}{\Gamma \vdash e_1\ e_2{:}T'}$$

so can't type

$$\nvdash (\textbf{fn}\ \ \mathrm{x}{:}\{\mathrm{p}{:}\mathsf{int}\} \Rightarrow \#\mathrm{p}\ \mathrm{x})\ \{\mathrm{p} = 3, \mathrm{q} = 4\} : \mathsf{int}$$

even though we're giving the function a *better* argument, with more structure, than it needs.

Subsumption

'Better'? Any value of type $\{\mathrm{p}{:}\mathsf{int}, \mathrm{q}{:}\mathsf{int}\}$ can be used wherever a value of type $\{\mathrm{p}{:}\mathsf{int}\}$ is expected. (*)

Introduce a *subtyping relation* between types, written $T <: T'$, read as T is a subtype of T' (a T is useful in more contexts than a T').

Will define it on the next slides, but it will include
$\{\mathrm{p}{:}\mathsf{int}, \mathrm{q}{:}\mathsf{int}\} <: \{\mathrm{p}{:}\mathsf{int}\} <: \{\}$

Introduce a *subsumption rule*

$$\text{(sub)}\quad \frac{\Gamma \vdash e{:}T \qquad T <: T'}{\Gamma \vdash e{:}T'}$$

allowing subtyping to be used, capturing (*).

Can then deduce $\{\mathrm{p} = 3, \mathrm{q} = 4\}{:}\{\mathrm{p}{:}\mathsf{int}\}$, hence can type the example.

Example

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathrm{x}{:}\{\mathrm{p}{:}\mathsf{int}\} \vdash \mathrm{x}{:}\{\mathrm{p}{:}\mathsf{int}\}}\,\text{(var)}}{\mathrm{x}{:}\{\mathrm{p}{:}\mathsf{int}\} \vdash \#\mathrm{p}\ \mathrm{x}{:}\mathsf{int}}\,\text{(record-proj)}}{\{\} \vdash (\mathbf{fn}\ \ \mathrm{x}{:}\{\mathrm{p}{:}\mathsf{int}\} \Rightarrow \#\mathrm{p}\ \mathrm{x}){:}\{\mathrm{p}{:}\mathsf{int}\} \rightarrow \mathsf{int}}\,\text{(fn)} \quad \dfrac{\dfrac{\dfrac{}{\{\} \vdash 3{:}\mathsf{int}}\,\text{(var)} \quad \dfrac{}{\{\} \vdash 4{:}\mathsf{int}}\,\text{(var)}}{\{\} \vdash \{\mathrm{p} = 3, \mathrm{q} = 4\}{:}\{\mathrm{p}{:}\mathsf{int}, \mathrm{q}{:}\mathsf{int}\}}\,\text{(record)} \quad (\bigstar)}{\{\} \vdash \{\mathrm{p} = 3, \mathrm{q} = 4\}{:}\{\mathrm{p}{:}\mathsf{int}\}}\,\text{(sub)}}{\{\} \vdash (\mathbf{fn}\ \ \mathrm{x}{:}\{\mathrm{p}{:}\mathsf{int}\} \Rightarrow \#\mathrm{p}\ \mathrm{x})\{\mathrm{p} = 3, \mathrm{q} = 4\}{:}\mathsf{int}}\,\text{(app)}$$

where $(\bigstar)$ is $\{\mathrm{p}{:}\mathsf{int}, \mathrm{q}{:}\mathsf{int}\} <: \{\mathrm{p}{:}\mathsf{int}\}$

Now, how do we define the subtype relation? First:

The Subtype Relation $\boxed{T <: T'}$

$$\text{(s-refl)}\quad \frac{}{T <: T}$$

$$\text{(s-trans)}\quad \frac{T <: T' \qquad T' <: T''}{T <: T''}$$

Now have to look at each type

Subtyping – Records

Forgetting fields on the right:

$\{lab_1{:}T_1, .., lab_k{:}T_k, lab_{k+1}{:}T_{k+1}, .., lab_{k+k'}{:}T_{k+k'}\}$

$<:$ (s-record-width)

$\{lab_1{:}T_1, .., lab_k{:}T_k\}$

Allowing subtyping within fields:

$$\text{(s-record-depth)}\quad \frac{T_1 <: T'_1 \quad .. \quad T_k <: T'_k}{\{lab_1{:}T_1, .., lab_k{:}T_k\} <: \{lab_1{:}T'_1, .., lab_k{:}T'_k\}}$$

Combining these:

$$\dfrac{\dfrac{}{\{\mathrm{p}{:}\mathsf{int}, \mathrm{q}{:}\mathsf{int}\} <: \{\mathrm{p}{:}\mathsf{int}\}}\,\text{(s-record-width)} \quad \dfrac{}{\{\mathrm{r}{:}\mathsf{int}\} <: \{\}}\,\text{(s-record-width)}}{\{\mathrm{x}{:}\{\mathrm{p}{:}\mathsf{int}, \mathrm{q}{:}\mathsf{int}\}, \mathrm{y}{:}\{\mathrm{r}{:}\mathsf{int}\}\} <: \{\mathrm{x}{:}\{\mathrm{p}{:}\mathsf{int}\}, \mathrm{y}{:}\{\}\}}\,\text{(s-record-depth)}$$

Another example:

$$\dfrac{\dfrac{}{\{x{:}\{p{:}\mathsf{int},q{:}\mathsf{int}\},y{:}\{r{:}\mathsf{int}\}\} <: \{x{:}\{p{:}\mathsf{int},q{:}\mathsf{int}\}\}}\ \text{(s-record-width)} \qquad \dfrac{\dfrac{}{\{p{:}\mathsf{int},q{:}\mathsf{int}\} <: \{p{:}\mathsf{int}\}}\ \text{(s-record-width)}}{\{x{:}\{p{:}\mathsf{int},q{:}\mathsf{int}\}\} <: \{x{:}\{p{:}\mathsf{int}\}\}}\ \text{(s-record-depth)}}{\{x{:}\{p{:}\mathsf{int},q{:}\mathsf{int}\},y{:}\{r{:}\mathsf{int}\}\} <: \{x{:}\{p{:}\mathsf{int}\}\}}\ \text{(s-trans)}$$

Allowing reordering of fields:

(s-record-order)

$$\dfrac{\pi \text{ a permutation of } 1,..,k}{\{lab_1{:}T_1,..,lab_k{:}T_k\} <: \{lab_{\pi(1)}{:}T_{\pi(1)},..,lab_{\pi(k)}{:}T_{\pi(k)}\}}$$

(the subtype order is *not* anti-symmetric – it is a preorder, not a partial order)

Subtyping - Functions

$$\text{(s-fn)}\quad \dfrac{T_1' <: T_1 \qquad T_2 <: T_2'}{T_1 \to T_2 <: T_1' \to T_2'}$$

contravariant on the left of →

covariant on the right of → (like (s-record-depth))

If $f{:}T_1 \to T_2$ then we can give f any argument which is a subtype of T_1; we can regard the result of f as any supertype of T_2. e.g., for

$$f = \mathbf{fn}\ x{:}\{p{:}\mathsf{int}\} \Rightarrow \{p = \#p\ x, q = 28\}$$

we have

$$\{\} \vdash f{:}\{p{:}\mathsf{int}\} \to \{p{:}\mathsf{int},q{:}\mathsf{int}\}$$
$$\{\} \vdash f{:}\{p{:}\mathsf{int}\} \to \{p{:}\mathsf{int}\}$$
$$\{\} \vdash f{:}\{p{:}\mathsf{int},q{:}\mathsf{int}\} \to \{p{:}\mathsf{int},q{:}\mathsf{int}\}$$
$$\{\} \vdash f{:}\{p{:}\mathsf{int},q{:}\mathsf{int}\} \to \{p{:}\mathsf{int}\}$$

as

$$\{p{:}\mathsf{int},q{:}\mathsf{int}\} <: \{p{:}\mathsf{int}\}$$

On the other hand, for

$$\mathbf{fn}\ x{:}\{p{:}\mathsf{int},q{:}\mathsf{int}\} \Rightarrow \{p = (\#p\ x) + (\#q\ x)\}$$

we have

$$\{\} \vdash f{:}\{p{:}\mathsf{int},q{:}\mathsf{int}\} \to \{p{:}\mathsf{int}\}$$
$$\{\} \not\vdash f{:}\{p{:}\mathsf{int}\} \to T \quad \text{for any } T$$
$$\{\} \not\vdash f{:}T \to \{p{:}\mathsf{int},q{:}\mathsf{int}\} \quad \text{for any } T$$

Subtyping – Products

Just like (s-record-depth)

$$\text{(s-pair)}\quad \dfrac{T_1 <: T_1' \qquad T_2 <: T_2'}{T_1 * T_2 <: T_1' * T_2'}$$

Subtyping – Sums

Exercise.

Subtyping – References

Are either of these any good?

$$\frac{T <: T'}{T \ \mathsf{ref} <: T' \ \mathsf{ref}} \qquad \frac{T' <: T}{T \ \mathsf{ref} <: T' \ \mathsf{ref}}$$

No...

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

$$\frac{\Gamma \vdash e{:}T'}{\Gamma \vdash (T)e{:}T}$$

then you need a dynamic type-check...

This gives flexibility, but at the cost of many potential run-time errors. Many uses might be better handled by Parametric Polymorphism, aka Generics. (cf. work by Martin Odersky at EPFL, Lausanne, to be in Java 1.5)

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.

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

```
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
```

$Counter = \{\text{get}{:}\mathsf{unit} \to \mathsf{int},\ \text{inc}{:}\mathsf{unit} \to \mathsf{unit}\}$.

Using Subtyping

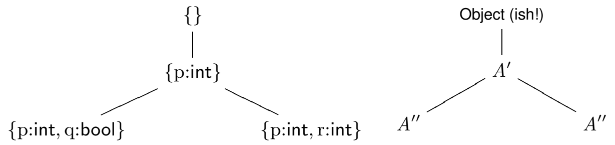
```
let val c:{get:unit → int, inc:unit → unit, reset:unit → unit} =
    let val x:int ref = ref 0 in
      {get = fn y:unit ⇒!x,
       inc = fn y:unit ⇒ x := 1+!x,
       reset = fn y:unit ⇒ x := 0}
    end
in
   (#inc c)(); (#get c)()
end
```

ResetCounter = {get:unit → int, inc:unit → unit, reset:unit → unit}

<: *Counter* = {get:unit → int, inc:unit → unit}.

Subtyping – Structural vs Named

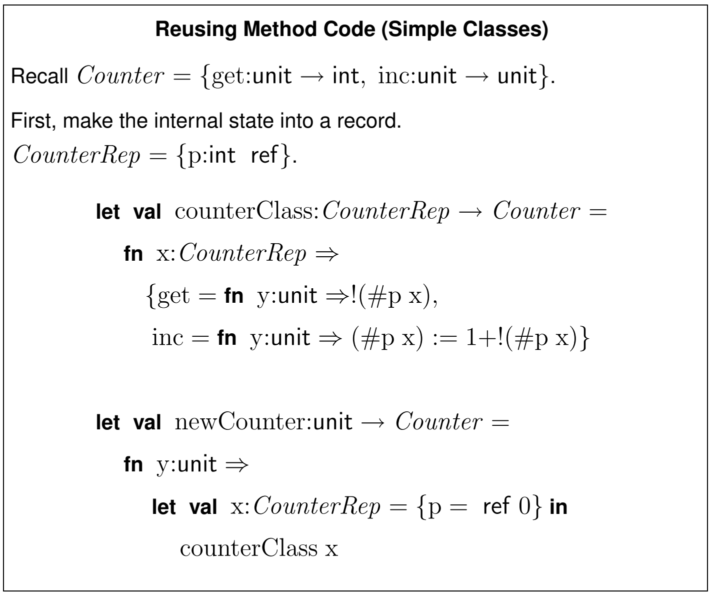
$$\begin{aligned} A' &= \{\}\ \text{with}\ \{\text{p:int}\} \\ A'' &= A'\ \text{with}\ \{\text{q:bool}\} \\ A''' &= A'\ \text{with}\ \{\text{r:int}\} \end{aligned}$$



Object Generators

```
let val newCounter:unit → {get:unit → int, inc:unit → unit} =
    fn y:unit ⇒
       let val x:int ref = ref 0 in
         {get = fn y:unit ⇒!x,
          inc = fn y:unit ⇒ x := 1+!x}
       end
in
   (#inc (newCounter ())) ()
end
```

and onwards to simple classes...



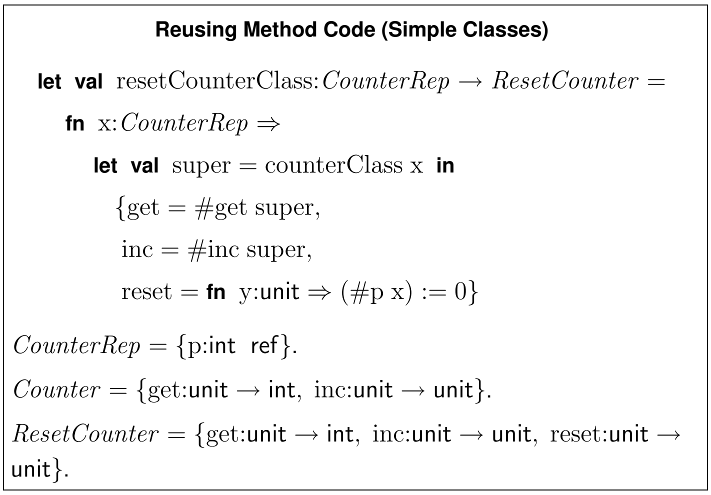
Reusing Method Code (Simple Classes)

Recall $\mathit{Counter} = \{\text{get:}\mathsf{unit} \rightarrow \mathsf{int},\ \text{inc:}\mathsf{unit} \rightarrow \mathsf{unit}\}$.

First, make the internal state into a record.

$\mathit{CounterRep} = \{\text{p:}\mathsf{int}\ \ \mathsf{ref}\}$.

```
let val counterClass:CounterRep → Counter =
  fn x:CounterRep ⇒
    {get = fn y:unit ⇒!(#p x),
     inc = fn y:unit ⇒ (#p x) := 1+!(#p x)}


let val newCounter:unit → Counter =
  fn y:unit ⇒
    let val x:CounterRep = {p = ref 0} in
      counterClass x
```



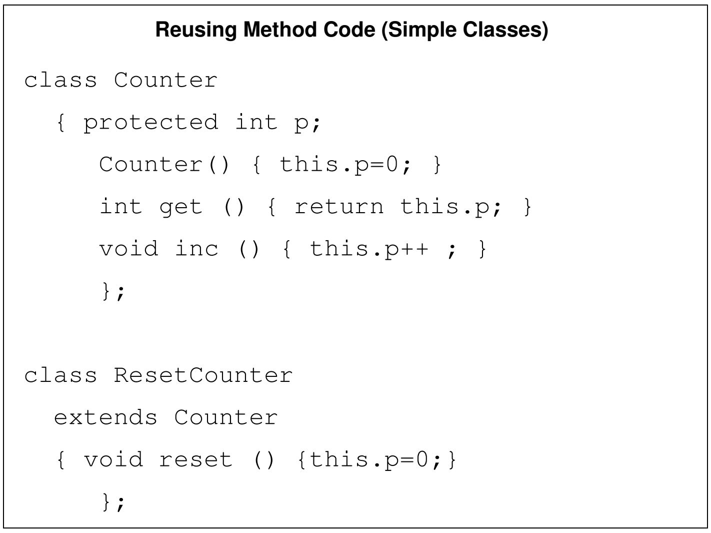
Reusing Method Code (Simple Classes)

```
let val resetCounterClass:CounterRep → ResetCounter =
  fn x:CounterRep ⇒
    let val super = counterClass x in
      {get = #get super,
       inc = #inc super,
       reset = fn y:unit ⇒ (#p x) := 0}
```

$\mathit{CounterRep} = \{\text{p:}\mathsf{int}\ \ \mathsf{ref}\}$.

$\mathit{Counter} = \{\text{get:}\mathsf{unit} \rightarrow \mathsf{int},\ \text{inc:}\mathsf{unit} \rightarrow \mathsf{unit}\}$.

$\mathit{ResetCounter} = \{\text{get:}\mathsf{unit} \rightarrow \mathsf{int},\ \text{inc:}\mathsf{unit} \rightarrow \mathsf{unit},\ \text{reset:}\mathsf{unit} \rightarrow \mathsf{unit}\}$.



Reusing Method Code (Simple Classes)

```
class Counter
  { protected int p;
     Counter() { this.p=0; }
     int get () { return this.p; }
     void inc () { this.p++ ; }
     };


class ResetCounter
  extends Counter
  { void reset () {this.p=0;}
     };
```

8.1 Exercises

Exercise 36 ★*For each of the following, either give a type derivation or explain why it is untypable.*

1. $\{\} \vdash \{\text{p:}\{\text{p:}\{\text{p:}\{\text{p} = 3\}\}\}\}\text{:}\{\text{p:}\{\}\}$

2. $\{\} \vdash$ **fn** $\mathrm{x}{:}\{\mathrm{p}{:}\mathsf{bool}, \mathrm{q}{:}\{\mathrm{p}{:}\mathsf{int}, \mathrm{q}{:}\mathsf{bool}\}\} \Rightarrow \#\mathrm{q}\ \#\mathrm{p}\ \mathrm{x} : ?$

3. $\{\} \vdash$ **fn** $\mathrm{f}{:}\{\mathrm{p}{:}\mathsf{int}\} \to \mathsf{int} \Rightarrow (\mathrm{f}\ \{\mathrm{q} = 3\}) + (\mathrm{f}\ \{\mathrm{p} = 4\}) : ?$

4. $\{\} \vdash$ **fn** $\mathrm{f}{:}\{\mathrm{p}{:}\mathsf{int}\} \to \mathsf{int} \Rightarrow (\mathrm{f}\ \{\mathrm{q} = 3, \mathrm{p} = 2\}) + (\mathrm{f}\ \{\mathrm{p} = 4\}) : ?$

Exercise 37 ★*For each of the two bogus T* ref *subtype rules on Page 8, give an example program that is typable with that rule but gets stuck at runtime.*

Exercise 38 ★★*What should the subtype rules for sums $T + T'$ be?*

Exercise 39 ★★*...and for* **let** *and* **let rec** *?*

9 Low-level semantics

Low-level semantics

Can usefully apply semantics not just to high-level languages but to

- Intermediate Languages (e.g. Java Bytecode, MS IL, C−−)
- Assembly languages (esp. for use as a compilation target)
- C-like languages (cf. Cyclone)

By making these type-safe we can make more robust systems.

(see separate handout)

10 Epilogue

Epilogue

Lecture Feedback

Please do fill in the lecture feedback form – we need to know how the course could be improved / what should stay the same.

My impression...

Internship

Interested in Semantics, can do system hacking and ML programming?

Looking for an intern over the summer, to work on

- Acute – our distributed programming language research; and/or
- Network semantics – precise HOL modelling of TCP/UDP/Sockets.

(see the Marshalling and Timing UDP papers on my www).

Probably for 2.5 months, maintenance grant of 2K.

To apply: CV, including contact data for two referees, and letter explaining briefly why you want the post. In pdf, filename `FirstLast.pdf`. Ideally by 3rd March, or if not by 15th.

Good language design?

Need:

- precise definition of what the language is (so can communicate among the designers)
- technical properties (determinacy, decidability of type checking, etc.)
- pragmatic properties (usability in-the-large, implementability)

What can *you* use semantics for?

1. to understand a particular language - what you can depend on as a programmer; what you must provide as a compiler writer
2. as a tool for language design:
   (a) for expressing design choices, understanding language features and how they interact.
   (b) for proving properties of a language, eg type safety, decidability of type inference.
3. as a foundation for proving properties of particular programs

The End

Global Semantics

(op +) $\langle n_1 + n_2, s, M\rangle \longrightarrow \langle n, s, M\rangle$ if $n = n_1 + n_2$

(op $\geq$) $\langle n_1 \geq n_2, s, M\rangle \longrightarrow \langle b, s, M\rangle$ if $b = (n_1 \geq n_2)$

(op1) $$\frac{\langle e_1, s, M\rangle \longrightarrow \langle e_1', s', M'\rangle}{\langle e_1 \;\; op \;\; e_2, s, M\rangle \longrightarrow \langle e_1' \;\; op \;\; e_2, s', M'\rangle}$$

(op2) $$\frac{\langle e_2, s, M\rangle \longrightarrow \langle e_2', s', M'\rangle}{\langle v \;\; op \;\; e_2, s, M\rangle \longrightarrow \langle v \;\; op \;\; e_2', s', M'\rangle}$$

(deref) $\langle !\ell, s, M\rangle \longrightarrow \langle n, s, M\rangle$ if $\ell \in \mathrm{dom}(s)$ and $s(\ell) = n$

(assign1) $\langle \ell := n, s, M\rangle \longrightarrow \langle \textbf{skip}, s + \{\ell \mapsto n\}, M\rangle$ if $\ell \in \mathrm{dom}(s)$

(assign2) $$\frac{\langle e, s, M\rangle \longrightarrow \langle e', s', M'\rangle}{\langle \ell := e, s, M\rangle \longrightarrow \langle \ell := e', s', M'\rangle}$$

(seq1) $\langle \textbf{skip}; e_2, s, M\rangle \longrightarrow \langle e_2, s, M\rangle$

(seq2) $$\frac{\langle e_1, s, M\rangle \longrightarrow \langle e_1', s', M'\rangle}{\langle e_1; e_2, s, M\rangle \longrightarrow \langle e_1'; e_2, s', M'\rangle}$$

(if1) $\langle \textbf{if} \;\; \textbf{true} \;\; \textbf{then} \;\; e_2 \;\; \textbf{else} \;\; e_3, s, M\rangle \longrightarrow \langle e_2, s, M\rangle$

(if2) $\langle \textbf{if} \;\; \textbf{false} \;\; \textbf{then} \;\; e_2 \;\; \textbf{else} \;\; e_3, s, M\rangle \longrightarrow \langle e_3, s, M\rangle$

(if3) $$\frac{\langle e_1, s, M\rangle \longrightarrow \langle e_1', s', M'\rangle}{\langle \textbf{if} \;\; e_1 \;\; \textbf{then} \;\; e_2 \;\; \textbf{else} \;\; e_3, s, M\rangle \longrightarrow \langle \textbf{if} \;\; e_1' \;\; \textbf{then} \;\; e_2 \;\; \textbf{else} \;\; e_3, s', M'\rangle}$$

(while)
$\langle \textbf{while} \;\; e_1 \;\; \textbf{do} \;\; e_2, s, M\rangle \longrightarrow \langle \textbf{if} \;\; e_1 \;\; \textbf{then} \; (e_2; \textbf{while} \;\; e_1 \;\; \textbf{do} \;\; e_2) \; \textbf{else} \;\; \textbf{skip}, s, M\rangle$

(parallel1) $$\frac{\langle e_1, s, M\rangle \longrightarrow \langle e_1', s', M'\rangle}{\langle e_1 \,|\, e_2, s, M\rangle \longrightarrow \langle e_1' \,|\, e_2, s', M'\rangle}$$

(parallel2) $$\frac{\langle e_2, s, M\rangle \longrightarrow \langle e_2', s', M'\rangle}{\langle e_1 \,|\, e_2, s, M\rangle \longrightarrow \langle e_1 \,|\, e_2', s', M'\rangle}$$

(lock) $\langle \textbf{lock} \;\; m, s, M\rangle \longrightarrow \langle (), s, M + \{m \mapsto \textbf{true}\}\rangle$ if $\neg M(m)$

(unlock) $\langle \textbf{unlock} \;\; m, s, M\rangle \longrightarrow \langle (), s, M + \{m \mapsto \textbf{false}\}\rangle$

Thread-Local Semantics

(t-op +) $n_1 + n_2 \xrightarrow{\tau} n$ if $n = n_1 + n_2$

(t-op $\geq$) $n_1 \geq n_2 \xrightarrow{\tau} b$ if $b = (n_1 \geq n_2)$

(t-op1) $$\frac{e_1 \xrightarrow{a} e_1'}{e_1 \;\; op \;\; e_2 \xrightarrow{a} e_1' \;\; op \;\; e_2}$$

(t-op2) $$\frac{e_2 \xrightarrow{a} e_2'}{v \;\; op \;\; e_2 \xrightarrow{a} v \;\; op \;\; e_2'}$$

(t-deref) $!\ell \xrightarrow{!\ell = n} n$

(t-assign1) $\ell := n \xrightarrow{\ell := n} \textbf{skip}$

(t-assign2) $$\frac{e \xrightarrow{a} e'}{\ell := e \xrightarrow{a} \ell := e'}$$

(t-seq1) $\textbf{skip}; e_2 \xrightarrow{\tau} e_2$

(t-seq2) $$\frac{e_1 \xrightarrow{a} e_1'}{e_1; e_2 \xrightarrow{a} e_1'; e_2}$$

(t-if1) $\textbf{if} \;\; \textbf{true} \;\; \textbf{then} \;\; e_2 \;\; \textbf{else} \;\; e_3 \xrightarrow{\tau} e_2$

(t-if2) $\textbf{if} \;\; \textbf{false} \;\; \textbf{then} \;\; e_2 \;\; \textbf{else} \;\; e_3 \xrightarrow{\tau} e_3$

(t-if3) $$\frac{e_1 \xrightarrow{a} e_1'}{\textbf{if} \;\; e_1 \;\; \textbf{then} \;\; e_2 \;\; \textbf{else} \;\; e_3 \xrightarrow{a} \textbf{if} \;\; e_1' \;\; \textbf{then} \;\; e_2 \;\; \textbf{else} \;\; e_3}$$

(t-while)
$\textbf{while} \;\; e_1 \;\; \textbf{do} \;\; e_2 \xrightarrow{\tau} \textbf{if} \;\; e_1 \;\; \textbf{then} \; (e_2; \textbf{while} \;\; e_1 \;\; \textbf{do} \;\; e_2) \; \textbf{else} \;\; \textbf{skip}$

(t-parallel1) $$\frac{e_1 \xrightarrow{a} e_1'}{e_1 \,|\, e_2 \xrightarrow{a} e_1' \,|\, e_2}$$

(t-parallel2) $$\frac{e_2 \xrightarrow{a} e_2'}{e_1 \,|\, e_2 \xrightarrow{a} e_1 \,|\, e_2'}$$

(t-lock) $\textbf{lock} \;\; m \xrightarrow{\textsf{lock} \; m} ()$

(t-unlock) $\textbf{unlock} \;\; m \xrightarrow{\textsf{unlock} \; m} ()$

(c-tau) $$\frac{e \xrightarrow{\tau} e'}{\langle e, s, M\rangle \longrightarrow \langle e', s, M\rangle}$$

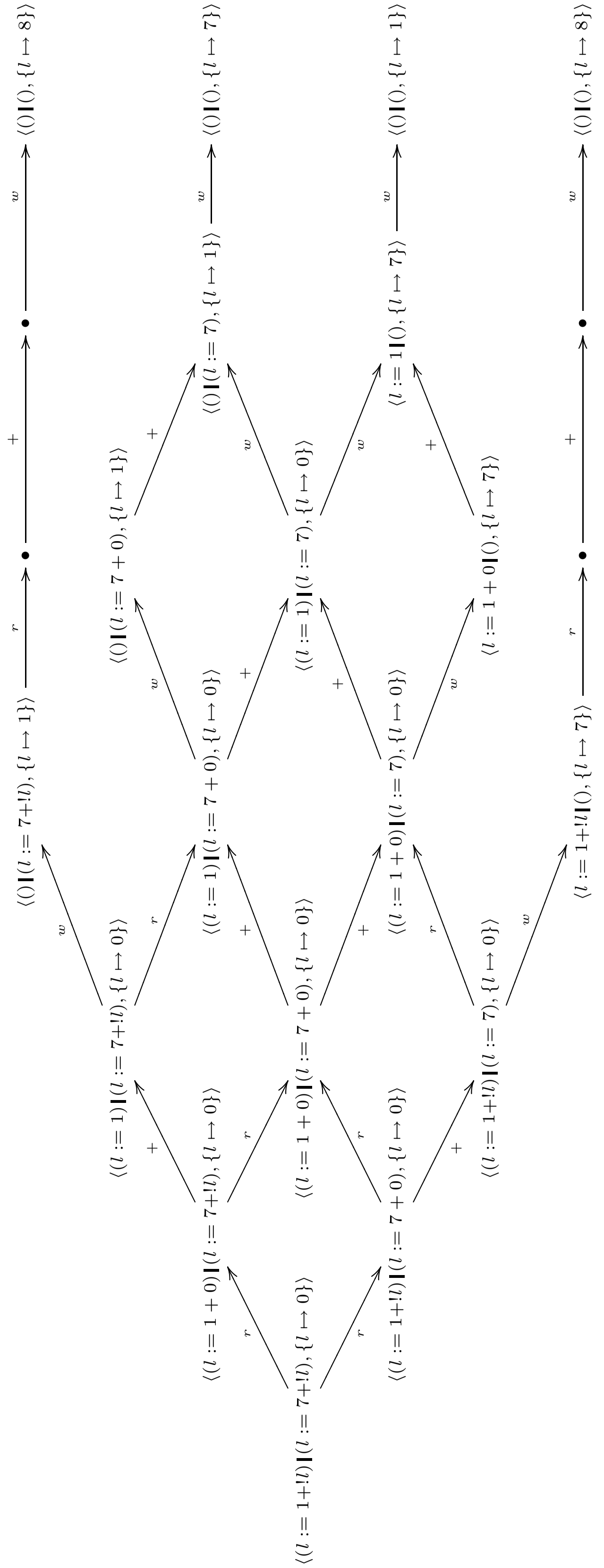
(c-assign) $$\frac{e \xrightarrow{\ell := n} e' \quad \ell \in \mathrm{dom}(s)}{\langle e, s, M\rangle \longrightarrow \langle e', s + \{\ell \mapsto n\}, M\rangle}$$

(c-lock) $$\frac{e \xrightarrow{\textsf{lock} \; m} e' \quad \neg M(m)}{\langle e, s, M\rangle \longrightarrow \langle e', s, M + \{m \mapsto \textbf{true}\}\rangle}$$

(c-deref) $$\frac{e \xrightarrow{!\ell = n} e' \quad \ell \in \mathrm{dom}(s) \wedge s(\ell) = n}{\langle e, s, M\rangle \longrightarrow \langle e', s, M\rangle}$$

(c-unlock) $$\frac{e \xrightarrow{\textsf{unlock} \; m} e'}{\langle e, s, M\rangle \longrightarrow \langle e', s, M + \{m \mapsto \textbf{false}\}\rangle}$$

The behaviour of $(l := 1+!l) \| (l := 7+!l)$ for the initial store $\{l \mapsto 0\}$:


⟨(l := 1+!l)‖(l := 7+!l), {l ↦ 0}⟩
⟨(l := 1 + 0)‖(l := 7+!l), {l ↦ 0}⟩
⟨(l := 1+!l)‖(l := 7 + 0), {l ↦ 0}⟩
⟨(l := 1)‖(l := 7+!l), {l ↦ 0}⟩
⟨(l := 1 + 0)‖(l := 7 + 0), {l ↦ 0}⟩
⟨(l := 1+!l)‖(l := 7), {l ↦ 0}⟩
⟨()‖(l := 7+!l), {l ↦ 1}⟩
⟨(l := 1)‖(l := 7 + 0), {l ↦ 0}⟩
⟨(l := 1 + 0)‖(l := 7), {l ↦ 0}⟩
⟨l := 1+!l‖(), {l ↦ 7}⟩
⟨()‖(l := 7 + 0), {l ↦ 1}⟩
⟨(l := 1)‖(l := 7), {l ↦ 0}⟩
⟨l := 1 + 0‖(), {l ↦ 7}⟩
⟨()‖(l := 7), {l ↦ 1}⟩
⟨l := 1‖(), {l ↦ 7}⟩
⟨()‖(), {l ↦ 8}⟩
⟨()‖(), {l ↦ 7}⟩
⟨()‖(), {l ↦ 1}⟩
⟨()‖(), {l ↦ 8}⟩
r
r
+
r
r
+
w
r
+
+
r
w
w
+
+
w
w
+
w
w
r
+
w
r
+
w

A 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 §A.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.

A.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 = (let x:int = 7+2 in x+x)
    ```

    Take care with choosing variable names.
    (c) Doing *equational reasoning*, e.g.

    ```
    e = e1   by ...
      = e2   by ...
      = e3   as ...
    ```

Here the `e` might be any mathematical object — arithmetic expressions, or expressions of some grammar, or formulae. Some handy equations over formulae are given in §A.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 §A.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 Φ 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 Φ 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.*

A.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.

A.2.1 Meet the Connectives

Here are the logical connectives and a very brief decription of what each means.

$P \wedge Q$	P and Q are both true
$P \vee Q$	P is true, or Q is true, or both are true
$\neg P$	P is not true (P is false)
$P \Rightarrow Q$	if P is true then Q is true
$P \Leftrightarrow Q$	P is true exactly when Q is true
$\forall x \in S.P(x)$	for all x in S, P is true of x
$\exists x \in S.P(x)$	there exists an x in S such that P holds of x

A.2.2 Equivalences

These are formulas that mean the same thing, and this is indicated by a $\simeq$ between them. The fact that they are equivalent to each other is justified by the truth tables of the connectives.

definition of $\Rightarrow$	$P \Rightarrow Q$	$\simeq$	$\neg P \vee Q$
definition of $\Leftrightarrow$	$P \Leftrightarrow Q$	$\simeq$	$(P \Rightarrow Q) \wedge (Q \Rightarrow P)$
definition of $\neg$	$\neg P$	$\simeq$	$P \Rightarrow \text{false}$
de Morgan's Laws	$\neg(P \wedge Q)$	$\simeq$	$\neg P \vee \neg Q$
	$\neg(P \vee Q)$	$\simeq$	$\neg P \wedge \neg Q$
extension to quantifiers	$\neg(\forall x.P(x))$	$\simeq$	$\exists x.\neg P(x)$
	$\neg(\exists x.P(x))$	$\simeq$	$\forall x.\neg P(x)$
distributive laws	$P \vee (Q \wedge R)$	$\simeq$	$(P \vee Q) \wedge (P \vee R)$
	$P \wedge (Q \vee R)$	$\simeq$	$(P \wedge Q) \vee (P \wedge R)$
coalescing quantifiers	$(\forall x.P(x)) \wedge (\forall x.Q(x))$	$\simeq$	$\forall x.(P(x) \wedge Q(x))$
	$(\exists x.P(x)) \vee (\exists x.Q(x))$	$\simeq$	$\exists x.(P(x) \vee Q(x))$
these ones apply if	$(\forall x.P(x)) \wedge Q$	$\simeq$	$(\forall x.P(x) \wedge Q)$
x is not free in Q	$(\forall x.P(x)) \vee Q$	$\simeq$	$(\forall x.P(x) \vee Q)$
	$(\exists x.P(x)) \wedge Q$	$\simeq$	$(\exists x.P(x) \wedge Q)$
	$(\exists x.P(x)) \vee Q$	$\simeq$	$(\exists x.P(x) \vee Q)$

A.2.3 How to Prove a Formula

For each of the logical connectives, I'll explain how to handle them.

$\boxed{\forall x \in S.P(x)}$ This means "For all x in S, P is true of x." Such a formula is called a universally quantified formula. The goal is to prove that the property P, which has some xs somewhere in it, is true no matter what value in S x takes on. Often the "$\in S$" is left out. For example, in a discussion of lists, you might be asked to prove $\forall l.\texttt{length}\, l > 0 \Rightarrow \exists x.\, \texttt{member}(x, l)$. Obviously, l is a list, even if it isn't explicitly stated as such.

There are several choices as to how to prove a formula beginning with $\forall x$. The standard thing to do is to just prove $P(x)$, not assuming anything about x. Thus, in doing the proof you sort of just mentally strip off the $\forall x$. What you would write when doing this is "Let x be any S". However, there are some subtleties—if you're already using an x for something else, you can't use the same x, because then you *would* be assuming something about x, namely that it equals the x you're already using. In this case, you need to use alpha-conversion[1] to change the formula you want to prove to $\forall y \in S.P(y)$, where y is some variable you're not already using, and then prove $P(y)$. What you could write in this case is "Since x is already in use, we'll prove the property of y".

An alternative is induction, if S is a set that is defined with a structural definition. Many objects you're likely to be proving properties of are defined with a structural definition. This includes natural numbers, lists, trees, and terms of a computer language. Sometimes you can use induction over the natural numbers to prove things about other objects, such as graphs, by inducting over the number of nodes (or edges) in a graph.

You use induction when you see that during the course of the proof you would need to use the property P for the subparts of x in order to prove it for x. This usually ends up being the case if P involves functions defined recursively (i.e., the return value for the function depends on the function value on the subparts of the argument).

A special case of induction is case analysis. It's basically induction where you don't use the inductive hypothesis: you just prove the property for each possible form that x could have. Case analysis can be used to prove the theorem about lists above.

A final possibility (which you can use for all formulas, not just for universally quantified ones) is to assume the contrary, and then derive a contradiction.

$\boxed{\exists x \in S.P(x)}$ This says "There exists an x in S such that P holds of x." Such a formula is called an existentially quantified formula. The main way to prove this is to figure out what x has to be (that is, to find a concrete representation of it), and then prove that P holds of that value. Sometimes you can't give a completely specified value, since the value you pick for x has to depend on the values of other things you have floating around. For example, say you want to prove

$$\forall x, y \in \Re . x < y \wedge \sin x < 0 \wedge \sin y > 0 \Rightarrow \exists z . x < z \wedge z < y \wedge \sin z = 0$$

where $\Re$ is the set of real numbers. By the time you get to dealing with the $\exists z . x < z \wedge z < y \wedge \sin z = 0$, you will have already assumed that x and y were any real numbers. Thus the value you choose for z has to depend on whatever x and y are.

An alternative way to prove $\exists x \in S.P(x)$ is, of course, to assume that no such x exists, and derive a contradiction.

To summarize what I've gone over so far: to *prove* a universally quantified formula, you must prove it for a generic variable, one that you haven't used before. To prove an existentially quantified formula, you get to choose a value that you want to prove the property of.

$\boxed{P \Rightarrow Q}$ This says "If P is true, then Q is true". Such a formula is called an implication, and it is often pronounced "P implies Q". The part before the $\Rightarrow$ sign (here P) is called the antecedent, and the part after the $\Rightarrow$ sign (here Q) is called the consequent. $P \Rightarrow Q$ is equivalent to $\neg P \vee Q$, and so if P is false, or if Q is true, then $P \Rightarrow Q$ is true.

The standard way to prove this is to assume P, then use it to help you prove Q. Note that I said that you will be *using* P. Thus you will need to follow the rules in Section A.2.4 to deal with the logical connectives in P.

Other ways to prove $P \Rightarrow Q$ involve the fact that it is equivalent to $\neg P \vee Q$. Thus, you can prove $\neg P$ without bothering with Q, or you can just prove Q without bothering with P. To reason by contradiction you assume that P is true and that Q is not true, and derive a contradiction.

Another alternative is to prove the contrapositive: $\neg Q \Rightarrow \neg P$, which is equivalent to it.

$\boxed{P \Leftrightarrow Q}$ This says "P is true if and only if Q is true". The phrase "if and only if" is usually abbreviated "iff". Basically, this means that P and Q are either both true, or both false.

Iff is usually used in two main ways: one is where the equivalence is due to one formula being a definition of another. For example, $A \subseteq B \Leftrightarrow (\forall x.\ x \in A \Rightarrow x \in B)$ is the standard definition of subset. For these

[1] Alpha-equivalence says that the name of a bound variable doesn't matter, so you can change it at will (this is called alpha-conversion). You'll get to know the exact meaning of this soon enough so I won't explain this here.

iff statements, you don't have to prove them. The other use of iff is to state the equivalence of two different things. For example, you could define an SML function `fact`:

```
fun fact 0 = 1
  | fact n = n * fact (n - 1)
```

Since in SML whole numbers are integers (both positive and negative) you may be asked to prove: $\texttt{fact}\ x$ terminates $\Leftrightarrow x \geq 0$. The standard way to do this is us the equivalence $P \Leftrightarrow Q$ is equivalent to $P \Rightarrow Q \wedge Q \Rightarrow P$. And so you'd prove that ($\texttt{fact}\ x$ terminates $\Rightarrow x \geq 0) \wedge (x \geq 0 \Rightarrow \texttt{fact}\ x$ terminates).

$\boxed{\neg P}$ This says "P is not true". It is equivalent to $P \Rightarrow$ 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.

$\boxed{P \wedge Q}$ This says "P is true and Q is true". Such a formula is called a conjunction. To prove this, you have to prove P, and you have to prove Q.

$\boxed{P \vee Q}$ This says "P is true or Q is true". This is *inclusive* or: if P and Q are both true, then $P \vee Q$ is still true. Such a formula is called a disjunction. To prove this, you can prove P or you can prove Q. You have to choose which one to prove. For example, if you need to prove $(5 \bmod 2 = 0) \vee (5 \bmod 2 = 1)$, then you'll choose the second one and prove that.

However, as with existentials, the choice of which one to prove will often depend on the values of other things, like universally quantified variables. For example, when you are studying the theory of programming languages (you will get a bit of this in Semantics), you might be asked to prove

$$\begin{aligned}\forall P \in ML.\quad & P \textit{ is properly typed} \Rightarrow \\ & (\textit{the evaluation of } P \textit{ runs forever}) \vee (P \textit{ evaluates to a value})\end{aligned}$$

where ML is the set of all ML programs. You don't know in advance which of these will be the case, since some programs do run forever, and some do evaluate to a value. Generally, the best way to prove the disjunction in this case (when you don't know in advance which will hold) is to use the equivalence with implication. For example, you can use the fact that $P \vee Q$ is equivalent to $\neg P \Rightarrow Q$, then assume $\neg P$, then use this to prove Q. For example, your best bet to proving this programming languages theorem is to assume that the evaluation of P doesn't run forever, and use this to prove that P evaluates to a value.

A.2.4 How to Use a Formula

You often end up using a formula to prove other formulas. You can use a formula if someone has already proved that it's true, or you are assuming it because it was in an implication, namely, the A in $A \Rightarrow B$. For each logical connective, I'll tell you how to use it.

$\boxed{\forall x \in S.P(x)}$ This formula says that something is true of *all* elements of S. Thus, when you use it, you can pick any value at all to use instead of x (call it v), and then you can use $P(v)$.

$\boxed{\exists x \in S.P(x)}$ This formula says that there is some x that satisfies P. However, you do not know what it is, so you can not assume anything about it. The usual approach 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.

$\boxed{\neg P}$ Usually, the main use of this formula is to prove the negation of something else. An example is the use of reduction to prove the unsolvability of various problems in the Computation Theory (you'll learn all about this in Lent term). You want to prove $\neg Q$, where Q states that a certain problem (Problem 1) is decidable (in other words, you want to prove that Problem 1 is not decidable). You know $\neg P$, where P states that another problem (Problem 2) is decidable (i.e. $\neg P$ says that Problem 2 is not decidable). What you do basically is this. You first prove $Q \Rightarrow P$, which says that if Problem 1 is decidable, then so is Problem 2. Since $Q \Rightarrow P \simeq \neg P \Rightarrow \neg Q$, you have now proved $\neg P \Rightarrow \neg Q$. You already know $\neg P$, so you use modus ponens[2] to get that $\neg Q$.

$\boxed{P \Rightarrow Q}$ 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.

$\boxed{P \Leftrightarrow Q}$ The main use of this is to replace an occurrence of P in a formula with Q, and vise versa.

$\boxed{P \wedge Q}$ Here you can use both P and Q. Note, you're not *required* to use both of them, but they are both true and are waiting to be used by you if you need them.

$\boxed{P \vee Q}$ Here, you know that one of P or Q is true, but you do not know which one. To use this to prove something else, you have to do a split: first you prove the thing using P, then you prove it using Q.

Note that in each of the above, there is again a difference in the way you use a formula, verses the way you prove it. They are in a way almost opposites. For example, in proving $P \wedge Q$, you have to prove both P and Q, but when you are using the formula, you don't have to use both of them.

A.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.

A.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.

$$\begin{array}{l}\forall L \in \mathit{RegularLanguages}.\\ \ \exists n.\ \forall z \in L.\ |z| \geq n \Rightarrow\\ \ \ \exists u\,v\,w.\ z = uvw \wedge |uv| \leq n \wedge |v| \geq 1 \wedge\\ \ \ \ \forall i \geq 0.\ uv^iw \in L\end{array}$$

Complicated, eh? Well, let's prove it, using the facts that Hopcroft and Ullman have established in the chapters previous to the one wih the PL. I'll give the proof and put in square brackets comments about what I'm doing.

[2] Modus ponens says that if $A \Rightarrow B$ and A are both true, then B is true.

Let L be any regular language. [Here I'm dealing with the $\forall L \in \mathit{RegularLanguages}$ by stating that I'm not assuming anything about L.] Let M be a minimal-state deterministic 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 $a_1 a_2 \ldots a_m$, where $m \geq n$. Consider the states that M is in during the processing of the first n symbols of z, $a_1 a_2 \ldots a_n$. 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 $u = a_1 a_2 \ldots a_j$. 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 $\wedge$ by proving each of them.]

Now, let i be a natural number (i.e. ≥ 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.

A.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 $L = \{0^{i^2} | i \text{ is an integer}, i \geq 1\}$ is not regular. Note that L consists of all strings of 0's whose length is a perfect square. I will *use* the PL. I want to prove that L is not regular. I'll assume the negation (i.e., that L is regular) and derive a contradiction. So here we go. Remember that what I'm emphasizing here is not the finite automata stuff itself, but how to use a complicated theorem to prove something else.

Assume L is regular. We will use the PL to get a contradiction. Since L is regular, the PL applies to it. [We note that we're using the $\forall$ part of the PL for this particular L.] Let n be as described in the PL. [This takes care of using the $\exists n$. Note that we *are not* assuming anything about its actual value, just that it's a natural number.] Let $z = 0^{n^2}$. [Since the PL says that something is true of *all* zs, we can choose the one we want to use it for.] So by the PL there exist u, v, and w such that $z = uvw$, $|uv| \leq n$, $|v| \geq 1$. [Note that we don't assume anything about what the u, v, and w actually are; the only thing we know about them is what the PL tells us about them. This is where people trying to use the PL usually screw up.] The PL then says that for any i, then $uv^i w \in L$. Well, then $uv^2 w \in L$. [This is using the $\forall i \geq 0$ bit.] However, $n^2 < |uv^2 w| \leq n^2 + n$, since $1 \leq |v| \leq n$. But $n^2 + n < (n+1)^2$. Thus $|uv^2 w|$ lies properly between n^2 and $(n+1)^2$ and is thus not a perfect square. Thus $uv^2 w$ is not in L. This is a contradiction. Thus our assumption (that L was regular) was incorrect. Thus L is not a regular language.

A.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 Γ and Δ are sets of formulas.[3] Technically, this means that

$$A_1 \wedge A_2 \wedge \ldots A_n \Rightarrow B_1 \vee B_2 \vee \ldots B_m \tag{1}$$

where $A_1, A_2, \ldots A_n$ are the formulas in Γ , and $B_1, B_2, \ldots B_n$ are the formulas in Δ. Less formally, this means "using the formulas in Γ we can prove that one of the formula in Δ is true." This is just the intuition I described above about using vs proving formulas, except that I only talked about proving that one formula is true, rather than proving that one of several formulas is true. In order to handle the $\vee$ connective, there can be any number of formulas on the right hand side of the $\vdash$.

[3] In your Logic and Proof notes, the symbol that divides Γ from Δ is $\Rightarrow$. However, that conflicts with the use of $\Rightarrow$ as implication. Thus I will use $\vdash$. You will see something similar in Semantics, where it separates assumptions (of the types of variables) from something that they allow you to prove.

For each logic connective,[4] I'll give the rules for it, and explain how it relates to the intuitive way of using or proving formulas. For each connective there are at least two rules for it: one for the left side of the $\vdash$, and one for the right side. This corresponds to having different ways to treat a formula depending on whether you're using it (for formulas on the left hand side of the $\vdash$) or proving it (for formulas on the right side of the $\vdash$).

It's easiest to understand these rules from the bottom up. The conclusion of the rule (the sequent below the horizontal line) is what we want to prove. The hypotheses of the rule (the sequents above the horizontal line) are how we go about proving it. We'll have to use more rules, adding to the top, to build up the proof of the hypothesis, but this at least tells us how to get going.

You can stop when the formula you have on the top is a *basic sequent.* This is $\Gamma \vdash \Delta$ where there's at least one formula (say P) that's in both Γ and Δ. You can see why this is the basic true formula: it says that if P and the other formulas in Γ are true, then P or one of the other formula in Δ is true.

In building proofs from these rules, there are several ways that you end up with formulas to the left of the $\vdash$, where you can use them rather than proving them. One is that you've already proved it before. This is shown with the cut rule:

$$\frac{\Gamma \vdash \Delta, P \quad P, \Gamma \vdash \Delta}{\Gamma \vdash \Delta}\ (cut)$$

The Δ, P in the first sequent in the hypotheses means that to the right of the $\vdash$ we have the set consisting of the formula P plus all the formulas in Δ, i.e., if all formulas in Γ are true, then P or one of the formulas in Δ is true. Similarly P, Γ to the left of the $\vdash$ in the second sequent means the set consisting of the formula P plus all the formulas in Γ.

We read this rule from the bottom up to make sense of it. Say we want to prove one of the formulas in Δ from the formulas in Γ, 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 Δ is shown in the right hand hypothesis. We continue to build the proof up from there, using P.

Some other ways of getting formulas to the left of the $\vdash$ are shown in the rules $(\neg r)$ and $(\Rightarrow r)$ below.

$\boxed{\forall x \in S.P(x)}$ The two rules for universally quantified formulas are:

$$\frac{P(v), \Gamma \vdash \Delta}{\forall x.P(x), \Gamma \vdash \Delta}\ (\forall l) \qquad \frac{\Gamma \vdash \Delta, P(x)}{\Gamma \vdash \Delta, \forall x.P(x)}\ (\forall r)$$

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 Γ) to prove something (Δ). 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 Γ or Δ. 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 Γ or Δ, 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.

[4] I won't mention iff here: as $P \Leftrightarrow Q$ is equivalent to $P \Rightarrow Q \wedge Q \Rightarrow P$, we don't need separate rules for it.

$\boxed{\exists x \in S.P(x)}$ The two rules for existentially quantified formulas are:

$$\frac{P(x), \Gamma \vdash \Delta}{\exists x.P(x), \Gamma \vdash \Delta}\ (\exists l) \qquad \frac{\Gamma \vdash \Delta, P(v)}{\Gamma \vdash \Delta, \exists x.P(x)}\ (\exists r)$$

In the $(\exists l)$ rule, x must not be free in the conclusion.

In $(\exists l)$, we are using $\exists x.P(x)$. Thus we cannot assume anything about the value that the formula says exists, so we just use it as x in the hypothesis. The side condition about x not being free in the conclusions comes from the requirement not to assume anything about x (since we don't know what it is). If x isn't free in the conclusion, then it's not free in Γ or Δ. If it were free in Γ or Δ, 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 Γ or Δ, since you can set it to anything you want.

$\boxed{\neg P}$ The rules for negation are:

$$\frac{\Gamma \vdash \Delta, P}{\neg P, \Gamma \vdash \Delta}\ (\neg l) \qquad \frac{P, \Gamma \vdash \Delta}{\Gamma \vdash \Delta, \neg P}\ (\neg r)$$

Let's start with the right rule first. I said that the way to prove $\neg P$ is to assume P and derive a contradiction. If Δ 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 Γ are inconsistent (that means, they cannot all be true at the same time). This means that you have derived a contradiction. So if Δ 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 Δ 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.

$\boxed{P \Rightarrow Q}$ The two rules for implication are:

$$\frac{\Gamma \vdash \Delta, P \quad Q, \Gamma \vdash \Delta}{P \Rightarrow Q, \Gamma \vdash \Delta}\ (\Rightarrow l) \qquad \frac{P, \Gamma \vdash \Delta, Q}{\Gamma \vdash \Delta, P \Rightarrow Q}\ (\Rightarrow r)$$

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.

$\boxed{P \wedge Q}$ The rules for conjunction are:

$$\frac{P, Q, \Gamma \vdash \Delta}{P \wedge Q, \Gamma \vdash \Delta}\ (\wedge l) \qquad \frac{\Gamma \vdash \Delta, P \quad \Gamma \vdash \Delta, Q}{\Gamma \vdash \Delta, P \wedge Q}\ (\wedge r)$$

Both of these rules are easily explained by the intuition above. The left rule $(\wedge l)$ says that when you use $P \wedge Q$, you can use P and Q. The right rule says that to prove $P \wedge Q$ you must prove P, and you must prove Q. You may wonder why we need separate hypotheses for the two different proofs. We can't just put P, Q to the right of the $\vdash$ in a single hypothesis, because that would mean that we're proving one of the other of them (see the meaning of the sequent given in Equation 1). So we need separate hypotheses to make sure that each of P and Q has actually been proved.

$\boxed{P \vee Q}$ The rules for disjunction are:

$$\frac{P, \Gamma \vdash \Delta \quad Q, \Gamma \vdash \Delta}{P \vee Q, \Gamma \vdash \Delta}\ (\vee l) \qquad \frac{\Gamma \vdash \Delta, P, Q}{\Gamma \vdash \Delta, P \vee Q}\ (\vee r)$$

These are also easily understood by the intuitive explanations above. The left rule says that to prove something (namely, one of the formulas in Δ) using $P \vee Q$, you need to prove it using P, then prove it using Q. The right rule says that in order to prove $P \vee Q$, you can prove one or the other. The hypothesis says that you can prove one or the other, because in order to show a sequent $\Gamma \vdash \Delta$ true, you only need to show that *one* of the formulas in Δ is true.