

Q

Lecture Notes on

# *Types*

for Part II of the Computer Science Tripos

Prof. Andrew M. Pitts  
University of Cambridge  
Computer Laboratory

First edition 1997.

Revised 1999, 2000, 2001, 2002, 2004.

# Contents

<b>Learning Guide</b>	<b>ii</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 ML Polymorphism</b>	<b>7</b>
2.1 An ML type system . . . . .	8
2.2 Examples of type inference, by hand . . . . .	17
2.3 Principal type schemes . . . . .	21
2.4 A type inference algorithm . . . . .	23
2.5 Exercises . . . . .	26
<b>3 Polymorphic Reference Types</b>	<b>29</b>
3.1 The problem . . . . .	29
3.2 Restoring type soundness . . . . .	34
3.3 Exercises . . . . .	36
<b>4 Polymorphic Lambda Calculus</b>	<b>37</b>
4.1 From type schemes to polymorphic types . . . . .	37
4.2 The PLC type system . . . . .	40
4.3 PLC type inference . . . . .	48
4.4 Datatypes in PLC . . . . .	50
4.5 Exercises . . . . .	57
<b>5 Further Topics</b>	<b>61</b>
5.1 Curry-Howard correspondence . . . . .	61
5.2 Dependent types . . . . .	65
5.3 Current areas of research . . . . .	66
<b>References</b>	<b>71</b>
<b>Lectures Appraisal Form</b>	<b>73</b>

## Learning Guide

These notes and slides are designed to accompany eight lectures on type systems for Part II of the Cambridge University Computer Science Tripos. The aim of this course is to show by example how type systems for programming languages can be defined and their properties developed, using techniques that were introduced in the Part IB course on *Semantics of Programming Languages*. So that course is a prerequisite (as, to a lesser extent, is the Part IB course on *Foundations of Functional Programming*.) We apply these techniques to a few selected topics centred mainly around the notion of “polymorphism” (or “generics” as it is known in the Java and C# communities).

Formal systems and mathematical proof play an important role in this subject—a fact which is reflected in the nature of the material presented here and in the kind of questions set on it in the Tripos. As well as learning some specific facts about the ML type system and the polymorphic lambda calculus, at the end of the course you should:

- appreciate how type systems can be used to constrain or describe the dynamic behaviour of programs
- be able to use a rule-based specification of a type system to infer typings and to establish type soundness results
- appreciate the expressive power of the polymorphic lambda calculus.

### Tripos questions and exercises

A list of past Tripos questions back to 1993 that are relevant to the current course is available at [www.cl.cam.ac.uk/tripos/t-Types.html](http://www.cl.cam.ac.uk/tripos/t-Types.html). (Watch out for a misprint in 1993 paper 9, question 11: in the notation of these notes,  $\exists\alpha.\sigma$  should be defined to be  $\forall\beta(\forall\alpha(\sigma \rightarrow \beta) \rightarrow \beta)$ .) In addition there are a few exercises at the end of most sections.

### Recommended reading

The recent graduate-level text by Pierce (2002) covers much of the material presented in these notes (although not always in the same way), plus much else besides. It is highly recommended. The following additional material may be useful:

**Sections 2–3** (Cardelli 1987) introduces the ideas behind ML polymorphism and type-checking. One could also take a look in (Milner, Tofte, Harper, and MacQueen 1997) at the chapter defining the static semantics for the core language, although it does not make light reading! If you want more help understanding the material in Section 3 (Polymorphic Reference Types), try Section 1.1.2.1 (Value Polymorphism) of the *SML'97 Conversion Guide* provided by the SML/NJ implementation of ML. (See the web page for this lecture course for a URL for this document.)

**Section 4** Read (Girard 1989) for an account by one of its creators of the polymorphic lambda calculus (Système F), its relation to proof theory and much else besides.

## **Note!**

The material in these notes has been drawn from several different sources, including those mentioned above and previous versions of this course by the author and by others. Any errors are of course all my own work. Please let me know if you find typos or possible errors: a list of corrections will be available from the course web page (follow links from [www.cl.cam.ac.uk/Teaching/](http://www.cl.cam.ac.uk/Teaching/)), which also contains pointers to some other useful material. A lecture(r) appraisal form is included at the end of the notes. Please take time to fill it in and return it. Alternatively, fill out an electronic version of the form via the URL [www.cl.cam.ac.uk/cgi-bin/lr/login](http://www.cl.cam.ac.uk/cgi-bin/lr/login).

Andrew Pitts  
Andrew.Pitts@cl.cam.ac.uk



# 1 Introduction

*“One of the most helpful concepts in the whole of programming is the notion of type, used to classify the kinds of object which are manipulated. A significant proportion of programming mistakes are detected by an implementation which does type-checking before it runs any program. Types provide a taxonomy which helps people to think and to communicate about programs.”*

R. Milner, “Computing Tomorrow” (CUP, 1996), p264

This short course is about the use of types in programming languages. Types also play an important role in specification languages and in formal logics. Indeed types first arose (in the work of Bertrand Russell (Russell 1903) around 1900) as a way of avoiding certain paradoxes in the logical foundations of mathematics. We will return to the interplay between types in programming languages and types in logic right at the end of the course.

Many programming languages permit, or even require, the use of certain kinds of phrases—types, structures, classes, interfaces, etc—for classifying expressions according to their structure (e.g. “this expression is an array of character strings”) and/or behaviour (e.g. “this function takes an integer argument and returns a list of booleans”). As indicated on Slide 1, a *type system* for a particular language is a formal specification of how such a classification of expressions into types is to be carried out.

The full title of this course is

## **Type Systems for Programming Languages**

What are “type systems” and what are they good for?

“A type system is a tractable syntactic method for proving the absence of certain program behaviours by classifying phrases according to the kinds of values they compute”

B. Pierce, “Types and Programming Languages” (MIT, 2002), p1

It is not an exaggeration to say that to date, type systems are the most important channel by which developments in theoretical computer science get applied in programming languages.

### **Slide 1**

Here are some ways (summarised on Slide 2) in which type systems for programming languages get used:

### Uses of type systems

---

- Detecting errors via *type-checking*, either statically (decidable errors detected before programs are executed) or dynamically (typing errors detected during program execution).
- Abstraction and support for structuring large systems.
- Documentation.
- Efficiency.
- Whole-language safety.

#### Slide 2

**Detecting errors** Experience shows that a significant proportion of programming mistakes (such as trying to multiply an integer by a string) can be detected by an implementation which does *static* type-checking, i.e. which checks for typing errors before it runs any program. Type systems used to implement such checks at compile-time necessarily involve *decidable* properties of program phrases, since otherwise the process of compilation is not guaranteed to terminate. (Recall the notion of (algorithmic) *decidability* from the CST IB ‘Computation Theory’ course.) For example, in a Turing-powerful language (one that can code all partial recursive functions), it is undecidable whether an arbitrary arithmetic expression evaluates to 0 or not; hence static type-checking will not be able to eliminate all “division by zero” errors. Of course the more properties of program phrases a type systems can express the better and the development of the subject is partly a search for greater expressivity; but expressivity is constrained in theory by this decidability requirement, and is constrained in practice by questions of computational feasibility.

**Abstraction and support for structuring large systems** Type information is a crucial part of *interfaces* for modules and classes, allowing the whole to be designed independently of particular implementations of its parts. Type systems form the backbone of various module languages in which modules (“structures”) are assigned types which are interfaces (“signatures”).

**Documentation** Type information in procedure/function declarations and in module/class interfaces are a form of documentation, giving useful hints about intended use and behaviour. Static type-checking ensures that this kind of “formal documentation” keeps in step with changes to the program.

**Efficiency** Typing information be used by a compilers to produce more efficient code. For example the first use of types in computer science (in the 1950s) was to improve the efficiency of numerical calculations in Fortran by distinguishing between integer and real-value expressions. Many static analyses carried out by optimising compilers make use of specialised type systems: an example is the “region inference” used in the ML Kit Compiler to replace much garbage collection in the heap by stack-based memory management (Tofte and Talpin 1997).

### Safety

---

Informal definitions from the literature.

“A safe language is one that protects its own high-level abstractions [no matter what legal program we write in it]”.

“A safe language is completely defined by its programmer’s manual [rather than which compiler we are using]”.

“A safe language may have *trapped* errors [one that can be handled gracefully], but can’t have *untrapped errors* [ones that cause unpredictable crashes]”.

### Slide 3

**Whole-language safety** Slide 3 gives some informal definitions from the literature of what constitutes a “safe language”. Type systems are an important tool for designing safe languages, but in principle, an untyped language could be safe by virtue of performing certain checks at run-time. Since such checks generally hamper efficiency, in practice very few untyped languages are safe; Cardelli (1997) cites LISP as an example of an untyped, safe language (and assembly language as the quintessential untyped, unsafe language). Although typed languages may use a combination of run- and compile-time checks to ensure safety, they usually emphasise the latter. In other words the ideal is to have a type system implementing

algorithmically decidable checks used at compile-time to rule out all untrapped run-time errors (and some kinds of trapped ones as well). Of course some languages (such as C) employ types without any pretensions to safety.

### Formal type systems

---

- Constitute the precise, mathematical characterisation of informal type systems (such as occur in the manuals of most typed languages.)
- Basis for *type soundness* theorems: “any well-typed program cannot produce run-time errors (of some specified kind)”.
- Can decouple specification of typing aspects of a language from algorithmic concerns: the formal type system can define typing independently of particular implementations of type-checking algorithms.

#### Slide 4

Some languages are designed to be safe by virtue of a type system, but turn out not to be—because of unforeseen or unintended uses of certain combinations of their features (object-oriented languages seem particularly prone to this problem). We will see an example of this in Section 3, where we consider the combination of ML polymorphism with mutable references. Such difficulties have been a great spur to the development of the formal mathematics and logic of type systems: one can only *prove* that a language is safe after its syntax and operational semantics have been formally specified. The main point of this course is to introduce a little of this formalism and illustrate its uses. Standard ML (Milner, Tofte, Harper, and MacQueen 1997) is the shining example of a full-scale language possessing a complete such specification and whose *type soundness* (cf. Slide 4) has been subject to proof.<sup>1</sup>

---

<sup>1</sup>Standard ML is a sufficiently large language that a fully formalised proof of its type safety is surely enormous and certainly requires machine-assistance to carry out. However, since the language design was semantically-driven and had type safety very much in mind, it is possible to give convincing, if semi-formal, proofs of type safety for large fragments of it.

### Typical type system “judgement”

---

is a relation between typing environments ( $\Gamma$ ), program phrases ( $M$ ) and type expressions ( $\tau$ ) that we write as

$$\Gamma \vdash M : \tau$$

and read as “given the assignment of types to free identifiers of  $M$  specified by type environment  $\Gamma$ , then  $M$  has type  $\tau$ ”.

E.g.

$$f : int\ list \rightarrow int, b : bool \vdash (\text{if } b \text{ then } f\ \text{nil} \text{ else } 3) : int$$

is a valid typing judgement about ML.

#### Slide 5

The study of formal type systems is part of *structural operational semantics*: to specify a formal type system one gives a number of axioms and rules for inductively generating the kind of assertion, or “judgement”, shown on Slide 5. Ideally the rules follow the structure of the phrase  $M$ , explaining how to type it in terms of how its subphrases can be types— one speaks of *syntax-directed* sets of rules. It is worth pointing out that different language families use widely differing notations for typing—see Slide 6.

Once we have formalised a particular type system, we are in a position to *prove* results about *type soundness* (Slide 4) and the notions of *type checking*, *typeability* and *type inference* described on Slide 7. You have already seen some examples in the CST IB *Semantics of Programming Languages* course of formal type systems defined using inductive definitions generated by syntax-directed axioms and rules. In this course we look at more involved examples revolving around the notion of “parametric polymorphism”, to which we turn next.

### Notations for the typing relation

---

“foo has type bar”

ML-style (used in this course):

`foo : bar`

Haskell-style:

`foo :: bar`

C/Java-style:

`bar foo`

**Slide 6**

### Type checking, typeability, and type inference

---

Suppose given a type system for a programming language with judgements of the form  $\Gamma \vdash M : \tau$ .

*Type-checking* problem: given  $\Gamma$ ,  $M$ , and  $\tau$ , is  $\Gamma \vdash M : \tau$  derivable in the type system?

*Typeability* problem: given  $\Gamma$  and  $M$ , is there any  $\tau$  for which  $\Gamma \vdash M : \tau$  is derivable in the type system?

Second problem is usually harder than the first. Solving it usually involves devising a *type inference algorithm* computing a  $\tau$  for each  $\Gamma$  and  $M$  (or failing, if there is none).

**Slide 7**

## 2 ML Polymorphism

As indicated in the Introduction, static type-checking is regarded by many as an important aid to building large, well-structured, and reliable software systems. On the other hand, early forms of static typing, for example as found in Pascal, tended to hamper the ability to write *generic code*. For example, a procedure for sorting lists of one type of data could not be applied to lists of a different type of data. It is natural to want a *polymorphic* sorting procedure—one which operates (uniformly) on lists of several different types. The potential significance for programming languages of this phenomenon of *polymorphism* was first emphasised by Strachey (1967), who identified several different varieties: see Slide 8. Here we will concentrate on parametric polymorphism. One way to get it is to make the type parameterisation an explicit part of the language syntax: we will see an example of this in Section 4. In this section we look at the *implicit* version of parametric polymorphism first implemented in the ML family of languages (and subsequently adopted elsewhere, for example in forthcoming versions of Java and C#, where it is known as “generics”). ML phrases need contain little explicit type information: the type inference algorithm infers a “most general” type (scheme) for each well-formed phrase, from which all the other types of the phrase can be obtained by specialising type variables. These ideas should be familiar to you from your previous experience of Standard ML. The point of this section is to see how one gives a precise formalisation of a type system and its associated type inference algorithm for a small fragment of ML, called Mini-ML.

### ***Polymorphism = ‘has many types’***

---

*Overloading* (or ‘ad hoc’ polymorphism): same symbol denotes operations with unrelated implementations. (E.g. + might mean both integer addition and string concatenation.)

*Subsumption*  $\tau_1 <: \tau_2$ : any  $M_1 : \tau_1$  can be used as  $M_1 : \tau_2$  without violating safety.

*Parametric polymorphism* (“generics”): same expression belongs to a family of structurally related types. (E.g. in SML, length function

```
fun length nil      = 0
   | length (x :: xs) = 1 + (length xs)
```

has type  $\tau \text{ list} \rightarrow \text{int}$  for all types  $\tau$ .)

Slide 8

### Type variables and type schemes in Mini-ML

---

To formalise statements like

“*length* has type  $\tau \text{ list} \rightarrow \text{int}$ , for all types  $\tau$ ”

it is natural to introduce *type variables*  $\alpha$  (i.e. variables for which types may be substituted), , and write

$$\text{length} : \forall \alpha (\alpha \text{ list} \rightarrow \text{int}).$$

$\forall \alpha (\alpha \text{ list} \rightarrow \text{int})$  is an example of a *type scheme*.

**Slide 9**

## 2.1 An ML type system

As indicated on Slide 9, to formalise parametric polymorphism, we have to introduce *type variables*. An interactive ML system will just display  $\alpha \text{ list} \rightarrow \text{int}$  as the type of the *length* function (cf. Slide 8), leaving the universal quantification over  $\alpha$  implicit. However, when it comes to formalising the ML type system (as is done in the definition of the Standard ML ‘static semantics’ in Milner, Tofte, Harper, and MacQueen 1997, chapter 4) it is necessary to make this universal quantification over types explicit in some way. The reason for this has to do with the typing of local declarations. Consider the example given on Slide 10. The expression  $(f \text{ true}) :: (f \text{ nil})$  has type *bool list*, given some assumption about the type of the variable  $f$ . Two possible such assumptions are shown on Slide 11. Here we are interested in the second possibility since it leads to a type system with very useful properties. The particular grammar of ML types and type schemes that we will use is shown on Slide 12.

### Polymorphism of let-bound variables in ML

---

For example in

$$\text{let } f = \lambda x(x) \text{ in } (f \text{ true}) :: (f \text{ nil})$$

$\lambda x(x)$  has type  $\tau \rightarrow \tau$  for any type  $\tau$ , and the variable  $f$  to which it is bound is used polymorphically:

- in  $(f \text{ true})$ ,  $f$  has type  $bool \rightarrow bool$
- in  $(f \text{ nil})$ ,  $f$  has type  $bool \text{ list} \rightarrow bool \text{ list}$

Overall, the expression has type  $bool \text{ list}$ .

Slide 10

“Ad hoc” polymorphism:

if  $f : bool \rightarrow bool$   
 and  $f : bool \text{ list} \rightarrow bool \text{ list}$ ,  
 then  $(f \text{ true}) :: (f \text{ nil}) : bool \text{ list}$ .

“Parametric” polymorphism:

if  $f : \forall \alpha (\alpha \rightarrow \alpha)$ ,  
 then  $(f \text{ true}) :: (f \text{ nil}) : bool \text{ list}$ .

Slide 11

**Mini-ML types and type schemes**

---

*Types*

$\tau ::= \alpha$	type variable
$bool$	type of booleans
$\tau \rightarrow \tau$	function type
$\tau list$	list type

where  $\alpha$  ranges over a fixed, countably infinite set  $\text{TyVar}$ .

*Type Schemes*

$$\sigma ::= \forall A(\tau)$$

where  $A$  ranges over finite subsets of the set  $\text{TyVar}$ .

When  $A = \{\alpha_1, \dots, \alpha_n\}$ , we write  $\forall A(\tau)$  as

$$\forall \alpha_1, \dots, \alpha_n(\tau).$$

**Slide 12**

The following points about type schemes  $\forall A(\tau)$  should be noted.

- (i) The case when  $A$  is empty,  $A = \{\}$ , is allowed:  $\forall \{\}(\tau)$  is a well-formed type scheme. **We will often regard the sets of types as a subset of the set of type schemes by identifying the type  $\tau$  with the type scheme  $\forall \{\}(\tau)$ .**
- (ii) Any occurrences in  $\tau$  of a type variable  $\alpha \in A$  become bound in  $\forall A(\tau)$ . Thus by definition, the *free type variables* of a type scheme  $\forall A(\tau)$  are all those type variables which occur in  $\tau$ , but which are not in the finite set  $A$ . (For example the set of free type variables of  $\forall \alpha(\alpha \rightarrow \alpha')$  is  $\{\alpha'\}$ .) As usual for variable-binding constructs, we are not interested in the particular names of  $\forall$ -bound type variables (since we may have to change them to avoid variable capture during substitution of types for free type variables). Therefore **we will identify type schemes up to alpha-conversion of  $\forall$ -bound type variables**. For example,  $\forall \alpha(\alpha \rightarrow \alpha')$  and  $\forall \alpha''(\alpha'' \rightarrow \alpha')$  determine the same alpha-equivalence class and will be used interchangeably. Of course the finite set

$$ftv(\forall A(\tau))$$

of free type variables of a type scheme is well-defined up to alpha-conversion of bound type variables. Just as in (ii) we identified Mini-ML types  $\tau$  with trivial type schemes  $\forall \{\}(\tau)$ , so we will sometimes write

$$ftv(\tau)$$

for the finite set of type variables occurring in  $\tau$  (of course all such occurrences are free, because Mini-ML types do not involve binding operations).

- (iii) **ML type schemes are not ML types!** So for example,  $\alpha \rightarrow \forall \alpha' (\alpha')$  is neither a well-formed Mini-ML type nor a well-formed Mini-ML type scheme.<sup>1</sup> Rather, Mini-ML type schemes are a notation for families of types, parameterised by type variables. We get types from type schemes by substituting types for type variables, as we explain next.

### The “generalises” relation between type schemes and types

We say a type scheme  $\sigma = \forall \alpha_1, \dots, \alpha_n (\tau')$  *generalises* a type  $\tau$ , and write  $\boxed{\sigma \succ \tau}$  if  $\tau$  can be obtained from the type  $\tau'$  by simultaneously substituting some types  $\tau_i$  for the type variables  $\alpha_i$  ( $i = 1, \dots, n$ ):

$$\tau = \tau'[\tau_1/\alpha_1, \dots, \tau_n/\alpha_n].$$

(N.B. The relation is unaffected by the particular choice of names of bound type variables in  $\sigma$ .)

The converse relation is called specialisation: a type  $\tau$  is a *specialisation* of a type scheme  $\sigma$  if  $\sigma \succ \tau$ .

### Slide 13

Slide 13 gives some terminology and notation to do with substituting types for the bound type variables of a type scheme. The notion of a type scheme *generalising* a type will feature in the way variables are assigned types in the Mini-ML type system that we are going to define in this section.

**Example 2.1.1.** Some simple examples of generalisation:

$$\begin{aligned} \forall \alpha (\alpha \rightarrow \alpha) &\succ \text{bool} \rightarrow \text{bool} \\ \forall \alpha (\alpha \rightarrow \alpha) &\succ \alpha' \text{ list} \rightarrow \alpha' \text{ list} \\ \forall \alpha (\alpha \rightarrow \alpha) &\succ (\alpha' \rightarrow \alpha') \rightarrow (\alpha' \rightarrow \alpha'). \end{aligned}$$

However

$$\forall \alpha (\alpha \rightarrow \alpha) \not\succeq (\alpha' \rightarrow \alpha') \rightarrow \alpha'.$$

<sup>1</sup>The step of making type schemes first class types will be taken in Section 4.

This is because in a substitution  $\tau[\tau'/\alpha]$ , by definition we have to replace *all* occurrences in  $\tau$  of the type variable  $\alpha$  by  $\tau'$ . Thus when  $\tau = \alpha \rightarrow \alpha$ , there is no type  $\tau'$  for which  $\tau[\tau'/\alpha]$  is the type  $(\alpha \rightarrow \alpha) \rightarrow \alpha$ . (Simply because in the syntax tree of  $\tau[\tau'/\alpha] = \tau' \rightarrow \tau'$ , the two subtrees below the outermost constructor ‘ $\rightarrow$ ’ are equal (namely to  $\tau'$ ), whereas this is false of  $(\alpha \rightarrow \alpha) \rightarrow \alpha$ .) Another example:

$$\forall \alpha_1, \alpha_2 (\alpha_1 \rightarrow \alpha_2) \succ \alpha \text{ list} \rightarrow \text{bool}.$$

However

$$\forall \alpha_1 (\alpha_1 \rightarrow \alpha_2) \not\succeq \alpha \text{ list} \rightarrow \text{bool}$$

because  $\alpha_2$  is a free type variable in the type scheme  $\forall \alpha_1 (\alpha_1 \rightarrow \alpha_2)$  and so cannot be substituted for during specialisation.

### Mini-ML typing judgement

---

takes the form  $\boxed{\Gamma \vdash M : \tau}$  where

- the *typing environment*  $\Gamma$  is a finite function from variables to *type schemes*.  
(We write  $\Gamma = \{x_1 : \sigma_1, \dots, x_n : \sigma_n\}$  to indicate that  $\Gamma$  has domain of definition  $dom(\Gamma) = \{x_1, \dots, x_n\}$  and maps each  $x_i$  to the type scheme  $\sigma_i$  for  $i = 1..n$ .)
- $M$  is an Mini-ML expression
- $\tau$  is an Mini-ML type.

### Slide 14

Slide 14 gives the form of typing judgement we will use to illustrate ML polymorphism and type inference. Just as we only consider a small subset of ML types, we restrict attention to typings for a small subset of ML expressions,  $M$ , generated by the grammar on Slide 15. We use a non-standard syntax compared with the definition in (Milner, Tofte, Harper, and MacQueen 1997). For example we write  $\lambda x(M)$  for `fn x => M` and `let x = M1 in M2` for `let val x = M1 in M2 end`. Furthermore we call the symbol ‘ $x$ ’ occurring in these expressions a *variable* rather than a “(value) identifier”. As usual, the free variables of  $\lambda x(M)$  are those of  $M$ , except for  $x$ . In the expression `let x = M1 in M2`, any free occurrences of the variable  $x$  in  $M_2$  become bound in the `let`-expression. Similarly, in the expression

case  $M_1$  of  $\text{nil} \Rightarrow M_2 \mid x_1 :: x_2 \Rightarrow M_3$ , any free occurrences of the variables  $x_1$  and  $x_2$  in  $M_3$  become bound in the case-expression. The axioms and rules inductively generating the Mini-ML typing relation for these expressions are given on Slides 16–17.

Mini-ML expressions, $M$	
$::= x$	variable
true	boolean values
false	
if $M$ then $M$ else $M$	conditional
$\lambda x(M)$	function abstraction
$MM$	function application
let $x = M$ in $M$	local declaration
nil	nil list
$M :: M$	list cons
case $M$ of $\text{nil} \Rightarrow M \mid x :: x \Rightarrow M$	case expression

**Slide 15**

Note the following points about the type system defined on Slides 16–19.

- (i) Given a type environment  $\Gamma$  we write  $\Gamma, x : \sigma$  to indicate a typing environment with domain  $\text{dom}(\Gamma) \cup \{x\}$ , mapping  $x$  to  $\sigma$  and otherwise mapping like  $\Gamma$ . When we use this notation it will almost always be the case that  $x \notin \text{dom}(\Gamma)$  (cf. rules (fn), (let) and (case)).
- (ii) In rule (fn) we use  $\Gamma, x : \tau_1$  as an abbreviation for  $\Gamma, x : \forall \{ \} (\tau_1)$ . Similarly, in rule (case),  $\Gamma, x_1 : \tau_1, x_2 : \tau_1 \text{ list}$  really means  $\Gamma, x_1 : \forall \{ \} (\tau_1), x_2 : \forall \{ \} (\tau_1 \text{ list})$ . (Recall that by definition, a typing environment has to map variables to type schemes, rather than to types.)
- (iii) In rule (let) the notation  $\text{ftv}(\Gamma)$  means the set of all type variables occurring free in some type scheme assigned in  $\Gamma$ . (For example, if  $\Gamma = \{x_1 : \sigma_1, \dots, x_n : \sigma_n\}$ , then  $\text{ftv}(\Gamma) = \text{ftv}(\sigma_1) \cup \dots \cup \text{ftv}(\sigma_n)$ .) Thus the set  $A = \text{ftv}(\tau_1) - \text{ftv}(\Gamma)$  used in that rule consists of all type variables in  $\tau$  that do not occur freely in any type scheme assigned in  $\Gamma$ .

---

**Mini-ML type system, I**


---

(var  $\succ$ )       $\Gamma \vdash x : \tau$  if  $(x : \sigma) \in \Gamma$  and  $\sigma \succ \tau$

(bool)       $\Gamma \vdash B : bool$  if  $B \in \{\mathbf{true}, \mathbf{false}\}$

(if)      
$$\frac{\Gamma \vdash M_1 : bool \quad \Gamma \vdash M_2 : \tau \quad \Gamma \vdash M_3 : \tau}{\Gamma \vdash \mathbf{if } M_1 \mathbf{ then } M_2 \mathbf{ else } M_3 : \tau}$$

Slide 16

---

**Mini-ML type system, II**


---

(nil)       $\Gamma \vdash \mathbf{nil} : \tau list$

(cons)      
$$\frac{\Gamma \vdash M_1 : \tau \quad \Gamma \vdash M_2 : \tau list}{\Gamma \vdash M_1 :: M_2 : \tau list}$$

(case)      
$$\frac{\Gamma \vdash M_1 : \tau_1 list \quad \Gamma \vdash M_2 : \tau_2 \quad \Gamma, x_1 : \tau_1, x_2 : \tau_1 list \vdash M_3 : \tau_2}{\Gamma \vdash \mathbf{case } M_1 \mathbf{ of nil } \Rightarrow M_2 \quad | x_1 :: x_2 \Rightarrow M_3 : \tau_2}$$
 if  $x_1, x_2 \notin \mathit{dom}(\Gamma)$  and  $x_1 \neq x_2$

Slide 17

**Mini-ML type system, III**

---

(fn) 
$$\frac{\Gamma, x : \tau_1 \vdash M : \tau_2}{\Gamma \vdash \lambda x(M) : \tau_1 \rightarrow \tau_2} \quad \text{if } x \notin \text{dom}(\Gamma)$$

(app) 
$$\frac{\Gamma \vdash M_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash M_2 : \tau_1}{\Gamma \vdash M_1 M_2 : \tau_2}$$

Slide 18

**Mini-ML type system, IV**

---

(let) 
$$\frac{\Gamma \vdash M_1 : \sigma \quad \Gamma, x : \sigma \vdash M_2 : \tau}{\Gamma \vdash \text{let } x = M_1 \text{ in } M_2 : \tau} \quad \text{if } x \notin \text{dom}(\Gamma)$$

where for a type scheme  $\sigma = \forall A (\tau_1)$ , we write

$$\Gamma \vdash M_1 : \sigma$$

to mean  $\Gamma \vdash M_1 : \tau_1$  and  $A = \text{ftv}(\tau_1) - \text{ftv}(\Gamma)$ .

Slide 19



## 2.2 Examples of type inference, by hand

As for the full ML type system, for the type system we have just introduced the typeability problem (Slide 7) turns out to be decidable. Moreover, among all the possible type schemes a given closed Mini-ML expression may possess, there is a most general one—one from which all the others can be obtained by substitution. Before showing why this is the case, we give some specific examples of type inference in this type system.

### Two examples involving self-application

$$M \stackrel{\text{def}}{=} \text{let } f = \lambda x_1(\lambda x_2(x_1)) \text{ in } f f$$

$$M' \stackrel{\text{def}}{=} (\lambda f(f f)) \lambda x_1(\lambda x_2(x_1))$$

Are  $M$  and  $M'$  typeable in the Mini-ML type system?

### Slide 20

Given a typing environment  $\Gamma$  and an expression  $M$ , how can we decide whether or not there is a type scheme  $\sigma$  for which  $\Gamma \vdash M : \sigma$  holds? We are aided in this task by the *syntax-directed* (or “structural”) nature of the axioms and rules on Slides 16–19: if  $\Gamma \vdash M : \forall A(\tau)$  is derivable, i.e. if  $A = f_{tv}(\tau) - f_{tv}(\Gamma)$  and  $\Gamma \vdash M : \tau$  is derivable, then the outermost form of the expression  $M$  dictates which must be the last axiom or rule used in the proof of  $\Gamma \vdash M : \tau$ . Consequently, as we try to build a proof of a typing judgement  $\Gamma \vdash M : \tau$  from the bottom up, the structure of the expression  $M$  determines the shape of the tree together with which rules are used at its nodes and which axioms at its leaves. For example, for the particular expression  $M$  given on Slide 20, any proof of  $\vdash M : \forall A_1(\tau_1)$  from the axioms and rules, has to look like the tree given in Figure 1. Node (C0) is supposed to be an instance of the (let) rule; nodes (C1) and (C2) instances of the (fn) rule; leaves (C3), (C5), and (C6) instances of the (var  $\succ$ ) axiom; and node (C4) an instance of the (app) rule. For these to be valid instances the constraints (C0)–(C6) listed on Slide 21 have to be satisfied.

$$\begin{array}{c}
\frac{}{} \quad (C3) \\
\frac{x_1 : \tau_3, x_2 : \tau_5 \vdash x_1 : \tau_6}{x_1 : \tau_3 \vdash \lambda x_2(x_1) : \tau_4} \quad (C2) \quad \frac{}{} \quad (C5) \quad \frac{}{} \quad (C6)}{f : \forall A(\tau_2) \vdash f : \tau_7} \quad \frac{}{} \quad (C4)}{f : \forall A(\tau_2) \vdash f f : \tau_1} \quad (C4) \\
\frac{\{\} \vdash \lambda x_1(\lambda x_2(x_1)) : \tau_2}{\{\} \vdash \text{let } f = \lambda x_1(\lambda x_2(x_1)) \text{ in } f f : \tau_1} \quad (C0)
\end{array}$$

Figure 1: Skeleton proof tree for  $\text{let } f = \lambda x_1(\lambda x_2(x_1)) \text{ in } f f$ 

Constraints generated while inferring a type for $\text{let } f = \lambda x_1(\lambda x_2(x_1)) \text{ in } f f$	
(C0)	$A = ftv(\tau_2)$
(C1)	$\tau_2 = \tau_3 \rightarrow \tau_4$
(C2)	$\tau_4 = \tau_5 \rightarrow \tau_6$
(C3)	$\forall \{\} (\tau_3) \succ \tau_6$ , i.e. $\tau_3 = \tau_6$
(C4)	$\tau_7 = \tau_8 \rightarrow \tau_1$
(C5)	$\forall A(\tau_2) \succ \tau_7$
(C6)	$\forall A(\tau_2) \succ \tau_8$

**Slide 21**

Thus  $M$  is typeable if and only if we can find types  $\tau_1, \dots, \tau_8$  satisfying the constraints on Slide 21. First note that they imply

$$\tau_2 \stackrel{(C1)}{=} \tau_3 \rightarrow \tau_4 \stackrel{(C2)}{=} \tau_3 \rightarrow (\tau_5 \rightarrow \tau_6) \stackrel{(C3)}{=} \tau_6 \rightarrow (\tau_5 \rightarrow \tau_6).$$

So let us take  $\tau_5, \tau_6$  to be type variables, say  $\alpha_2, \alpha_1$  respectively. Hence by (C0),  $A = ftv(\tau_2) = ftv(\alpha_1 \rightarrow (\alpha_2 \rightarrow \alpha_1)) = \{\alpha_1, \alpha_2\}$ . Then (C4), (C5) and (C6) require that

$$\forall \alpha_1, \alpha_2 (\alpha_1 \rightarrow (\alpha_2 \rightarrow \alpha_1)) \succ \tau_8 \rightarrow \tau_1 \quad \text{and} \quad \forall \alpha_1, \alpha_2 (\alpha_1 \rightarrow (\alpha_2 \rightarrow \alpha_1)) \succ \tau_8.$$

In other words there have to be some types  $\tau_9, \dots, \tau_{12}$  such that

$$(C7) \quad \tau_9 \rightarrow (\tau_{10} \rightarrow \tau_9) = \tau_8 \rightarrow \tau_1$$

$$(C8) \quad \tau_{11} \rightarrow (\tau_{12} \rightarrow \tau_{11}) = \tau_8.$$

Now (C7) can only hold if

$$\tau_9 = \tau_8 \quad \text{and} \quad \tau_{10} \rightarrow \tau_9 = \tau_1$$

and hence

$$\tau_1 = \tau_{10} \rightarrow \tau_9 = \tau_{10} \rightarrow \tau_8 \stackrel{(C8)}{=} \tau_{10} \rightarrow (\tau_{11} \rightarrow (\tau_{12} \rightarrow \tau_{11})).$$

with  $\tau_{10}, \tau_{11}, \tau_{12}$  otherwise unconstrained. So if we take them to be type variables  $\alpha_3, \alpha_4, \alpha_5$  respectively, all in all, we can satisfy all the constraints on Slide 21 by defining

$$\begin{aligned} A &= \{\alpha_1, \alpha_2\} \\ \tau_1 &= \alpha_3 \rightarrow (\alpha_4 \rightarrow (\alpha_5 \rightarrow \alpha_4)) \\ \tau_2 &= \alpha_1 \rightarrow (\alpha_2 \rightarrow \alpha_1) \\ \tau_3 &= \alpha_1 \\ \tau_4 &= \alpha_2 \rightarrow \alpha_1 \\ \tau_5 &= \alpha_2 \\ \tau_6 &= \alpha_1 \\ \tau_7 &= (\alpha_4 \rightarrow (\alpha_5 \rightarrow \alpha_4)) \rightarrow (\alpha_3 \rightarrow (\alpha_4 \rightarrow (\alpha_5 \rightarrow \alpha_4))) \\ \tau_8 &= \alpha_4 \rightarrow (\alpha_5 \rightarrow \alpha_4). \end{aligned}$$

With these choices, Figure 1 becomes a valid proof of

$$\{ \} \vdash \text{let } f = \lambda x_1(\lambda x_2(x_1)) \text{ in } f f : \alpha_3 \rightarrow (\alpha_4 \rightarrow (\alpha_5 \rightarrow \alpha_4))$$

from the typing axioms and rules on Slides 16–19, i.e. we do have

$$(4) \quad \vdash \text{let } f = \lambda x_1(\lambda x_2(x_1)) \text{ in } f f : \forall \alpha_3, \alpha_4, \alpha_5 (\alpha_3 \rightarrow (\alpha_4 \rightarrow (\alpha_5 \rightarrow \alpha_4)))$$

as expected from Exercise 2.5.1.

If we go through the same type inference process for the expression  $M'$  on Slide 20 we generate a tree and set of constraints as in Figure 2. These imply in particular that

$$\tau_7 \stackrel{(C13)}{=} \tau_4 \stackrel{(C12)}{=} \tau_6 \stackrel{(C11)}{=} \tau_7 \rightarrow \tau_5.$$

But there are no types  $\tau_5, \tau_7$  satisfying  $\tau_7 = \tau_7 \rightarrow \tau_5$ , because  $\tau_7 \rightarrow \tau_5$  contains at least one more ‘ $\rightarrow$ ’ symbol than does  $\tau_7$ . So we conclude that  $(\lambda f(f f)) \lambda x_1(\lambda x_2(x_1))$  is not typeable within the ML type system.

---


$$\begin{array}{c}
\frac{}{f : \tau_4 \vdash f : \tau_6} \text{ (C12)} \quad \frac{}{f : \tau_4 \vdash f : \tau_7} \text{ (C13)} \quad \frac{}{x_1 : \tau_8, x_2 : \tau_{10} \vdash x_1 : \tau_{11}} \text{ (C16)} \\
\frac{}{f : \tau_4 \vdash f f : \tau_5} \text{ (C10)} \quad \frac{}{x_1 : \tau_8 \vdash \lambda x_2(x_1) : \tau_9} \text{ (C14)} \\
\frac{\frac{}{\{\} \vdash \lambda f(f f) : \tau_2} \text{ (C10)} \quad \frac{}{\{\} \vdash \lambda x_1(\lambda x_2(x_1)) : \tau_3} \text{ (C14)}}{\{\} \vdash (\lambda f(f f)) \lambda x_1(\lambda x_2(x_1)) : \tau_1} \text{ (C9)}
\end{array}$$

Constraints:

$$\begin{array}{ll}
\text{(C9)} & \tau_2 = \tau_3 \rightarrow \tau_1 \\
\text{(C10)} & \tau_2 = \tau_4 \rightarrow \tau_5 \\
\text{(C11)} & \tau_6 = \tau_7 \rightarrow \tau_5 \\
\text{(C12)} & \forall \{\} (\tau_4) \succ \tau_6, \text{ i.e. } \tau_4 = \tau_6 \\
\text{(C13)} & \forall \{\} (\tau_4) \succ \tau_7, \text{ i.e. } \tau_4 = \tau_7 \\
\text{(C14)} & \tau_3 = \tau_8 \rightarrow \tau_9 \\
\text{(C15)} & \tau_9 = \tau_{10} \rightarrow \tau_{11} \\
\text{(C16)} & \forall \{\} (\tau_{11}) \succ \tau_8, \text{ i.e. } \tau_{11} = \tau_8
\end{array}$$


---

Figure 2: Skeleton proof tree and constraints for  $(\lambda f(f f)) \lambda x_1(\lambda x_2(x_1))$

## 2.3 Principal type schemes

The type scheme  $\forall \alpha_3, \alpha_4, \alpha_5 (\alpha_3 \rightarrow (\alpha_4 \rightarrow (\alpha_5 \rightarrow \alpha_4)))$  not only satisfies (4), it is in fact the most general, or *principal* type scheme for  $\text{let } f = \lambda x_1 (\lambda x_2 (x_1)) \text{ in } f f$ , as defined on Slide 22. This makes use of the natural extension of the “generalises” relation,  $\succ$  (Slide 13), to a binary relation between (closed) type schemes. Exercise 2.5.3 gives an alternative characterisation of this relation. It is worth pointing out that in the presence of (a), the converse of condition (b) on Slide 22 holds: if  $\sigma \succ \sigma'$  and  $\vdash M : \sigma$ , then  $\vdash M : \sigma'$ . This is a consequence of the substitution property of valid Mini-ML typing judgements given in Exercise 2.5.6.

Slide 23 gives the main result about the Mini-ML typeability problem. It was first proved for a simple type system without polymorphic  $\text{let}$ -expressions by Hindley (1969) and extended to the full system by Damas and Milner (1982).

### Principal type schemes for closed expressions

A type scheme  $\sigma$  is the *principal* type scheme of a closed Mini-ML expression  $M$  if

- (a)  $\vdash M : \sigma$  is provable
- (b) for all  $\sigma'$ , if  $\vdash M : \sigma'$  is provable, then  $\sigma \succ \sigma'$

where by definition  $\sigma \succ \sigma'$  holds if  $\sigma' = \forall A' (\tau')$  with  $A' \cap \text{ftv}(\sigma) = \{ \}$  and  $\sigma \succ \tau'$  (as defined on Slide 13).

(Note that since we identify type schemes up to alpha-conversion of  $\forall$ -bound type variables, we can always satisfy the condition  $A' \cap \text{ftv}(\sigma) = \{ \}$  by suitably renaming the bound type variables of  $\sigma'$ .)

#### Slide 22

**Remark 2.3.1 (Complexity of the type checking algorithm).** Although typeability is decidable, it is known to be exponential-time complete. Furthermore, the principal type scheme of an expression can be exponentially larger than the expression itself, even if the type involved is represented efficiently as a directed acyclic graph. More precisely, the time taken to decide typeability and the space needed to display the principal type are both exponential in the number of nested  $\text{let}$ 's in the expression. For example the expression on Slide 24 (taken from Mairson 1990) has a principal type scheme which would take hundreds of pages to print out. It seems that such pathology does not arise naturally, and that the type checking phase of an ML compiler is not a bottle neck in practice. For more details about the complexity of ML type inference see (Mitchell 1996, Section 11.3.5).

---

**Theorem (Hindley; Damas-Milner)**


---

If the closed Mini-ML expression  $M$  is typeable (i.e.  $\vdash M : \sigma$  holds for some type scheme  $\sigma$ ), then there is a principal type scheme for  $M$ .

Indeed, there is an algorithm which, given any  $M$  as input, decides whether or not it is typeable and returns a principal type scheme if it is.

**Slide 23**

---

**An ML expression with a principal type scheme  
hundreds of pages long**


---

```

let pair =  $\lambda x(\lambda y(\lambda z(z x y)))$  in
  let  $x_1 = \lambda y(pair y y)$  in
    let  $x_2 = \lambda y(x_1(x_1 y))$  in
      let  $x_3 = \lambda y(x_2(x_2 y))$  in
        let  $x_4 = \lambda y(x_3(x_3 y))$  in
          let  $x_5 = \lambda y(x_4(x_4 y))$  in
             $x_5(\lambda y(y))$ 

```

(Taken from Mairson 1990.)

**Slide 24**

## 2.4 A type inference algorithm

The aim of this subsection is to sketch the proof of the Hindley-Damas-Milner theorem stated on Slide 23, by describing an algorithm,  $pt$ , for deciding typeability and returning a most general type scheme.  $pt$  is defined recursively, following structure of expressions (and its termination is proved by induction on the structure of expressions). As the examples in Section 2.2 should suggest, the algorithm depends crucially upon *unification*—the fact that the solvability of a finite set of equations between algebraic terms is decidable and that a most general solution exists, if any does. This fact was discovered by Robinson (1965) and has been a key ingredient in several logic-related areas of computer science (automated theorem proving, logic programming, and of course type systems, to name three). The form of unification algorithm,  $mgu$ , we need here is specified on Slide 25. Although we won't bother to give an implementation of  $mgu$  here (see for example (Rydeheard and Burstall 1988, Chapter 8), (Mitchell 1996, Section 11.2.2), or (Aho, Sethi, and Ullman 1986, Section 6.7) for more details), we do need to explain the notation for type substitutions introduced on Slide 25.

### Unification of ML types

There is an algorithm  $mgu$  which when input two Mini-ML types  $\tau_1$  and  $\tau_2$  decides whether  $\tau_1$  and  $\tau_2$  are *unifiable*, i.e. whether there exists a type-substitution  $S \in \text{Sub}$  with

$$(a) \ S(\tau_1) = S(\tau_2).$$

Moreover, if they are unifiable,  $mgu(\tau_1, \tau_2)$  returns the *most general unifier*—an  $S$  satisfying both (a) and

$$(b) \ \text{for all } S' \in \text{Sub}, \text{ if } S'(\tau_1) = S'(\tau_2), \text{ then } S' = TS \text{ for some } T \in \text{Sub}.$$

By convention  $mgu(\tau_1, \tau_2) = \text{FAIL}$  if (and only if)  $\tau_1$  and  $\tau_2$  are not unifiable.

### Slide 25

**Definition 2.4.1 (Type substitutions).** A *type substitution*  $S$  is a (totally defined) function from type variables to Mini-ML types with the property that  $S(\alpha) = \alpha$  for all but finitely many  $\alpha$ . We write  $\text{Sub}$  for the set of all such functions. The *domain* of  $S \in \text{Sub}$  is the finite set of variables

$$\text{dom}(S) \stackrel{\text{def}}{=} \{\alpha \in \text{TyVar} \mid S(\alpha) \neq \alpha\}$$

Given a type substitution  $S$ , the effect of applying the substitution to a type is written

$S \tau$ ; thus if  $\text{dom}(S) = \{\alpha_1, \dots, \alpha_n\}$  and  $S(\alpha_i)$  is the type  $\tau_i$  for each  $i = 1..n$ , then  $S(\tau)$  is the type resulting from simultaneously replacing each occurrence of  $\alpha_i$  in  $\tau$  with  $\tau_i$  (for all  $i = 1..n$ ), i.e.

$$S \tau = \tau[\tau_1/\alpha_1, \dots, \tau_n/\alpha_n]$$

using the notation for substitution from Slide 13. Notwithstanding the notation on the right hand side of the above equation, we prefer to write the application of a type substitution function  $S$  on the left of the type to which it is being applied.<sup>1</sup> As a result, the *composition*  $TS$  of two type substitutions  $S, T \in \text{Sub}$  means first apply  $S$  and then  $T$ . Thus by definition  $TS$  is the function mapping each type variable  $\alpha$  to the type  $T(S(\alpha))$  (apply the type substitution  $T$  to the type  $S(\alpha)$ ). Note that the function  $TS$  does satisfy the finiteness condition required of a substitution and we do have  $TS \in \text{Sub}$ ; indeed,  $\text{dom}(TS) \subseteq \text{dom}(T) \cup \text{dom}(S)$ .

More generally, if  $\text{dom}(S) = \{\alpha_1, \dots, \alpha_n\}$  and  $\sigma$  is an Mini-ML type scheme, then  $S \sigma$  will denote the result of the (capture-avoiding<sup>2</sup>) substitution of  $S(\alpha_i)$  for each free occurrence of  $\alpha_i$  in  $\sigma$  (for  $i = 1..n$ ).

### Principal type schemes for open expressions

A *solution* for the typing problem  $\Gamma \vdash M : ?$  is a pair  $(S, \sigma)$  consisting of a type substitution  $S$  and a type scheme  $\sigma$  satisfying

$$S \Gamma \vdash M : \sigma$$

(where  $S \Gamma = \{x_1 : S \sigma_1, \dots, x_n : S \sigma_n\}$ , if  $\Gamma = \{x_1 : \sigma_1, \dots, x_n : \sigma_n\}$ ).

Such a solution is *principal* if given any other,  $(S', \sigma')$ , there is some  $T \in \text{Sub}$  with  $TS = S'$  and  $T(\sigma) \succ \sigma'$ .

### Slide 26

Even though we are ultimately interested in the typeability of *closed* expressions, since the algorithm *pt* descends recursively through the subexpressions of the input expression, inevitably it has to generate typings for expressions with free variables. Hence we have to define the notions of typeability and principal type scheme for open expressions in the presence of a non-empty typing environment. This is done on Slide 26. To compute principal type schemes it suffices to compute ‘principal solutions’ in the sense of Slide 26: for if  $M$  is

<sup>1</sup>i.e. we write  $S \tau$  rather than  $\tau S$  as in the Part IB *Logic and Proof* course.

<sup>2</sup>Since we identify type schemes up to renaming their  $\forall$ -bound type variables, we always assume the bound type variables in  $\sigma$  are different from any type variables in the types  $S(\alpha_i)$ .

in fact closed, then any principal solution  $(S, \sigma)$  for the typing problem  $\{\} \vdash M : ?$  has the property that  $\sigma$  is a principal type scheme for  $M$  in the sense of Slide 22 (see Exercise 2.5.5).

### Specification for the principal typing algorithm, $pt$

$pt$  operates on typing problems  $\Gamma \vdash M : ?$  (consisting of a typing environment  $\Gamma$  and an Mini-ML expression  $M$ ). It returns either a pair  $(S, \tau)$  consisting of a type substitution  $S \in \text{Sub}$  and an Mini-ML type  $\tau$ , or the exception *FAIL*.

- If  $\Gamma \vdash M : ?$  has a solution (cf. Slide 26), then  $pt(\Gamma \vdash M : ?)$  returns  $(S, \tau)$  for some  $S$  and  $\tau$ ; moreover, setting  $A = (ftv(\tau) - ftv(S\Gamma))$ , then  $(S, \forall A(\tau))$  is a principal solution for the problem  $\Gamma \vdash M : ?$ .
- If  $\Gamma \vdash M : ?$  has no solution, then  $pt(\Gamma \vdash M : ?)$  returns *FAIL*.

### Slide 27

Slide 27 sets out in more detail what is required of the principal typing algorithm,  $pt$ . One possible algorithm in somewhat informal pseudocode (and leaving out the cases for `nil`, `cons`, and `case`-expressions) is sketched on Slide 28 and in Figure 3.<sup>1</sup> Note the following points about the definitions on Slide 28 and in Figure 3:

- (i) We implicitly assume that all bound variables in expressions and bound type variables in type schemes are distinct from each other and from any other variables in context. So, for example, the clause for function abstractions tacitly assumes that  $x \notin \text{dom}(\Gamma)$ ; and the clause for variables assumes that  $A \cap \text{ftv}(\Gamma) = \{\}$ .
- (ii) The type substitution  $Id$  occurring in the clauses for variables and booleans is the *identity* substitution which maps each type variable  $\alpha$  to itself.
- (iii) We have not given the clauses of the definition for `nil`, `cons`, and `case`-expressions (Exercise 2.5.4).

<sup>1</sup>A complete implementation of this algorithm in Fresh O’Caml ([www.freshml.org/foc/](http://www.freshml.org/foc/)) can be found on the course web page. The Fresh O’Caml code is remarkably close to the informal pseudocode given here, because of Fresh O’Caml’s sophisticated facilities for dealing with binding operations and fresh names.

- (iv) We do not give the proof that the definition in Figure 3 is correct (i.e. meets the specification on Slide 27): but see Exercise 2.5.7. The correctness of the algorithm depends upon an important property of Mini-ML typing, namely that *it is respected by the operation of substituting types for type variables*: see Exercise 2.5.6.

**Some of the clauses in a definition of  $pt$**

---

*Function abstractions:*  $pt(\Gamma \vdash \lambda x(M) : ?) \stackrel{\text{def}}{=} \text{let } \alpha = \text{fresh in}$   
 $\text{let } (S, \tau) = pt(\Gamma, x : \alpha \vdash M : ?) \text{ in } (S, S(\alpha) \rightarrow \tau)$

*Function applications:*  $pt(\Gamma \vdash M_1 M_2 : ?) \stackrel{\text{def}}{=} \text{let } (S_1, \tau_1) = pt(\Gamma \vdash M_1 : ?) \text{ in}$   
 $\text{let } (S_2, \tau_2) = pt(S_1 \Gamma \vdash M_2 : ?) \text{ in}$   
 $\text{let } \alpha = \text{fresh in}$   
 $\text{let } S_3 = mgu(S_2 \tau_1, \tau_2 \rightarrow \alpha) \text{ in } (S_3 S_2 S_1, S_3(\alpha))$

**Slide 28**

More efficient algorithms make use of a different approach to substitution and unification, based on equivalence relations on directed acyclic graphs and union-find algorithms: see (Rémy 2002, Sect. 2.4.2), for example. In that reference, and also in Pierce's book (Pierce 2002, Section 22.3), you will see an approach to type inference algorithms that views them as part of the more general problem of generating and solving *constraint problems*. This seems to be a fruitful viewpoint, because it accommodates a wide range of different type inference problems.

## 2.5 Exercises

**Exercise 2.5.1.** Here are some type checking problems, in the sense of Slide 7. Prove the following typings hold for the Mini-ML type system:

- $\vdash \lambda x(x :: \text{nil}) : \forall \alpha (\alpha \rightarrow \alpha \text{ list})$
- $\vdash \lambda x(\text{case } x \text{ of nil} \Rightarrow \text{true} \mid x_1 :: x_2 \Rightarrow \text{false}) : \forall \alpha (\alpha \text{ list} \rightarrow \text{bool})$
- $\vdash \lambda x_1(\lambda x_2(x_1)) : \forall \alpha_1, \alpha_2 (\alpha_1 \rightarrow (\alpha_2 \rightarrow \alpha_1))$
- $\vdash \text{let } f = \lambda x_1(\lambda x_2(x_1)) \text{ in } f f : \forall \alpha_1, \alpha_2, \alpha_3 (\alpha_1 \rightarrow (\alpha_2 \rightarrow (\alpha_3 \rightarrow \alpha_2)))$ .

- 
- **Variables:**  $pt(\Gamma \vdash x : ?) \stackrel{\text{def}}{=} \text{let } \forall A(\tau) = \Gamma(x) \text{ in } (Id, \tau)$
  - **let-Expressions:**  $pt(\Gamma \vdash \text{let } x = M_1 \text{ in } M_2 : ?) \stackrel{\text{def}}{=} \text{let } (S_1, \tau_1) = pt(\Gamma \vdash M_1 : ?) \text{ in}$   
 $\text{let } A = ftv(\tau_1) - ftv(S_1 \Gamma); \text{ in}$   
 $\text{let } (S_2, \tau_2) = pt(S_1 \Gamma, x : \forall A(\tau_1) \vdash M_2 : ?) \text{ in } (S_2 S_1, \tau_2)$
  - **Booleans ( $B = \text{true}, \text{false}$ ):**  $pt(\Gamma \vdash B : ?) \stackrel{\text{def}}{=} (Id, \text{bool})$
  - **Conditionals:**  $pt(\Gamma \vdash \text{if } M_1 \text{ then } M_2 \text{ else } M_3 : ?) \stackrel{\text{def}}{=} \text{let } (S_1, \tau_1) = pt(\Gamma \vdash M_1 : ?) \text{ in}$   
 $\text{let } S_2 = mgu(\tau_1, \text{bool}) \text{ in}$   
 $\text{let } (S_3, \tau_3) = pt(S_2 S_1 \Gamma \vdash M_2 : ?) \text{ in}$   
 $\text{let } (S_4, \tau_4) = pt(S_3 S_2 S_1 \Gamma \vdash M_3 : ?) \text{ in}$   
 $\text{let } S_5 = mgu(S_4 \tau_3, \tau_4) \text{ in } (S_5 S_4 S_3 S_2 S_1, S_5 \tau_4)$
- 

Figure 3: Some of the clauses in a definition of  $pt$ 

**Exercise 2.5.2.** Show that if  $\{ \} \vdash M : \sigma$  is provable, then  $M$  must be *closed*, i.e. have no free variables. [Hint: use rule induction for the rules on Slides 16–19 to show that the provable typing judgements,  $\Gamma \vdash M : \tau$ , all have the property that  $fv(M) \subseteq dom(\Gamma)$ .]

**Exercise 2.5.3.** Let  $\sigma$  and  $\sigma'$  be Mini-ML type schemes. Show that the relation  $\sigma \succ \sigma'$  defined on Slide 22 holds if and only if

$$\forall \tau (\sigma' \succ \tau \Rightarrow \sigma \succ \tau).$$

[Hint: use the following property of simultaneous substitution:

$$(\tau[\tau_1/\alpha_1, \dots, \tau_n/\alpha_n])[\vec{\tau}'/\vec{\alpha}'] = \tau[\tau_1[\vec{\tau}'/\vec{\alpha}']/\alpha_1, \dots, \tau_n[\vec{\tau}'/\vec{\alpha}']/\alpha_n]$$

which holds provided the type variables  $\vec{\alpha}'$  do not occur in  $\tau$ .]

**Exercise 2.5.4.** Try to augment the definition of  $pt$  on Slide 28 and in Figure 3 with clauses for `nil`, `cons`, and `case-expressions`.

**Exercise 2.5.5.** Suppose  $M$  is a closed expression and that  $(S, \sigma)$  is a principal solution for the typing problem  $\{ \} \vdash M : ?$  in the sense of Slide 26. Show that  $\sigma$  must be a principal type scheme for  $M$  in the sense of Slide 22. [Hint: first show that  $S$  has to be a (finite) permutation of type variables.]

**Exercise 2.5.6.** Show that if  $\Gamma \vdash M : \sigma$  is provable from the axioms and rules on Slides 16–19 and  $S \in \text{Sub}$  is a type substitution, then  $S \Gamma \vdash M : S \sigma$  is also provable.

**Exercise 2.5.7. [hard]** Try to give some of the proof that the definition in Figure 3 meets the specification on Slide 27. For example, try to prove that if

$$\forall \Gamma (pt(\Gamma \vdash M_i : ?) \text{ has correct properties})$$

for  $i = 1, 2$ , then

$$\forall \Gamma (pt(\Gamma \vdash M_1 M_2 : ?) \text{ has correct properties}).$$

(Why is it necessary to build the quantification over  $\Gamma$  into the inductive hypotheses?)

## 3 Polymorphic Reference Types

### 3.1 The problem

Recall from the Introduction that an important purpose of type systems is to provide *safety* (Slide 3) via *type soundness* results (Slide 4). Even if a programming language is intended to be safe by virtue of its type system, it can happen that separate features of the language, each desirable in themselves, can combine in unexpected ways to produce an unsound type system. In this section we look at an example of this which occurred in the development of the ML family of languages. The two features which combine in a nasty way are:

- ML's style of implicitly typed `let`-bound polymorphism, and
- reference types.

We have already treated the first topic in Section 2. The second concerns ML's imperative features, which are based upon the ability to dynamically create locally scoped storage locations which can be written to and read from. We begin by giving the syntax and typing rules for this. We augment the grammar for Mini-ML types (Slide 12) with a unit type (a type with a single value) and *reference* types; and correspondingly, we augment the grammar for Mini-ML expressions (Slide 15) with a unit value, and operations for reference creation, dereferencing and assignment. These additions are shown on Slide 29. We call the resulting language Midi-ML. The typing rules for these new forms of expression are given on Slide 30.

ML types and expressions for mutable references		
$\tau$	::=	...
		<i>unit</i> unit type
		$\tau$ <i>ref</i> reference type.
$M$	::=	...
		()            unit value
		<code>ref</code> $M$ reference creation
		<code>!</code> $M$ dereference
		$M$ <code>:=</code> $M$ assignment

<b>Midi-ML's extra typing rules</b>	
(unit)	$\Gamma \vdash () : unit$
(ref)	$\frac{\Gamma \vdash M : \tau}{\Gamma \vdash \mathbf{ref} M : \tau \mathit{ref}}$
(get)	$\frac{\Gamma \vdash M : \tau \mathit{ref}}{\Gamma \vdash !M : \tau}$
(set)	$\frac{\Gamma \vdash M_1 : \tau \mathit{ref} \quad \Gamma \vdash M_2 : \tau}{\Gamma \vdash M_1 := M_2 : unit}$

**Slide 30**

<b>Example 3.1.1</b>	
The expression	$\mathbf{let} r = \mathbf{ref} \lambda x(x) \mathbf{in}$ $\mathbf{let} u = (r := \lambda x'(\mathbf{ref} !x')) \mathbf{in}$ $(!r)()$
has type <i>unit</i> .	

**Slide 31**

**Example 3.1.1.** Here is an example of the typing rules on Slide 30 in use. The expression given on Slide 31 has type *unit*.

*Proof.* This can be deduced by applying the (let) rule (Slide 19) to the judgements

$$\begin{aligned} \{ \} \vdash \mathbf{ref} \lambda x(x) : (\alpha \rightarrow \alpha) \mathbf{ref} \\ r : \forall \alpha ((\alpha \rightarrow \alpha) \mathbf{ref}) \vdash \mathbf{let} u = (r := \lambda x'(\mathbf{ref} !x')) \mathbf{in} (!r)() : \mathbf{unit}. \end{aligned}$$

The first of these judgements has the following proof:

$$\frac{\frac{\frac{}{x : \alpha \vdash x : \alpha} (\mathbf{var} \succ)}{\{ \} \vdash \lambda x(x) : \alpha \rightarrow \alpha} (\mathbf{fn})}{\{ \} \vdash \mathbf{ref} \lambda x(x) : (\alpha \rightarrow \alpha) \mathbf{ref}} (\mathbf{ref})$$

The second judgement can be proved by applying the (let) rule to

$$(5) \quad r : \forall \alpha ((\alpha \rightarrow \alpha) \mathbf{ref}) \vdash r := \lambda x'(\mathbf{ref} !x') : \mathbf{unit}$$

$$(6) \quad r : \forall \alpha ((\alpha \rightarrow \alpha) \mathbf{ref}), u : \mathbf{unit} \vdash (!r)() : \mathbf{unit}$$

Writing  $\Gamma$  for the typing environment  $\{r : \forall \alpha ((\alpha \rightarrow \alpha) \mathbf{ref})\}$ , the proof of (5) is

$$\frac{\frac{\frac{\frac{}{\Gamma, x' : \alpha \mathbf{ref} \vdash x' : \alpha \mathbf{ref}} (\mathbf{var} \succ)}{\Gamma, x' : \alpha \mathbf{ref} \vdash !x' : \alpha} (\mathbf{get})}{\Gamma, x' : \alpha \mathbf{ref} \vdash \mathbf{ref} !x' : \alpha \mathbf{ref}} (\mathbf{ref})}{\Gamma \vdash \lambda x'(\mathbf{ref} !x') : \alpha \mathbf{ref} \rightarrow \alpha \mathbf{ref}} (\mathbf{fn})}{\Gamma \vdash r := \lambda x'(\mathbf{ref} !x') : \mathbf{unit}} (\mathbf{set})$$

while the proof of (6) is

$$\frac{\frac{\frac{}{\Gamma, u : \mathbf{unit} \vdash r : (\mathbf{unit} \rightarrow \mathbf{unit}) \mathbf{ref}} (\mathbf{var} \succ)}{\Gamma, u : \mathbf{unit} \vdash !r : \mathbf{unit} \rightarrow \mathbf{unit}} (\mathbf{get})}{\Gamma, u : \mathbf{unit} \vdash (!r)() : \mathbf{unit}} (\mathbf{app}) \quad \frac{}{\Gamma, u : \mathbf{unit} \vdash () : \mathbf{unit}} (\mathbf{unit})$$

□

Although the typing rules for references seem fairly innocuous, they combine with the previous typing rules, and with the (let) rule in particular, to produce a type system for which type soundness fails with respect to ML's operational semantics. For consider what happens when the expression on Slide 31, call it  $M$ , is evaluated.

Evaluation of the outermost **let**-binding in  $M$  creates a fresh storage location bound to  $r$  and containing the value  $\lambda x(x)$ . Evaluation of the second **let**-binding updates the contents

of  $r$  to the value  $\lambda x'(\text{ref } !x')$  and binds the unit value to  $u$ .<sup>1</sup> Next  $(!r)()$  is evaluated. This involves applying the current contents of  $r$ , which is  $\lambda x'(\text{ref } !x')$ , to the unit value  $()$ . This results in an attempt to evaluate  $!()$ , i.e. to dereference something which is not a storage location, an unsafe operation which should be trapped. Put more formally, we have

$$\langle M, \{\} \rangle \rightarrow \text{FAIL}$$

in the transition system defined in Figure 4 and Slide 32 (using the rather terse “evaluation contexts” style of Wright and Felleisen (1994)). The configurations of the transition system are of two kinds:

- A pair  $\langle M, s \rangle$ , where  $M$  is an ML expression and  $s$  is a *state*—a finite function mapping variables,  $x$ , (here being used as the names of storage locations) to syntactic *values*,  $V$ . (The possible forms of  $V$  for this fragment of ML are defined in Figure 4.) Furthermore, we require a well-formedness condition for such a pair to be a configuration: the free variables of  $M$  and of each value  $s(x)$  (as  $x$  ranges over  $\text{dom}(s)$ ) should be contained in the finite set  $\text{dom}(s)$ .
- The symbol *FAIL*, representing a run-time error.

(The notation  $s[x \mapsto V]$  used on Slide 32 means the state with domain of definition  $\text{dom}(s) \cup \{x\}$  mapping  $x$  to  $V$  and otherwise acting like  $s$ .)

<b>Midi-ML transitions involving references</b>
$\langle !x, s \rangle \rightarrow \langle s(x), s \rangle \quad \text{if } x \in \text{dom}(s)$
$\langle !V, s \rangle \rightarrow \text{FAIL} \quad \text{if } V \text{ not a variable}$
$\langle x := V', s \rangle \rightarrow \langle (), s[x \mapsto V'] \rangle$
$\langle V := V', s \rangle \rightarrow \text{FAIL} \quad \text{if } V \text{ not a variable}$
$\langle \text{ref } V, s \rangle \rightarrow \langle x, s[x \mapsto V] \rangle \quad \text{if } x \notin \text{dom}(s)$
where $V$ ranges over <i>values</i> :
$V ::= x \mid \lambda x(M) \mid () \mid \text{true} \mid \text{false} \mid \text{nil} \mid V :: V$

---

The axioms and rules inductively defining the transition system for Midi-ML are those on Slide 32 together with the following ones:

- $\langle \text{if true then } M_1 \text{ else } M_2, s \rangle \rightarrow \langle M_1, s \rangle$
- $\langle \text{if false then } M_1 \text{ else } M_2, s \rangle \rightarrow \langle M_2, s \rangle$
- $\langle \text{if } V \text{ then } M_1 \text{ else } M_2, s \rangle \rightarrow \text{FAIL}$ , if  $V \notin \{\text{true}, \text{false}\}$
- $\langle (\lambda x(M))V', s \rangle \rightarrow \langle M[V'/x], s \rangle$
- $\langle V V', s \rangle \rightarrow \text{FAIL}$ , if  $V$  not a function abstraction
- $\langle \text{let } x = V \text{ in } M, s \rangle \rightarrow \langle M[V/x], s \rangle$
- $\langle \text{case nil of nil} \Rightarrow M \mid x_1 :: x_2 \Rightarrow M', s \rangle \rightarrow \langle M, s \rangle$
- $\langle \text{case } V_1 :: V_2 \text{ of nil} \Rightarrow M \mid x_1 :: x_2 \Rightarrow M', s \rangle \rightarrow \langle M'[V_1/x_1, V_2/x_2], s \rangle$
- $\langle \text{case } V \text{ of nil} \Rightarrow M \mid x_1 :: x_2 \Rightarrow M', s \rangle \rightarrow \text{FAIL}$ , if  $V$  is neither nil nor a cons-value
- $$\frac{\langle M, s \rangle \rightarrow \langle M', s' \rangle}{\langle \mathcal{E}[M], s \rangle \rightarrow \langle \mathcal{E}[M'], s' \rangle}$$
- $$\frac{\langle M, s \rangle \rightarrow \text{FAIL}}{\langle \mathcal{E}[M], s \rangle \rightarrow \text{FAIL}}$$

where  $V$  ranges over *values*:

$$V ::= x \mid \lambda x(M) \mid () \mid \text{true} \mid \text{false} \mid \text{nil} \mid V :: V$$

$\mathcal{E}$  ranges over *evaluation contexts*:

$$\begin{aligned} \mathcal{E} ::= & - \mid \text{if } \mathcal{E} \text{ then } M \text{ else } M \mid \mathcal{E} M \mid V \mathcal{E} \mid \text{let } x = \mathcal{E} \text{ in } M \mid \mathcal{E} :: M \mid V :: \mathcal{E} \\ & \mid \text{case } \mathcal{E} \text{ of nil} \Rightarrow M \mid x :: x \Rightarrow M \mid \text{ref } \mathcal{E} \mid !\mathcal{E} \mid \mathcal{E} := M \mid V := \mathcal{E} \end{aligned}$$

and  $\mathcal{E}[M]$  denotes the Midi-ML expression that results from replacing all occurrences of “-” by  $M$  in  $\mathcal{E}$ .

---

Figure 4: Transition system for Midi-ML

### 3.2 Restoring type soundness

The root of the problem described in the previous section lies in the fact that typing expressions like  $\text{let } r = \text{ref } M_1 \text{ in } M_2$  with the (let) rule allows the storage location (bound to)  $r$  to have a type scheme  $\sigma$  generalising the reference type of the type of  $M_1$ . Occurrences of  $r$  in  $M_2$  refer to the same, shared location and evaluation of  $M_2$  may cause assignments to this shared location which restrict the possible type of subsequent occurrences of  $r$ . But the typing rule allows all these occurrences of  $r$  to have *any* type which is a specialisation of  $\sigma$ , and this can lead to unsafe expressions being assigned types, as we have seen.

We can avoid this problem by devising a type system that prevents generalisation of type variables occurring in the types of shared storage locations. A number of ways of doing this have been proposed in the literature: see (Wright 1995) for a survey of them. The one adopted in the original, 1990, definition of Standard ML (Milner, Tofte, and Harper 1990) was that proposed by Tofte (1990). It involves partitioning the set of type variables into two (countably infinite) halves, the “applicative type variables” (ranged over by  $\alpha$ ) and the “imperative type variables” (ranged over by  $\_ \alpha$ ). The rule (ref) is restricted by insisting that  $\tau$  only involve imperative type variables; in other words the principal type scheme of  $\lambda x(\text{ref } x)$  becomes  $\forall \_ \alpha (\_ \alpha \rightarrow \_ \alpha \text{ ref})$ , rather than  $\forall \alpha (\alpha \rightarrow \alpha \text{ ref})$ . Furthermore, and crucially, the (let) rule (Slide 19) is restricted by requiring that when the type scheme  $\sigma = \forall A(\tau)$  assigned to  $M_1$  is such that  $A$  contains imperative type variables, then  $M_1$  must be a value (and hence in particular its evaluation does not create any fresh storage locations).

This solution has the advantage that in the new system the typeability of expressions not involving references is just the same as in the old system. However, it has the disadvantage that the type system makes distinctions between expressions which are behaviourally equivalent (i.e. which should be contextually equivalent). For example there are many list-processing functions that can be defined in the pure functional fragment of ML by recursive definitions, but which have more efficient definitions using local references. Unfortunately, if the type scheme of the former is something like  $\forall \alpha (\alpha \text{ list} \rightarrow \alpha \text{ list})$ , the type scheme of the latter may well be the different type scheme  $\forall \_ \alpha (\_ \alpha \text{ list} \rightarrow \_ \alpha \text{ list})$ . So we will not be able to use the two versions of such a function interchangeably.

The authors of the revised, 1996, definition of Standard ML (Milner, Tofte, Harper, and MacQueen 1997) adopt a simpler solution, proposed independently by Wright (1995). This removes the distinction between applicative and imperative type variables (in effect, all type variables are imperative, but the usual symbols  $\alpha, \alpha' \dots$  are used) while retaining a value-restricted form of the (let) rule, as shown on Slide 33.<sup>1</sup> Thus our version of this type system is based upon exactly the same form of type, type scheme and typing judgement as before, with the typing relation being generated inductively by the axioms and rules on Slides 16–19 and 30, except that the applicability of the (let) rule is restricted as on Slide 33.

---

<sup>1</sup>Since the variable  $u$  does not occur in its body,  $M$ 's innermost `let`-expression is just a way of expressing the sequence  $(r := \lambda x'(\text{ref } !x')) ; (!r)()$  in the fragment of ML that we are using for illustrative purposes.

<sup>1</sup>N.B. what we call a value, (Milner, Tofte, Harper, and MacQueen 1997) calls a *non-expansive expression*.

**Value-restricted typing rule for let-expressions**

---


$$\text{(letv)} \quad \frac{\Gamma \vdash M_1 : \tau_1 \quad \Gamma, x : \forall A(\tau_1) \vdash M_2 : \tau_2}{\Gamma \vdash \text{let } x = M_1 \text{ in } M_2 : \tau_2} \quad (\dagger)$$

( $\dagger$ ) provided  $x \notin \text{dom}(\Gamma)$  and

$$A = \begin{cases} \{\} & \text{if } M_1 \text{ is not a value} \\ \text{ftv}(\tau_1) - \text{ftv}(\Gamma) & \text{if } M_1 \text{ is a value} \end{cases}$$

(Recall that values are given by  
 $V ::= x \mid \lambda x(M) \mid () \mid \text{true} \mid \text{false} \mid \text{nil} \mid V :: V.$ )

**Slide 33**

**Example 3.2.1.** The expression on Slide 31 is not typeable in the type system for Midi-ML resulting from replacing rule (let) by the value-restricted rule (letv) on Slide 33 (keeping all the other axioms and rules the same).

*Proof.* Because of the form of the expression, the last rule used in any proof of its typeability must end with (letv). Because of the side condition on that rule and since  $\text{ref } \lambda x(x)$  is *not* a value, the rule has to be applied with  $A = \{\}$ . This entails trying to type

$$(7) \quad \text{let } u = (r := \lambda x'(\text{ref } !x')) \text{ in } (!r)()$$

in the typing environment  $\Gamma = \{r : (\alpha \rightarrow \alpha) \text{ ref}\}$ . But this is impossible, because the type variable  $\alpha$  is not universally quantified in this environment, whereas the two instances of  $r$  in (7) are of different implicit types (namely  $(\alpha \text{ ref} \rightarrow \alpha \text{ ref}) \text{ ref}$  and  $(\text{unit} \rightarrow \text{unit}) \text{ ref}$ ).  $\square$

The above example is all very well, but how do we know that we have achieved safety with this type system for Midi-ML? The answer lies in a formal proof of the *type soundness* property stated on Slide 34. To prove this result, one first has to formulate a definition of typing for general configurations  $\langle M, s \rangle$  when the state  $s$  is non-empty and then show

- typing is preserved under steps of transition,  $\rightarrow$ ;
- if a configuration can be typed, it cannot possess a transition to *FAIL*.

Thus a sequence of transitions from such a well-typed configuration can never lead to the *FAIL* configuration. We do not have the time to give the details in this course: the interested reader is referred to (Wright and Felleisen 1994; Harper 1994) for examples of similar type soundness results.

### Type soundness for Midi-ML with the value restriction

---

For any closed Midi-ML expression  $M$ , if there is some type scheme  $\sigma$  for which

$$\vdash M : \sigma$$

is provable in the value-restricted type system (axioms and rules on Slides 16–18, 30 and 33), then *evaluation of  $M$  does not fail*, i.e. there is no sequence of transitions of the form

$$\langle M, \{ \} \rangle \rightarrow \dots \rightarrow \text{FAIL}$$

for the transition system  $\rightarrow$  defined in Figure 4 (where  $\{ \}$  denotes the empty state).

#### Slide 34

Although the typing rule (letv) does allow one to achieve type soundness for polymorphic references in a pleasingly straightforward way, it does mean that some expressions not involving references that are typeable in the original ML type system are no longer typeable (Exercise 3.3.2.) Wright (1995, Sections 3.2 and 3.3) analyses the consequences of this and presents evidence that it is not a hindrance to the use of Standard ML in practice.

### 3.3 Exercises

**Exercise 3.3.1.** Letting  $M$  denote the expression on Slide 31 and  $\{ \}$  the empty state, show that  $\langle M, \{ \} \rangle \rightarrow^* \text{FAIL}$  is provable in the transition system defined in Figure 4.

**Exercise 3.3.2.** Give an example of a Mini-ML let-expression which is typeable in the type system of Section 2.1, but not in the type system of Section 3.2 for Midi-ML with the value-restricted rule (letv).

## 4 Polymorphic Lambda Calculus

In this section we take a look at a type system for explicitly typed parametric polymorphism, variously called the *polymorphic lambda calculus*, the *second order typed lambda calculus*, or *system F*. It was invented by the logician Girard (1972) and, independently and for different purposes, by the computer scientist Reynolds (1974). It has turned out to play a foundational role in the development of type systems somewhat similar to that played by Church's untyped lambda calculus in the development of functional programming. Although it is syntactically very simple, it turns out that a wide range of types and type constructions can be represented in the polymorphic lambda calculus.

### 4.1 From type schemes to polymorphic types

We have seen examples (Example 2.1.2 and the first example on Slide 20) of the fact that the ML type system permits `let`-bound variables to be used polymorphically within the body of a `let`-expression. As Slide 35 points out, the same is not true of  $\lambda$ -bound variables within the body of a function abstraction. This is a consequence of the fact that ML types and type schemes are separate syntactic categories and the function type constructor,  $\rightarrow$ , operates on the former, but not on the latter. Recall that an important purpose of type systems is to provide *safety* (Slide 3) via *type soundness* (Slide 4). Use of expressions such as those mentioned on Slide 35 does not seem intrinsically unsafe (although use of the second one may cause non-termination—cf. the definition of the fixed point combinator in untyped lambda calculus). So it is not unreasonable to seek type systems more powerful than the ML type system, in the sense that more expressions become typeable.

One apparently attractive way of achieving this is just to merge types and type schemes together: this results in the so-called *polymorphic types* shown on Slide 36. So let us consider extending the ML type system to assign polymorphic types to expressions. So we consider judgements of the form  $\Gamma \vdash M : \pi$  where:

- $\pi$  is a polymorphic type;
- $\Gamma$  is a finite function from variables to polymorphic types.

In order to make full use of the mixing of  $\rightarrow$  and  $\forall$  present in polymorphic types we have to replace the axiom (`var`  $\succ$ ) of Slide 16 by the axiom and two rules shown on Slide 37. (These are in fact versions for polymorphic types of ‘admissible rules’ in the original ML type system.) In rule (`spec`),  $\pi[\pi'/\alpha]$  indicates the polymorphic type resulting from substituting  $\pi'$  for all free occurrences of  $\alpha$  in  $\pi$ .

**$\lambda$ -bound variables in ML cannot be used  
polymorphically within a function abstraction**

---

E.g.  $\lambda f((f \text{ true}) :: (f \text{ nil}))$  and  $\lambda f(f f)$  are not typeable in the ML type system.

---

**Syntactically**, because in rule

$$(\text{fn}) \frac{\Gamma, x : \tau_1 \vdash M : \tau_2}{\Gamma \vdash \lambda x(M) : \tau_1 \rightarrow \tau_2}$$

the abstracted variable has to be assigned a *trivial* type scheme (recall  $x : \tau_1$  stands for  $x : \forall \{ \} (\tau_1)$ ).

**Semantically**, because  $\forall A (\tau_1) \rightarrow \tau_2$  is not semantically equivalent to an ML type when  $A \neq \{ \}$ .

**Slide 35**

*Monomorphic types ...*

$$\tau ::= \alpha \mid \text{bool} \mid \tau \rightarrow \tau \mid \tau \text{ list}$$

*...and type schemes*

$$\sigma ::= \tau \mid \forall \alpha (\sigma)$$

*Polymorphic types*

$$\pi ::= \alpha \mid \text{bool} \mid \pi \rightarrow \pi \mid \pi \text{ list} \mid \forall \alpha (\pi)$$


---

E.g.  $\alpha \rightarrow \alpha'$  is a type,  $\forall \alpha (\alpha \rightarrow \alpha')$  is a type scheme and a polymorphic type (but not a monomorphic type),  $\forall \alpha (\alpha) \rightarrow \alpha'$  is a polymorphic type, but not a type scheme.

**Slide 36**

Identity, Generalisation and Specialisation	
(id)	$\Gamma \vdash x : \pi \quad \text{if } (x : \pi) \in \Gamma$
(gen)	$\frac{\Gamma \vdash M : \pi}{\Gamma \vdash M : \forall \alpha (\pi)} \quad \text{if } \alpha \notin \text{ftv}(\Gamma)$
(spec)	$\frac{\Gamma \vdash M : \forall \alpha (\pi)}{\Gamma \vdash M : \pi[\pi'/\alpha]}$

**Slide 37**

**Example 4.1.1.** In the modified ML type system (with polymorphic types and  $(\text{var } \succ)$  replaced by (id), (gen), and (spec)) one can prove the following typings for expressions which are untypeable in ML:

- (8)  $\{\} \vdash \lambda f((f \text{ true}) :: (f \text{ nil})) : \forall \alpha (\alpha \rightarrow \alpha) \rightarrow \text{bool list}$   
 (9)  $\{\} \vdash \lambda f(f f) : \forall \alpha (\alpha) \rightarrow \forall \alpha (\alpha).$

*Proof.* The proof of (8) is rather easy to find and is left as an exercise. Here is a proof for (9):

$$\begin{array}{c}
 \frac{}{f : \forall \alpha_1 (\alpha_1) \vdash f : \forall \alpha_1 (\alpha_1)} \text{ (id)} \quad \frac{}{f : \forall \alpha_1 (\alpha_1) \vdash f : \forall \alpha_1 (\alpha_1)} \text{ (id)} \\
 \frac{}{f : \forall \alpha_1 (\alpha_1) \vdash f : \alpha_2 \rightarrow \alpha_2} \text{ (1)} \quad \frac{}{f : \forall \alpha_1 (\alpha_1) \vdash f : \alpha_2} \text{ (2)} \\
 \frac{}{f : \forall \alpha_1 (\alpha_1) \vdash f f : \alpha_2} \text{ (app)} \\
 \frac{}{f : \forall \alpha_1 (\alpha_1) \vdash f f : \forall \alpha_2 (\alpha_2)} \text{ (gen)} \\
 \frac{}{\{\} \vdash \lambda f(f f) : \forall \alpha_1 (\alpha_1) \rightarrow \forall \alpha_2 (\alpha_2)} \text{ (fn)}.
 \end{array}$$

Nodes (1) and (2) are both instances of the (spec) rule: the first uses the substitution  $(\alpha_2 \rightarrow \alpha_2)/\alpha_1$ , whilst the second uses  $\alpha_2/\alpha_1$ .  $\square$

**Fact** (see Wells 1994):

For the modified ML type system with polymorphic types and  $(\text{var } \succ)$  replaced by the axiom and rules on Slide 37, *the type checking and typeability problems* (cf. Slide 7) *are equivalent and undecidable.*

### Slide 38

So why does the ML programming language not use this extended type system with polymorphic types? The answer lies in the result stated on Slide 38: there is no algorithm to decide typeability for this type system (Wells 1994). The difficulty with automatic type inference for this type system lies in the fact that the generalisation and specialisation rules are not syntax-directed: since an application of either (gen) or (spec) does not change the expression  $M$  being checked, it is hard to know when to try to apply them in the bottom-up construction of proof inference trees. By contrast, in an ML type system based on (id), (gen) and (spec), but retaining the two-level stratification of types into monomorphic types and type schemes, this difficulty can be overcome. For in that case one can in fact push uses of (spec) right up to the leaves of a proof tree (where they merge with (id) axioms to become  $(\text{var } \succ)$  axioms) and push uses of (gen) right down to the root of the tree (and leave them implicit, as we did on Slide 19).

## 4.2 The PLC type system

The negative result on Slide 38 does not rule out the use of the polymorphic types of Slide 36 in programming languages, since one may consider *explicitly typed* languages (Slide 39) where the tagging of expressions with type information renders the typeability problem essentially trivial. We consider such a language in this subsection, the *polymorphic lambda calculus* (PLC).

### Explicitly versus implicitly typed languages

---

*Implicit:* little or no type information is included in program phrases and typings have to be inferred (ideally, entirely at compile-time). (E.g. Standard ML.)

*Explicit:* most, if not all, types for phrases are explicitly part of the syntax. (E.g. Java.)

---

E.g. self application function of type  $\forall \alpha (\alpha \rightarrow \forall \alpha (\alpha))$   
(cf. Example 4.1.1)

Implicitly typed version:  $\lambda f (f f)$

Explicitly type version:  $\lambda f : \forall \alpha_1 (\alpha_1) (\Lambda \alpha_2 (f(\alpha_2 \rightarrow \alpha_2)(f \alpha_2)))$

#### Slide 39

**Remark 4.2.1 (Explicitly typed languages).** One often hears the view that programming languages which enforce a large amount of explicit type information in programs are inconveniently verbose and/or force the programmer to make algorithmically irrelevant decisions about typings. But of course it really depends upon the intended applications. At one extreme, in a scripting language (interpreted interactively, used by a single person to develop utilities in a rapid edit-run-debug cycle) implicit typing may be desirable. Whereas at the opposite extreme, a language used to develop large software systems (involving separate compilation of modules by different teams of programmers) may benefit greatly from explicit typing (not least as a form of documentation of programmer's intentions, but also of course to enforce interfaces between separate program parts). Apart from these issues, explicitly typed languages are useful as *intermediate languages* in optimising compilers, since certain optimising transformations depend upon the type information they contain. See (Harper and Stone 1997), for example.

PLC syntax		
<i>Types</i>	$\tau ::=$	$\alpha$ type variable $  \quad \tau \rightarrow \tau$ function type $  \quad \forall \alpha (\tau)$ $\forall$ -type
<i>Expressions</i>	$M ::=$	$x$ variable $  \quad \lambda x : \tau (M)$ function abstraction $  \quad M M$ function application $  \quad \Lambda \alpha (M)$ type generalisation $  \quad M \tau$ type specialisation
( $\alpha$ and $x$ range over fixed, countably infinite sets $\text{TyVar}$ and $\text{Var}$ respectively.)		

**Slide 40**

Functions on types
<p>In PLC, <math>\Lambda \alpha (M)</math> is an anonymous notation for the function <math>F</math> mapping each type <math>\tau</math> to the value of <math>M[\tau/\alpha]</math> (of some particular type). <math>F \tau</math> denotes the result of applying such a function to a type.</p> <p>Computation in PLC involves beta-reduction for such functions on types</p> $(\Lambda \alpha (M)) \tau \rightarrow M[\tau/\alpha]$ <p>as well as the usual form of beta-reduction from <math>\lambda</math>-calculus</p> $(\lambda x : \tau (M_1)) M_2 \rightarrow M_1[M_2/x]$

**Slide 41**

The explicit type information we need to add to expressions to get syntax-directed versions of the (gen) and (spec) rules (Slide 37) concerns the operations of *type generalisation* and *type specialisation*. These are forms of function abstraction and application respectively—for functions defined on the collection of all types (and taking values in one particular type), rather than on the values of one particular type. See Slide 41. The polymorphic lambda calculus, PLC, provides rather sparse means for defining such functions—for example there is no “typecase” construct that allows branching according to which type expression is input. As a result, PLC is really a calculus of *parametrically polymorphic* functions (cf. Slide 8). The PLC syntax is given on Slide 40. Its types,  $\tau$ , are like the polymorphic types,  $\pi$ , given on Slide 36, except that we have omitted *bool* and *(\_) list*—because in fact these and many other forms of datatype are representable in PLC (see Section 4.4 below). We have also omitted *let*-expressions, because (unlike the ML type system presented in Section 2.1) they are definable from function abstraction and application with the correct typing properties: see Exercise 4.5.3.

**Remark 4.2.2 (Operator association and scoping).** As in the ordinary lambda calculus, one often writes a series of PLC applications without parentheses, using the convention that application associates to the left. Thus  $M_1 M_2 M_3$  means  $(M_1 M_2)M_3$ , and  $M_1 M_2 \tau_3$  means  $(M_1 M_2)\tau_3$ . Note that an expression like  $M_1 \tau_2 M_3$  can only associate as  $(M_1 \tau_2)M_3$ , since association the other way involves an ill-formed expression  $(\tau_2 M_3)$ . Similarly  $M_1 \tau_2 \tau_3$  can only be associated as  $(M_1 \tau_2)\tau_3$  (since  $\tau_1 \tau_2$  is an ill-formed type). On the other hand it is conventional to associate a series of function types to the right. Thus  $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$  means  $\tau_1 \rightarrow (\tau_2 \rightarrow \tau_3)$ .

We delimit the scope of  $\forall$ -,  $\lambda$ -, and  $\Lambda$ -binders with parentheses. Another common way of writing these binders employs ‘dot’ notation

$$\forall \alpha . \tau \quad \lambda x : \tau . M \quad \Lambda \alpha . M$$

with the convention that the scope extends as far to the right as possible. For example  $\forall \alpha_1 . (\forall \alpha_2 . \tau \rightarrow \alpha_1) \rightarrow \alpha_1$  means  $\forall \alpha_1 (\forall \alpha_2 (\tau \rightarrow \alpha_1) \rightarrow \alpha_1)$ . One often writes iterated binders using lists of bound (type) variables:

$$\begin{aligned} \forall \alpha_1, \alpha_2 (\tau) &\stackrel{\text{def}}{=} \forall \alpha_1 (\forall \alpha_2 (\tau)) \\ \lambda x_1 : \tau_1, x_2 : \tau_2 (M) &\stackrel{\text{def}}{=} \lambda x_1 : \tau_1 (\lambda x_2 : \tau_2 (M)) \\ \Lambda \alpha_1, \alpha_2 (M) &\stackrel{\text{def}}{=} \Lambda \alpha_1 (\Lambda \alpha_2 (M)) . \end{aligned}$$

It is also common to write a type specialisation by subscripting the type:  $M_\tau \stackrel{\text{def}}{=} M \tau$ .

**Remark 4.2.3 (Free and bound (type) variables).** Any occurrences in  $\tau$  of a type variable  $\alpha$  become bound in  $\forall \alpha (\tau)$ . Thus by definition, the finite set,  $ftv(\tau)$ , of *free type variables of a type  $\tau$* , is given by

$$\begin{aligned} ftv(\alpha) &\stackrel{\text{def}}{=} \{\alpha\} \\ ftv(\tau_1 \rightarrow \tau_2) &\stackrel{\text{def}}{=} ftv(\tau_1) \cup ftv(\tau_2) \\ ftv(\forall \alpha (\tau)) &\stackrel{\text{def}}{=} ftv(\tau) - \{\alpha\}. \end{aligned}$$

Any occurrences in  $M$  of a variable  $x$  become bound in  $\lambda x : \tau (M)$ . Thus by definition, the finite set,  $fv(M)$ , of *free variables of an expression*  $M$ , is given by

$$\begin{aligned} fv(x) &\stackrel{\text{def}}{=} \{x\} \\ fv(\lambda x : \tau (M)) &\stackrel{\text{def}}{=} fv(M) - \{x\} \\ fv(M_1 M_2) &\stackrel{\text{def}}{=} fv(M_1) \cup fv(M_2) \\ fv(\Lambda \alpha (M)) &\stackrel{\text{def}}{=} fv(M) \\ fv(M \tau) &\stackrel{\text{def}}{=} fv(M). \end{aligned}$$

Moreover, since types occur in expressions, we have to consider the *free type variables of an expression*. The only type variable binding construct at the level of expressions is generalisation: any occurrences in  $M$  of a type variable  $\alpha$  become bound in  $\Lambda \alpha (M)$ . Thus

$$\begin{aligned} ftv(x) &\stackrel{\text{def}}{=} \{ \} \\ ftv(\lambda x : \tau (M)) &\stackrel{\text{def}}{=} ftv(\tau) \cup ftv(M) \\ ftv(M_1 M_2) &\stackrel{\text{def}}{=} ftv(M_1) \cup ftv(M_2) \\ ftv(\Lambda \alpha (M)) &\stackrel{\text{def}}{=} ftv(M) - \{\alpha\} \\ ftv(M \tau) &\stackrel{\text{def}}{=} ftv(M) \cup ftv(\tau). \end{aligned}$$

As usual, we implicitly identify PLC types and expressions up to alpha-conversion of bound type variables and bound variables. For example

$$(\lambda x : \alpha (\Lambda \alpha (x \alpha))) x \quad \text{and} \quad (\lambda x' : \alpha (\Lambda \alpha' (x' \alpha'))) x$$

are alpha-convertible. We will always choose names of bound variable as in the second expression rather than the first, i.e. distinct from any free variables (and from each other).

**Remark 4.2.4 (Substitution).** For PLC, there are three forms of (capture-avoiding) substitution, well-defined up to alpha-conversion:

- $\tau[\tau'/\alpha]$  denotes the type resulting from substituting a type  $\tau'$  for all free occurrences of the type variable  $\alpha$  in a type  $\tau$ .
- $M[M'/x]$  denotes the expression resulting from substituting an expression  $M'$  for all free occurrences of the variable  $x$  in the expression  $M$ .
- $M[\tau/\alpha]$  denotes the expression resulting from substituting a type  $\tau$  for all free occurrences of the type variable  $\alpha$  in an expression  $M$ .

The PLC type system uses typing judgements of the form shown on Slide 42. Its typing relation is the collection of such judgements inductively defined by the axiom and rules on Slide 43.

### PLC typing judgement

---

takes the form  $\boxed{\Gamma \vdash M : \tau}$  where

- the *typing environment*  $\Gamma$  is a finite function from variables to PLC types.  
(We write  $\Gamma = \{x_1 : \tau_1, \dots, x_n : \tau_n\}$  to indicate that  $\Gamma$  has domain of definition  $dom(\Gamma) = \{x_1, \dots, x_n\}$  and maps each  $x_i$  to the PLC type  $\tau_i$  for  $i = 1..n$ .)
- $M$  is a PLC expression
- $\tau$  is a PLC type.

**Slide 42**

### PLC type system, I

---

(var)	$\Gamma \vdash x : \tau$ if $(x : \tau) \in \Gamma$
(fn)	$\frac{\Gamma, x : \tau_1 \vdash M : \tau_2}{\Gamma \vdash \lambda x : \tau_1 (M) : \tau_1 \rightarrow \tau_2}$ if $x \notin dom(\Gamma)$
(app)	$\frac{\Gamma \vdash M_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash M_2 : \tau_1}{\Gamma \vdash M_1 M_2 : \tau_2}$
(gen)	$\frac{\Gamma \vdash M : \tau}{\Gamma \vdash \Lambda \alpha (M) : \forall \alpha (\tau)}$ if $\alpha \notin ftv(\Gamma)$
(spec)	$\frac{\Gamma \vdash M : \forall \alpha (\tau_1)}{\Gamma \vdash M \tau_2 : \tau_1[\tau_2/\alpha]}$

**Slide 43**

**An incorrect “proof”**

---


$$\frac{\frac{\frac{}{x_1 : \alpha, x_2 : \alpha \vdash x_2 : \alpha} \text{ (var)}}{x_1 : \alpha \vdash \lambda x_2 : \alpha (x_2) : \alpha \rightarrow \alpha} \text{ (fn)}}{x_1 : \alpha \vdash \Lambda \alpha (\lambda x_2 : \alpha (x_2)) : \forall \alpha (\alpha \rightarrow \alpha)} \text{ (wrong!)}$$

**Slide 44**

**Remark 4.2.5 (Side-condition on rule (gen)).** To illustrate the force of the side-condition on rule (gen) on Slide 43, consider the last step of the ‘proof’ on Slide 44. It is not a correct instance of the (gen) rule, because the concluding judgement, whose typing environment  $\Gamma = \{x_1 : \alpha\}$ , does not satisfy  $\alpha \notin \text{ftv}(\Gamma)$  (since  $\text{ftv}(\Gamma) = \{\alpha\}$  in this case). On the other hand, the expression  $\Lambda \alpha (\lambda x_2 : \alpha (x_2))$  does have type  $\forall \alpha (\alpha \rightarrow \alpha)$  given the typing environment  $\{x_1 : \alpha\}$ . Here is a correct proof of that fact:

$$\frac{\frac{\frac{}{x_1 : \alpha, x_2 : \alpha' \vdash x_2 : \alpha'} \text{ (var)}}{x_1 : \alpha \vdash \lambda x_2 : \alpha' (x_2) : \alpha' \rightarrow \alpha'} \text{ (fn)}}{x_1 : \alpha \vdash \Lambda \alpha' (\lambda x_2 : \alpha' (x_2)) : \forall \alpha' (\alpha' \rightarrow \alpha')} \text{ (gen)}$$

where we have used the freedom afforded by alpha-conversion to rename the bound type variable to make it distinct from the free type variables of the typing environment: since we identify types and expressions up to alpha-conversion, the judgement

$$x_1 : \alpha \vdash \Lambda \alpha (\lambda x_2 : \alpha (x_2)) : \forall \alpha (\alpha \rightarrow \alpha)$$

is the same as

$$x_1 : \alpha \vdash \Lambda \alpha' (\lambda x_2 : \alpha' (x_2)) : \forall \alpha' (\alpha' \rightarrow \alpha')$$

and indeed, is the same as

$$x_1 : \alpha \vdash \Lambda \alpha' (\lambda x_2 : \alpha' (x_2)) : \forall \alpha'' (\alpha'' \rightarrow \alpha'').$$

**Example 4.2.6.** On Slide 39 we claimed that  $\lambda f : \forall \alpha_1 (\alpha_1) (\Lambda \alpha_2 (f(\alpha_2 \rightarrow \alpha_2)(f \alpha_2)))$  has type  $\forall \alpha (\alpha) \rightarrow \forall \alpha (\alpha)$ . Here is a proof of that in the PLC type system:

$$\frac{\frac{\frac{}{f : \forall \alpha_1 (\alpha_1) \vdash f : \forall \alpha_1 (\alpha_1)}{\text{(var)}}}{f : \forall \alpha_1 (\alpha_1) \vdash f(\alpha_2 \rightarrow \alpha_2) : \alpha_2 \rightarrow \alpha_2} \text{(spec)}}{\frac{f : \forall \alpha_1 (\alpha_1) \vdash f(\alpha_2 \rightarrow \alpha_2)(f \alpha_2) : \alpha_2}{f : \forall \alpha_1 (\alpha_1) \vdash \Lambda \alpha_2 (f(\alpha_2 \rightarrow \alpha_2)(f \alpha_2)) : \forall \alpha_2 (\alpha_2)} \text{(gen)}}{\frac{}{\{ \} \vdash \lambda f : \forall \alpha_1 (\alpha_1) (\Lambda \alpha_2 (f(\alpha_2 \rightarrow \alpha_2)(f \alpha_2))) : (\forall \alpha_1 (\alpha_1)) \rightarrow \forall \alpha_2 (\alpha_2)} \text{(fn)}} \text{(app)}$$

**Example 4.2.7.** There is no PLC type  $\tau$  for which

$$(10) \quad \{ \} \vdash \Lambda \alpha ((\lambda x : \alpha (x)) \alpha) : \tau$$

is provable within the PLC type system.

*Proof.* Because of the syntax-directed nature of the axiom and rules of the PLC type system, any proof of (10) would have to look like

$$\frac{\frac{\frac{}{x : \alpha \vdash x : \alpha} \text{(var)}}{\{ \} \vdash \lambda x : \alpha (x) : \tau''} \text{(fn)}}{\{ \} \vdash (\lambda x : \alpha (x)) \alpha : \tau'} \text{(spec)}}{\{ \} \vdash \Lambda \alpha ((\lambda x : \alpha (x)) \alpha) : \tau} \text{(gen)}$$

for some types  $\tau$ ,  $\tau'$  and  $\tau''$ . For the application of rule (fn) to be correct, it must be that  $\tau'' = \alpha \rightarrow \alpha$ . But then the application of rule (spec) is impossible, because  $\alpha \rightarrow \alpha$  is not a  $\forall$ -type. So no such proof can exist.  $\square$

**Decidability of the PLC typeability  
and type-checking problems**

---

**Theorem.**

For each PLC typing problem,  $\Gamma \vdash M : ?$ , there is at most one PLC type  $\tau$  for which  $\Gamma \vdash M : \tau$  is provable. Moreover there is an algorithm, *typ*, which when given any  $\Gamma \vdash M : ?$  as input, returns such a  $\tau$  if it exists and *FAILs* otherwise.

**Corollary.**

The PLC type checking problem is decidable: we can decide whether or not  $\Gamma \vdash M : \tau$  is provable by checking whether  $\text{typ}(\Gamma \vdash M : ?) = \tau$ .

(N.B. equality of PLC types up to alpha-conversion is decidable.)

**Slide 45**

### 4.3 PLC type inference

As Examples 4.2.6 and 4.2.7 suggest, the type checking and typeability problems (Slide 7) are very easy to solve for the PLC type system, compared with the ML type system. This is because of the explicit type information contained in PLC expressions together with the syntax-directed nature of the typing rules. The situation is summarised on Slide 45. The “uniqueness of types” property stated on the slide is easy to prove by induction on the structure of the expression  $M$ , exploiting the syntax-directed nature of the axiom and rules of the PLC type system. Moreover, the type inference algorithm *typ* emerges naturally from this proof, defined recursively according to the structure of PLC expressions. The clauses of its definition are given on Slides 46 and 47.<sup>1</sup> The definition relies upon the easily verified fact that equality of PLC types up to alpha-conversion is decidable. It also assumes that the various implicit choices of names of bound variables and bound type variables are made so as to keep them distinct from the relevant free variables and free type variables. For example, in the clause for type generalisations  $\Lambda \alpha (M)$ , we assume the bound type variable  $\alpha$  is chosen so that  $\alpha \notin \text{ftv}(\Gamma)$ .

---

<sup>1</sup>An implementation of this algorithm in Fresh O’Caml ([www.freshml.org/foc/](http://www.freshml.org/foc/)) can be found on the course web page. The Fresh O’Caml code is remarkably close to the informal pseudocode given here, because of Fresh O’Caml’s sophisticated facilities for dealing with binding operations and fresh names.

---

**PLC type-checking algorithm, I**


---

*Variables:*

$$\text{typ}(\Gamma, x : \tau \vdash x : ?) \stackrel{\text{def}}{=} \tau$$

*Function abstractions:*

$$\text{typ}(\Gamma \vdash \lambda x : \tau_1 (M) : ?) \stackrel{\text{def}}{=} \text{let } \tau_2 = \text{typ}(\Gamma, x : \tau_1 \vdash M : ?) \text{ in } \tau_1 \rightarrow \tau_2$$

*Function applications:*

$$\text{typ}(\Gamma \vdash M_1 M_2 : ?) \stackrel{\text{def}}{=} \text{let } \tau_1 = \text{typ}(\Gamma \vdash M_1 : ?) \text{ in}$$

let  $\tau_2 = \text{typ}(\Gamma \vdash M_2 : ?)$  in

case  $\tau_1$  of  $\tau \rightarrow \tau' \mapsto$  if  $\tau = \tau_2$  then  $\tau'$  else *FAIL*

|  $- \mapsto$  *FAIL*

**Slide 46**

---

**PLC type-checking algorithm, II**


---

*Type generalisations:*

$$\text{typ}(\Gamma \vdash \Lambda \alpha (M) : ?) \stackrel{\text{def}}{=} \text{let } \tau = \text{typ}(\Gamma \vdash M : ?) \text{ in } \forall \alpha (\tau)$$

let  $\tau = \text{typ}(\Gamma \vdash M : ?)$  in  $\forall \alpha (\tau)$

*Type specialisations:*

$$\text{typ}(\Gamma \vdash M \tau_2 : ?) \stackrel{\text{def}}{=} \text{let } \tau = \text{typ}(\Gamma \vdash M : ?) \text{ in}$$

case  $\tau$  of  $\forall \alpha (\tau_1) \mapsto \tau_1[\tau_2/\alpha]$

|  $- \mapsto$  *FAIL*

**Slide 47**

## 4.4 Datatypes in PLC

The aim of this subsection is to give some impression of just how expressive is the PLC type system. Many kinds of datatype, including both concrete data (booleans, natural numbers, lists, various kinds of tree, ...) and also abstract datatypes involving information hiding, can be represented in PLC. Such representations involve

- defining a suitable PLC type for the data,
- defining some PLC expressions for the various operations associated with the data,
- demonstrating that these expressions have both the correct typings and the expected computational behaviour.

In order to deal with the last point, we first have to consider some operational semantics for PLC. Most studies of the computational properties of polymorphic lambda calculus have been based on the PLC analogue of the notion of *beta-reduction* from untyped lambda calculus. This is defined on Slide 48.

**Beta-reduction of PLC expressions**

---

$M$  *beta-reduces to*  $M'$  *in one step*,  $\boxed{M \rightarrow M'}$ , means  $M'$  can be obtained from  $M$  (up to alpha-conversion, of course) by replacing a subexpression which is a *redex* by its corresponding *reduct*. The redex-reduct pairs are of two forms:

$$(\lambda x : \tau (M_1)) M_2 \rightarrow M_1[M_2/x]$$

$$(\Lambda \alpha (M)) \tau \rightarrow M[\tau/\alpha].$$

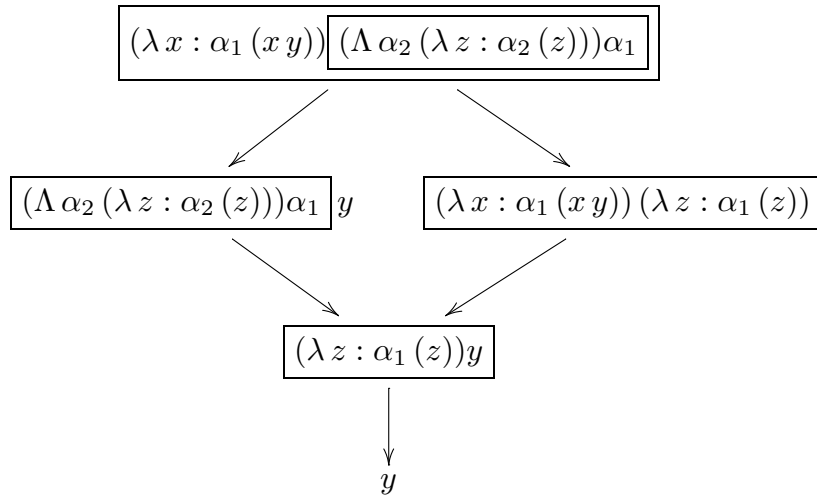
$M \rightarrow^* M'$  indicates a chain of finitely<sup>†</sup> many beta-reductions.  
 († possibly zero—which just means  $M$  and  $M'$  are alpha-convertible).

$M$  is in *beta-normal form* if it contains no redexes.

**Slide 48**

**Example 4.4.1.** Here are some examples of beta-reductions. The various redexes are shown

boxed. Clearly, the final expression  $y$  is in beta-normal form.



#### Properties of PLC beta-reduction on typeable expressions

Suppose  $\Gamma \vdash M : \tau$  is provable in the PLC type system. Then the following properties hold:

**Subject Reduction.** If  $M \rightarrow M'$ , then  $\Gamma \vdash M' : \tau$  is also a provable typing.

**Church Rosser Property.** If  $M \rightarrow^* M_1$  and  $M \rightarrow^* M_2$ , then there is  $M'$  with  $M_1 \rightarrow^* M'$  and  $M_2 \rightarrow^* M'$ .

**Strong Normalisation Property.** There is no infinite chain  $M \rightarrow M_1 \rightarrow M_2 \rightarrow \dots$  of beta-reductions starting from  $M$ .

#### Slide 49

Slide 49 lists some important properties of *typeable* PLC expressions. The first is a weak form of type soundness result (Slide 4) and its proof is sufficiently straightforward that it may be set as an exercise (Exercise 4.5.6). We do not give the proofs of the Church Rosser and Strong Normalisations properties here. The latter is a very difficult result.<sup>1</sup> It was first

<sup>1</sup>Since it in fact implies the consistency of second order arithmetic, it furnishes a concrete example of Gödel's famous incompleteness theorem: the strong normalisation property of PLC is a statement

proved by (Girard 1972) using a clever technique called “reducibility candidates”; if you are interested in seeing the details, look at (Girard 1989, Chapter 14) for an accessible account of the proof.

**PLC beta-conversion,  $=_{\beta}$**

---

By definition,  $M =_{\beta} M'$  holds if there is a finite chain

$$M - \dots - M'$$

where each  $-$  is either  $\rightarrow$  or  $\leftarrow$ , i.e. a beta-reduction in one direction or the other. (A chain of length zero is allowed—in which case  $M$  and  $M'$  are equal, up to alpha-conversion, of course.)

Church Rosser + Strong Normalisation properties imply that, for typeable PLC expressions,  $M =_{\beta} M'$  holds if and only if there is some beta-normal form  $N$  with

$$M \rightarrow^* N \leftarrow^* M'$$

**Slide 50**

**Theorem 4.4.2.** *The properties listed on Slide 49 have the following consequences.*

- (i) *Each typeable PLC expression,  $M$ , possesses a beta-normal form, i.e. an  $N$  such that  $M \rightarrow^* N \nrightarrow$ , which is unique (up to alpha-conversion).*
- (ii) *The equivalence relation of beta-conversion (Slide 50) between typeable PLC expressions is decidable, i.e. there is an algorithm which, when given two typeable PLC expressions, decides whether or not they are beta-convertible.*

*Proof.* For (i), first note that such a beta-normal form exists because if we start reducing redexes in  $M$  (in any order) the chain of reductions cannot be infinite (by Strong Normalisation) and hence terminates in a beta-normal form. Uniqueness of the beta-normal form follows by the Church Rosser property: if  $M \rightarrow^* N_1$  and  $M \rightarrow^* N_2$ , then  $N_1 \rightarrow^* M' \leftarrow^* N_2$  holds for some  $M'$ ; so if  $N_1$  and  $N_2$  are beta-normal forms, then it must be that  $N_1 \rightarrow^* M'$  and  $N_2 \rightarrow^* M'$  are chains of beta-reductions of zero length and hence  $N_1 = M' = N_2$  (equality up to alpha-conversion).

For (ii), we can use an algorithm which reduces the beta-redexes of each expression in any order until beta-normal forms are reached (in finitely many steps, by Strong Normalisation); these normal forms are equal (up to alpha-conversion) if and only if the original

---

that can be formalised within second order arithmetic, is true (as witnessed by a proof that goes outside second order arithmetic), but cannot be proved within that system.

expressions are beta-convertible. (And of course, the relation of alpha-convertibility is decidable.)  $\square$

**Remark 4.4.3.** In fact, the Church Rosser property holds for all PLC expressions, whether or not they are typeable. However, the Strong Normalisation properties definitely fails for *untypeable* expressions. For example, consider

$$\Omega \stackrel{\text{def}}{=} (\lambda f : \alpha (f f))(\lambda f : \alpha (f f))$$

from which there is an infinite chain of beta-reductions, namely  $\Omega \rightarrow \Omega \rightarrow \Omega \rightarrow \dots$ . As with the untyped lambda calculus, one can regard polymorphic lambda calculus as a rather pure kind of typed functional programming language in which computation consists of reducing typeable expressions to beta-normal form. From this viewpoint, the properties on Slide 49 tell us that (unlike the case of untyped lambda calculus) PLC cannot be “Turing powerful”, i.e. not all partial recursive functions can be programmed in it (using a suitable encoding of numbers). This is simply because, by virtue of Strong Normalisation, computation always terminates on well-typed programs.

Now that we have explained PLC dynamics, we return to the question of representing datatypes as PLC types. We consider first the simple example of booleans and then the more complicated example of polymorphic lists.

### Polymorphic booleans

---

$$bool \stackrel{\text{def}}{=} \forall \alpha (\alpha \rightarrow (\alpha \rightarrow \alpha))$$

$$True \stackrel{\text{def}}{=} \Lambda \alpha (\lambda x_1 : \alpha, x_2 : \alpha (x_1))$$

$$False \stackrel{\text{def}}{=} \Lambda \alpha (\lambda x_1 : \alpha, x_2 : \alpha (x_2))$$

$$if \stackrel{\text{def}}{=} \Lambda \alpha (\lambda b : bool, x_1 : \alpha, x_2 : \alpha (b \alpha x_1 x_2))$$

**Example 4.4.4 (Booleans).** The PLC type corresponding to the ML datatype

$$\text{datatype } \textit{bool} = \textit{True} \mid \textit{False}$$

is shown on Slide 51. The idea behind this representation is that the “algorithmic essence” of a boolean,  $b$ , is the operation  $\lambda x_1 : \alpha, x_2 : \alpha (\text{if } b \text{ then } x_1 \text{ else } x_2)$  of type  $\alpha \rightarrow \alpha \rightarrow \alpha$ ,<sup>1</sup> taking a pair of expressions of the same type and returning one or other of them. Clearly, this operation is parametrically polymorphic in the type  $\alpha$ , so in PLC we can take the step of identifying booleans with expressions of the corresponding  $\forall$ -type,  $\forall \alpha (\alpha \rightarrow \alpha \rightarrow \alpha)$ . Note that for the PLC expressions  $\textit{True}$  and  $\textit{False}$  defined on Slide 51 the typings

$$\{ \} \vdash \textit{True} : \forall \alpha (\alpha \rightarrow \alpha \rightarrow \alpha) \quad \text{and} \quad \{ \} \vdash \textit{False} : \forall \alpha (\alpha \rightarrow \alpha \rightarrow \alpha)$$

are both provable. The `if_then_else_` construct, given for the above ML datatype by a case-expression

$$\text{case } M_1 \text{ of } \textit{True} \Rightarrow M_2 \mid \textit{False} \Rightarrow M_3$$

has an explicitly typed analogue in PLC, viz.  $\textit{if } \tau M_1 M_2 M_3$ , where  $\tau$  is supposed to be the common type of  $M_2$  and  $M_3$  and  $\textit{if}$  is the PLC expression given on Slide 51. It is not hard to see that

$$\{ \} \vdash \textit{if} : \forall \alpha (\textit{bool} \rightarrow (\alpha \rightarrow (\alpha \rightarrow \alpha))).$$

Thus if  $\Gamma \vdash M_1 : \textit{bool}$ ,  $\Gamma \vdash M_2 : \tau$  and  $\Gamma \vdash M_3 : \tau$ , then  $\Gamma \vdash \textit{if } \tau M_1 M_2 M_3 : \tau$  (cf. the typing rule (if) on Slide 16). Furthermore, the expressions  $\textit{True}$ ,  $\textit{False}$ , and  $\textit{if}$  have the expected dynamic behaviour:

- if  $M_1 \rightarrow^* \textit{True}$  and  $M_2 \rightarrow^* N$ , then  $\textit{if } \tau M_1 M_2 M_3 \rightarrow^* N$ ;
- if  $M_1 \rightarrow^* \textit{False}$  and  $M_3 \rightarrow^* N$ , then  $\textit{if } \tau M_1 M_2 M_3 \rightarrow^* N$ .

It is in fact the case that *True and False are the only closed beta-normal forms in PLC of type bool* (up to alpha-conversion, of course), but it is beyond the scope of this course to prove it.

---

<sup>1</sup>Recall our notational conventions:  $\alpha \rightarrow \alpha \rightarrow \alpha$  means  $\alpha \rightarrow (\alpha \rightarrow \alpha)$ .

### Polymorphic lists

---

$$\alpha \text{ list} \stackrel{\text{def}}{=} \forall \alpha' (\alpha' \rightarrow (\alpha \rightarrow \alpha' \rightarrow \alpha') \rightarrow \alpha')$$

$$\text{Nil} \stackrel{\text{def}}{=} \Lambda \alpha, \alpha' (\lambda x' : \alpha', f : \alpha \rightarrow \alpha' \rightarrow \alpha' (x'))$$

$$\begin{aligned} \text{Cons} \stackrel{\text{def}}{=} \Lambda \alpha (\lambda x : \alpha, \ell : \alpha \text{ list} (\Lambda \alpha' ( \\ \lambda x' : \alpha', f : \alpha \rightarrow \alpha' \rightarrow \alpha' ( \\ f x (\ell \alpha' x' f)))))) \end{aligned}$$

Slide 52

### Iteratively defined functions on finite lists

---

$A^*$   $\stackrel{\text{def}}{=}$  finite lists of elements of the set  $A$

Given a set  $A'$ , an element  $x' \in A'$ , and a function  $f : A \rightarrow A' \rightarrow A'$ , the *iteratively defined function*  $\text{listIter } x' f$  is the unique function  $g : A^* \rightarrow A'$  satisfying:

$$\begin{aligned} g \text{ Nil} &= x' \\ g (x :: \ell) &= f x (g \ell). \end{aligned}$$

for all  $x \in A$  and  $\ell \in A^*$ .

Slide 53

**Example 4.4.5 (Lists).** The polymorphic type corresponding to the ML datatype

$$\text{datatype } \alpha \text{ list} = \text{Nil} \mid \text{Cons of } \alpha * (\alpha \text{ list})$$

is shown on Slide 52. Undoubtedly it looks rather mysterious at first sight. The idea behind this representation has to do with the operation of *iteration over a list* shown on Slide 53. The existence of such functions  $\text{listIter } x' f$  does in fact characterise the set  $A^*$  of finite lists over a set  $A$  uniquely up to bijection. We can take the operation

$$(11) \quad \lambda x' : \alpha', f : \alpha \rightarrow \alpha' \rightarrow \alpha' (\text{listIter } x' f \ell)$$

(of type  $\alpha' \rightarrow (\alpha \rightarrow \alpha' \rightarrow \alpha') \rightarrow \alpha'$ ) as the “algorithmic essence” of the list  $\ell : \alpha \text{ list}$ . Clearly this operation is parametrically polymorphic in  $\alpha'$  and so we are led to the  $\forall$ -type given on Slide 52 as the polymorphic type of lists represented via the iterator operations they determine. Note that from the perspective of this representation, the `nil` list is characterised as that list which when any  $\text{listIter } x' f$  is applied to it yields  $x'$ . This motivates the definition of the PLC expression `Nil` on Slide 52. Similarly for the constructor `Cons` for adding an element to the head of a list. It is not hard to prove the typings:

$$\begin{aligned} \{ \} &\vdash \text{Nil} : \forall \alpha (\alpha \text{ list}) \\ \{ \} &\vdash \text{Cons} : \forall \alpha (\alpha \rightarrow \alpha \text{ list} \rightarrow \alpha \text{ list}). \end{aligned}$$

As shown on Slide 54, an explicitly typed version of the operation of list iteration can be defined in PLC:  $\text{iter } \alpha \alpha' x' f$  satisfies the defining equations for an iteratively defined function (11) up to beta-conversion.

#### List iteration in PLC

---

$$\text{iter} \stackrel{\text{def}}{=} \Lambda \alpha, \alpha' (\lambda x' : \alpha', f : \alpha \rightarrow \alpha' \rightarrow \alpha' ( \\ \lambda \ell : \alpha \text{ list} (\ell \alpha' x' f)))$$

satisfies:

- $\vdash \text{iter} : \forall \alpha, \alpha' (\alpha' \rightarrow (\alpha \rightarrow \alpha' \rightarrow \alpha') \rightarrow \alpha \text{ list} \rightarrow \alpha')$
- $\text{iter } \alpha \alpha' x' f (\text{Nil } \alpha) =_{\beta} x'$
- $\text{iter } \alpha \alpha' x' f (\text{Cons } \alpha x \ell) =_{\beta} f x (\text{iter } \alpha \alpha' x' f \ell)$

ML	PLC
<code>datatype null = ;</code>	$null \stackrel{\text{def}}{=} \forall \alpha (\alpha)$
<code>datatype unit = Unit;</code>	$unit \stackrel{\text{def}}{=} \forall \alpha (\alpha \rightarrow \alpha)$
$\alpha_1 * \alpha_2$	$\alpha_1 * \alpha_2 \stackrel{\text{def}}{=} \forall \alpha ((\alpha_1 \rightarrow \alpha_2 \rightarrow \alpha) \rightarrow \alpha)$
<code>datatype (<math>\alpha_1, \alpha_2</math>)sum = Inl of <math>\alpha_1</math>   Inr of <math>\alpha_2</math>;</code>	$(\alpha_1, \alpha_2)sum \stackrel{\text{def}}{=} \forall \alpha ((\alpha_1 \rightarrow \alpha) \rightarrow (\alpha_2 \rightarrow \alpha) \rightarrow \alpha)$
<code>datatype nat = Zero   Succ of nat;</code>	$nat \stackrel{\text{def}}{=} \forall \alpha (\alpha \rightarrow (\alpha \rightarrow \alpha) \rightarrow \alpha)$
<code>datatype binTree = Leaf   Node of binTree * binTree;</code>	$binTree \stackrel{\text{def}}{=} \forall \alpha (\alpha \rightarrow (\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha)$

Figure 5: Some more algebraic datatypes

Booleans and lists are examples of “algebraic” datatypes, i.e. ones which can be specified (usually recursively) using products, sums and previously defined algebraic datatypes. Thus in Standard ML such a datatype (called *alg*, with constructors  $C_1, \dots, C_m$ ) might be declared by

$$\text{datatype } (\alpha_1, \dots, \alpha_n)alg = C_1 \text{ of } \tau_1 \mid \dots \mid C_m \text{ of } \tau_m$$

where the types  $\tau_1, \dots, \tau_m$  are built up from the type variables  $\alpha_1, \dots, \alpha_n$  and the type  $(\alpha_1, \dots, \alpha_n)alg$  itself, just using products and previously defined algebraic datatype constructors, but not, for example, using function types. Figure 5 gives some other algebraic datatypes and their representations as polymorphic types. In fact all algebraic datatypes can be represented in PLC: see (Girard 1989, Sections 11.3–5) for more details.

## 4.5 Exercises

**Exercise 4.5.1.** Give a proof inference tree for (8) in Example 4.1.1. Show that

$$\forall \alpha_1 (\alpha_1 \rightarrow \forall \alpha_2 (\alpha_2)) \rightarrow \text{bool list}$$

is another possible polymorphic type for  $\lambda f((f \text{ true}) :: (f \text{ nil}))$ .

**Exercise 4.5.2.** Show that if  $\Gamma \vdash M : \tau$  and  $\Gamma \vdash M : \tau'$  are both provable in the PLC type system, then  $\tau = \tau'$  (equality up to  $\alpha$ -conversion). [Hint: show that  $H \stackrel{\text{def}}{=} \{(\Gamma, M, \tau) \mid \Gamma \vdash M : \tau \ \& \ \forall \tau' (\Gamma \vdash M : \tau' \Rightarrow \tau = \tau')\}$  is closed under the axioms and rules on Slide 43.]

**Exercise 4.5.3.** In PLC, defining the expression  $\text{let } x = M_1 : \tau \text{ in } M_2$  to be an abbreviation for  $(\lambda x : \tau (M_2)) M_1$ , show that the typing rule

$$\frac{\Gamma \vdash M_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash M_2 : \tau_2}{\Gamma \vdash (\text{let } x = M_1 : \tau_1 \text{ in } M_2) : \tau_2} \quad \text{if } x \notin \text{dom}(\Gamma)$$

is admissible—in the sense that the conclusion is provable if the hypotheses are.

**Exercise 4.5.4.** The *erasure*,  $\text{erase}(M)$ , of a PLC expression  $M$  is the expression of the untyped lambda calculus obtained by deleting all type information from  $M$ :

$$\begin{aligned} \text{erase}(x) &\stackrel{\text{def}}{=} x \\ \text{erase}(\lambda x : \tau (M)) &\stackrel{\text{def}}{=} \lambda x (\text{erase}(M)) \\ \text{erase}(M_1 M_2) &\stackrel{\text{def}}{=} \text{erase}(M_1) \text{erase}(M_2) \\ \text{erase}(\Lambda \alpha (M)) &\stackrel{\text{def}}{=} \text{erase}(M) \\ \text{erase}(M \tau) &\stackrel{\text{def}}{=} \text{erase}(M). \end{aligned}$$

- (i) Find PLC expressions  $M_1$  and  $M_2$  satisfying  $\text{erase}(M_1) = \lambda x (x) = \text{erase}(M_2)$  such that  $\vdash M_1 : \forall \alpha (\alpha \rightarrow \alpha)$  and  $\vdash M_2 : \forall \alpha_1 ((\alpha_1 \rightarrow \forall \alpha_2 (\alpha_1)))$  are provable PLC typings.
- (ii) We saw in Example 4.2.6 that there is a closed PLC expression  $M$  of type  $\forall \alpha (\alpha) \rightarrow \forall \alpha (\alpha)$  satisfying  $\text{erase}(M) = \lambda f (f f)$ . Find some other closed, typeable PLC expressions with this property.
- (iii) [For this part you will need to recall, from the CST Part IB *Foundations of Functional Programming* course, some properties of beta reduction of expressions in the untyped lambda calculus.] A theorem of Girard says that if  $\vdash M : \tau$  is provable in the PLC type system, then  $\text{erase}(M)$  is strongly normalisable in the untyped lambda calculus, i.e. there are no infinite chains of beta-reductions starting from  $\text{erase}(M)$ . Assuming this result, exhibit an expression of the untyped lambda calculus which is not equal to  $\text{erase}(M)$  for any closed, typeable PLC expression  $M$ .

**Exercise 4.5.5.** Attack or defend the following statement.

“A typed programming language is *polymorphic* if a well-formed phrase of the language may have several different types.”

[Hint: consider the property of PLC given in the theorem on Slide 45.]

**Exercise 4.5.6.** Prove the various typings and beta-reductions asserted in Example 4.4.4.

**Exercise 4.5.7.** Prove the various typings asserted in Example 4.4.5 and the beta-conversions on Slide 54.

**Exercise 4.5.8.** For the polymorphic product type  $\alpha_1 * \alpha_2$  defined in the right-hand column of Figure 5, show that there are PLC expressions  $Pair$ ,  $fst$ , and  $snd$  satisfying:

$$\begin{aligned} \{ \} &\vdash Pair : \forall \alpha_1, \alpha_2 (\alpha_1 \rightarrow \alpha_2 \rightarrow (\alpha_1 * \alpha_2)) \\ \{ \} &\vdash fst : \forall \alpha_1, \alpha_2 ((\alpha_1 * \alpha_2) \rightarrow \alpha_1) \\ \{ \} &\vdash snd : \forall \alpha_1, \alpha_2 ((\alpha_1 * \alpha_2) \rightarrow \alpha_2) \\ fst \alpha_1 \alpha_2 (Pair \alpha_1 \alpha_2 x_1 x_2) &=_{\beta} x_1 \\ snd \alpha_1 \alpha_2 (Pair \alpha_1 \alpha_2 x_1 x_2) &=_{\beta} x_2. \end{aligned}$$

**Exercise\* 4.5.9.** Suppose that  $\tau$  is a PLC type with a single free type variable,  $\alpha$ . Suppose also that  $T$  is a closed PLC expression satisfying

$$\{ \} \vdash T : \forall \alpha_1, \alpha_2 ((\alpha_1 \rightarrow \alpha_2) \rightarrow (\tau[\alpha_1/\alpha] \rightarrow \tau[\alpha_2/\alpha])).$$

Define  $\iota$  to be the closed PLC type

$$\iota \stackrel{\text{def}}{=} \forall \alpha ((\tau \rightarrow \alpha) \rightarrow \alpha).$$

Show how to define PLC expressions  $R$  and  $I$  satisfying

$$\begin{aligned} \{ \} &\vdash R : \forall \alpha ((\tau \rightarrow \alpha) \rightarrow \iota \rightarrow \alpha) \\ \{ \} &\vdash I : \tau[\iota/\alpha] \rightarrow \iota \\ (R \alpha f)(I x) &\rightarrow^* f (T \iota \alpha (R \alpha f) x). \end{aligned}$$



## 5 Further Topics

### 5.1 Curry-Howard correspondence

The concept of “type” first arose in the logical foundations of mathematics. Russell (1903) circumvented the paradox he discovered in Frege’s set theory by stratifying the universe of untyped sets into levels, or types. Church (1940) proposed a typed, higher order logic based on functions rather than sets and which is capable of formalising large areas of mathematics. A version of this logic is the one underlying the HOL system (Gordon and Melham 1993). See (Lamport and Paulson 1999) for a stimulating discussion of the pros and cons of untyped logics (typically, set theory) versus typed logics for mechanising mathematics.

The interplay between logic and types has often been mediated by the correspondence between certain systems of constructive logic and certain typed lambda calculi first noted by the logician Curry in the 1950s and brought to the attention of computer scientists by the work of Howard in the 1980s. As a result, this connection between logic and type systems is often known as the *Curry-Howard correspondence* (and also as the “proposition as types” idea); it is sketched on Slide 55. To see how the Curry-Howard correspondence works, we will look at a specific instance, namely the correspondence between the PLC type system of Section 4 and the logic known as *second-order intuitionistic propositional calculus* (2IPC), which is defined on Slide 56.

Curry-Howard correspondence	
<i>Logic</i>	$\leftrightarrow$ <i>Type system</i>
propositions, $\phi$	types, $\tau$
(constructive) proofs, $p$	expressions, $M$
“ $p$ is a proof of $\phi$ ”	“ $M$ is an expression of type $\tau$ ”
simplification of proofs	reduction of expressions

Slide 55

**Second-order intuitionistic propositional calculus (2IPC)**

2IPC propositions:  $\boxed{\phi ::= p \mid \phi \rightarrow \phi \mid \forall p(\phi)}$ , where  $p$  ranges over an infinite set of propositional variables.

2IPC sequents:  $\boxed{\Phi \vdash \phi}$ , where  $\Phi$  is a finite set of 2IPC propositions and  $\phi$  is a 2IPC proposition.

$\Phi \vdash \phi$  is *provable* if it is in the set of sequents inductively generated by:

(Id)  $\Phi \vdash \phi$  if  $\phi \in \Phi$

$$(\rightarrow I) \frac{\Phi, \phi \vdash \phi'}{\Phi \vdash \phi \rightarrow \phi'}$$

$$(\rightarrow E) \frac{\Phi \vdash \phi \rightarrow \phi' \quad \Phi \vdash \phi}{\Phi \vdash \phi'}$$

$$(\forall I) \frac{\Phi \vdash \phi}{\Phi \vdash \forall p(\phi)} \text{ if } p \notin fv(\Phi)$$

$$(\forall E) \frac{\Phi \vdash \forall p(\phi)}{\Phi \vdash \phi[\phi'/p]}$$

**Slide 56**

Note that if we identify propositional variables with PLC's type variables, then 2IPC propositions *are* just PLC types. Every proof of a 2IPC sequent  $\Phi \vdash \phi$  can be described by a PLC expression  $M$  satisfying  $\Gamma \vdash M : \phi$ , one we have fixed a labelling  $\Gamma = \{x_1 : \phi_1, \dots, x_n : \phi_n\}$  of the propositions in  $\Phi = \{\phi_1, \dots, \phi_n\}$  with variables  $x_1, \dots, x_n$ .  $M$  is built up by recursion on the structure of the proof of the sequent using the following transformations:

$$\begin{array}{ll}
 \text{(Id) } \Phi, \phi \vdash \phi & \mapsto \text{(id) } \bar{x} : \Phi, x : \phi \vdash x : \phi \\
 (\rightarrow I) \frac{\Phi, \phi \vdash \phi'}{\Phi \vdash \phi \rightarrow \phi'} & \mapsto \text{(fn) } \frac{\bar{x} : \Phi, x : \phi \vdash M : \phi'}{\bar{x} : \Phi \vdash \lambda x : \phi (M) : \phi \rightarrow \phi'} \\
 (\rightarrow E) \frac{\Phi \vdash \phi \rightarrow \phi' \quad \Phi \vdash \phi}{\Phi \vdash \phi'} & \mapsto \text{(app) } \frac{\bar{x} : \Phi \vdash M_1 : \phi \rightarrow \phi' \quad \bar{x} : \Phi \vdash M_2 : \phi}{\bar{x} : \Phi \vdash M_1 M_2 : \phi'} \\
 (\forall I) \frac{\Phi \vdash \phi}{\Phi \vdash \forall p(\phi)} & \mapsto \text{(gen) } \frac{\bar{x} : \Phi \vdash M : \phi}{\bar{x} : \Phi \vdash \Lambda p (M) : \forall p(\phi)} \\
 (\forall E) \frac{\Phi \vdash \forall p(\phi)}{\Phi \vdash \phi[\phi'/p]} & \mapsto \text{(spec) } \frac{\bar{x} : \Phi \vdash M : \forall p(\phi)}{\bar{x} : \Phi \vdash M \phi' : \phi[\phi'/p]}
 \end{array}$$

This is illustrated on Slide 57. The example on that slide uses the fact that the logical

operation of conjunction can be defined in 2IPC. Slide 58 gives some other logical operators that are definable in 2IPC. Compare it with Figure 5: the richness of PLC for expressing datatypes is mirrored under the Curry-Howard correspondence by the richness of 2IPC for expressing logical constructions.

**A 2IPC proof**

---


$$\frac{}{\{p \ \& \ q, p, q\} \vdash p} \text{ (Id)}$$

$$\frac{}{\{p \ \& \ q, p\} \vdash q \rightarrow p} \text{ (}\rightarrow\text{I)}$$

$$\frac{}{\{p \ \& \ q\} \vdash \forall r ((p \rightarrow q \rightarrow r) \rightarrow r)} \text{ (Id)}$$

$$\frac{}{\{p \ \& \ q\} \vdash (p \rightarrow q \rightarrow p) \rightarrow p} \text{ (}\rightarrow\text{E)}$$

$$\frac{}{\{p \ \& \ q\} \vdash p} \text{ (}\rightarrow\text{I)}$$

$$\frac{}{\{\} \vdash p \ \& \ q \rightarrow p} \text{ (}\forall\text{I)}$$

$$\frac{}{\{\} \vdash \forall q (p \ \& \ q \rightarrow p)} \text{ (}\forall\text{I)}$$

$$\frac{}{\{\} \vdash \forall p, q (p \ \& \ q \rightarrow p)} \text{ (}\forall\text{I)}$$

where  $p \ \& \ q$  is an abbreviation for  $\forall r ((p \rightarrow q \rightarrow r) \rightarrow r)$ .

The PLC expression corresponding to this proof is:

$$\Lambda p, q (\lambda z : p \ \& \ q (z p (\lambda x : p, y : q (x))))).$$

Slide 57

The Curry-Howard correspondence gives us a different perspective on the typing judgement  $\Gamma \vdash M : \sigma$ , outlined on Slide 59. As well as the undecidability result mentioned on that slide, it should be noted that 2IPC is a *constructive* rather than a *classical* logic, in the sense that the *Law of Excluded Middle* is not provable in 2IPC. More precisely (using the encoding of disjunction and negation given on Slide 58), there is no proof of  $\forall p (p \vee \neg p)$ ; in other words, there is no PLC expression  $M$  satisfying  $\{\} \vdash M : \forall p (p \vee \neg p)$ . (This can be proved using the technique developed in the Tripo question 13 on paper 9 in 2000).

The Curry-Howard correspondence cuts both ways: in one direction it has proved very helpful to use lambda terms as notations for proofs in mechanised proof assistants; in the other it has helped to suggest new type systems for programming and specification languages. Two examples of the second kind of application are the transfer of ideas from Girard's *linear logic* (Girard 1987) into systems of *linear types* in usage analyses (see Chirimar, Gunter, and Riecke (1996), for example); and the use type systems based on *modal logics* for analysing *partial evaluation* and *run-time code generation* (Davis and Pfenning 1996).

### Logical operations definable in 2IPC

---

- *Truth*:  $true \stackrel{\text{def}}{=} \forall p (p \rightarrow p)$ .
- *Falsity*:  $false \stackrel{\text{def}}{=} \forall p (p)$ .
- *Conjunction*:  $\phi \& \phi' \stackrel{\text{def}}{=} \forall p ((\phi \rightarrow \phi' \rightarrow p) \rightarrow p)$   
(where  $p \notin fv(\phi, \phi')$ ).
- *Disjunction*:  $\phi \vee \phi' \stackrel{\text{def}}{=} \forall p ((\phi \rightarrow p) \rightarrow (\phi' \rightarrow p) \rightarrow p)$  (where  
 $p \notin fv(\phi, \phi')$ ).
- *Negation*:  $\neg\phi \stackrel{\text{def}}{=} \phi \rightarrow false$ .
- *Existential quantification*:  $\exists p (\phi) \stackrel{\text{def}}{=} \forall p' (\forall p (\phi \rightarrow p') \rightarrow p')$   
(where  $p' \notin fv(\phi, p)$ ).

Slide 58

### Type-inference versus proof search

---

*Type-inference*: “given  $\Gamma$  and  $M$ , is there a type  $\sigma$  such that  
 $\Gamma \vdash M : \sigma$ ?”

(For PLC/2IPC this is decidable.)

*Proof-search*: “given  $\Gamma$  and  $\sigma$ , is there a proof term  $M$  such that  
 $\Gamma \vdash M : \sigma$ ?”

(For PLC/2IPC this is undecidable.)

Slide 59

## 5.2 Dependent types

Consider programming a function *taut* that takes in  $n$ -ary boolean operations (in “curried” form)

$$f : \underbrace{bool \rightarrow bool \rightarrow \cdots \rightarrow bool}_{n \text{ arguments}} \rightarrow bool$$

and returns *true* if  $f$  is a tautology, i.e. has value *true* for all of its  $2^n$  possible arguments, and returns *false* otherwise. One might try to program *taut* in Standard ML as on Slide 60. This is algorithmically correct, but does not type-check in ML. Why? Intuitively, the type of *taut*  $n$  for each natural number  $n = 0, 1, 2, \dots$  is the type  $bool \rightarrow^n bool$  of “ $n$ -ary curried boolean functions” defined (by induction on  $n$ ) on Slide 60. Thus *taut* is really a *dependently typed function*—the type of its result depends on the *value* of the argument supplied to it—and so it is rejected by the ML type-checker.

In general a *dependent type* is a family of types indexed by individual values of a datatype. (In the above example the family of types  $bool \rightarrow^n bool$  is indexed by values  $n$  of a type of numbers.) Typing rules for dependent function types are given on Slide 61. Note that the usual typing rules for function types  $\tau \rightarrow \tau'$  are the special case where the type  $\tau'$  has no dependency on values.

### A tautology checker

```
fun taut n f = if n = 0 then f else
              (taut(n - 1)(f true))
              andalso (taut(n - 1)(f false))
```

Defining types

$$\begin{cases} bool \rightarrow^0 bool & \stackrel{\text{def}}{=} bool \\ bool \rightarrow^{n+1} bool & \stackrel{\text{def}}{=} bool \rightarrow (bool \rightarrow^n bool) \end{cases}$$

then *taut*  $n$  has type  $bool \rightarrow^n bool$ , i.e. the result type of the function *taut* depends upon the value of its argument.

**Dependent function types**  $\prod x : \sigma(\sigma'(x))$

---


$$\frac{\Gamma, x : \sigma \vdash M : \sigma'(x)}{\Gamma \vdash \lambda x : \sigma (M) : \prod x : \sigma(\sigma'(x))}$$
  

$$\frac{\Gamma \vdash M : \prod x : \sigma(\sigma'(x)) \quad \Gamma \vdash M' : \sigma}{\Gamma \vdash M M' : \sigma'(M')}$$

Slide 61

Type systems featuring dependent types are able to express much more refined properties of programs than ones without this feature. So why don't they get used in programming languages? The answer lies in the fact that type-checking with dependent types naturally involves checking equalities between the data values upon which the types depend; in a Turing-powerful language such value-equality tends to be undecidable and hence static type-checking becomes impossible. How to get round this problem is an active area of research. For example the Cayenne language (Augustsson 1998) takes a general-purpose, pragmatic, but incomplete approach; whereas (Xi and Pfenning 1998) uses dependent types for a specific task, namely static elimination of run-time array bound checking, by restricting dependency to a language of integer expressions where checking equality reduces to solving linear programming problems. In machine-assisted reasoning systems, decidability of type-checking is not such an important issue (since the user has to guide the system to proofs of other kinds of undecidable property anyway). Type theories with dependent types have been used extensively in computer systems for formalising mathematics, for proof construction, and for checking the correctness of proofs. In this respect Martin-Löf's *intuitionistic type theory* (which first popularised the notion of "dependent type") has been highly influential: see Nordström, Petersson, and Smith 1990 for an introduction to it.

### 5.3 Current areas of research

The study of types forms a very vigorous area of computer science research, both for computing theory and in the application of theory to practice. This course has aimed

at reasonably detailed coverage of a few selected topics, centred around the notion of polymorphism in programming languages. To finish, I enumerate some other topics which are of interest in the development of the theory and application of type systems in computer science, together with some pointers to the literature.<sup>1</sup>

**Concurrency and distributed systems** The typing of languages involving concurrent threads of computation and associated notions of mobility and distribution is so current a topic of research that it is difficult to give pointers to well-digested accounts. A basic motivation for the use of type systems here is the same as for more traditional languages: to avoid unsafe or undesirable behaviour via static checks. However the kinds of unsafe behaviour are now much more complicated, or at least, less well-understood. For example, there are type systems which can ensure that locks are used correctly (Flanagan and Abadi 1999); and in distributed (and possibly mobile) settings, there are a number of type systems which further classify values by the place at which they reside in a network and/or the resources to which they have access—(Hennessy and Riely 2002), for example. This is not only a very interesting and a very challenging area, but also one of rather immediate practical concern.

**Security** Extending the idea mentioned in the Introduction of compilers ensuring whole-language safety through static type-checking, type systems are the foundation of several systems for deciding whether compiled code obtained from a potentially untrustworthy source is safe to execute. Both Sun's Java Virtual Machine (JVM) and Microsoft's .NET Common Language Run-time (CLR) include type-checkers (or *verifiers*), which are run before compiled code is executed. A nice overview of the internals of JVM verification has been written by Leroy (2003). The correctness of higher-level security operations (such as the management of explicit permissions to perform potentially-unsafe operations) relies on the typability of any untrusted code which will be allowed to execute. Type systems are also being used to formalise and check properties which are more security-specific. One line of research classifies the inputs and outputs of a program as either high-security or low-security. A type system can then be used to ensure that high-security information cannot affect low-security outputs (imagine downloading a banking application which has to communicate over the network to retrieve current tax rates, etc., but which you wish to be sure will not leak any of your personal information). See Volpano, Smith, and Irvine (1996) for example.

**Low-level languages** Traditionally, sophisticated type systems are used in *high-level* programming languages, i.e. ones that abstract away from the low-level details of machine services. Low-level languages such as C or assembler have made do with either no types or type systems which are both inexpressive and unsafe. In particular, high-level, safe, typed languages have been compiled into low-level, untyped, unsafe ones. Recent years have seen a great deal of research activity on typed assembly language (TAL) and type-preserving compilation (Morrisett, Walker, Crary, and Glew 1999). The idea here is to compile an ML program, for example, into a typed assembly language program in such a way that checking the types on the assembly code gives the same safety guarantees as one

---

<sup>1</sup>Some of this material is taken from Nick Benton's 2003 version of these notes.

gets from the ML type system with respect to a high-level operational semantics for ML. This is clearly similar to the use of bytecode verification discussed above; the difference is that the intermediate languages of the JVM and CLR are fairly high-level, so verification is similar to type checking Java or C<sup>#</sup> source (and therefore not too difficult) but the interpreter or JIT compiler which runs *after* the verifier has to be part of the “trusted computing base” (TCB). In the type-preserving compilation approach, the types (and hence the type checker) for the low-level code tend to be rather more complex—indeed they require the use of the kind of “impredicative” polymorphism we studied in Section 4); but only the type-checker need be part of the TCB: bugs or maliciousness in the compiler are either benign or yield TAL programs which fail to typecheck; see also work on *proof-carrying code*, such as Necula (1997). An active area of related work concerns designing C-like languages with safe type systems (and which may be compiled to TAL). Examples include CCured (Necula, McPeak, and Weimer 2002) and Cyclone (Jim, Morrisett, Grossman, Hicks, Cheney, and Wang 2002). These languages vary in their degree of compatibility with legacy C code and in the extent to which safety is ensured by static, rather than dynamic, checks.

**Database query languages** Schema for relational databases or DTDs for XML documents are a kind of type. The last few years have seen a great deal of research on integrating types for these sorts of data into the type systems of (new and existing) programming languages. This has many potential advantages, such as being able to check statically that a program which transforms XML documents always produces valid output (e.g. well-formed HTML) from valid input. Languages such as XDuce, CDuce (Benzaken, Castagna, and Frisch 2003) and Xtatic have types which can express regular expressions over tree-structured data; language inclusion thus induces a subtype relation, and type inference and checking involve computing with these regular expressions.

## References

- Aho, A. V., R. Sethi, and J. D. Ullman (1986). *Compilers. Principles, Techniques, and Tools*. Addison Wesley.
- Augustsson, L. (1998). Cayenne—a language with dependent types. In *ACM SIGPLAN International Conference on Functional Programming, ICFP 1998, Baltimore, Maryland, USA*. ACM Press.
- Benzaken, V., G. Castagna, and A. Frisch (2003). CDuce: An XML-centric general-purpose language. In *Proceedings of the 8th ACM International Conference on Functional Programming (ICFP'03), Uppsala, Sweden*, pp. 51–64.
- Cardelli, L. (1987). Basic polymorphic typechecking. *Science of Computer Programming* 8, 147–172.
- Cardelli, L. (1997). Type systems. In *CRC Handbook of Computer Science and Engineering*, Chapter 103, pp. 2208–2236. CRC Press.
- Chirimar, J., C. A. Gunter, and J. G. Riecke (1996). Reference counting as a computational interpretation of linear logic. *Journal of Functional Programming* 6(2), 195–244.
- Church, A. (1940). A formulation of the simple theory of types. *Journal of Symbolic Logic* 5, 56–68.
- Damas, L. and R. Milner (1982). Principal type schemes for functional programs. In *Proc. 9th ACM Symposium on Principles of Programming Languages*, pp. 207–212.
- Davis, R. and F. Pfenning (1996). A modal analysis of staged computation. In *ACM Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida*, pp. 258–270. ACM Press.
- Flanagan, C. and M. Abadi (1999). Types for safe locking. In *8th European Symposium on Programming (ESOP '99)*, Lecture Notes in Computer Science, pp. 91–108. Springer-Verlag.
- Girard, J.-Y. (1972). *Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieur*. Ph. D. thesis, Université Paris VII. Thèse de doctorat d'état.
- Girard, J.-Y. (1987). Linear logic. *Theoretical Computer Science* 50, 1–101.
- Girard, J.-Y. (1989). *Proofs and Types*. Cambridge University Press. Translated and with appendices by Y. Lafont and P. Taylor.
- Gordon, M. J. C. and T. F. Melham (1993). *Introduction to HOL. A theorem proving environment for higher order logic*. Cambridge University Press.
- Harper, R. (1994). A simplified account of polymorphic references. *Information Processing Letters* 51, 201–206.
- Harper, R. and C. Stone (1997). An interpretation of Standard ML in type theory. Technical Report CMU-CS-97-147, Carnegie Mellon University, Pittsburgh, PA.
- Hennessy, M. and J. Riely (2002). Resource access control in systems of mobile agents. *Information and Computation* 173, 82–120.

- Hindley, J. R. (1969). The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society* 146, 29–40.
- Jim, T., G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang (2002). Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, pp. 275–288.
- Lampert, L. and L. C. Paulson (1999). Should your specification language be typed? *ACM Transactions on Programming Languages and Systems* 21(3), 502–526.
- Leroy, X. (2003). Java bytecode verification: Algorithms and formalizations. *Journal of Automated Reasoning* 30, 235–269.
- Mairson, H. G. (1990). Deciding ML typability is complete for deterministic exponential time. In *Proc. 17th ACM Symposium on Principles of Programming Languages*, pp. 382–401.
- Milner, R., M. Tofte, and R. Harper (1990). *The Definition of Standard ML*. MIT Press.
- Milner, R., M. Tofte, R. Harper, and D. MacQueen (1997). *The Definition of Standard ML (Revised)*. MIT Press.
- Mitchell, J. C. (1996). *Foundations for Programming Languages*. Foundations of Computing series. MIT Press.
- Morrisett, G., D. Walker, K. Crary, and N. Glew (1999). From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems* 21(3), 528–569.
- Necula, G. (1997). Proof-carrying code. In *24th Annual ACM Symposium on Principles of Programming Languages (POPL)*. ACM Press.
- Necula, G., S. McPeak, and W. Weimer (2002). CCured: Type-safe retrofitting of legacy code. In *29th Annual ACM Symposium on Principles of Programming Languages (POPL)*, pp. 128–139.
- Nordström, B., K. Petersson, and J. M. Smith (1990). *Programming in Martin-Löf's Type Theory*. Oxford University Press.
- Pierce, B. C. (2002). *Types and Programming Languages*. MIT Press.
- Rémy, D. (2002). Using, understanding, and unravelling the ocaml language: From practice to theory and vice versa. In G. Barthe, P. Dybjer, and J. Saraiva (Eds.), *Applied Semantics, Advanced Lectures*, Volume 2395 of *Lecture Notes in Computer Science, Tutorial*, pp. 413–537. Springer-Verlag. International Summer School, APPSEM 2000, Caminha, Portugal, September 9–15, 2000.
- Reynolds, J. C. (1974). Towards a theory of type structure. In *Paris Colloquium on Programming*, Volume 19 of *Lecture Notes in Computer Science*, pp. 408–425. Springer-Verlag, Berlin.
- Robinson, J. A. (1965). A machine oriented logic based on the resolution principle. *Jour. ACM* 12, 23–41.
- Russell, B. (1903). *The Principles of Mathematics*. Cambridge.
- Rydeheard, D. E. and R. M. Burstall (1988). *Computational Category Theory*. Series in Computer Science. Prentice Hall International.

- Strachey, C. (1967). Fundamental concepts in programming languages. Lecture notes for the International Summer School in Computer Programming, Copenhagen.
- Tofte, M. (1990). Type inference for polymorphic references. *Information and Computation* 89, 1–34.
- Tofte, M. and J.-P. Talpin (1997). Region-based memory management. *Information and Computation* 132(2), 109–176.
- Volpano, D., G. Smith, and C. Irvine (1996). A sound type system for secure flow analysis. *Journal of Computer Security* 4(3), 167–187.
- Wells, J. B. (1994). Typability and type-checking in the second-order  $\lambda$ -calculus are equivalent and undecidable. In *Proceedings, 9th Annual IEEE Symposium on Logic in Computer Science*, Paris, France, pp. 176–185. IEEE Computer Society Press.
- Wright, A. K. (1995). Simple imperative polymorphism. *LISP and Symbolic Computation* 8, 343–355.
- Wright, A. K. and M. Felleisen (1994). A syntactic approach to type soundness. *Information and Computation* 115, 38–94.
- Xi, H. and F. Pfenning (1998). Eliminating array bound checking through dependent types. In *Proc. ACM-SIGPLAN Conference on Programming Language Design and Implementation, Montreal, Canada*, pp. 249–257. ACM Press.



## Lectures Appraisal Form

If lecturing standards are to be maintained where they are high, and improved where they are not, it is important for the lecturers to receive feedback about their lectures. Consequently, we would be grateful if you would complete this questionnaire, and either return it to the lecturer in question, or to Student Administration, Computer Laboratory, William Gates Building. Thank you.

1. Name of Lecturer: Prof. Andrew M. Pitts
2. Title of Course: CST Part II Types
3. How many lectures have you attended in this series so far? .....  
Do you intend to go to the rest of them? Yes/No/Series finished
4. What do you expect to gain from these lectures? (Underline as appropriate)  
Detailed coverage of selected topics *or* Advanced material  
Broad coverage of an area *or* Elementary material  
Other (please specify)
5. Did you find the content: (place a vertical mark across the line)  
Too basic ----- Too complex  
Too general ----- Too specific  
Well organised ----- Poorly organised  
Easy to follow ----- Hard to follow
6. Did you find the lecturer's delivery: (place a vertical mark across the line)  
Too slow ----- Too fast  
Too general ----- Too specific  
Too quiet ----- Too loud  
Halting ----- Smooth  
Monotonous ----- Lively  
Other comments on the delivery:
7. Was a satisfactory reading list provided? Yes/No  
How might it be improved.
8. Apart from the recommendations suggested by your answers above, how else might these lectures be improved? Do any specific lectures in this series require attention? (Continue overleaf if necessary)