Q

Lecture Notes on

Types

for Part II of the Computer Science Tripos

Dr Dominic Orchard
(previously Prof. Andrew M. Pitts)

University of Cambridge
Computer Laboratory

Contents

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*. 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

There is an exercise sheet at the end of these notes. A list of past Tripos questions back to 1993 that are relevant to the current course is available from the course web page.

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 addition 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 et al. (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/`⟩), which also contains pointers to some other useful material.

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

Types

8 lectures for CST Part II by Dominic Orchard
(previously Andrew Pitts)

⟨www.cl.cam.ac.uk/teaching/1415/Types/⟩

> *"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

Slide 1

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. In a similar way, we can use types to rule out paradoxical or non-sensical programs. We will return to the interplay between types in programming languages and types in logic 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 2, 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*

Type systems are one of the most important channels by which developments in theoretical computer science get applied in programming language design and software verification.

Slide 2

Here are some ways (summarised on Slide 3) 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 3

Detecting errors Experience shows that a significant proportion of programming mistakes (such as trying to divide 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 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 can be used by 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 4

Whole-language safety Slide 4 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 (via the use of casting and `void`).

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 5

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 et al. (1997) is the shining example of a full-scale language possessing a complete such specification and whose *type soundness/safety* (cf. Slide 5, Slide 9) has been subject to proof.

Typical type system 'judgement'

is a relation between typing environments (Γ), program phrases (M) and type expressions (τ) 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 Γ, then M has type τ.*
E.g.

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

is a valid typing judgement about ML.

Slide 6

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 7

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 6. 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 7.

Once we have formalised a particular type system, we are in a position to *prove* results about *type soundness* (Slide 5) and the notions of *type checking*, *typeability* and *type inference* described on Slide 8. 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 (progress and type preservation, cf. Slide 9). In this course we look at more involved examples revolving around the notion of 'parametric polymorphism', to which we turn to in the next section.

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 Γ, M, and τ, is $\Gamma \vdash M : \tau$ derivable in the type system?
- Typeability problem: given Γ and M, is there any τ 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 τ for each Γ and M (or failing, if there is none).

Slide 8

A definition for type soundness, progress & preservation

Recall from CST Part IB Semantics:

- Progress:

$$\forall s.\ (\Gamma \vdash e : \tau)\ \wedge\ (dom(\Gamma) \subseteq dom(s))$$
$$\Rightarrow value(e)\ \vee\ \exists e', s'. \langle e, s\rangle \rightarrow \langle e', s'\rangle$$

- Preservation:

$$(\Gamma \vdash e : \tau)\ \wedge\ (\langle e, s\rangle \rightarrow \langle e', s'\rangle)\ \Rightarrow\ \Gamma \vdash e' : \tau'$$

Slide 9

Course outline

- **Introduction**. The role of type systems in programming languages. Formalizing type systems. [1 lecture]
- **ML polymorphism**. ML-style polymorphism. Principal type schemes and type inference. [2 lectures]
- **Polymorphic reference types**. The pitfalls of combining ML polymorphism with reference types. [1 lecture]
- **Polymorphic lambda calculus**. Syntax and reduction semantics. Examples of datatypes definable in the polymorphic lambda calculus. Applications. [2 lectures]
- **Further topics**. The Curry-Howard correspondence (types-as-formulae, terms-as-proofs) as a source of type systems. Dependent types. [2 lectures]

Slide 10

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 11. Here we will concentrate on parametric polymorphism, also known as 'generics'. 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 Haskell, Java and C#. 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

$$\begin{array}{llcl} \texttt{fun} & \mathit{length}\,\texttt{nil} & = & 0 \\ \mid & \mathit{length}\,(x :: xs) & = & 1 + (\mathit{length}\,xs) \end{array}$$

has type $\tau\ \mathit{list} \to \mathit{int}$ for all types τ.)

Slide 11

Type variables and type schemes in Mini-ML

To formalise statements like

'*length has type* τ *list* $\to$ *int*, *for all types* τ'

it is natural to introduce type variables α (i.e. variables for which types may be substituted) and write

$$\mathit{length} : \forall\,\alpha\,(\alpha\;\mathit{list} \to \mathit{int}).$$

$\forall\,\alpha\,(\alpha\;\mathit{list} \to \mathit{int})$ is an example of a type scheme.

Slide 12

2.1 An ML type system

As indicated on Slide 12, to formalise parametric polymorphism, we have to introduce *type variables*. An interactive ML system will just display $\alpha\;\mathit{list} \to \mathit{int}$ as the type of the *length* function (Slide 11), leaving the universal quantification over α implicit. However, when it comes to formalising the ML type system (as in the definition of the Standard ML 'static semantics' in (Milner et al., 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 on Slide 13. The expression $(f\,\mathtt{true}) :: (f\,\mathtt{nil})$ has type $\mathit{bool}\;\mathit{list}$, given some assumption about the type of the variable f. Two possible such assumptions are shown on Slide 14. 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 15.

Polymorphism of **let**-bound variables in ML

For example in

$$\mathtt{let}\; f = \lambda x(x)\;\mathtt{in}\;(f\,\mathtt{true}) :: (f\,\mathtt{nil})$$

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

- in $(f\,\mathtt{true})$, f has type $\mathit{bool} \to \mathit{bool}$
- in $(f\,\mathtt{nil})$, f has type $\mathit{bool}\;\mathit{list} \to \mathit{bool}\;\mathit{list}$

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

Slide 13

- 'Parametric' polymorphism:

  if $f : \forall\,\alpha\,(\alpha \to \alpha)$,
  then $(f\,\texttt{true}) :: (f\,\texttt{nil}) : \mathit{bool\ list}$.

  Behaviour is uniform for different type instantiations– does not depend on the type.

- 'Ad hoc' polymorphism (overloading):

  if $f : \mathit{bool} \to \mathit{bool}$
  and $f : \mathit{bool\ list} \to \mathit{bool\ list}$,
  then $(f\,\texttt{true}) :: (f\,\texttt{nil}) : \mathit{bool\ list}$.

  Type-dependent behaviour.

Slide 14

Mini-ML types and type schemes

Types	τ	$::=$	α	type variable
		$\mid$	bool	type of booleans
		$\mid$	$\tau \to \tau$	function type
		$\mid$	$\tau\ \mathit{list}$	list type

where α ranges over a fixed, countably infinite set TyVar.

Type Schemes $\sigma \ ::= \ \forall\,A\,(\tau)$

where A ranges over finite subsets of the set TyVar.

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

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

Slide 15

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 set of types as a subset of the set of type schemes by identifying the type τ with the type scheme $\forall\,\{\,\}\,(\tau)$.**

(ii) Any occurrences in τ 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 τ, but which are not in the finite set A. (For example the set of free type variables of $\forall\,\alpha\,(\alpha \to \alpha')$ is $\{\alpha'\}$.) We call a type scheme $\forall\,A\,(\tau)$ *closed* if it has no free type variables, that is, if A contains all the type variables occurring in τ. 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, $\forall\,\alpha\,(\alpha \to \alpha')$ and $\forall\,\alpha''\,(\alpha'' \to \alpha')$ 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 (i) we identify Mini-ML types τ with trivial type schemes $\forall\,\{\,\}\,(\tau)$, so we sometimes write

$$ftv(\tau)$$

for the finite set of type variables occurring in τ (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, $\alpha \to \forall\,\alpha'\,(\alpha')$ is neither a well-formed Mini-ML type nor a well-formed Mini-ML type scheme.[1] 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 $\sigma = \forall\,\alpha_1, \ldots, \alpha_n\,(\tau')$ generalises a type τ, and write $\boxed{\sigma \succ \tau}$ if τ can be obtained from the type τ' by simultaneously substituting some types τ_i for the type variables α_i $(i = 1, \ldots, n)$:

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

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

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

Slide 16

Slide 16 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 1. Some simple examples of generalisation:

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

[1]The step of making type schemes first class types will be taken in Section 4.

However

$$\forall\,\alpha\,(\alpha \to \alpha) \;\not\succ\; (\alpha' \to \alpha') \to \alpha'.$$

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

$$\forall\,\alpha_1, \alpha_2\,(\alpha_1 \to \alpha_2) \;\succ\; \alpha\; list \to bool.$$

However

$$\forall\,\alpha_1\,(\alpha_1 \to \alpha_2) \;\not\succ\; \alpha\; list \to bool$$

because α_2 is a free type variable in the type scheme $\forall\,\alpha_1\,(\alpha_1 \to \alpha_2)$ and so cannot be substituted for during specialisation.

Mini-ML typing judgement

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

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

Slide 17

Mini-ML expressions, M

$::=$	x	variable
$\mid$	`true`	boolean values
$\mid$	`false`	
$\mid$	`if` M `then` M `else` M	conditional
$\mid$	$\lambda x(M)$	function abstraction
$\mid$	$M\,M$	function application
$\mid$	`let` $x = M$ `in` M	local declaration
$\mid$	`nil`	nil list
$\mid$	$M :: M$	list cons
$\mid$	`case` M `of` `nil` $=>$ $M \mid x :: x => M$	case expression

Slide 18

Slide 17 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 18. We use a non-standard syntax compared with the definition in Milner et al. (1997). For example we write $\lambda x(M)$ for `fn x => M` and `let` $x = M_1$ `in` M_2 for `let val x = M1 in M2 end`. (Furthermore we will call the symbol 'x' occurring in these expressions a *variable* rather than a '(value) identifier'.) The axioms and rules inductively generating the Mini-ML typing relation for these expressions are given on Slides 19–21.

Note the following points about the type system defined on Slides 19–21.

(i) As usual, any free occurrences of x in M become bound in $\lambda x(M)$. In the expression `let` $x = M_1$ `in` M_2, any free occurrences of the variable x in M_2 become bound in the `let`-expression. Similarly, in the expression `case` M_1 `of` `nil` $=> M_2 \mid x_1 :: x_2 => M_3$, any free occurrences of the variables x_1 and x_2 in M_3 become bound in the `case`-expression. **We identify expressions up to alpha-conversion of bound variables.** For example, `let` $x = \lambda x(x)$ `in` $x\,x$ and `let` $f = \lambda x(x)$ `in` $f\,f$ determine the same alpha-equivalence class and will be used interchangeably.

(ii) Given a type environment Γ we write $\Gamma, x : \sigma$ to indicate a typing environment with domain $dom(\Gamma) \cup \{x\}$, mapping x to σ and otherwise mapping like Γ. When we use this notation it will almost always be the case that $x \notin dom(\Gamma)$: cf. rules (fn), (let) and (case). Note also that side conditions such as $x \notin dom(\Gamma)$ in these rules can often be satisfied by suitably renaming bound variables to be fresh (relying upon the previous point).

(iii) In rule (fn) we use $\Gamma, x : \tau_1$ as an abbreviation for $\Gamma, x : \forall\,\{\,\}\,(\tau_1)$. Similarly, in rule (case), $\Gamma, x_1 : \tau_1, x_2 : \tau_1\ list$ really means $\Gamma, x_1 : \forall\,\{\,\}\,(\tau_1), x_2 : \forall\,\{\,\}\,(\tau_1\ list)$. (Recall that by definition, a typing environment has to map variables to type schemes, rather than to types.)

(iv) In rule (let) the notation $ftv(\Gamma)$ means the set of all type variables occurring free in some type scheme assigned in Γ. (For example, if $\Gamma = \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$, then $ftv(\Gamma) = ftv(\sigma_1) \cup \cdots \cup ftv(\sigma_n)$.) Thus the set $A = ftv(\tau) - ftv(\Gamma)$ used in that rule consists of all type variables in τ that do not occur freely in any type scheme assigned in Γ.

Mini-ML type system, I

$$\Gamma \vdash x : \tau \quad \text{if } (x : \sigma) \in \Gamma \quad \text{and } \sigma \succ \tau \qquad (\text{var} \succ)$$

$$\Gamma \vdash B : bool \quad \text{if } B \in \{\texttt{true}, \texttt{false}\} \qquad (\text{bool})$$

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

Slide 19

Mini-ML type system, II

$$\Gamma \vdash \texttt{nil} : \tau\ list \qquad (\text{nil})$$

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

$$\frac{\begin{array}{c} \Gamma \vdash M_1 : \tau_1\ list \\ \Gamma \vdash M_2 : \tau_2 \qquad \Gamma, x_1 : \tau_1, x_2 : \tau_1\ list \vdash M_3 : \tau_2 \end{array}}{\Gamma \vdash \text{case } M_1 \text{ of } \texttt{nil} \Rightarrow M_2 \mid x_1 :: x_2 \Rightarrow M_3 : \tau_2} \quad \begin{array}{l} \text{if } x_1, x_2 \notin dom(\Gamma) \\ \wedge\ \ x_1 \neq x_2 \\ \qquad (\text{case}) \end{array}$$

Slide 20

Mini-ML type system, III

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

$$\frac{\Gamma \vdash M_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash M_2 : \tau_1}{\Gamma \vdash M_1\, M_2 : \tau_2} \qquad \text{(app)}$$

$$\frac{\Gamma \vdash M_1 : \tau \qquad \Gamma, x : \forall A\,(\tau) \vdash M_2 : \tau'}{\Gamma \vdash \mathtt{let}\, x = M_1\, \mathtt{in}\, M_2 : \tau'} \quad \begin{array}{l} \text{if} \quad x \notin dom(\Gamma) \\ \wedge \quad A = ftv(\tau) \end{array} \qquad \text{(let)}$$

Slide 21

Assigning type schemes to Mini-ML expressions

Given a type scheme $\sigma = \forall A\,(\tau)$, write

$$\boxed{\Gamma \vdash M : \sigma}$$

if $A = ftv(\tau) - ftv(\Gamma)$ and $\Gamma \vdash M : \tau$ is derivable from the axiom and rules on Slides 19–21.

When $\Gamma = \{\,\}$ we just write $\boxed{\vdash M : \sigma}$ for $\{\,\} \vdash M : \sigma$ and say that the (necessarily closed—see Exercise 2) expression M is *typeable* in Mini-ML with type scheme σ.

Slide 22

As usual, the axioms and rules on Slides 19–21 are schematic: Γ, M, and τ stand for *any* well-formed type environment, expression, and type. The axiom and rules are used to inductively generate the *typing relation*—a subset of all possible triples $\Gamma \vdash M : \tau$. We say that a particular triple $\Gamma \vdash M : \tau$ is *derivable* (or *provable*, or *valid*) in the type system if there is a proof of it using the axioms and rules. Thus the typing relation consists of exactly those triples for which there is such a proof.

In fact we often use the typing relation to assign not just types, but also type schemes to Mini-ML expressions, as described on Slide 22.

Example 2. We verify that the example of polymorphism of `let`-bound variables given on Slide 13 has the type claimed there, i.e. that the following holds.

$$\vdash \mathtt{let}\, f = \lambda x(x)\, \mathtt{in}\, (f\, \mathtt{true}) :: (f\, \mathtt{nil}) : \mathit{bool\ list}.$$

Proof. First note that proof:

$$\text{(fn)}\,\frac{\text{(var}\succ\text{)}\quad \text{using } \forall\,\{\,\}\,(\alpha) \succ \alpha\ \ \dfrac{}{x : \alpha \vdash x : \alpha}}{\{\,\} \vdash \lambda x(x) : \alpha \to \alpha} \tag{1}$$

Next note that since $\forall\,\alpha\,(\alpha \to \alpha) \succ \mathit{bool} \to \mathit{bool}$, by (var $\succ$) we have

$$f : \forall\,\alpha\,(\alpha \to \alpha) \vdash f : \mathit{bool} \to \mathit{bool}.$$

By (bool) we also have

$$f : \forall\,\alpha\,(\alpha \to \alpha) \vdash \mathtt{true} : \mathit{bool}$$

and applying the rule (app) to these two judgements we get

$$f : \forall\,\alpha\,(\alpha \to \alpha) \vdash f\, \mathtt{true} : \mathit{bool}. \tag{2}$$

Similarly, using (app) on (var $\succ$) and (nil), we have

$$f : \forall\,\alpha\,(\alpha \to \alpha) \vdash f\, \mathtt{nil} : \mathit{bool\ list}. \tag{3}$$

Applying rule (cons) to (2) and (3) we get

$$f : \forall\,\alpha\,(\alpha \to \alpha) \vdash (f\, \mathtt{true}) :: (f\, \mathtt{nil}) : \mathit{bool\ list}.$$

Finally we can apply rule (let) to this and (1) to conclude

$$\{\,\} \vdash \mathtt{let}\, f = \lambda x(x)\, \mathtt{in}\, (f\, \mathtt{true}) :: (f\, \mathtt{nil}) : \mathit{bool\ list}$$

as required. □

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 8) 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

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

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

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

Slide 23

$$
\dfrac{\dfrac{\dfrac{\dfrac{}{x_1 : \tau_3, x_2 : \tau_5 \vdash x_1 : \tau_6}\,(C3)}{x_1 : \tau_3 \vdash \lambda x_2(x_1) : \tau_4}\,(C2)}{\{\,\} \vdash \lambda x_1(\lambda x_2(x_1)) : \tau_2}\,(C1) \qquad \dfrac{\dfrac{}{f : \forall A\,(\tau_2) \vdash f : \tau_7}\,(C5) \quad \dfrac{}{f : \forall A\,(\tau_2) \vdash f : \tau_8}\,(C6)}{f : \forall A\,(\tau_2) \vdash f\,f : \tau_1}\,(C4)}{\{\,\} \vdash \mathtt{let}\, f = \lambda x_1(\lambda x_2(x_1))\,\mathtt{in}\, f\,f : \tau_1}\,(C0)
$$

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

Given a typing environment Γ and an expression M, how can we decide whether or not there is a type scheme σ 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: if $\Gamma \vdash M : \forall A\,(\tau)$ is derivable, i.e. if $A = \mathit{ftv}(\tau) - \mathit{ftv}(\Gamma)$ and $\Gamma \vdash M : \tau$ is derivable from Slides 19–21, 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 23, any proof of $\{\,\} \vdash M : \tau_1$ 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 24 have to be satisfied.

Constraints generated while inferring a type for $\mathtt{let}\, f = \lambda x_1(\lambda x_2(x_1))\,\mathtt{in}\, f\,f$

$$
\begin{aligned}
A &= \mathit{ftv}(\tau_2) && \text{(C0)}\\
\tau_2 &= \tau_3 \to \tau_4 && \text{(C1)}\\
\tau_4 &= \tau_5 \to \tau_6 && \text{(C2)}\\
\forall\,\{\,\}\,(\tau_3) \succ \tau_6,\ \text{i.e. } \tau_3 &= \tau_6 && \text{(C3)}\\
\tau_7 &= \tau_8 \to \tau_1 && \text{(C4)}\\
\forall A\,(\tau_2) &\succ \tau_7 && \text{(C5)}\\
\forall A\,(\tau_2) &\succ \tau_8 && \text{(C6)}
\end{aligned}
$$

Slide 24

Thus M is typeable if and only if we can find types $\tau_1, \ldots, \tau_8$ satisfying the constraints on Slide 24. First note that they imply

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

So let us take τ_5, τ_6 to be type variables, say α_2, α_1 respectively. Hence by (C0), $A = \mathit{ftv}(\tau_2) = \mathit{ftv}(\alpha_1 \to (\alpha_2 \to \alpha_1)) = \{\alpha_1, \alpha_2\}$. Then (C4), (C5) and (C6) require that

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

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

$$\tau_9 \to (\tau_{10} \to \tau_9) = \tau_8 \to \tau_1 \tag{C7}$$

$$\tau_{11} \to (\tau_{12} \to \tau_{11}) = \tau_8. \tag{C8}$$

$$\text{(C9)}\dfrac{\text{(C10)}\dfrac{\text{(C11)}\dfrac{\text{(C12)}\dfrac{}{f:\tau_4 \vdash f:\tau_6} \qquad \text{(C13)}\dfrac{}{f:\tau_4\vdash f:\tau_7}}{f:\tau_4 \vdash f\,f:\tau_5}}{\{\,\}\vdash \lambda f(f\,f):\tau_2} \qquad \text{(C14)}\dfrac{\text{(C15)}\dfrac{\text{(C16)}\dfrac{}{x_1:\tau_8, x_2:\tau_{10}\vdash x_1:\tau_{11}}}{x_1:\tau_8\vdash \lambda x_2(x_1):\tau_9}}{\{\,\}\vdash: \lambda x_1(\lambda x_2(x_1)):\tau_3}}{\{\,\}\vdash (\lambda f(f\,f))\,\lambda x_1(\lambda x_2(x_1)):\tau_1}$$

Constraints:

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

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

Now (C7) can only hold if

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

and hence

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

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 24 by defining

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

With these choices, Figure 1 becomes a valid proof of

$$\{\,\} \vdash \texttt{let}\, f = \lambda x_1(\lambda x_2(x_1))\, \texttt{in}\, f\, f : \alpha_3 \to (\alpha_4 \to (\alpha_5 \to \alpha_4))$$

from the typing axioms and rules on Slides 19–21, i.e. we do have

$$\vdash \texttt{let}\, f = \lambda x_1(\lambda x_2(x_1))\, \texttt{in}\, f\, f : \forall\, \alpha_3, \alpha_4, \alpha_5\, (\alpha_3 \to (\alpha_4 \to (\alpha_5 \to \alpha_4))) \tag{4}$$

If we go through the same type inference process for the expression M' on Slide 23 we generate a tree and set of constraints as in Figure 2. These imply in particular that

$$\tau_7 \overset{(C13)}{=} \tau_4 \overset{(C12)}{=} \tau_6 \overset{(C11)}{=} \tau_7 \to \tau_5.$$

But there are no types τ_5, τ_7 satisfying $\tau_7 = \tau_7 \to \tau_5$, because $\tau_7 \to \tau_5$ contains at least one more '$\to$' symbol than does τ_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.

2.3 Principal type schemes

The type scheme $\forall\, \alpha_3, \alpha_4, \alpha_5\, (\alpha_3 \to (\alpha_4 \to (\alpha_5 \to \alpha_4)))$ not only satisfies (4), it is in fact the most general, or *principal* type scheme for $\texttt{let}\, f = \lambda x_1(\lambda x_2(x_1))\, \texttt{in}\, f\, f$, as defined on Slide 25. It is worth pointing out that in the presence of (a), the converse of condition (b) on Slide 25 holds: if $\vdash M : \forall\, A\,(\tau)$ and $\forall\, A\,(\tau) \succ \tau'$, then $\vdash M : \forall\, A'\,(\tau')$ (where $A' = \mathit{ftv}(\tau')$). This is a consequence of the substitution property of valid Mini-ML typing judgements given in Exercise 6.

Slide 26 gives the main result about the Mini-ML typeability problem. It was first proved for a simple type system without polymorphic `let`-expressions by Hindley (1969) and extended to the full system by Damas and Milner (1982).

Principal type schemes for closed expressions

A closed type scheme $\forall\, A\,(\tau)$ is the principal type scheme of a closed Mini-ML expression M if

(a) $\vdash M : \forall\, A\,(\tau)$

(b) for any other closed type scheme $\forall\, A'\,(\tau')$, if $\vdash M : \forall\, A'\,(\tau')$, then $\forall\, A\,(\tau) \succ \tau'$

Slide 25

Remark 3 (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 `let`'s in the expression. For example the expression on Slide 27 (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)

Theorem
If the closed Mini-ML expression M is typeable (i.e. $\vdash M : \sigma$ holds for some type scheme σ), 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 26

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

$$
\begin{array}{l}
\mathtt{let}\ pair = \lambda x(\lambda y(\lambda z(z\,x\,y)))\ \mathtt{in} \\
\quad \mathtt{let}\ x_1 = \lambda y(pair\,y\,y)\ \mathtt{in} \\
\quad\quad \mathtt{let}\ x_2 = \lambda y(x_1(x_1\,y))\ \mathtt{in} \\
\quad\quad\quad \mathtt{let}\ x_3 = \lambda y(x_2(x_2\,y))\ \mathtt{in} \\
\quad\quad\quad\quad \mathtt{let}\ x_4 = \lambda y(x_3(x_3\,y))\ \mathtt{in} \\
\quad\quad\quad\quad\quad \mathtt{let}\ x_5 = \lambda y(x_4(x_4\,y))\ \mathtt{in} \\
\quad\quad\quad\quad\quad\quad x_5(\lambda y(y))
\end{array}
$$

(Taken from Mairson (1990).)

Slide 27

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 26, 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 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 28. 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 et al., 1986, Section 6.7) for more details), we do need to explain the notation for type substitutions introduced on Slide 28.

Unification of ML types

There is an algorithm *mgu* which when input two Mini-ML types τ_1 and τ_2 decides whether τ_1 and τ_2 are unifiable, i.e. whether there exists a type-substitution $S \in \mathrm{Sub}$ 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) for all $S' \in \mathrm{Sub}$, if $S'(\tau_1) = S'(\tau_2)$, then $S' = TS$ for some $T \in \mathrm{Sub}$
(any other substitution S' can be factored through S, by specialising S with T)

By convention $mgu(\tau_1, \tau_2) = FAIL$ if (and only if) τ_1 and τ_2 are not unifiable.

Slide 28

Definition 4 (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 α. We write Sub for the set of all such functions. The *domain* of $S \in \mathrm{Sub}$ is the finite set of variables

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

Given a type substitution S, the effect of applying the substitution to a type is written $S\,\tau$; thus if $dom(S) = \{\alpha_1, \ldots, \alpha_n\}$ and $S(\alpha_i)$ is the type τ_i for each $i = 1..n$, then $S(\tau)$ is the type resulting from simultaneously replacing each occurrence of α_i in τ with τ_i (for all $i = 1..n$), i.e.

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

using the notation for substitution from Slide 16. 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.[2] As a result, the *composition* TS of two type substitutions $S, T \in \mathrm{Sub}$ means first apply S and then T. Thus by definition TS is the function mapping each type variable α 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 $TS \in Sub$; indeed, $dom(TS) \subseteq dom(T) \cup dom(S)$.

More generally, if $dom(S) = \{\alpha_1, \ldots, \alpha_n\}$ and σ is a Mini-ML type scheme, then $S\,\sigma$ will denote the result of the (capture-avoiding[3]) substitution of $S(\alpha_i)$ for each free occurrence of α_i in σ (for $i = 1..n$).

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

[2] i.e. we write $S\,\tau$ rather than $\tau\,S$ as in the Part IB *Logic and Proof* course.

[3] Since we identify type schemes up to renaming their $\forall$-bound type variables, we always assume the bound type variables in σ are different from any type variables in the types $S(\alpha_i)$.

principal type scheme for open expressions in the presence of a non-empty typing environment. This is done on Slide 29. For the definitions on that slide to be reasonable, we need some properties of the typing relation with respect to type substitutions and specialisation. These are stated on Slide 30; we leave the proofs as exercises (see Exercise 6). To compute principal type schemes it suffices to compute 'principal solutions' in the sense of Slide 29: for if M is in fact closed, then any principal solution (S, σ) for the typing problem $\{\,\} \vdash M : ?$ has the property that σ is a principal type scheme for M in the sense of Slide 25 (see Exercise 5).

Principal type schemes for open expressions

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

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

(where $S\,\Gamma = \{x_1 : S\,\sigma_1, \ldots, x_n : S\,\sigma_n\}$, if $\Gamma = \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$).
Such a solution is principal if given any other, (S', σ'), there is some $T \in \mathrm{Sub}$ with $TS = S'$ and $T(\sigma) \succ \sigma'$.

[For type schemes σ and σ', with $\sigma' = \forall\, A'\,(\tau')$ say, we define $\boxed{\sigma \succ \sigma'}$ to mean $A' \cap ftv(\sigma) = \{\}$ and $\sigma \succ \tau'$.]

Slide 29

Properties of the Mini-ML typing relation

- If $\Gamma \vdash M : \sigma$, then for any type substitution $S \in \mathrm{Sub}$
$$S\Gamma \vdash M : S\sigma$$
- If $\Gamma \vdash M : \sigma$ and $\sigma \succ \sigma'$, then $\Gamma \vdash M : \sigma'$.

Slide 30

Specification for the principal typing algorithm, *pt*

pt operates on typing problems $\Gamma \vdash M : ?$ (consisting of a typing environment Γ and a Mini-ML expression M).

It returns either a pair (S, τ) consisting of a type substitution $S \in \mathrm{Sub}$ and a Mini-ML type τ, or the exception *FAIL*.

- If $\Gamma \vdash M : ?$ has a solution (cf. Slide 2), then $pt(\Gamma \vdash M : ?)$ returns (S, τ) for some S and τ;
  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 31

Slide 31 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 32 and in Figure 3.[4] Note the following points about the definitions on Slide 32 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 $x \notin dom(\Gamma)$; and the clause for variables assumes that $A \cap ftv(\Gamma) = \{\,\}$.

(ii) The type substitution *Id* occurring in the clauses for variables and booleans is the *identity* substitution which maps each type variable α to itself.

(iii) We have not given the clauses of the definition for `nil`, cons, and `case`-expressions (Exercise 4).

(iv) We do not give the proof that the definition in Figure 3 is correct (i.e. meets the specification on Slide 31). 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 6.

[4] An implementation in Fresh O'Caml (`www.cl.cam.ac.uk/users/amp12/fresh-ocaml/`) 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 facilities for dealing with binding operations and fresh names.

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

Some of the clauses in a definition of *pt*

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

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

Slide 32

More efficient algorithms make use of a different approach to substitution and unification, based on equivalence relations on directed acylic 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.

3 Polymorphic Reference Types

3.1 The problem

Recall from the Introduction that an important purpose of type systems is to provide *safety* (Slide 4) via *type soundness* results (Slide 5). 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 15) with a unit type (a type with a single value) and *reference* types; and correspondingly, we augment the grammar for Mini-ML expressions (Slide 18) with a unit value, and operations for reference creation, dereferencing and assignment. These additions are shown on Slide 33. We call the resulting language Midi-ML. The typing rules for these new forms of expression are given on Slide 34.

ML types and expressions for mutable references

τ	$::=$	$\dots$	
	$\mid$	$unit$	unit type
	$\mid$	$\tau\ ref$	reference type.
M	$::=$	$\dots$	
	$\mid$	$()$	unit value
	$\mid$	$\mathtt{ref}\ M$	reference creation
	$\mid$	$!M$	dereference
	$\mid$	$M := M$	assignment

Slide 33

Midi-ML's extra typing rules

$$\Gamma \vdash () : \mathit{unit} \qquad \text{(unit)}$$

$$\frac{\Gamma \vdash M : \tau}{\Gamma \vdash \mathtt{ref}\, M : \tau\ \mathit{ref}} \qquad \text{(ref)}$$

$$\frac{\Gamma \vdash M : \tau\ \mathit{ref}}{\Gamma \vdash\, !M : \tau} \qquad \text{(get)}$$

$$\frac{\Gamma \vdash M_1 : \tau\ \mathit{ref} \qquad \Gamma \vdash M_2 : \tau}{\Gamma \vdash M_1 := M_2 : \mathit{unit}} \qquad \text{(set)}$$

Slide 34

Example

The expression

$$\begin{array}{l} \mathtt{let}\ r = \mathtt{ref}\ \lambda x(x)\ \mathtt{in} \\ \quad \mathtt{let}\ u = (r := \lambda x'(\mathtt{ref}\ !x'))\ \mathtt{in} \\ \qquad (!r)() \end{array}$$

has type *unit*.

Slide 35

Example 5. Here is an example of the typing rules on Slide 34 in use. The expression given on Slide 35 has type *unit*.

Proof. This can be deduced by applying the (let) rule (Slide 21) to the judgements

$$\begin{array}{l} \{\,\} \vdash \mathtt{ref}\ \lambda x(x) : (\alpha \to \alpha)\ \mathit{ref} \\ r : \forall\, \alpha\, ((\alpha \to \alpha)\ \mathit{ref}) \vdash \mathtt{let}\ u = (r := \lambda x'(\mathtt{ref}\ !x'))\ \mathtt{in}\ (!r)() : \mathit{unit}. \end{array}$$

The first of these judgements has the following proof:

$$\dfrac{\dfrac{\dfrac{}{x : \alpha \vdash x : \alpha}\,(\mathrm{var} \succ)}{\{\,\} \vdash \lambda x(x) : \alpha \to \alpha}\,(\mathrm{fn})}{\{\,\} \vdash \mathtt{ref}\, \lambda x(x) : (\alpha \to \alpha)\, \mathit{ref}}\,(\mathrm{ref})$$

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

$$r : \forall\, \alpha\, ((\alpha \to \alpha)\, \mathit{ref}) \vdash r := \lambda x'(\mathtt{ref}\, !x') : \mathit{unit} \tag{5}$$

$$r : \forall\, \alpha\, ((\alpha \to \alpha)\, \mathit{ref}), u : \mathit{unit} \vdash (!r)() : \mathit{unit} \tag{6}$$

Writing Γ for the typing environment $\{r : \forall\, \alpha\, ((\alpha \to \alpha)\, \mathit{ref})\}$, the proof of (5) is

$$\dfrac{\dfrac{}{\Gamma \vdash r : (\alpha\, \mathit{ref} \to \alpha\, \mathit{ref})\, \mathit{ref}}\,(\mathrm{var} \succ) \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, x' : \alpha\, \mathit{ref} \vdash x' : \alpha\, \mathit{ref}}\,(\mathrm{var} \succ)}{\Gamma, x' : \alpha\, \mathit{ref} \vdash !x' : \alpha}\,(\mathrm{get})}{\Gamma, x' : \alpha\, \mathit{ref} \vdash \mathtt{ref}\, !x' : \alpha\, \mathit{ref}}\,(\mathrm{ref})}{\Gamma \vdash \lambda x'(\mathtt{ref}\, !x') : \alpha\, \mathit{ref} \to \alpha\, \mathit{ref}}\,(\mathrm{fn})}{\Gamma \vdash r := \lambda x'(\mathtt{ref}\, !x') : \mathit{unit}}\,(\mathrm{set})$$

while the proof of (6) is

$$\dfrac{\dfrac{\dfrac{}{\Gamma, u : \mathit{unit} \vdash r : (\mathit{unit} \to \mathit{unit})\, \mathit{ref}}\,(\mathrm{var} \succ)}{\Gamma, u : \mathit{unit} \vdash !r : \mathit{unit} \to \mathit{unit}}\,(\mathrm{get}) \qquad \dfrac{}{\Gamma, u : \mathit{unit} \vdash () : \mathit{unit}}\,(\mathrm{unit})}{\Gamma, u : \mathit{unit} \vdash (!r)() : \mathit{unit}}\,(\mathrm{app})$$

□

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 35, 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 $\lambda x'(\mathtt{ref}\, !x')$ and binds the unit value to u. (Since the variable u does not occur in its body, M's innermost `let`-expression is just a way of expressing the sequence $(r := \lambda x'(\mathtt{ref}\, !x')); (!r)()$ in the fragment of ML that we are using for illustrative purposes.) Next $(!r)()$ is evaluated. This involves applying the current contents of r, which is $\lambda x'(\mathtt{ref}\, !x')$, 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

$$\langle M, \{\,\} \rangle \to \mathit{FAIL}$$

in the transition system defined in Figure 4 and Slide 36 (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 $\mathit{dom}(s)$) should be contained in the finite set $\mathit{dom}(s)$.
- The symbol *FAIL*, representing a run-time error.

(The notation $s[x \mapsto V]$ used on Slide 36 means the state with domain of definition $\mathit{dom}(s) \cup \{x\}$ mapping x to V and otherwise acting like s.)

The axioms and rules inductively defining the transition system for Midi-ML are those on Slide 36 together with the following ones:

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

where V ranges over *values*:

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

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

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

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

Midi-ML transitions involving references

$$\langle !x, s\rangle \to \langle s(x), s\rangle \quad \text{if } x \in \mathit{dom}(s)$$

$$\langle !V, s\rangle \to \mathit{FAIL} \quad \text{if } V \text{ not a variable}$$

$$\langle x := V', s\rangle \to \langle (), s[x \mapsto V']\rangle$$

$$\langle V := V', s\rangle \to \mathit{FAIL} \quad \text{if } V \text{ not a variable}$$

$$\langle \texttt{ref}\ V, s\rangle \to \langle x, s[x \mapsto V]\rangle \quad \text{if } x \notin \mathit{dom}(s)$$

where V ranges over values:

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

Slide 36

3.2 Restoring type soundness

The root of the problem described in the previous section lies in the fact that typing expressions like $\mathtt{let}\, r = \mathtt{ref}\, M_1 \,\mathtt{in}\, M_2$ with the (let) rule allows the storage location (bound to) r to have a type scheme σ 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 σ, 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 et al. (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 α) and the 'imperative type variables' (ranged over by $\_\alpha$). The rule (ref) is restricted by insisting that τ only involve imperative type variables; in other words the principal type scheme of $\lambda x(\mathtt{ref}\, x)$ becomes $\forall\, \_\alpha\, (\_\alpha \to \_\alpha\, \mathit{ref})$, rather than $\forall\, \alpha\, (\alpha \to \alpha\, \mathit{ref})$. Furthermore, and crucially, the (let) rule (Slide 21) 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 behaviourly 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 $\forall\, \alpha\, (\alpha\, \mathit{list} \to \alpha\, \mathit{list})$, the type scheme of the latter may well be the different type scheme $\forall\, \_\alpha\, (\_\alpha\, \mathit{list} \to \_\alpha\, \mathit{list})$. 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 et al. (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 $\alpha, \alpha' \ldots$ are used) while retaining a value-restricted form of the (let) rule, as shown on Slide 37.[5] 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 19–21 and 34, except that the applicability of the (let) rule is restricted as on Slide 37.

[5] N.B. what we call a value, Milner et al. (1997) calls a *non-expansive expression*.

Value-restricted typing rule for `let`-expressions

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

($\dagger$) provided $x \notin dom(\Gamma)$ and

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

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

Slide 37

Example 6. The expression on Slide 35 is not typeable in the type system for Midi-ML resulting from replacing rule (let) by the value-restricted rule (letv) on Slide 37 (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 $\mathtt{ref}\,\lambda x(x)$ is *not* a value, the rule has to be applied with $A = \{\,\}$. This entails trying to type

$$\mathtt{let}\, u = (r := \lambda x'(\mathtt{ref}\,!x'))\,\mathtt{in}\,(!r)() \tag{7}$$

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

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 38. 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, $\to$;
- if a configuration can be typed, it cannot posses 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 σ for which

$$\vdash M : \sigma$$

is provable in the value-restricted type system (axioms and rules on Slides 7–8, 2 and 1), then evaluation of M does not fail, i.e. there is no sequence of transitions of the form

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

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

Slide 38

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 8) (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.

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 and the first example on Slide 23) 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 39 points out, the same is not true of λ-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 4) via *type soundness* (Slide 5). Use of expressions such as those mentioned on Slide 39 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 40. 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:

- π is a polymorphic type;
- Γ 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 19 by the axiom and two rules shown on Slide 41. (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 π' for all free occurrences of α in π.

λ-bound variables in ML cannot be used polymorphically within a function abstraction

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

Syntactically, because in rule

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

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 39

Monomorphic types . . .

$$\tau ::= \alpha \mid \mathit{bool} \mid \tau \to \tau \mid \tau\ \mathit{list}$$

. . . and type schemes

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

Polymorphic types

$$\pi ::= \alpha \mid \mathit{bool} \mid \pi \to \pi \mid \pi\ \mathit{list} \mid \forall\,\alpha\,(\pi)$$

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

Slide 40

Identity, Generalisation and Specialisation

$$\Gamma \vdash x : \pi \quad \text{if } (x : \pi) \in \Gamma \qquad \text{(id)}$$

$$\frac{\Gamma \vdash M : \pi}{\Gamma \vdash M : \forall\,\alpha\,(\pi)} \quad \text{if } \alpha \notin \mathit{ftv}(\Gamma) \qquad \text{(gen)}$$

$$\frac{\Gamma \vdash M : \forall\,\alpha\,(\pi)}{\Gamma \vdash M : \pi[\pi'/\alpha]} \qquad \text{(spec)}$$

Slide 41

Example 7. In the modified ML type system (with polymorphic types and (var $\succ$) replaced by (id), (gen), and (spec)) one can prove the following typings for expressions which are untypeable in ML:

$$\{\,\} \vdash \lambda f((f\,\mathtt{true}) :: (f\,\mathtt{nil})) : \forall\,\alpha\,(\alpha \to \alpha) \to \mathit{bool\ list} \tag{8}$$
$$\{\,\} \vdash \lambda f(f\,f) : \forall\,\alpha\,(\alpha) \to \forall\,\alpha\,(\alpha). \tag{9}$$

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

$$
(\mathrm{fn})\,\dfrac{(\mathrm{gen})\,\dfrac{(\mathrm{app})\,\dfrac{(1)\,\dfrac{(\mathrm{id})\,\dfrac{}{f : \forall\,\alpha_1\,(\alpha_1) \vdash f : \forall\,\alpha_1\,(\alpha_1)}}{f : \forall\,\alpha_1\,(\alpha_1) \vdash f : \alpha_2 \to \alpha_2} \qquad (2)\,\dfrac{(\mathrm{id})\,\dfrac{}{f : \forall\,\alpha_1\,(\alpha_1) \vdash f : \forall\,\alpha_1\,(\alpha_1)}}{f : \forall\,\alpha_1\,(\alpha_1) \vdash f : \alpha_2}}{f : \forall\,\alpha_1\,(\alpha_1) \vdash f\,f : \alpha_2}}{f : \forall\,\alpha_1\,(\alpha_1) \vdash f\,f : \forall\,\alpha_2\,(\alpha_2)}}{\{\,\} \vdash \lambda f(f\,f) : \forall\,\alpha_1\,(\alpha_1) \to \forall\,\alpha_2\,(\alpha_2)}
$$

Nodes (1) and (2) are both instances of the (spec) rule: the first uses the substitution $(\alpha_2 \to \alpha_2)/\alpha_1$, whilst the second uses α_2/α_1. □

Fact (see Wells (1994)):

> For the modified ML type system with polymorphic types and (var $\succ$) replaced by the axiom and rules on Slide 41, *the type checking and typeability problems* (cf. Slide 8) *are equivalent and undecidable.*

Slide 42

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 42: 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 (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 21).

4.2 The PLC type system

The negative result on Slide 42 does not rule out the use of the polymorphic types of Slide 40 in programming languages, since one may consider *explicitly typed* languages (Slide 43) 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 $\forall\,\alpha\,(\alpha) \to \forall\,\alpha\,(\alpha)$
(cf. Example 7)
Implicitly typed version: $\lambda\, f\,(f\, f)$
Explicitly type version: $\lambda\, f : \forall\,\alpha_1\,(\alpha_1)\,(\Lambda\,\alpha_2\,(f(\alpha_2 \to \alpha_2)(f\,\alpha_2)))$

Slide 43

Remark 8 (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

Types

τ	$::=$	α	type variable
	$\mid$	$\tau \to \tau$	function type
	$\mid$	$\forall\,\alpha\,(\tau)$	$\forall$-type

Expressions

M	$::=$	x	variable
	$\mid$	$\lambda\, x : \tau\,(M)$	function abstraction
	$\mid$	$M\, M$	function application
	$\mid$	$\Lambda\,\alpha\,(M)$	type generalisation
	$\mid$	$M\,\tau$	type specialisation

(α and x range over fixed, countably infinite sets TyVar and Var respectively.)

Slide 44

Functions on types

In PLC, $\boxed{\Lambda\,\alpha\,(M)}$ is an anonymous notation for the function F mapping each type τ to the value of $M[\tau/\alpha]$ (of some particular type). $\boxed{F\,\tau}$ denotes the result of applying such a function to a type.

Computation in PLC involves beta-reduction for such functions on types

$$(\Lambda\,\alpha\,(M))\,\tau \to M[\tau/\alpha]$$

as well as the usual form of beta-reduction from λ-calculus

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

Slide 45

The explicit type information we need to add to expressions to get syntax-directed versions of the (gen) and (spec) rules (Slide 41) 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 45. 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 11). The PLC syntax is given on Slide 44. Its types, τ, are like the polymorphic types, π, given on Slide 40, 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 11.

Remark 9 (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 $\tau_1 \to \tau_2 \to \tau_3$ means $\tau_1 \to (\tau_2 \to \tau_3)$.

We delimit the scope of $\forall$-, λ-, and Λ-binders with parentheses. Another common way of writing these binders employs 'dot' notation

$$\forall\alpha\,.\tau \qquad \lambda x : \tau\,.\,M \qquad \Lambda\alpha\,.\,M$$

with the convention that the scope extends as far to the right as possible. For example $\forall\alpha_1\,.\,(\forall\alpha_2\,.\,\tau \to \alpha_1) \to \alpha_1$ means $\forall\,\alpha_1\,(\forall\,\alpha_2\,(\tau \to \alpha_1) \to \alpha_1)$. 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 10 (Free and bound (type) variables)**.** Any occurrences in τ of a type variable α become bound in $\forall\,\alpha\,(\tau)$. Thus by definition, the finite set, $\mathit{ftv}(\tau)$, of *free type variables of a type* τ, is given by

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

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

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

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 α become bound in $\Lambda\,\alpha\,(M)$. Thus

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

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 variables as in the second expression rather than the first, i.e. distinct from any free variables (and from each other).

Remark 11 (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 τ' for all free occurrences of the type variable α in a type τ.
- $M[M'/x]$ 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 τ for all free occurrences of the type variable α in an expression M.

The PLC type system uses typing judgements of the form shown on Slide 46. Its typing relation is the collection of such judgements inductively defined by the axiom and rules on Slide 47.

PLC typing judgement

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

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

Slide 46

PLC type system

$$\Gamma \vdash x : \tau \quad \text{if } (x : \tau) \in \Gamma \qquad \text{(var)}$$

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

$$\frac{\Gamma \vdash M_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash M_2 : \tau_1}{\Gamma \vdash M_1\, M_2 : \tau_2} \qquad \text{(app)}$$

$$\frac{\Gamma \vdash M : \tau}{\Gamma \vdash \Lambda\, \alpha\,(M) : \forall\, \alpha\,(\tau)} \quad \text{if } \alpha \notin ftv(\Gamma) \qquad \text{(gen)}$$

$$\frac{\Gamma \vdash M : \forall\, \alpha\,(\tau_1)}{\Gamma \vdash M\, \tau_2 : \tau_1[\tau_2/\alpha]} \qquad \text{(spec)}$$

Slide 47

An incorrect 'proof'

$$
\textbf{(wrong!)}\ \dfrac{(\mathrm{fn})\ \dfrac{(\mathrm{var})\ \dfrac{}{x_1 : \alpha, x_2 : \alpha \vdash x_2 : \alpha}}{x_1 : \alpha \vdash \lambda\, x_2 : \alpha\,(x_2) : \alpha \to \alpha}}{x_1 : \alpha \vdash \Lambda\, \alpha\,(\lambda\, x_2 : \alpha\,(x_2)) : \forall\, \alpha\,(\alpha \to \alpha)}
$$

Slide 48

Remark 12 (Side-condition on rule (gen))**.** To illustrate the force of the side-condition on rule (gen) on Slide 47, consider the last step of the 'proof' on Slide 48. It is not a correct instance of the (gen) rule, because the concluding judgement, whose typing environment $\Gamma = \{x_1 : \alpha\}$, does not satisfy $\alpha \notin \mathit{ftv}(\Gamma)$ (since $\mathit{ftv}(\Gamma) = \{\alpha\}$ in this case). On the other hand, the expression $\Lambda\, \alpha\,(\lambda\, x_2 : \alpha\,(x_2))$ does have type $\forall\, \alpha\,(\alpha \to \alpha)$ given the typing environment $\{x_1 : \alpha\}$. Here is a correct proof of that fact:

$$
(\mathrm{gen})\ \dfrac{(\mathrm{fn})\ \dfrac{(\mathrm{var})\ \dfrac{}{x_1 : \alpha, x_2 : \alpha' \vdash x_2 : \alpha'}}{x_1 : \alpha \vdash \lambda\, x_2 : \alpha'\,(x_2) : \alpha' \to \alpha'}}{x_1 : \alpha \vdash \Lambda\, \alpha'\,(\lambda\, x_2 : \alpha'\,(x_2)) : \forall\, \alpha'\,(\alpha' \to \alpha')}
$$

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

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

is the same as

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

and indeed, is the same as

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

Example 13. On Slide 43 we claimed that $\lambda\, f : \forall\, \alpha_1\,(\alpha_1)\,(\Lambda\, \alpha_2\,(f(\alpha_2 \to \alpha_2)(f\,\alpha_2)))$ has type $\forall\, \alpha\,(\alpha) \to \forall\, \alpha\,(\alpha)$. Here is a proof of that in the PLC type system:

$$
(\mathrm{fn}).\ \dfrac{(\mathrm{gen})\ \dfrac{(\mathrm{app})\ \dfrac{(\mathrm{spec})\ \dfrac{(\mathrm{var})\ \dfrac{}{f : \forall\, \alpha_1\,(\alpha_1) \vdash f : \forall\, \alpha_1\,(\alpha_1)}}{f : \forall\, \alpha_1\,(\alpha_1) \vdash f(\alpha_2 \to \alpha_2) : \alpha_2 \to \alpha_2} \quad (\mathrm{spec})\ \dfrac{(\mathrm{var})\ \dfrac{}{f : \forall\, \alpha_1\,(\alpha_1) \vdash f : \forall\, \alpha_1\,(\alpha_1)}}{f : \forall\, \alpha_1\,(\alpha_1) \vdash f\,\alpha_2 : \alpha_2}}{f : \forall\, \alpha_1\,(\alpha_1) \vdash f(\alpha_2 \to \alpha_2)(f\,\alpha_2) : \alpha_2}}{f : \forall\, \alpha_1\,(\alpha_1) \vdash \Lambda\, \alpha_2\,(f(\alpha_2 \to \alpha_2)(f\,\alpha_2)) : \forall\, \alpha_2\,(\alpha_2)}}{\{\,\} \vdash \lambda\, f : \forall\, \alpha_1\,(\alpha_1)\,(\Lambda\, \alpha_2\,(f(\alpha_2 \to \alpha_2)(f\,\alpha_2))) : (\forall\, \alpha_1\,(\alpha_1)) \to \forall\, \alpha_2\,(\alpha_2)}
$$

Example 14. There is no PLC type τ for which

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

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

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

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

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 τ 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 τ if it exists and *FAIL*s otherwise.

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

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

Slide 49

4.3 PLC type inference

As Examples 13 and 14 suggest, the type checking and typeability problems (Slide 8) 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 49. 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 50 and 51.[6] 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 α is chosen so that $\alpha \notin ftv(\Gamma)$.

[6] An implementation of this algorithm in Fresh O'Caml can be found on the course web page.

PLC type-checking algorithm, I

Variables:

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

Function abstractions:

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

Function applications:

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

$$
\begin{array}{l}
\text{let } \tau_1 = typ(\Gamma \vdash M_1 : ?) \text{ in} \\
\text{let } \tau_2 = typ(\Gamma \vdash M_2 : ?) \text{ in} \\
\text{case } \tau_1 \text{ of } \quad \tau \to \tau' \;\mapsto\; \text{if } \tau = \tau_2 \text{ then } \tau' \text{ else } \mathit{FAIL} \\
\qquad\qquad \mid \qquad\quad \_ \;\mapsto\; \mathit{FAIL}
\end{array}
$$

Slide 50

PLC type-checking algorithm, II

Type generalisations:

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

Type specialisations:

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

$$
\begin{array}{l}
\text{let } \tau = typ(\Gamma \vdash M : ?) \text{ in} \\
\text{case } \tau \text{ of } \quad \forall \alpha \, (\tau_1) \;\mapsto\; \tau_1[\tau_2/\alpha] \\
\qquad\qquad \mid \qquad\quad \_ \;\mapsto\; \mathit{FAIL}
\end{array}
$$

Slide 51

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 52.

Beta-reduction of PLC expressions

M beta-reduces to M' in one step, $\boxed{M \to M'}$ 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:

$$(\lambda\, x : \tau\,(M_1))\, M_2 \to M_1[M_2/x]$$
$$(\Lambda\, \alpha\,(M))\, \tau \to M[\tau/\alpha].$$

$M \to^* M'$ indicates a chain of finitely † 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 52

Example 15. Here are some examples of beta-reductions. The various redexes are shown boxed. Clearly, the final expression y is in beta-normal form.

$$\begin{tikzcd}[column sep=small]
& \boxed{(\lambda\, x : \alpha_1 \to \alpha_1\,(x\, y))\boxed{(\Lambda\, \alpha_2\,(\lambda\, z : \alpha_2\,(z)))(\alpha_1 \to \alpha_1)}} \arrow[dl] \arrow[dr] & \\
\boxed{(\Lambda\, \alpha_2\,(\lambda\, z : \alpha_2\,(z)))(\alpha_1 \to \alpha_1)}\, y \arrow[dr] & & \boxed{(\lambda\, x : \alpha_1 \to \alpha_1\,(x\, y))\,(\lambda\, z : \alpha_1 \to \alpha_1\,(z))} \arrow[dl] \\
& \boxed{(\lambda\, z : \alpha_1 \to \alpha_1\,(z))y} \arrow[d] & \\
& y &
\end{tikzcd}$$

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 $M \to M'$, then $\Gamma \vdash M' : \tau$ is also a provable typing.

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

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

Slide 53

Slide 53 lists some important properties of *typeable* PLC expressions that we state without proof. The first is a weak form of type soundness result (Slide 5) and its proof is straightforward. The proof of the Church Rosser property is also quite easy whereas the proof of Strong Normalisations is difficult.[7] It was first 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, $\boxed{M =_\beta M'}$ holds if there is a finite chain

$$M - \cdot - \cdots - \cdot - M'$$

where each $-$ is either $\to$ 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

$$M \to^* N \; {}^*\!\!\leftarrow M'$$

Slide 54

[7] 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 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.

Theorem 16. *The properties listed on Slide 53 have the following consequences.*

(i) Each typeable PLC expression, M, possesses a beta-normal form, *i.e. an N such that $M \to^* N \not\to$, which is unique (up to alpha-conversion).*

(ii) The equivalence relation of beta-conversion *(Slide 54) 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 $M \to^* N_1$ and $M \to^* N_2$, then $N_1 \to^* M' \, {}^*\!\!\leftarrow N_2$ holds for some M'; so if N_1 and N_2 are beta-normal forms, then it must be that $N_1 \to^* M'$ and $N_2 \to^* M'$ 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 expressions are beta-convertible. (And of course, the relation of alpha-convertibility is decidable.) □

Remark 17. In fact, the Church Rosser property holds for all PLC expressions, whether or not they are typeable. However, the Strong Normalisation property 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 $\Omega \to \Omega \to \Omega \to \cdots$. 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 53 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

$$\mathit{bool} \stackrel{\text{def}}{=} \forall\, \alpha\, (\alpha \to (\alpha \to \alpha))$$

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

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

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

Slide 55

Example 18 (Booleans). The PLC type corresponding to the ML datatype

$$\texttt{datatype}\ \mathit{bool} = \mathit{True} \mid \mathit{False}$$

is shown on Slide 55. The idea behind this representation is that the 'algorithmic essence' of a boolean, b, is the operation $\lambda x_1 : \alpha, x_2 : \alpha(\texttt{if}\ b\ \texttt{then}\ x_1\ \texttt{else}\ x_2)$ of type $\alpha \to \alpha \to \alpha$,[8] 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 α, so in PLC we can take the step of identifying booleans with expressions of the corresponding $\forall$-type, $\forall\,\alpha\,(\alpha \to \alpha \to \alpha)$. Note that for the PLC expressions *True* and *False* defined on Slide 55 the typings

$$\{\,\} \vdash \mathit{True} : \forall\,\alpha\,(\alpha \to \alpha \to \alpha) \quad \text{and} \quad \{\,\} \vdash \mathit{False} : \forall\,\alpha\,(\alpha \to \alpha \to \alpha)$$

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

$$\texttt{case}\ M_1\ \texttt{of}\ \mathit{True} \Rightarrow M_2 \mid \mathit{False} \Rightarrow M_3$$

has an explicitly typed analogue in PLC, viz. $\mathit{if}\ \tau\ M_1\ M_2\ M_3$, where τ is supposed to be the common type of M_2 and M_3 and *if* is the PLC expression given on Slide 55. It is not hard to see that

$$\{\,\} \vdash \mathit{if} : \forall\,\alpha\,(\mathit{bool} \to (\alpha \to (\alpha \to \alpha))).$$

Thus if $\Gamma \vdash M_1 : \mathit{bool}$, $\Gamma \vdash M_2 : \tau$ and $\Gamma \vdash M_3 : \tau$, then $\Gamma \vdash \mathit{if}\ \tau\ M_1\ M_2\ M_3 : \tau$ (cf. the typing rule (if) on Slide 19). Furthermore, the expressions *True*, *False*, and *if* have the expected dynamic behaviour:

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

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.

Polymorphic lists

$$\alpha\ \mathit{list} \stackrel{\text{def}}{=} \forall\,\alpha'\,(\alpha' \to (\alpha \to \alpha' \to \alpha') \to \alpha')$$

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

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

Slide 56

[8] Recall our notational conventions: $\alpha \to \alpha \to \alpha$ means $\alpha \to (\alpha \to \alpha)$.

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 $f : A \to A' \to A'$, the iteratively defined function $listIter\, x'\, f$ is the unique function $g : A^* \to A'$ satisfying:

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

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

Slide 57

Example 19 (Lists). The polymorphic type corresponding to the ML datatype

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

is shown on Slide 56. 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 57. The existence of such functions $listIter\, x'\, f$ does in fact characterise the set A^* of finite lists over a set A uniquely up to bijection. We can take the operation

$$\lambda x' : \alpha', f : \alpha \to \alpha' \to \alpha' (listIter\, x'\, f\, \ell) \tag{11}$$

(of type $\alpha' \to (\alpha \to \alpha' \to \alpha') \to \alpha'$) as the 'algorithmic essence' of the list $\ell : \alpha\, list$. Clearly this operation is parametrically polymorphic in α' and so we are led to the $\forall$-type given on Slide 56 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 $listIter\, x'\, f$ is applied to it yields x'. This motivates the definition of the PLC expression Nil on Slide 56. Similarly for the constructor $Cons$ for adding an element to the head of a list. It is not hard to prove the typings:

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

As shown on Slide 58, an explicitly typed version of the operation of list iteration can be defined in PLC: $iter\, \alpha\, \alpha'\, x'\, f$ satisfies the defining equations for an iteratively defined function (11) up to beta-conversion.

ML	PLC
`datatype` $null = ;$	$null \stackrel{\text{def}}{=} \forall\,\alpha\,(\alpha)$
`datatype` $unit = Unit;$	$unit \stackrel{\text{def}}{=} \forall\,\alpha\,(\alpha \to \alpha)$
$\alpha_1 * \alpha_2$	$\alpha_1 * \alpha_2 \stackrel{\text{def}}{=} \forall\,\alpha\,((\alpha_1 \to \alpha_2 \to \alpha) \to \alpha)$
`datatype` $(\alpha_1, \alpha_2)sum =$ Inl `of` $\alpha_1 \mid Inr$ `of` $\alpha_2;$	$(\alpha_1, \alpha_2)sum \stackrel{\text{def}}{=}$ $\forall\,\alpha\,((\alpha_1 \to \alpha) \to (\alpha_2 \to \alpha) \to \alpha)$
`datatype` $nat =$ $Zero \mid Succ$ `of` $nat;$	$nat \stackrel{\text{def}}{=}$ $\forall\,\alpha\,(\alpha \to (\alpha \to \alpha) \to \alpha)$
`datatype` $binTree =$ $Leaf \mid Node$ `of` $binTree * binTree;$	$binTree \stackrel{\text{def}}{=}$ $\forall\,\alpha\,(\alpha \to (\alpha \to \alpha \to \alpha) \to \alpha)$

Figure 5: Some more algebraic datatypes

List iteration in PLC

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

satisfies:

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

Slide 58

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

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

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.

5 Further Topics

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 briefly survey a couple of other general topics which are of importance in the development of the theory and application of type systems in computer science. The book Pierce (2005) is still a good source for essays on further topics in type systems for programming languages.

5.1 Dependent types

A tautology checker

$$
\begin{aligned}
\texttt{fun}\ \mathit{taut}\ x\ f = {}& \texttt{if}\ x = 0\ \texttt{then}\ f\ \texttt{else}\\
& (\mathit{taut}(x-1)(f\ \texttt{true}))\\
& \texttt{andalso}\ (\mathit{taut}(x-1)(f\ \texttt{false}))
\end{aligned}
$$

Defining types $n\,\mathit{AryBoolOp}$ for each natural number $n \in \mathbb{N}$

$$
\begin{cases}
0\ \mathit{AryBoolOp} & \stackrel{\text{def}}{=} \mathit{bool}\\
(n+1)\ \mathit{AryBoolOp} & \stackrel{\text{def}}{=} \mathit{bool} \to (n\,\mathit{AryBoolOp})
\end{cases}
$$

then $\mathit{taut}\,n$ has type $(n\,\mathit{AryBoolOp}) \to \mathit{bool}$, i.e. the result type of the function *taut depends upon the value of its argument*.

Slide 59

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

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

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 59. This is algorithmically correct, but does not type-check in ML. Why? Intuitively, the type of $\mathit{taut}\,n$ for each natural number $n = 0, 1, 2, \ldots$ is the type $n\,\mathit{AryBoolOp}$ of 'n-ary curried boolean operations' defined (by induction on n) on Slide 59. 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, because ML does not permit such dependence in its types. Slide 60 programs the tautology-checker in Agda (`wiki.portal.chalmers.se/agda/agda.php`), a popular dependently typed functional programming language with syntax reminiscent of Haskell (`www.haskell.org`).

The tautology checker in Agda

```
data Bool : Set where
  True : Bool
  False : Bool

_and_ : Bool -> Bool -> Bool
True and True = True
True and False = False
False and _ = False

data Nat : Set where
  Zero : Nat
  Succ : Nat -> Nat

_AryBoolOp : Nat -> Set
Zero AryBoolOp = Bool
(Succ n) AryBoolOp = Bool -> n AryBoolOp

taut : (n : Nat) -> n AryBoolOp -> Bool
taut Zero f = f
taut (Succ n) f = taut n (f True) and taut n (f False)
```

Slide 60

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 n *AryBoolOp* is indexed by values n of a type of numbers.) Some typing rules for dependent function types are given on Slide 61. Note that the usual typing rules for function types $\tau \to \tau'$ are the special case where the type τ' has no dependency on values.

Type systems featuring dependent types are able to express much more refined properties of programs than ones without this feature. So why do they not 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. For example, if we add to the Agda code in Slide 60 a definition of the addition function

```
_plus_ : Nat -> Nat -> Nat
n plus Zero = n
n plus (Succ n') = Succ(n plus n')
```

then terms of type

```
((Succ Zero)plus(Succ Zero))AryBoolOp
```

are also terms of type

```
(Succ(Succ(Zero)))AryBoolOp
```

Dependent function types $(x : \tau) \to \tau'$

$$\frac{\Gamma, x : \tau \vdash M : \tau'}{\Gamma \vdash \lambda x : \tau\,(M) : (x : \tau) \to \tau'} \quad \text{if } x \notin \mathit{dom}(\Gamma) \cup \mathit{fv}(\Gamma)$$

$$\frac{\Gamma \vdash M : (x : \tau) \to \tau' \qquad \Gamma \vdash M' : \tau}{\Gamma \vdash M\,M' : \tau'[M'/x]}$$

τ' may 'depend' on x, i.e. have free occurrences of x.

(Free occurrences of x in τ' are bound in $(x : \tau) \to \tau'$.)

Slide 61

In a Turing-powerful language (which Agda is not) one would expect such value-equality 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.

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. Coq (`coq.inria.fr`) is an increasingly popular example of such a system. In this respect Martin-Löf's *intuitionistic type theory* (which first popularised the notion of 'dependent type') has been highly influential; see Nordström et al. (1990) for an introduction. The Agda language is based upon it (and as it says on its home page, 'Agda is a proof assistant' as well as a dependently typed functional programming language).

Curry-Howard correspondence

Logic	$\leftrightarrow$	Type system
propositions, ϕ	$\leftrightarrow$	types, τ
(constructive) proofs, p	$\leftrightarrow$	expressions, M
'p is a proof of ϕ'	$\leftrightarrow$	'M is an expression of type τ'
simplification of proofs	$\leftrightarrow$	reduction of expressions

Slide 62

5.2 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 62. 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 63.

Second-order intuitionistic propositional calculus (2IPC)

2IPC propositions: $\boxed{\phi ::= p \mid \phi \to \phi \mid \forall\, p\,(\phi)}$, where p ranges over an infinite set of propositional variables.
2IPC sequents: $\boxed{\Phi \vdash \phi}$, where Φ is a finite (multi)set of 2IPC propositions and ϕ is a 2IPC proposition.
$\Phi \vdash \phi$ is provable if it is in the set of sequents inductively generated by:

$$(\mathrm{Id})\ \Phi \vdash \phi \quad \text{if } \phi \in \Phi$$

$$(\to\mathrm{I})\ \frac{\Phi, \phi \vdash \phi'}{\Phi \vdash \phi \to \phi'} \qquad (\to\mathrm{E})\ \frac{\Phi \vdash \phi \to \phi' \quad \Phi \vdash \phi}{\Phi \vdash \phi'}$$

$$(\forall\mathrm{I})\ \frac{\Phi \vdash \phi}{\Phi \vdash \forall\, p\,(\phi)} \text{ if } p \notin fv(\Phi) \qquad (\forall\mathrm{E})\ \frac{\Phi \vdash \forall\, p\,(\phi)}{\Phi \vdash \phi[\phi'/p]}$$

Slide 63

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 $\Gamma \vdash M : \phi$, once we have fixed a labelling $\Gamma = \{x_1 : \phi_1, \ldots, x_n : \phi_n\}$ of the propositions in $\Phi = \{\phi_1, \ldots, \phi_n\}$ with variables $x_1, \ldots, x_n$. M is built up by recursion on the structure of the proof of the sequent using the following transformations:

$$(\mathrm{Id})\ \Phi, \phi \vdash \phi \quad \mapsto \quad (\mathrm{id})\ \overline{x} : \Phi, x : \phi \vdash x : \phi$$

$$(\to\mathrm{I})\ \frac{\Phi, \phi \vdash \phi'}{\Phi \vdash \phi \to \phi'} \quad \mapsto \quad (\mathrm{fn})\ \frac{\overline{x} : \Phi, x : \phi \vdash M : \phi'}{\overline{x} : \Phi \vdash \lambda\, x : \phi\,(M) : \phi \to \phi'}$$

$$(\to\mathrm{E})\ \frac{\Phi \vdash \phi \to \phi' \quad \Phi \vdash \phi}{\Phi \vdash \phi'} \quad \mapsto \quad (\mathrm{app})\ \frac{\overline{x} : \Phi \vdash M_1 : \phi \to \phi' \quad \overline{x} : \Phi \vdash M_2 : \phi}{\overline{x} : \Phi \vdash M_1\, M_2 : \phi'}$$

$$(\forall\mathrm{I})\ \frac{\Phi \vdash \phi}{\Phi \vdash \forall\, p\,(\phi)} \quad \mapsto \quad (\mathrm{gen})\ \frac{\overline{x} : \Phi \vdash M : \phi}{\overline{x} : \Phi \vdash \Lambda\, p\,(M) : \forall\, p\,(\phi)}$$

$$(\forall\mathrm{E})\ \frac{\Phi \vdash \forall\, p\,(\phi)}{\Phi \vdash \phi[\phi'/p]} \quad \mapsto \quad (\mathrm{spec})\ \frac{\overline{x} : \Phi \vdash M : \forall\, p\,(\phi)}{\overline{x} : \Phi \vdash M\, \phi' : \phi[\phi'/p]}$$

This is illustrated on Slide 64. The example on that slide uses the fact that the logical operation of conjunction can be defined in 2IPC. Slide 67 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

$$
(\forall I)\,\dfrac{(\forall I)\,\dfrac{(\to I)\,\dfrac{(\to E)\,\dfrac{(\to I)\,\dfrac{(\to I)\,\dfrac{(Id)\,\dfrac{}{\{p \mathbin{\&} q, p, q\} \vdash p}}{\{p \mathbin{\&} q, p\} \vdash q \to p}}{\{p \mathbin{\&} q\} \vdash p \to q \to p} \qquad (\forall E)\,\dfrac{(Id)\,\dfrac{}{\{p \mathbin{\&} q\} \vdash \forall r\,((p \to q \to r) \to r)}}{\{p \mathbin{\&} q\} \vdash (p \to q \to p) \to p}}{\{p \mathbin{\&} q\} \vdash p}}{\{\,\} \vdash p \mathbin{\&} q \to p}}{\{\,\} \vdash \forall q\,(p \mathbin{\&} q \to p)}}{\{\,\} \vdash \forall p, q\,(p \mathbin{\&} q \to p)}
$$

where $p \mathbin{\&} q$ is an abbreviation for $\forall r\,((p \to q \to r) \to r)$.
The PLC expression corresponding to this proof is:

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

Slide 64

The Curry-Howard correspondence gives us a different perspective on the typing judgement $\Gamma \vdash M : \sigma$, outlined on Slide 65. As well as the undecidablity 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—see Slide 66. The Law of Excluded Middle is so familiar that, when reasoning in classical logic, we may hardly be aware we are using it. Slide 68 gives an example of a proof in classical logic that perhaps leaves a bad taste in the mouth: it proves that there are irrational numbers whose exponential is rational, but it does not give any explicit example of such numbers.

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 (such as Coq); 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 et al. (1996), for example); and the use of type systems based on *modal logics* for analysing *partial evaluation* and *run-time code generation* Davis and Pfenning (1996).

Type-inference versus proof search

Type-inference: 'given Γ and M, is there a type τ such that $\Gamma \vdash M : \tau$?'
(For PLC/2IPC this is decidable.)

Proof-search: 'given Γ and ϕ, is there a proof term M such that $\Gamma \vdash M : \phi$?'

(For PLC/2IPC this is undecidable.)

Slide 65

2IPC is a constructive logic

For example, there is no proof of the Law of Excluded Middle

$$\forall\, p\, (p \vee \neg p)$$

Using the definitions on Slide 5, this is an abbreviation for

$$\forall\, p, q\, ((p \to q) \to ((p \to \forall\, r\, (r)) \to q) \to q)$$

(The fact that there is no closed PLC term of type $\forall\, p\, (p \vee \neg p)$ can be proved using the technique developed in the Tripos question 13 on paper 9 in 2000.)

Slide 66

Logical operations definable in 2IPC

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

Slide 67

Example of a non-constructive proof

Theorem. There exist two irrational numbers a and b such that b^a is rational.

Proof. Either $\sqrt{2}^{\sqrt{2}}$ is rational, or it is not (LEM!).
If it is, we can take $a = b = \sqrt{2}$, since $\sqrt{2}$ is irrational by a well-known theorem attributed to Euclid.
If it is not, we can take $a = \sqrt{2}$ and $b = \sqrt{2}^{\sqrt{2}}$, since then $b^a = (\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = \sqrt{2}^{\sqrt{2}\times\sqrt{2}} = \sqrt{2}^2 = 2$.
QED

Slide 68

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.

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 Lanuages*, 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.

Girard, J.-Y. (1972). *Interprétation fonctionelle et élimination des coupures dans l'arithmetique 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.

Hindley, J. R. (1969). The principal type scheme of an object in combinatory logic. *Transations of the American Mathematical Society 146*, 29–40.

Lamport, L. and L. C. Paulson (1999). Should your specification language be typed? *ACM Transactions on Programming Languages and Systems 21*(3), 502–526.

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. ACM Press.

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.

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.

Pierce, B. C. (Ed.) (2005). *Advanced Topics in 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. *Journal of the 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.

Wells, J. B. (1994). Typability and type-checking in the second-order λ-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.

CST Part II Types: Exercise Sheet

ML Polymorphism

Exercise 1. Here are some type checking problems, in the sense of Slide 8. Prove the following typings hold for the Mini-ML type system:

$$\begin{aligned}
&\vdash \lambda x(x :: \mathtt{nil}) : \forall\,\alpha\,(\alpha \to \alpha\; \mathit{list})\\
&\vdash \lambda x(\mathtt{case}\, x \,\mathtt{of}\, \mathtt{nil} \Rightarrow \mathtt{true} \mid x_1 :: x_2 \Rightarrow \mathtt{false}) : \forall\,\alpha\,(\alpha\; \mathit{list} \to \mathit{bool})\\
&\vdash \lambda x_1(\lambda x_2(x_1)) : \forall\,\alpha_1, \alpha_2\,(\alpha_1 \to (\alpha_2 \to \alpha_1))\\
&\vdash \mathtt{let}\, f = \lambda x_1(\lambda x_2(x_1))\,\mathtt{in}\, f\, f : \forall\,\alpha_1, \alpha_2, \alpha_3\,(\alpha_1 \to (\alpha_2 \to (\alpha_3 \to \alpha_2))).
\end{aligned}$$

Exercise 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 19–21 to show that the provable typing judgements, $\Gamma \vdash M : \tau$, all have the property that $fv(M) \subseteq dom(\Gamma)$.]

Exercise 3. Let σ and σ' be Mini-ML type schemes. Show that the relation $\sigma \succ \sigma'$ defined on Slide 29 holds if and only if

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

[Hint: use the following property of simultaneous substitution:

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

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

Exercise 4. Try to augment the definition of pt on Slide 32 and in Figure 3 with clauses for `nil`, cons, and `case`-expressions.

Exercise 5. Suppose M is a closed expression and that (S, σ) is a principal solution for the typing problem $\{\,\} \vdash M : ?$ in the sense of Slide 29. Show that σ must be a principal type scheme for M in the sense of Slide 25.

Exercise 6. Show that if $\Gamma \vdash M : \sigma$ is provable and $S \in \mathrm{Sub}$ is a type substitution, then $S\,\Gamma \vdash M : S\,\sigma$ is also provable.

Polymorphic Reference Types

Exercise 7. Letting M denote the expression on Slide 35 and $\{\,\}$ the empty state, show that $\langle M, \{\,\}\rangle \to^* \mathit{FAIL}$ is provable in the transition system defined in Figure 4.

Exercise 8. 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).

Polymorphic Lambda Calculus

Exercise 9. Give a proof inference tree for (8) in Example 7. Show that

$$\forall\,\alpha_1\,(\alpha_1 \to \forall\,\alpha_2\,(\alpha_2)) \to \mathit{bool\; list}$$

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

Exercise 10. 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 α-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 47.]

Exercise 11. In PLC, defining the expression $\mathtt{let}\, x = M_1 : \tau\,\mathtt{in}\, M_2$ to be an abbreviation for $(\lambda\, x : \tau\,(M_2))\, M_1$, show that the typing rule

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

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

Exercise 12. The *erasure*, $erase(M)$, of a PLC expression M is the expression of the untyped lambda calculus obtained by deleting all type information from M:

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

(i) Find PLC expressions M_1 and M_2 satisfying $erase(M_1) = \lambda\, x\,(x) = erase(M_2)$ such that $\vdash M_1 : \forall\, \alpha\,(\alpha \to \alpha)$ and $\vdash M_2 : \forall\, \alpha_1\,(\alpha_1 \to \forall\, \alpha_2\,(\alpha_1))$ are provable PLC typings.

(ii) We saw in Example 13 that there is a closed PLC expression M of type $\forall\, \alpha\,(\alpha) \to \forall\, \alpha\,(\alpha)$ satisfying $erase(M) = \lambda\, f\,(f\, f)$. 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 $erase(M)$ is strongly normalisable in the untyped lambda calculus, i.e. there are no infinite chains of beta-reductions starting from $erase(M)$. Assuming this result, exhibit an expression of the untyped lambda calculus which is not equal to $erase(M)$ for any closed, typeable PLC expression M.

Exercise 13. Prove the various typings and beta-reductions asserted in Example 18.

Exercise 14. Prove the various typings asserted in Example 19 and the beta-conversions on Slide 58.

Exercise 15. 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:

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

Exercise 16. [hard] Suppose that τ is a PLC type with a single free type variable, α. Suppose also that T is a closed PLC expression satisfying

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

Define ι to be the closed PLC type

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

Show how to define PLC expressions R and I satisfying

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