Q

Lecture Notes on

Types

for Part II of the Computer Science Tripos

Prof. Andrew M. Pitts
University of Cambridge
Computer Laboratory

Contents

Learning Guide

These notes and slides are designed to accompany 12 lectures on type systems for Part II of the Cambridge University Computer Science Tripos. The course builds on the techniques introduced in the Part IB course on *Semantics of Programming Languages* for specifying type systems for programming languages and reasoning about their properties. The emphasis here is on type systems for functional languages and their connection to constructive logic. We pay particular attention to the notion of *parametric polymorphism* (also known as *generics*), both because it has proven useful in practice and because its theory is quite subtle.

Tripos questions and exercises 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. At the end of the course you should be able to use a rule-based specification of a type system to carry out type checking and type inference; understand by example the Curry-Howard correspondence between type systems and logics; and appreciate the expressive power of parametric polymorphism and dependent types. 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 a lot 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.)

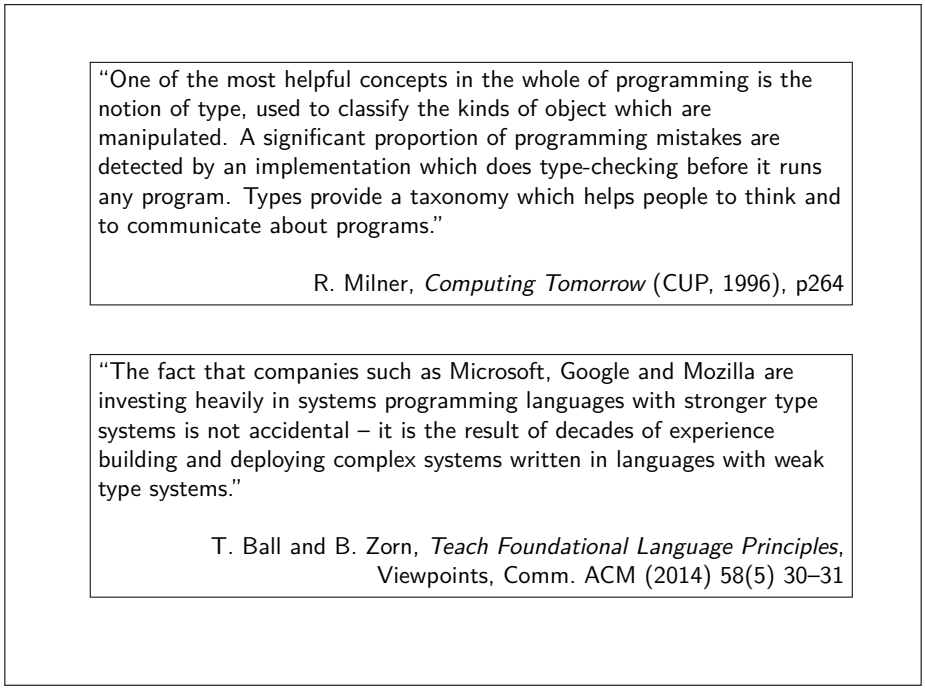
Section 4 Read Girard (1989) for an account by one of its creators of the polymorphic lambda calculus (System F), its relation to proof theory and much else besides.

Sections 5–7 Aspinall and Hofmann (2005) give a readable introduction to the topic of dependent types. Pure Type Systems are described in the handbook chapter by Barendregt (1992). Bove and Dybjer (2009) provide a nice introduction to inductive types (and Curry-Howard) using the Agda system as a dependently typed functional programming language.

Note. The material in these notes has been drawn from several different sources, including previous versions of this course by the author and by others. In particular I am grateful to Dominic Orchard who gave the course in 2014-15 and who converted the LaTeX sources to use the `beamer` package. 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`

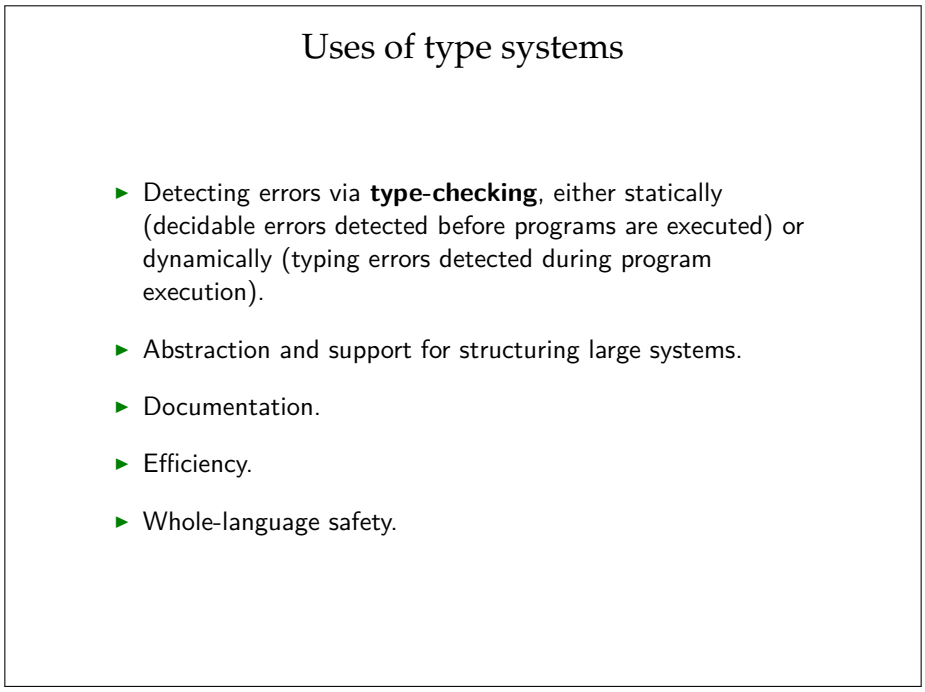
1 Introduction



Slide 1

This short course is primarily about the use of types in programming languages. Types also play an important role in logic and specification languages; indeed the concept of *type* in the sense we use it here first arose in the work of Russell (1903) as a way of avoiding certain paradoxes in the logical foundations of mathematics. In a similar way, one can use types to rule out paradoxical, non-sensical, or badly-behaved programs. We will return to the interplay between types in programming languages and types in logic in the last part of this course (Section 6).

Many programming language systems use types (and related notions such as structures, classes, interfaces, etc.) to classify 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"). Opinions differ about the extent to which it is a Good Thing for a programming language design to force users to specify typing information. Slide 1 contains a couple of quotes from the pro camp (of which I am a member), one from the previous century from the father of the ML family of languages and one more recent one from a couple of senior researchers at Microsoft Research (whose article is subtitled "Industry is ready and waiting for more graduates educated in the principles of programming languages" – amen to that). What I think is indisputable is that type systems have been, and continue to be, 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 2) in which type systems for programming languages get used:

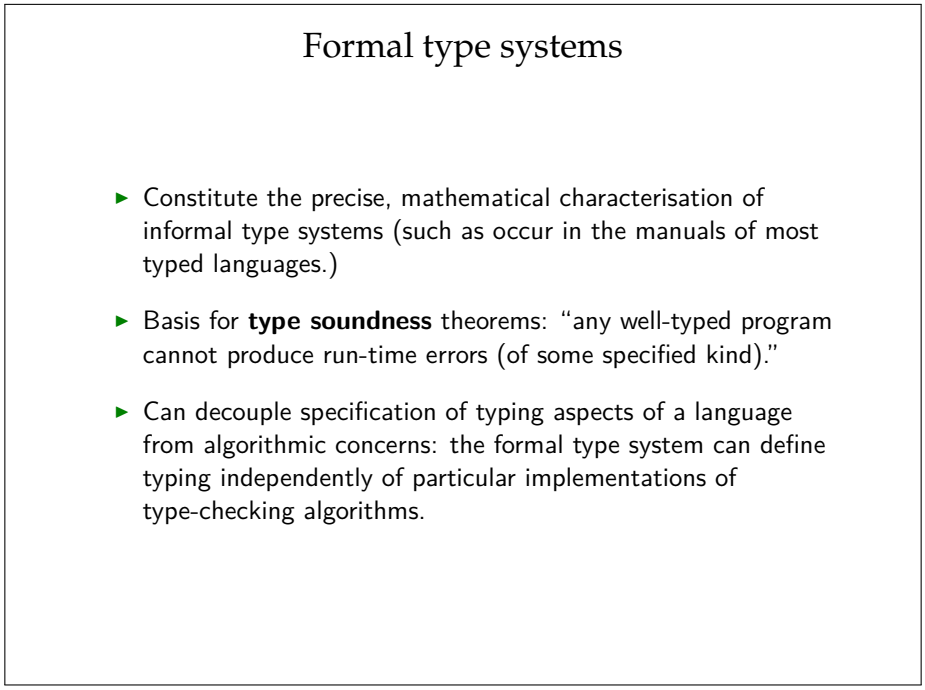
Detecting errors Experience suggests that a significant proportion of programming mistakes (such as trying to divide an integer by a string) can be detected by an implementation which does *static* type-checking, i.e. which checks for typing errors before it runs any program. Type systems used to implement 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 recursive partial 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 and user comprehension.

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).

Whole-language safety A programming language is sometimes called *safe* if its programs may have *trapped* errors (one that can be handled gracefully), but cannot have *untrapped errors* (ones that cause unpredictable crashes). 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`).



Slide 3

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. 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* (Slide 3) has been subject to proof. In this course we will look at smaller examples of

formally specified type systems. To do so, we use the techniques introduced in the CST Part IB course on *Semantics of Programming Languages*: to specify a formal type system one gives a number of axioms and rules for inductively generating typing *judgements*, a typical example of which is shown on Slide 4. Ideally the rules follow the structure of the phrases, explaining how to type them in terms of how their subphrases can be typed – one speaks of *syntax-directed* sets of rules. It is worth pointing out that different language families use different notations for the *has-type* relation (Slide 5).

Typical type system judgement

is a relation between typing environments (Γ), program phrases (e) and type expressions (τ) that we write as

$$\Gamma \vdash e : \tau$$

and read as: *given the assignment of types to free identifiers of e specified by type environment Γ, then e 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.

We consider **structural** type systems, in which there is a language of type expressions built up using type constructs (e.g. $int\ list \to int$ in ML).
(By contrast, in **nominal** type systems, type expressions are just unstructured names.)

Slide 4

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 5

Once we have formalised a particular type system, we are in a position to *prove* results about *type soundness* (Slide 3) and the notions of *type checking*, *typeability* and *type inference* described on Slide 6. 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, together with *progress*, *type-preservation* and *safety* theorems (Slide 7). In this course we will look at more involved examples revolving around the notions of *parametric polymorphism* (Sections 2–4) and *dependent functions* (Sections 5 and 6).

Type checking, typeability, and type inference

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

- **Type-checking** problem: given Γ, e, and τ, is $\Gamma \vdash e : \tau$ derivable in the type system?
- **Typeability** problem: given Γ and e, is there any τ for which $\Gamma \vdash e : \tau$ is derivable in the type system?

Solving the second problem usually involves devising a **type inference algorithm** computing a τ for each Γ and e (or failing, if there is none).

Slide 6

Progress, type preservation & safety

Recall that the simple, typed imperative language considered in CST Part IB *Semantics of Programming Languages* satisfies:

Progress. If $\Gamma \vdash e : \tau$ and $dom(\Gamma) \subseteq dom(s)$, then either e is a value, or there exist e', s' such that $\langle e, s\rangle \to \langle e', s'\rangle$.

Type preservation. If $\Gamma \vdash e : \tau$ and $dom(\Gamma) \subseteq dom(s)$ and $\langle e, s\rangle \to \langle e', s'\rangle$, then $\Gamma \vdash e' : \tau$ and $dom(\Gamma) \subseteq dom(s')$.

Hence well-typed programs don't get stuck:

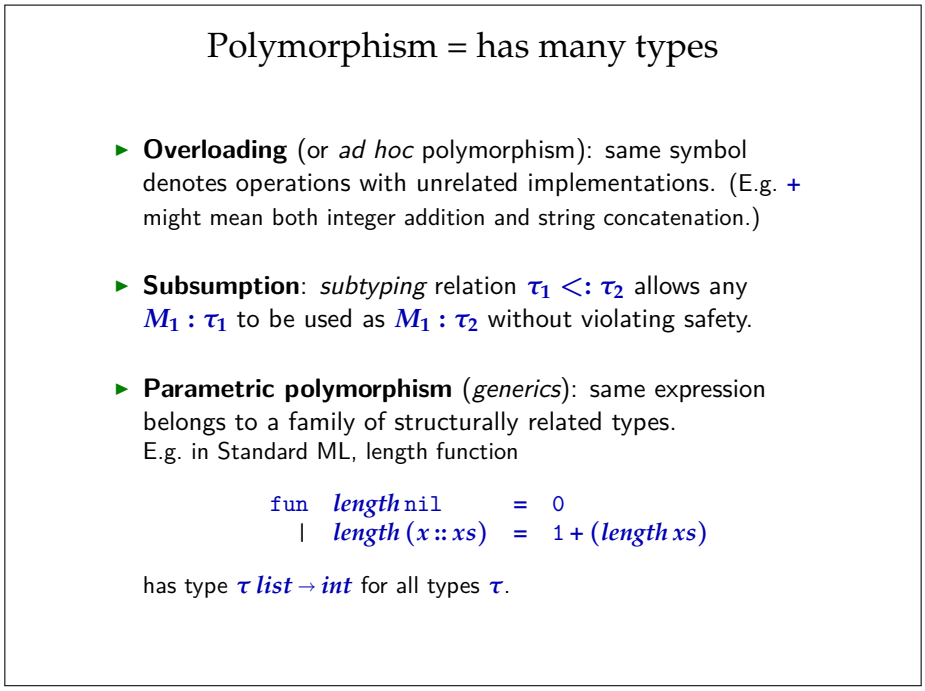
Safety. If $\Gamma \vdash e : \tau$, $dom(\Gamma) \subseteq dom(s)$ and $\langle e, s\rangle \to^* \langle e', s'\rangle$, then either e' is a value, or there exist e'', s'' such that $\langle e', s'\rangle \to \langle e'', s''\rangle$.

Slide 7

2 ML Polymorphism

As indicated in the Introduction, static type-checking is regarded by many as an important aid to building large, well-structured, and reliable software systems. On the other hand, early forms of static typing, for example as found in Pascal, tended to hamper the ability to write *generic code*. For example, a procedure for sorting lists of one type of data could not be applied to lists of a different type of data. It is natural to want a *polymorphic* sorting procedure—one which operates (uniformly) on lists of several different types. The potential significance for programming languages of this phenomenon of *polymorphism* was first emphasised by Strachey (1967), who identified several different varieties: see Slide 8. Here we will concentrate on *parametric polymorphism*, also known as *generics*.

One way to get parametric polymorphism 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 no 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 mathematically precise formalisation of a type system and its associated type inference algorithm for a small fragment of ML, called *Mini-ML*.



Slide 8

2.1 Mini-ML type system

As indicated on Slide 9, to formalise parametric polymorphism, we introduce *type variables*. An interactive ML system will just display $\alpha\ list \to int$ as the type of the *length* function (Slide 8), leaving the universal quantification over α implicit. However, when it comes to formalising the ML type system (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 10. The expression $(f\ \texttt{true}) :: (f\ \texttt{nil})$ has type *bool list*, given some assumption about the type of the variable f. Two possible such forms of assumption are shown on Slide 11. Here we are interested in the second possibility since it leads to a type system with very useful properties. The form of Mini-ML typing judgements is shown on Slide 12. The particular grammar of types and type schemes that we will use for Mini-ML is shown on Slide 13.

Type variables and type schemes in Mini-ML

To formalise statements like

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

we introduce **type variables** α (i.e. variables for which types may be substituted) and write

$$length : \forall\alpha\ (\alpha\ list \to int).$$

$\forall\alpha\ (\alpha\ list \to int)$ is an example of a **type scheme**.

Slide 9

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

For example in

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

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

in $(f\ \texttt{true})$, f has type $bool \to bool$
in $(f\ \texttt{nil})$, f has type $bool\ list \to bool\ list$

Overall, the expression has type $bool\ list$.

Slide 10

Forms of hypothesis in typing judgements

- *Ad hoc* (overloading):

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

  Appropriate for expressions that have different behaviour at different types.

- *Parametric*:

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

  Appropriate if expression behaviour is uniform for different type instantiations.

ML uses parametric hypotheses (type schemes) in its typing judgements.

Slide 11

Mini-ML typing judgement

takes the form

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

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 $dom(\Gamma) = \{x_1, \ldots, x_n\}$ (mutually distinct variables) and maps each x_i to the type scheme σ_i for $i = 1 \ldots n$.)

- M is a Mini-ML expression

- τ is a Mini-ML type.

Slide 12

Mini-ML types and type schemes

Types

$$\begin{array}{rcll} \tau & ::= & \alpha & \text{type variable} \\ & | & \mathit{bool} & \text{type of booleans} \\ & | & \tau \to \tau & \text{function type} \\ & | & \tau\ \mathit{list} & \text{list type} \end{array}$$

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

Type Schemes

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

where A ranges over finite subsets of the set **TyVar**.

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

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

When $A = \{\}$ is empty, we write $\forall A\,(\tau)$ just as τ. In other words, **we regard the set of types as a subset of the set of type schemes by identifying the type τ with the type scheme $\forall\{\}\,(\tau)$.**

Slide 13

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. Since 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 cannot be used where an ML type is expected.** 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.

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

Specialising type schemes to types

A type τ is a **specialisation** of a type scheme $\sigma = \forall \alpha_1, \ldots, \alpha_n\,(\tau')$ if τ can be obtained from the type τ' by simultaneously substituting some types τ_i for the type variables α_i $(i = 1, \ldots, n)$:

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

In this case we write $\boxed{\sigma \succ \tau}$

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

The converse relation is called **generalisation**: a type scheme σ generalises a type τ if $\sigma \succ \tau$.

Slide 14

Slide 14 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 specialising type schemes:

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

However

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

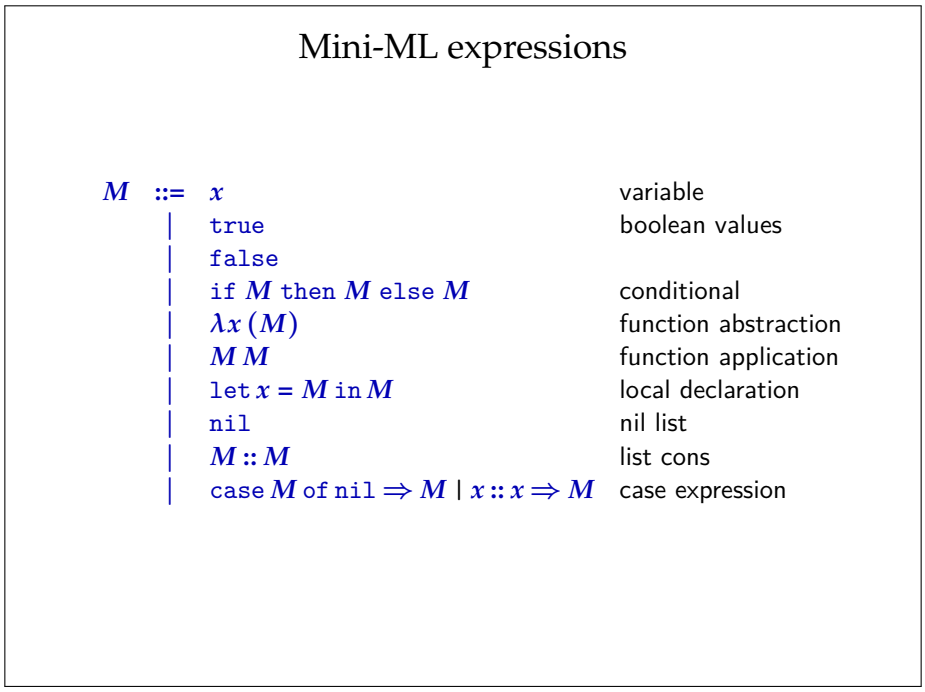
This is because in a substitution $\tau[\tau'/\alpha]$, by definition we have to replace *all* occurrences in τ of the type variable α by τ'. Thus when $\tau = \alpha \to \alpha$, there is no type τ' for which $\tau[\tau'/\alpha]$ is the type $(\alpha \to \alpha) \to \alpha$. (Simply because in the syntax 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:

$$\forall\alpha_1, \alpha_2\,(\alpha_1 \to \alpha_2) \succ \alpha\ \mathit{list} \to \mathit{bool}.$$

However

$$\forall\alpha_1\,(\alpha_1 \to \alpha_2) \not\succ \alpha\ \mathit{list} \to \mathit{bool}$$

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.



Slide 15

Just as we only consider a small subset of ML types, for Mini-ML we restrict attention to typings for a small subset of ML expressions, M, generated by the grammar on Slide 15. We use a non-standard syntax compared with the definition in Milner 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 16–18. As usual, the axioms and rules on Slides 16–18 are schematic: Γ stands for any type environment; x and ℓ for any variables; M, M_1, M_2, M_3, L, N and C for any expressions; and τ, τ', τ_1 and τ_2 for any types.

Note the following points about the type system defined on Slides 16–18.

(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` $\Rightarrow M_2 \mid x_1 :: x_2 \Rightarrow M_3$, any free occurrences of the variables x_1 and x_2 in M_3 become bound in the `case`-expression. **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), $\Gamma, x : \tau, \ell : \tau\,list$ really means $\Gamma, x : \forall\{\,\}\,(\tau), \ell : \forall\{\,\}\,(\tau\,list)$. (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 the rule consists of all type variables in τ that do not occur freely in any type scheme assigned in Γ. Slide 19 gives an example instance of the (let) rule.

Mini-ML type system, I

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

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

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

Slide 16

Mini-ML type system, II

$$\textbf{(nil)}\ \frac{}{\Gamma \vdash \texttt{nil} : \tau\ list}$$

$$\textbf{(cons)}\ \frac{\Gamma \vdash M : \tau \quad \Gamma \vdash L : \tau\ list}{\Gamma \vdash M :: L : \tau\ list}$$

$$\textbf{(case)}\ \frac{\Gamma \vdash L : \tau\ list \quad \Gamma \vdash N : \tau' \quad \Gamma, x:\tau, \ell : \tau\ list \vdash C : \tau'}{\Gamma \vdash (\texttt{case } L \texttt{ of nil} \Rightarrow N \mid x :: \ell \Rightarrow C) : \tau'}\ \text{if } x \neq \ell \text{ and } x, \ell \notin dom(\Gamma)$$

Slide 17

Mini-ML type system, III

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

$$(\textbf{app})\ \frac{\Gamma \vdash M : \tau_1 \to \tau_2 \qquad \Gamma \vdash N : \tau_1}{\Gamma \vdash M\,N : \tau_2}$$

$$(\textbf{let})\ \frac{\Gamma \vdash M_1 : \tau \qquad \Gamma, x : \forall A\,(\tau) \vdash M_2 : \tau'}{\Gamma \vdash (\texttt{let}\, x = M_1\, \texttt{in}\, M_2) : \tau'}\ \text{if } x \notin dom(\Gamma) \text{ and } A = ftv(\tau) - ftv(\Gamma)$$

Definition. We write $\boxed{\Gamma \vdash M : \forall A\,(\tau)}$ to mean $\Gamma \vdash M : \tau$ is derivable from the Mini-ML typing rules and that $A = ftv(\tau) - ftv(\Gamma)$.

(So **(let)** is equivalent to $\dfrac{\Gamma \vdash M_1 : \sigma \qquad \Gamma, x : \sigma \vdash M_2 : \tau'}{\Gamma \vdash (\texttt{let}\, x = M_1\, \texttt{in}\, M_2) : \tau'}$ if $x \notin dom(\Gamma)$.)

Slide 18

Example of using the (let) rule

$$(\textbf{let})\ \frac{\Gamma \vdash M_1 : \tau \qquad \Gamma, x : \forall A\,(\tau) \vdash M_2 : \tau'}{\Gamma \vdash (\texttt{let}\, x = M_1\, \texttt{in}\, M_2) : \tau'}\ \text{if } x \notin dom(\Gamma) \text{ and } A = ftv(\tau) - ftv(\Gamma)$$

If $\Gamma \vdash M_1 : \tau$ is $\boxed{y : \beta, z : \forall\gamma\,(\gamma \to \gamma \to bool) \vdash \lambda u\,(y) : \alpha \to \beta}$

then $A = \{\alpha, \beta\} - \{\beta\} = \{\alpha\}$ and $\forall A\,(\tau) = \forall\alpha\,(\alpha \to \beta)$.

So if $\Gamma, x : \forall A\,(\tau) \vdash M_2 : \tau'$ is

$\boxed{y : \beta, z : \forall\gamma\,(\gamma \to \gamma \to bool), x : \forall\alpha\,(\alpha \to \beta) \vdash z\,(x\,y)\,(x\,\texttt{nil}) : bool}$

then applying **(let)** yields

$\boxed{y : \beta, z : \forall\gamma\,(\gamma \to \gamma \to bool) \vdash \texttt{let}\, x = \lambda u\,(y)\, \texttt{in}\, z\,(x\,y)\,(x\,\texttt{nil}) : bool}$

Slide 19

The Mini-ML rules are used to inductively generate the *typing relation*. We say that $\Gamma \vdash M : \forall A\,(\tau)$ is *derivable* (or *provable*, or *valid*) in the type system if there is a proof of $\Gamma \vdash M : \tau$ using the axioms and rules and $A = ftv(\tau) - ftv(\Gamma)$ (see the definition on Slide 18). When $\Gamma = \{\,\}$ we just write

$$\vdash M : \forall A\,(\tau) \tag{1}$$

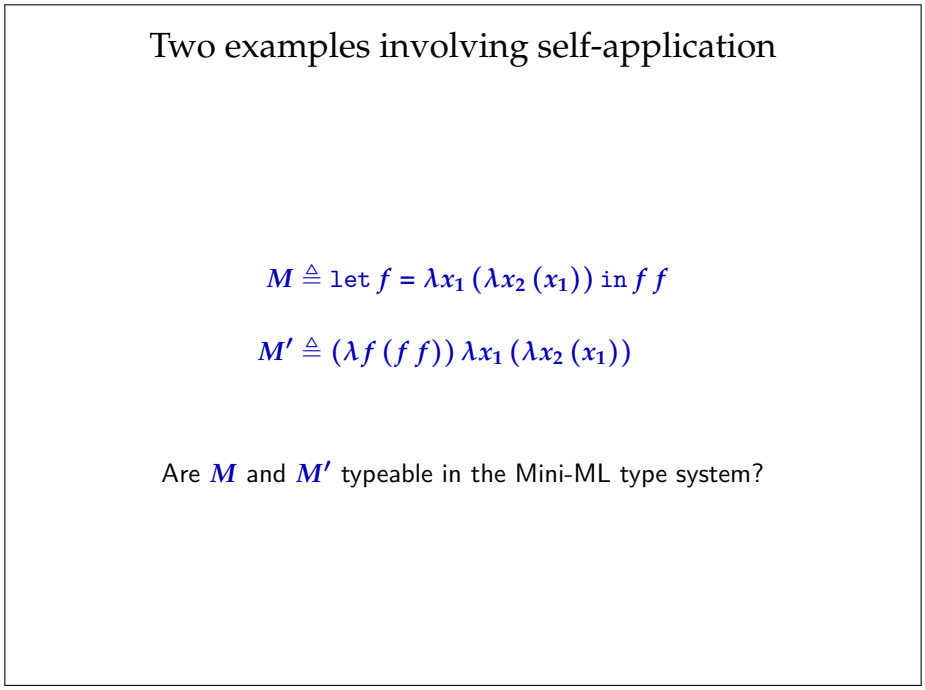
for $\{\,\} \vdash M : \forall A\,(\tau)$ and say that the (necessarily closed – see Exercise 2) expression M is *typeable* in Mini-ML with type scheme $\forall A\,(\tau)$.

Example 2. We verify that the example of polymorphism of `let`-bound variables given on Slide 10 has the type claimed there (and illustrate one style of setting out proofs of typing – as a list of judgements successively derived by applying the rules; proof trees exhibit the structure of typing proofs more clearly, but are less convenient to fit on a page!).

1. $x : \alpha \vdash x : \alpha$ by (var), since $\forall\{\,\}\,(\alpha) \succ \alpha$.
2. $\{\,\} \vdash \lambda x\,(x) : \alpha \to \alpha$ by (fn) on 1.
3. $f : \forall\alpha\,(\alpha \to \alpha) \vdash f : \mathit{bool} \to \mathit{bool}$ by (var), since $\forall\alpha\,(\alpha \to \alpha) \succ \mathit{bool} \to \mathit{bool}$.
4. $f : \forall\alpha\,(\alpha \to \alpha) \vdash \texttt{true} : \mathit{bool}$ by (bool).
5. $f : \forall\alpha\,(\alpha \to \alpha) \vdash f\ \texttt{true} : \mathit{bool}$ by (app) on 3 and 4.
6. $f : \forall\alpha\,(\alpha \to \alpha) \vdash f : \mathit{bool\ list} \to \mathit{bool\ list}$ by (var), since $\forall\alpha\,(\alpha \to \alpha) \succ \mathit{bool\ list} \to \mathit{bool\ list}$.
7. $f : \forall\alpha\,(\alpha \to \alpha) \vdash \texttt{nil} : \mathit{bool\ list}$ by (nil).
8. $f : \forall\alpha\,(\alpha \to \alpha) \vdash f\ \texttt{nil} : \mathit{bool\ list}$ by (app) on 6 and 7.
9. $f : \forall\alpha\,(\alpha \to \alpha) \vdash (f\ \texttt{true}) :: (f\ \texttt{nil}) : \mathit{bool\ list}$ by (cons) on 5 and 8.
10. $\{\,\} \vdash \texttt{let}\ f = \lambda x\,(x)\ \texttt{in}\ (f\ \texttt{true}) :: (f\ \texttt{nil}) : \mathit{bool\ list}$ by (let) on 2 and 9, since $\{\alpha\} = \mathit{ftv}(\alpha \to \alpha) - \mathit{ftv}(\{\,\})$.

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 6) 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.



Slide 20

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

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

Given a typing environment Γ and an expression M, how can we decide whether or not there is a type scheme $\forall A\,(\tau)$ for which $\Gamma \vdash M : \forall A\,(\tau)$ 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 $A = ftv(\tau) - ftv(\Gamma)$ and $\Gamma \vdash M : \tau$ is derivable from the rules on Slides 16–18, then the outermost form of the expression M dictates which must be the last axiom or rule used in the proof of $\Gamma \vdash M : \tau$. Consequently, as we try to build a proof of a typing judgement $\Gamma \vdash M : \tau$ from the bottom up, the structure of the expression M determines the shape of the tree together with which rules are used at its nodes and which axioms at its leaves. For example, for the particular expression M given on Slide 20, any proof of $\{\,\} \vdash M : \tau_1$ from the axioms and rules, has to look like the tree given in Figure 1. Node (C0) is supposed to be an instance of the (let) rule; nodes (C1) and (C2) instances of the (fn) rule; leaves (C3), (C5), and (C6) instances of the (var $\succ$) axiom; and node (C4) an instance of the (app) rule. For these to be valid instances the constraints (C0)–(C6) listed on Slide 21 have to be satisfied.

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

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

Slide 21

Thus M is typeable if and only if we can find types $\tau_1, \ldots, \tau_8$ satisfying the constraints on Slide 21. 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), $A = ftv(\tau_2) =$

$ftv(\alpha_1 \to (\alpha_2 \to \alpha_1)) = \{\alpha_1, \alpha_2\}$. 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

$$\tau_9 \to (\tau_{10} \to \tau_9) = \tau_8 \to \tau_1 \tag{C7}$$
$$\tau_{11} \to (\tau_{12} \to \tau_{11}) = \tau_8. \tag{C8}$$

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 21 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 16–18, i.e. we do have

$$\vdash \texttt{let}\ f = \lambda x_1\, (\lambda x_2\, (x_1))\ \texttt{in}\ f\, f : \forall \alpha_3, \alpha_4, \alpha_5\, (\alpha_3 \to (\alpha_4 \to (\alpha_5 \to \alpha_4))) \tag{2}$$

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

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

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 (2), 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 22. It is worth pointing out that in the presence of (a), the converse of condition (b) on Slide 22 holds: if $\vdash M : \forall A\, (\tau)$ and $\forall A\, (\tau) \succ \tau'$, then $\vdash M : \forall A'\, (\tau')$ (where $A' = ftv(\tau')$). This is a consequence of the substitution property of valid Mini-ML typing judgements given in the Exercises.

Slide 23 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).

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

Constraints:

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

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

Principal type schemes for closed expressions

A 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 type scheme $\forall A'\,(\tau')$, if $\vdash M : \forall A'\,(\tau')$, then $\forall A\,(\tau) \succ \tau'$

Slide 22

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, in the worst case 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 24 (taken from Mairson (1990)) has a principal type scheme which would take hundreds of pages to print out. It seems that such pathology does not arise naturally, and that the type checking phase of an ML compiler is not a bottle neck in practice. For more details about the

complexity of ML type inference see (Mitchell, 1996, Section 11.3.5).

Theorem (Hindley; Damas-Milner)

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 closed Mini-ML expression M as input, decides whether or not it is typeable and returns a principal type scheme if it is.

Slide 23

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

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

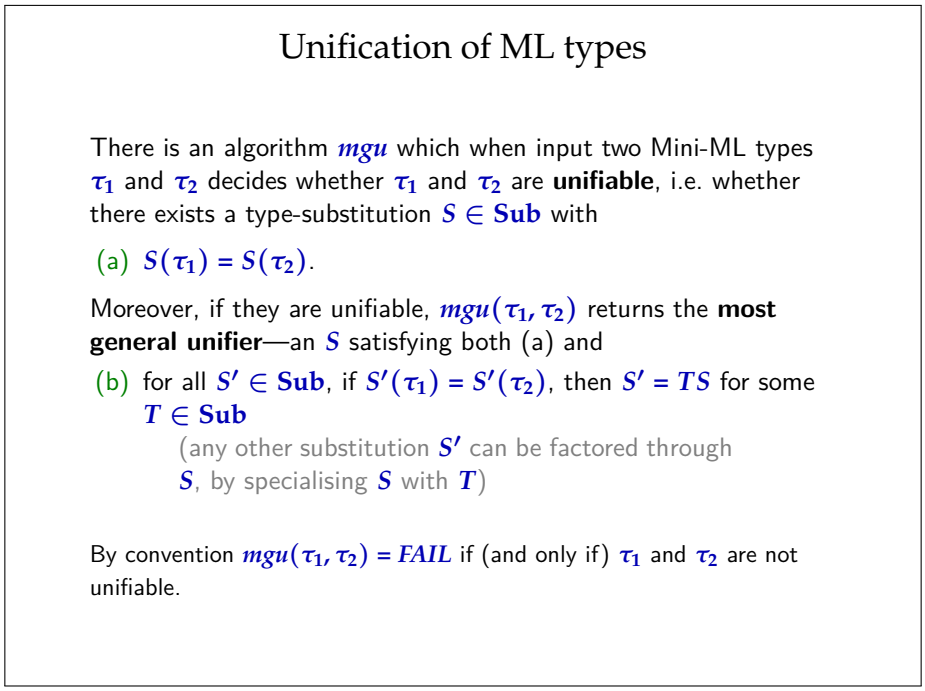
Slide 24

2.4 A type inference algorithm

The aim of this subsection is to sketch the proof of the Hindley-Damas-Milner theorem stated on Slide 23, by describing an algorithm, *pt*, for deciding typeability and returning a most gen-

eral 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 25. Although we won't bother to give an implementation of *mgu* here (see for example (Rydeheard and Burstall, 1988, Chapter 8), (Mitchell, 1996, Section 11.2.2), or (Aho et al., 1986, Section 6.7) for more details), we do need to explain the notation for type substitutions introduced on Slide 25.



Slide 25

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 $S \in \text{Sub}$ is the finite set of variables

$$dom(S) \triangleq \{\alpha \in \text{TyVar} \mid S(\alpha) \neq \alpha\}$$

Given a type substitution S, the effect of applying the substitution to a type is written $S\,\tau$; thus if $dom(S) = \{\alpha_1, \ldots, \alpha_n\}$ 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 14. 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 $S, T \in \text{Sub}$ means first apply S and then T. Thus by definition TS is

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

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 principal type scheme for open expressions in the presence of a non-empty typing environment. This is done on Slide 26. 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 28; we leave the proofs as exercises. To compute principal type schemes it suffices to compute *principal solutions* in the sense of Slide 26. (Slide 27 illustrates this notion of a *solution to a typing problem*.) 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 22 (see the exercises).

Principal type schemes for open expressions

A **solution** for the typing problem $\Gamma \vdash M : ?$ is a pair (S, σ) 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 $T \in \mathbf{Sub}$ with $TS = S'$ and $T(\sigma) \succ \sigma'$.

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

Slide 26

[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 $S(\alpha_i)$.

Example typing problem and solutions

Typing problem

$$x : \forall\alpha\,(\beta \to (\gamma \to \alpha)) \vdash x\,\mathtt{true} : ?$$

has solutions:

- $S_1 = \{\beta \mapsto bool\}, \sigma_1 = \forall\alpha\,(\gamma \to \alpha)$
- $S_2 = \{\beta \mapsto bool, \gamma \mapsto \alpha\}, \sigma_2 = \forall\alpha'\,(\alpha \to \alpha')$
- $S_3 = \{\beta \mapsto bool, \gamma \mapsto \alpha\}, \sigma_3 = \forall\alpha'\,(\alpha \to (\alpha' \to \alpha'))$
- $S_4 = \{\beta \mapsto bool, \gamma \mapsto bool\}, \sigma_3 = \forall\{\,\}\,(bool \to bool)$

Both (S_1, σ_1) and (S_2, σ_2) are in fact principal solutions.

Slide 27

Properties of the Mini-ML typing relation with respect to substitution and type scheme specialisation

- If $\Gamma \vdash M : \sigma$, then for any type substitution $S \in \mathbf{Sub}$

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

- If $\Gamma \vdash M : \sigma$ and $\sigma \succ \sigma'$, then

$$\Gamma \vdash M : \sigma'$$

Slide 28

Requirements for a 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 $S \in \mathbf{Sub}$ and a Mini-ML type τ, or the exception $FAIL$.

- If $\Gamma \vdash M : ?$ has a solution (cf. Slide 28), 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 29

How the principal typing algorithm pt works

$$pt(\Gamma \vdash M : ?) = (S, \tau) \mid FAIL$$

- Call pt recursively following the structure of M and guided by the typing rules, bottom-up.
- Thread substitutions sequentially and compose them together when returning from a recursive call.
- When types need to agree to satisfy a typing rule, use mgu (and pt returns $FAIL$ only if mgu does).
- When types are unknown, generate a fresh type variable.

Slide 30

- *Variables*: $pt(\Gamma \vdash x : ?) \triangleq$ let $\forall A\,(\tau) = \Gamma(x)$ in (Id, τ)
- `let`-*Expressions*: $pt(\Gamma \vdash \texttt{let}\ x = M_1\ \texttt{in}\ M_2 : ?) \triangleq$
  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_2S_1, τ_2)
- *Booleans* (B = `true`, `false`): $pt(\Gamma \vdash B : ?) \triangleq (Id, bool)$
- *Conditionals*: $pt(\Gamma \vdash \texttt{if}\ M_1\ \texttt{then}\ M_2\ \texttt{else}\ M_3 : ?) \triangleq$
  let $(S_1, \tau_1) = pt(\Gamma \vdash M_1 : ?)$ in
  let $S_2 = mgu(\tau_1, bool)$ in
  let $(S_3, \tau_3) = pt(S_2S_1\,\Gamma \vdash M_2 : ?)$ in
  let $(S_4, \tau_4) = pt(S_3S_2S_1\,\Gamma \vdash M_3 : ?)$ in
  let $S_5 = mgu(S_4\,\tau_3, \tau_4)$ in $(S_5S_4S_3S_2S_1, S_5\,\tau_4)$

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

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

Function abstractions: $pt(\Gamma \vdash \lambda x\,(M) : ?) \triangleq$
let α = fresh in
let $(S, \tau) = pt(\Gamma, x : \alpha \vdash M : ?)$ in $(S, S(\alpha) \to \tau)$

Function applications: $pt(\Gamma \vdash M_1\,M_2 : ?) \triangleq$
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 α = fresh in
let $S_3 = mgu(S_2\,\tau_1, \tau_2 \to \alpha)$ in $(S_3S_2S_1, S_3(\alpha))$

Slide 31

Slide 29 specifies what is required of the principal typing algorithm, *pt*; and Slide 30 indicates, in general terms, how to meet the specification. One possible implementation, written in somewhat informal pseudocode (and leaving out the cases for `nil`, cons, and `case`-expressions), is sketched on Slide 31 and in Figure 3.[4] Note the following points about the definitions on Slide 31 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 $A \cap ftv(\Gamma) = \{\,\}$.

[4] An implementation in Fresh OCaml (`www.cl.cam.ac.uk/users/amp12/fresh-ocaml/`) can be found on the course web page. The Fresh OCaml code is remarkably close to the informal pseudocode given here, because of Fresh OCaml's facilities for dealing with binding operations and fresh names.

(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) The clauses of the definition for `nil`, cons, and `case`-expressions are left as exercises.

(iv) We do not give the proof that the definition in Figure 3 is correct (i.e. meets the specification on Slide 29). 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*.

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

3 Polymorphic Reference Types

3.1 The problem

Recall from the Introduction that an important purpose of type systems is to provide safety via *type soundness* results (Slide 3). 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, specifically its 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 13) with a unit type (a type with a single value) and *reference* types; and correspondingly, we augment the grammar for Mini-ML expressions (Slide 15) with a unit value, and operations for reference creation, dereferencing and assignment. These additions are shown on Slide 32. We call the resulting language Midi-ML. The typing rules for these new forms of expression are given on Slide 33.

ML types and expressions for mutable references

$$
\begin{array}{rcll}
\tau & ::= & \ldots & \\
 & | & \mathit{unit} & \text{unit type} \\
 & | & \tau\ \mathit{ref} & \text{reference type} \\
 & & & \\
M & ::= & \ldots & \\
 & | & () & \text{unit value} \\
 & | & \texttt{ref}\ M & \text{reference creation} \\
 & | & !M & \text{dereference} \\
 & | & M := M & \text{assignment}
\end{array}
$$

Slide 32

Midi-ML's extra typing rules

$$\textbf{(unit)}\ \frac{}{\Gamma \vdash () : \mathit{unit}}$$

$$\textbf{(ref)}\ \frac{\Gamma \vdash M : \tau}{\Gamma \vdash \mathtt{ref}\ M : \tau\ \mathit{ref}}$$

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

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

Slide 33

Example

The expression

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

has type ***unit***.

Slide 34

Example 5. Here is an example of the typing rules on Slide 33 in use. The expression given on Slide 34 has type *unit*.

Proof. This can be deduced by applying the (let) rule (Slide 18) to the judgements

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

The first of these judgements has the following proof:

$$\dfrac{\dfrac{\dfrac{}{x : \alpha \vdash x : \alpha}\,(\text{var}\succ)}{\{\,\} \vdash \lambda x\,(x) : \alpha \to \alpha}\,(\text{fn})}{\{\,\} \vdash \texttt{ref}\;\lambda x\,(x) : (\alpha \to \alpha)\;ref}\,(\text{ref})$$

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

$$r : \forall\alpha\,((\alpha \to \alpha)\;ref) \vdash r := \lambda x'\,(\texttt{ref}\;!x') : unit \tag{3}$$

$$r : \forall\alpha\,((\alpha \to \alpha)\;ref), u : unit \vdash (!r)() : unit \tag{4}$$

Writing Γ for the typing environment $\{r : \forall\alpha\,((\alpha \to \alpha)\;ref)\}$, the proof of (3) is

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

while the proof of (4) is

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

□

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. To see this we need to define the operational semantics of Midi-ML.

Midi-ML transition system

Small-step transition relations

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

where

- M, M' range over Midi-ML expressions
- s, s' range over **states** = finite functions
  $s = \{x_1 \mapsto V_1, \ldots, x_n \mapsto V_n\}$ mapping variables x_i to **values** V_i:

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

- configurations $\langle M, s\rangle$ are required to satisfy that the free variables of expression M are in the domain of definition of the state s
- symbol FAIL represents a run-time error

are inductively defined by syntax-directed rules...

Slide 35

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
- $\langle \texttt{let } x = V \texttt{ in } M, s\rangle \to \langle M[V/x], s\rangle$
- $\langle \texttt{case nil of nil} \Rightarrow M \mid x_1 :: x_2 \Rightarrow M', s\rangle \to \langle M, s\rangle$
- $\langle \texttt{case } V_1 :: V_2 \texttt{ of nil} \Rightarrow M \mid x_1 :: x_2 \Rightarrow M', s\rangle \to \langle M'[V_1/x_1, V_2/x_2], s\rangle$
- $\langle \texttt{case } V \texttt{ of nil} \Rightarrow M \mid x_1 :: x_2 \Rightarrow M', s\rangle \to \mathit{FAIL}$, if V is neither `nil` nor a cons-value
- $$\frac{\langle M, s\rangle \to \langle M', s'\rangle}{\langle \mathcal{E}[M], s\rangle \to \langle \mathcal{E}[M'], s'\rangle}$$
- $$\frac{\langle M, s\rangle \to \mathit{FAIL}}{\langle \mathcal{E}[M], s\rangle \to \mathit{FAIL}}$$

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*:

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

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

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

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

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

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

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

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

The operational semantics of Midi-ML uses an inductively defined transition relation of the form shown on Slide 35 and which is inductively defined in Figure 4 and Slide 36 (using the rather terse 'evaluation contexts' style of Wright and Felleisen (1994)). (The notation $s[x \mapsto V]$ used on Slide 36 means the state with domain of definition $dom(s) \cup \{x\}$ mapping x to V and otherwise acting like s.)

$$\left\langle \begin{array}{l} \texttt{let}\ r = \texttt{ref}\ \lambda x\,(x)\ \texttt{in} \\ \texttt{let}\ u = (r := \lambda x'\,(\texttt{ref}\ !x'))\ \texttt{in}\ (!r)()\ ,\ \{\} \end{array} \right\rangle$$

$$\to^* \quad \langle \texttt{let}\ u = (r := \lambda x'\,(\texttt{ref}\ !x'))\ \texttt{in}\ (!r)()\ ,\ \{r \mapsto \lambda x\,(x)\}\rangle$$

$$\to^* \quad \langle (!r)()\ ,\ \{r \mapsto \lambda x'\,(\texttt{ref}\ !x')\}\rangle$$

$$\to \quad \langle \lambda x'\,(\texttt{ref}\ !x')\ ()\ ,\ \{r \mapsto \lambda x'\,(\texttt{ref}\ !x')\}\rangle$$

$$\to \quad \langle \texttt{ref}\ !()\ ,\ \{r \mapsto \lambda x'\,(\texttt{ref}\ !x')\}\rangle$$

$$\to \quad FAIL$$

Slide 37

Returning to the expression on Slide 34, it evaluates as shown on Slide 37. 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 $\lambda x'\,(\texttt{ref}\,!x')$ 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 $(r := \lambda x'\,(\texttt{ref}\,!x')); (!r)()$ 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 $\lambda x'\,(\texttt{ref}\,!x')$, 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. Thus we have

$$\langle \texttt{let}\, r = \texttt{ref}\, \lambda x\,(x)\, \texttt{in}\, \texttt{let}\, u = (r := \lambda x'\,(\texttt{ref}\,!x'))\, \texttt{in}\, (!r)(), \{\,\}\rangle \to \mathit{FAIL}$$

3.2 Restoring type soundness

The root of the problem described in the previous section lies in the fact that typing expressions like $\texttt{let}\, r = \texttt{ref}\, M_1\, \texttt{in}\, M_2$ 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 $\lambda x\,(\texttt{ref}\, x)$ becomes $\forall \_\alpha\,(\_\alpha \to \_\alpha\, \mathit{ref})$, rather than $\forall \alpha\,(\alpha \to \alpha\, \mathit{ref})$. Furthermore, and crucially, the (let) rule (Slide 18) is restricted by requiring that when the type scheme $\sigma = \forall A\,(\tau)$ assigned to M_1 is such that A contains imperative type variables, then M_1 must be a value (and hence in particular its evaluation does not create any fresh storage locations).

This solution has the advantage that in the new system the typeability of expressions not involving references is just the same as in the old system. However, it has the disadvantage that the type system makes distinctions between expressions which are behaviourally equivalent (i.e. which should be contextually equivalent). For example there are many list-processing functions that can be defined in the pure functional fragment of ML by recursive definitions, but which have more efficient definitions using local references. Unfortunately, if the type scheme of the former is something like $\forall \alpha\,(\alpha\, \mathit{list} \to \alpha\, \mathit{list})$, 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 38.[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 16–18 and 33, except that the applicability of the (let) rule is restricted as on Slide 38.

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

Value-restricted typing rule for `let`-expressions

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

$(\dagger)$ 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

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

Slide 38

Example 6. The expression on Slide 34 is not typeable in the type system for Midi-ML resulting from replacing rule (let) by the value-restricted rule (letv) on Slide 38 (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 $\texttt{ref}\,\lambda x\,(x)$ is *not* a value, the rule has to be applied with $A = \{\,\}$. This entails trying to type

$$\texttt{let}\, u = (r := \lambda x'\,(\texttt{ref}\,!x'))\,\texttt{in}\,(!r)() \tag{5}$$

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 (5) 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 39. 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. I do not give the details; 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

$(\mathbf{var}\succ) + (\mathbf{bool}) + (\mathbf{if}) + (\mathbf{nil}) + (\mathbf{cons}) + (\mathbf{case}) + (\mathbf{fn}) + (\mathbf{app}) + (\mathbf{unit}) + (\mathbf{ref}) + (\mathbf{get}) + (\mathbf{set}) + (\mathbf{letv})$

then **evaluation of M does not fail**,
i.e. there is no sequence of transitions of the form

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

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

Slide 39

In Midi-ML's value-restricted type system, some expressions that were typeable using $(\mathbf{let})$ become untypeable using $(\mathbf{letv})$.

For example (exercise):

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

But one can often[1] use η-expansion

replace M by $\lambda x\,(M\,x)$ (where $x \notin fv(M)$)

or β-reduction

replace $(\lambda x\,(M))\,N$ by $M[N/x]$

to get around the problem.

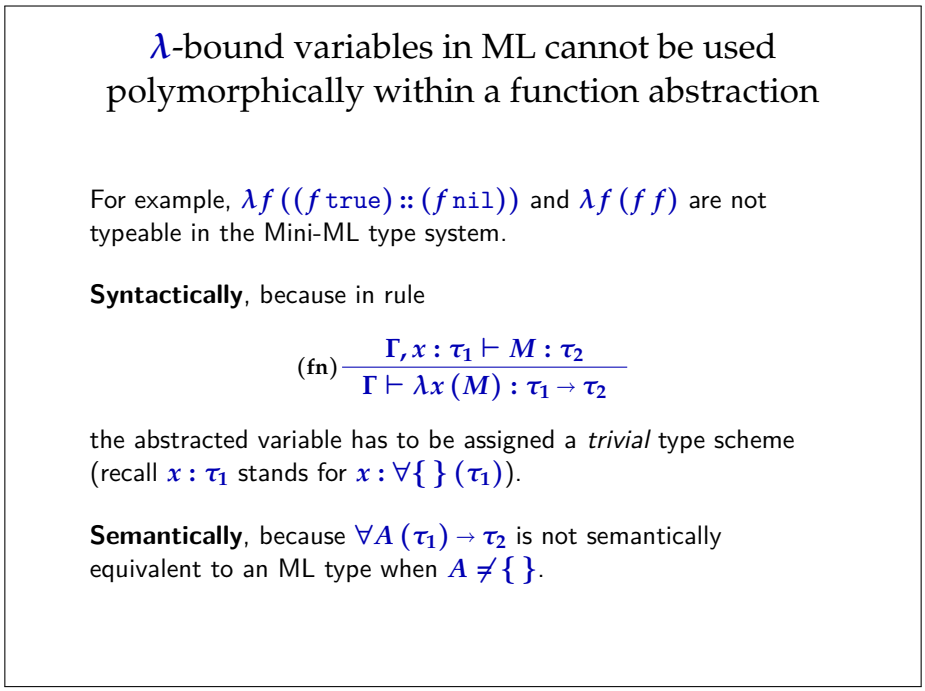
(**1** These transformations do not always preserve meaning [contextual equivalence].)

Slide 40

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 (Slide 40). 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* (PLC), 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.



Slide 41

4.1 From type schemes to polymorphic types

We have seen examples (Example 2 and the first example on Slide 20) of the fact that the ML type system permits `let`-bound variables to be used polymorphically within the body of a `let`-expression. As Slide 41 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 via *type soundness* (Slide 3). Use of expressions such as those mentioned on Slide 41 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 42. 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;

- $\Gamma = \{x_1 : \pi_1, \ldots, x_n : \pi_n\}$ is a finite function from variables to polymorphic types.

In order to make full use of the mixing of $\to$ and $\forall$ present in polymorphic types we have to replace the axiom (var $\succ$) of Slide 16 by the rules shown on Slide 43. (These are in fact versions for polymorphic types of valid properties of 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.

Monomorphic types ...

$$\tau ::= \alpha \mid bool \mid \tau \to \tau \mid \tau\ list$$

... and **type schemes**

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

Polymorphic types

$$\pi ::= \alpha \mid bool \mid \pi \to \pi \mid \pi\ list \mid \forall\alpha\ (\pi)$$

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 42

Identity, Generalisation and Specialisation

$$(\mathbf{id})\ \frac{}{\Gamma \vdash x : \pi} \text{ if } (x : \pi) \in \Gamma$$

$$(\mathbf{gen})\ \frac{\Gamma \vdash M : \pi}{\Gamma \vdash M : \forall\alpha\ (\pi)} \text{ if } \alpha \notin ftv(\Gamma)$$

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

Slide 43

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

$$\{\,\} \vdash \lambda f\,((f\,\texttt{true}) :: (f\,\texttt{nil})) : \forall\alpha\,(\alpha \to \alpha) \to \mathit{bool\ list} \tag{6}$$

$$\{\,\} \vdash \lambda f\,(f\,f) : \forall\alpha\,(\alpha) \to \forall\alpha\,(\alpha). \tag{7}$$

Proof. The proof of (6) is rather easy to find and is left as an exercise. Here is a proof for (7):

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

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. □

ML + full polymorphic types
has undecidable type-checking

Fact (Wells, 1994). For the modified Mini-ML type system with

- full polymorphic types replacing types and type schemes
- **(id)** + **(gen)** + **(spec)** replacing **(var $\succ$)**

the type checking and typeability problems are undecidable.

Slide 44

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 44: 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 18).

The negative result on Slide 44 does not rule out the use of the polymorphic types of Slide 42 in programming languages, since one may consider *explicitly typed* languages (Slide 45) where the tagging of expressions with type information renders the typeability problem essentially trivial. A common view is 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.

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 45

PLC syntax

Types

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

Expressions

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

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

Slide 46

Functions on types

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

$F\,\tau$ 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 47

4.2 The Polymorphic Lambda Calculus (PLC) type system

The explicit type information we need to add to expressions to get syntax-directed versions of the (gen) and (spec) rules (Slide 43) 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 47. The polymorphic lambda calculus,

PLC, provides rather sparse means for defining such functions—for example there is no *type-case* construct that allows branching according to which type expression is input. As a result, PLC is really a calculus of *parametrically polymorphic* functions (cf. Slide 8). The PLC syntax is given on Slide 46. Its types, τ, are like the polymorphic types, π, given on Slide 42, 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.

Remark 8 (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:

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

It is also common to write a type specialisation by subscripting the type: $M_\tau \triangleq M\,\tau$.

Remark 9 (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, $ftv(\tau)$, of *free type variables of a type* τ, is given by

$$\begin{aligned}ftv(\alpha) &\triangleq \{\alpha\}\\ ftv(\tau_1 \to \tau_2) &\triangleq ftv(\tau_1) \cup ftv(\tau_2)\\ ftv(\forall\alpha\,(\tau)) &\triangleq ftv(\tau) - \{\alpha\}.\end{aligned}$$

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

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

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

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

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

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

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

Remark 10 (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 48. Its typing relation is the collection of such judgements inductively defined by the axiom and rules on Slide 49.

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 $i = 1 \ldots n$.)
- M is a PLC expression
- τ is a PLC type.

Slide 48

PLC type system

$$\textbf{(var)}\ \frac{}{\Gamma \vdash x : \tau}\ \text{if } (x : \tau) \in \Gamma$$

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

$$\textbf{(app)}\ \frac{\Gamma \vdash M : \tau_1 \to \tau_2 \qquad \Gamma \vdash M' : \tau_1}{\Gamma \vdash M\,M' : \tau_2}$$

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

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

Slide 49

An incorrect proof

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

Slide 50

Remark 11 (Side-condition on rule (gen**)).** To illustrate the force of the side-condition on rule (gen) on Slide 49, consider the last step of the incorrect proof on Slide 50. It is not a correct instance of the (gen) rule, because the concluding judgement, whose typing environment $\Gamma = \{x_1 : \alpha\}$, does not satisfy $\alpha \notin ftv(\Gamma)$ (since $ftv(\Gamma) = \{\alpha\}$ 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:

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

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 12. On Slide 45 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:

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

Example 13. There is no PLC type τ for which

$$\{\,\} \vdash \Lambda\alpha\ ((\lambda x : \alpha\ (x))\ \alpha) : \tau \tag{8}$$

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 (8) would have to look like

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

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 51

4.3 PLC type inference

As Examples 12 and 13 suggest, the type checking and typeability problems (Slide 6) 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 51. 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 52 and 53.[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 OCaml can be found on the course web page.

PLC type-checking algorithm, I

Variables

$typ(\Gamma, x : \tau \vdash x : ?) \triangleq \tau$

Function abstractions

$typ(\Gamma \vdash \lambda x : \tau_1\,(M) : ?) \triangleq$
$\quad \textbf{let}\ \ \tau_2 = typ(\Gamma, x : \tau_1 \vdash M : ?)\ \textbf{in}\ \tau_1 \to \tau_2$

Function applications

$typ(\Gamma \vdash M_1\, M_2 : ?) \triangleq$
$\quad \textbf{let}\ \tau_1 = typ(\Gamma \vdash M_1 : ?)\ \textbf{in}$
$\quad \textbf{let}\ \tau_2 = typ(\Gamma \vdash M_2 : ?)\ \textbf{in}$
$\quad \textbf{case}\ \tau_1\ \textbf{of}\quad \tau \to \tau' \ \mapsto\ \textbf{if}\ \tau = \tau_2\ \textbf{then}\ \tau'\ \textbf{else}\ \mathit{FAIL}$
$\qquad\qquad\quad |\qquad\ \ \_ \ \mapsto\ \mathit{FAIL}$

Slide 52

PLC type-checking algorithm, II

Type generalisations

$typ(\Gamma \vdash \Lambda\alpha\,(M) : ?) \triangleq$
$\quad \textbf{let}\ \tau = typ(\Gamma \vdash M : ?)\ \textbf{in}\ \forall\alpha\,(\tau)$

Type specialisations

$typ(\Gamma \vdash M\,\tau_2 : ?) \triangleq$
$\quad \textbf{let}\ \tau = typ(\Gamma \vdash M : ?)\ \textbf{in}$
$\quad \textbf{case}\ \tau\ \textbf{of}\quad \forall\alpha\,(\tau_1) \ \mapsto\ \tau_1[\tau_2/\alpha]$
$\qquad\qquad\quad |\qquad\ \ \_ \ \mapsto\ \mathit{FAIL}$

Slide 53

4.4 Datatypes in PLC

The aim of this subsection and the next 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 54.

Beta-reduction of PLC expressions

M **beta-reduces to** M' **in one step**, $M \to M'$ 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:

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

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

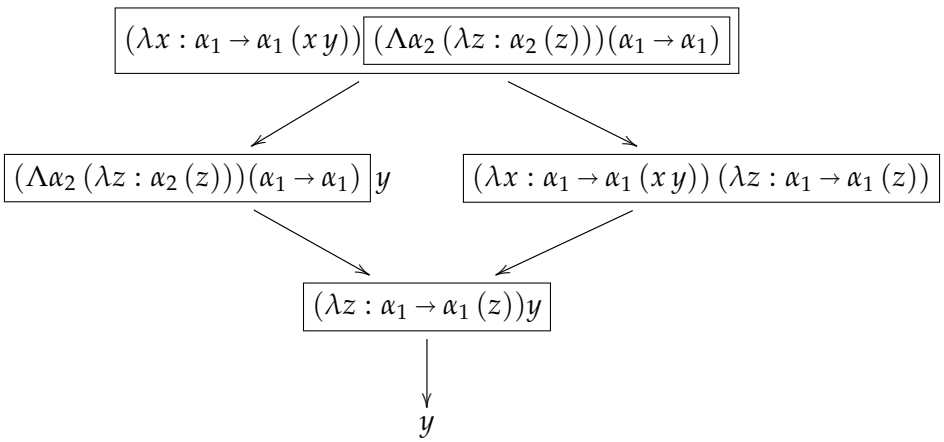
$M \to^* M'$ indicates a chain of finitely† many beta-reductions.

(† possibly zero – which just means M and M' are alpha-convertible).

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

Slide 54

Example 14. Here are some examples of beta-reductions. The various redexes are shown boxed. Clearly, the final expression y is in beta-normal form.



Properties of PLC beta-reduction on typeable expressions

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

Subject Reduction. If $M \to M'$, 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 55

PLC beta-conversion, $=_\beta$

By definition, $M =_\beta M'$ 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 56

Slide 55 lists some important properties of *typeable* PLC expressions that we state without proof. The first is a weak form of type soundness result (Slide 3) 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

[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.

'reducibility candidates'; if you are interested in seeing the details, look at (Girard, 1989, Chapter 14) for an accessible account of the proof.

Theorem 15. *The properties listed on Slide 55 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 56) 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).

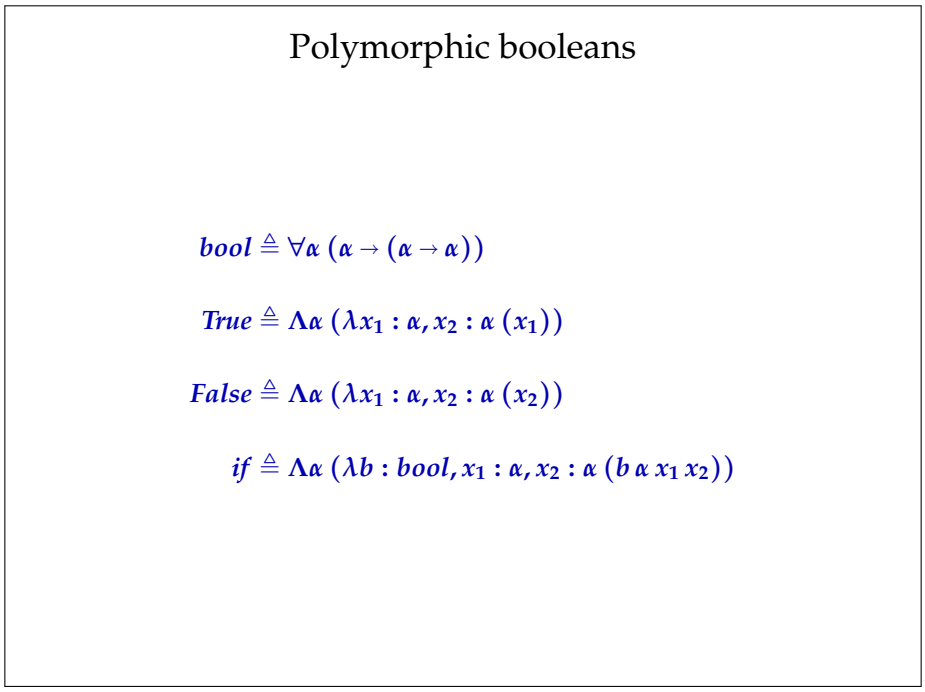
For (ii), we can use an algorithm which reduces the beta-redexes of each expression in any order until beta-normal forms are reached (in finitely many steps, by Strong Normalisation); these normal forms are equal (up to alpha-conversion) if and only if the original expressions are beta-convertible. (And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

Remark 16. 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

$$\Omega \triangleq (\lambda f : \alpha\,(f\,f))(\lambda f : \alpha\,(f\,f))$$

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 55 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.



Slide 57

Example 17 (Booleans). The PLC type corresponding to the ML datatype

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

is shown on Slide 57. 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 57 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 57. 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 16). 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.

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

Iteratively defined functions on finite lists

$A^* \triangleq$ finite lists of elements of the set A

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

$$g\, Nil = x'$$
$$g\, (x :: \ell) = f\, x\, (g\, \ell)$$

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

Slide 58

Polymorphic lists

$$\alpha\ list \triangleq \forall \alpha'\ (\alpha' \to (\alpha \to \alpha' \to \alpha') \to \alpha')$$

$$Nil \triangleq \Lambda \alpha, \alpha'\ (\lambda x' : \alpha', f : \alpha \to \alpha' \to \alpha'\ (x'))$$

$$\begin{aligned} Cons \triangleq \Lambda \alpha (\lambda x : \alpha, \ell : \alpha\ list (\Lambda \alpha' (\\ \lambda x' : \alpha', f : \alpha \to \alpha' \to \alpha' (\\ f\, x\, (\ell\, \alpha'\, x'\, f))))) \end{aligned}$$

Slide 59

Example 18 (Lists). The polymorphic type corresponding to the ML datatype

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

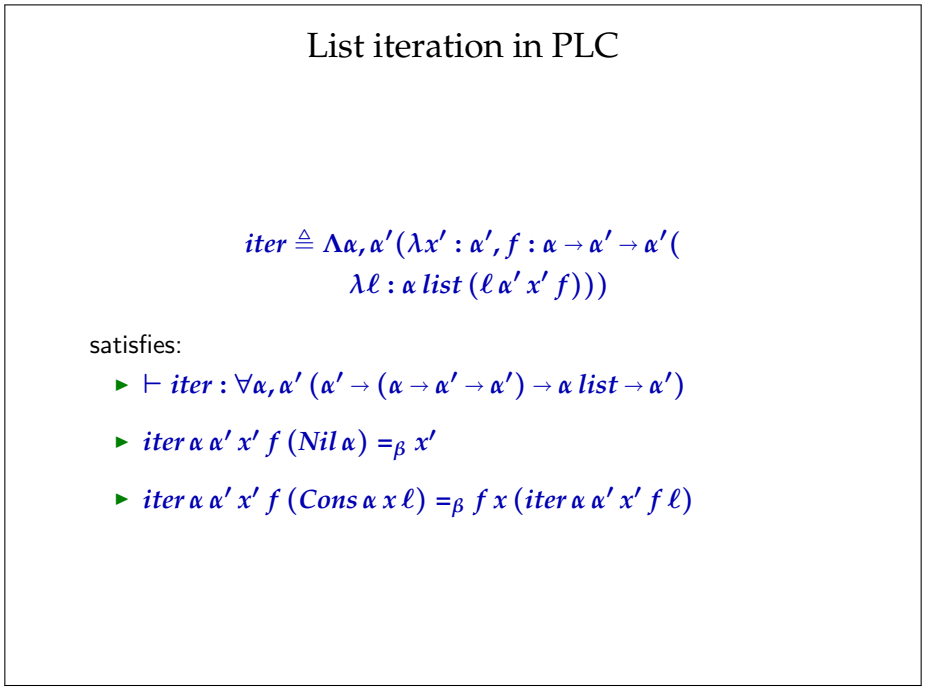
is shown on Slide 59. Undoubtedly it looks rather mysterious at first sight. The idea behind this representation has to do with the operation of *iteration over a list* shown on Slide 58. The existence of such functions $listIter\, x'\, f$ does in fact characterise the set A^* of finite lists over a set A uniquely up to bijection. We can take the operation

$$\lambda x' : \alpha', f : \alpha \to \alpha' \to \alpha'\ (listIter\, x'\, f\, \ell) \tag{9}$$

(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 59 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 59. Similarly for the constructor *Cons* for adding an element to the head of a list. It is not hard to prove the typings:

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

As shown on Slide 60, 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 (9) up to beta-conversion.



Slide 60

Remark 19. The syntax of ML expressions we used in Section 2 featured the usual `case`-expressions for lists. In PLC we might hope to define an expression *case* of type

$$\forall\alpha, \alpha''\ (\alpha'' \to (\alpha \to \alpha\ list \to \alpha'') \to \alpha\ list \to \alpha'')$$

such that

$$case\ \alpha\ \alpha''\ x''\ g\ (Nil\ \alpha) = x''$$
$$case\ \alpha\ \alpha''\ x''\ g\ (Cons\ \alpha\ x\ \ell) = g\ x\ \ell.$$

This is possible (but not too easy), by defining an operator for list primitive recursion. This is alluded to on page 92 of Girard (1989); product types are mentioned there because the definition of the primitive recursion operator can be done by a simultaneous iterative definition of the operator itself and an auxiliary function. We omit the details. However, it is important to note that the above equations will hold up to beta-conversion only for x and ℓ restricted to range over beta-normal forms. (Alternatively, the equations hold in full generality so long as '=' is taken to be some form of contextual equivalence.)

ML	PLC
`datatype` *null* = ;	$null \triangleq \forall\alpha\,(\alpha)$
`datatype` *unit* = *Unit*;	$unit \triangleq \forall\alpha\,(\alpha \to \alpha)$
$\alpha_1 * \alpha_2$	$\alpha_1 * \alpha_2 \triangleq \forall\alpha\,((\alpha_1 \to \alpha_2 \to \alpha) \to \alpha)$
`datatype` (α_1, α_2)*sum* = *Inl* `of` α_1 | *Inr* `of` α_2;	$(\alpha_1, \alpha_2)sum \triangleq \forall\alpha\,((\alpha_1 \to \alpha) \to (\alpha_2 \to \alpha) \to \alpha)$
`datatype` *nat* = *Zero* | *Succ* `of` *nat*;	$nat \triangleq \forall\alpha\,(\alpha \to (\alpha \to \alpha) \to \alpha)$
`datatype` *binTree* = *Leaf* | *Node* `of` *binTree* $*$ *binTree*;	$binTree \triangleq \forall\alpha\,(\alpha \to (\alpha \to \alpha \to \alpha) \to \alpha)$

Figure 5: Some more algebraic datatypes

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

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

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

4.5 Existential types

Recall the notion of *abstract data type*, for example as provided by Standard ML's notions of *signature* and *structure*, illustrated on Slide 61. The signature `QUEUE` classifies structures consisting of some type `'a queue` and some values providing the usual operations on queues of values of type `'a`. The structure `Queue` matches this signature using a concrete representation of `'a queue` (namely the type of pairs of `'a` lists) which is hidden outside the `structure` declaration. Mitchell and Plotkin (1988) observe that abstract data types can be classified by *existentially quantified* types; for example the signature `QUEUE` classifies structures for which there exists a type `'a queue` such that ....

Standard ML signatures and structures

```
signature QUEUE =
  sig
    type 'a queue
    exception Empty
    val empty : 'a queue
    val insert : 'a * 'a queue -> 'a queue
    val remove : 'a queue -> 'a * 'a queue
end

structure Queue =
  struct
    type 'a queue = 'a list * 'a list
    exception Empty
    val empty = (nil, nil)
    fun insert (f, (front,back)) = (f::front, back)
    fun remove (nil, nil) = raise Empty
      | remove (front, nil) = remove (nil, rev front)
      | remove (front, b::back) = (b, (front, back))
  end
```

Slide 61

PLC + existential types

Types

$t ::= \cdots \mid \exists\,\alpha\,(\tau)$

Expressions

$M ::= \cdots \mid \mathtt{pack}\,(\tau, M) : \exists\,\alpha\,(\tau) \mid$
$\mathtt{unpack}\,M : \exists\,\alpha\,(\tau)\,\mathtt{as}\,(\alpha, x)\,\mathtt{in}\,M : \tau$

Typing rules

$$(\exists\textbf{intro})\ \frac{\Gamma \vdash M : \tau[\tau'/\alpha]}{\Gamma \vdash (\mathtt{pack}\,(\tau', M) : \exists\,\alpha\,(\tau)) : \exists\,\alpha\,(\tau)}$$

$$(\exists\textbf{elim})\ \frac{\Gamma \vdash E : \exists\,\alpha\,(\tau) \qquad \Gamma, x : \tau \vdash M' : \tau'}{\Gamma \vdash (\mathtt{unpack}\,E : \exists\,\alpha\,(\tau)\,\mathtt{as}\,(\alpha, x)\,\mathtt{in}\,M' : \tau') : \tau'} \quad \text{if } \alpha \notin ftv(\Gamma, \tau')$$

Reduction

$$\mathtt{unpack}\,(\mathtt{pack}\,(\tau', M) : \exists\,\alpha\,(\tau))\,\mathtt{as}\,(\alpha, x)\,\mathtt{in}\,M' : \tau' \rightarrow M'[\tau'/\alpha, M/x]$$

Slide 62

Typing and reduction rules for an extension of PLC with existential types are given on Slide 62. I have included rather a lot of explicit type information in the syntax of `pack` and `unpack` expressions in order to keep type inference as simple as it is for pure PLC. Free occurrences of the type variable α in τ become bound in the type $\exists\,\alpha\,(\tau)$; free occurrences of the type variable α and the variable x in M' become bound in the expression $\mathtt{unpack}\,E : \exists\,\alpha\,(\tau)\,\mathtt{as}\,(\alpha, x)\,\mathtt{in}\,M' : \tau$; and α should not occur free in the type τ' occurring at the end of an `unpack`-expression – see the side condition on the typing rule ($\exists$elim), which is comparable to the side condition on the PLC typing rule (gen).

Existential types in PLC

$$\exists\,\alpha\,(\tau) \triangleq \forall\beta\,((\forall\alpha\,(\tau \to \beta)) \to \beta)$$

$$\texttt{pack}\,(\tau', M) : \exists\,\alpha\,(\tau) \triangleq \Lambda\beta\,(\lambda y : \forall\alpha\,(\tau \to \beta)\,(y\,\tau' M))$$

$$\texttt{unpack}\,E : \exists\,\alpha\,(\tau)\,\texttt{as}\,(\alpha, x)\,\texttt{in}\,M' : \tau' \triangleq E\,\tau'(\Lambda\alpha\,(\lambda x : \tau\,(M')))$$

(where $\beta \notin ftv(\alpha\,\tau\,\tau'\,M\,M')$)

These definitions satisfy the typing and reduction rules on the previous slide (exercise).

Slide 63

In fact this extension of PLC with existential types does not increase its expressive power, because it is possible to translate the extended type system back into pure PLC, as indicated on Slide 63.

5 Dependent Types

In the Mini-ML language, expressions (Slide 15) can *depend* upon variables, in the sense that they can contain occurrences of free variables; and Mini-ML type schemes can depend upon type variables. In PLC (Slide 46), expressions can depend upon both variables and type variables, but types can only depend upon type variables. Is it useful to consider type systems in which the types depend upon variables (and possibly upon type variables as well)? The answer is an emphatic yes. Such *dependently-typed* systems go back to the pioneering work of the logician Martin-Löf (1975) and mathematician de Bruijn (1970), and are still the subject of vigorous research by people interested in functional programming languages and in machine-checked formalisation of mathematics. The next subsection gives a small example to illustrate why it is useful to consider types that depend on variables ranging over the values of another type.

5.1 Dependent functions

It is helpful to recall some discrete mathematics. If A and B are sets, then the set $A \to B$ of mathematical functions from A to B is by definition the set of binary relations $F \subseteq A \times B$ that are single-valued

$$\forall a \in A, b \in B, b' \in B\ ((a,b) \in F \wedge (a,b') \in F \to b = b')$$

and total

$$\forall a \in A\ (\exists b \in B\ ((a,b) \in F)).$$

Suppose now instead of a single set B we are given a family of sets B_a indexed by elements $a \in A$. (In other words $\{(a, B_a) \mid a \in A\}$ is a mathematical function from A into a set of sets.) Defining

$$B \triangleq \bigcup_{a \in A} B_a = \{b \mid \exists a \in A\ (b \in B_a)\}$$

to be the union of the sets B_a as a ranges over all the elements of A, we can consider those mathematical functions $F \in A \to B$ with the property that F maps each $a \in A$ into the set B_a; see Slide 64.

Dependent Functions

Given a set A and a family of sets B_a indexed by the elements a of A, we get a set

$$\prod_{a \in A} B_a \triangleq \{F \in A \to \bigcup_{a \in A} B_a \mid \forall (a,b) \in F\ (b \in B_a)\}$$

of **dependent functions**. Each $F \in \prod_{a \in A} B_a$ is a single-valued and total relation that associates to each $a \in A$ an element in B_a (usually written $F\,a$).

For example if $A = \mathbb{N}$ and for each $n \in \mathbb{N}$, $B_n = \{0,1\}^n \to \{0,1\}$, then $\prod_{n \in \mathbb{N}} B_n$ consists of functions mapping each number n to an n-ary Boolean operation.

Slide 64

A tautology checker

```
fun taut x f = if x = 0 then f else
                 (taut(x − 1)(f true))
                 andalso (taut(x − 1)(f false))
```

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

$$\begin{cases} 0\ AryBoolOp & \triangleq bool \\ (n+1)\ AryBoolOp & \triangleq bool \to (n\ AryBoolOp) \end{cases}$$

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

Slide 65

Now consider programming, in Standard ML say, a function *taut* that for any number n takes in n-ary boolean operations (in 'curried' form)

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

and returns *true* if f is a tautology. In other words *taut f* has value *true* if f gives *true* when applied to all of its 2^n possible arguments, and otherwise *taut f* has value *false*. One might try to program *taut* in Standard ML as on Slide 65. This is algorithmically correct, but does not type-check in ML. Why? Intuitively, the type of *taut n* for each natural number $n = 0, 1, 2, \ldots$ is the type *n AryBoolOp* of 'n-ary curried boolean operations' defined (by induction on n) on Slide 65. Then *taut* really should be 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 66 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`).

In general a *dependent type* is a type expression containing a free variable for which expressions ranging over some type can be substituted For example, on Slide 66 the type expression *x AryBoolOp* contains a variable x for which we can substitute expressions e of Type `Nat` to get types *e AryBoolOp* (which are themselves expressions of the Agda type `Set`).

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 x) AryBoolOp = Bool -> x AryBoolOp

taut : (x : Nat) -> x AryBoolOp -> Bool
taut zero     f = f
taut (succ x) f = taut x (f true) and taut x (f false)
```

Slide 66

Some typing rules for dependent function types are given on Slide 67, using the (more common) notation $\Pi x : \tau\,(\tau')$ for what Agda writes as $(x : \tau) \to \tau'$. Compare them with the usual rules for non-dependent function types $\tau \to \tau'$ (Slide 18); note that those rules are the special case of the ones on Slide 67 where the type τ' has no free occurrences of the variable x.

Dependent function types $\Pi x : \tau\,(\tau')$

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

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

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

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

Slide 67

Conversion typing rule

Dependent type systems usually feature a rule of the form

$$\frac{\Gamma \vdash M : \tau}{\Gamma \vdash M : \tau'} \quad \text{if } \tau \approx \tau'$$

where $\tau \approx \tau'$ is some relation of **equality between types** (e.g. inductively defined in some way).

For example one would expect $(\mathbf{1}+\mathbf{1})\,\texttt{AryBoolOp} \approx \mathbf{2}\,\texttt{AryBoolOp}$.

For decidability of type-checking, one needs $\approx$ to be a decidable relation between type expressions.

Slide 68

For reasons that we will explore in Section 6, type theories with dependent types have been used extensively in computer systems for formalising mathematics, for proof construction, and for checking the correctness of proofs. In this respect Martin-Löf's *intuitionistic type theory* (which first popularised the notion of 'dependent type') has been highly influential; see Nordström 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). A related and increasingly popular example of such a system is Coq (`coq.inria.fr`), which is based on a dependent type theory called the *Calculus of Inductive Constructions*.

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 routinely in general-purpose 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: see Slide 68. For example, if we add to the Agda code in Slide 66 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
```

should also be terms of type

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

In a 'Turing-powerful' language, that is one whose programs can express all computable partial functions[9], 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.

[9] Agda is not by default Turing powerful, because by design only total functions are programmable in it – reflecting its intended use as a proof assistant.

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. Haskell (and later versions of OCaml) incorporate many ideas from dependently typed systems while maintaining static type-checking. For further reading on type systems with dependent-types I recommend Aspinall and Hofmann (2005).

5.2 Pure Type Systems

Pure Type Systems (Barendregt, 1992, Sect. 5.2) is a family of dependently-typed languages for expressing various kinds of polymorphic function abstraction. PLC is in this family and the general case of a Pure Type System is syntactically as simple as PLC. However, the family of Pure Type Systems contains other instances that go well beyond PLC in the kind of function abstraction they can express. Two examples are Girard's *System F-omega*, which Greg Morrisett in some lectures from 2005 calls "the workhorse of modern compilers" and the *Calculus of Constructions* of Coquand and Huet (1988), which forms an important part of the logical basis of the popular Coq proof assistant (`coq.inria.fr`). Barendregt analysed the forms of abstraction present in the Calculus of Constructions by introducing a cube of related typed λ-calculi; and Pure Type Systems were introduced by Berardi and Terlouw as a natural generalisation of Barendregt's λ-cube.

Pure Type Systems (PTS) – syntax

In a PTS type expressions and term expressions are lumped together into a single syntactic category of **pseudo-terms**:

t	::=	x	variable
	|	s	sort
	|	$\Pi x : t\,(t)$	dependent function type
	|	$\lambda x : t\,(t)$	function abstraction
	|	$t\,t$	function application

where x ranges over a countably infinite set **Var** of variables and s ranges over a disjoint set **Sort** of **sort symbols** – constants that denote various universes (= types whose elements denote types of various sorts) [**kind** is a commonly used synonym for *sort*]. $\lambda x : t\,(t')$ and $\Pi x : t\,(t')$ both bind free occurrences of x in the pseudo-term t'.

$t[t'/x]$ denotes result of capture-avoiding substitution of t' for all free occurrences of x in t.

$t \to t' \triangleq \Pi x : t\,(t')$ where $x \notin fv(t')$.

Slide 69

The syntax of Pure Type Systems (PTS) is given on Slide 69. There is a single collection of *pseudo-terms*, some of which denote types and some of which denote elements of types. Operator association, scoping, free and bound variables and substitution are like those for the λ-calculus and PLC:

Remark 20 (Operator association and scoping)**.** As usual, application associates to the left. We continue to delimit the scope of Π- and λ-binders with parentheses, but 'dot' notation is also

very common in the literature

$$\Pi x : t.\, t' \qquad \lambda x : t.\, t'$$

in which case the scope of the binder extends as far to the right of the dot as possible. For iterated bindings we will surpress the repeated Π or λ symbols and write just one followed by a list of typed variables.

Remark 21 (Free and bound variables; substitution)**.** Any free occurrences of x in a pseudo-term t' become bound in $\lambda x : t\,(t')$ and $\Pi x : t\,(t')$. Thus the finite set $fv(t)$ of free variables of a pseudo-term t is given by:

$$\begin{aligned} fv(x) &= \{x\} \\ fv(s) &= \{\,\} \\ fv(\Pi x : t\,(t')) &= fv(t) \cup (fv(t') - \{x\}) \\ fv(\lambda x : t\,(t')) &= fv(t) \cup (fv(t') - \{x\}) \\ fv(t\,t') &= fv(t) \cup fv(t') \end{aligned}$$

As usual, *we implicitly identify pseudo-terms up to alpha-conversion of bound variables*. $t[t'/x]$ denotes the capture-avoiding substitution of t' for all free occurrences of x in the pseudo-term t.

Pure Type Systems – beta-conversion

- **beta-reduction** of pseudo-terms: $t \longrightarrow t'$ means t' can be obtained from t (up to alpha-conversion, of course) by replacing a subexpression which is a **redex** by its corresponding **reduct**. There is only one form of redex-reduct pair:

$$(\lambda x : t\,(t_1))\,t_2 \longrightarrow t_1[t_2/x]$$

- As usual, $\longrightarrow^*$ denotes the reflexive-transitive closure of $\longrightarrow$.
- **beta-conversion** of pseudo-terms: $=_\beta$ is the reflexive-symmetric-transitive closure of $\longrightarrow$ (i.e. the smallest equivalence relation containing $\longrightarrow$).

Slide 70

As remarked on Slide 68 for dependently-typed systems in general, typing in PTS involves a notion of equality between pseudo-expressions: PTS uses the usual notion of *beta-conversion* from the λ-calculus (Slide 70). The form of PTS typing judgements is shown on Slide 72 and the rues for generating the *derivable* judgements on Slide 73; the notation $dom(\Gamma)$ used there refers to the *domain* of a context Γ:

$$\begin{aligned} dom(\diamond) &= \{\,\} \\ dom(\Gamma, x : t) &= dom(\Gamma) \cup \{x\}. \end{aligned}$$

The rules on Slide 73 are parameterized by a *specification*, defined on slide 71. We write $\lambda\mathbf{S}$ for the PTS with specification $\mathbf{S}$.

Pure Type Systems – specifications

The typing rules for a particular PTS are parameterised by a **specification** $\mathbf{S} = (\mathcal{S}, \mathcal{A}, \mathcal{R})$ where:

- $\mathcal{S} \subseteq \mathbf{Sort}$
  Elements $s \in \mathcal{S}$ denote the different universes of types in this PTS.
- $\mathcal{A} \subseteq \mathbf{Sort} \times \mathbf{Sort}$
  Elements $(s_1, s_2) \in \mathcal{A}$ are called **axioms**. They determine the typing relation between universes in this PTS.
- $\mathcal{R} \subseteq \mathbf{Sort} \times \mathbf{Sort} \times \mathbf{Sort}$
  Elements $(s_1, s_2, s_3) \in \mathcal{R}$ are called rules. They determine which kinds of dependent function can be formed and in which universes they live.

The PTS with specification $\mathbf{S}$ will be denoted $\boxed{\lambda \mathbf{S}}$.

Slide 71

Pure Type Systems – typing judgements

take the form

$$\boxed{\Gamma \vdash t : t'}$$

where t, t' are pseudo-terms and Γ is a **context**, a form of typing environment given by the grammar

$$\Gamma ::= \diamond \mid \Gamma, x : t$$

(Thus contexts are finite ordered lists of (variable,pseudo-term)-pairs, with the empty list denoted $\diamond$, the head of the list on the right and list-cons denoted by $\_,\_$. Unlike previous type systems in this course, **the order in which typing declarations $x : t$ occur in a context is important**.)
A typing judgement is **derivable** if it is in the set inductively generated by the rules on the next slide, which are parameterised by a given specification $\mathbf{S} = (\mathcal{S}, \mathcal{A}, \mathcal{R})$.

Slide 72

Pure Type Systems – typing rules

$$\textbf{(axiom)}\ \frac{}{\diamond \vdash s_1 : s_2}\ \text{if } (s_1, s_2) \in \mathcal{A}$$

$$\textbf{(start)}\ \frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}\ \text{if } x \notin dom(\Gamma)$$

$$\textbf{(weaken)}\ \frac{\Gamma \vdash M : A \quad \Gamma \vdash B : s}{\Gamma, x : B \vdash M : A}\ \text{if } x \notin dom(\Gamma)$$

$$\textbf{(conv)}\ \frac{\Gamma \vdash M : A \quad \Gamma \vdash B : s}{\Gamma \vdash M : B}\ \text{if } A =_\beta B$$

$$\textbf{(prod)}\ \frac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash \Pi x : A\,(B) : s_3}\ \text{if } (s_1, s_2, s_3) \in \mathcal{R}$$

$$\textbf{(abs)}\ \frac{\Gamma, x : A \vdash M : B \quad \Gamma \vdash \Pi x : A\,(B) : s}{\Gamma \vdash \lambda x : A\,(M) : \Pi x : A\,(B)}$$

$$\textbf{(app)}\ \frac{\Gamma \vdash M : \Pi x : A\,(B) \quad \Gamma \vdash N : A}{\Gamma \vdash M\,N : B[N/x]}$$

(A, B, M, N range over pseudoterms, s, s_1, s_2, s_3 over sort symbols)

Slide 73

Example PTS typing derivations

$$\dfrac{\textbf{(axiom)}\ \dfrac{}{\diamond \vdash * : \Box} \qquad \textbf{(weaken)}\ \dfrac{\textbf{(axiom)}\ \dfrac{}{\diamond \vdash * : \Box}}{\diamond, x : * \vdash * : \Box}}{\textbf{(axiom)}\ \textbf{(prod)}\ \diamond \vdash * \to * : \Box}$$

$$\dfrac{\textbf{(axiom)}\ \dfrac{}{\diamond \vdash * : \Box}}{\textbf{(start)}\ \diamond, x : * \vdash x : *} \qquad \dfrac{\vdots}{\diamond \vdash * \to * : \Box}$$
$$\textbf{(abs)}\ \diamond \vdash \lambda x : *\,(x) : * \to *$$

Here we assume that the PTS specification $\mathbf{S} = (\mathcal{S}, \mathcal{A}, \mathcal{R})$ has $* \in \mathcal{S}$, $\Box \in \mathcal{S}$, $(*, \Box) \in \mathcal{A}$ and $(\Box, \Box, \Box) \in \mathcal{R}$.
(Recall that $* \to * \triangleq \Pi x : *\,(*)$.)

Slide 74

Slide 74 illustrates how PTS typing derivations begin, using (axiom), (start) and (weaken). Note that the rules (conv) and (prod) feature extra hypotheses (compared with what one might expect, for example from Slide 68) which ensure that every pseudo-term that appears in a position in a derivable judgement where a type is expected has a sort, or is already a sort. This is the *Correctness of types* property on Slide 75. Note that the rules on Slide 73 are not all syntax-directed; in particular, in a syntax-directed search for a type for a given pseudo-term, it is not clear when rule (conv) should be used. For this reason the type-checking and typeability problems (Slide 6) for a PTS can be difficult: for some PTS specifications these problems are

decidable (and good algorithms are known – see Barthe (1999) for example), and for some they are not. Slide 76 states two results of this kind.

Properties of Pure Type Systems in general

- **Correctness of types.** If $\Gamma \vdash M : A$, then either $A \in \mathcal{S}$, or $\Gamma \vdash A : s$ for some $s \in \mathcal{S}$.
- **Church-Rosser Property** (aka **confluence**). $t =_\beta t'$ iff $\exists u\,(t \to^* u \ \wedge\ t' \to^* u)$
- **Subject Reduction.** If $\Gamma \vdash M : A$ and $M \to M'$, then $\Gamma \vdash M' : A$.
- **Uniqueness of Types.** A PTS specification $\mathbf{S} = (\mathcal{S}, \mathcal{A}, \mathcal{R})$ is said to be **functional** if both $\mathcal{A}$ and $\mathcal{R}_s \triangleq \{(s_2, s_3) \mid (s, s_2, s_3) \in \mathcal{R}\}$ for each $s \in \mathcal{S}$, are single-valued binary relations.
  In this case $\lambda\mathbf{S}$ satisfies: if $\Gamma \vdash M : A$ and $\Gamma \vdash M : B$, then $A =_\beta B$.

Slide 75

Type-checking for a PTS, $\lambda\mathbf{S}$

Definition. A pseudo-term t is **legal** for a PTS specification $\mathbf{S} = (\mathcal{S}, \mathcal{A}, \mathcal{R})$ if either $t \in \mathcal{S}$ or $\Gamma \vdash t : t'$ is derivable in $\lambda\mathbf{S}$ for some Γ and t'.

Recall the **type-checking** and **typeability** problems for a type system.

Fact(van Bentham Jutting): these problems for $\lambda\mathbf{S}$ are decidable if $\mathbf{S}$ is finite and $\lambda\mathbf{S}$ is **normalizing**, meaning that for every legal pseudo-term there is some finite chain of beta-reductions leading to a beta-normal form.

Fact (Meyer): the problems are undecidable for the PTS $\lambda*$ with specification $\mathcal{S} = \{*\}$, $\mathcal{A} = \{(*, *)\}$ and $\mathcal{R} = \{(*, *, *)\}$.

Slide 76

Remark 22 (Equality judgements). Pure Type Systems are a good first example of formally defined type systems featuring dependent types, because they are syntactically quite simple. In particular, they have only one form of judgement (Slide 72) and a notion of equality ($=_\beta$) which is defined independently of typing. In contrast, many type systems for dependent types

feature both a typing judgement $\Gamma \vdash M : A$ and an *equality judgement*

$$\Gamma \vdash M \equiv M' : A$$

defined mutually inductively. For example, if one wanted equality to satisfy η-conversion, it makes sense to equate M with $\lambda x : A\,(M\,x)$ only if M is already known to have a (dependent) function type:

$$\frac{\Gamma \vdash M : \Pi x : A\,(B)}{\Gamma \vdash M \equiv \lambda x : A\,(M\,x) : \Pi x : A\,(B)} \text{ if } x \notin fv(M)$$

For example, see the type system λLF in the chapter by Aspinall and Hofmann (2005).

In the rest of this section and the next we give some examples of Pure Type Systems.

Example 23 (**PLC as a Pure Type System**). Consider the PTS signature on Slide 77. The sort $\Box$ plays a trivial, but important role: the only pseudo-term t satisfying $\diamond \vdash t : \Box$ is $t = *$. The sort $*$ classifies types in $\lambda 2$. Because we have $\diamond \vdash * : \Box$ we can use the rules (start) and (weaken) to introduce type variable $\Gamma, \alpha : * \vdash \alpha : *$ (assuming $\alpha \in \mathrm{Var}$). Then the typing rule (prod) for rule $(\Box, *, *) \in \mathcal{R}_2$ allows the formation of types $\Gamma \vdash \Pi\alpha : *\,(\tau) : *$, given that $\Gamma, \alpha : * \vdash \tau : *$ is derivable – these behave like PLC's $\forall\alpha\,(\tau)$ types. The other rule in the specification, $(*, *, *) \in \mathcal{R}_2$, allows the formation of types $\Gamma \vdash \Pi x : \tau\,(\tau') : *$, given that $\Gamma \vdash \tau : *$ and $\Gamma, x : \tau \vdash \tau' : *$ is derivable. In fact legal pseudo-terms in $\lambda 2$ of type $*$ can only involve free type variables, but not free variables of some given type $\tau : *$; so the $\lambda 2$ type $\Pi x : \tau\,(\tau')$ is in fact the simple function type $\tau \to \tau'$, rather than a properly dependent function type.

As the above remarks suggest, there is a strong relationship between $\lambda 2$ and the Polymorphic Lambda Calculus (Section 4). For example the polymorphic identity function in PLC

$$\{\,\} \vdash \Lambda\alpha\,(\lambda x : \alpha\,(x)) \vdash \forall\alpha\,(\alpha \to \alpha)$$

corresponds in the PTS $\lambda 2$ to

$$\diamond \vdash \lambda\alpha : *\,(\lambda x : \alpha\,(x)) : \Pi\alpha : *\,(\alpha \to \alpha) \tag{10}$$

(I am making the simplifying assumption that the set of variables from which $\lambda 2$ pseudo-terms are formed consists of the disjoint union of the sets of type variables and variables used in PLC type and term expressions.)

In fact the PTS $\lambda 2$ and PLC are essentially the same type system. Using the translation of PLC types and terms into $\lambda 2$ pseudo-terms given on Slide 77 one can show the properties on Slide 78. The proof of these properties necessarily involves considering PLC typing judgements with non-empty typing environments and the mis-match between how I defined those and how PTS contexts are defined makes statements and proofs messy: it would be better to reformulate PLC using typing environments that are lists (rather than finite functions) and that record the use of free type variables in the environment (rather than leaving that information implicit as I did in Section 4).

PLC versus the Pure Type System $\lambda 2$

PTS signature:

$$2 \triangleq (\mathcal{S}_2, \mathcal{A}_2, \mathcal{R}_2) \text{ where } \begin{cases} \mathcal{S}_2 & \triangleq & \{*, \Box\} \\ \mathcal{A}_2 & \triangleq & \{(*, \Box)\} \\ \mathcal{R}_2 & \triangleq & \{(*,*,*), (\Box,*,*)\} \end{cases}$$

Translation of PLC types and terms to $\lambda 2$ pseudo-terms:

$$\begin{aligned} [\![\alpha]\!] &= \alpha \\ [\![\tau \to \tau']\!] &= \Pi x : [\![\tau]\!]\,([\![\tau']\!]) \\ [\![\forall \alpha\,(\tau)]\!] &= \Pi \alpha : *\,([\![\tau']\!]) \\ [\![x]\!] &= x \\ [\![\lambda x : \tau\,(M)]\!] &= \lambda x : [\![\tau]\!]\,([\![M]\!]) \\ [\![M\,M']\!] &= [\![M]\!]\,[\![M']\!] \\ [\![\Lambda \alpha\,(M)]\!] &= \lambda \alpha : *\,([\![M]\!]) \\ [\![M\,\tau]\!] &= [\![M]\!]\,[\![\tau]\!] \end{aligned}$$

Slide 77

Properties of the translation from PLC to $\lambda 2$

- If $\{\,\} \vdash M : \tau$ is derivable in PLC, then $\diamond \vdash [\![\tau]\!] : *$ and $\diamond \vdash [\![M]\!] : [\![\tau]\!]$ are derivable in $\lambda 2$
- In $\lambda 2$, if $\diamond \vdash t : \Box$, then $t = *$; if $\diamond \vdash t : *$, then $t = [\![\tau]\!]$ for some closed PLC type τ; and if $\diamond \vdash t : t'$ then $t = [\![M]\!]$ and $t' = [\![\tau]\!]$ for PLC expressions satisfying $\{\,\} \vdash M : \tau$.
- Under the translation, the reduction behaviour of PLC terms is preserved and reflected by beta-reduction in $\lambda 2$. (Note in particular that PLC types are translated to pseudo-terms in beta-normal form.)

Slide 78

5.3 System F_ω

I mentioned at the start of Section 4 that PLC was invented by the logician Jean-Yves Girard, under the name System F, as part of his seminal work on the proof-theory of second-order arithmetic. In fact, as the name suggests (*Interprétation fonctionelle et élimination des coupures dans l'arithmetique d'ordre supérieur*), the work of Girard (1972) treats the proof theory of not just second-order, but also higher-order arithmetic. (I discuss what Type Systems have to do with proofs and their theory in Section 6.) To do so he introduced a higher-order version

of Polymorphic Lambda Calculus, called *System* F_ω. From a programming language (rather than a logic) point of view, System F_ω allows us to write functions that are parametrically polymorphic not only in types, but also in type constructions (functions from types to types) and higher-order versions of such constructions. F_ω can be defined as an instance of the notion of Pure Type System: see Slide 79.

System $\mathbf{F}_\omega$ as a Pure Type System: $\boldsymbol{\lambda\omega}$

PTS specification $\omega = (\mathcal{S}_\omega, \mathcal{A}_\omega, \mathcal{R}_\omega)$:

$$\mathcal{S}_\omega \triangleq \{*, \Box\}$$
$$\mathcal{A} \triangleq \{(*, \Box)\}$$
$$\mathcal{R} \triangleq \{(*,*,*), (\Box,*,*), (\Box,\Box,\Box)\}$$

As in $\boldsymbol{\lambda 2}$, sort $*$ is a universe of types; but in $\boldsymbol{\lambda\omega}$, the rule (**prod**) for $(\Box,\Box,\Box)$ means that $\diamond \vdash t : \Box$ holds for all the 'simple types' over the ground type $*$ – the ts generated by the grammar $t ::= * \mid t \to t$

Hence rule (**prod**) for $(\Box,*,*)$ now gives many more legal pseudo-terms of type $*$ in $\boldsymbol{\lambda\omega}$ compared with $\boldsymbol{\lambda 2}$ (PLC), such as

$$\diamond \vdash (\Pi T : * \to * \, (\Pi\alpha : * \, (\alpha \to T\,\alpha))) : *$$
$$\diamond \vdash (\Pi T : * \to * \, (\Pi\alpha,\beta : * \, ((\alpha \to T\,\beta) \to T\,\alpha \to T\,\beta))) : *$$

Slide 79

F_ω underlies work on type-directed compilation of strongly-typed functional programming languages to lower-level languages (Greg Morrisett in some lectures from 2005 calls F_ω the "the workhorse of modern compilers"). Later versions of Haskell and OCaml give the programmer access to such higher-order polymorphism.

As an example of the kind of type abstractions that F_ω provides, consider the concept of a *monad* from the mathematical theory of categories (MacLane, 1971, Chapter VI), widely used in typed functional programming to tame 'impure' forms of computation, such as exceptions, I/O, or foreign language calls; see Wadler (1992) and Peyton Jones (2001). In Standard ML, a monad is specified by the data on Slide 80: one thinks of values of type $\tau(\alpha)$ as denoting computations of some kind that produce values of type α. For example in the global state monad, a computation in $\sigma \to (\alpha * \sigma)$ takes in an initial state represented as a value of type σ and returns, if anything, a pair consisting of a value of type α and a final state. Not only are there many different monads, but also there are lots of useful ways of transforming and combining existing monads to form new ones. To express such *monad transformers* we need a way of abstracting over the data specifying a monad – in particular of writing functions with arguments $T : * \to *$, where $*$ is a sort of types. F_ω allows us to do this; see Slide 81.

Monads in ML

A monad in ML is given by type $\tau(\alpha)$ with a free type variable α together with expressions

$$\begin{aligned} &\mathit{unit} : \alpha \to \tau(\alpha) \\ &\mathit{lift} : (\alpha \to \tau(\beta)) \to \tau(\alpha) \to \tau(\beta) \end{aligned}$$

(writing $\tau(\beta)$ for the result of replacing α by β in τ) satisfying some equational properties [omitted].
E.g.

- list monad $\tau(\alpha) = \alpha\ \mathit{list}$
- global state monad $\tau(\alpha) = \sigma \to (\alpha * \sigma)$ (for some type σ of states)
- simple exception monad $\tau(\alpha) = (\alpha, \varepsilon)\mathit{sum}$ (for some type ε of exception names)

[definitions of unit and lift in each case omitted]

Slide 80

Examples of $\lambda\omega$ type constructions

- Product types (cf. the PLC representation of product types):

$$\begin{aligned} &\mathtt{P} \triangleq \lambda\alpha, \beta : * \,(\Pi\gamma : * \,((\alpha \to \beta \to \gamma) \to \gamma)) \\ &\diamond \vdash \mathtt{P} : * \to * \to * \end{aligned}$$

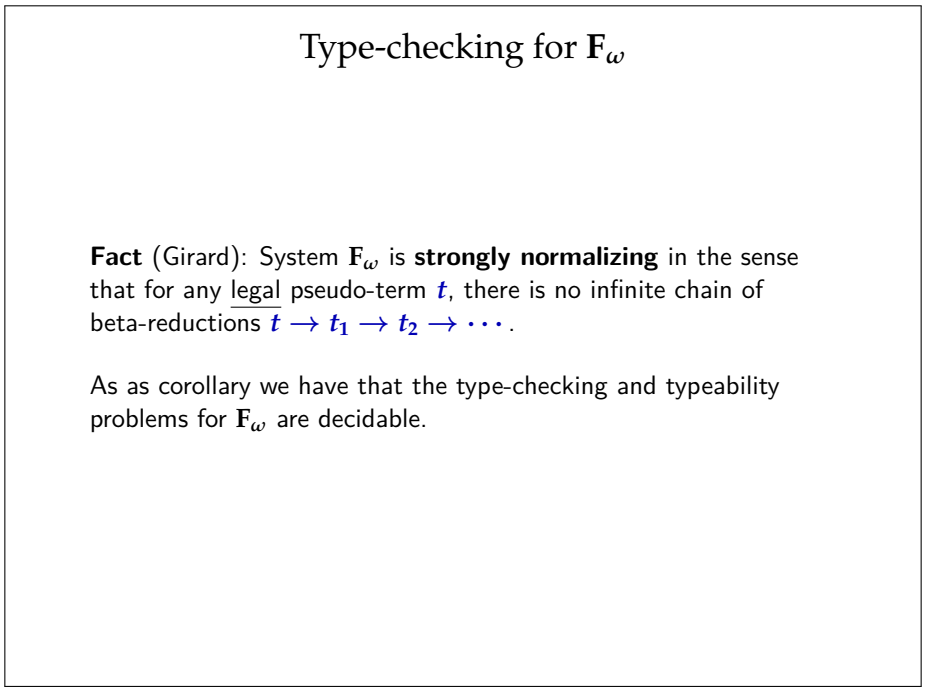
- Monad transformer for state (using a type $\diamond \vdash S : *$ for states):

$$\begin{aligned} &\mathtt{M} \triangleq \lambda T : * \to * \,(\lambda\alpha : * \,(S \to T(\mathtt{P}\,\alpha\,S))) \\ &\diamond \vdash \mathtt{M} : (* \to *) \to * \to * \end{aligned}$$

- Existential types (cf. the PLC representation of existential types):

$$\begin{aligned} &\exists \triangleq \lambda T : * \to * \,(\Pi\beta : * \,((\Pi\alpha : * \,(T\,\alpha \to \beta)) \to \beta)) \\ &\diamond \vdash \exists : (* \to *) \to * \end{aligned}$$

Slide 81

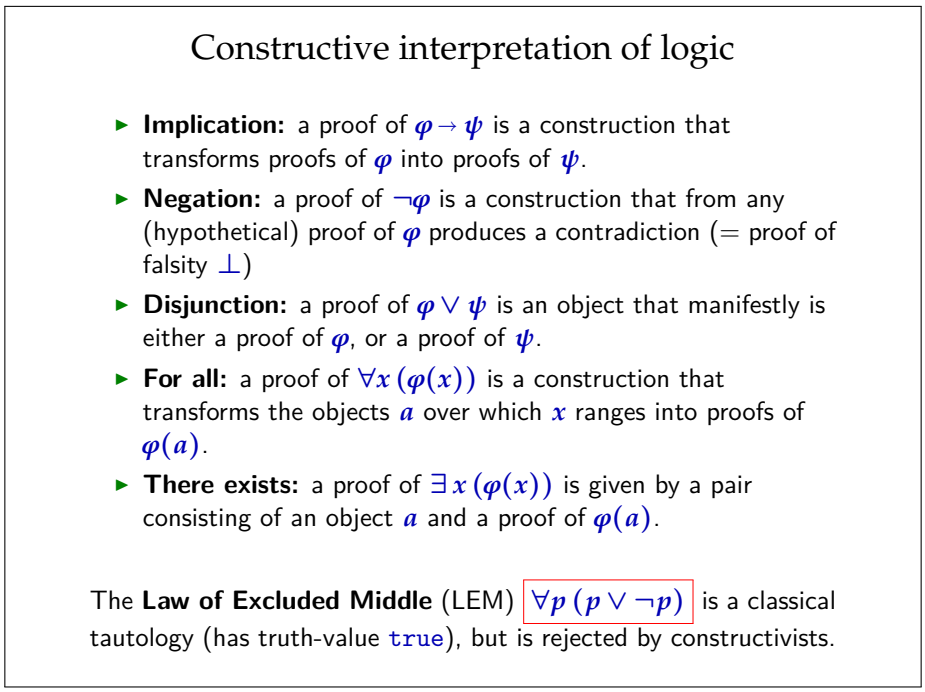


Slide 82

One of the important results of Girard (1972) is that Systems F and F_ω are *strongly normalizing*; see Slide 82. His proof introduced the 'candidates of reducibility' method, which has been successfully generalized to other dependently-typed lambda calculi. Girard (1989, Chapter 14) gives the method applied to PLC.

6 Propositions as Types

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 proofs in certain systems of logic and the terms in certain typed lambda calculi. This was 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*. If is also known as *propositions-as-types* for reasons that I explain in Sect. 6.2. Before that, I make an digression into *intuitionistic*, or *constructive* logic, since it is that kind of logic for which the Curry-Howard correspondence first arose.



Slide 83

6.1 Intuitionistic logics

The *constructivist* approach to logic is outlined on Slide 83. Constructivists reject the Law of Excluded Middle $\forall p\,(p \vee \neg p)$ (as well as some other logical principles that are classically valid) and consequently some classically valid proofs are not constructive: see Slide 84. The proof of the theorem on that slide does not give us an example of irrational numbers a and b for which b^a is rational. Slide 85 gives a constructively valid proof of the same theorem (due to Thierry Coquand, I think). It is certainly preferable to the previous proof, because we get more information. This is typical: even if you accept the validity of LEM (and other non-constructive principles of classical mathematics), constructive proofs can be more interesting and useful.

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 $b^a = (\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = \sqrt{2}^{\sqrt{2}.\sqrt{2}} = \sqrt{2}^2 = 2$.

QED

Slide 84

Example of a constructive proof

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

Proof. $\sqrt{2}$ is irrational by a well-known constructive proof attributed to Euclid.

$2\log_2 3$ is irrational, by an easy constructive proof (exercise).

So we can take $a = 2\log_2 3$ and $b = \sqrt{2}$, for which we have that $b^a = (\sqrt{2})^{2\log_2 3} = (\sqrt{2^2})^{\log_2 3} = 2^{\log_2 3} = 3$ is rational.

QED

Slide 85

For the Dutch mathematician Brouwer (1881–1966), constructivism meant rejecting truth in some Platonic objective reality in favour of subjective mental constructions – human intuition. For that reason his approach was called *intuitionistic* mathematics and the logical systems that his follower Heyting (1898–1980) and others devised to formalize Brouwer's ideas are called *intuitionistic* logics. An example of an intuitionistic logic is given on Slides 86 and 87 – the *second-order intuitionistic propositional calculus* (2IPC). The rules for generating proofs in 2IPC involving its two constructs $\to$ and $\forall$ come in pairs: rules for *introducing* the logical construct

$(\to\mathrm{I})/(\forall\mathrm{I})$, and rules for *eliminating* them $(\to\mathrm{E})/(\forall\mathrm{E})$. This way of organising proof rules is called *Natural Deduction* and is due to the logician Gentzen (1909–1945).

Second-order intuitionistic propositional calculus (2IPC)

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

2IPC sequents: $\Phi \vdash \phi$ where Φ is a finite multiset (= unordered list) 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:

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

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

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

Slide 86

A 2IPC proof

Writing $p \wedge q$ as an abbreviation for $\forall r\,((p \to q \to r) \to r)$, the sequent

$$\{\} \vdash \forall p\,(\forall q\,((p \wedge q) \to p))$$

is provable in 2IPC:

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

Slide 87

6.2 Curry-Howard correspondence

Computer scientists naturally take the intuitionistic notion of *construction* as having something to do with the notion of *algorithm* from computation theory. They are interested in construc-

tive proof and intuitionistic logics because of their connections with computation. The Curry-Howard correspondence, outlined on Slide 88 helps to formalize the connection. To see how the Curry-Howard correspondence works, we will look at a specific instance, namely the correspondence between 2IPC (Slide 86) and the PLC type system of Section 4.

Curry-Howard correspondence

Logic	$\leftrightarrow$	Type system
propositions ϕ	$\leftrightarrow$	types τ
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
	E.g.	
2IPC	$\leftrightarrow$	PLC

Slide 88

Mapping 2IPC proofs to PLC expressions

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

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

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

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

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

Slide 89

The proof of the 2IPC sequent

$$\{\} \vdash \forall p\,(\forall q\,((p \wedge q) \to p))$$

given before is transformed by the mapping of 2IPC proofs to PLC expressions to

$$\{\} \vdash \Lambda p, q\,(\lambda z : p \wedge q\,(z\,p\,(\lambda x : p, y : q\,(x)))) : \forall p\,(\forall q\,((p \wedge q) \to p))$$

with typing derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{z : p \wedge q, x : p, y : q\} \vdash x : p}\,(\mathbf{id})}{\{z : p \wedge q, x : p\} \vdash \lambda y : q\,(x) : q \to p}\,(\mathbf{fn})}{\{z : p \wedge q\} \vdash \lambda x : p, y : q\,(x) : p \to q \to p}\,(\mathbf{fn}) \quad \dfrac{\dfrac{}{\{z : p \wedge q\} \vdash z : \forall r\,((p \to q \to r) \to r)}\,(\mathbf{id})}{\{z : p \wedge q\} \vdash z\,p : (p \to q \to p) \to p}\,(\mathbf{spec})}{\{z : p \wedge q\} \vdash z\,p\,(\lambda x : p, y : q\,(x)) : p}\,(\mathbf{app})}{\{\} \vdash \lambda z : p \wedge q\,(z\,p\,(\lambda x : p, y : q\,(x))) : (p \wedge q) \to p}\,(\mathbf{fn})$$
$$\dfrac{\{\} \vdash \Lambda q\,(\lambda z : p \wedge q\,(z\,p\,(\lambda x : p, y : q\,(x)))) : \forall q\,((p \wedge q) \to p)}{\{\} \vdash \Lambda p, q\,(\lambda z : p \wedge q\,(z\,p\,(\lambda x : p, y : q\,(x)))) : \forall p, q\,((p \wedge q) \to p)}\,(\mathbf{gen})\,(\mathbf{gen})$$

Slide 90

Logical operations definable in 2IPC

- **Truth** $\top \triangleq \forall p\,(p \to p)$
- **Falsity** $\bot \triangleq \forall p\,(p)$
- **Conjunction** $\phi \wedge \psi \triangleq \forall p\,((\phi \to \psi \to p) \to p)$ (where $p \notin fv(\phi, \psi)$)
- **Disjunction** $\phi \vee \psi \triangleq \forall p\,((\phi \to p) \to (\psi \to p) \to p)$ (where $p \notin fv(\phi, \psi)$)
- **Negation** $\neg\phi \triangleq \phi \to \bot$
- **Bi-implication** $\phi \leftrightarrow \psi \triangleq (\phi \to \psi) \wedge (\psi \to \phi)$
- **Existential quantification** $\exists p\,(\phi) \triangleq \forall q\,(\forall p\,(\phi \to q) \to q)$ (where $q \notin fv(\phi, p)$)

LEM $\forall p\,(p \vee \neg p) = \forall p, q\,((p \to q) \to ((p \to \forall r\,(r)) \to q) \to q)$

Fact: $\{\} \vdash M : \forall p\,(p \vee \neg p)$ is not provable in PLC for any expression M.

Slide 91

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 transformations on Slide 89. (On that slide we abbreviate $\{x_1 : \phi_1, \ldots, x_n : \phi_n\}$ as $\overline{x} : \Phi$.) This is illustrated on Slide 90. The example on that slide uses the fact that the logical operation of conjunction can be defined in 2IPC. Slide 91 gives some other logical operators that are definable in 2IPC. Compare it with what we saw

in Sections 4.4 and 4.5: the richness of PLC for expressing datatypes is mirrored under the Curry-Howard correspondence by the richness of 2IPC for expressing logical connectives and quantifiers. As one might expect from its name, in 2IPC these connectives and quantifiers behave constructively rather than classically. For example, the Law of Excluded Middle is not provable in 2IPC (Slide 91). This fact can be proved using the technique developed in Tripos question 2000.9.13, which is part of the very interesting topic of the *model theory* (denotational semantics) of PLC – a topic beyond the scope of these lectures.

The correspondence between PLC types/expressions and 2IPC propositions/proofs allows one to transfer the PLC notion of computation (beta-reduction) to a notion of computation on 2IPC proofs, namely *proof simplification*. Slide 92 shows how the PLC beta-reduction

$$(\lambda x : \phi\,(M))\,N \to M[N/x]$$

corresponds to a simplification of 2IPC proofs in which a use of the proof rule $(\to\mathrm{I})$ immediately followed by a use of $(\to\mathrm{E})$ is eliminated. Similarly, the other form of PLC beta-reduction

$$(\Lambda p\,(M))\,\phi \to M[\phi/p]$$

corresponds to a proof simplification of the form

$$\begin{array}{l}(\forall\mathrm{I})\\(\forall\mathrm{E})\end{array}\dfrac{\dfrac{\vdots}{\Phi \vdash \phi}}{\dfrac{\Phi \vdash \forall p\,(\phi)}{\Phi \vdash \phi[\psi/p]}}\; p \notin fv(\Phi) \;\mapsto\; (\mathrm{spec})\;\dfrac{\dfrac{\vdots}{\Phi \vdash \phi}}{\Phi \vdash \phi[\psi/p]}\; p \notin fv(\Phi)$$

where the rule (spec) is *admissible* in the sense described on Slide 92: one can show that if $\Phi \vdash \phi$ is provable in 2IPC and $p \notin fv(\Phi)$,then $\Phi \vdash \phi[\psi/p]$ is also provable. Finally, note that the Strong Normalization property for PLC (Slide 55) implies that if we repeatedly simplify a 2IPC proof we eventually reach a proof in normal form containing no $(\to\mathrm{I})$–$(\to\mathrm{E})$ or $(\forall\mathrm{I})$–$(\forall\mathrm{E})$ pairs.

Proof simplification ↔ Expression reduction

$$\begin{array}{l}(\to\mathrm{I})\\(\to\mathrm{E})\end{array}\dfrac{\dfrac{\vdots}{\Phi,\phi \vdash \psi}}{\dfrac{\Phi \vdash \phi \to \psi}{\Phi \vdash \psi}}\quad\dfrac{\vdots}{\Phi \vdash \phi} \;\mapsto\; \dfrac{\dfrac{\dfrac{\vdots}{\overline{x} : \Phi, x : \phi \vdash M : \psi}}{\overline{x} : \Phi \vdash \lambda x : \phi\,(M) : \phi \to \psi}\quad\dfrac{\vdots}{\overline{x} : \Phi \vdash N : \phi}}{\overline{x} : \Phi \vdash (\lambda x : \phi\,(M))\,N : \psi}$$

simplify proof ↓ ↓ beta-reduce expression

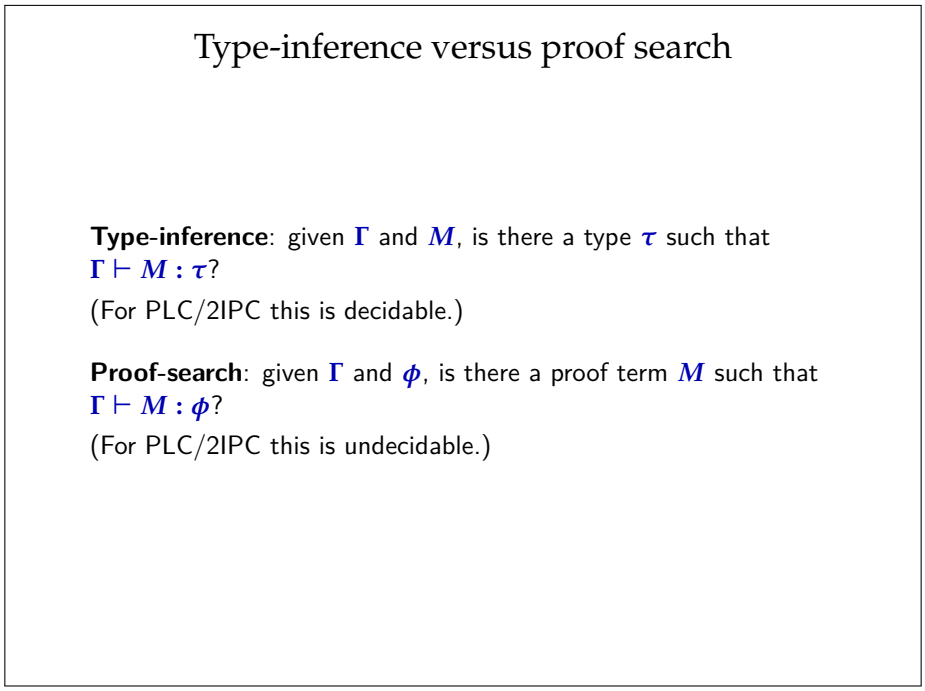
$$(\mathbf{cut})\;\dfrac{\dfrac{\vdots}{\Phi,\phi \vdash \psi}}{\Phi \vdash \psi}\quad\dfrac{\vdots}{\Phi \vdash \phi} \;\leftarrow\; \dfrac{\dfrac{\vdots}{\overline{x} : \Phi, x : \phi \vdash M : \psi}}{\overline{x} : \Phi \vdash M[N/x] : \psi}\quad\dfrac{\vdots}{\overline{x} : \Phi \vdash N : \phi}\;(\mathbf{subst})$$

The rule **(subst)** for PLC is **admissible**: if its hypotheses are valid PLC typing judgements, then so is its conclusion.

Hence, the rule **(cut)** is admissible for 2IPC.

Slide 92

The Curry-Howard correspondence gives us a different perspective on the typing judgement $\Gamma \vdash M : \sigma$, outlined on Slide 93. The correspondence cuts both ways: in one direction it has proved very helpful to use typed lambda terms as notations for proofs and their type systems in proof-search mode as part of interactive theorem-proving systems such as Coq (`coq.inria.fr`) and Agda (`wiki.portal.chalmers.se/agda/agda.php`). In the other direction, the Curry-Howard correspondence has helped to suggest new type systems for programming and specification languages, based on various kinds of logic. Two examples of this 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)) and language-based security (see Gaboardi et al. (2013)); and the use of type systems based on *modal logics* for analysing partial evaluation and run-time code generation Davis and Pfenning (1996).



Slide 93

6.3 Calculus of Constructions, λC

2IPC is a logic of *propositions* (properties without any parameters) rather than *predicates* (properties of individuals of some type). The Curry-Howard correspondence applies to predicate logics as well as propositional logics. Here we take a brief look at a higher-order intuitionistic predicate logic where, among other things, one can universally quantify over individuals of a particular type and where the collection of these types of individuals is closed under formation of function types (this is why the logic is called *higher-order*). The typed lambda calculus corresponding to this under Curry-Howard is the *Calculus of Constructions* of Coquand and Huet (1988). If can be specified as a Pure Type System λC: see Slide 94.

Calculus of Constructions

is the Pure Type System $\lambda\mathbf{C}$, where $\mathbf{C} = (\mathcal{S}_\mathbf{C}, \mathcal{A}_\mathbf{C}, \mathcal{R}_\mathbf{C})$ is the PTS specification with

$\mathcal{S}_\mathbf{C} \triangleq \{\mathtt{Prop}, \mathtt{Set}\}$ (Prop = a sort of propositions, Set = a sort of types)
$\mathcal{A}_\mathbf{C} \triangleq \{(\mathtt{Prop}, \mathtt{Set})\}$ (Prop is one of the types)
$\mathcal{R}_\mathbf{C} \triangleq \{(\mathtt{Prop}, \mathtt{Prop}, \mathtt{Prop})^1, (\mathtt{Set}, \mathtt{Prop}, \mathtt{Prop})^2,$
$(\mathtt{Prop}, \mathtt{Set}, \mathtt{Set})^3, (\mathtt{Set}, \mathtt{Set}, \mathtt{Set})^4\}$

1. Prop has implications, $\phi \to \psi = \Pi x : \phi\,(\psi)$ (where $\phi, \psi : \mathtt{Prop}$ and $x \notin fv(\psi)$).

2. Prop has universal quantifications over elements of a type, $\Pi x : A\,(\phi(x))$ (where $A : \mathtt{Set}$ and $x : A \vdash \phi(x) : \mathtt{Prop}$).
N.B. A might be Prop ($\lambda 2 \subseteq \lambda\mathbf{C}$).

3. Set has types of function dependent on proofs of a proposition, $\Pi x : p\,(A(x))$ (where $p : \mathtt{Prop}$ and $x : p \vdash A(x) : \mathtt{Set}$).

4. Set has dependent function types, $\Pi x : A\,(B(x))$ (where $A : \mathtt{Set}$ and $x : A \vdash B(x) : \mathtt{Set}$).

Slide 94

Some general properties of $\lambda\mathbf{C}$

- It extends both $\lambda 2$ (PLC) and $\lambda\omega$ ($\mathbf{F}_\omega$).
- $\lambda\mathbf{C}$ is strongly normalizing.
- Type-checking and typeability are decidable.
- $\lambda\mathbf{C}$ is logically consistent (relative to the usual foundations of classical mathematics), that is, there is no pseudo-term t satisfying $\diamond \vdash t : \Pi p : \mathtt{Prop}\,(p)$.

  Indeed there is no proof of LEM ($\Pi p : \mathtt{Prop}\,(\neg p \vee p)$).

Slide 95

Slide 95 gives some properties of λC. The expressivity of the Calculus of Constructions goes well beyond PLC and F_ω. For example, the ability to quantify over predicates allows one to define equality predicates for each type using a technique that goes back to Leibniz; see Slide 96. It is possible, in principle, to express a lot of constructive mathematics within λC. It was the starting point for the Coq theorem-proving system (`coq.inria.fr`). Compared with classical higher-order predicate logic (Church, 1940), λC is a *proof-relevant* logic: it is possible to express constructions on proofs of propositions as well as on elements of types (cf. item 3 on Slide 94).

Leibniz equality in $\lambda\mathbf{C}$

Gottfried Wilhelm Leibniz (1646–1716),
identity of indiscernibles:
duo quaedam communes proprietates eorum nequaquam possit
(two distinct things cannot have all their properties in common).

Given $\Gamma \vdash A : \mathtt{Set}$ in $\lambda\mathbf{C}$, we can define

$$\mathtt{Eq}_A \triangleq \lambda x, y : A\,(\Pi P : A \to \mathtt{Prop}\,(P\,x \leftrightarrow P\,y))$$

satisfying $\Gamma \vdash \mathtt{Eq}_A : A \to A \to \mathtt{Prop}$ and giving a well-behaved (but not extensional) equality predicate for elements of type A.

Slide 96

Extensionality

Functional extensionality:

$$\mathtt{FunExt}_{A,B} \triangleq \Pi f, g : A \to B\,(\\ (\Pi x : A\,(\mathtt{Eq}_B\,(f\,x)\,(g\,x))) \to \mathtt{Eq}_{A\to B}\,f\,g)$$

If $\Gamma \vdash A, B : \mathtt{Set}$ in $\lambda\mathbf{C}$, then $\Gamma \vdash \mathtt{Ext}_{A,B} : \mathtt{Prop}$ is derivable, but for some A,B there does not exist a pseudo-term t for which $\Gamma \vdash t : \mathtt{Ext}_{A,B}$ is derivable.

Propositional extensionality:

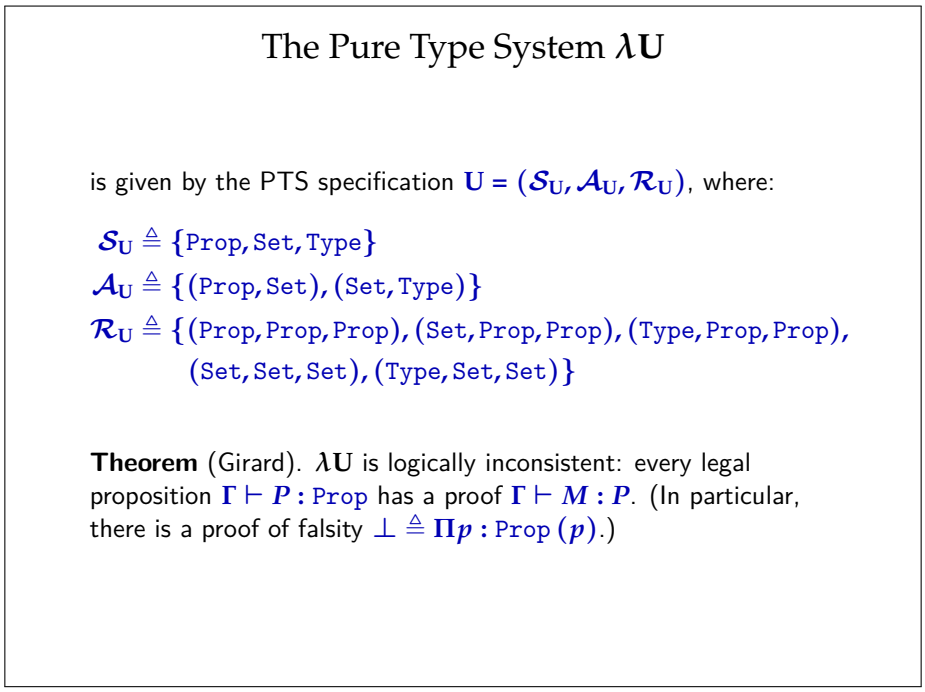
$$\mathtt{PropExt} \triangleq \Pi p, q : \mathtt{Prop}\,((p \leftrightarrow q) \to \mathtt{Eq}_{\mathtt{Prop}}\,p\,q)$$

$\diamond \vdash \mathtt{PropExt} : \mathtt{Prop}$ is derivable in $\lambda\mathbf{C}$, but there does not exist a pseudo-term t for which $\diamond \vdash t : \mathtt{PropExt}$ is derivable.

Slide 97

The universe Prop of propositions in λC is said to be *impredicative*, because in order to construct a proposition, we are allowed to quantify over *all* propositions (as on Slide 91), including the one we are trying to construct! That the collection of all subsets of a set is itself a set is a similar form of impredicativity and strict constructivists do not admit the existence of power*sets*, even though their use is widespread in every day mathematics. λC is also *intensional* in the sense that it fails to validate the well-known principle of *function extensionality*, or the less well-known one of *propositional extensionality* (Slide 97). The latter is a special case of

Voevodsky's *Univalence Axiom*, part of *Homotopy Type Theory* (Univalent Foundations Program, 2013), a topic which is currently under very active development.



Slide 98

6.4 Inductive types

We saw in Section 4.4 how PLC's (impredicative!) ability to quantify over all types when forming type expressions can be used to represent algebraic data types as PLC types. PLC, viewed as the Pure Type System $\lambda 2$, is a subsystem of the Calculus of Constructions, λC. This casts PLC as the language of proofs of the logic 2IPC within λC: its types become propositions (pseudo-terms of sort `Prop`), not types (pseudo-terms of sort `Set`). So for example the polymorphic natural numbers $\forall\alpha\,(\alpha \to (\alpha \to \alpha) \to \alpha)$ becomes a proposition $\diamond \vdash \Pi p : \mathtt{Prop}\,(p \to (p \to p) \to p) : \mathtt{Prop}$ that is provable in countably many different ways (given by the countably many different normal forms of that type). As far as proof-search (Slide 93) is concerned, there is not much to choose between $\Pi p : \mathtt{Prop}\,(p \to (p \to p) \to p)$ and $\Pi p : \mathtt{Prop}\,(p \to p)$, since both are provable. Can we use this kind of impredicative encoding of algebraic data types to produce λC types, rather than propositions? Not as it stands: the pseudo-term $\Pi x : \mathtt{Set}\,(x \to (x \to x) \to x)$ is not legal in λC (exercise). To make it legal one can extend λC with a sort `Type` for 'big' types (in particular `Set` : `Type`), enabling one to add a PTS specification rule $(\mathtt{Type}, \mathtt{Set}, \mathtt{Set})$ expressing that the sort `Set`, like `Prop`, is closed under forming impredicative Π-types. This extension of λC is the PTS known as λU, described on Slide 98.

The theorem on Slide 98, due to Girard (1972), shows that λU is not useful as a logic. A similiar, but slightly easier result holds for the PTS $\lambda*$ mentioned on Slide 76. These results are known as *Girard's Paradox* and are a type-theoretic version of the Burali-Forti set-theoretic paradox; see Coquand (1986) and Barendregt (1992, Section 5.5). If one wishes to use dependently-typed lambda calculi as the basis of theorem-proving systems, then they had better correspond under Curry-Howard to consistent logical systems. So λU and $\lambda*$ are not useful from that point of view. They are of interest from a programming-language point of view, but the impredicative encodings of data types that they admit really are of theoretical rather than practical interest: when using such representations, some standard functions can be hard to

program (for example, subtraction of numbers) and/or computationally inefficient. Instead, one can just extend λC with syntax and reductions that directly (rather than impredicatively) express such data types, following the pattern on Slide 99.

Inductive types (informally)

An inductive type is specified by giving

- **constructor functions** that allow us to inductively generate data values of that type
  (Some restrictions on how the inductive type appears in the domain type of constructors is needed to ensure termination of reduction and logical consistency.)
- **eliminators** for constructing functions on the data
- **computation rules** that explain how to simplify an eliminator applied to constructors.

Slide 99

Extending $\lambda\mathbf{C}$ with an inductive type of natural numbers

Pseudo-terms

$$t ::= \cdots \mid \mathtt{Nat} \mid \mathtt{zero} \mid \mathtt{succ} \mid \mathtt{elimNat}(x.t)\ t\ t$$

Typing rules

- formation: $\diamond \vdash \mathtt{Nat} : \mathtt{Set}$
- introduction: $\diamond \vdash \mathtt{zero} : \mathtt{Nat}$ $\quad$ $\diamond \vdash \mathtt{succ} : \mathtt{Nat} \to \mathtt{Nat}$
- elimination:
$$\frac{\Gamma, x : \mathtt{Nat} \vdash A(x) : s \qquad \Gamma \vdash M : A(\mathtt{zero}) \qquad \Gamma \vdash F : \Pi x : \mathtt{Nat}\ (A(x) \to A(\mathtt{succ}\, x))}{\Gamma \vdash \mathtt{elimNat}(x.A)\ M\ F : \Pi x : \mathtt{Nat}\ (A(x))}$$
  (where $A(t)$ stands for $A[t/x]$)

Computation rules

$$\mathtt{elimNat}(x.A)\ M\ F\ \mathtt{zero} \to M$$
$$\mathtt{elimNat}(x.A)\ M\ F\ (\mathtt{succ}\ N) \to F\ N\ (\mathtt{elimNat}(x.A)\ M\ F\ N)$$

Slide 100

Slide 100 gives the simple example of extending Calculus of Constructions with an inductive type of natural numbers. The pseudo-terms `zero` and `succ` are the constructors for the inductively defined type `Nat` and the pseudo-terms $\mathtt{elimNat}(x.A)\ M\ F$ are its eliminators. The

elimination typing rule in case the sort s is $\mathtt{Set}$ gives a dependently-typed version of *primitive recursive functions* (cf. the CST IB Computation Theory course). For example addition $\diamond \vdash \mathit{add} : \mathtt{Nat} \to \mathtt{Nat} \to \mathtt{Nat}$ is given by

$$\mathit{add} \triangleq \lambda x : \mathtt{Nat}\,(\mathtt{elimNat}(y.\mathtt{Nat})\,x\,(\lambda y : \mathtt{Nat}\,(\mathtt{succ}))) \tag{11}$$

for which the computation rules on Slide 100 yield

$$\mathit{add}\,x\,\mathtt{zero} \to^* x \tag{12}$$

$$\mathit{add}\,x\,(\mathtt{succ}\,y) \to^* \mathtt{succ}(\mathit{add}\,x\,y). \tag{13}$$

Furthermore, the elimination typing rule in case the sort s is $\mathtt{Prop}$ yields, under the Curry-Howard correspondence, a version of the usual principle of *mathematical induction*

$$\frac{\Phi \vdash \phi(\mathtt{zero}) \qquad \Phi \vdash \forall x : \mathtt{Nat}\,(\phi(x) \to \phi(\mathtt{succ}\,x))}{\Phi \vdash \forall x : \mathtt{Nat}\,(\phi(x))} \tag{14}$$

This is typical of the Curry-Howard correspondence for inductive types in a dependently-typed setting: *definition by structural recursion* and *proof by structural induction* are two sides of the same coin, given by the eliminator and its computation rules.

An important consequence of working with dependently-typed systems is that one can consider inductive definitions of whole indexed families of data types with dependently-typed constructor functions: the particular type of data constructed can depend upon the type of the argument to which the constructor is applied. Slides 101 and 102 give two examples. Many more examples can be found in the book by Nordström et al. (1990). The article by Oury and Swierstra (2008) is a nice illustration of the usefulness of dependently-typed inductive types in functional programming, making use of Agda.

Inductive types of vectors

For a fixed parameter $\Gamma \vdash A : s$, the indexed family $(\mathtt{Vec}_A\,x \mid x : \mathtt{Nat})$ of types $\mathtt{Vec}_A\,x$ of **lists of A-values of length x** is inductively defined as follows:

Formation:

$$\frac{\Gamma \vdash N : \mathtt{Nat}}{\Gamma \vdash \mathtt{Vec}_A\,N : \mathtt{Set}}$$

Introduction:

$$\Gamma \vdash \mathtt{vnil}_A : \mathtt{Vec}_A\,\mathtt{zero}$$

$$\Gamma \vdash \mathtt{vcons}_A : A \to \Pi x : \mathtt{Nat}\,(\mathtt{Vec}_A\,x \to \mathtt{Vec}_A\,(\mathtt{succ}\,x))$$

Elimination and Computation:

[do-it-yourself]

Slide 101

Inductive identity propositions

For fixed parameters $\Gamma \vdash A : s$ and $\Gamma \vdash a : A$, the indexed family $(\mathtt{Id}_{A,a}\, x \mid x : A)$ of propositions $\mathtt{Id}_{A,a}\, x$ that a **and** x **are equal elements of type** A is inductively defined as follows:

Formation:

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \mathtt{Id}_{A,a}\, M : \mathtt{Prop}}$$

Introduction:

$$\Gamma \vdash \mathtt{refl}_{A,a} : \mathtt{Id}_{A,a}\, a$$

Elimination:

$$\frac{\Gamma, x : A, p : \mathtt{Id}_{A,a}\, x \vdash B(x,p) : s \qquad \Gamma \vdash N : B(a, \mathtt{refl}_{A,a})}{\Gamma \vdash \mathtt{J}_{A,a}(x, p.B)\, N : \Pi x : A\, (\Pi p : \mathtt{Id}_{A,a}\, x\, (B(x,p)))}$$

Computation:

$$\mathtt{J}_{A,a}(x, p.B)\, N\ a\, \mathtt{refl}_{A,a} \to N$$

Slide 102

Agda proof of $\forall x \in \mathbb{N}\ (x = 0 + x)$

```
data Nat : Set where
  zero : Nat
  succ : Nat -> Nat

add : Nat -> Nat -> Nat
add x zero     = x
add x (succ y) = succ (add x y)

data Id (A : Set)(x : A) : A -> Set where
  refl : Id A x x

cong : (A B : Set)(f : A -> B)(x y : A) ->
       Id A x y -> Id B (f x) (f y)
cong A B f x .x refl = refl

P : (x : Nat) -> Id Nat x (add zero x)
P zero     = refl
P (succ x) = cong Nat Nat succ x (add zero x) (P x)
```

Slide 103

Both Agda and Coq allow the user to define their own inductive types; arguably, this is their most useful feature. The theoretical basis for this is the fact that *it is possible to automatically generate elimination and computation rules from user-supplied formation and introduction rules for a given inductive type* (Coquand and Paulin, 1990). However, writing functions on inductive data in terms of eliminators rapidly becomes very unmanageable. For example, if λC is extended with a type of natural numbers as on Slide 100 and with equality propositions as on Slide 102, then it is possible, but rather tedious, to construct a pseudo-term P satisfying

$$\diamond \vdash P : \Pi x : \mathtt{Nat}\, (\mathtt{Id}_{\mathtt{Nat},x}(add\ \mathtt{zero}\ x)) \tag{15}$$

in other words, to construct a proof (by induction!) of the trivial proposition $\forall x \in \mathbb{N}\,(x = 0 + x)$ (exercise).[10] Instead, both Coq and Agda support fixed point definitions involving case-by-case matching of data to constructor patterns, similar to the kind of definitions allowed by languages in the ML family and in Haskell.[11] For example, the proof of (15) in Agda is shown on Slide 103.

Uniqueness of identity proofs

In $\lambda\mathbf{C}$ extended with inductive identity propositions, there are some types $\Gamma \vdash A : s$ for which it is impossible to prove that all equality proofs in $\mathtt{Id}_{A,x}\,y$ (where $x, y : A$) are identical. That is, there is no pseudo-term uip satisfying

$$\Gamma \vdash uip : \Pi x, y : A\,(\Pi p, q : \mathtt{Id}_{A,x}\,y\,(\mathtt{Id}_{(\mathtt{Id}_{A,x}\,y),p}\,q))$$

By contrast, in Agda we have:

```
data Id (A : Set)(x : A) : A -> Set where
  refl : Id A x x

uip : (A : Set)(x y : A)(p q : Id A x y) -> Id (Id A x y) p q
uip A x .x refl refl = refl
```

Slide 104

Pattern-matching in a dependently-typed setting is more involved than in the simply-typed case. For example the Agda definition of `cong` on Slide 103 matches expressions of type `Id A x y` against the constructor pattern `refl`; since this has type `Id A x y` only when `x` and `y` are the same, the defining clause for `cong` replaces `y` by an 'inaccessible' pattern `.x` that is determined by the pattern variable `x`. What is being implemented by Agda is the pattern-matching algorithm for dependent types due to Coquand (1992); see (Norell, 2007, Chapter 2) for a user-oriented discussion of this. It turns out to be strictly more expressive than algorithms that elaborate patterns into eliminator forms; see Slide 104. The impossibility of proving uniqueness of identity proofs from the rules on Slide 102 is a model-theoretic argument due to Hofmann and Streicher (1998). Recent versions of Agda use the work of Cockx et al. (2014) to optionally constrain pattern-matching to make proofs like the one at the bottom of Slide 104 impossible.

[10] Note that the symmetric version $\forall x \in \mathbb{N}\,(x = x + 0)$ can be proved merely by calculation: by (12), $\Pi x : \mathtt{Nat}\,(\mathtt{Id}_{\mathtt{Nat},x}(add\ x\ \mathtt{zero}))$ reduces to $\Pi x : \mathtt{Nat}\,(\mathtt{Id}_{\mathtt{Nat},x}\,x)$, which is inhabited by $\mathtt{refl}_{\mathtt{Nat},x}$.

[11] One difference is that syntactic restrictions are placed on occurrences of the defined function in the fixed point definition in order to ensure termination of reduction and logical consistency.

7 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 and the propositions-as-type paradigm. The book Pierce (2005) is a good source for essays on further topics in type systems for programming languages.

References

Aho, A. V., R. Sethi, and J. D. Ullman (1986). *Compilers. Principles, Techniques, and Tools*. Addison Wesley.

Aspinall, D. and M. Hofmann (2005). Dependent types. In B. C. Pierce (Ed.), *Advanced Topics in Types and Programming Languages*, Chapter 2, pp. 45–86. The MIT Press.

Augustsson, L. (1998). Cayenne–a language with dependent types. In *ACM SIGPLAN International Conference on Functional Programming, ICFP 1998, Baltimore,Maryland, USA*. ACM Press.

Barendregt, H. P. (1992). Lambda calculi with types. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum (Eds.), *Handbook of Logic in Computer Science*, Volume 2, pp. 117–309. Oxford University Press.

Barthe, G. (1999). Type-checking injective pure type systems. *Journal of Functional Programming 9*(6), 675–698.

Bove, A. and P. Dybjer (2009). Dependent types at work. In *Language Engineering and Rigorous Software Development*, pp. 57–99. Springer.

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.

Cockx, J., D. Devriese, and F. Piessens (2014). Pattern matching without K. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2014, New York, NY, USA, pp. 257–268. ACM.

Coquand, T. (1986, June). An analysis of girard's paradox. In *Proceedings of the First Annual IEEE Symposium on Logic in Computer Science (LICS 1986)*, pp. 227–236. IEEE Computer Society Press.

Coquand, T. (1992, June). Pattern matching with dependent types. In B. Nordström, K. Petersson, and G. D. Plotkin (Eds.), *Proceedings of the 1992 Workshop on Types for Proofs and Programs, Båstad, Sweden*, pp. 66–79.

Coquand, T. and G. Huet (1988). The calculus of constructions. *Information and Computation 76*(2–3), 95–120.

Coquand, T. and C. Paulin (1990). Inductively defined types. In P. Martin-LÃűf and G. Mints (Eds.), *COLOG-88*, Volume 417 of *Lecture Notes in Computer Science*, pp. 50–66. Springer Berlin Heidelberg.

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.

de Bruijn, N. G. (1970). The mathematical language automath, its usage, and some of its extensions. In M. Laudet, D. Lacombe, L. Nolin, and M. SchÃijtzenberger (Eds.), *Symposium on Automatic Demonstration*, Volume 125 of *Lecture Notes in Mathematics*, pp. 29–61. Springer Berlin Heidelberg.

Gaboardi, M., A. Haeberlen, J. Hsu, A. Narayan, and B. C. Pierce (2013). Linear dependent types for differential privacy. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, New York, NY, USA, pp. 357–370. ACM.

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.

Hofmann, M. and T. Streicher (1998). The groupoid interpretation of type theory. In G. Sambin (Ed.), *Twenty-Five Years of Constructive Type Theory*, Volume 36 of *Oxford Logic Guides*, pp. 83–111. Oxford University Press.

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

MacLane, S. (1971). *Categories for the Working Mathematician*. Graduate Texts in Mathematics 5. Springer.

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.

Martin-Löf, P. (1975). An intuituionistic theory of types: Predicative part. In H. E. Rose and J. C. Shepherdson (Eds.), *Logic Colloquium '73*, pp. 73–118. North-Holland.

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.

Mitchell, J. C. and G. D. Plotkin (1988). Abstract types have existential types. *ACM Transactions on Programming Languages and Systems 10*, 470–502.

Nordström, B., K. Petersson, and J. M. Smith (1990). *Programming in Martin-Löf's Type Theory*. Oxford University Press.

Norell, U. (2007, September). *Towards a Practical Programming Language Based on Dependent Type Theory*. Ph. D. thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden.

Oury, N. and W. Swierstra (2008). The power of pi. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, New York, NY, USA, pp. 39–50. ACM.

Peyton Jones, S. L. (2001). Tackling the awkward squad: Monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In A. Hoare, M. Broy, and R. Steinbruggen (Eds.), *Engineering Theories of Software Construction*, pp. 47–96. IOS 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.

Univalent Foundations Program, T. (2013). *Homotopy Type Theory: Univalent Foundations for Mathematics*. Institute for Advanced Study: `http://homotopytypetheory.org/book`.

Wadler, P. (1992). Comprehending monads. *Mathematical Structures in Computer Science 2*, 461–493.

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 2015/16 Part II Types – Errata

Page 41, line 3: Conclusion of the proof at the top of the page should read $x_1 : \alpha \vdash \Lambda\alpha' \, (\lambda x_2 : \alpha' \, (x_2)) : \forall\alpha' \, (\alpha' \to \alpha')$

Pages 52, 52: For the translation into PLC on Slide 63 to make sense, we need to include more explicit type information in `unpack` terms. On Slides 62 and 63, replace "`unpack` E `as` ..." by "`unpack` $E : \exists \, \alpha \, (\tau)$ `as` ..." throughout. Then on Slide 63, replace $E \, \tau'(\lambda\alpha \, (\lambda x : \alpha \, (M')))$ by $E \, \tau'(\Lambda\alpha \, (\lambda x : \tau \, (M')))$.

Page 57, Slide 69: Replace $t \to t$ by $t \to t'$.

Page 57, line -6: Replace "sytax" by "syntax".

Page 61, line 1: Replace "know" by "known".

Page 65, Slide 81: Replace $(\alpha \to \gamma) \to (\beta \to \gamma)$ by $(\alpha \to \beta \to \gamma)$,

Page 68: Replace "Bouwer" by "Brouwer".

Page 69, Slide 86: Replace Γ by Φ.

Page 74, Slide 94: Replace $fv(q)$ by $fv(\psi)$.