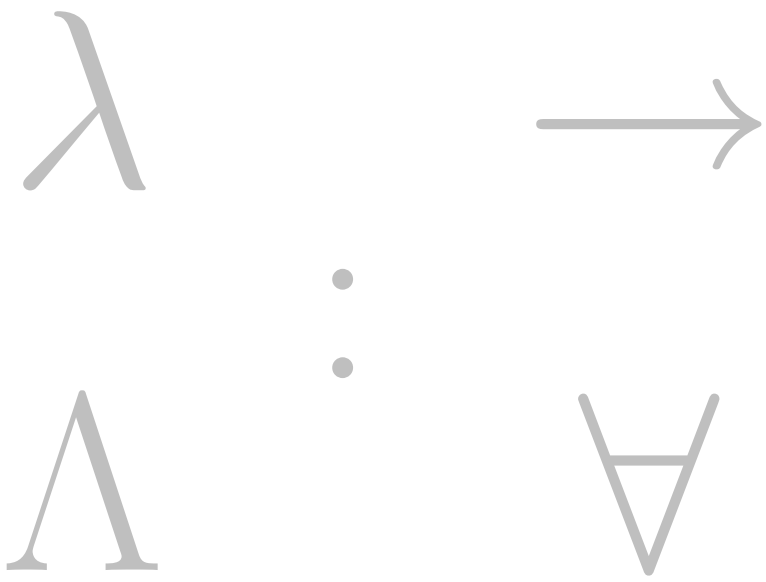
Q

Lecture Notes on

Types

for Part II of the Computer Science Tripos



Dr. Nick Benton
Microsoft Research

Prof. Andrew M. Pitts
Cambridge University Computer Laboratory

First edition 1997.
Revised 1999, 2000, 2001, 2002.

Contents

Learning Guide

These notes and slides are designed to accompany eight lectures on type systems for Part II of the Cambridge University Computer Science Tripos. The Part IB courses on 'Semantics of Programming Languages' and 'Foundations of Functional Programming' are prerequisites.

The course aims to discuss some of the key ideas about the role of types in programming languages, to introduce the mathematical formalism of type systems, and to apply it to a few selected topics—centred mainly around the notion of polymorphism. Formal systems and formal proof play an important role in this subject—a fact which is reflected in the nature of the material presented here and in the kind of questions set on it in the Tripos. As well as learning some specific facts about the ML type system and the polymorphic lambda calculus, at the end of the course you should:

- appreciate how type systems can be used to constrain the dynamic behaviour of programs;
- be able to use a rule-based specification of a type system to show, in simple cases, that a given expression is, or is not typeable;
- be able to describe, in outline, a particular type inference algorithm involving parametric polymorphism.

The dependency between the material in these notes and the lectures will be something like:

	Lecture	Sections	depends on
A	1	1	-
B	2–4	2–3	A
C	5	4	A & B
D	6–8	5–7	A & B

Tripos questions

Here is a list of past Tripos questions that are relevant to the current course.

Year	02	02	01	01	00	00	99	99	98	98	97	97	96	95	95	94	94	93	93	92	91	91	90	90
Paper	7	9	7	9	7	9	7	9	7	9	7	9	9	7	9	8	9	8	9	8	8	9	8	9
Question	13	6	13	6	11	13	12	13	10	13	10	13	13	12	12	13	13	11	11†	11	11	11	9	10

†N.B. watch out for a misprint in this question. In the notation of these notes, $\exists\alpha.\sigma$ should be defined to be $\forall\,\beta\,(\forall\,\alpha\,(\sigma \to \beta) \to \beta)$.

In addition there are a few exercises at the end of most sections.

Additional reading

Section 1 (Cardelli 1997) is highly recommended. Copies of this article are available in the booklocker of the Computer Laboratory Library.

Sections 2–4 (Cardelli 1987) introduces the ideas behind ML polymorphism and type-checking. One could also take a look in (Milner, Tofte, Harper, and MacQueen 1997) at the chapter defining the static semantics for the core language, although it does not make light reading! If you want more help understanding the material in Section 4 (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.)

Sections 5–6 Read (Girard 1989) for an account by one of its creators of the polymorphic lambda calculus (Système F), its relation to proof theory and much else besides.

The recent graduate-level, but still very approachable, text by Pierce (2002) covers much of the material presented in these notes (although not always in the same way). It is also a good starting point for finding out about more advanced topics which are not on the syllabus for this course, such as recursive types and type systems for object-oriented languages.

Note!

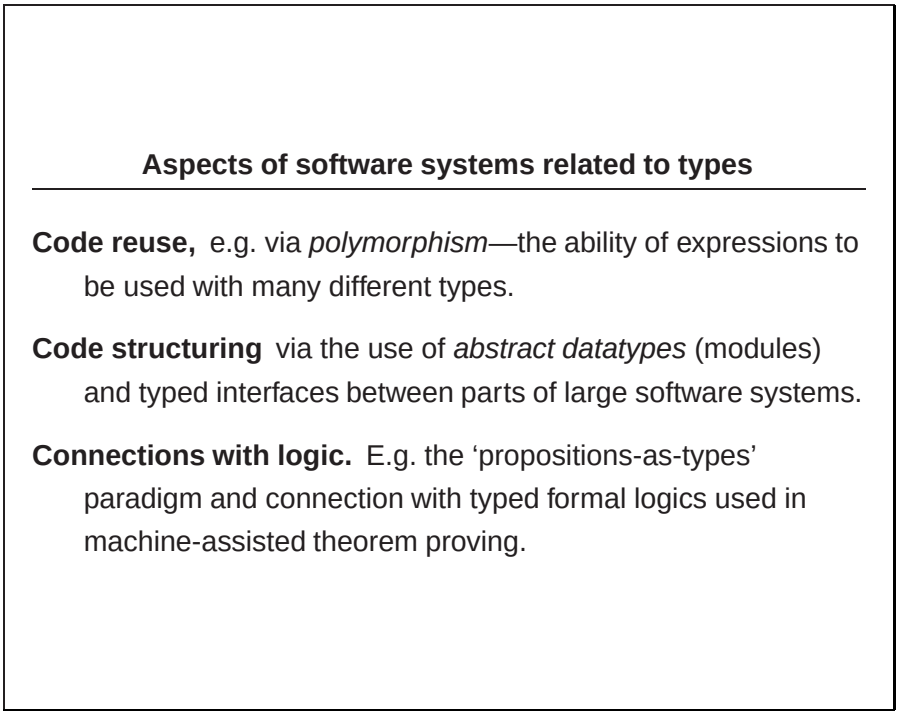
The material in these notes has been drawn from several different sources, including those mentioned above and previous versions of this course by the author and by others. Any errors are of course all my own work. Please let me know if you find typos or possible errors: **a list of corrections will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

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

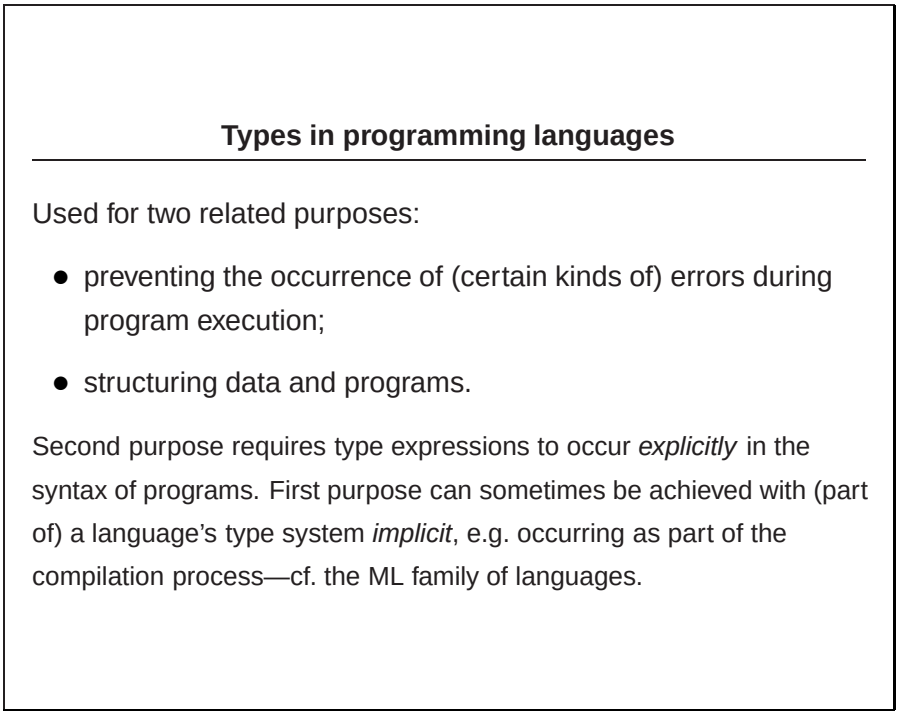
1 Introduction

1.1 The role of types in programming languages

Slides 1 and 2 list some reasons why types are an increasingly crucial aspect of software systems and of programming languages in particular. In very general terms, type systems are used to formulate properties of program phrases. However, unlike the annotation of programs with assertions, for example, the kind of properties expressed by type systems are of a very specific kind. Types classify expressions in a language 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"). Such classifications can not only help with the structuring of programs, but also with efficiency of compilation (by allowing different kinds of representation for different types of data).



Slide 1

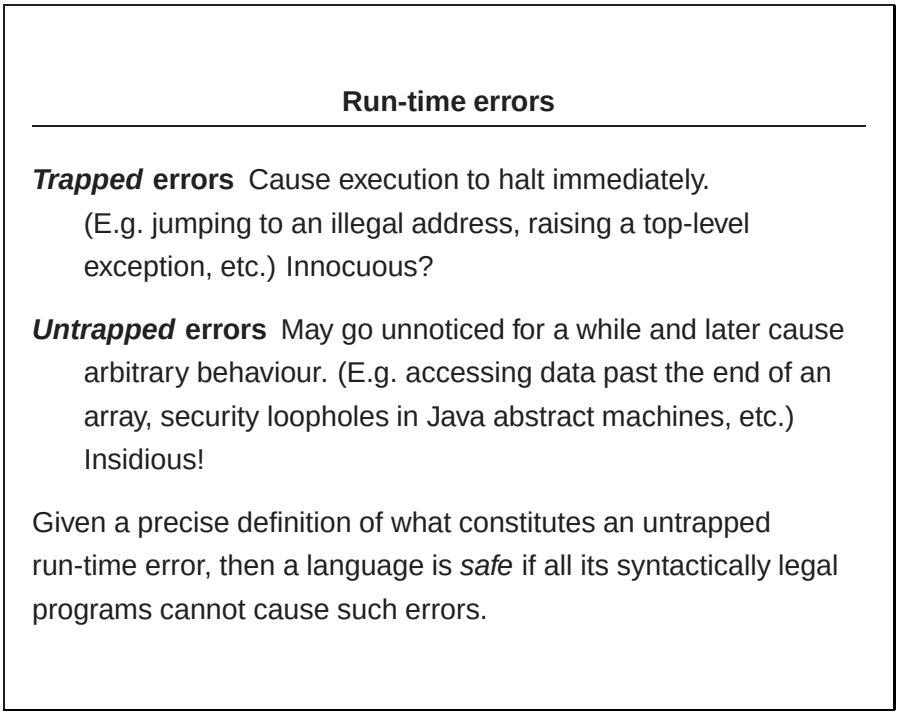


Slide 2

Type systems used to implement checks at compile-time necessarily involve *decidable* properties of program phrases, since otherwise the process of compilation is not guaranteed to terminate. (Recall the notion of (algorithmic) *decidability* from the CST IB 'Computation Theory' course.) For example, in a Turing-powerful language (one that can code all partial recursive functions), it is undecidable whether an arbitrary function definition yields a totally defined function (i.e. one that terminates on all legal arguments). So we cannot expect to have a type system that rules out non-termination at compile-time. The more properties of program phrases a type systems can express the better. But expressivity is constrained in theory by this decidability requirement, and is constrained in practice by questions of computational feasibility.

1.2 Safety via types

Type systems are the principle means to the desirable end of 'safety' (as defined on Slide 3). Of course type systems may be designed to rule out some kinds of trapped error as well: one of the main motivations in the design of type systems for object-oriented languages is to avoid trapped errors of the "method not understood" kind. 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.



Slide 3

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). Many languages (such as C) employ types without any pretensions to safety. 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.[1] We will see an example of this in Section 4, 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. The main point of this course is to introduce a little of this formalism and illustrate its uses.

1.3 Formalising type systems

One can only *prove* that a language is safe after its syntax and operational semantics have been formally specified. Standard ML (Milner, Tofte, Harper, and MacQueen 1997) is the shining example of a full-scale language possessing a complete such specification and whose *type soundness* (cf. Slide 4) has been subject to proof.[2] The study of formal type systems

[1] Object-oriented languages seem particularly prone to this problem and it is a matter of current research to understand why and to design round the problem.

[2] Standard ML is a sufficiently large language that a fully formalised proof of its type safety is surely enormous and certainly requires machine-assistance to carry out. However, since the language design

uses similar techniques as for *structural operational semantics* (cf. CST IB 'Semantics of Programming Languages' course): inductive definitions generated by syntax-directed axioms and rules. A formal type system consists of a number of axioms and rules for inductively generating the kind of assertion, or 'judgement', shown on Slide 5. (Sometimes the type system may involve several different kinds of judgement.) A judgement such as

$$x_1 : \tau_1, x_2 : \tau_2, x_3 : \tau_3 \vdash M : \tau$$

is really just a notation for a formula in predicate calculus, viz.

$$(x_1 : \tau_1)\ \&\ (x_2 : \tau_2)\ \&\ (x_3 : \tau_3)\ \Rightarrow\ (M : \tau)$$

built up from the basic, or atomic, formulas for typing (such as $x_1 : \tau_1$ and $M : \tau$). The reason for adopting special notation for typing judgements is that only very restricted kinds of predicate calculus formulas occur in a type system. (E.g. we want the atomic formulas to the left of $\Rightarrow$ to only refer to identifiers, not compound expressions, and the identifiers should all be distinct from each other.) Furthermore, the axioms and rules for generating valid typing judgements in a given type system will only employ a small part of predicate logic. Finally, the notation emphasises that the phrase M is the main 'subject' of the judgement.

Formal type systems

- Constitute the precise, mathematical characterisation of informal type systems (such as occur in the manuals of most typed language.)
- Basis for *type soundness* theorems: "any well-typed program cannot produce run-time errors (of some specified kind)".
- Can decouple specification of typing aspects of a language from algorithmic concerns (via type inference algorithms).

Slide 4

was semantically-driven and had type safety very much in mind, it is possible to give convincing, if semi-formal, proofs of type safety for large fragments of it.

Typical type system 'judgement'

is a *typing relation* of the form

$$\underset{\text{typing environment}}{\Gamma} \vdash \overset{\text{phrase}}{M} : \underset{\text{type}}{\tau}$$

whose intended meaning is: "given the assignment of types to free identifiers of M specified by type environment Γ, then M has type τ".

E.g.

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

Slide 5

A first example of a formal type system is given on Slides 6 and 7. It assigns types, τ, in the grammar

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

to the expressions, M, of a lambda calculus in which binding occurrences of variables in function abstractions are explicitly tagged with a type:

$$\begin{array}{rcll} M & ::= & x & \text{variable} \\ & | & \mathbf{true} \mid \mathbf{false} & \text{boolean values} \\ & | & \mathbf{if}\ M\ \mathbf{then}\ M\ \mathbf{else}\ M & \text{conditional} \\ & | & \lambda x : \tau(M) & \text{function abstraction} \\ & | & M\ M & \text{function application.} \end{array}$$

Note that the lambda abstraction $\lambda x : \tau(M)$ is a variable-binding construct: free occurrences of x in M become bound in $\lambda x : \tau(M)$. **Here, and throughout this course, we will implicitly identify expressions up to renaming of bound variables**, i.e. up to the equivalence relation of *alpha-conversion*. Thus $\Gamma \vdash M : \tau$ and $\Gamma \vdash M' : \tau$ will be regarded as the same judgement if M is alpha-convertible to M'.

A simple type system, I

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

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

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

Slide 6

A simple type system, II

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

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

Slide 7

Note. If the notions of alpha-conversion and capture-avoiding substitution of expressions for free variables in an expression are at all unfamiliar, please review the relevant material in the lecture notes for the Part IB CST courses on 'Foundations of Functional Programming' and/or 'Semantics of Programming Languages'.

In this particular type system *typing environments*, Γ, are finite functions from variables to types which we write concretely as comma-separated lists of $(identifier : type)$-pairs

$$x_1 : \tau_1, x_2 : \tau_2, \ldots, x_n : \tau_n$$

in which the variables $x_1, x_2, \ldots, x_n$ are all distinct. The set of these variables form the *domain* (of definition of) Γ, written $dom(\Gamma)$. The notation

$$\Gamma, x : \tau$$

used on Slide 7 means the typing environment obtained by extending Γ by mapping a variable x not in $dom(\Gamma)$ to type τ. Thus

$$dom(\Gamma, x : \tau) = dom(\Gamma) \cup \{x\}.$$

As usual, the axiom and rules on Slides 6 and 7 are schematic: Γ, M, τ stand for *any* well-formed type environment, expression and type. The axiom and rules are used to inductively generate the *typing relation*—a subset of all possible triples $\Gamma \vdash M : \tau$. We say that a particular triple $\Gamma \vdash M : \tau$ is *derivable* (or *provable*, or *valid*) in the type system if there is a proof of it using the axioms and rules. Thus the typing relation consists of exactly those triples for which there is such a proof. The construction of proofs is greatly aided by the fact that the axioms and rules are *syntax-directed*: if $\Gamma \vdash M : \tau$ is derivable, then the outermost form of the expression M dictates which must be the last axiom or rule used in any proof of its derivability. For example, if $\emptyset$ denotes the empty type environment then for any types τ_1, τ_2, and τ_3

$$\text{(1)}\quad \emptyset \vdash \lambda x_1 : \tau_2 \to \tau_3(\lambda x_2 : \tau_1 \to \tau_2(\lambda x_3 : \tau_1(x_1\,(x_2\,x_3)))) : (\tau_2 \to \tau_3) \to ((\tau_1 \to \tau_2) \to (\tau_1 \to \tau_3))$$

is derivable in the type system of Slides 6 and 7. (Why? Give a proof for it.) Whereas for no types τ_1, τ_2, and τ_3 is

$$\text{(2)}\qquad \emptyset \vdash \lambda x : \tau_1 \to \tau_2(x\,x) : \tau_3$$

derivable in the system. (Why?)

Note. If the notion of an inductive definition given by axioms and rules is at all unfamiliar, please review the section on Induction in the lecture notes for the CST Part IB course on 'Semantics of Programming Languages'.

Once we have formalised a particular type system, we are in a position to *prove* results about type soundness (Slide 4) and the notions of *type checking*, *typeability* and *type inference*

described on Slide 8. We will see non-trivial examples of such problems in the rest of the course, beginning with type inference for ML-polymorphism. But first let's examine them for the case of the simple type system on Slides 6 and 7.

Type checking, typeability, and type inference

Suppose given a type system TS, say with judgements of the form $\Gamma \vdash M : \tau$.

Type checking problem for TS: given Γ, M, and τ, is $\Gamma \vdash M : \tau$ derivable in TS?

Typeability problem(s) for TS: given M (resp. Γ and M), find Γ, τ (resp. τ) such that $\Gamma \vdash M : \tau$ is derivable in TS (or show there are none).

Second problem is usually harder than the first. Solving it usually results in a *type inference algorithm* computing Γ, τ (resp. τ) for each M (resp. Γ, M) (or failing, if there are none).

Slide 8

The explicit tagging of λ-bound variables with a type means that given a particular typing environment, expressions have a unique type, if any. Because of the structural nature of the typing rules, it is easy to devise a type inference algorithm defined by recursion on the structure of expressions and which shows that the typeability problem is decidable in this case. (Exercise 1.4.3.) To illustrate the notion of type soundness (Slide 4), consider a transition system whose configurations are the *closed* expressions (i.e. the ones with no free variables) together with a distinguished configuration $FAIL$. The terminal configurations are $FAIL$, $\mathbf{true}$, $\mathbf{false}$, and any closed function abstraction $\lambda x : \tau(M)$. A basic step of computation is β-reduction

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

where $M_1[M_2/x]$ is the notation we will use to indicate the result of substituting M_2 for all free occurrences of x in M_1 (as usual, well-defined up to alpha-conversion of λ-bound variables in M_1 to avoid capture of free variables in M_2). An example of a computation failing is

$$\mathbf{true}\, M \to FAIL.$$

The whole transition relation is inductively defined in Figure 1 and the type soundness result is stated on Slide 9. We leave its proof as an exercise (see Exercises 1.4.4 and 1.4.5).

$$\frac{M_1 \to M_1'}{M_1\, M_2 \to M_1'\, M_2}$$

$$\frac{M_1 \to \mathit{FAIL}}{M_1\, M_2 \to \mathit{FAIL}}$$

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

$$\mathbf{true}\, M \to \mathit{FAIL}$$

$$\mathbf{false}\, M \to \mathit{FAIL}$$

$$\frac{M_1 \to M_1'}{\mathbf{if}\, M_1\, \mathbf{then}\, M_2\, \mathbf{else}\, M_3 \to \mathbf{if}\, M_1'\, \mathbf{then}\, M_2\, \mathbf{else}\, M_3}$$

$$\frac{M_1 \to \mathit{FAIL}}{\mathbf{if}\, M_1\, \mathbf{then}\, M_2\, \mathbf{else}\, M_3 \to \mathit{FAIL}}$$

$$\mathbf{if}\, \mathbf{true}\, \mathbf{then}\, M_1\, \mathbf{else}\, M_2 \to M_1$$

$$\mathbf{if}\, \mathbf{false}\, \mathbf{then}\, M_1\, \mathbf{else}\, M_2 \to M_2$$

$$\mathbf{if}\ \lambda x : \tau(M_1)\, \mathbf{then}\, M_2\, \mathbf{else}\, M_3 \to \mathit{FAIL}$$

Figure 1: Axioms and rules for a transition system

A simple type soundness result

For the type system defined on Slides 6 and 7 and the transition system in Figure 1 we have that *if a closed term M is typeable then its evaluation does not fail*. In other words

> if $\emptyset \vdash M : \tau$ holds for some τ, then $M \rightarrow^* FAIL$ does not hold.

(As usual, $\rightarrow^*$ indicates the reflexive-transitive closure of $\rightarrow$).

Slide 9

1.4 Exercises

These exercises refer to the type system defined on Slides 6 and 7.

Exercise 1.4.1. Give a proof of (1) from the axioms and rules.

Exercise 1.4.2. Show that there can be no proof of (2) from the axiom and rules.

Exercise 1.4.3. Show that given Γ and M, there is at most one type τ for which $\Gamma \vdash M : \tau$ is derivable. Describe a type checking algorithm which when given any Γ and M decides whether such a τ exists. Define your algorithm as a Standard ML function on a suitable datatype.

Exercise 1.4.4. Prove the following *substitution property* for the type system defined on Slides 6 and 7: $\Gamma \vdash M_1 : \tau_1 \ \& \ \Gamma, x : \tau_1 \vdash M_2 : \tau_2 \ \Rightarrow \ \Gamma \vdash M_2[M_1/x] : \tau_2$.
[Hint: show by induction on the structure of M_2 that for all Γ, M_1, τ_1, $x \notin dom(\Gamma)$ and τ_2 that if $\Gamma \vdash M_1 : \tau_1$ and $\Gamma, x : \tau_1 \vdash M_2 : \tau_2$ hold, then so does $\Gamma \vdash M_2[M_1/x] : \tau_2$.]

Exercise 1.4.5. Deduce the type soundness result stated on Slide 9 by proving:

(i) If $\emptyset \vdash M : \tau$ and $M \rightarrow M'$, then $\emptyset \vdash M' : \tau$.

(ii) If $M \rightarrow FAIL$, then M is not typeable.

[Hint: prove both by induction on the derivation of transitions from the axioms and rules in Figure 1. You will need the substitution property of the previous exercise for part (i).]

2 ML Polymorphism

2.1 Varieties of polymorphism

Recall the relevance of type systems to code structuring and code reuse mentioned on Slide 1. Static typing (Slide 10) is regarded as an important aid to building large, highly structured, and reliable software systems. On the other hand, early forms of static typing, for example as found in Pascal, tended to hamper code reuse. For example, a procedure for sorting lists of one type of data could not be applied to lists of a different type of data. To try to get the benefits of static typing for code structuring without hampering code reuse too much, it is natural to want, for example, a *polymorphic* sorting procedure—one which operates (uniformly) on lists of several different types. The potential significance for programming languages of this phenomenon of *polymorphism* was first emphasised by Strachey (1967), who identified several different varieties: see Slide 11. Here we will concentrate on parametric polymorphism. One way to achieve it is to make the type parameterisation an explicit part of the language syntax: we will see an example of this in Section 5. Here we will look at the *implicit* version of parametric polymorphism first implemented in the ML family of languages (and subsequently adopted elsewhere). ML phrases need contain little explicit type information: the type inference algorithm infers a 'most general' type (scheme) for each well-formed phrase, from which all the other types of the phrase can be obtained by specialising type variables. These ideas should be familiar to you from your previous experience of Standard ML. The point of this section is to see how one gives a precise formalisation of a type system and its associated type inference algorithm for (a small fragment of) ML.

Note. We use a non-standard ML syntax compared with the definition in (Milner, Tofte, Harper, and MacQueen 1997). For example we write $\lambda x(M)$ for $\texttt{fn}\, x \texttt{=>} M$ and $\mathbf{let}\, x = M_1\, \mathbf{in}\, M_2$ for $\texttt{let val}\, x = M_1\, \texttt{in}\, M_2\, \texttt{end}$. Furthermore we call the symbol 'x' occurring in these expressions a *variable* rather than a '(value) identifier'.

Static versus dynamic typing

Dynamic typing: types of phrases are inferred during program execution.

Static typing: types are inferred at compile time (hence typeability problem has to be decidable).

Slide 10

***Polymorphism* = 'has many types'**

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

Subsumption ordering $\tau_1 <: \tau_2$: any $M_1 : \tau_1$ can be used as $M_1 : \tau_2$ without violating safety (Slide 3).

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

$$\begin{array}{lll} \mathbf{fun} \quad length\,\mathbf{nil} & = & 0 \\ \quad | \quad length\,x :: xs & = & 1 + (length\,xs) \end{array}$$

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

Slide 11

Type variables

To formalise statements like

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

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

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

Slide 12

Polymorphism of $\mathbf{let}$-bound variables

For example in

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

$\lambda x(x)$ has type $\alpha \to \alpha$ (for all α) and the variable f to which it is bound is used polymorphically:

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

Both types are substitution instances of $\alpha \to \alpha$. Overall, the expression has type $bool\ list$.

Slide 13

2.2 Type schemes

As indicated on Slide 12, to formalise parametric polymorphism, we have to introduce *type variables*. An interactive ML system will just display $\alpha\ list \to int$ as the type of the $length$ function (cf. Slide 11), leaving the universal quantification over α implicit. However, when it comes to formalising the ML type system (as is done in the definition of the Standard ML 'static semantics' in Milner, Tofte, Harper, and MacQueen 1997, chapter 4) it is necessary to make this universal quantification over types explicit in some way. The reason for this has to do with the typing of local declarations. Consider the example given on Slide 13. The expression $(f\,\mathbf{true}) :: (f\,\mathbf{nil})$ has type $bool\ list$, given some assumption about the type of the variable f. Two possible such assumptions are shown on Slide 14. Here we are interested in the second possibility since it leads to a type system with very useful properties.

'Ad hoc' polymorphism:

$$(f : bool \to bool)\ \&\ (f : bool\ list \to bool\ list) \Rightarrow (f\,\mathbf{true}) :: (f\,\mathbf{nil}) : bool\ list.$$

'Parametric' polymorphism:

$$\forall\,\alpha\,(f : \alpha \to \alpha) \Rightarrow (f\,\mathbf{true}) :: (f\,\mathbf{nil}) : bool\ list.$$

Slide 14

The simple type system mentioned in Section 1.3 codified predicate calculus formulas of the form

$$\ldots\ \&\ x : \tau\ \&\ \ldots \Rightarrow M : \tau'.$$

To deal with parametric polymorphism and examples like that on Slide 13 we have to use logically more complex formulas:

$$\ldots\ \&\ \forall\,\alpha\,(x : \tau(\alpha))\ \&\ \ldots \Rightarrow \forall\,\alpha'\,(M : \tau'(\alpha')). \tag{3}$$

Since $\Phi \Rightarrow \forall\,\alpha'\,(\Psi(\alpha'))$ is logically equivalent to $\forall\,\alpha'\,(\Phi \Rightarrow \Psi(\alpha'))$, instead of (3) we can use the slightly simpler form

$$\ldots\ \&\ \forall\,\alpha\,(x : \tau(\alpha))\ \&\ \ldots \Rightarrow M : \tau'(\alpha') \tag{4}$$

leaving the outermost quantification over α' implicit. The ML type system makes use of judgements which are a compact notation for predicate calculus formulas like (4). To emphasise that the typing relation is the prime concern, one writes $\forall\, \alpha\, (x : \tau)$ as

$$x : \forall\, \alpha\, (\tau).$$

The expression $\forall\, \alpha\, (\tau)$ is called a *type scheme*. The particular grammar of ML types and type schemes that we will use is shown on Slide 15.

Type schemes over a collection of ML types

Types

τ	$::=$	α	type variable
	$\mid$	$bool$	type of booleans
	$\mid$	$\tau \rightarrow \tau$	function type
	$\mid$	$\tau\ list$	list type

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.

Slide 15

Note. The following points about type schemes should be noted.

(i) If the elements of A are given explicitly, say by a list of distinct type variables, $A = \{\alpha_1, \ldots, \alpha_n\}$, then we write $\forall\, A\, (\tau)$ as

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

In particular, if $A = \{\alpha\}$, then $\forall\, A\, (\tau)$ is written $\forall\, \alpha\, (\tau)$.

(ii) The case when A is empty, $A = \emptyset$, is allowed: $\forall\, \emptyset\, (\tau)$ is a well-formed type scheme. **We will often regard the sets of types as a subset of the set of type schemes by identifying the type τ with the type scheme $\forall\, \emptyset\, (\tau)$.**

(iii) 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 $\{\alpha'\}$.) As usual for variable-binding constructs, we are not interested in the particular names of $\forall$-bound type variables (since we may have to change them to avoid variable capture during substitution of types for free type variables). Therefore *we will identify type schemes up to alpha-conversion of $\forall$-bound type variables.* For example, $\forall\,\alpha\,(\alpha \to \alpha')$ and $\forall\,\alpha''\,(\alpha'' \to \alpha')$ determine the same alpha-equivalence class and will be used interchangeably. Of course the finite set

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

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

$$ftv(\tau)$$

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

(iv) ML type schemes are not ML types! So for example, $\alpha \to \forall\,\alpha'\,(\alpha')$ is neither a well-formed ML type nor a well-formed ML type scheme.[1] Rather, ML type schemes are families of types, parameterised by type variables. We get types from type schemes by substituting types for type variables, as we explain next.

Slide 16 gives some terminology and notation to do with substituting types for the bound type variables of a type scheme. The notion of a type scheme *generalising* a type will feature in the way variables are assigned types in the ML type system of the next subsection.

The 'generalises' relation

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

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

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

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

Slide 16

[1] The step of making type schemes first class types will be taken in Section 5.

Examples 2.2.1. Some simple examples of generalisation:

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

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.

2.3 The ML type system

Slide 17 gives the form of typing judgement we will use to illustrate ML polymorphism and type inference.

ML typing judgement

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

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

Slide 17

As well as only considering a small subset of ML types, we will restrict attention to typings for expressions, M, generated by the following grammar:

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

As usual, the free variables of $\lambda x(M)$ are those of M, except for x. In the expression $\mathbf{let}\, x = M_1\, \mathbf{in}\, M_2$, any free occurrences of the variable x in M_2 become bound in the **let**-expression. Similarly, in the expression $\mathbf{case}\, M_1\, \mathbf{of}\, \mathbf{nil} \Rightarrow M_2 \mid x_1 :: x_2 \Rightarrow M_3$, any free occurrences of the variables x_1 and x_2 in M_3 become bound in the **case**-expression.

The axioms and rules inductively generating the ML typing relation for these expressions are given on Slides 18–20.

ML type system, I

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

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

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

Slide 18

ML type system, II

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

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

(let) $$\frac{\Gamma \vdash M_1 : \tau_1 \qquad \Gamma, x : \forall A\,(\tau_1) \vdash M_2 : \tau_2}{\Gamma \vdash \mathbf{let}\, x = M_1\, \mathbf{in}\, M_2 : \tau_2}$$ if $x \notin dom(\Gamma)$ and $A = ftv(\tau_1) - ftv(\Gamma)$

Slide 19

ML type system, III

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

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

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

Slide 20

Notes 2.3.1.

(i) 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)).

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

(iii) In rule (let) the notation $ftv(\Gamma)$ 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 $A = ftv(\tau_1) - ftv(\Gamma)$ used in that rule consists of all type variables in τ_1 that do not occur freely in any type scheme assigned in Γ. Rule (let) is the characteristic rule of the ML type system: M_1 has type τ_1 for any types we care to substitute for type variables in $A = ftv(\tau_1) - ftv(\Gamma)$, so we type M_2 under the assumption that the variable x bound to M_1 has type scheme $\forall\, A\, (\tau_1)$.

Assigning type schemes

We write $\boxed{\Gamma \vdash_{ML} M : \sigma}$ to indicate that

- $\sigma = \forall\, A\,(\tau)$ with $A \cap \mathit{ftv}(\Gamma) = \emptyset$, and
- the typing judgement $\Gamma \vdash M : \tau$ is derivable from the axioms and rules on Slides 18–20.

When $\Gamma = \emptyset$ we just write $\boxed{\vdash_{ML} M : \sigma}$ for $\emptyset \vdash_{ML} M : \sigma$ and say that the (necessarily closed—see Exercise 2.4.2) expression M is *typeable* in ML with type scheme σ.

Also, when $A = \emptyset$ we write $\Gamma \vdash_{ML} M : \tau$ for $\Gamma \vdash_{ML} M : \forall\, \emptyset\,(\tau)$.

Slide 21

We verify that the example of polymorphism of **let**-bound variables given on Slide 13 has the type claimed there.

Example 2.3.2. Using the notation introduced on Slide 21, we have:

$$\vdash_{ML} \mathbf{let}\ f = \lambda x(x)\ \mathbf{in}\ (f\ \mathbf{true}) :: (f\ \mathbf{nil}) : \mathit{bool}\ \mathit{list}.$$

Proof. First note that $\vdash_{ML} \lambda x(x) : \forall\, \alpha\,(\alpha \to \alpha)$, as witnessed by the following proof:

$$\dfrac{\dfrac{}{x : \alpha \vdash x : \alpha}\ (\text{var} \succ) \quad \text{using } \forall\, \emptyset\,(\alpha) \succ \alpha}{\emptyset \vdash \lambda x(x) : \alpha \to \alpha}\ (\text{fn}). \tag{5}$$

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

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

By (bool) we also have

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

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

$$f : \forall\, \alpha\,(\alpha \to \alpha) \vdash f\ \mathbf{true} : \mathit{bool}. \tag{6}$$

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

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

Applying rule (cons) to (6) and (7) we get

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

Finally we can apply rule (let) to this and (5) to conclude

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

as required. □

2.4 Exercises

Exercise 2.4.1. Here are some type checking problems, in the sense of Slide 8. Prove the following typings:

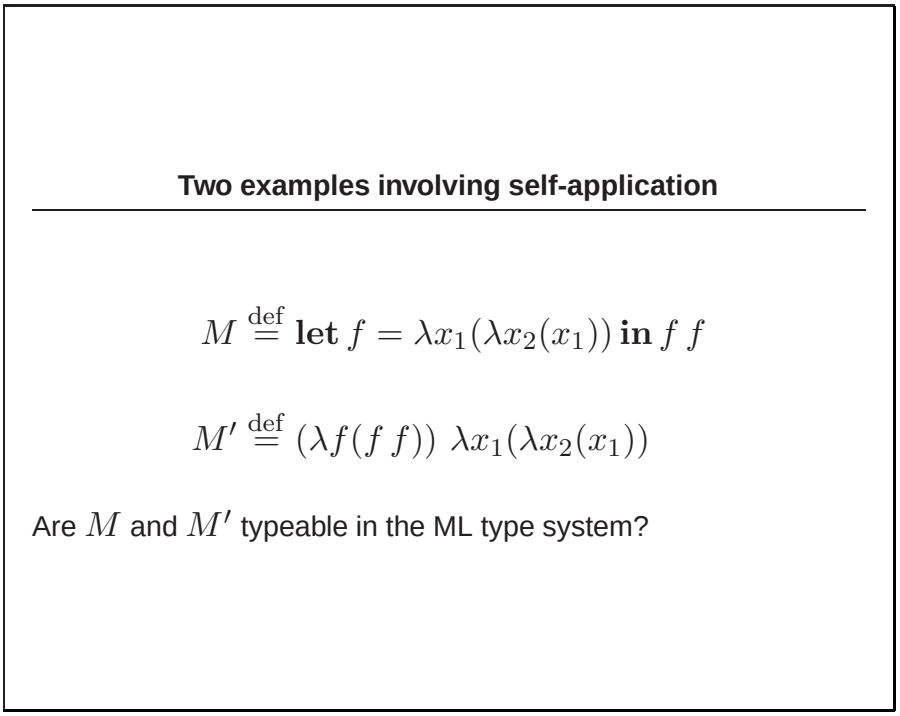
$\vdash_{ML} \lambda x(x :: \mathbf{nil}) : \forall\, \alpha\, (\alpha \to \alpha\ \mathit{list})$
$\vdash_{ML} \lambda x(\mathbf{case}\, x\, \mathbf{of\ nil} \Rightarrow \mathbf{true} \mid x_1 :: x_2 \Rightarrow \mathbf{false}) : \forall\, \alpha\, (\alpha\ \mathit{list} \to \mathit{bool})$
$\vdash_{ML} \lambda x_1(\lambda x_2(x_1)) : \forall\, \alpha_1, \alpha_2\, (\alpha_1 \to (\alpha_2 \to \alpha_1))$
$\vdash_{ML} \mathbf{let}\ f = \lambda x_1(\lambda x_2(x_1))\ \mathbf{in}\ f\ f : \forall\, \alpha_1, \alpha_2, \alpha_3\, (\alpha_1 \to (\alpha_2 \to (\alpha_3 \to \alpha_2))).$

Exercise 2.4.2. Show that if $\emptyset \vdash_{ML} M : \sigma$ (cf. Slide 21), then M must be *closed*, i.e. have no free variables. [Hint: use rule induction for the rules on Slides 18–20 to show that the derivable ML typing judgements, $\Gamma \vdash M : \tau$, all have the property that $\mathit{fv}(M) \subseteq \mathit{dom}(\Gamma)$.]

3 ML Type Inference

For the ML type system introduced in the previous section, the typeability problem (Slide 8) is decidable. Moreover, amongst all the possible type schemes a given closed ML expression may possess, there is a most general one—one from which all the others can be obtained by substitution. Before launching into the general problem of ML typeability, we give some specific examples.

3.1 Examples of type inference, by hand



Slide 22

Consider the expression M given on Slide 22. Exercise 2.4.1 sets a type-checking problem involving M. But if we are not given a type scheme to check, how can we infer one? We are aided in such a task by the *syntax-directed* (or 'structural') nature of the axioms and rules on Slides 18–20: there is one axiom or rule for each clause of the grammar defining ML expressions on page 18. 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 22, any proof of $\vdash_{ML} M : \forall\, A_1\, (\tau_1)$, i.e. any proof of $\emptyset \vdash M : \tau_1$ from the axioms and rules, has to look like the tree given in Figure 2. 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)–

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

Figure 2: Proof tree for $\mathbf{let}\, f = \lambda x_1(\lambda x_2(x_1))\, \mathbf{in}\, f\, f$

(C6) listed on Slide 23 have to be satisfied.

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

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

Slide 23

Thus M is typeable if and only if we can find types $\tau_1, \ldots, \tau_8$ satisfying the constraints on Slide 23. First note that they imply

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

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

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

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

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

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

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 23 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 2 becomes a valid proof of

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

from the typing axioms and rules on Slides 18–20. Hence according to Slide 21 we do have

(8) $$\vdash_{ML} \mathbf{let}\, f = \lambda x_1(\lambda x_2(x_1))\, \mathbf{in}\, f\, f : \forall\, \alpha_3, \alpha_4, \alpha_5\, (\alpha_3 \to (\alpha_4 \to (\alpha_5 \to \alpha_4)))$$

as expected from Exercise 2.4.1.

If we go through the same type inference process for the expression M' on Slide 22 we generate a tree and set of constraints as in Figure 3. 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.

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

Constraints:

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

Figure 3: Proof tree and constraints for $(\lambda f(f\ f))\ \lambda x_1(\lambda x_2(x_1))$

3.2 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 (8), it is in fact the most general, or *principal* type scheme for $\mathbf{let}\ f = \lambda x_1(\lambda x_2(x_1))\ \mathbf{in}\ f\ f$, as defined on Slide 24. This makes use of the natural extension of the 'generalises' relation, $\succ$ (Slide 16), to a binary relation between (closed) type schemes. Exercise 3.4.1 gives an alternative characterisation of this relation.

It is worth pointing out that in the presence of (a), the converse of condition (b) on Slide 24 holds: if $\sigma \succ \sigma'$ and $\vdash_{ML} M : \sigma$, then $\vdash_{ML} M : \sigma'$. This is a consequence of the substitution property of valid ML typing judgements given on Slide 31 below.

Slide 25 gives the main result about the ML typeability problem. It was first proved for a simple type system without polymorphic $\mathbf{let}$-expressions by Hindley (1969) and extended to the full system by Damas and Milner (1982).

Principal type schemes for closed expressions

A type scheme σ is the *principal* type scheme of a closed ML expression M if

(a) $\vdash_{ML} M : \sigma$

(b) for all σ', if $\vdash_{ML} M : \sigma'$ then $\sigma \succ \sigma'$

where by definition $\sigma \succ \sigma'$ holds if $\sigma' = \forall\, A'\, (\tau')$ with $A' \cap \mathit{ftv}(\sigma) = \emptyset$ and $\sigma \succ \tau'$ (as defined on Slide 16).

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

Slide 24

Remark 3.2.1 (Complexity of the type checking algorithm). Although typeability is decidable, it is known to be exponential-time complete. Furthermore, the principal type scheme of an expression can be exponentially larger than the expression itself, even if the type involved is represented efficiently as a directed acyclic graph. More precisely, the time taken to decide typeability and the space needed to display the principal type are both exponential in the number of nested $\mathbf{let}$'s in the expression. For example the expression on Slide 26 (taken from Mairson 1990) has a principal type scheme which would take hundreds of pages to print out. It seems that such pathology does not arise naturally, and that the type checking phase of an ML compiler is not a bottle neck in practice. For more details about the complexity of ML type inference see (Mitchell 1996, Section 11.3.5).

Theorem (Hindley; Damas-Milner)

If the closed ML expression M is typeable (i.e. $\vdash_{ML} M : \sigma$ holds for some type scheme σ), then there is a principal type scheme for M.

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

Slide 25

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

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

(Taken from Mairson 1990.)

Slide 26

Principal typing algorithm, pt

pt is defined recursively, following structure of expressions (and its termination is proved by induction on the structure of expressions).

Clause of definition of pt for each expression-former tries to create an instance of the corresponding axiom/rule in the type system, calling *unification algorithm*, mgu, to solve any necessary equational constraints between types.

pt only $FAIL$s if a call to mgu $FAIL$s.

pt returns *principal* typings because mgu returns *most general* unifiers.

Slide 27

3.3 A type inference algorithm

The aim of this subsection is to sketch the proof of the Hindley-Damas-Milner theorem stated on Slide 25, by describing an algorithm, pt, for deciding typeability and returning a most general type scheme. The main features of pt are set out on Slide 27. As the examples in Section 3.1 should suggest, the algorithm depends crucially upon *unification*—the fact that the solvability of a finite set of equations between algebraic terms is decidable and that a most general solution exists, if any does. This fact was discovered by Robinson (1965) and has been a key ingredient in several logic-related areas of computer science (automated theorem proving, logic programming, and of course type systems, to name three). The form of unification algorithm, mgu, we need here is specified on Slide 28. Although we won't bother to give an implementation of mgu here (see for example (Rydeheard and Burstall 1988, Chapter 8), (Mitchell 1996, Section 11.2.2), or (Aho, Sethi, and Ullman 1986, Section 6.7) for more details), we do need to explain the notation for type substitutions introduced on Slide 28.

Unification of ML types

There is an algorithm mgu which when input two ML types τ_1 and τ_2 decides whether τ_1 and τ_2 are *unifiable*, i.e. whether there exists a type-substitution $S \in \mathrm{Sub}$ with

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

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

(b) $\forall S' \in \mathrm{Sub}(S'(\tau_1) = S'(\tau_2)$
$\Rightarrow \exists T \in \mathrm{Sub}\,(S' = TS))$.

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

Slide 28

Definition 3.3.1 (Type substitutions). A *type substitution* S is a (totally defined) function from type variables to ML types with the property that $S(\alpha) = \alpha$ for all but finitely many α. We write Sub for the set of all such functions. The *domain* of $S \in \mathrm{Sub}$ is the finite set of variables

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

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

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

using the notation for substitution from Slide 16. Notwithstanding the notation on the right hand side of the above equation, we prefer to write the application of a type substitution function S on the left of the type to which it is being applied.[1] As a result, the *composition* TS of two type substitutions $S, T \in \mathrm{Sub}$ means first apply S and then T. Thus by definition TS is the function mapping each type variable α to the type $T(S(\alpha))$ (apply the type substitution T to the type $S(\alpha)$). Note that the function TS does satisfy the finiteness condition required of a substitution and we do have $TS \in Sub$; indeed, $dom(TS) \subseteq dom(T) \cup dom(S)$.

[1] i.e. we write $S\,\tau$ rather than $\tau\,S$ as in last year's Part IB Logic and Proof course.

More generally, if $dom(S) = \{\alpha_1, \ldots, \alpha_n\}$ and σ is an ML type scheme, then $S\,\sigma$ will denote the result of the (capture-avoiding[2]) substitution of $S(\alpha_i)$ for each free occurrence of α_i in σ (for $i = 1..n$).

Principal type schemes for open expressions

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

$$S\,\Gamma \vdash_{ML} M : \sigma$$

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

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

Slide 29

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 29. To compute principal type schemes it suffices to compute 'principal solutions' in the sense of Slide 29: for if M is in fact closed, then any principal solution (S, σ) for the typing problem $\emptyset \vdash M : ?$ has the property that σ is a principal type scheme for M in the sense of Slide 24 (see Exercise 3.4.3).

Slide 30 sets out in more detail what is required of the principal typing algorithm, pt. One possible implementation (in somewhat informal pseudocode, and leaving out the cases for **nil**, cons, and **case**-expressions) is sketched in Figure 4. No claims for its efficiency are made, just for its correctness. This depends crucially upon an important property of ML typing, namely that *it is respected by the operation of substituting types for type variables*. See Slide 31. The proof of this property can be carried out by induction on the proof of the typing judgement from the axioms and rules on Slides 18–20.

[2]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 $S(\alpha_i)$.

Specification for the principal typing algorithm, pt

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

- If $\Gamma \vdash M : ?$ has a solution (cf. Slide 29), 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 30

Type substitutions preserve ML typeability

If

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

is provable from the axioms and rules on Slides 18–20 and $S \in \mathrm{Sub}$ is a type substitution, then

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

is also provable.

Slide 31

Variables:

$$pt(\Gamma \vdash x : ?) \stackrel{\text{def}}{=} \begin{array}{ll} \text{let} & \forall\, A\,(\tau) = \Gamma(x),\ \text{with each } \alpha \in A \text{ fresh} \\ \text{in} & \\ & (Id, \tau) \end{array}$$

Function abstractions:

$$pt(\Gamma \vdash \lambda x(M) : ?) \stackrel{\text{def}}{=} \begin{array}{ll} \text{let} & \text{fresh } \alpha; \\ & (S, \tau) = pt(\Gamma, x : \alpha \vdash M : ?) \\ \text{in} & \\ & (S - \alpha, S(\alpha) \to \tau) \end{array}$$

Function applications:

$$pt(\Gamma \vdash M_1\, M_2 : ?) \stackrel{\text{def}}{=} \begin{array}{ll} \text{let} & (S_1, \tau_1) = pt(\Gamma \vdash M_1 : ?); \\ & (S_2, \tau_2) = pt(S_1\,\Gamma \vdash M_2 : ?); \\ & \text{fresh } \alpha; \\ & S_3 = mgu(S_2\,\tau_1, \tau_2 \to \alpha) \\ \text{in} & \\ & ((S_3 S_2 S_1) - \alpha, S_3(\alpha)) \end{array}$$

let-Expressions:

$$pt(\Gamma \vdash \mathbf{let}\, x = M_1\, \mathbf{in}\, M_2 : ?) \stackrel{\text{def}}{=} \begin{array}{ll} \text{let} & (S_1, \tau_1) = pt(\Gamma \vdash M_1 : ?); \\ & A = ftv(\tau_1) - ftv(S_1\,\Gamma); \\ & (S_2, \tau_2) = pt(S_1\Gamma, x : \forall\, A\,(\tau_1) \vdash M_2 : ?) \\ \text{in} & \\ & (S_2 S_1, \tau_2) \end{array}$$

Booleans ($B = \mathbf{true}, \mathbf{false}$):

$$pt(\Gamma \vdash B : ?) \stackrel{\text{def}}{=} (Id, bool)$$

Conditionals:

$$pt(\Gamma \vdash \mathbf{if}\, M_1\, \mathbf{then}\, M_2\, \mathbf{else}\, M_3 : ?) \stackrel{\text{def}}{=} \begin{array}{ll} \text{let} & (S_1, \tau_1) = pt(\Gamma \vdash M_1 : ?); \\ & S_2 = mgu(\tau_1, bool); \\ & (S_3, \tau_3) = pt(S_2 S_1\,\Gamma \vdash M_2 : ?); \\ & (S_4, \tau_4) = pt(S_3 S_2 S_1\,\Gamma \vdash M_3 : ?); \\ & S_5 = mgu(S_4\,\tau_3, \tau_4) \\ \text{in} & \\ & (S_5 S_4 S_3 S_2 S_1, S_5\,\tau_4) \end{array}$$

Figure 4: Some of the clauses in a definition of pt

Notes 3.3.2 (on the definition in Figure 4).

(i) We have not given the clauses of the definition for **nil**, cons, and **case**-expressions.

(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 notation $S - \alpha$ used in the clause for function abstractions (and also in the one for function applications) is the substitution obtained from S by removing α from its domain. In other words $S - \alpha$ maps α to α and elsewhere maps like S.

(iv) We implicitly assume that all bound variables in expressions are distinct from each other and from any other variables in context. So, for example, the clause for function abstractions tacitly assumes that $x \notin dom(\Gamma)$.

(vi) We do not give the proof that the definition in Figure 4 is correct (i.e. meets the specification on Slide 30): but see Exercise 3.4.5.

3.4 Exercises

Exercise 3.4.1. Let σ and σ' be ML type schemes. Show that the relation $\sigma \succ \sigma'$ defined on Slide 24 holds if and only if

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

[Hint: use the following property of simultaneous substitution:

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

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

Exercise 3.4.2. Try to augment the definition of pt in Figure 4 with clauses for **nil**, cons, and **case**-expressions.

Exercise 3.4.3. Suppose M is a closed expression and that (S, σ) is a principal solution for the typing problem $\emptyset \vdash M : ?$ in the sense of Slide 29. Show that σ must be a principal type scheme for M in the sense of Slide 24. [Hint: first show that S has to be a (finite) permutation of type variables.]

Exercise 3.4.4. Prove the property of the ML type system stated on Slide 31.

Exercise 3.4.5. [hard] Try to give some of the proof that the definition in Figure 4 meets the specification on Slide 30. For example, try to prove that if

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

for $i = 1, 2$, then

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

(Why is it necessary to build the quantification over Γ into the inductive hypotheses?)

4 Polymorphic Reference Types

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

- ML's style of implicitly typed $\mathbf{let}$-bound polymorphism, and
- reference types.

We have already treated the first topic in Sections 2 and 3. The second concerns ML's imperative features, which are based upon the ability to dynamically create locally scoped storage locations which can be written to and read from.[1] We begin by giving the syntax and typing rules for this.

4.1 The problem

We augment the grammar for ML types given on Slide 15 with a unit type (a type with a single value) and *reference* types:

$$\begin{array}{rcll} \tau & ::= & \ldots & \\ & | & \mathit{unit} & \text{unit type} \\ & | & \tau\ \mathit{ref} & \text{reference type.} \end{array}$$

Correspondingly, we augment the grammar for ML expressions given on page 18 as follows:

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

The typing rules for these new forms of expression are given on Slide 32.

[1] The dynamic creation of names of exceptions causes similar problems for type soundness.

Unit, dereference, assignment and reference creation

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

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

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

(ref) $$\frac{\Gamma \vdash M : \tau}{\Gamma \vdash \mathbf{ref}\, M : \tau\ \mathit{ref}}$$

Slide 32

Example 4.1.1

The expression

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

has type unit.

Slide 33

Here is an example of the typing rules on Slide 32 in use.

Example 4.1.1. The expression given on Slide 33 has type $unit$.

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

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

The first of these judgements has the following proof:

$$\dfrac{\dfrac{\dfrac{}{x : \alpha \vdash x : \alpha}\ (\text{var} \succ)}{\emptyset \vdash \lambda x(x) : \alpha \to \alpha}\ (\text{fn})}{\emptyset \vdash \mathbf{ref}\ \lambda x(x) : (\alpha \to \alpha)\ ref}\ (\text{ref}).$$

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

(9) $$r : \forall\, \alpha\, ((\alpha \to \alpha)\ ref) \vdash r := \lambda x'(\mathbf{ref}\, !x') : unit$$

(10) $$r : \forall\, \alpha\, ((\alpha \to \alpha)\ ref), u : unit \vdash (!r)() : unit$$

Writing Γ for the typing environment $\{r : \forall\, \alpha\, ((\alpha \to \alpha)\ ref)\}$, the proof of (9) is

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

while the proof of (10) is

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

□

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. For consider what happens when the expression on Slide 33, call it M, is evaluated.

Evaluation of the outermost **let**-binding in M creates a fresh storage location bound to r and containing the value $\lambda x(x)$. Evaluation of the second **let**-binding updates the contents of r to the value $\lambda x'(\mathbf{ref}\,!x')$ and binds the unit value to u.[1] Next $(!r)()$ is evaluated. This involves applying the current contents of r, which is $\lambda x'(\mathbf{ref}\,!x')$, 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.

Transitions involving references

$\langle !x, s\rangle \rightarrow \langle s(x), s\rangle$ if $x \in dom(s)$

$\langle !V, s\rangle \rightarrow FAIL$ if V not a variable

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

$\langle V := V', s\rangle \rightarrow FAIL$ if V not a variable

$\langle \mathbf{ref}\, V, s\rangle \rightarrow \langle x, s[x \mapsto V]\rangle$ if $x \notin dom(s)$

where V ranges over *values*: $V ::= x \mid \lambda x(M) \mid ()$.

Slide 34

Put more formally, we have $\langle M, \emptyset\rangle \rightarrow FAIL$ in the transition system defined in Figure 5 and Slide 34. The configurations of the transition system are of two kinds:

- A pair $\langle M, s\rangle$, where M is an ML expression and s is a *state*—a finite function mapping variables, x, (here being used as the names of storage locations) to syntactic *values*, V. (The possible forms of V for this fragment of ML are defined in Figure 5.) Furthermore, we require a well-formedness condition for such a pair to be a configuration: the free variables of M and of each value $s(x)$ (as x ranges over $dom(s)$) should be contained in the finite set $dom(s)$.
- The symbol $FAIL$, representing a run-time error.

In giving the axioms and rules for inductively generating the transition system, we have restricted attention to the fragment of ML relevant to Example 4.1.1. The notation $s[x \mapsto V]$

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

The axioms and rules inductively defining the transition system for ML+ref are those on Slide 34 together with the following ones:

$$\langle (\lambda x(M))V', s\rangle \to \langle M[V'/x], s\rangle$$

$$\langle V\,V', s\rangle \to FAIL \quad \text{if } V \text{ not a function abstraction}$$

$$\langle \mathbf{let}\, x = V\, \mathbf{in}\, M, s\rangle \to \langle M[V/x], s\rangle$$

$$\frac{\langle M, s\rangle \to \langle M', s'\rangle}{\langle \mathcal{E}[M], s\rangle \to \langle \mathcal{E}[M'], s'\rangle}$$

$$\frac{\langle M, s\rangle \to FAIL}{\langle \mathcal{E}[M], s\rangle \to FAIL}$$

where V ranges over *values*:

$$V ::= x \mid \lambda x(M) \mid ()$$

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

$$\mathcal{E} ::= - \mid \mathcal{E}\, M \mid V\, \mathcal{E} \mid \mathbf{let}\, x = \mathcal{E}\, \mathbf{in}\, M \mid\, !\mathcal{E} \mid \mathcal{E} := M \mid V := \mathcal{E} \mid \mathbf{ref}\, \mathcal{E}$$

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

Figure 5: Transition system for a fragment of ML with references

used on Slide 34 means the state with domain of definition $dom(s) \cup \{x\}$ mapping x to V and otherwise acting like s. The rules for defining transitions in Figure 5 are specified in the rather compact style of Wright and Felleisen (1994).

4.2 Restoring type soundness

The root of the problem described in the previous section lies in the fact that typing expressions like $\mathbf{let}\, r = \mathbf{ref}\, M_1\, \mathbf{in}\, M_2$ 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.

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

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

Value-restricted typing rule for let-expressions

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

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

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

(Recall that values are given by $V ::= x \mid \lambda x(M) \mid ().$)

Slide 35

The authors of the revised, 1996, definition of Standard ML (Milner, Tofte, Harper, and MacQueen 1997) adopt a simpler solution, proposed independently by Wright (1995). This removes the distinction between applicative and imperative type variables (in effect, all type variables are imperative, but the usual symbols $\alpha, \alpha' \ldots$ are used) while retaining a value-restricted form of the (let) rule, as shown on Slide 35.[1] Thus our version of this type system, call it ML+ref, 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 18–20 and 32, except that the applicability of the (let) rule is restricted as on Slide 35.

Example 4.2.1. The expression on Slide 33 is not typeable in the ML+ref type system.

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 $\mathbf{ref}\ \lambda x(x)$ is *not* a value, the rule has to be applied with $A = \emptyset$. This entails trying to type

$$\mathbf{let}\, u = (r := \lambda x'(\mathbf{ref}\, !x'))\, \mathbf{in}\, (!r)() \tag{11}$$

[1] N.B. what we call a value, (Milner, Tofte, Harper, and MacQueen 1997) calls a *non-expansive expression*. For the fragment of ML we are considering here, such an expression is just an identifier, a function abstraction, or the unit value $()$; but in the full language, there are other kinds of non-expansive expression.

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 (11) 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 the type system ML+ref? The answer lies in a formal proof of a *type soundness* result (generalising the one on Slide 9). One first has to formulate a definition of typing for the configurations of the form $\langle M, s\rangle$ in the transition system we defined in the previous section. Then one proves that a sequence of transitions from such a well-typed configuration can never lead to the $FAIL$ configuration. We do not have the time to give the details in this course: the interested reader is referred to (Wright and Felleisen 1994; Harper 1994) for examples of similar type soundness results.

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 in ML+ref (Exercise 4.3.2.) Wright (1995, Sections 3.2 and 3.3) analyses the consequences of this and presents evidence that it is not a hindrance to the use of Standard ML in practice.

4.3 Exercises

Exercise 4.3.1. Letting M denote the expression on Slide 33 and $\emptyset$ the empty state, show that $\langle M, \emptyset\rangle \to^* FAIL$ is provable in the transition system defined in Figure 5.

Exercise 4.3.2. Give an example of a **let**-expression not involving references which is typeable in the ML type system of Section 2.3, but not in the type system ML+ref of Section 4.2.

5 Polymorphic Lambda Calculus

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

5.1 From type schemes to polymorphic types

We have seen examples (Example 2.3.2 and the first example on Slide 22) of the fact that the ML type system permits $\mathbf{let}$-bound variables to be used polymorphically within the body of a $\mathbf{let}$-expression. As Slide 36 points out, the same is not true of λ-bound variables within the body of a function abstraction. This is a consequence of the fact that ML types and type schemes are separate syntactic categories and the function type constructor, $\rightarrow$, operates on the former, but not on the latter. Recall that an important purpose of type systems is to provide *safety* (Slide 3) via *type soundness* (Slide 4). Use of expressions such as those mentioned on Slide 36 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 37. So let us consider extending the ML type system to assign polymorphic types to expressions. So we consider judgements of the form $\Gamma \vdash M : \pi$ where:

- π is a polymorphic type;
- Γ is a finite function from variables to polymorphic types.

In order to make full use of the mixing of $\rightarrow$ and $\forall$ present in polymorphic types we have to replace the axiom (var $\succ$) of Slide 18 by the axiom and two rules shown on Slide 38. (These are in fact versions for polymorphic types of 'admissible rules' in the original ML type system.) In rule (spec), $\pi[\pi'/\alpha]$ indicates the polymorphic type resulting from substituting π' for all free occurrences of α in π.

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

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

Syntactically, because in rule

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

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

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

Slide 36

Monomorphic types . . .

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

. . . and **type schemes**

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

Polymorphic types

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

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

Slide 37

Identity, Generalisation and Specialisation

(id) $\Gamma \vdash x : \pi$ if $(x : \pi) \in \Gamma$

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

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

Slide 38

Example 5.1.1. In the modified ML type system (with polymorphic types and (var $\succ$) replaced by (id), (gen), and (spec)) one can prove the following typings for expressions which are untypeable in ML:

(12) $\emptyset \vdash \lambda f((f\,\mathbf{true}) :: (f\,\mathbf{nil})) : \forall\, \alpha\, (\alpha \to \alpha) \to \mathit{bool\ list}$

(13) $\emptyset \vdash \lambda f(f\, f) : \forall\, \alpha\, (\alpha) \to \forall\, \alpha\, (\alpha).$

Proof. The proof of (12) is rather easy to find and is left as an exercise. Here is a proof for (13):

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

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. □

Fact (see Wells 1994):

For the modified ML type system with polymorphic types and $(\mathrm{var}\succ)$ replaced by the axiom and rules on Slide 38, *the type checking and typeability problems* (cf. Slide 8) *are equivalent and undecidable.*

Slide 39

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 39: 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 $(\mathrm{var}\succ)$ axioms) and push uses of (gen) right down to the root of the tree (and leave them implicit).

5.2 The PLC type system

The negative result on Slide 39 does not rule out the use of the polymorphic types of Slide 37 in programming languages, since one may consider *explicitly typed* languages (Slide 40) where the tagging of expressions with type information renders the typeability problem essentially trivial. We consider such a language in this subsection.

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 5.1.1)

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

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

Slide 40

Remark 5.2.1 (Explicitly typed languages). One often hears the view that programming languages which enforce a large amount of explicit type information in programs are inconveniently verbose and/or force the programmer to make algorithmically irrelevant decisions about typings. But of course it really depends upon the intended applications. At one extreme, in a scripting language (interpreted interactively, used by a single person to develop utilities in a rapid edit-run-debug cycle) implicit typing may be desirable. Whereas at the opposite extreme, a language used to develop large software systems (involving separate compilation of modules by different teams of programmers) may benefit greatly from explicit typing (not least as a form of documentation of programmer's intentions, but also of course to enforce interfaces between separate program parts). Apart from these issues, explicitly typed languages are useful as *intermediate languages* in optimising compilers, since certain optimising transformations depend upon the type information they contain. See (Harper and Stone 1997), for example.

Functions on types

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

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

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

as well as the usual form of beta-reduction from λ-calculus, $((\lambda\,x:\tau\,(M_1))\,M_2 \to M_1[M_2/x])$.

Slide 41

PLC syntax

Types	τ	$::=$	α	type variable
		$\mid$	$\tau \to \tau$	function type
		$\mid$	$\forall\,\alpha\,(\tau)$	$\forall$-type
Expressions				
	M	$::=$	x	variable
		$\mid$	$\lambda\,x:\tau\,(M)$	function abstraction
		$\mid$	$M\,M$	function application
		$\mid$	$\Lambda\,\alpha\,(M)$	type generalisation
		$\mid$	$M\,\tau$	type specialisation

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

Slide 42

The explicit type information we need to add to expressions to get syntax-directed versions of the (gen) and (spec) rules (Slide 38) concerns the operations of *type generalisation* and *type specialisation*. These are forms of function abstraction and application respectively—for functions defined on the collection of all types (and taking values in one particular type), rather than on the values of one particular type. See Slide 41. The polymorphic lambda calculus, PLC, provides rather sparse means for defining such functions—for example there is no 'typecase' construct that allows branching according to which type expression is input. As a result, PLC is really a calculus of *parametrically polymorphic* functions (cf. Slide 11). The PLC syntax is given on Slide 42. Its types, τ, are like the polymorphic types, π, given on Slide 37, except that we have omitted $bool$ and $(\_)\ list$—because in fact these and many other forms of datatype are representable in PLC (see Section 6 below). We have also omitted **let**-expressions, because (unlike the ML type system presented in Section 2.3) they are definable from function abstraction and application with the correct typing properties: see Exercise 5.4.2.

Notes 5.2.2. The following points about PLC syntax should be noted.

(i) **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

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

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

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

Moreover, since types occur in expressions, we have to consider the *free type variables of an expression*. The only type variable binding construct at the level of expressions is

generalisation: Any occurrences in M of a type variable α become bound in $\Lambda\,\alpha\,(M)$. Thus

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

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

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

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

(ii) **Operator association and scoping**:
As in the ordinary lambda calculus, one often writes a series of 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

$$\forall\,\alpha_1\,(\forall\,\alpha_2\,(\tau \to \alpha_1) \to \alpha_1).$$

(iii) One often writes iterated binders using lists of bound (type) variables:

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

(iv) It is common to write a type specialisation by subscripting the type:

$$M_\tau \stackrel{\text{def}}{=} M\, \tau.$$

The PLC type system uses typing judgements of the form shown on Slide 43. Its typing relation is the collection of such judgements inductively defined by the axiom and rules in Figure 6. By now there should be no surprises here: the axiom for variables and rules for function abstraction and application are just like those for the simple type system we used in Section 1.3; the rules for generalisation and specialisation are just the explicitly typed versions of the ones on Slide 38.

PLC typing judgement

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

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

Slide 43

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

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

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

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

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

Figure 6: Axiom and rules of the PLC type system

An incorrect 'proof'

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

Slide 44

Remark 5.2.3 (Side-condition on rule (gen)). To illustrate the force of the side-condition on rule (gen), consider the last step of the 'proof' on Slide 44. It is not a correct instance of the (gen) rule, because the concluding judgement, whose typing environment $\Gamma = \{x_1 : \alpha\}$, does not satisfy $\alpha \notin \mathit{ftv}(\Gamma)$ (since $\mathit{ftv}(\Gamma) = \{\alpha\}$ in this case). On the other hand, the expression $\Lambda\,\alpha\,(\lambda\,x_2 \,:\, \alpha\,(x_2))$ does have type $\forall\,\alpha\,(\alpha \to \alpha)$ given the typing environment $\{x_1 : \alpha\}$. Here is a correct proof of that fact:

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

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

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

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 5.2.4. On Slide 40 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:

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

Example 5.2.5. There is no PLC type τ for which

$$\emptyset \vdash \Lambda\,\alpha\,((\lambda\,x : \alpha\,(x))\,\alpha) : \tau \tag{14}$$

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 (14) would have to look like

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

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 45

5.3 PLC type inference

As Examples 5.2.4 and 5.2.5 suggest, the type checking and typeability problems (Slide 8) are very easy to solve for the PLC 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 very similar to that for the simple type system of Section 1.3 (Exercise 1.4.3) and is summarised on Slide 45. The 'uniqueness of types' property stated on the slide is easy

Variables:

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

Function abstractions:

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

Function applications:

$$\begin{array}{lll} typ(\Gamma \vdash M_1\, M_2 : ?) \stackrel{\text{def}}{=} & \text{let} & \\ & \quad \tau_1 = typ(\Gamma \vdash M_1 : ?); & \\ & \quad \tau_2 = typ(\Gamma \vdash M_2 : ?) & \\ & \text{in} \quad \text{case } \tau_1 \quad \text{of} \quad \tau_2 \to \tau \quad \Rightarrow \quad \tau & \\ & \qquad\qquad\qquad\qquad \mid \qquad \_ \quad \Rightarrow \quad FAIL & \end{array}$$

Type generalisations:

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

Type specialisations:

$$\begin{array}{lll} typ(\Gamma \vdash M\, \tau_2 : ?) \stackrel{\text{def}}{=} & \text{let} & \\ & \quad \tau = typ(\Gamma \vdash M : ?) & \\ & \text{in} \quad \text{case } \tau \quad \text{of} \quad \forall\, \alpha\, (\tau_1) \quad \Rightarrow \quad \tau_1[\tau_2/\alpha] & \\ & \qquad\qquad\qquad\qquad \mid \qquad \_ \quad \Rightarrow \quad FAIL & \end{array}$$

Figure 7: Definition of typ

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 in Figure 7. 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 $\alpha \notin ftv(\Gamma)$.

5.4 Exercises

Exercise 5.4.1. Give a proof inference tree for (12) in Example 5.1.1. Show that

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

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

Exercise 5.4.2. In PLC, defining the expression $\mathbf{let}\,x = M_1 : \tau\,\mathbf{in}\,M_2$ to be an abbreviation for $(\lambda\,x : \tau\,(M_2))\,M_1$, show that the typing rule

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

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

Exercise 5.4.3. The *erasure*, $\mathit{erase}(M)$, of a PLC expression M is the expression of the untyped lambda calculus obtained by deleting all type information from M:

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

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

(ii) We saw in Example 5.2.4 that there is a closed PLC expression M of type $\forall\,\alpha\,(\alpha) \to \forall\,\alpha\,(\alpha)$ satisfying $\mathit{erase}(M) = \lambda\,f\,(f\,f)$. Find some other closed, typeable PLC expressions with this property.

(iii) [For this part you will need to recall, from the CST Part IB 'Foundations of Functional Programming' course, some properties of beta reduction of expressions in the untyped lambda calculus.] A theorem of Girard says that if $\vdash M : \tau$ is provable in the PLC type system, then $\mathit{erase}(M)$ is strongly normalisable in the untyped lambda calculus, i.e. there are no infinite chains of beta-reductions starting from $\mathit{erase}(M)$. Assuming this result, exhibit an expression of the untyped lambda calculus which is not equal to $\mathit{erase}(M)$ for any closed, typeable PLC expression M.

Exercise 5.4.4. Attack or defend the following statement.

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

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

6 Datatypes in PLC

The aim of this section 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.

Beta-reduction of PLC expressions

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

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

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

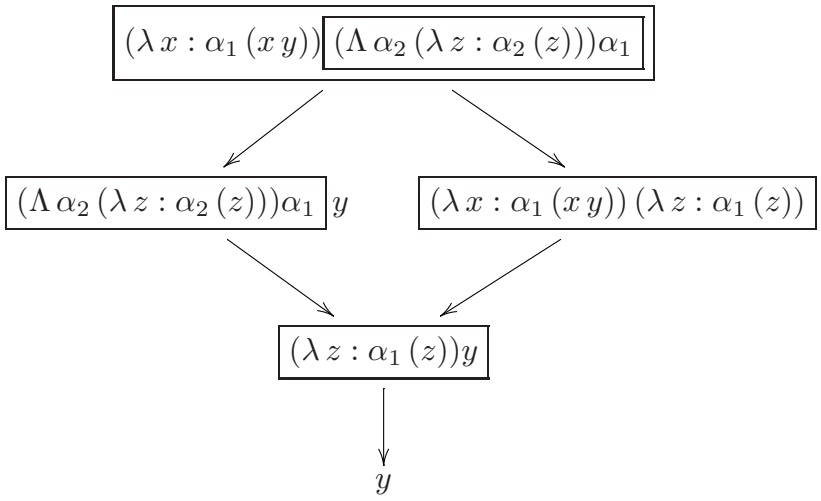
M is in *beta-normal form* if it contains no redexes.

Slide 46

6.1 PLC dynamics

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 46.

Example 6.1.1. Here are some examples of beta-reductions. The various redexes are shown boxed. Clearly, the final expression y is in beta-normal form.



Properties of PLC beta-reduction on typeable expressions

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

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

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

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

Slide 47

Slide 47 lists some important properties of *typeable* PLC expressions. The first is a weak form of type soundness result (Slide 4) and its proof is sufficiently straightforward that it may be set as an exercise (Exercise 6.3.1). We do not give the proofs of the Church Rosser and Strong Normalisations properties here. The latter is a very difficult result.[1] It was first

[1]Since it in fact implies the consistency of second order arithmetic, it furnishes a concrete example

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

PLC beta-conversion, $=_\beta$

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

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

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 48

Theorem 6.1.2. *The properties listed on Slide* 47 *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* 48*) 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

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.

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

Remark 6.1.3. In fact, the Church Rosser property holds for all PLC expressions, whether or not they are typeable. However, the Strong Normalisation properties definitely fails for *untypeable* expressions. For example, consider

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

from which there is an infinite chain of beta-reductions, namely

$$\Omega_\alpha \to \Omega_\alpha \to \Omega_\alpha \to \cdots\ .$$

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 47 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.

6.2 Algebraic datatypes

Roughly speaking an *algebraic* datatype (or datatype constructor) is one which is defined (usually recursively) using products, sums and previously defined algebraic datatype constructors. Thus in Standard ML such a datatype constructor (called alg, with constructors $C_1, \ldots, C_m$) might be declared by

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

where the types $\tau_1, \ldots, \tau_m$ 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. Such algebraic datatypes can be represented in polymorphic lambda calculus. We indicate this by giving some examples. For a more systematic treatment see (Girard 1989, Sections 11.3–5).

Polymorphic booleans

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

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

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

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

Slide 49

Example 6.2.1 (Booleans). The polymorphic type corresponding to the ML algebraic type

$$\mathbf{datatype}\ bool = True \mid False;$$

is shown on Slide 49. The idea behind this representation is that the 'algorithmic essence' of a boolean, b, is the operation $\lambda x_1 : \alpha, x_2 : \alpha(\mathbf{if}\, b\, \mathbf{then}\, x_1\, \mathbf{else}\, x_2)$ (of type $\alpha \to \alpha \to \alpha$[1]), which takes a pair of expressions of the same type and returns 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 49 the typings

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

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

$$\mathbf{case}\, M_1\, \mathbf{of}\ True \Rightarrow M_2 \mid False \Rightarrow M_3$$

has an explicitly typed analogue in PLC, viz. $if\, \tau\, M_1\, M_2\, M_3$, where τ is supposed to be the common type of M_2 and M_3 and if is the PLC expression given on Slide 49. It is not hard to see that

$$\emptyset \vdash if : \forall\, \alpha\, (bool \to (\alpha \to (\alpha \to \alpha))).$$

[1] Recall the notational conventions of Notes 5.2.2: $\alpha \to \alpha \to \alpha$ means $\alpha \to (\alpha \to \alpha)$.

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 6.) Furthermore, the expressions True, False, and if have the expected dynamic behaviour: if

$$M_1 \to^* \mathit{True} \quad \text{and} \quad M_2 \to^* N$$

or

$$M_1 \to^* \mathit{False} \quad \text{and} \quad M_3 \to^* N$$

then it is not hard to see that

$$\mathit{if}\,\tau\,M_1\,M_2\,M_3 \to^* N.$$

(Cf. the transition rules in Figure 1.) It is the case that True *and* False *are the only closed beta-normal forms in PLC of type* bool (up to alpha-conversion, of course), but it is beyond the scope of this course to prove it.

Polymorphic lists

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

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

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

Slide 50

Iteratively defined functions on finite lists

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

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

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

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

Slide 51

Example 6.2.2 (Lists). The polymorphic type corresponding to the ML algebraic datatype

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

is shown on Slide 50. 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 51. 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

$$\lambda x' : \alpha', f : \alpha \to \alpha' \to \alpha' (listIter\ x'\ f\ \ell) \tag{15}$$

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

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

As shown on Slide 52, 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 (15) up to beta-conversion.

List iteration in PLC

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

satisfies:

$$\emptyset \vdash iter : \forall\,\alpha, \alpha'\,(\alpha' \to (\alpha \to \alpha' \to \alpha') \to \alpha\;list \to \alpha')$$
$$iter\,\alpha\,\alpha'\,x'\,f\,(Nil\,\alpha) =_\beta x'$$
$$iter\,\alpha\,\alpha'\,x'\,f\,(Cons\,\alpha\,x\,\ell) =_\beta f\,x\,(iter\,\alpha\,\alpha'\,x'\,f\,\ell).$$

Slide 52

Remark 6.2.3. 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.)

Figure 8 gives some other algebraic datatypes and their representations as polymorphic types. See (Girard 1989, Sections 11.3–5) for more details.

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

Figure 8: Some more algebraic datatypes

6.3 Exercises

Exercise 6.3.1. Prove the various typings and beta-reductions asserted in Example 6.2.1.

Exercise 6.3.2. Prove the various typings asserted in Example 6.2.2 and the beta-conversions on Slide 52.

Exercise 6.3.3. For the polymorphic product type $\alpha_1 * \alpha_2$ defined in the right-hand column of Figure 8, show that there are PLC expressions $Pair$, fst, and snd satisfying:

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

Exercise* 6.3.4. Suppose that τ is a PLC type with a single free type variable, α. Suppose also that T is a closed PLC expression satisfying

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

Define ι to be the closed PLC type

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

Show how to define PLC expressions R and I satisfying

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

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. In this section I enumerate some of the other topics which have been, or may be, important in the development of the theory and application of type systems in computer science, together with some pointers to the literature.

Logical type theories

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). More generally, logical type theories 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* has been highly influential: see Nordström, Petersson, and Smith 1990 for an introduction to it. 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 'propositions-as-types' correspondence. If one identifies a logical proposition with the set of all its proofs, then propositions can be regarded as (maybe special kinds of) types, and "p is a proof of ϕ" gets replaced by "M is an expression of type τ". A formal version of this idea is the so-called *Curry-Howard correspondence* between certain systems of constructive logic and certain typed lambda calculi: see (Girard 1989, Chapter 3). The correspondence cuts both ways: in one direction it has proved very helpful to use lambda terms as notations for proofs in mechanised proof assistants; in the other it has helped to suggest new type systems for programming and specification languages.

Abstract types and types for objects

Existentially quantified types, $\exists\,\alpha\,(\tau)$, are dual to the $\forall$-types we considered in the polymorphic lambda calculus (Section 5). Roughly speaking the values of such a type are pairs (τ', M) where τ' is a type and M is a value of type $\tau[\tau'/\alpha]$.[1] Such $\exists$-types are like ML's notion of signature: they classify implementations of abstract data types: see (Mitchell and Plotkin 1988). It is another indication of the expressive power of polymorphic lambda calculus that $\exists$-types can in fact be encoded in PLC via the formula

$$\exists\,\alpha\,(\tau) \stackrel{\text{def}}{=} \forall\,\alpha'\,(\forall\,\alpha\,(\tau \to \alpha') \to \alpha')$$

[1] In an explicitly typed language, such as PLC, one has to include some extra type information with such a pair; see (Cardelli 1997, Section 5), for example.

(where $\alpha' \notin ftv(\tau)$).

Existential types have also proved useful in formulating type systems for object-oriented languages: see (Pierce and Turner 1994) for example. Such languages inevitably involve consideration of a *subtype* relation between types, $\tau <: \tau'$. This relation is often motivated by the simple notion of a set (of values of some type) being a *subset* of another set (of values of some other type). However, its use in type systems for programming languages is more often tied up with the notion of *subsumption* given on Slide 11. In class-based object-oriented languages one also has similar-looking notions of subclassing and inheritance. Type systems have been used to analyse the meaning and inter-relationship of concepts like 'subtype', 'subsumption', 'subclass', and 'inheritance' as they occur in object-oriented languages (and their relationship to the dynamic semantics of such languages). This is currently a vigorous area of research: see the book by Abadi and Cardelli (1996).

Concurrency and distributed systems

The typing of languages involving concurrent threads of computation and associated notions of mobility and distribution is so current a topic of research that it is difficult to give pointers to well-digested accounts (and the references in this section will certainly be more accessible after you have attended the Part II Topics in Concurrency course). A basic motivation for the use of type systems here is the same as for more traditional languages: to avoid unsafe or undesirable behaviour via static checks. However the kinds of unsafe behaviour are now much more complicated, or at least, less well-understood. For classical concurrent programming, there are, for example, type systems which can ensure that locks are used correctly (Flanagan and Abadi 1999). In distributed (and possibly mobile) settings, there are a number of type systems which further classify values by the *place* at which they reside in a network and/or the resources to which they have access, see Hennessy and Riely (2002), for example.

Another direction of research looks at type systems which can be used to express and check that a sequence of interactions between two systems adheres to some protocol. The technical report by Gay, Vasconcelos, and Ravara (2003) is fairly readable, see also Chaki, Rajamani, and Rehof (2002). In the case that the two systems belong to different organisations (for example, a travel agent and an airline) such type systems suggest the idea of a type as a kind of contract being taken rather literally. This is not only a very interesting and a very challenging area, but also one of rather immediate practical concern.

Types and security

We have seen how static type checking can ensure that certain kinds of error will never happen when a program is executed. Such basic safety guarantees (for example, proving that an integer will never be treated as a pointer) are the foundation of many systems for deciding whether code obtained from a potentially untrustworthy source is safe to execute. Both the Java virtual machine and the .NET CLR include type checkers (or *verifiers*), which are run before code is executed. A nice overview of the internals of JVM verification has been written by Leroy (2003). The correctness of higher-level security operations (such as

the management of explicit permissions to perform potentially-unsafe operations) relies on the typability of any untrusted code which will be allowed to execute.

Type systems are also being used to formalise and check properties which are more security-specific. One line of research classifies the inputs and outputs of a program as either high-security and low-security. A type system can then be used to ensure that high-security information cannot affect low-security outputs (imagine downloading a banking application which has to communicate over the network to retrieve current tax rates, etc., but which you wish to be sure will not leak any of your personal information). See Volpano, Smith, and Irvine (1996) for example. There are also type systems which can statically check access-control systems such as that of Java – if a program typechecks in such a system then one can be sure that all the dynamic permission tests will succeed (and can therefore be removed). See the paper by Skalka and Smith (2000) for example.

Types for low-level languages

Traditionally, sophisticated type systems have been associated with high-level programming languages. Low-level languages such as C or assembler have made do with either no types or type systems which are both inexpressive and unsafe. In particular, high-level, safe, typed languages have been compiled into low-level, untyped, unsafe ones. Recent years have seen a great deal of research activity on typed assembly language (TAL) and type-preserving compilation (Morrisett, Walker, Crary, and Glew 1999). The idea here is to compile an ML program, for example, into a typed assembly language program in such a way that checking the types on the assembly code gives the same safety guarantees as one gets from the ML type system with respect to a high-level operational semantics for ML. This is clearly similar to the use of bytecode verification discussed above; the difference is that the intermediate languages of the JVM and CLR are fairly high-level, so verification is similar to type checking Java or C♯ source (and therefore not too difficult) but the interpreter or JIT compiler which runs *after* the verifier has to be part of the trusted computing base (TCB). In the type-preserving compilation approach, the types (and hence the type checker) for the low-level code tend to be rather more complex, but only the type-checker need be part of the TCB: bugs or maliciousness in the compiler are either benign or yield TAL programs which fail to typecheck; see also work on *proof-carrying code*, such as Necula (1997).

An active area of related work concerns designing C-like languages with safe type systems (and which may be compiled to TAL). Examples include CCured (Necula, McPeak, and Weimer 2002) and Cyclone (Jim, Morrisett, Grossman, Hicks, Cheney, and Wang 2002). These languages vary in their degree of compatibility with legacy C code and in the extent to which safety is ensured by static, rather than dynamic, checks.

Static analysis and optimizing compilation

Many compile-time program analyses which are performed by optimizing compilers have been formulated as non-standard type inference problems. These often take the form of *type and effect* systems, which further classify expressions according to a static approximation of their possible side-effects. A function in such a system will have a type something like

$\tau_1 \xrightarrow{\epsilon} \tau_2$ which is the type of functions which take arguments of type τ_1, return results of type τ_2 and have side-effects at most ϵ. The precise grammar of effects ϵ will be different for different analyses, but might include information about reading, writing or allocating mutable storage or throwing particular exceptions. Type and effect systems are discussed in the Part II Optimizing Compilers course and the book by Nielson, Nielson, and Hankin (1999).

One particularly impressive application of effect inference is *region-based memory management* (Tofte and Talpin 1997). Types for values are extended with an abstraction of the region of memory in which they are allocated, and type and effect inference is used to track which regions are potentially read and written by each expression in the program. This information can then be used by the compiler to soundly insert calls for allocating and freeing regions into the compiled code (the safety condition being that there will be no further accesses to a region after it has been freed). This static approach to memory management can be used instead of, or as well as, the dynamic approach of using a garbage collector. Region-based memory management has been implemented in the ML Kit compiler and in the Cyclone language mentioned above (Grossman, Morrisett, Jim, Hicks, Wang, and Cheney 2002), and sometimes performs dramatically better than garbage collection alone.

Types for schema and query languages

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

```
type Bib = <bib>[ Book* ];;
type Book = <book>[ Title Year Author+ ];;
type Year = <year>[ '0'--'9'+ ];;
type Title = <title>[ PCDATA ];;
type Author = <author>[ PCDATA ];;
```

These declarations say that `Bib` is the type of XML trees consisting of a `<bib>` tag containing a sequence of zero or more `Book`s, each of which is a `<book>` tag containing a `Title`, a `Year` and one or more `Author`s, and so on. An example of an XML document having type `Bib` is

```
<bib>
  <book>
    <title>Persistent Object Systems</title>
    <year>1994</year>
    <author>M. Atkinson</author>
```

```
    <author>V. Benzaken</author>
    <author>D. Maier</author>
  </book>
  <book>
    <title>OOP: a unified foundation</title>
    <year>1997</year>
    <author>G. Castagna</author>
  </book>
</bib>
```

References

Abadi, M. and L. Cardelli (1996). *A Theory of Objects*. Springer-Verlag, New York.

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

Benzaken, V., G. Castagna, and A. Frisch (2003). Cduce: An XML-centric general-purpose language. In *Proceedings of the ACM International Conference on Functional Programming*.

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.

Chaki, S., S. K. Rajamani, and J. Rehof (2002). Types as models: Model checking message-passing programs. In *29th ACM Symposium on Principles of Programming Languages*, pp. 45–57.

Church, A. (1940). A formulation of the simple theory of types. *Journal of Symbolic Logic 5*, 56–68.

Damas, L. and R. Milner (1982). Principal type schemes for functional programs. In *Proc. 9th ACM Symposium on Principles of Programming Lanuages*, pp. 207–212.

Flanagan, C. and M. Abadi (1999). Types for safe locking. In *European Symposium on Programming (ESOP)*.

Gay, S., V. Vasconcelos, and A. Ravara (2003). Session types for interprocess communication. Technical Report TR-2003-133, University of Glasgow.

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. (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.

Grossman, D., G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney (2002, June). Region-based memory management in Cyclone. In *ACM Conference on Programming Language Design and Implementation*.

Harper, R. (1994). A simplified account of polymorphic references. *Information Processing Letters 51*, 201–206.

Harper, R. and C. Stone (1997). An interpretation of Standard ML in type theory. Technical Report CMU–CS–97–147, Carnegie Mellon University, Pittsburgh, PA.

Hennessy, M. and J. Riely (2002). Resource access control in systems of mobile agents. *Information and Computation 173*, 82–120.

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

Jim, T., G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang (2002, June). Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, pp. 275–288.

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

Leroy, X. (2003). Java bytecode verification: Algorithms and formalizations. *Journal of Automated Reasoning*. To appear.

Mairson, H. G. (1990). Deciding ML typability is complete for deterministic exponential time. In *Proc. 17th ACM Symposium on Principles of Programming Languages*, pp. 382–401.

Milner, R., M. Tofte, and R. Harper (1990). *The Definition of Standard ML*. MIT Press.

Milner, R., M. Tofte, R. Harper, and D. MacQueen (1997). *The Definition of Standard ML (Revised)*. MIT Press.

Mitchell, J. C. (1996). *Foundations for Programming Languages*. Foundations of Computing series. MIT Press.

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

Morrisett, G., D. Walker, K. Crary, and N. Glew (1999, May). From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems 21*(3), 528–569.

Necula, G. (1997). Proof-carrying code. In *24th Annual ACM Symposium on Principles of Programming Languages (POPL)*.

Necula, G., S. McPeak, and W. Weimer (2002). CCured: Type-safe retrofitting of legacy code. In *29th Annual ACM Symposium on Principles of Programming Languages (POPL)*.

Nielson, F., H. R. Nielson, and C. Hankin (1999). *Principles of Program Analysis*. Springer.

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

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

Pierce, B. C. and D. N. Turner (1994). Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming 4*, 207–247.

Reynolds, J. C. (1974). Towards a theory of type structure. In *Paris Colloquium on Programming*, Volume 19 of *Lecture Notes in Computer Science*, pp. 408–425. Springer-Verlag, Berlin.

Robinson, J. A. (1965). A machine oriented logic based on the resolution principle. *Jour. ACM 12*, 23–41.

Russell, B. (1903). *The Principles of Mathematics*. Cambridge.

Rydeheard, D. E. and R. M. Burstall (1988). *Computational Category Theory*. Series in Computer Science. Prentice Hall International.

Skalka, C. and S. Smith (2000). Static enforcement of security with types. *ACM SIGPLAN Notices 35*(9), 34–45.

Strachey, C. (1967). Fundamental concepts in programming languages. Lecture notes for the International Summer School in Computer Programming, Copenhagen.

Tofte, M. (1990). Type inference for polymorphic references. *Information and Computation 89*, 1–34.

Tofte, M. and J.-P. Talpin (1997). Region-based memory management. *Information and Computation 132*(2), 109–176.

Volpano, D., G. Smith, and C. Irvine (1996). A sound type system for secure flow analysis. *Journal of Computer Security 4*(3), 167–187.

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

Lectures Appraisal Form

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

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