Lecture Notes on Types for Part II of the Computer Science Tripos

Prof. Andrew M. Pitts
Cambridge University Computer Laboratory

© A. M. Pitts
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).

Aspects of software systems related to types

- **Code reuse**, e.g. via *polymorphism*—the ability of expressions to be used with many different types.
- **Code structuring** via the use of *abstract datatypes* (modules) and typed interfaces between parts of large software systems.
- **Connections with logic**. E.g. the ‘propositions-as-types’ paradigm and connection with typed formal logics used in machine-assisted theorem proving.

Slide 1
Types in programming languages

Used for two related purposes:

- preventing the occurrence of (certain kinds of) errors during program execution;
- structuring data and programs.

Second purpose requires type expressions to occur explicitly in the syntax of programs. First purpose can sometimes be achieved with (part of) a language’s type system implicitly, e.g. occurring as part of the compilation process—cf. the ML family of languages.

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.
1.3 Formalising type systems

Run-time errors

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

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

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

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

\[ \Gamma \vdash M : \tau \]

whose intended meaning is: “given the assignment of types to free identifiers of \( M \) specified by type environment \( \Gamma \), then \( M \) has type \( \tau \).”

E.g.

\[
f : \text{int list} \to \text{int}, b : \text{bool} \vdash (\text{if } b \text{ then } f \text{ nil else } 3) : \text{int}
\]

---

**Slide 5**

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

\[
\tau ::= \text{bool} \quad \text{type of booleans} \\
      | \tau \to \tau \quad \text{function type}
\]

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

\[
M ::= x \quad \text{variable} \\
     | \text{true} | \text{false} \quad \text{boolean values} \\
     | \text{if } M \text{ then } M \text{ else } M \quad \text{conditional} \\
     | \lambda x : \tau(M) \quad \text{function abstraction} \\
     | MM \quad \text{function application.}
\]

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) \[ \Gamma \vdash x : \tau \quad \text{if } (x : \tau) \in \Gamma \]

(bool) \[ \Gamma \vdash B : \text{bool} \quad \text{where } B \in \{\text{true, false}\} \]

(if) \[ \frac{\Gamma \vdash M_1 : \text{bool} \quad \Gamma \vdash M_2 : \tau \quad \Gamma \vdash M_3 : \tau}{\Gamma \vdash \text{if } M_1 \text{ then } M_2 \text{ else } M_3 : \tau} \]

A simple type system, II

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

(app) \[ \frac{\Gamma \vdash M_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash M_2 : \tau_1}{\Gamma \vdash M_1 \ M_2 : \tau_2} \]
1.3 Formalising type systems

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, \( \Gamma \), are finite function from variables to types which we write concretely as comma-separated lists of \((\text{identifier} : \text{type})\)-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) \( \Gamma \), written \( \text{dom}(\Gamma) \). The notation

\[
\Gamma, x : \tau
\]

used on Slide 7 means the typing environment obtained by extending \( \Gamma \) by mapping a variable \( x \) not in \( \text{dom}(\Gamma) \) to type \( \tau \). Thus

\[
\text{dom}(\Gamma, x : \tau) = \text{dom}(\Gamma) \cup \{x\}.
\]

As usual, the axiom and rules on Slides 6 and 7 are schematic: \( \Gamma, M, \tau \) 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 \( \tau_1, \tau_2, \) and \( \tau_3 \)

(1) \( \emptyset \vdash \lambda x_1 : \tau_2 \rightarrow \tau_3(\lambda x_2 : \tau_1 \rightarrow \tau_2(\lambda x_3 : \tau_1(x_1(x_2(x_3))))) : (\tau_2 \rightarrow \tau_3) \rightarrow ((\tau_1 \rightarrow \tau_2) \rightarrow (\tau_1 \rightarrow \tau_3)) \)

is derivable in the type system of Slides 6 and 7. (Why? Give a proof for it.) Whereas for no types \( \tau_1, \tau_2, \) and \( \tau_3 \) is

(2) \( \emptyset \vdash \lambda x : \tau_1 \rightarrow \tau_2(x\ x) : \tau_3 \)

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 $\Gamma$, $M$, and $\tau$, is $\Gamma \vdash M : \tau$ derivable in $TS$?

Typeability problem(s) for $TS$: given $M$ (resp. $\Gamma$ and $M$), find $\Gamma$, $\tau$ (resp. $\tau$) 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 $\Gamma$, $\tau$ (resp. $\tau$) for each $M$ (resp. $\Gamma$, $M$) (or failing, if there are none).

Slide 8

The explicit tagging of $\lambda$-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, true, false, and any closed function abstraction $\lambda x : \tau(M)$. A basic step of computation is $\beta$-reduction

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

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 $\lambda$-bound variables in $M_1$ to avoid capture of free variables in $M_2$). An example of a computation failing is

true $M \rightarrow$ 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).
\[
\frac{M_1 \rightarrow M'_1}{M_1 \cdot M_2 \rightarrow M'_1 \cdot M_2}
\]

\[
\frac{M_1 \rightarrow \text{FAIL}}{M_1 \cdot M_2 \rightarrow \text{FAIL}}
\]

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

\[
\text{true} \cdot M \rightarrow \text{FAIL}
\]

\[
\text{false} \cdot M \rightarrow \text{FAIL}
\]

\[
\frac{M_1 \rightarrow M'_1}{\text{if } M_1 \text{ then } M_2 \text{ else } M_3 \rightarrow \text{if } M'_1 \text{ then } M_2 \text{ else } M_3}
\]

\[
\frac{M_1 \rightarrow \text{FAIL}}{\text{if } M_1 \text{ then } M_2 \text{ else } M_3 \rightarrow \text{FAIL}}
\]

\[
\text{if true then } M_1 \text{ else } M_2 \rightarrow M_1
\]

\[
\text{if false then } M_1 \text{ else } M_2 \rightarrow M_2
\]

\[
\text{if } \lambda x : \tau(M_1) \text{ then } M_2 \text{ else } M_3 \rightarrow \text{FAIL}
\]

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 $\tau$, then $M \rightarrow^* \text{FAIL}$ 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 axiom 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 $\Gamma$ and $M$, there is at most one type $\tau$ for which $\Gamma \vdash M : \tau$ is derivable. Describe a type checking algorithm which when given any $\Gamma$ and $M$ decides whether such a $\tau$ 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:  

$\Gamma \vdash M_1 : \tau_1 \land \Gamma, x : \tau_1 \vdash M_2 : \tau_2 \Rightarrow \Gamma \vdash M_2[M_1/x] : \tau_2.$ 

[Hint: show by induction on the structure of $M_2$ that for all $\Gamma$, $M_1$, $\tau_1$, $x \notin \text{dom}(\Gamma)$ and $\tau_2$ 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 $M \rightarrow \text{FAIL}$, 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).]