12 lectures for CST Part II by Andrew Pitts

<www.cl.cam.ac.uk/teaching/1516/Types/>
“One of the most helpful concepts in the whole of programming is the notion of type, used to classify the kinds of object which are manipulated. A significant proportion of programming mistakes are detected by an implementation which does type-checking before it runs any program. Types provide a taxonomy which helps people to think and to communicate about programs.”


“The fact that companies such as Microsoft, Google and Mozilla are investing heavily in systems programming languages with stronger type systems is not accidental – it is the result of decades of experience building and deploying complex systems written in languages with weak type systems.”

Uses of type systems

- Detecting errors via *type-checking*, either **statically** (decidable errors detected before programs are executed) or **dynamically** (typing errors detected during program execution).

\[
\text{static} = \text{compile-time} = \text{decidable} \\
\text{dynamic} = \text{run-time} = \text{possibly undecidable}
\]
Uses of type systems

- Detecting errors via *type-checking*, either statically (decidable errors detected before programs are executed) or dynamically (typing errors detected during program execution).

- Abstraction and support for structuring large systems.

  eg. types in \{ module interfaces object classes \}
Uses of type systems

- Detecting errors via *type-checking*, either statically (decidable errors detected before programs are executed) or dynamically (typing errors detected during program execution).

- Abstraction and support for structuring large systems.

- Documentation.

*type systems as checkable documentation of programmer intentions*
Uses of type systems

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

\textit{goes back to FORTRAN!}
Uses of type systems

- Detecting errors via *type-checking*, either statically (decidable errors detected before programs are executed) or dynamically (typing errors detected during program execution).

- Abstraction and support for structuring large systems.

- Documentation.

- Efficiency.

- Whole-language safety.

PL "meta-theory" - properties of all legal progs
E.g. §4 of this course

Requires formal math/logic methods
Formal type systems

part of PL semantics

- Constitute the precise, mathematical characterisation of informal type systems (such as occur in the manuals of most typed languages.)
Formal type systems

- Constitute the precise, mathematical characterisation of informal type systems (such as occur in the manuals of most typed languages.)

- Basis for *type soundness* theorems: “any well-typed program cannot produce run-time errors (of some specified kind).”
Formal type systems

- Constitute the precise, mathematical characterisation of informal type systems (such as occur in the manuals of most typed languages.)

- Basis for type soundness theorems: “any well-typed program cannot produce run-time errors (of some specified kind).”

- Can decouple specification of typing aspects of a language from algorithmic concerns: the formal type system can define typing independently of particular implementations of type-checking algorithms.
Typical type system judgement

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

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

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

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

\[ \Gamma \vdash e : \tau \]

and read as: *given the assignment of types to free identifiers of* \(e\) *specified by type environment* \(\Gamma\), *then* \(e\) *has type* \(\tau\).

E.g.

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

is a valid typing judgement about ML.
Typical type system judgement

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

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

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

E.g.

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

is a valid typing judgement about ML.

We consider structural type systems, in which there is a language of type expressions built up using type constructs (e.g. $\text{int list} \rightarrow \text{int}$ in ML).

(By contrast, in nominal type systems, type expressions are just unstructured names.)
Notations for the typing relation

‘foo has type bar’
Notations for the typing relation

‘foo has type bar’

ML-style (used in this course):

foo : bar
Notations for the typing relation

‘foo has type bar’

ML-style (used in this course):

foo : bar

Haskell-style:

foo :: bar
Notations for the typing relation

‘foo has type bar’

ML-style (used in this course):

foo : bar

Haskell-style:

foo :: bar

C/Java-style:

bar foo
Type checking, typeability, and type inference

Suppose given a type system for a programming language with judgements of the form $\Gamma \vdash e : \tau$. 
Type checking, typeability, and type inference

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

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

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

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

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

Solving the second problem usually involves devising a type inference algorithm computing a $\tau$ for each $\Gamma$ and $e$ (or failing, if there is none).
Progress, type preservation & safety

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

Progress.

\[ G \xrightarrow{e} t \text{ and } \operatorname{dom}(G) \vdash \operatorname{dom}(s), \text{ then either } e \text{ is a value, or there exist } e_0, s_0 \text{ such that } \]

\[ G \xrightarrow{e_0} t \text{ and } \operatorname{dom}(G) \vdash \operatorname{dom}(s_0). \]

Type preservation.

\[ G \xrightarrow{e} t \text{ and } \operatorname{dom}(G) \vdash \operatorname{dom}(s) \text{ and } \]

\[ G \xrightarrow{e_0} t \text{ and } \operatorname{dom}(G) \vdash \operatorname{dom}(s_0). \]

Hence well-typed programs don't get stuck:

Safety.

\[ G \xrightarrow{e} t \text{, } \operatorname{dom}(G) \vdash \operatorname{dom}(s) \text{ and } \]

\[ e \xrightarrow{\cdots} e_0, s_0 \text{, then either } e_0 \text{ is a value, or there exist } e_0, s_0 \text{ such that } \]

\[ e_0 \xrightarrow{\cdots} e_0, s_0. \]
Progress, type preservation & safety

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

**Progress.** If $\Gamma \vdash e : \tau$ and $\text{dom}(\Gamma) \subseteq \text{dom}(s)$, then either $e$ is a value, or there exist $e', s'$ such that $\langle e, s \rangle \rightarrow \langle e', s' \rangle$. 
Progress, type preservation & safety

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

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

**Type preservation.** If $\Gamma \vdash e : \tau$ and $\text{dom}(\Gamma) \subseteq \text{dom}(s)$ and $\langle e, s \rangle \rightarrow \langle e', s' \rangle$, then $\Gamma \vdash e' : \tau$ and $\text{dom}(\Gamma) \subseteq \text{dom}(s')$. 
Progress, type preservation & safety

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

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

**Type preservation.** If $\Gamma \vdash e : \tau$ and $\text{dom}(\Gamma) \subseteq \text{dom}(s)$ and $\langle e, s \rangle \rightarrow \langle e', s' \rangle$, then $\Gamma \vdash e' : \tau$ and $\text{dom}(\Gamma) \subseteq \text{dom}(s')$.

Hence well-typed programs don’t get stuck:

**Safety.** If $\Gamma \vdash e : \tau$, $\text{dom}(\Gamma) \subseteq \text{dom}(s)$ and $\langle e, s \rangle \rightarrow^* \langle e', s' \rangle$, then either $e'$ is a value, or there exist $e'', s''$ such that $\langle e', s' \rangle \rightarrow \langle e'', s'' \rangle$. 
Outline of the rest of the course

- **ML polymorphism.** Principal type schemes and type inference. [2]

- **Polymorphic reference types.** The pitfalls of combining ML polymorphism with reference types. [1]

- **Polymorphic lambda calculus (PLC).** Explicit versus implicitly typed languages. PLC syntax and reduction semantics. Examples of datatypes definable in the polymorphic lambda calculus. [3]

- **Dependent types.** Dependent function types. Pure type systems. System F-omega. [2]

- **Propositions as types.** Example of a non-constructive proof. The Curry-Howard correspondence between intuitionistic second-order propositional calculus and PLC. The calculus of Constructions. Inductive types. [3]
Polymorphism = has many types
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.)
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**: subtyping relation \( \tau_1 <: \tau_2 \) allows any \( M_1 : \tau_1 \) to be used as \( M_1 : \tau_2 \) without violating safety.
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**: subtyping relation \( \tau_1 <: \tau_2 \) allows any \( M_1 : \tau_1 \) to be used as \( M_1 : \tau_2 \) without violating safety.

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

  \[
  \text{fun length nil = 0} \\
  | \quad \text{length (x :: xs) = 1 + (length xs)}
  \]

  has type \( \tau \text{ list} \to \text{int} \) for all types \( \tau \).
Type variables and type schemes in Mini-ML

To formalise statements like

```
“length has type \( \tau \text{ list} \rightarrow \text{int} \), for all types \( \tau \)”
```
Type variables and type schemes in Mini-ML

To formalise statements like

"\textit{length} has type $\tau \text{ list} \rightarrow \text{int}$, for all types $\tau$"

we introduce \textit{type variables} $\alpha$ (i.e. variables for which types may be substituted) and write

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

$\forall \alpha \ (\alpha \text{ list} \rightarrow \text{int})$ is an example of a \textit{type scheme}. 
Polymorphism of \textbf{let}-bound variables in ML

For example in

\[
\text{let } \, f = \lambda x. (x) \text{ in } (f \, \text{true}) :: (f \, \text{nil})
\]
Polymorphism of `let`-bound variables in ML

For example in

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

\(\lambda x (x)\) has type \(\tau \rightarrow \tau\) for any type \(\tau\), and the variable \(f\) to which it is bound is used polymorphically:
Polymorphism of \textbf{let}-bound variables in ML

For example in

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

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

in \((f \text{ true})\), \(f\) has type \textit{bool} \rightarrow \textit{bool}
Polymorphism of \texttt{let}-bound variables in ML

For example in

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

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

- in $(f \,\text{true})$, $f$ has type $\texttt{bool} \rightarrow \texttt{bool}$
- in $(f \,\text{nil})$, $f$ has type $\texttt{bool list} \rightarrow \texttt{bool list}$
Polymorphism of \textbf{let}-bound variables in ML

For example in

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

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

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

Overall, the expression has type \(\text{bool list}\).
Forms of hypothesis in typing judgements

- *Ad hoc* (overloading):
  
  \[
  \text{if } f : \text{bool} \rightarrow \text{bool} \\
  \text{and } f : \text{bool list} \rightarrow \text{bool list}, \\
  \text{then } (f \text{ true}) :: (f \text{ nil}) : \text{bool list}.
  \]

  Appropriate for expressions that have different behaviour at different types.

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

- **Ad hoc (overloading):**
  
  if \( f : \texttt{bool} \rightarrow \texttt{bool} \)
  
  and \( f : \texttt{bool list} \rightarrow \texttt{bool list} \),
  
  then \((f \texttt{true}) :: (f \texttt{nil}) : \texttt{bool list}\).

  Appropriate for expressions that have different behaviour at different types.

- **Parametric:**

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

  Appropriate if expression behaviour is uniform for different type instantiations.

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

takes the form

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

where
Mini-ML typing judgement

takes the form

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

where

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

takes the form

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

where

- the **typing environment** \( \Gamma \) is a finite function from variables to **type schemes**.
  (We write \( \Gamma = \{ x_1 : \sigma_1, \ldots, x_n : \sigma_n \} \) to indicate that \( \Gamma \) has domain of definition \( \text{dom}(\Gamma) = \{ x_1, \ldots, x_n \} \) (mutually distinct variables) and maps each \( x_i \) to the type scheme \( \sigma_i \) for \( i = 1 \ldots n \).)

- \( M \) is a Mini-ML expression

- \( \tau \) is a Mini-ML type.
Types

\[ \tau ::= \alpha \quad \text{type variable} \]
\[ | \quad \text{bool} \quad \text{type of booleans} \]
\[ | \quad \tau \rightarrow \tau \quad \text{function type} \]
\[ | \quad \tau \text{list} \quad \text{list type} \]
Mini-ML types and type schemes

Types

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

where \( \alpha \) ranges over a fixed, countably infinite set \( \text{TyVar} \).
Mini-ML types and type schemes

**Types**

\[ \tau ::= \alpha \quad \text{type variable} \]

| \text{bool} \quad \text{type of booleans} |
| \tau \rightarrow \tau \quad \text{function type} |
| \tau \text{ list} \quad \text{list type} |

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

**Type Schemes**

\[ \sigma ::= \forall A (\tau) \]

where \( A \) ranges over finite subsets of the set \( \text{TyVar} \).
Mini-ML types and type schemes

**Types**

\[ \tau ::= \alpha \quad \text{type variable} \]
\[ \mid \text{bool} \quad \text{type of booleans} \]
\[ \mid \tau \to \tau \quad \text{function type} \]
\[ \mid \tau \text{list} \quad \text{list type} \]

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

**Type Schemes**

\[ \sigma ::= \forall A \, (\tau) \]

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

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

\[ \forall \alpha_1, \ldots, \alpha_n \, (\tau). \]
Mini-ML types and type schemes

**Types**

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

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

**Type Schemes**

\[
\sigma ::= \forall A \left( \tau \right)
\]

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

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

\[
\forall \alpha_1, \ldots, \alpha_n \left( \tau \right).
\]

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