Types

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."

R. Milner, *Computing Tomorrow* (CUP, 1996), p264

"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."

T. Ball and B. Zorn, *Teach Foundational Language Principles*, Viewpoints, Comm. ACM (2014) 58(5) 30–31

Type systems channel TCS into PLs & Verification

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).

static = compile-time = decidable

dynamic = run-time = 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 $\begin{cases} \text{module interfaces} \\ \text{object classes} \end{cases}$

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.

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 (Γ), program phrases (e) and type expressions (τ) that we write as

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

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

E.g.

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

is a valid typing judgement about ML.

Typical type system judgement

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

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

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

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

is a valid typing judgement about ML.

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

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):

$$\texttt{foo} : \texttt{bar}$$

Haskell-style:

$$\texttt{foo} :: \texttt{bar}$$

Notations for the typing relation

'`foo` has type `bar`'

ML-style (used in this course):

$$\texttt{foo} : \texttt{bar}$$

Haskell-style:

$$\texttt{foo} :: \texttt{bar}$$

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 Γ, e, and τ, 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 Γ, e, and τ, is $\Gamma \vdash e : \tau$ derivable in the type system?

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

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

Progress, type preservation & safety

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

Progress, type preservation & safety

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

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

Progress, type preservation & safety

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

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

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

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

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

Hence well-typed programs don't get stuck:

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

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]

new material this year

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

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

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

Type variables and type schemes in Mini-ML

To formalise statements like

"$\boldsymbol{length}$ has type $\boldsymbol{\tau}\ \boldsymbol{list} \to \boldsymbol{int}$, for all types $\boldsymbol{\tau}$"

Type variables and type schemes in Mini-ML

To formalise statements like

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

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

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

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

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

For example in

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

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

For example in

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

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

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

For example in

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

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

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

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

For example in

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

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

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

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

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

For example in

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

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

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

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

Forms of hypothesis in typing judgements

- *Ad hoc* (overloading):

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

  Appropriate for expressions that have different behaviour at different types.

Forms of hypothesis in typing judgements

- *Ad hoc* (overloading):

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

  Appropriate for expressions that have different behaviour at different types.

- *Parametric*:

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

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

Mini-ML typing judgement

takes the form

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

where

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

- M is a Mini-ML expression

- τ is a Mini-ML type.

Mini-ML types and type schemes

Types

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

Mini-ML types and type schemes

Types

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

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

Mini-ML types and type schemes

Types

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

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

Type Schemes

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

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

Mini-ML types and type schemes

Types

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

where α ranges over a fixed, countably infinite set $\mathbf{TyVar}$.

Type Schemes

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

where A ranges over finite subsets of the set $\mathbf{TyVar}$.

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

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

where α ranges over a fixed, countably infinite set $\mathbf{TyVar}$.

Type Schemes

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

where A ranges over finite subsets of the set $\mathbf{TyVar}$.

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

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

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