Types

8 lectures for CST Part II by Andrew Pitts

⟨www.cl.cam.ac.uk/teaching/1314/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

- lecture notes
- web page
- further reading

The full title of this course is

Type Systems for Programming Languages

What are ‘type systems’ and what are they good for?

> ‘A type system is a tractable syntactic method for proving the absence of certain program behaviours by classifying phrases according to the kinds of values they compute’
>
> B. Pierce, ‘Types and Programming Languages’ (MIT, 2002), p1

Type systems are one of the most important channels by which developments in theoretical computer science get applied in programming language design and software verification.

& logic

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.

Safety

Informal definitions from the literature.

‘A safe language is one that protects its own high-level abstractions [no matter what legal program we write in it]’.

‘A safe language is completely defined by its programmer’s manual [rather than which compiler we are using]’.

‘A safe language may have *trapped* errors [one that can be handled gracefully], but can’t have *untrapped errors* [ones that cause unpredictable crashes]’.

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 (M) and type expressions (τ) that we write as

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

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

E.g.

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

is a valid typing judgement about ML.

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 $\Gamma \vdash M : \tau$.

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

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

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

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 $\tau_1 <: \tau_2$: any $M_1 : \tau_1$ can 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 SML, length function

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

has type $\tau\ \mathit{list} \longrightarrow \mathit{int}$ for all types τ.)

Type variables and type schemes in Mini-ML

To formalise statements like

‘ $length$ has type $\tau\ list \longrightarrow int$, for all types τ’

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

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

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

Polymorphism of $\mathtt{let}$-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 $\tau \longrightarrow \tau$ for any type τ, and the variable f to which it is bound is used polymorphically:

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

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

‘Ad hoc’ polymorphism:

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

‘Parametric’ polymorphism:

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

Mini-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 Mini-ML expression
- τ is an Mini-ML type.