Types

8 lectures for CST Part II by Andrew Pitts

⟨www.cl.cam.ac.uk/teaching/1112/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 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 verifiction.

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.

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

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

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

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

Type variables and type schemes in Mini-ML

To formalise statements like

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

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

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

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

Polymorphism of let-bound variables in ML

For example in

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

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

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

'Parametric' polymorphism:

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

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 $A = \{\alpha_1, \ldots, \alpha_n\}$, we write $\forall\, A\,(\tau)$ as

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

The ‘generalises’ relation between type schemes and types

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 $\sigma \succ \tau$.

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.

Mini-ML expressions, M

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

Mini-ML type system, I

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

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

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

Mini-ML type system, II

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

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

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

Mini-ML type system, III

(fn)

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

(app)

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

Mini-ML type system, IV

(let)

$$\frac{\Gamma \vdash M_1 : \tau \qquad \Gamma, x : \forall A\,(\tau) \vdash M_2 : \tau'}{\Gamma \vdash \mathtt{let}\, x = M_1 \,\mathtt{in}\, M_2 : \tau'}$$

if $x \notin dom(\Gamma)$ and $A = ftv(\tau) - ftv(\Gamma)$

Assigning type schemes to Mini-ML expressions

Given a type scheme $\sigma = \forall A\,(\tau)$, write

$$\boxed{\Gamma \vdash M : \sigma}$$

if $A = ftv(\tau) - ftv(\Gamma)$ and $\Gamma \vdash M : \tau$ is derivable from the axiom and rules on Slides 16–19.

When $\Gamma = \{\,\}$ we just write $\boxed{\vdash M : \sigma}$ for $\{\,\} \vdash M : \sigma$ and say that the (necessarily closed—see Exercise 2.5.2) expression M is *typeable* in Mini-ML with type scheme σ.

Two examples involving self-application

$$M \stackrel{\text{def}}{=} \texttt{let}\ f = \lambda x_1(\lambda x_2(x_1))\ \texttt{in}\ f\ f$$

$$M' \stackrel{\text{def}}{=} (\lambda f(f\ f))\ \lambda x_1(\lambda x_2(x_1))$$

Are M and M' typeable in the Mini-ML type system?

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

(C0) $A = ftv(\tau_2)$

(C1) $\tau_2 = \tau_3 \rightarrow \tau_4$

(C2) $\tau_4 = \tau_5 \rightarrow \tau_6$

(C3) $\forall\,\{\,\}\,(\tau_3) \succ \tau_6$, i.e. $\tau_3 = \tau_6$

(C4) $\tau_7 = \tau_8 \rightarrow \tau_1$

(C5) $\forall\, A\,(\tau_2) \succ \tau_7$

(C6) $\forall\, A\,(\tau_2) \succ \tau_8$

Principal type schemes for closed expressions

A closed type scheme $\forall\, A\, (\tau)$ is the *principal* type scheme of a closed Mini-ML expression M if

(a) $\vdash M : \forall\, A\, (\tau)$

(b) for any other closed type scheme $\forall\, A'\, (\tau')$,
if $\vdash M : \forall\, A'\, (\tau')$, then $\forall\, A\, (\tau) \succ \tau'$

Theorem (Hindley; Damas-Milner)

If the closed Mini-ML expression M is typeable (i.e. $\vdash M : \sigma$ 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.

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

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

(Taken from Mairson 1990.)

Unification of ML types

There is an algorithm mgu which when input two Mini-ML types τ_1 and τ_2 decides whether τ_1 and τ_2 are *unifiable*, i.e. whether there exists a type-substitution $S \in \mathbf{Sub}$ 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) for all $S' \in \mathbf{Sub}$, if $S'(\tau_1) = S'(\tau_2)$, then $S' = TS$ for some $T \in \mathbf{Sub}$.

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

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

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

(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 $T \in \mathbf{Sub}$ with $T\,S = S'$ and $T(\sigma) \succ \sigma'$.

[For type schemes σ and σ', with $\sigma' = \forall\, A'\,(\tau')$ say, we define $\boxed{\sigma \succ \sigma'}$ to mean $A' \cap ftv(\sigma) = \{\}$ and $\sigma \succ \tau'$.]

Properties of the Mini-ML typing relation

- If $\Gamma \vdash M : \sigma$, then for any type substitution $S \in \mathbf{Sub}$ $S\Gamma \vdash M : S\sigma$.
- If $\Gamma \vdash M : \sigma$ and $\sigma \succ \sigma'$, then $\Gamma \vdash M : \sigma'$.

Specification for the principal typing algorithm, pt

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

- If $\Gamma \vdash M : ?$ has a solution (cf. Slide 27), 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 $FAIL$.

Some of the clauses in a definition of pt

Function abstractions: $pt(\Gamma \vdash \lambda x(M) : ?) \stackrel{\text{def}}{=}$

$$\text{let } \alpha = \text{fresh in}$$
$$\text{let } (S, \tau) = pt(\Gamma, x : \alpha \vdash M : ?) \text{ in } (S, S(\alpha) \to \tau)$$

Function applications: $pt(\Gamma \vdash M_1\, M_2 : ?) \stackrel{\text{def}}{=}$

$$\text{let } (S_1, \tau_1) = pt(\Gamma \vdash M_1 : ?) \text{ in}$$
$$\text{let } (S_2, \tau_2) = pt(S_1\,\Gamma \vdash M_2 : ?) \text{ in}$$
$$\text{let } \alpha = \text{fresh in}$$
$$\text{let } S_3 = mgu(S_2\,\tau_1, \tau_2 \to \alpha) \text{ in } (S_3 S_2 S_1, S_3(\alpha))$$

ML types and expressions for mutable references

τ	$::=$	$\ldots$	
	$\mid$	$unit$	unit type
	$\mid$	$\tau\ ref$	reference type.
M	$::=$	$\ldots$	
	$\mid$	$()$	unit value
	$\mid$	$\mathtt{ref}\ M$	reference creation
	$\mid$	$!M$	dereference
	$\mid$	$M := M$	assignment

Midi-ML's extra typing rules

(unit) $$\Gamma \vdash () : unit$$

(ref) $$\frac{\Gamma \vdash M : \tau}{\Gamma \vdash \mathtt{ref}\ M : \tau\ ref}$$

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

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

Example 3.1.1

The expression

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

has type $unit$.

Midi-ML transitions involving references

$$\langle !x, s\rangle \to \langle s(x), s\rangle \quad \text{if } x \in dom(s)$$

$$\langle !V, s\rangle \to FAIL \quad \text{if } V \text{ not a variable}$$

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

$$\langle V := V', s\rangle \to FAIL \quad \text{if } V \text{ not a variable}$$

$$\langle \mathtt{ref}\, V, s\rangle \to \langle x, s[x \mapsto V]\rangle \quad \text{if } x \notin dom(s)$$

where V ranges over *values*:

$$V ::= x \mid \lambda x(M) \mid () \mid \mathtt{true} \mid \mathtt{false} \mid \mathtt{nil} \mid V :: V$$

Value-restricted typing rule for `let`-expressions

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

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

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

(Recall that values are given by

$V ::= x \mid \lambda x(M) \mid () \mid \mathtt{true} \mid \mathtt{false} \mid \mathtt{nil} \mid V :: V$.)

Type soundness for Midi-ML with the value restriction

For any closed Midi-ML expression M, if there is some type scheme σ for which

$$\vdash M : \sigma$$

is provable in the value-restricted type system (axioms and rules on Slides 16–18, 32 and 35), then *evaluation of M does not fail*, i.e. there is no sequence of transitions of the form

$$\langle M, \{\,\} \rangle \to \cdots \to \mathit{FAIL}$$

for the transition system $\longrightarrow$ defined in Figure 4 (where $\{\,\}$ denotes the empty state).

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

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

Syntactically, because in rule

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

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

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

Monomorphic types . . .

$$\tau ::= \alpha \mid bool \mid \tau \to \tau \mid \tau\, list$$

. . . and *type schemes*

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

Polymorphic types

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

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.

Identity, Generalisation and Specialisation

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

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

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

Fact (see Wells 1994):

For the modified ML type system with polymorphic types and $(\mathbf{var} \succ)$ replaced by the axiom and rules on Slide 39, *the type checking and typeability problems* (cf. Slide 7) *are equivalent and undecidable.*

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 4.1.1)

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

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

PLC syntax

Types

$$\begin{array}{rcll} \tau & ::= & \alpha & \text{type variable} \\ & | & \tau \rightarrow \tau & \text{function type} \\ & | & \forall\, \alpha\, (\tau) & \forall\text{-type} \end{array}$$

Expressions

$$\begin{array}{rcll} M & ::= & x & \text{variable} \\ & | & \lambda\, x : \tau\, (M) & \text{function abstraction} \\ & | & M\, M & \text{function application} \\ & | & \Lambda\, \alpha\, (M) & \text{type generalisation} \\ & | & M\, \tau & \text{type specialisation} \end{array}$$

(α and x range over fixed, countably infinite sets $\mathbf{TyVar}$ and $\mathbf{Var}$ respectively.)

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

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

as well as the usual form of beta-reduction from λ-calculus

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

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.

PLC type system

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

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

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

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

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

An incorrect ‘proof’

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

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.)

PLC type-checking algorithm, I

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:

$$typ(\Gamma \vdash M_1\, M_2 : ?) \stackrel{\text{def}}{=}$$
$$\text{let } \tau_1 = typ(\Gamma \vdash M_1 : ?) \text{ in}$$
$$\text{let } \tau_2 = typ(\Gamma \vdash M_2 : ?) \text{ in}$$
$$\begin{array}{llcl} \text{case } \tau_1 \text{ of} & \tau \to \tau' & \mapsto & \text{if } \tau = \tau_2 \text{ then } \tau' \text{ else } \mathit{FAIL} \\ \mid & \_ & \mapsto & \mathit{FAIL} \end{array}$$

PLC type-checking algorithm, II

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:

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

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.

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 $M \to M'$, then $\Gamma \vdash M' : \tau$ is also a provable typing.

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

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

PLC beta-conversion, $=_\beta$

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

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

where each $-$ is either $\rightarrow$ 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

$$M \rightarrow^* N \ {}^*\!\!\leftarrow M'$$

Polymorphic booleans

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

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

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

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

Polymorphic lists

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

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

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

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:

$$
\begin{aligned}
g\,Nil &= x' \\
g\,(x :: \ell) &= f\,x\,(g\,\ell).
\end{aligned}
$$

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

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:

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

A tautology checker

$$
\begin{aligned}
\texttt{fun}\ \mathit{taut}\ n\ f = \ & \texttt{if}\ n = 0\ \texttt{then}\ f\ \texttt{else} \\
& (\mathit{taut}(n-1)(f\ \texttt{true})) \\
& \texttt{andalso}\ (\mathit{taut}(n-1)(f\ \texttt{false}))
\end{aligned}
$$

Defining types

$$
\begin{cases}
0\ \mathit{AryBoolOp} & \stackrel{\text{def}}{=}\ \mathit{bool} \\
(n+1)\ \mathit{AryBoolOp} & \stackrel{\text{def}}{=}\ \mathit{bool} \longrightarrow (n\ \mathit{AryBoolOp})
\end{cases}
$$

then $\mathit{taut}\ n$ has type $(n\ \mathit{AryBoolOp}) \longrightarrow \mathit{bool}$, i.e. the result type of the function taut *depends upon the value of its argument*.

The tautology checker in Agda

```
data Bool : Set where
  True : Bool
  False : Bool

_and_ : Bool -> Bool -> Bool
True and True = True
True and False = False
False and _ = False

data Nat : Set where
  Zero : Nat
  Succ : Nat -> Nat
```

```
_AryBoolOp : Nat -> Set
Zero AryBoolOp = Bool
(Succ n) AryBoolOp = Bool -> n AryBoolOp

taut : (n : Nat) -> n AryBoolOp -> Bool
taut Zero f = f
taut (Succ n) f = taut n (f True) and taut n (f False)
```

Dependent function types $(x : \tau) \to \tau'$

$$\frac{\Gamma, x : \tau \vdash M : \tau'}{\Gamma \vdash \lambda\, x : \tau\, (M) : (x : \tau) \to \tau'} \quad \text{if } x \notin dom(\Gamma) \cup fv(\Gamma)$$

$$\frac{\Gamma \vdash M : (x : \tau) \to \tau' \quad \Gamma \vdash M' : \tau}{\Gamma \vdash M\, M' : \tau'[M'/x]}$$

τ' may ‘depend’ on x, i.e. have free occurrences of x.

(Free occurrences of x in τ' are bound in $(x : \tau) \to \tau'$.)

Curry-Howard correspondence

Logic	$\longleftrightarrow$	*Type system*
propositions, ϕ	$\longleftrightarrow$	types, τ
(constructive) proofs, p	$\longleftrightarrow$	expressions, M
'p is a proof of ϕ'	$\longleftrightarrow$	'M is an expression of type τ'
simplification of proofs	$\longleftrightarrow$	reduction of expressions

Second-order intuitionistic propositional calculus (2IPC)

2IPC propositions: $\boxed{\phi ::= p \mid \phi \to \phi \mid \forall p\,(\phi)}$, where p ranges over an infinite set of propositional variables.

2IPC sequents: $\boxed{\Phi \vdash \phi}$, where Φ is a finite set of 2IPC propositions and ϕ is a 2IPC proposition.

$\Phi \vdash \phi$ is *provable* if it is in the set of sequents inductively generated by:

$$\text{(Id)}\quad \Phi \vdash \phi \quad \text{if } \phi \in \Phi$$

$$(\longrightarrow\text{I})\ \frac{\Phi, \phi \vdash \phi'}{\Phi \vdash \phi \to \phi'} \qquad (\longrightarrow\text{E})\ \frac{\Phi \vdash \phi \to \phi' \quad \Phi \vdash \phi}{\Phi \vdash \phi'}$$

$$(\forall\text{I})\ \frac{\Phi \vdash \phi}{\Phi \vdash \forall p\,(\phi)} \text{ if } p \notin fv(\Phi) \qquad (\forall\text{E})\ \frac{\Phi \vdash \forall p\,(\phi)}{\Phi \vdash \phi[\phi'/p]}$$

A 2IPC proof

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{p \,\&\, q, p, q\} \vdash p}\,(Id)}{\{p \,\&\, q, p\} \vdash q \to p}\,(\to I)}{\{p \,\&\, q\} \vdash p \to q \to p}\,(\to I) \qquad \dfrac{\dfrac{}{\{p \,\&\, q\} \vdash \forall\, r\,((p \to q \to r) \to r)}\,(Id)}{\{p \,\&\, q\} \vdash (p \to q \to p) \to p}\,(\forall E)}{\{p \,\&\, q\} \vdash p}\,(\to E)}{\{\,\} \vdash p \,\&\, q \to p}\,(\to I)}{\{\,\} \vdash \forall\, q\,(p \,\&\, q \to p)}\,(\forall I)}{\{\,\} \vdash \forall\, p, q\,(p \,\&\, q \to p)}\,(\forall I)
$$

where $p \,\&\, q$ is an abbreviation for $\forall\, r\,((p \to q \to r) \to r)$.

The PLC expression corresponding to this proof is:

$$\Lambda\, p, q\,(\lambda\, z : p \,\&\, q\,(z\, p\,(\lambda\, x : p, y : q\,(x)))).$$

Type-inference versus proof search

Type-inference: ‘given Γ and M, is there a type σ such that $\Gamma \vdash M : \sigma$?’

(For PLC/2IPC this is decidable.)

Proof-search: ‘given Γ and σ, is there a proof term M such that $\Gamma \vdash M : \sigma$?’

(For PLC/2IPC this is undecidable.)

2IPC is a constructive logic

For example, there is no proof of the *Law of Excluded Middle*

$$\forall\, p\, (p \vee \neg p)$$

Using the definitions on Slide 65, this is an abbreviation for

$$\forall\, p, q\, ((p \rightarrow q) \rightarrow ((p \rightarrow \forall\, r\, (r)) \rightarrow q) \rightarrow q)$$

(The fact that there is no closed PLC term of type $\forall\, p\, (p \vee \neg p)$ can be proved using the technique developed in the Tripos question 13 on paper 9 in 2000.)

Logical operations definable in 2IPC

- *Truth*: $\boldsymbol{true} \stackrel{\text{def}}{=} \forall\, p\, (p \to p)$.
- *Falsity*: $\boldsymbol{false} \stackrel{\text{def}}{=} \forall\, p\, (p)$.
- *Conjunction*: $\phi \,\&\, \phi' \stackrel{\text{def}}{=} \forall\, p\, ((\phi \to \phi' \to p) \to p)$
  (where $p \notin fv(\phi, \phi')$).
- *Disjunction*: $\phi \vee \phi' \stackrel{\text{def}}{=} \forall\, p\, ((\phi \to p) \to (\phi' \to p) \to p)$
  (where $p \notin fv(\phi, \phi')$).
- *Negation*: $\neg\phi \stackrel{\text{def}}{=} \phi \to \boldsymbol{false}$.
- *Existential quantification*:
  $\exists\, p\, (\phi) \stackrel{\text{def}}{=} \forall\, p'\, (\forall\, p\, (\phi \to p') \to p')$
  (where $p' \notin fv(\phi, p)$).

Example of a non-constructive proof

Theorem. There exist two irrational numbers a and b such that b^a is rational.

Proof. Either $\sqrt{2}^{\sqrt{2}}$ is rational, or it is not (LEM!).

If it is, we can take $a = b = \sqrt{2}$, since $\sqrt{2}$ is irrational by a well-known theorem attributed to Euclid.

If it is not, we can take $a = \sqrt{2}$ and $b = \sqrt{2}^{\sqrt{2}}$, since then
$b^a = (\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = \sqrt{2}^{\sqrt{2}\times\sqrt{2}} = \sqrt{2}^2 = 2$.

QED