Types

12 lectures for CST Part II by Neel Krishnaswami

⟨www.cl.cam.ac.uk/teaching/1718/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

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.

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.
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`'

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** 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. 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]

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

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

  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 $\tau\ list \to int$, for all types τ”

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

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

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

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

  Appropriate for expressions that have different behaviour at different types.

- *Parametric*:

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

  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

- 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

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

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

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 $\forall\{\,\}\,(\tau)$.**

Specialising type schemes to types

A type τ is a **specialisation** of a type scheme $\sigma = \forall \alpha_1, \ldots, \alpha_n\,(\tau')$ if τ can be obtained from the type τ' by simultaneously substituting some types τ_i for the type variables α_i $(i = 1, \ldots, n)$:

$$\tau = \tau'[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$$

In this case we write $\boxed{\sigma \succ \tau}$

(N.B. The relation is unaffected by the particular choice of names of bound type variables in σ.)

The converse relation is called **generalisation**: a type scheme σ generalises a type τ if $\sigma \succ \tau$.

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

$$(\mathbf{var} \succ)\ \frac{}{\Gamma \vdash x : \tau} \text{ if } (x : \sigma) \in \Gamma \text{ and } \sigma \succ \tau$$

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

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

Mini-ML type system, II

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

$$(\textbf{cons})\ \frac{\Gamma \vdash M : \tau \qquad \Gamma \vdash L : \tau\ \mathit{list}}{\Gamma \vdash M :: L : \tau\ \mathit{list}}$$

$$(\textbf{case})\ \frac{\begin{array}{c}\Gamma \vdash L : \tau\ \mathit{list} \qquad \Gamma \vdash N : \tau' \\ \Gamma, x : \tau, \ell : \tau\ \mathit{list} \vdash C : \tau'\end{array}}{\Gamma \vdash (\texttt{case}\ L\ \texttt{of}\ \texttt{nil} \Rightarrow N \mid x :: \ell \Rightarrow C) : \tau'}\ \text{if } x \neq \ell \text{ and } x, \ell \notin \mathit{dom}(\Gamma)$$

Mini-ML type system, III

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

$$(\mathbf{app})\ \frac{\Gamma \vdash M : \tau_1 \to \tau_2 \qquad \Gamma \vdash N : \tau_1}{\Gamma \vdash M\,N : \tau_2}$$

$$(\mathbf{let})\ \frac{\begin{array}{c}\Gamma \vdash M_1 : \tau \\ \Gamma, x : \forall A\,(\tau) \vdash M_2 : \tau'\end{array}}{\Gamma \vdash (\mathtt{let}\, x = M_1 \,\mathtt{in}\, M_2) : \tau'} \text{ if } x \notin dom(\Gamma) \text{ and } A = ftv(\tau) - ftv(\Gamma)$$

Definition. We write $\boxed{\Gamma \vdash M : \forall A\,(\tau)}$ to mean $\Gamma \vdash M : \tau$ is derivable from the Mini-ML typing rules and that $A = ftv(\tau) - ftv(\Gamma)$.

(So $(\mathbf{let})$ is equivalent to $\dfrac{\Gamma \vdash M_1 : \sigma \qquad \Gamma, x : \sigma \vdash M_2 : \tau'}{\Gamma \vdash (\mathtt{let}\, x = M_1 \,\mathtt{in}\, M_2) : \tau'}$ if $x \notin dom(\Gamma)$.)

Example of using the (let) rule

$$(\mathbf{let})\ \frac{\Gamma \vdash M_1 : \tau \qquad \Gamma, x : \forall A\,(\tau) \vdash M_2 : \tau'}{\Gamma \vdash (\mathtt{let}\, x = M_1 \,\mathtt{in}\, M_2) : \tau'} \ \text{if } x \notin dom(\Gamma) \text{ and } A = ftv(\tau) - ftv(\Gamma)$$

If $\Gamma \vdash M_1 : \tau$ is $\boxed{y : \beta, z : \forall \gamma\,(\gamma \to \gamma \to bool) \vdash \lambda u\,(y) : \alpha \to \beta}$

then $A = \{\alpha, \beta\} - \{\beta\} = \{\alpha\}$ and $\forall A\,(\tau) = \forall \alpha\,(\alpha \to \beta)$.

So if $\Gamma, x : \forall A\,(\tau) \vdash M_2 : \tau'$ is

$$\boxed{y : \beta, z : \forall \gamma\,(\gamma \to \gamma \to bool), x : \forall \alpha\,(\alpha \to \beta) \vdash z\,(x\,y)\,(x\,\mathtt{nil}) : bool}$$

then applying (**let**) yields

$$\boxed{y : \beta, z : \forall \gamma\,(\gamma \to \gamma \to bool) \vdash \mathtt{let}\, x = \lambda u\,(y)\,\mathtt{in}\, z\,(x\,y)\,(x\,\mathtt{nil}) : bool}$$

Two examples involving self-application

$$M \triangleq \texttt{let}\ f = \lambda x_1\,(\lambda x_2\,(x_1))\ \texttt{in}\ f\ f$$

$$M' \triangleq (\lambda f\,(f\ f))\ \lambda x_1\,(\lambda x_2\,(x_1))$$

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

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

$$A = ftv(\tau_2) \qquad \text{(C0)}$$

$$\tau_2 = \tau_3 \to \tau_4 \qquad \text{(C1)}$$

$$\tau_4 = \tau_5 \to \tau_6 \qquad \text{(C2)}$$

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

$$\tau_7 = \tau_8 \to \tau_1 \qquad \text{(C4)}$$

$$\forall A\,(\tau_2) \succ \tau_7 \qquad \text{(C5)}$$

$$\forall A\,(\tau_2) \succ \tau_8 \qquad \text{(C6)}$$

Principal type schemes for closed expressions

A 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 type scheme $\forall A'\,(\tau')$,
if $\vdash M : \forall A'\,(\tau')$, then $\forall A\,(\tau) \succ \tau'$

Theorem (Hindley; Damas-Milner)

Theorem. 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 closed Mini-ML expression 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

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

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 $T \in \mathbf{Sub}$

(any other substitution S' can be factored through S, by specialising S with T)

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 $TS = S'$ 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 $\sigma \succ \tau'$.)

Example typing problem and solutions

Typing problem

$$\boxed{x : \forall \alpha \, (\beta \to (\gamma \to \alpha)) \vdash x \, \mathtt{true} : ?}$$

has solutions:

- $S_1 = \{\beta \mapsto bool\}, \sigma_1 = \forall \alpha \, (\gamma \to \alpha)$
- $S_2 = \{\beta \mapsto bool, \gamma \mapsto \alpha\}, \sigma_2 = \forall \alpha' \, (\alpha \to \alpha')$
- $S_3 = \{\beta \mapsto bool, \gamma \mapsto \alpha\}, \sigma_3 = \forall \alpha' \, (\alpha \to (\alpha' \to \alpha'))$
- $S_4 = \{\beta \mapsto bool, \gamma \mapsto bool\}, \sigma_3 = \forall \{\,\} \, (bool \to bool)$

Both (S_1, σ_1) and (S_2, σ_2) are in fact principal solutions.

Properties of the Mini-ML typing relation with respect to substitution and type scheme specialisation

- 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

$$\Gamma \vdash M : \sigma'$$

Requirements for a 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 $S \in \mathbf{Sub}$ and a Mini-ML type τ, or the exception $FAIL$.

- If $\Gamma \vdash M : ?$ has a solution (cf. Slide 28), 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 $FAIL$.

How the principal typing algorithm pt works

$$pt(\Gamma \vdash M : ?) = (S, \tau) \mid FAIL$$

- Call pt recursively following the structure of M and guided by the typing rules, bottom-up.
- Thread substitutions sequentially and compose them together when returning from a recursive call.
- When types need to agree to satisfy a typing rule, use mgu (and pt returns $FAIL$ only if mgu does).
- When types are unknown, generate a fresh type variable.

Some of the clauses in a definition of pt

Function abstractions: $pt(\Gamma \vdash \lambda x\,(M) : ?) \triangleq$

$\quad$ let α = fresh in

$\quad$ let $(S, \tau) = pt(\Gamma, x : \alpha \vdash M : ?)$ in $(S, S(\alpha) \to \tau)$

Function applications: $pt(\Gamma \vdash M_1\,M_2 : ?) \triangleq$

$\quad$ let $(S_1, \tau_1) = pt(\Gamma \vdash M_1 : ?)$ in

$\quad$ let $(S_2, \tau_2) = pt(S_1\,\Gamma \vdash M_2 : ?)$ in

$\quad$ let α = fresh in

$\quad$ let $S_3 = mgu(S_2\,\tau_1, \tau_2 \to \alpha)$ in $(S_3 S_2 S_1, S_3(\alpha))$

ML types and expressions for mutable references

$$
\begin{array}{rcll}
\tau & ::= & \ldots & \\
 & | & \mathit{unit} & \text{unit type} \\
 & | & \tau\ \mathit{ref} & \text{reference type} \\
 & & & \\
M & ::= & \ldots & \\
 & | & () & \text{unit value} \\
 & | & \mathtt{ref}\ M & \text{reference creation} \\
 & | & !M & \text{dereference} \\
 & | & M := M & \text{assignment}
\end{array}
$$

Midi-ML's extra typing rules

$$(\textbf{unit})\ \frac{}{\Gamma \vdash () : \mathit{unit}}$$

$$(\textbf{ref})\ \frac{\Gamma \vdash M : \tau}{\Gamma \vdash \texttt{ref}\ M : \tau\ \mathit{ref}}$$

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

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

Example

The expression

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

has type $\boldsymbol{unit}$.

Midi-ML transition system

Small-step transition relations

$$\langle M, s \rangle \to \langle M', s' \rangle$$
$$\langle M, s \rangle \to \mathit{FAIL}$$

where

- M, M' range over Midi-ML expressions
- s, s' range over **states** = finite functions $s = \{x_1 \mapsto V_1, \ldots, x_n \mapsto V_n\}$ mapping variables x_i to **values** V_i:

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

- configurations $\langle M, s \rangle$ are required to satisfy that the free variables of expression M are in the domain of definition of the state s
- symbol FAIL represents a run-time error

are inductively defined by syntax-directed rules...

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:

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

$$\left\langle \begin{array}{l} \texttt{let}\ r = \texttt{ref}\ \lambda x\,(x)\ \texttt{in} \\ \texttt{let}\ u = (r := \lambda x'\,(\texttt{ref}\ !x'))\ \texttt{in}\ (!r)\,()\ ,\ \{\} \end{array} \right\rangle$$

$$\begin{array}{ll}
\to^* & \langle \texttt{let}\ u = (r := \lambda x'\,(\texttt{ref}\ !x'))\ \texttt{in}\ (!r)\,()\ ,\ \{r \mapsto \lambda x\,(x)\}\rangle \\
\to^* & \langle (!r)\,()\ ,\ \{r \mapsto \lambda x'\,(\texttt{ref}\ !x')\}\rangle \\
\to & \langle \lambda x'\,(\texttt{ref}\ !x')\,()\ ,\ \{r \mapsto \lambda x'\,(\texttt{ref}\ !x')\}\rangle \\
\to & \langle \texttt{ref}\ !()\ ,\ \{r \mapsto \lambda x'\,(\texttt{ref}\ !x')\}\rangle \\
\to & \mathit{FAIL}
\end{array}$$

Value-restricted typing rule for `let`-expressions

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

$(\dagger)$ 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

$$\begin{aligned}&(\mathbf{var}\succ) + (\mathbf{bool}) + (\mathbf{if}) + (\mathbf{nil}) + (\mathbf{cons}) + (\mathbf{case}) + (\mathbf{fn}) + \\ &(\mathbf{app}) + (\mathbf{unit}) + (\mathbf{ref}) + (\mathbf{get}) + (\mathbf{set}) + (\mathbf{letv})\end{aligned}$$

then **evaluation of M does not fail**,
i.e. there is no sequence of transitions of the form

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

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

In Midi-ML's value-restricted type system, some expressions that were typeable using **(let)** become untypeable using **(letv)**.

For example (exercise):

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

But one can often[1] use η-expansion

replace M by $\lambda x\,(M\,x)$ (where $x \notin fv(M)$)

or β-reduction

replace $(\lambda x\,(M))\,N$ by $M[N/x]$

to get around the problem.

(**1** These transformations do not always preserve meaning [contextual equivalence].)

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

For example, $\lambda f\,((f\,\texttt{true}) :: (f\,\texttt{nil}))$ and $\lambda f\,(f\,f)$ are not typeable in the Mini-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

$$(\mathbf{id})\,\frac{}{\Gamma \vdash x : \pi} \text{ if } (x : \pi) \in \Gamma$$

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

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

ML + full polymorphic types has undecidable type-checking

Fact (?). For the modified Mini-ML type system with

- full polymorphic types replacing types and type schemes
- $(\mathbf{id}) + (\mathbf{gen}) + (\mathbf{spec})$ replacing $(\mathbf{var} \succ)$

the type checking and typeability problems are 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 7)
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

$$\begin{array}{lll} \tau & ::= & \alpha & \text{type variable} \\ & \mid & \tau \to \tau & \text{function type} \\ & \mid & \forall \alpha\, (\tau) & \forall\text{-type} \end{array}$$

Expressions

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

(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

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

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

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

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 $i = 1 \ldots n$.)
- M is a PLC expression
- τ is a PLC type.

PLC type system

$$(\mathbf{var})\ \frac{}{\Gamma \vdash x : \tau} \quad \text{if } (x : \tau) \in \Gamma$$

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

$$(\mathbf{app})\ \frac{\Gamma \vdash M : \tau_1 \to \tau_2 \qquad \Gamma \vdash M' : \tau_1}{\Gamma \vdash M\,M' : \tau_2}$$

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

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

An incorrect proof

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

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

$$typ(\Gamma, x : \tau \vdash x : ?) \triangleq \tau$$

Function abstractions

$$\begin{array}{l} typ(\Gamma \vdash \lambda x : \tau_1\,(M) : ?) \triangleq \\ \ \ \mathbf{let}\ \ \tau_2 = typ(\Gamma, x : \tau_1 \vdash M : ?)\ \mathbf{in}\ \tau_1 \to \tau_2 \end{array}$$

Function applications

$$\begin{array}{l} typ(\Gamma \vdash M_1\,M_2 : ?) \triangleq \\ \ \ \mathbf{let}\ \tau_1 = typ(\Gamma \vdash M_1 : ?)\ \mathbf{in} \\ \ \ \mathbf{let}\ \tau_2 = typ(\Gamma \vdash M_2 : ?)\ \mathbf{in} \\ \ \ \mathbf{case}\ \tau_1\ \ \mathbf{of} \quad \tau \to \tau' \ \mapsto\ \mathbf{if}\ \tau = \tau_2\ \mathbf{then}\ \tau'\ \mathbf{else}\ FAIL \\ \qquad\qquad\quad\ \mid \qquad\quad \_ \ \ \mapsto\ FAIL \end{array}$$

PLC type-checking algorithm, II

Type generalisations

$$
\begin{array}{l}
\mathit{typ}(\Gamma \vdash \Lambda\alpha\,(M) : ?) \triangleq \\
\;\textbf{let}\ \tau = \mathit{typ}(\Gamma \vdash M : ?)\ \textbf{in}\ \forall\alpha\,(\tau)
\end{array}
$$

Type specialisations

$$
\begin{array}{l}
\mathit{typ}(\Gamma \vdash M\,\tau_2 : ?) \triangleq \\
\;\textbf{let}\ \tau = \mathit{typ}(\Gamma \vdash M : ?)\ \textbf{in} \\
\;\textbf{case}\ \tau\ \textbf{of}\quad \forall\alpha\,(\tau_1) \quad \mapsto \quad \tau_1[\tau_2/\alpha] \\
\qquad\qquad\;\; | \qquad\qquad\quad \_ \quad \mapsto \quad \mathit{FAIL}
\end{array}
$$

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 $\to$ 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

$$M \to^* N \;{}^*\!\!\leftarrow M'$$

Polymorphic booleans

$$bool \triangleq \forall\alpha\,(\alpha \to (\alpha \to \alpha))$$

$$True \triangleq \Lambda\alpha\,(\lambda x_1 : \alpha, x_2 : \alpha\,(x_1))$$

$$False \triangleq \Lambda\alpha\,(\lambda x_1 : \alpha, x_2 : \alpha\,(x_2))$$

$$if \triangleq \Lambda\alpha\,(\lambda b : bool, x_1 : \alpha, x_2 : \alpha\,(b\,\alpha\,x_1\,x_2))$$

Iteratively defined functions on finite lists

$A^* \triangleq$ finite lists of elements of the set A

Given a set B, an element $x' \in B$, and a function $f : A \to B \to B$, the **iteratively defined function** $\mathit{listIter}\, x'\, f$ is the unique function $g : A^* \to B$
satisfying:

$$g\,\mathit{Nil} = x'$$
$$g\,(x :: \ell) = f\,x\,(g\,\ell)$$

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

Polymorphic lists

$$\alpha\, list \triangleq \forall \alpha' \,(\alpha' \to (\alpha \to \alpha' \to \alpha') \to \alpha')$$

$$Nil \triangleq \Lambda \alpha, \alpha' \,(\lambda x' : \alpha', f : \alpha \to \alpha' \to \alpha' \,(x'))$$

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

List iteration in PLC

$$
\begin{aligned}
iter \triangleq \Lambda\alpha, \alpha' (\lambda x' : \alpha', f : \alpha \to \alpha' \to \alpha' (\\
\lambda \ell : \alpha\, list\, (\ell\, \alpha'\, x'\, f)))
\end{aligned}
$$

satisfies:

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

Standard ML signatures and structures

```
signature QUEUE =
  sig
    type 'a queue
    exception Empty
    val empty : 'a queue
    val insert : 'a * 'a queue -> 'a queue
    val remove : 'a queue -> 'a * 'a queue
end

structure Queue =
  struct
    type 'a queue = 'a list * 'a list
    exception Empty
    val empty = (nil, nil)
    fun insert (f, (front,back)) = (f::front, back)
    fun remove (nil, nil) = raise Empty
      | remove (front, nil) = remove (nil, rev front)
      | remove (front, b::back) = (b, (front, back))
  end
```

PLC + existential types

Types

$t ::= \cdots \mid \exists\, \alpha\, (\tau)$

Expressions

$M ::= \cdots \mid \mathtt{pack}\, (\tau, M) : \exists\, \alpha\, (\tau) \mid$
$\qquad \mathtt{unpack}\, M : \exists\, \alpha\, (\tau)\ \mathtt{as}\ (\alpha, x)\ \mathtt{in}\ M : \tau$

Typing rules

$$(\exists\mathbf{intro})\ \frac{\Gamma \vdash M : \tau[\tau'/\alpha]}{\Gamma \vdash (\mathtt{pack}\, (\tau', M) : \exists\, \alpha\, (\tau)) : \exists\, \alpha\, (\tau)}$$

$$(\exists\mathbf{elim})\ \frac{\Gamma \vdash E : \exists\, \alpha\, (\tau) \qquad \Gamma, x : \tau \vdash M' : \tau'}{\Gamma \vdash (\mathtt{unpack}\, E : \exists\, \alpha\, (\tau)\ \mathtt{as}\ (\alpha, x)\ \mathtt{in}\ M' : \tau') : \tau'}$$
if $\alpha \notin ftv(\Gamma, \tau')$

Reduction

$$\mathtt{unpack}\, (\mathtt{pack}\, (\tau', M) : \exists\, \alpha\, (\tau)) : \exists\, \alpha\, (\tau)\ \mathtt{as}\ (\alpha, x)\ \mathtt{in}\ M' : \tau' \to M'[\tau'/\alpha, M/x]$$

Existential types in PLC

$$\exists\,\alpha\,(\tau) \triangleq \forall\beta\,((\forall\alpha\,(\tau \to \beta)) \to \beta)$$

$$\mathtt{pack}\,(\tau', M) : \exists\,\alpha\,(\tau) \triangleq \Lambda\beta\,(\lambda y : \forall\alpha\,(\tau \to \beta)\,(y\,\tau'\,M))$$

$$\mathtt{unpack}\,E : \exists\,\alpha\,(\tau)\ \mathtt{as}\ (\alpha, x)\ \mathtt{in}\ M' : \tau' \triangleq E\,\tau'(\Lambda\alpha\,(\lambda x : \tau\,(M')))$$

(where $\beta \notin ftv(\alpha\,\tau\,\tau'\,M\,M')$)

These definitions satisfy the typing and reduction rules on the previous slide (exercise).

Dependent Functions

Given a set A and a family of sets B_a indexed by the elements a of A, we get a set

$$\prod_{a \in A} B_a \triangleq \{F \in A \to \bigcup_{a \in A} B_a \mid \forall (a, b) \in F \ (b \in B_a)\}$$

of **dependent functions**. Each $F \in \prod_{a \in A} B_a$ is a single-valued and total relation that associates to each $a \in A$ an element in B_a (usually written $F\,a$).

For example if $A = \mathbb{N}$ and for each $n \in \mathbb{N}$, $B_n = \{0, 1\}^n \to \{0, 1\}$, then $\prod_{n \in \mathbb{N}} B_n$ consists of functions mapping each number n to an n-ary Boolean operation.

A tautology checker

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

Defining types $n\,\mathit{AryBoolOp}$ for each natural number $n \in \mathbb{N}$

$$\begin{cases} 0\,\mathit{AryBoolOp} & \triangleq \mathit{bool} \\ (n+1)\,\mathit{AryBoolOp} & \triangleq \mathit{bool} \to (n\,\mathit{AryBoolOp}) \end{cases}$$

then $\mathit{taut}\,n$ has type $(n\,\mathit{AryBoolOp}) \to \mathit{bool}$, 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 x) AryBoolOp = Bool -> x AryBoolOp

taut : (x : Nat) -> x AryBoolOp -> Bool
taut zero     f = f
taut (succ x) f = taut x (f true) and taut x (f false)
```

Dependent function types $\Pi x : \tau\,(\tau')$

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

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

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

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

Conversion typing rule

Dependent type systems usually feature a rule of the form

$$\frac{\Gamma \vdash M : \tau}{\Gamma \vdash M : \tau'} \quad \text{if } \tau \approx \tau'$$

where $\tau \approx \tau'$ is some relation of **equality between types** (e.g. inductively defined in some way).

For example one would expect $(1 + 1)\ \mathtt{AryBoolOp} \approx 2\ \mathtt{AryBoolOp}$.

For decidability of type-checking, one needs $\approx$ to be a decidable relation between type expressions.

Pure Type Systems (PTS) – syntax

In a PTS type expressions and term expressions are lumped together into a single syntactic category of **pseudo-terms**:

$$
\begin{array}{rcll}
t & ::= & x & \text{variable} \\
 & | & s & \text{sort} \\
 & | & \Pi x : t\,(t) & \text{dependent function type} \\
 & | & \lambda x : t\,(t) & \text{function abstraction} \\
 & | & t\,t & \text{function application}
\end{array}
$$

where x ranges over a countably infinite set $\mathbf{Var}$ of variables and s ranges over a disjoint set $\mathbf{Sort}$ of **sort symbols** – constants that denote various universes (= types whose elements denote types of various sorts) [**kind** is a commonly used synonym for *sort*]. $\lambda x : t\,(t')$ and $\Pi x : t\,(t')$ both bind free occurrences of x in the pseudo-term t'.

$\boxed{t[t'/x]}$ denotes result of capture-avoiding substitution of t' for all free occurrences of x in t.

$\boxed{t \to t'} \triangleq \Pi x : t\,(t')$ where $x \notin fv(t')$.

Pure Type Systems – beta-conversion

- **beta-reduction** of pseudo-terms: $\boxed{t \longrightarrow t'}$ means t' can be obtained from t (up to alpha-conversion, of course) by replacing a subexpression which is a **redex** by its corresponding **reduct**. There is only one form of redex-reduct pair:

$$(\lambda x : t\,(t_1))\,t_2 \longrightarrow t_1[t_2/x]$$

- As usual, $\longrightarrow^*$ denotes the reflexive-transitive closure of $\longrightarrow$.
- **beta-conversion** of pseudo-terms: $=_\beta$ is the reflexive-symmetric-transitive closure of $\longrightarrow$ (i.e. the smallest equivalence relation containing $\longrightarrow$).

Pure Type Systems – specifications

The typing rules for a particular PTS are parameterised by a **specification** $\mathbf{S} = (\mathcal{S}, \mathcal{A}, \mathcal{R})$ where:

- $\mathcal{S} \subseteq \mathbf{Sort}$

  Elements $s \in \mathcal{S}$ denote the different universes of types in this PTS.

- $\mathcal{A} \subseteq \mathbf{Sort} \times \mathbf{Sort}$

  Elements $(s_1, s_2) \in \mathcal{A}$ are called **axioms**. They determine the typing relation between universes in this PTS.

- $\mathcal{R} \subseteq \mathbf{Sort} \times \mathbf{Sort} \times \mathbf{Sort}$

  Elements $(s_1, s_2, s_3) \in \mathcal{R}$ are called rules. They determine which kinds of dependent function can be formed and in which universes they live.

The PTS with specification $\mathbf{S}$ will be denoted $\boxed{\lambda\mathbf{S}}$.

Pure Type Systems – typing judgements

take the form

$$\boxed{\Gamma \vdash t : t'}$$

where t, t' are pseudo-terms and Γ is a **context**, a form of typing environment given by the grammar

$$\Gamma ::= \diamond \mid \Gamma, x : t$$

(Thus contexts are finite ordered lists of (variable,pseudo-term)-pairs, with the empty list denoted $\diamond$, the head of the list on the right and list-cons denoted by $\_,\_$. Unlike previous type systems in this course, **the order in which typing declarations $x : t$ occur in a context is important**.)
A typing judgement is **derivable** if it is in the set inductively generated by the rules on the next slide, which are parameterised by a given specification $\mathbf{S} = (\mathcal{S}, \mathcal{A}, \mathcal{R})$.

Pure Type Systems – typing rules

$$(\textbf{axiom})\ \frac{}{\diamond \vdash s_1 : s_2}\ \text{if } (s_1, s_2) \in \mathcal{A}$$

$$(\textbf{start})\ \frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}\ \text{if } x \notin dom(\Gamma)$$

$$(\textbf{weaken})\ \frac{\Gamma \vdash M : A \qquad \Gamma \vdash B : s}{\Gamma, x : B \vdash M : A}\ \text{if } x \notin dom(\Gamma)$$

$$(\textbf{conv})\ \frac{\Gamma \vdash M : A \qquad \Gamma \vdash B : s}{\Gamma \vdash M : B}\ \text{if } A =_\beta B$$

$$(\textbf{prod})\ \frac{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash \Pi x : A\,(B) : s_3}\ \text{if } (s_1, s_2, s_3) \in \mathcal{R}$$

$$(\textbf{abs})\ \frac{\Gamma, x : A \vdash M : B \qquad \Gamma \vdash \Pi x : A\,(B) : s}{\Gamma \vdash \lambda x : A\,(M) : \Pi x : A\,(B)}$$

$$(\textbf{app})\ \frac{\Gamma \vdash M : \Pi x : A\,(B) \qquad \Gamma \vdash N : A}{\Gamma \vdash M\,N : B[N/x]}$$

(A, B, M, N range over pseudoterms, s, s_1, s_2, s_3 over sort symbols)

Example PTS typing derivations

$$
(\mathbf{prod})\;\dfrac{(\mathbf{axiom})\;\dfrac{}{\diamond \vdash * : \Box} \qquad (\mathbf{weaken})\;\dfrac{(\mathbf{axiom})\;\dfrac{}{\diamond \vdash * : \Box} \qquad (\mathbf{axiom})\;\dfrac{}{\diamond \vdash * : \Box}}{\diamond, x : * \vdash * : \Box}}{\diamond \vdash * \to * : \Box}
$$

$$
(\mathbf{abs})\;\dfrac{(\mathbf{start})\;\dfrac{(\mathbf{axiom})\;\dfrac{}{\diamond \vdash * : \Box}}{\diamond, x : * \vdash x : *} \qquad \dfrac{\vdots}{\diamond \vdash * \to * : \Box}}{\diamond \vdash \lambda x : *\,(x) : * \to *}
$$

Here we assume that the PTS specification $\mathbf{S} = (\mathcal{S}, \mathcal{A}, \mathcal{R})$ has $* \in \mathcal{S}$, $\Box \in \mathcal{S}$, $(*, \Box) \in \mathcal{A}$ and $(\Box, \Box, \Box) \in \mathcal{R}$.
(Recall that $* \to * \triangleq \Pi x : *\,(*)$.)

Properties of Pure Type Systems in general

- **Correctness of types.** If $\Gamma \vdash M : A$, then either $A \in \mathcal{S}$, or $\Gamma \vdash A : s$ for some $s \in \mathcal{S}$.
- **Church-Rosser Property** (aka **confluence**). $t =_\beta t'$ iff $\exists u\,(t \to^* u \;\wedge\; t' \to^* u)$
- **Subject Reduction.** If $\Gamma \vdash M : A$ and $M \to M'$, then $\Gamma \vdash M' : A$.
- **Uniqueness of Types.** A PTS specification $\mathbf{S} = (\mathcal{S}, \mathcal{A}, \mathcal{R})$ is said to be **functional** if both $\mathcal{A}$ and $\mathcal{R}_s \triangleq \{(s_2, s_3) \mid (s, s_2, s_3) \in \mathcal{R}\}$ for each $s \in \mathcal{S}$, are single-valued binary relations.
  In this case $\lambda\mathbf{S}$ satisfies: if $\Gamma \vdash M : A$ and $\Gamma \vdash M : B$, then $A =_\beta B$.

Type-checking for a PTS, $\lambda\mathbf{S}$

Definition. A pseudo-term t is **legal** for a PTS specification $\mathbf{S} = (\mathcal{S}, \mathcal{A}, \mathcal{R})$ if either $t \in \mathcal{S}$ or $\Gamma \vdash t : t'$ is derivable in $\lambda\mathbf{S}$ for some Γ and t'.

Recall the **type-checking** and **typeability** problems for a type system.

Fact(van Bentham Jutting): these problems for $\lambda\mathbf{S}$ are decidable if $\mathbf{S}$ is finite and $\lambda\mathbf{S}$ is **normalizing**, meaning that for every legal pseudo-term there is some finite chain of beta-reductions leading to a beta-normal form.

Fact (Meyer): the problems are undecidable for the PTS $\lambda *$ with specification $\mathcal{S} = \{*\}$, $\mathcal{A} = \{(*,*)\}$ and $\mathcal{R} = \{(*,*,*)\}$.

PLC versus the Pure Type System $\boldsymbol{\lambda 2}$

PTS signature:

$$\mathbf{2} \triangleq (\mathcal{S}_2, \mathcal{A}_2, \mathcal{R}_2) \text{ where } \begin{cases} \mathcal{S}_2 & \triangleq & \{*, \Box\} \\ \mathcal{A}_2 & \triangleq & \{(*, \Box)\} \\ \mathcal{R}_2 & \triangleq & \{(*, *, *), (\Box, *, *)\} \end{cases}$$

Translation of PLC types and terms to $\boldsymbol{\lambda 2}$ pseudo-terms:

$$\begin{aligned} [\![\alpha]\!] &= \alpha \\ [\![\tau \to \tau']\!] &= \Pi x : [\![\tau]\!] \, ([\![\tau']\!]) \\ [\![\forall \alpha \, (\tau)]\!] &= \Pi \alpha : * \, ([\![\tau']\!]) \\ [\![x]\!] &= x \\ [\![\lambda x : \tau \, (M)]\!] &= \lambda x : [\![\tau]\!] \, ([\![M]\!]) \\ [\![M \, M']\!] &= [\![M]\!] \, [\![M']\!] \\ [\![\Lambda \alpha \, (M)]\!] &= \lambda \alpha : * \, ([\![M]\!]) \\ [\![M \, \tau]\!] &= [\![M]\!] \, [\![\tau]\!] \end{aligned}$$

Properties of the translation from PLC to $\boldsymbol{\lambda 2}$

- If $\{\,\} \vdash M : \tau$ is derivable in PLC, then $\diamond \vdash [\![\tau]\!] : *$ and $\diamond \vdash [\![M]\!] : [\![\tau]\!]$ are derivable in $\lambda 2$
- In $\lambda 2$, if $\diamond \vdash t : \square$, then $t = *$; if $\diamond \vdash t : *$, then $t = [\![\tau]\!]$ for some closed PLC type τ; and if $\diamond \vdash t : t'$ then $t = [\![M]\!]$ and $t' = [\![\tau]\!]$ for PLC expressions satisfying $\{\,\} \vdash M : \tau$.
- Under the translation, the reduction behaviour of PLC terms is preserved and reflected by beta-reduction in $\lambda 2$. (Note in particular that PLC types are translated to pseudo-terms in beta-normal form.)

System $\mathbf{F}_\omega$ as a Pure Type System: $\boldsymbol{\lambda\omega}$

PTS specification $\omega = (\mathcal{S}_\omega, \mathcal{A}_\omega, \mathcal{R}_\omega)$:

$$\mathcal{S}_\omega \triangleq \{*, \square\}$$
$$\mathcal{A} \triangleq \{(*, \square)\}$$
$$\mathcal{R} \triangleq \{(*, *, *), (\square, *, *), (\square, \square, \square)\}$$

As in $\boldsymbol{\lambda 2}$, sort $*$ is a universe of types; but in $\boldsymbol{\lambda\omega}$, the rule $(\mathbf{prod})$ for $(\square, \square, \square)$ means that $\diamond \vdash t : \square$ holds for all the 'simple types' over the ground type $*$ – the ts generated by the grammar $t ::= * \mid t \to t$

Hence rule $(\mathbf{prod})$ for $(\square, *, *)$ now gives many more legal pseudo-terms of type $*$ in $\boldsymbol{\lambda\omega}$ compared with $\boldsymbol{\lambda 2}$ (PLC), such as

$$\diamond \vdash (\Pi T : * \to * \, (\Pi \alpha : * \, (\alpha \to T\,\alpha))) : *$$
$$\diamond \vdash (\Pi T : * \to * \, (\Pi \alpha, \beta : * \, ((\alpha \to T\,\beta) \to T\,\alpha \to T\,\beta))) : *$$

Monads in ML

A monad in ML is given by type $\tau(\alpha)$ with a free type variable α together with expressions

$$\begin{aligned} unit &: \alpha \to \tau(\alpha) \\ lift &: (\alpha \to \tau(\beta)) \to \tau(\alpha) \to \tau(\beta) \end{aligned}$$

(writing $\tau(\beta)$ for the result of replacing α by β in τ) satisfying some equational properties [omitted].
E.g.

- list monad $\tau(\alpha) = \alpha\ list$
- global state monad $\tau(\alpha) = \sigma \to (\alpha * \sigma)$ (for some type σ of states)
- simple exception monad $\tau(\alpha) = (\alpha, \varepsilon) sum$ (for some type ε of exception names)

[definitions of $unit$ and $lift$ in each case omitted]

Examples of $\lambda\omega$ type constructions

- Product types (cf. the PLC representation of product types):

$$\mathtt{P} \triangleq \lambda \alpha, \beta : * \,(\Pi \gamma : * \,((\alpha \to \beta \to \gamma) \to \gamma))$$
$$\diamond \vdash \mathtt{P} : * \to * \to *$$

- Monad transformer for state (using a type $\diamond \vdash S : *$ for states):

$$\mathtt{M} \triangleq \lambda T : * \to * \,(\lambda \alpha : * \,(S \to T(\mathtt{P}\,\alpha\, S)))$$
$$\diamond \vdash \mathtt{M} : (* \to *) \to * \to *$$

- Existential types (cf. the PLC representation of existential types):

$$\exists \triangleq \lambda T : * \to * \,(\Pi \beta : * \,((\Pi \alpha : * \,(T\,\alpha \to \beta)) \to \beta))$$
$$\diamond \vdash \exists : (* \to *) \to *$$

Type-checking for $\mathbf{F}_\omega$

Fact (Girard): System $\mathbf{F}_\omega$ is **strongly normalizing** in the sense that for any legal pseudo-term t, there is no infinite chain of beta-reductions $t \to t_1 \to t_2 \to \cdots$.

As as corollary we have that the type-checking and typeability problems for $\mathbf{F}_\omega$ are decidable.

Constructive interpretation of logic

- **Implication:** a proof of $\varphi \to \psi$ is a construction that transforms proofs of φ into proofs of ψ.
- **Negation:** a proof of $\neg\varphi$ is a construction that from any (hypothetical) proof of φ produces a contradiction (= proof of falsity $\bot$)
- **Disjunction:** a proof of $\varphi \vee \psi$ is an object that manifestly is either a proof of φ, or a proof of ψ.
- **For all:** a proof of $\forall x\,(\varphi(x))$ is a construction that transforms the objects a over which x ranges into proofs of $\varphi(a)$.
- **There exists:** a proof of $\exists\, x\,(\varphi(x))$ is given by a pair consisting of an object a and a proof of $\varphi(a)$.

The **Law of Excluded Middle** (LEM) $\boxed{\forall p\,(p \vee \neg p)}$ is a classical tautology (has truth-value `true`), but is rejected by constructivists.

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 $b^a = (\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = \sqrt{2}^{\sqrt{2}.\sqrt{2}} = \sqrt{2}^2 = 2$.

QED

Example of a constructive proof

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

Proof. $\sqrt{2}$ is irrational by a well-known constructive proof attributed to Euclid.

$2\log_2 3$ is irrational, by an easy constructive proof (exercise).

So we can take $a = 2\log_2 3$ and $b = \sqrt{2}$, for which we have that $b^a = (\sqrt{2})^{2\log_2 3} = (\sqrt{2}^2)^{\log_2 3} = 2^{\log_2 3} = 3$ is rational.

QED

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 multiset (= unordered list) 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:

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

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

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

A 2IPC proof

Writing $p \wedge q$ as an abbreviation for $\forall r\,((p \to q \to r) \to r)$, the sequent

$$\{\} \vdash \forall p\,(\forall q\,((p \wedge q) \to p))$$

is provable in 2IPC:

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

Curry-Howard correspondence

Logic	$\leftrightarrow$	Type system
propositions ϕ	$\leftrightarrow$	types τ
proofs p	$\leftrightarrow$	expressions M
'p is a proof of ϕ'	$\leftrightarrow$	'M is an expression of type τ'
simplification of proofs	$\leftrightarrow$	reduction of expressions

E.g.

2IPC $\leftrightarrow$ PLC

Mapping 2IPC proofs to PLC expressions

$$(\mathbf{Id})\ \Phi, \phi \vdash \phi \quad \mapsto \quad (\mathbf{id})\ \overline{x} : \Phi, x : \phi \vdash x : \phi$$

$$(\to\mathbf{I})\ \frac{\Phi, \phi \vdash \phi'}{\Phi \vdash \phi \to \phi'} \quad \mapsto \quad (\mathbf{fn})\ \frac{\overline{x} : \Phi, x : \phi \vdash M : \phi'}{\overline{x} : \Phi \vdash \lambda x : \phi\,(M) : \phi \to \phi'}$$

$$(\to\mathbf{E})\ \frac{\Phi \vdash \phi \to \phi' \qquad \Phi \vdash \phi}{\Phi \vdash \phi'} \quad \mapsto \quad (\mathbf{app})\ \frac{\overline{x} : \Phi \vdash M_1 : \phi \to \phi' \qquad \overline{x} : \Phi \vdash M_2 : \phi}{\overline{x} : \Phi \vdash M_1\, M_2 : \phi'}$$

$$(\forall\mathbf{I})\ \frac{\Phi \vdash \phi}{\Phi \vdash \forall p\,(\phi)} \quad \mapsto \quad (\mathbf{gen})\ \frac{\overline{x} : \Phi \vdash M : \phi}{\overline{x} : \Phi \vdash \Lambda p\,(M) : \forall p\,(\phi)}$$

$$(\forall\mathbf{E})\ \frac{\Phi \vdash \forall p\,(\phi)}{\Phi \vdash \phi[\phi'/p]} \quad \mapsto \quad (\mathbf{spec})\ \frac{\overline{x} : \Phi \vdash M : \forall p\,(\phi)}{\overline{x} : \Phi \vdash M\,\phi' : \phi[\phi'/p]}$$

The proof of the 2IPC sequent

$$\{\} \vdash \forall p\,(\forall q\,((p \wedge q) \to p))$$

given before is transformed by the mapping of 2IPC proofs to PLC expressions to

$$\{\} \vdash \Lambda p, q\,(\lambda z : p \wedge q\,(z\,p\,(\lambda x : p, y : q\,(x)))) \\ : \forall p\,(\forall q\,((p \wedge q) \to p))$$

with typing derivation:

$$
\textbf{(gen)}\,\dfrac{\textbf{(gen)}\,\dfrac{\textbf{(fn)}\,\dfrac{\textbf{(app)}\,\dfrac{\textbf{(fn)}\,\dfrac{\textbf{(fn)}\,\dfrac{\textbf{(id)}\,\dfrac{}{\{z : p \wedge q, x : p, y : q\} \vdash x : p}}{\{z : p \wedge q, x : p\} \vdash \lambda y : q\,(x) : q \to p}}{\{z : p \wedge q\} \vdash \lambda x : p, y : q\,(x) : p \to q \to p} \qquad \textbf{(spec)}\,\dfrac{\textbf{(id)}\,\dfrac{}{\{z : p \wedge q\} \vdash z : \forall r\,((p \to q \to r) \to r)}}{\{z : p \wedge q\} \vdash z\,p : (p \to q \to p) \to p}}{\{z : p \wedge q\} \vdash z\,p\,(\lambda x : p, y : q\,(x)) : p}}{\{\} \vdash \lambda z : p \wedge q\,(z\,p\,(\lambda x : p, y : q\,(x))) : (p \wedge q) \to p}}{\{\} \vdash \Lambda q\,(\lambda z : p \wedge q\,(z\,p\,(\lambda x : p, y : q\,(x)))) : \forall q\,((p \wedge q) \to p)}}{\{\} \vdash \Lambda p, q\,(\lambda z : p \wedge q\,(z\,p\,(\lambda x : p, y : q\,(x)))) : \forall p, q\,((p \wedge q) \to p)}
$$

Logical operations definable in 2IPC

- **Truth** $\top \triangleq \forall p\,(p \to p)$
- **Falsity** $\bot \triangleq \forall p\,(p)$
- **Conjunction** $\phi \wedge \psi \triangleq \forall p\,((\phi \to \psi \to p) \to p)$ (where $p \notin fv(\phi, \psi)$)
- **Disjunction** $\phi \vee \psi \triangleq \forall p\,((\phi \to p) \to (\psi \to p) \to p)$ (where $p \notin fv(\phi, \psi)$)
- **Negation** $\neg\phi \triangleq \phi \to \bot$
- **Bi-implication** $\phi \leftrightarrow \psi \triangleq (\phi \to \psi) \wedge (\psi \to \phi)$
- **Existential quantification** $\exists p\,(\phi) \triangleq \forall q\,(\forall p\,(\phi \to q) \to q)$ (where $q \notin fv(\phi, p)$)

LEM $\forall p\,(p \vee \neg p) = \forall p, q\,((p \to q) \to ((p \to \forall r\,(r)) \to q) \to q)$

Fact: $\{\} \vdash M : \forall p\,(p \vee \neg p)$ is not provable in PLC for any expression M.

Proof simplification $\leftrightarrow$ Expression reduction

$$(\to\mathbf{E})\ \dfrac{(\to\mathbf{I})\ \dfrac{\dfrac{\vdots}{\Phi, \phi \vdash \psi}}{\Phi \vdash \phi \to \psi} \qquad \dfrac{\vdots}{\Phi \vdash \phi}}{\Phi \vdash \psi} \quad \mapsto \quad \dfrac{\dfrac{\dfrac{\vdots}{\overline{x} : \Phi, x : \phi \vdash M : \psi}}{\overline{x} : \Phi \vdash \lambda x : \phi\,(M) : \phi \to \psi} \qquad \dfrac{\vdots}{\overline{x} : \Phi \vdash N : \phi}}{\overline{x} : \Phi \vdash (\lambda x : \phi\,(M))\, N : \psi}$$

simplify proof $\downarrow$ $\qquad\qquad$ $\downarrow$ beta-reduce expression

$$(\mathbf{cut})\ \dfrac{\dfrac{\vdots}{\Phi, \phi \vdash \psi} \qquad \dfrac{\vdots}{\Phi \vdash \phi}}{\Phi \vdash \psi} \quad \leftarrow\!\shortmid \quad \dfrac{\dfrac{\vdots}{\overline{x} : \Phi, x : \phi \vdash M : \psi} \qquad \dfrac{\vdots}{\overline{x} : \Phi \vdash N : \phi}}{\overline{x} : \Phi \vdash M[N/x] : \psi}\ (\mathbf{subst})$$

The rule $(\mathbf{subst})$ for PLC is **admissible**: if its hypotheses are valid PLC typing judgements, then so is its conclusion.

Hence, the rule $(\mathbf{cut})$ is admissible for 2IPC.

Type-inference versus proof search

Type-inference: given Γ and M, is there a type τ such that $\Gamma \vdash M : \tau$?

(For PLC/2IPC this is decidable.)

Proof-search: given Γ and ϕ, is there a proof term M such that $\Gamma \vdash M : \phi$?

(For PLC/2IPC this is undecidable.)

Calculus of Constructions

is the Pure Type System $\lambda\mathbf{C}$, where $\mathbf{C} = (\mathcal{S}_\mathrm{C}, \mathcal{A}_\mathrm{C}, \mathcal{R}_\mathrm{C})$ is the PTS specification with

$$\mathcal{S}_\mathrm{C} \triangleq \{\mathtt{Prop}, \mathtt{Set}\} \quad (\mathtt{Prop} = \text{a sort of propositions}, \mathtt{Set} = \text{a sort of types})$$
$$\mathcal{A}_\mathrm{C} \triangleq \{(\mathtt{Prop}, \mathtt{Set})\} \quad (\mathtt{Prop} \text{ is one of the types})$$
$$\mathcal{R}_\mathrm{C} \triangleq \{(\mathtt{Prop}, \mathtt{Prop}, \mathtt{Prop})^1, (\mathtt{Set}, \mathtt{Prop}, \mathtt{Prop})^2, (\mathtt{Prop}, \mathtt{Set}, \mathtt{Set})^3, (\mathtt{Set}, \mathtt{Set}, \mathtt{Set})^4\}$$

1. $\mathtt{Prop}$ has implications, $\phi \to \psi = \Pi x : \phi\,(\psi)$ (where $\phi, \psi : \mathtt{Prop}$ and $x \notin fv(\psi)$).

2. $\mathtt{Prop}$ has universal quantifications over elements of a type, $\Pi x : A\,(\phi(x))$ (where $A : \mathtt{Set}$ and $x : A \vdash \phi(x) : \mathtt{Prop}$).
N.B. A might be $\mathtt{Prop}$ ($\lambda 2 \subseteq \lambda\mathbf{C}$).

3. $\mathtt{Set}$ has types of function dependent on proofs of a proposition, $\Pi x : p\,(A(x))$ (where $p : \mathtt{Prop}$ and $x : p \vdash A(x) : \mathtt{Set}$).

4. $\mathtt{Set}$ has dependent function types, $\Pi x : A\,(B(x))$ (where $A : \mathtt{Set}$ and $x : A \vdash B(x) : \mathtt{Set}$).

Some general properties of $\boldsymbol{\lambda}\mathbf{C}$

- It extends both $\boldsymbol{\lambda}\mathbf{2}$ (PLC) and $\boldsymbol{\lambda\omega}$ ($\mathbf{F}_\omega$).

- $\boldsymbol{\lambda}\mathbf{C}$ is strongly normalizing.
- Type-checking and typeability are decidable.

- $\boldsymbol{\lambda}\mathbf{C}$ is logically consistent (relative to the usual foundations of classical mathematics), that is, there is no pseudo-term $\boldsymbol{t}$ satisfying $\diamond \vdash \boldsymbol{t} : \boldsymbol{\Pi p} : \texttt{Prop}\,(\boldsymbol{p})$.

  Indeed there is no proof of LEM ($\boldsymbol{\Pi p} : \texttt{Prop}\,(\neg \boldsymbol{p} \vee \boldsymbol{p})$).

Leibniz equality in $\lambda\mathbf{C}$

Gottfried Wilhelm Leibniz (1646–1716),
identity of indiscernibles:
duo quaedam communes proprietates eorum nequaquam possit
(two distinct things cannot have all their properties in common).

Given $\Gamma \vdash A : \mathtt{Set}$ in $\lambda\mathbf{C}$, we can define

$$\mathtt{Eq}_A \triangleq \lambda x, y : A\,(\Pi P : A \to \mathtt{Prop}\,(P\,x \leftrightarrow P\,y))$$

satisfying $\Gamma \vdash \mathtt{Eq}_A : A \to A \to \mathtt{Prop}$ and giving a well-behaved (but not extensional) equality predicate for elements of type A.

Extensionality

Functional extensionality:

$$\texttt{FunExt}_{A,B} \triangleq \Pi f, g : A \to B\,((\Pi x : A\,(\texttt{Eq}_B\,(f\,x)\,(g\,x))) \to \texttt{Eq}_{A\to B}\,f\,g)$$

If $\Gamma \vdash A, B : \texttt{Set}$ in $\lambda\mathbf{C}$, then $\Gamma \vdash \texttt{FunExt}_{A,B} : \texttt{Prop}$ is derivable, but for some A,B there does not exist a pseudo-term t for which $\Gamma \vdash t : \texttt{FunExt}_{A,B}$ is derivable.

Propositional extensionality:

$$\texttt{PropExt} \triangleq \Pi p, q : \texttt{Prop}\,((p \leftrightarrow q) \to \texttt{Eq}_{\texttt{Prop}}\,p\,q)$$

$\diamond \vdash \texttt{PropExt} : \texttt{Prop}$ is derivable in $\lambda\mathbf{C}$, but there does not exist a pseudo-term t for which $\diamond \vdash t : \texttt{PropExt}$ is derivable.

The Pure Type System $\lambda\mathbf{U}$

is given by the PTS specification $\mathbf{U} = (\mathcal{S}_{\mathrm{U}}, \mathcal{A}_{\mathrm{U}}, \mathcal{R}_{\mathrm{U}})$, where:

$$\begin{aligned}
\mathcal{S}_{\mathrm{U}} &\triangleq \{\mathtt{Prop}, \mathtt{Set}, \mathtt{Type}\} \\
\mathcal{A}_{\mathrm{U}} &\triangleq \{(\mathtt{Prop}, \mathtt{Set}), (\mathtt{Set}, \mathtt{Type})\} \\
\mathcal{R}_{\mathrm{U}} &\triangleq \{(\mathtt{Prop}, \mathtt{Prop}, \mathtt{Prop}), (\mathtt{Set}, \mathtt{Prop}, \mathtt{Prop}), (\mathtt{Type}, \mathtt{Prop}, \mathtt{Prop}), \\
&\qquad (\mathtt{Set}, \mathtt{Set}, \mathtt{Set}), (\mathtt{Type}, \mathtt{Set}, \mathtt{Set})\}
\end{aligned}$$

Theorem (Girard). $\lambda\mathbf{U}$ is logically inconsistent: every legal proposition $\Gamma \vdash P : \mathtt{Prop}$ has a proof $\Gamma \vdash M : P$. (In particular, there is a proof of falsity $\bot \triangleq \Pi p : \mathtt{Prop}\,(p)$.)

Inductive types (informally)

An inductive type is specified by giving

- **constructor functions** that allow us to inductively generate data values of that type
  (Some restrictions on how the inductive type appears in the domain type of constructors is needed to ensure termination of reduction and logical consistency.)
- **eliminators** for constructing functions on the data
- **computation rules** that explain how to simplify an eliminator applied to constructors.

Extending λC with an inductive type of natural numbers

Pseudo-terms

$$t ::= \cdots \mid \mathtt{Nat} \mid \mathtt{zero} \mid \mathtt{succ} \mid \mathtt{elimNat}(x.t)\; t\; t$$

Typing rules

- formation: $\diamond \vdash \mathtt{Nat} : \mathtt{Set}$
- introduction: $\diamond \vdash \mathtt{zero} : \mathtt{Nat} \qquad \diamond \vdash \mathtt{succ} : \mathtt{Nat} \to \mathtt{Nat}$
- elimination: $$\frac{\Gamma, x : \mathtt{Nat} \vdash A(x) : s \qquad \Gamma \vdash M : A(\mathtt{zero}) \qquad \Gamma \vdash F : \Pi x : \mathtt{Nat}\,(A(x) \to A(\mathtt{succ}\, x))}{\Gamma \vdash \mathtt{elimNat}(x.A)\; M\; F : \Pi x : \mathtt{Nat}\,(A(x))}$$
  (where $A(t)$ stands for $A[t/x]$)

Computation rules

$$\mathtt{elimNat}(x.A)\; M\; F\; \mathtt{zero} \longrightarrow M$$

$$\mathtt{elimNat}(x.A)\; M\; F\; (\mathtt{succ}\, N) \longrightarrow F\; N\; (\mathtt{elimNat}(x.A)\; M\; F\; N)$$

Inductive types of vectors

For a fixed parameter $\Gamma \vdash A : s$, the indexed family $(\mathtt{Vec}_A\, x \mid x : \mathtt{Nat})$ of types $\mathtt{Vec}_A\, x$ of **lists of A-values of length** x is inductively defined as follows:

Formation:

$$\frac{\Gamma \vdash N : \mathtt{Nat}}{\Gamma \vdash \mathtt{Vec}_A\, N : \mathtt{Set}}$$

Introduction:

$$\Gamma \vdash \mathtt{vnil}_A : \mathtt{Vec}_A\, \mathtt{zero}$$

$$\Gamma \vdash \mathtt{vcons}_A : A \to \Pi x : \mathtt{Nat}\, (\mathtt{Vec}_A\, x \to \mathtt{Vec}_A\, (\mathtt{succ}\, x))$$

Elimination and Computation:

[do-it-yourself]

Inductive identity propositions

For fixed parameters $\Gamma \vdash A : s$ and $\Gamma \vdash a : A$, the indexed family $(\mathtt{Id}_{A,a}\, x \mid x : A)$ of propositions $\mathtt{Id}_{A,a}\, x$ that a **and** x **are equal elements of type** A is inductively defined as follows:

Formation:

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \mathtt{Id}_{A,a}\, M : \mathtt{Prop}}$$

Introduction:

$$\Gamma \vdash \mathtt{refl}_{A,a} : \mathtt{Id}_{A,a}\, a$$

Elimination:

$$\frac{\Gamma, x : A, p : \mathtt{Id}_{A,a}\, x \vdash B(x,p) : s \qquad \Gamma \vdash N : B(a, \mathtt{refl}_{A,a})}{\Gamma \vdash \mathtt{J}_{A,a}(x,p.\, B)\, N : \Pi x : A\, (\Pi p : \mathtt{Id}_{A,a}\, x\, (B(x,p)))}$$

Computation:

$$\mathtt{J}_{A,a}(x,p.\, B)\, N\ a\, \mathtt{refl}_{A,a} \to N$$

Agda proof of $\forall x \in \mathbb{N}\,(x = 0 + x)$

```
data Nat : Set where
  zero : Nat
  succ : Nat -> Nat

add : Nat -> Nat -> Nat
add x zero     = x
add x (succ y) = succ (add x y)

data Id (A : Set)(x : A) : A -> Set where
  refl : Id A x x

cong : (A B : Set)(f : A -> B)(x y : A) ->
       Id A x y -> Id B (f x) (f y)
cong A B f x .x refl = refl

P : (x : Nat) -> Id Nat x (add zero x)
P zero     = refl
P (succ x) = cong Nat Nat succ x (add zero x) (P x)
```

Uniqueness of identity proofs

In $\lambda\mathbf{C}$ extended with inductive identity propositions, there are some types $\Gamma \vdash A : s$ for which it is impossible to prove that all equality proofs in $\mathtt{Id}_{A,x}\, y$ (where $x, y : A$) are identical.
That is, there is no pseudo-term uip satisfying

$$\Gamma \vdash uip : \Pi x, y : A\,(\Pi p, q : \mathtt{Id}_{A,x}\, y\,(\mathtt{Id}_{(\mathtt{Id}_{A,x}\, y),p}\, q))$$

By contrast, in Agda we have:

```
data Id (A : Set)(x : A) : A -> Set where
  refl : Id A x x

uip : (A : Set)(x y : A)(p q : Id A x y) -> Id (Id A x y) p q
uip A x .x refl refl = refl
```