Limitations of OCAML records

- The record types must be declared before they are used;
- a label e can belong to only one record type (otherwise $\texttt{fun}\ x \rightarrow x.e$) would have several incompatible types;
- we cannot build a record *incrementally*.

We will define a system that has:

polymorphic records: $\texttt{fun}\ x \rightarrow x.e$ can be applied to all records that have a field e);

extensible records: $\texttt{fun}\ x \rightarrow \texttt{fun}\ v \rightarrow x@\{e = v\}$ returns a record like x to which a field e containing v has been added.

Extensible records, reduction semantics

Let L be a finite set of labels.

Expressions:	$a ::= \ldots \mid \{(e = a_e)_{e \in L}\}$
Operators:	$op ::= \ldots \mid \texttt{proj}_e \mid \texttt{exten}_e$
Values:	$v ::= \ldots \mid \{(e = v_e)_{e \in L}\}$
Evaluation contexts:	$E ::= \ldots \mid \{(e = a_e)_{e \in L}; e = E\}$

Reduction rules:

$$\{(e = v_e)_{e \in L}\}.e \xrightarrow{\varepsilon} v_e \qquad \text{if } e \in L$$

$$\{(e = v_e)_{e \in L}\}@\{e' = w\} \xrightarrow{\varepsilon} \{(e = v'_e)_{e \in L \cup \{e'\}}\} \qquad \text{where } v' = v[e' \mapsto w]$$

Simplified type rules for extensible records

Idea: suppose that the set of labels is fixed, known, and reasonably small...

Types:

$$\begin{array}{rll} \tau ::= & \alpha \mid T \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 & \text{as before} \\ & \mid \{\mathtt{e} : \tau_1; \ \mathtt{f} : \tau_2; \ \mathtt{g} : \tau_3\} & \text{record type} \\ & \mid \mathtt{Abs} & \text{undefined} \\ & \mid \mathtt{Pre}\ \tau & \text{defined, with type } \tau \end{array}$$

Examples:

- $\{\mathtt{e} : \mathtt{Pre\ int};\ \mathtt{f} : \mathtt{Abs};\ \mathtt{g} : \mathtt{Abs}\}$: type of records with a field `e` of type `int`.
- $\{\mathtt{e} : \mathtt{Pre\ bool};\ \mathtt{f} : \mathtt{Abs};\ \mathtt{g} : \mathtt{Pre\ int}\}$: type of records with a field `e` of type `bool` and a field `g` of type `int`.

More examples

- $\{\mathtt{e} : \mathtt{Abs};\ \mathtt{f} : \mathtt{Abs};\ \mathtt{g} : \mathtt{Abs}\}$: type of the empty record.
- $\{\mathtt{e} : \alpha_1;\ \mathtt{f} : \alpha_2;\ \mathtt{g} : \alpha_3\} \to \{\mathtt{e} : \mathtt{Pre\ int};\ \mathtt{f} : \alpha_2;\ \mathtt{g} : \alpha_3\}$: type of a function that takes an arbitrary record, and extends it with a field `e` of type `int`.

Type rule

$$\frac{\forall e \in L \quad \Gamma \vdash a_e : \tau_e}{\Gamma \vdash \{(e = a_e)_{e \in L}\} : \{(e : \texttt{Pre}\ \tau_e)_{e \in L}; (e : \texttt{Abs})_{e \notin L}\}}$$

$$\begin{array}{rcl}
\texttt{proj}_\texttt{e} & : & \forall \alpha, \alpha_1, \alpha_2.\ \{\texttt{e} : \texttt{Pre}\ \alpha;\ \texttt{f} : \alpha_1;\ \texttt{g} : \alpha_2\} \to \alpha \\
\texttt{proj}_\texttt{f} & : & \forall \alpha, \alpha_1, \alpha_2.\ \{\texttt{e} : \alpha_1;\ \texttt{f} : \texttt{Pre}\ \alpha;\ \texttt{g} : \alpha_2\} \to \alpha \\
\texttt{proj}_\texttt{g} & : & \forall \alpha, \alpha_1, \alpha_2.\ \{\texttt{e} : \alpha_1;\ \texttt{f} : \alpha_2;\ \texttt{g} : \texttt{Pre}\ \alpha\} \to \alpha \\
\texttt{exten}_\texttt{e} & : & \forall \alpha, \alpha_1, \alpha_2, \alpha_3.\ \{\texttt{e} : \alpha_1;\ \texttt{f} : \alpha_2;\ \texttt{g} : \alpha_3\} \times \alpha \to \{\texttt{e} : \texttt{Pre}\ \alpha;\ \texttt{f} : \alpha_2;\ \texttt{g} : \alpha_3\} \\
\texttt{exten}_\texttt{f} & : & \forall \alpha, \alpha_1, \alpha_2, \alpha_3.\ \{\texttt{e} : \alpha_1;\ \texttt{f} : \alpha_2;\ \texttt{g} : \alpha_3\} \times \alpha \to \{\texttt{e} : \alpha_1;\ \texttt{f} : \texttt{Pre}\ \alpha;\ \texttt{g} : \alpha_3\} \\
\texttt{exten}_\texttt{g} & : & \forall \alpha, \alpha_1, \alpha_2, \alpha_3.\ \{\texttt{e} : \alpha_1;\ \texttt{f} : \alpha_2;\ \texttt{g} : \alpha_3\} \times \alpha \to \{\texttt{e} : \alpha_1;\ \texttt{f} : \alpha_2;\ \texttt{g} : \texttt{Pre}\ \alpha\}
\end{array}$$

Free extension vs. strict extension

Observe that $\texttt{exten}_{\mathsf{e}}$ can be used with the type

$$\{\texttt{e} : \texttt{Abs} \ldots\} \times \tau \to \{\texttt{e} : \texttt{Pre}\ \tau;\ \ldots\}$$

and with the type

$$\{\texttt{e} : \texttt{Pre}\ \tau' \ldots\} \times \tau \to \{\texttt{e} : \texttt{Pre}\ \tau;\ \ldots\}$$

In the first case, we extend the record with the label e, in the second, we replace the content of the field e.

If we want the *strict extension*, we must consider less polymorphic types as

$$\texttt{exten}_{\mathsf{e}} \quad : \quad \forall \alpha, \alpha_2, \alpha_3.\ \{\texttt{e} : \texttt{Abs};\ \texttt{f} : \alpha_2;\ \texttt{g} : \alpha_3\} \times \alpha \to \{\texttt{e} : \texttt{Pre}\ \alpha;\ \texttt{f} : \alpha_2;\ \texttt{g} : \alpha_3\}$$

Rows

Idea: add the concept of *model* for the fields that are not explicitely mentioned in the record type:

- $\partial\mathtt{Abs}$ to say that all other fields are absent;
- a variable α that represents an arbitrary set of presence informations.

Example: $\{e : \mathtt{Pre\ int};\ \partial\mathtt{Abs}\}$: type of records that have a field `e` of type `int`, and the other fields are absent.

Rows, formally

Types: $\tau ::= \alpha \mid T \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2$ as before
$\mid \{\tau\}$ records
$\mid \partial\texttt{Abs}$ empty row
$\mid e : \tau_1;\ \tau_2$ row containing $e : \tau_1$ and the row τ_2
$\mid \texttt{Abs}$ undefined
$\mid \texttt{Pre}\ \tau$ defined, with type τ

$$e_1 : \tau_1;\ e_2 : \tau_2;\ \tau \quad = \quad e_2 : \tau_2;\ e_1 : \tau_1;\ \tau \qquad \text{(commutativity)}$$

$$\partial\texttt{Abs} \quad = \quad e : \texttt{Abs};\ \partial\texttt{Abs} \qquad \text{(absorption)}$$

Example: these two types are equal:

$$\{e_1 : \texttt{Pre int};\ \partial\texttt{Abs}\} \text{ and } \{e_2 : \texttt{Abs};\ e_1 : \texttt{Pre int};\ \partial\texttt{Abs}\}$$

Type rules

$$\frac{\forall e \in L \quad \Gamma \vdash a_e : \tau_e}{\Gamma \vdash \{(e = a_e)_{e \in L}\} : \{(e : \texttt{Pre}\ \tau_e)_{e \in L};\ \partial\texttt{Abs}\}}$$

The schemas associated with the operators:

$$\begin{aligned}
\texttt{proj}_e &\ :\ \forall \alpha, \beta.\ \{e : \texttt{Pre}\ \alpha;\ \beta\} \to \alpha \\
\texttt{exten}_e &\ :\ \forall \alpha, \beta, \gamma.\ \{e : \alpha;\ \beta\} \times \gamma \to \{e : \texttt{Pre}\ \gamma;\ \beta\}
\end{aligned}$$

For the strict semantics:

$$\texttt{exten}_e \ :\ \forall \beta, \gamma.\ \{e : \texttt{Abs};\ \beta\} \times \gamma \to \{e : \texttt{Pre}\ \gamma;\ \beta\}$$

Meaningless types

In the type algebra we now have some meaningless types:

- $\partial\texttt{Abs} \to \partial\texttt{Abs}$ or $\texttt{Abs} \times \texttt{Pre}\ \tau$ or $\alpha \to \texttt{Pre}\ \alpha$;

We need some discipline to not mix:

- "normal" types, like $\texttt{int}$ ou $\texttt{int} \to \texttt{bool}$;
- rows, that can appear inside a record $\{\ldots\}$, like. $\partial\texttt{Abs}$ or $(e : \texttt{Abs};\ \ldots)$.
- the presence informations $\texttt{Abs}$ and $\texttt{Pre}\ \tau$, that can appear as annotations of a label in a row.

Other meaningless types

- $\{a : \texttt{Pre int};\ a : \texttt{Abs};\ \partial\texttt{Abs}\}$;
- $\{a : \texttt{Pre int};\ a : \texttt{Pre bool};\ \partial\texttt{Abs}\}$

We need a stronger invariant:

a label e cannot appear more than once in a given row.

It is difficult to prevent that a substitution of row variables breaks the invariant: the type $\tau = \{a : \texttt{Pre int};\ \rho\}$ satisfies the invariant, as the row $\varphi = a : \texttt{Pre bool};\ \partial\texttt{Abs}$. But the substitution $\tau[\rho \leftarrow \varphi]$ breaks the invariant.

Kinds

Kinds: $\kappa ::= \texttt{TYPE} \mid \texttt{PRE} \mid \texttt{R}(\{e_1, \ldots, e_n\})$

- $\texttt{TYPE}$ is the kind of well-formed types;
- $\texttt{PRE}$ is the kind of well-formed presence informations;
- $\texttt{R}(\{e_1, \ldots, e_n\})$ is the kind of well-formed rows that *do not* associate informations to the labels $e \in L$.

Kind rules

Let K be a function that associates to every variable α its kind.

$$\vdash \alpha :: K(\alpha) \qquad \vdash T :: \mathtt{TYPE} \qquad \frac{\vdash \tau_1 :: \mathtt{TYPE} \qquad \vdash \tau_2 :: \mathtt{TYPE}}{\vdash \tau_1 \longrightarrow \tau_2 :: \mathtt{TYPE}}$$

$$\frac{\vdash \tau_1 :: \mathtt{TYPE} \qquad \vdash \tau_2 :: \mathtt{TYPE}}{\vdash \tau_1 \times \tau_2 :: \mathtt{TYPE}} \qquad \frac{\vdash \tau :: \mathtt{R}(\emptyset)}{\vdash \{\tau\} :: \mathtt{TYPE}} \qquad \vdash \partial\mathtt{Abs} :: \mathtt{R}(L)$$

$$\frac{e \notin L \qquad \vdash \tau_1 :: \mathtt{PRE} \qquad \vdash \tau_2 :: \mathtt{R}(L \cup \{e\})}{\vdash (e : \tau_1;\ \tau_2) :: \mathtt{R}(L)}$$

$$\vdash \mathtt{Abs} :: \mathtt{PRE} \qquad \frac{\vdash \tau :: \mathtt{TYPE}}{\vdash \mathtt{Pre}\ \tau :: \mathtt{PRE}}$$

Example

Suppose a row τ defines twice the same label.

By commutativity we obtain

$$\tau = e : \tau_1;\ e : \tau_2;\ \tau'$$

As $\vdash \tau :: \mathtt{R}(L)$, it should hold

$$(e : \tau_2;\ \tau') :: \mathtt{R}(L \cup \{e\})$$

but this is impossible because $e \in L \cup \{e\}$.

Some care is required...

- A substitution θ preserves kinding if and only if for all variable α, it holds $\vdash \theta(\alpha) :: K(\alpha)$.

  It is easy to see that if θ preserves kinding, then $\vdash \tau :: \kappa$ implies $\vdash \theta(\tau) :: \kappa$.

- Every type scheme $\forall \vec{\alpha}.\ \tau$ must satisfy $\vdash \tau :: \mathtt{TYPE}$.

The safety proof follows by parametrising the proof for mini-ML with the new algebra of values and the new operators.

Type inference

We added an equational theory to a free algebra (the algebra of types), and this can radically change the nature and the properties of the unification problem.

Examples:

- $\partial\texttt{Abs}$ and $(e : \alpha;\ \beta)$ can be unified by taking $\alpha \leftarrow \texttt{Abs}$, $\beta \leftarrow \partial\texttt{Abs}$, and by using the absorption axiom.
- the types $e : \texttt{Pre int}; \alpha$ and $f : \texttt{Pre bool}; \beta$ can be unified by the substitution

$$\alpha \leftarrow f : \texttt{Pre bool};\ \tau \qquad \beta \leftarrow e : \texttt{Pre int};\ \tau$$

where τ is an arbitrary type.

Unification algorithm

$$
\begin{array}{rcl}
\texttt{mgu}(\emptyset) & = & id \\
\texttt{mgu}(\{\alpha \stackrel{?}{=} \alpha\} \cup C & = & \texttt{mgu}(C) \\
\texttt{mgu}(\{\alpha \stackrel{?}{=} \tau\} \cup C) & = & \texttt{mgu}(C[\alpha \leftarrow \tau]) \circ [\alpha \leftarrow \tau] \text{ if } \alpha \text{ is not free in } \tau \\
\texttt{mgu}(\{\tau \stackrel{?}{=} \alpha\} \cup C) & = & \texttt{mgu}(C[\alpha \leftarrow \tau]) \circ [\alpha \leftarrow \tau] \text{ if } \alpha \text{ is not free in } \tau \\
\texttt{mgu}(\{\tau_1 \rightarrow \tau_2 \stackrel{?}{=} \tau_1' \rightarrow \tau_2'\} \cup C) & = & \texttt{mgu}(\{\tau_1 \stackrel{?}{=} \tau_1'; \tau_2 \stackrel{?}{=} \tau_2'\} \cup C) \\
\texttt{mgu}(\{\tau_1 \times \tau_2 \stackrel{?}{=} \tau_1' \times \tau_2'\} \cup C) & = & \texttt{mgu}(\{\tau_1 \stackrel{?}{=} \tau_1'; \tau_2 \stackrel{?}{=} \tau_2'\} \cup C) \\
\texttt{mgu}(\{\ \{\tau_1\} \stackrel{?}{=} \{\tau_2\}\ \} \cup C) & = & \texttt{mgu}(\{\tau_1 \stackrel{?}{=} \tau_2\} \cup C) \\
\texttt{mgu}(\{\texttt{Abs} \stackrel{?}{=} \texttt{Abs}\} \cup C) & = & \texttt{mgu}(C) \\
\texttt{mgu}(\{\texttt{Pre}\ \tau_1 \stackrel{?}{=} \texttt{Pre}\ \tau'\} \cup C) & = & \texttt{mgu}(\{\tau_1 \stackrel{?}{=} \tau_2\} \cup C)
\end{array}
$$

Unification, ctd.

$$
\begin{array}{rcl}
\texttt{mgu}(\{\partial\texttt{Abs} \stackrel{?}{=} \partial\texttt{Abs}\} \cup C) & = & \texttt{mgu}(C) \\
\texttt{mgu}(\{e : \tau;\ \tau' \stackrel{?}{=} \partial\texttt{Abs}\} \cup C) & = & \texttt{mgu}(\{\tau \stackrel{?}{=} \texttt{Abs};\ \tau' \stackrel{?}{=} \partial\texttt{Abs}\} \cup C) \\
\texttt{mgu}(\{\partial\texttt{Abs} \stackrel{?}{=} e : \tau;\ \tau'\} \cup C) & = & \texttt{mgu}(\{\tau \stackrel{?}{=} \texttt{Abs};\ \tau' \stackrel{?}{=} \partial\texttt{Abs}\} \cup C) \\
\texttt{mgu}(\{e : \tau_1;\ \tau'_1 \stackrel{?}{=} e : \tau_2;\ \tau'_2\} \cup C) & = & \texttt{mgu}(\{\tau_1 \stackrel{?}{=} \tau_2;\ \tau'_1 \stackrel{?}{=} \tau'_2\} \cup C) \\
\texttt{mgu}(\{(e : \tau_1;\ \tau'_1) \stackrel{?}{=} (f : \tau_2;\ \tau'_2)\} \cup C) & = & \texttt{do}\ \alpha = \texttt{fresh} \\
(e \neq f) & & \texttt{mgu}(\{\tau'_1 \stackrel{?}{=} (f : \tau_2;\ \alpha); \\
& & \qquad \tau'_2 \stackrel{?}{=} (e : \tau_1;\ \alpha)\} \cup C)
\end{array}
$$

Modiffying the W algorithm to take into account extensible records is easy.

Row polymorphism in OCaml

Some (weird?) syntactic sugar:

```
let o =                       let o =
  {                             object
    x = 3;                         method x = 3
    y = "foo";                     method y = "string"
  }                             end
```

OCaml answers:

```
  val o : < x : int; y : string >
```

Observe that the ∂`Abs` annotation is ommitted in the row.

Row polymorphism, ctd.

The polymorphic projection function

```
let f = fun z -> z#x
```

can be associated with a schema

$$\forall \alpha \beta . \langle z : \texttt{Pre}\ \alpha ;\ \beta \rangle \to \alpha$$

written by OCaml as (the `..` stand for a type variable in the row)

```
val f : < x : 'a; .. > -> 'a
```

The “row polymorphism” comes from the fact that, when typing function application, the variable β can be unified with an arbitrary row.

A simple object calculus (without classes)

Idea: an object can be seen as a polymorphic record[2], each field corresponds to a method of the object; use auto-application to implement the `self` parameter (*self-application semantics*).

Expressions:	$a ::= \ldots \mid \texttt{obj}(x)\langle(m = a_m)_{m\in M}\rangle$	object construction
Oprateurs:	$op ::= \ldots \mid \#m$	method selection
Values:	$v ::= \ldots \mid \texttt{obj}(x)\langle(m = a_m)_{m\in M}\rangle$	

Reduction rules:

$$v\#m \quad \xrightarrow{\varepsilon} \quad a_m[x \leftarrow v] \quad \text{if } v = \texttt{obj}(x)\langle(m = a_m)_{m\in M}\rangle$$

[2]This explains OCaml syntax for polymorphic records.

Example

We can encode recursive functions using the self-application semantics:

```
let o = obj(s)
        < method fact = fun n →
                              if n = 0 then 1 else n * s#fact (n-1) >
in o#fact 5
```

reduces to

```
5 * (
    (obj(s)
       < method fact = fun n →
                              if n = 0 then 1 else n * s#fact (n-1) >) # 4)
```

Types for simple objects

Idea: use polymorphic record types...

Types:	$\tau ::= \alpha \mid T \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2$	as before
	$\mid \langle \tau \rangle$	type of objects
	$\mid \partial\texttt{Abs}$	empty row
	$\mid m : \tau_1;\ \tau_2$	row containing $m : \tau_1$, and τ_2
	$\mid \texttt{Abs}$	undefined
	$\mid \texttt{Pre}\ \tau$	defined, with type τ

$$\frac{\tau = \langle (m : \texttt{Pre}\ \tau_m)_{m \in M};\ \partial\texttt{Abs} \rangle \qquad \forall m \in M \quad \Gamma; x : \tau \vdash a_m : \tau_m}{\Gamma \vdash \texttt{obj}(x)\langle (m = a_m)_{m \in M} \rangle : \tau}$$

$$\#m \quad : \quad \forall \alpha, \beta.\ \langle m : \texttt{Pre}\ \alpha;\ \beta \rangle \to \alpha$$

Example

The type

$$\langle m : \texttt{Pre int};\ \alpha \rangle$$

is the type of the objects with a method m returning an integer, and possibly other methods. Such "open" types arise naturally for the function parameters:

```
fun obj → 1 + obj#m
```

can be associated to the type schema

$$\forall \alpha. \langle m : \texttt{Pre int};\ \alpha \rangle \rightarrow \texttt{int}$$

How to forget methods

If

$$a : \langle \mathtt{m} : \mathtt{Pre\ int};\ \partial\mathtt{Abs} \rangle$$

and

$$b : \langle \mathtt{m} : \mathtt{Pre\ int};\ \mathtt{n} : \mathtt{Pre\ string};\ \partial\mathtt{Abs} \rangle$$

the expression

$$\mathtt{if}\ cond\ \mathtt{then}\ a\ \mathtt{else}\ b$$

cannot be typed. In particular, it does not have the “natural” type

$$\langle \mathtt{m} : \mathtt{Pre\ int};\ \partial\mathtt{Abs} \rangle$$

“natural” type

Can we formalise this idea of *“natural” type*?

The term

$$b : \langle \mathtt{m} : \mathtt{Pre\ int};\ \mathtt{n} : \mathtt{Pre\ string};\ \partial\mathtt{Abs} \rangle$$

can be used in all contexts where a term of type

$$\langle \mathtt{m} : \mathtt{Pre\ int};\ \partial\mathtt{Abs} \rangle$$

is expected (these contexts will never call the method $\mathtt{n}$).

We can relax the typing relation, and say that b can be seen as a term that has the type $\langle \mathtt{m} : \mathtt{Pre\ int};\ \partial\mathtt{Abs} \rangle$.

The subtyping relation

The subtyping relation $<:$ specifies which types can be *seen as* other types.

The two key rules are

$$\tau <: \partial\texttt{Abs} \qquad\qquad \frac{\varphi <: \varphi'}{(m : \tau;\ \varphi) <: (m : \tau;\ \varphi')}$$

The subtyping relation lifts to all the other types in the natural way, for instance

$$\frac{\tau <: \tau' \qquad \varphi <: \varphi'}{(\tau \times \varphi) <: (\tau' \times \varphi')}$$

but some care is required with function types.

Subtyping function types

When is it safe to pass a function of one type in a context where a different function type is expected?

$$\frac{\tau' <: \tau \qquad \varphi <: \varphi'}{(\tau \to \varphi) <: (\tau' \to \varphi')}$$

Intuition: we have a function f of type $\tau \to \varphi$, and a context that expects a function of type $\tau' \to \varphi'$.

- if $\tau' <: \tau$, then none of values passed by the context to the function will surprise it;
- if $\varphi <: \varphi'$, then none of the values returned by the function will surprise the context.

The subtyping relation, formally

$$T <: T \qquad \alpha <: \alpha \qquad \frac{\tau' <: \tau \qquad \varphi <: \varphi'}{(\tau \to \varphi) <: (\tau' \to \varphi')} \qquad \frac{\tau <: \tau' \qquad \varphi <: \varphi'}{(\tau \times \varphi) <: (\tau' \times \varphi')}$$

$$\frac{\tau <: \tau'}{\langle \tau \rangle <: \langle \tau' \rangle} \qquad \tau <: \partial\texttt{Abs} \qquad \frac{\tau <: \tau' \qquad \varphi <: \varphi'}{(m : \tau;\ \varphi) <: (m : \tau';\ \varphi')}$$

$$\texttt{Abs} <: \texttt{Abs} \qquad \frac{\tau <: \tau'}{\texttt{Pre}\ \tau <: \texttt{Pre}\ \tau'}$$

Examples

$$
\begin{array}{rcl}
\langle m : \texttt{Pre int};\ \partial\texttt{Abs}\rangle & <: & \langle \partial\texttt{Abs}\rangle \\
\langle o : \langle m : \texttt{Pre int};\ \partial\texttt{Abs}\rangle\rangle & <: & \langle o : \langle \partial\texttt{Abs}\rangle\rangle \\
\langle m : \texttt{Pre int};\ \alpha\rangle & \not<: & \langle \alpha\rangle \\
\texttt{int} \rightarrow \langle m : \texttt{Pre int};\ \partial\texttt{Abs}\rangle & <: & \texttt{int} \rightarrow \langle \partial\texttt{Abs}\rangle \\
\langle m : \texttt{Pre int};\ \partial\texttt{Abs}\rangle \rightarrow \texttt{int} & \not<: & \langle \partial\texttt{Abs}\rangle \rightarrow \texttt{int} \\
\langle \partial\texttt{Abs}\rangle \rightarrow \texttt{int} & <: & \langle m : \texttt{Pre int};\ \partial\texttt{Abs}\rangle \rightarrow \texttt{int}
\end{array}
$$

Explicit subtyping

Idea: add an explicit operator to see the object type τ as a "supertype" τ'.

Operators: $op ::= \texttt{coerce}_{\tau,\tau'}$ for all τ, τ' such that $\tau <: \tau'$

Type rule:

$$\texttt{coerce}_{\tau,\tau'} : \forall\vec{\alpha}.\ \tau \to \tau' \quad \text{if } \tau <: \tau' \text{ and } \vec{\alpha} = \mathcal{L}(\tau) \cup \mathcal{L}(\tau')$$

Reduction rule:

$$\texttt{coerce}_{\tau,\tau'}(v) \ \xrightarrow{\varepsilon}\ v$$

Example

Let $\tau_1 = \langle \mathtt{m} : \mathtt{Pre}\ \mathtt{int};\ \partial\mathtt{Abs}\rangle$ and $\tau_2 = \langle \mathtt{m} : \mathtt{Pre}\ \mathtt{int};\ \mathtt{n} : \mathtt{Pre}\ \mathtt{string};\ \partial\mathtt{Abs}\rangle$. Let $a : \tau_1$ and let $b : \tau_2$. The expression

$$\mathtt{if}\ cond\ \mathtt{then}\ a\ \mathtt{else}\ (\mathtt{coerce}_{\tau_2,\tau_1}\ b)$$

is now well typed, with type

$$\langle \mathtt{m} : \mathtt{Pre}\ \mathtt{int};\ \partial\mathtt{Abs}\rangle$$

Implict subtyping

Idea: add the *subsumption* rule:

$$\frac{\Gamma \vdash a : \tau \qquad \tau <: \tau'}{\Gamma \vdash a : \tau'} \text{ (sub)}$$

(Observe that to prove the subject reduction of the simple object calculus we need this rule in the type system.)

*Q*uestion: can we get rid of row polymorphism and use subtyping on records?

*A*nswer: yes, but type inference is now undecidable (see all the papers on *local type inference*).

A simple object calculus with classes

Idea: classes are stamps for objects.

Expressions:

$a ::= \ldots$	
$\mid \texttt{new}$	creation of an object from a class
$\mid \texttt{class}(x)\langle (m = a_m)_{m \in M} \rangle$	class
$\mid \texttt{class}(x)\langle \texttt{inherit}\ a;\ m = a \rangle$	inheritance, (re)definition of a method

(x is bound in the method body, but not in the inherit clause.)

Values: $v ::= \ldots \mid \texttt{class}(x)\langle (m = a_m)_{m \in M} \rangle$

Reduction semantics for classes

Evaluation contexts: $E ::= \ldots \mid \texttt{new}\ E \mid \texttt{class}(x)\langle\texttt{inherit}\ E;\ m = a\rangle$

Reduction rules:

$$\texttt{new class}(x)\langle(m = a_m)_{m\in M}\rangle \xrightarrow{\varepsilon} \texttt{obj}(x)\langle(m = a_m)_{m\in M}\rangle$$

$$\texttt{class}(x)\langle\texttt{inherit class}(x)\langle(m = a_m)_{m\in M}\rangle);\ n = b\rangle \xrightarrow{\varepsilon}$$

$$\texttt{class}(x)\langle(m = a'_m)_{m\in M\cup\{n\}}\rangle$$

where $a' = a[n \mapsto b]$

Types for classes

Types: $\tau ::= \ldots \mid \texttt{class}(\tau_1)\ \tau_2$ type of a class

We record two types:

- τ_1 is the type of the parameter `self`,
- τ_2 is an object type, representing the types of the methods defined in the class.

Quite often these will coincide, unless we added an explicit type constraint on τ_1. Why do we want to do so?

Virtual classes

A *virtual class* defines some "default methods", and relies on classes inheriting it to provide the other methods. Using OCaml syntax, we can define:

```
class virtual c =
  object(self)
    method virtual m : int
    method n = 1 + self#m
  end
```

has type

$$\mathtt{class}(\langle \mathtt{m} : \mathtt{Pre\ int};\ \mathtt{n} : \mathtt{Pre\ int};\ \alpha\rangle) \quad \langle \mathtt{n} : \mathtt{Pre\ int};\ \partial\mathtt{Abs}\rangle$$

Before creating an object of this class with `new`, we must define an implementation of the method m, by inheriting the class and defining the method.

Type rules

$$\frac{\Gamma \vdash a : \texttt{class}(\tau)\ \tau}{\Gamma \vdash \texttt{new}\ a : \tau}\ \text{(new)}$$

$$\frac{\tau = \langle (m : \texttt{Pre}\ \tau_m)_{m \in M};\ \tau' \rangle \qquad \forall m \in M \quad \Gamma; x : \tau \vdash a_m : \tau_m}{\Gamma \vdash \texttt{class}(x)\langle (m = a_m)_{m \in M} \rangle : \texttt{class}(\tau)\ \langle (m : \texttt{Pre}\ \tau_m)_{m \in M};\ \partial\texttt{Abs} \rangle}\ \text{(class)}$$

$$\frac{\Gamma \vdash a : \texttt{class}(\tau)\ \langle m : \tau_m^0;\ \tau' \rangle \qquad \Gamma; x : \tau \vdash b : \tau_m \qquad \tau = \langle m : \texttt{Pre}\ \tau_m;\ \tau'' \rangle}{\Gamma \vdash \texttt{class}(x)\langle \texttt{inherit}\ a;\ m = b \rangle : \texttt{class}(\tau)\ \langle m : \texttt{Pre}\ \tau_m;\ \tau' \rangle}\ \text{(inherit)}$$

Inheritance is not subtyping

If a class A is defined by inheritance from a class B, then the type of the objects of the class B is *sometimes, but not always,* a subtype of the type od the obects of the class A.

W. R. Cook, W. L. Hill, and P. S. Canning. *Inheritance is not subtyping*. ACM Press, Proceedings of POPL'90.

Inheritance is not subtyping, ctd.

```
class point =
  object (self: 'selftype)
    val x = 0                 method coord = x
    method equal (p : 'selftype) = (p#coord = x)
  end
class colorpoint =
  object (self: 'selftype)
    inherit point as super
    val c = "black"           method colour = c
    method equal (p : 'selftype) = (p#coord = x) && (p#colour = c)
  end
```

`colorpoint` must not be a subtype of `point`; otherwise the wrong code below would pass the typecheck:

```
let p = new point and cp = (new colorpoint :> point) in cp#equal p
```

Concrete types in an object soup

```
class virtual ['a] list =
  object (self)
    method virtual isnil : bool      method virtual tail : 'a list
    method virtual head : 'a
    method length = if self#isnil then 0 else 1 + self#tail#length
  end
```

```
class ['a] nil =
  object
    inherit ['a] list
    method isnil = true
    method head = failwith "nil"
    method tail = failwith "nil"
  end
```

```
class ['a] cons h0 t0 =
  object
    val h = h0
    val t = t0
    inherit ['a] list
    method isnil = false
    method head = h
    method tail = t
  end
```

The `Marshal` module

OCaml standard library defines a `Marshal` module. Its signature looks like:

```
sig
  val to_string : 'a -> string
  val from_string : string -> 'a
  [...]
end
```

Idea: the function `to_string` converts an arbitrary value into a sequence of bytes (which can then be written on file, sent over a network connection,...). The function `from_string` converts a sequence of bytes back into a value.

The `Marshal` module is unsafe

Suppose

```
Network.send : string -> unit
Network.receive : unit -> string
```

Consider these two programs, running on different machines:

```
program A:
  let x = 5
  in Network.send (Marshal.to_string x)
program B:
  let y = Marshal.from_string (Network.receive())
  in print_bool y
end
```

Both programs are well-typed, but executing them will raise a run-time error.

Dynamic types

Idea: send the type of the value together with its byte representation (eg, (v, τ)).

Add a new type `dyn`, that represents pairs of values together with their type.

Operators:

$$
\begin{array}{rcll}
\texttt{dyn}_\tau & : & \tau \longrightarrow \texttt{dyn} & \text{if } \tau \text{ is a type without type variables} \\
\texttt{hastype}_\tau & : & \texttt{dyn} \rightarrow \texttt{bool} & \\
\texttt{coerce}_\tau & : & \texttt{dyn} \rightarrow \tau & \text{if } \tau \text{ is a type without type variables}
\end{array}
$$

Example:

```
fun d →
  if hastype string (d) then print_string(coerce string(d))
  else if hastype int(d) then print_int(coerce int(d))
  else print_string "???"
```

Dynamic types, ctd.

Values: $v ::= \ldots \mid \texttt{dyn}_\tau(v)$

Reduction rules:

$$
\begin{array}{rcl}
\texttt{hastype}_\tau(\texttt{dyn}_{\tau'}(v)) & \xrightarrow{\varepsilon} & \texttt{true} \quad \text{if } \tau = \tau' \\
\texttt{hastype}_\tau(\texttt{dyn}_{\tau'}(v)) & \xrightarrow{\varepsilon} & \texttt{false} \quad \text{if } \tau \neq \tau' \\
\texttt{coerce}_\tau(\texttt{dyn}_{\tau'}(v)) & \xrightarrow{\varepsilon} & v \quad \text{if } \tau = \tau'
\end{array}
$$

Exercises

1. Prove that the reduction rules above respect the hypothesis **H1**, that is, show that if $E \vdash a : \tau$ and $a \xrightarrow{\varepsilon} a'$ using one the rules above, then $E \vdash a' : \tau$.

2. Prove hypothesis **H2** for the operator $\mathtt{hastype}_\tau$, that is, show that if $\emptyset \vdash \mathtt{hastype}_\tau(v) : \tau'$, then the term $\mathtt{hastype}_\tau(v)$ can be reduced.

3. Does the operator $\mathtt{coerce}_\tau$ preserve hypothesis **H2**? If yes, prove it. If not, give a counter-example, and suggest a way to solve this problem (add some reduction rules, or propose another operator that satisfies **H2** and has the same expressive power as $\mathtt{coerce}_\tau$).

4. Show that if the condition that the type τ in $\mathtt{coerce}_\tau$ and $\mathtt{hastype}_\tau$ must not contain type variables is removed, the language is not safe (hint, show that **H1** does not hold).

What we covered

- A simple higher-order call-by-value language, called *mini-ML*;
- monomorphic type system, type inference;
- polymorphic type system, importance of `let`, algorithm W;
- proof of safety of the polymorphic type system;
- simple extensions: tuples, sums, algebraic data types;
- imperative programming: references, exceptions;
- polymorphic records, a (simple) object system, subtyping.

- OCaml modules.

Key ideas

- The idea of *safe language*;
- type vs. type schemas, generalisation of type variables;
- type inference as unification of equations;
- compromise between expressiveness, feasibility of type inference, and simplicity of use;
- the polymorphic reference problem;
- row polymorphism vs. subtyping.

What we did not cover

...too many things.