A Recursive Type System with Type Abbreviations and Abstract Types

Keiko Nakata
Institute of Cybernetics, Tallinn
Joint work with Hyeonseung Im and Sungwoo Park

18 May 2014, Narva-Jõesuu
The ML module system supports program structuring, code reuse and representation independence (implementation hiding) with

- nested structures,
- functoros, and
- signatures (with abstract types).
module type Comparable = sig
  type t
  val compare : t → t → int
end

module Make_interval(Endpoint : Comparable) = struct
  module E = Endpoint
  type t = Interval of E.t * E.t | Empty
  let create low high = ...
  let is_empty = function Empty → true | Interval _ → false
  let contains t x = match t with | Empty → false
    | Interval (l,h) → E.compare x l ≥ 0 and E.compare x h ≤ 0
  let intersect t1 t2 = ...
end
Instantiating Make_interval

```ocaml
module Int_interval = Make_interval(struct type t = int let compare = Int.compare end)
let i1 = Int_interval.create 3 8
module Rev_int_interval = Make_interval(struct
    type t = int let compare x y = Int.compare y x end)

(* Int_interval.t ≠ Rev_int_interval.t *)
let rev_interval = Rev_int_interval.create 4 3
Int_interval.contains rev_interval 3
```
Constraining the result type of functors

```ocaml
module type Interval_intf = sig
  type t
  type endpoint
  val create : endpoint → endpoint → t
  val is_empty : t → bool
  val contains : t → endpoint → bool
  val intersect : t → t → t
end

module Make_interval(Endpoint : Comparable)
  : (Interval_intf with type endpoint = Endpoint.t)
  = struct
    module E = Endpoint
    type endpoint = E.t
    type t = Interval of E.t * E.t | Empty
    ...
  end
```
Polymorphic recursive types

are equi-recursive types:

- $\mu X.\tau$ is equal to its one-step unfolding $\{X \mapsto \mu X.\tau\}\tau$.
- Equivalence of equi-recursive types is structural.

Principle type inference is available.

Code reuse is archived with structural polymorphism.

The module machinery can be combined with polymorphic recursive types, e.g., private types.
Contractiveness

A type variable $X$ is *contractive* in $\tau$, if $X$ occurs in $\tau$ only under a type constructor.

A recursive type is *contractive* if every recursive variable is contractive in its scope.

In a simple type language, contractiveness can be enforced syntactically

$$\tau, \sigma ::= X \mid \tau \rightarrow \sigma \mid \mu X.(\tau \rightarrow \sigma)$$

Contractiveness guarantees the unique solution of recursive equations introduced by equi-recursive types.
Contractiveness in OCaml

In an advanced type system, such as in OCaml...

Syntactic contractiveness is not sufficient:

```ocaml
type 'a t = ['A of 'a | 'B];;
type s = s t;;
```

We may not be able to know (without breaking type abstraction).

```ocaml
module rec M : sig type t end =
    struct type t = N.t end
and N : sig type t end =
    struct type t = M.t end
```
Our work

A equi-recursive type system with type abbreviations and abstract types.

We allow non-contractive types in the implementation, but disallow them in the signature.

The type system is proved sound, formalized in Coq.
Non-contractive types in the signature

```
module M : S = struct
  module type S = sig
    type 'a t = 'a
    type u = int and v = bool
    let f x = x
    let g x = x
  end
  let h x = M.g (M.f x)
  let y = h 3 (* run-time error *)
end
```

We found the bug together with Jacques Garrigue, which has been fixed in the latest release OCaml 4.00.1.
Type language

type constructor \( s, t, u \)
type \( \tau, \sigma \) ::= unit
     \( \alpha | \beta | \gamma \)
     \( \tau \rightarrow \sigma \)
     \( \tau_1 \ast \tau_2 \)
     \( \tau \ t \)
type abbreviation \( D \) ::= type \( \alpha \ t = \tau \)
type variable set \( \Sigma \) ::= \( \cdot \) | \( \Delta, D \)
Expression language

value

\[ v ::= () | \lambda a : \tau. e | (v_1, v_2) \]

term

\[ e ::= () | a | x | \lambda a : \tau. e | e_1 \ e_2 | (e_1, e_2) | l \]

\[ \quad | \text{fst} \ e | \text{snd} \ e | \text{fix} \ a : \tau. e \]

value context

\[ \Gamma ::= \cdot | \Gamma, x : \tau \]
## Module language

<table>
<thead>
<tr>
<th>Specification</th>
<th>Definition</th>
<th>Signature</th>
<th>Structure</th>
<th>Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>( D \ ::= )</td>
<td>( d_{τ} \ ::= )</td>
<td>( S \ ::= )</td>
<td>( M \ ::= )</td>
<td>( P \ ::= )</td>
</tr>
<tr>
<td>( \text{type } α \ t )</td>
<td>( \text{type } α \ t = τ )</td>
<td>( \cdot \mid S, D )</td>
<td>( (d_{τ}, d_{e}) )</td>
<td>( (M, S, e) )</td>
</tr>
<tr>
<td>(</td>
<td>)</td>
<td>( \mid )</td>
<td>( \mid )</td>
<td>(</td>
</tr>
<tr>
<td>( \text{abstract type} )</td>
<td>( \text{type equation} )</td>
<td>( \text{value specification} )</td>
<td>( )</td>
<td>( \mid )</td>
</tr>
<tr>
<td>( \mid )</td>
<td>( \text{value definition} )</td>
<td>( \text{value definition} )</td>
<td>( )</td>
<td>( )</td>
</tr>
</tbody>
</table>

---

**Example:**

- **Specification:**
  - \( D \ ::= \text{type } α \ t \)
  - \( D \ ::= \text{type } α \ t = τ \)
  - \( D \ ::= \text{val } l : τ \)

- **Definition:**
  - \( d_{τ} \ ::= \text{type } α \ t = τ \)
  - \( d_{e} \ ::= \text{let } l = e \)

- **Signature:**
  - \( S \ ::= \cdot \mid S, D \)

- **Structure:**
  - \( M \ ::= (d_{τ}, d_{e}) \)

- **Program:**
  - \( P \ ::= (M, S, e) \)
  - \( P \ ::= (M, e) \)
Type equivalence

The judgment $S \vdash \tau \rightarrow \sigma$ states that type $\tau$ unfolds into $\sigma$ by expanding a type name in $\tau$ into its definition under $S$.

$$\Delta \ni \text{type } \alpha \; t = \sigma \quad \frac{\Delta \vdash \tau \; t \rightarrow \{\alpha \mapsto \tau\}_\sigma}{\text{unfold}}$$
Type equivalence

Inductive type equivalence

\[ \Delta; \Sigma \vdash \tau_1 R \tau_2 \]

\[ \Delta; \Sigma \vdash \text{unit} R \text{unit} \]

\[ \Delta; \Sigma \vdash \alpha R \alpha \]

\[ \Delta; \Sigma \vdash \tau_i R \sigma_i \quad (i = 1, 2) \]

\[ \Delta; \Sigma \vdash \tau_1 \to \tau_2 R \sigma_1 \to \sigma_2 \]

\[ \Delta; \Sigma \vdash \tau_i R \sigma_i \quad (i = 1, 2) \]

\[ \Delta; \Sigma \vdash \tau_1 \ast \tau_2 R \sigma_1 \ast \sigma_2 \]

\[ S \ni \text{type } \alpha \ t \]

\[ S; \Sigma \vdash \tau R \sigma \]

\[ S; \Sigma \vdash \tau R \sigma \]

\[ S; \Sigma \vdash \tau \ t \]

\[ S; \Sigma \vdash \tau \ t \]

\[ \Delta \vdash \tau \to \tau' \]

\[ \Delta; \Sigma \vdash \tau' \ R \sigma \]

\[ \Delta \vdash \sigma \to \sigma' \]

\[ \Delta; \Sigma \vdash \tau \ R \sigma' \]

\[ \Delta; \Sigma \vdash \tau \ R \sigma' \]

\[ \Delta; \Sigma \vdash \tau \ R \sigma \]

\[ \Delta; \Sigma \vdash \tau \ R \sigma \]

\[ \Delta; \Sigma \vdash \tau \ R \sigma \]
Type equivalence

Inductive type equivalence

\[ \Delta; \Sigma \vdash \tau_1 \overset R \equiv \tau_2 \]

\[ \Delta; \Sigma \vdash \text{unit} \overset R \equiv \text{unit} \]  \[ \text{eq-unit} \]

\[ \alpha \in \Sigma \]  \[ \Delta; \Sigma \vdash \alpha \overset R \equiv \alpha \]  \[ \text{eq-var} \]

\[ \Delta; \Sigma \vdash \tau_i \overset R \equiv \sigma_i \quad (i = 1, 2) \]  \[ \text{eq-fun} \]

\[ \Delta; \Sigma \vdash \tau_1 \rightarrow \tau_2 \overset R \equiv \sigma_1 \rightarrow \sigma_2 \]  \[ \text{eq-prod} \]

\[ S \ni \text{type} \quad \alpha \quad t \]

\[ S; \Sigma \vdash \tau \overset R \equiv \sigma \quad t \]

\[ S; \Sigma \vdash \tau \overset R \equiv \sigma \quad t \]

\[ \Delta; \Sigma \vdash \tau \overset R \equiv \sigma \]

\[ \Delta; \Sigma \vdash \tau \overset R \equiv \sigma \]

\[ \Delta; \Sigma \vdash \tau' \overset R \equiv \sigma' \]

\[ \Delta; \Sigma \vdash \tau \overset R \equiv \sigma' \]

\[ \Delta; \Sigma \vdash \tau \overset R \equiv \sigma \]  \[ \text{eq-abs} \]

\[ \text{eq-lunfold} \]

\[ \text{eq-runfold} \]
Type equivalence

Coinductive type equivalence

\[ \Delta; \Sigma \vdash \tau \equiv \sigma \quad \Delta; \Sigma \vdash \tau \text{ type} \quad \Delta; \Sigma \vdash \sigma \text{ type} \]
\[ \Delta; \Sigma \vdash \tau \equiv \sigma \quad \text{eq-ind} \]

\[ \Delta; \Sigma \vdash \tau \text{ type} \quad \Delta; \Sigma \vdash \sigma \text{ type} \quad \Delta \vdash \tau \rightarrow \tau' \quad \Delta \vdash \sigma \rightarrow \sigma' \quad \Delta; \Sigma \vdash \tau' \equiv \sigma' \]
\[ \Delta; \Sigma \vdash \tau \equiv \sigma \quad \text{eq-coind} \]
Contractive types and signatures

\[ S \downarrow\downarrow \tau \quad \text{ctr-coind} \]
\[ S \downarrow \tau \]

\[ (S, \tau) \in C \quad (S, \sigma) \in C \]
\[ S \downarrow_C \tau \rightarrow \sigma \quad \text{ctr-fun} \]

\[ S \ni \text{type } \alpha \ t \quad S \downarrow_C \tau \]
\[ S \downarrow_C \tau \ t \quad \text{ctr-abs} \]

\[ BN(S) \text{ distinct} \quad \forall (\text{type } \alpha \ t = \tau) \in S, \ S \downarrow \tau \]
\[ S \downarrow \quad \text{ctr-sig} \]

\[ S \downarrow_C \alpha \quad \text{ctr-var} \]
\[ S \downarrow_C \text{unit} \quad \text{ctr-unit} \]
\[ S \downarrow_C \tau_1 \ast \tau_2 \quad \text{ctr-prod} \]
\[ S \downarrow_C \tau \rightarrow \sigma \quad \text{ctr-type} \]
Type soundness of $\lambda_{\text{abs}}^{\text{rec}}$

The key lemma in the soundness states that a well-formed type is contractive:

Lemma

Suppose $S \text{ ok}$, $S \Downarrow$, and $S; \Sigma \vdash \tau$ type. Then $S \Downarrow \tau$.

which enables us to prove that type equivalence is preserved by signature elimination:

Lemma

If $S_1 \leq S_2$, $S_2 \Downarrow$, and $S_2; \Sigma \vdash \tau \equiv \sigma$, then $S_1; \Sigma \vdash \tau \equiv \sigma$.

Theorem

The type system for $\lambda_{\text{abs}}^{\text{rec}}$ is sound.

(We prove the progress and preservation properties.)