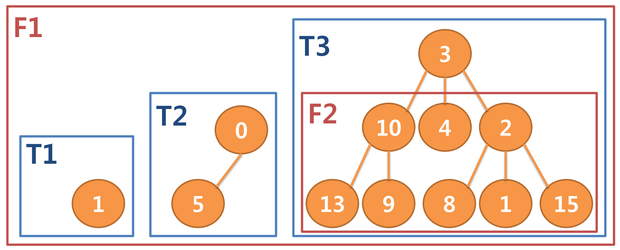
A syntactic type system for recursive modules

Presented by Hyeonseung Im

Joint work with Jacques Garrigue, Keiko Nakata, and Sungwoo Park

ROSAEC Center Workshop, 25-28 August 2010

Tree-Forest recursive data structures



Basic operations:

Tree.max T1 = 1

Tree.max T3 = Tree.max (3, Forest.max F2)

Forest.max F1 = max (Tree.max T1, Tree.max T2, Tree.max T3)

Extensions:

Tree.mk_tree F1 =

Forest.combine T1 T2 = Tree.mk_tree

OCaml implementation using recursive modules

```
module rec Tree : sig
  type t
  val max : t -> int
end = struct
  type t = Leaf of int
         | Node of int * Forest.t
  let max x = match x with
    | Leaf i -> i
    | Node (i, f) ->
      let j = Forest.max f in
        if i > j then i else j
end
and Forest : sig
  type t
  val max : t -> int
end = struct
  type t = Tree.t list
  let rec max x = match x with
    | [] -> 0
    | hd :: tl ->
      let i = Tree.max hd in
      let j = max tl in
        if i > j then i else j
end
```

The double vision problem

```
module rec Tree : sig
  type t
  val max : t -> int
  val mk_tree : Forest.t -> t
end = struct
  type t = Leaf of int
         | Node of int * Forest.t
  let max x = ...
  let mk_tree x =
    let i = Forest.max x in Node (i, x)
end
and Forest : sig
  type t
  val max : t -> int
  val combine : Tree.t -> Tree.t -> Tree.t
end = struct
  type t = Tree.t list
  let rec max x = ...
  let combine x y = Tree.mk_tree [x; y]
end
```

Does not typecheck in OCaml!

Why?

- [x; y] : 'a list
- Tree.mk_tree : Forest.t -> Tree.t
- Forest.t is an abstract type, and thus Forest.t and 'a list cannot be unified.

Dreyer's RMC type system

- Only his system solves double vision in the context of recursive modules.
- For each abstract type t, it assigns a unique undefined semantic type variable α.
- Each type variable is defined only once and its definition is visible only inside its defining module.
- e.g. Tree.t → α and t (inside Tree) → α / Forest.t → β and t (inside Forest) → β
- Fails to support cyclic type definitions.

Our objectives and main ideas

- To solve the double vision problem by introducing **path equalities**.
- To support cyclic type definitions by identifying **weakly bisimilar types**.
- Assumptions:
  - Every module has its own recursion variable which is annotated with a forward reference signature.
  - Only forward references via recursion variables are allowed.

Cyclic type definitions

```
module rec M : sig
  type t
  type s
end = struct
  type t = N.t
  type s = A of N.s
end
and N : sig
  type t
  type s
end = struct
  type t = A of M.t
  type s = M.s
end

module type S = sig
  type t
end
module type FS =
  functor (X : S) -> S

module F : FS =
functor (X : S) -> struct
  type t = A of X.t
end

module rec Fix : S = F (Fix)
```

- Does not typecheck in Dreyer's RMC type system.
- Typechecks in OCaml!

X.Forest.t -> X.Tree.t ⟹ X.Tree.mk_tree [x; y]

Z.t ⟹ X.Tree.t list

end

Our solution to double vision

```
module type S = rec(X)sig
  module Tree : ST
  module Forest : SF
end where
module type ST = rec(Y)sig
  type t
  val max : Y.t -> int
  val mk_tree : X.Forest.t -> Y.t
end
module type SF = rec(Z)sig
  type t
  val max : Z.t -> int
  val combine : X.Tree.t -> X.Tree.t -> X.Tree.t
end

rec(X : S)struct
  module Tree = (rec(Y : ST with
    type t = Leaf of int
           | Node of int * X.Forest.t)struct
      type t = ...
      let max x = ...
      let mk_tree x = ...
    end : ST)
  module Forest = (rec(Z : SF with
    type t = X.Tree.t list)struct
      type t = ...
      let rec max x = ...
      let combine x y =
        X.Tree.mk_tree [x; y]
    end : SF)
```

Y = X.Tree

Z = X.Forest

To support cyclic type definitions

To solve double vision

Abstract syntax of the source language

Module paths				
p, q, r	::=	X		recursion variable
	|	$p.M$		
Module expressions				
m	::=	rec$(X : S)$struct $d_1 \ \ldots \ d_n$ end		recursive structure
	|	$(m : S)$		opaque sealing
	|	p		module path
Definitions				
d	::=	module $M = m$		module definition
	|	datatype $t = c \ of \ \tau$		datatype definition
	|	type $t = \tau$		type abbreviation
	|	val $l = e$		value definition
Signatures				
S	::=	rec(X)sig $D_1 \ \ldots \ D_n$ end		recursive signature
Specifications				
D	::=	module $M : S$		module specification
	|	datatype $t = c \ of \ \tau$		datatype specification
	|	type $t = \tau$		manifest type specification
	|	type t		abstract type specification
	|	val $l : \tau$		value specification
Programs				
P	::=	(rec$(X : S)$struct $d_1 \ \ldots \ d_n$ end, e)		

Core types	τ	::=	$1 \mid \tau_1 \to \tau_2 \mid \tau_1 * \tau_2 \mid p.t$
Core expressions	e	::=	$x \mid () \mid \lambda x : \tau.e \mid e_1 \ e_2 \mid (e_1, e_2) \mid \pi_i(e) \mid p.c \ e \mid$ case e of $p.c \ x \Rightarrow e' \mid p.l$

Typing rules and weak bisimulation relations

Module contexts	Γ	::=	$\cdot \mid \Gamma, X : S$
Path equalities	Δ	::=	$\cdot \mid \Delta, p = q$
Core typing contexts	Σ	::=	$\cdot \mid \Sigma, x : \tau$

$\Gamma; \Delta; p \vdash m : S$

$$\frac{\Gamma \vdash S \text{ wf} \quad \Gamma, X : S; \Delta, X = p.X \vdash d_i : D_i \quad (1 \le i \le n) \quad \Gamma; \Delta; p \vdash \text{rec}(X)\text{sig } D_1 \ \ldots \ D_n \text{ end} \equiv S}{\Gamma; \Delta; p \vdash \text{rec}(X : S)\text{struct } d_1 \ \ldots \ d_n \text{ end} : \text{rec}(X)\text{sig } D_1 \ \ldots \ D_n \text{ end}} \ \text{T-Str}$$

$\Gamma; \Delta; p \vdash d : D$

$$\frac{\Gamma; \Delta; p.M \vdash m : S \quad m \text{ is not a path}}{\Gamma; \Delta; p \vdash \text{module } M = m : \text{module } M : S} \ \text{T-MDef} \qquad \frac{\Gamma \vdash \tau \text{ wf}}{\Gamma; \Delta; p \vdash \text{type } t = \tau : \text{type } t = \tau}$$

$\Gamma; \Delta; \Sigma \vdash e : \tau$

$$\frac{\Gamma; \Delta; \Sigma \vdash e_1 : \tau_1 \to \tau \quad \Gamma; \Delta; \Sigma \vdash e_2 : \tau_2 \quad \Gamma; \Delta \vdash \tau_1 \approx \tau_2}{\Gamma; \Delta; \Sigma \vdash e_1 \ e_2 : \tau} \ \text{CT-App}$$

$\Gamma \vdash \tau$ wf

$$\frac{\Gamma \vdash p \ni \text{datatype } t = c \ of \ \tau}{\Gamma \vdash p.t \text{ wf}} \qquad \frac{\Gamma \vdash p \ni \text{type } t = \tau}{\Gamma \vdash p.t \text{ wf}} \qquad \frac{\Gamma \vdash p \ni \text{type } t}{\Gamma \vdash p.t \text{ wf}}$$

Labeled transition rules

$$\Gamma; \Delta \vdash 1 \xrightarrow{\perp} 0 \qquad \Gamma; \Delta \vdash \tau_1 \to \tau_2 \xrightarrow{\text{arr}_i} \tau_i \qquad \Gamma; \Delta \vdash \tau_1 * \tau_2 \xrightarrow{\text{prd}_i} \tau_i$$

$$\frac{\Gamma \vdash p \ni \text{type } t}{\Gamma; \Delta \vdash p.t \xrightarrow{p.t} 0} \ \text{EQ-Abs} \qquad \frac{\Gamma \vdash p \ni \text{datatype } t = c \ of \ \tau}{\Gamma; \Delta \vdash p.t \xrightarrow{p.t} 0} \ \text{EQ-Data}$$

Silent transition rules

$$\frac{\Gamma \vdash p \ni \text{type } t = \tau}{\Gamma; \Delta \vdash p.t \rightarrowtail \tau} \ \text{EQ-Type} \qquad \frac{p = q \in \Delta}{\Gamma; \Delta \vdash p.t \rightarrowtail q.t} \ \text{EQ-Path1} \qquad \frac{p = q \in \Delta}{\Gamma; \Delta \vdash q.t \rightarrowtail p.t} \ \text{EQ-Path2}$$

Weak bisimulation relations

Weak bisimulation relation $\Gamma; \Delta \vdash \tau_1 \approx \tau_2$ means that τ_1 and τ_2 are weakly bisimilar under Γ and Δ. Weak bisimulation relation $\Gamma; \Delta \vdash \tau_1 \approx \tau_2$ holds if there exists a weak bisimulation $\mathcal{R}$ such that $\Gamma; \Delta \vdash \tau_1 \mathcal{R} \tau_2$ and

i) if $\Gamma; \Delta \vdash \tau_1 \to \tau_1'$, then there exists τ_2' such that $\Gamma; \Delta \vdash \tau_2 \rightarrowtail^* \tau_2'$ and $\Gamma; \Delta \vdash \tau_1' \mathcal{R} \tau_2'$;

ii) if $\Gamma; \Delta \vdash \tau_1 \xrightarrow{l} \tau_1'$, then there exists τ_2' such that $\Gamma; \Delta \vdash \tau_2 \xrightarrow{l}{}^* \tau_2'$ and $\Gamma; \Delta \vdash \tau_1' \mathcal{R} \tau_2'$;

iii) if $\Gamma; \Delta \vdash \tau_2 \to \tau_2'$, then there exists τ_1' such that $\Gamma; \Delta \vdash \tau_1 \rightarrowtail^* \tau_1'$ and $\Gamma; \Delta \vdash \tau_1' \mathcal{R} \tau_2'$;

iv) if $\Gamma; \Delta \vdash \tau_2 \xrightarrow{l} \tau_2'$, then there exists τ_1' such that $\Gamma; \Delta \vdash \tau_1 \xrightarrow{l}{}^* \tau_1'$ and $\Gamma; \Delta \vdash \tau_1' \mathcal{R} \tau_2'$.

Operational semantics and type soundness

Values	v	::=	$() \mid \lambda x : \tau.e \mid (v_1, v_2) \mid p.c \ v$
Evaluation contexts	κ	::=	$\{\} \mid \kappa \ e \mid v \ \kappa \mid (\kappa, e) \mid (v, \kappa) \mid \pi_i(\kappa) \mid p.c \ \kappa \mid$ case κ of $p.c \ x \Rightarrow e$

$$(\lambda x.\tau : e) \ v \ \to_\beta \ [x \mapsto v]e$$
$$\pi_i(v_1, v_2) \ \to_\beta \ v_i$$
$$\text{case } p.c \ v \text{ of } q.c \ x \Rightarrow e \ \to_\beta \ [x \mapsto v]e$$

$$\frac{m \vdash p \ni \text{val } l = e}{m \vdash p.l \to e} \ \text{R-PExp} \qquad \frac{e_1 \to_\beta e_2}{m \vdash e_1 \to e_2} \ \text{R-Beta} \qquad \frac{m \vdash e_1 \to e_2 \quad \kappa \neq \{\}}{m \vdash \kappa\{e_1\} \to \kappa\{e_2\}} \ \text{R-Ctx}$$

- The evaluation of a program (*m, e*) begins by reducing the given expression *e* with respect to the top-level recursive structure *m*.
- We show type soundness by the usual progress and preservation properties (in progress).