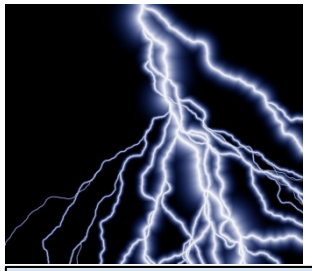


번개 발표

A Recursive Type System with Type Abbreviations and Abstract Types

타입에 이름 붙이기와 타입의 속내용 감추기를 지원하는 재귀 타입 시스템

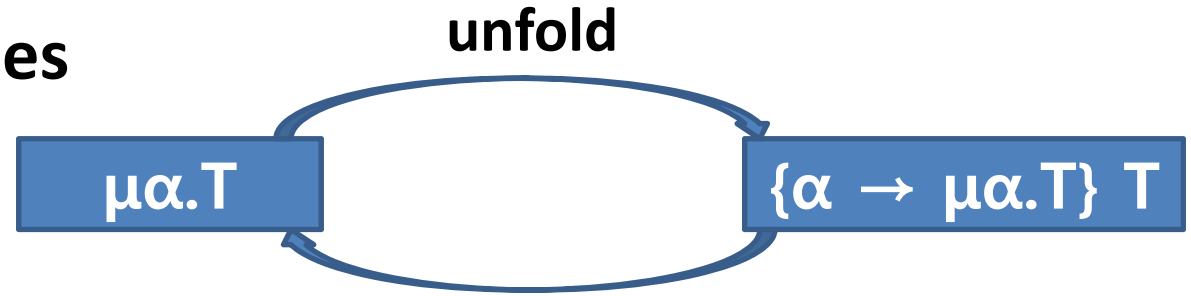
임현승 Keiko Nakata 박성우

ROSAEC Center Workshop @ 이천
2012년 7월 27일 금요일

재귀 타입 (Recursive Types)

```
type α tree = Leaf of α
            | Node of α tree * α * α tree
```

- **Iso-recursive types**



- **Equi-recursive types**

  - Structural polymorphism, e.g., polymorphic variants or objects, supported in OCaml requires **structural type equivalence**.

수축하지 않는 재귀 타입 (Non-contractive Types)

- A type is **non-contractive** if unfolding type definitions **diverges** and is **not guarded** by a type constructor
  - `type t = t, type s = u and u = s`
- Contractive types
  - `type t = int, type α t = α, type t = t * t`
- We cannot detect non-contractive types accurately.

```
module rec P : sig type t end =
  struct type t = Q.t end
and Q : sig type t end =
  struct type t = P.t end
```

타입의 속내용 감추기 (Abstract Types)

- Abstract types by signature sealing in ML

```
module M = struct
  type 'a t = 'a
end
module M = (M : S)
```

```
module type S = sig
  type 'a t
end
(* signature sealing *)
```

타입의 속내용 감추기와 수축하지 않는 타입

- Non-contractive types in a signature are a source of **type unsoundness**

```
module M = struct
  type t = int
  type s = bool
  let succ x = x + 1
  let bval = true
end
module M = (M : S)
let x = M.succ M.sval
```

```
module type S = sig
  type t = t
  type s = s
  val succ : t -> t
  val bval : s
end

(* run-time error *)
```

연구 요약

- **Features of our recursive type system**
  - Equi-recursive types, structural type equivalence
  - **Type parameters, non-contractive types in the implementation, abstract types**
    (supported in OCaml, but no sound type theory)
  - Type equivalence, contractiveness defined in **mixed induction-coinduction**
- **Contributions**
  - First sound type system with all these three features
  - Interesting proof techniques
  - Whole system and proofs formalized in Coq

보다 자세한 이야기는 포스터 발표에서~!

A Recursive Type System with Type Abbreviations and Abstract Types

타입에 이름 붙이기와 타입의 속내용 감추기를 지원하는 재귀 타입 시스템

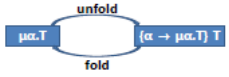
임현승, Keiko Nakata, 박성우

제 8회 ROSAEC Center Workshop @ 이천, 25-28 July 2012

Recursive Types

```
type 'a tree = Leaf of 'a
             | Node of 'a tree * 'a * 'a tree
```

- Iso-recursive types



- Equi-recursive types
- Structural polymorphism, e.g., polymorphic variants or objects, supported in OCaml requires structural type equivalence (equi-recursive types).

Non-contractive Types

- A type is non-contractive if unfolding type definitions diverges and is not guarded by a type constructor
  - `type t = t, type s = u and u = s`
- Contractive types
  - `type t = int, type 'a t = 'a, type t = t * t`
- We cannot detect non-contractive types accurately.

```
module rec P : sig type t end =
  struct type t = Q.t end
and Q : sig type t end =
  struct type t = P.t end
```

- Our type equivalence relation should be able to handle non-contractive types.

Abstract Types, Signature Sealing, Non-contractive Types

- Abstract types by signature sealing in ML

```
module M = struct
  type 'a t = 'a
end
module M = (M : S)
```

```
module type S = sig
  type 'a t
end
(* signature sealing *)
```

- Non-contractive types in a signature are a source of type unsoundness

```
module M = struct
  type t = int
  type s = bool
  let succ x = x + 1
  let bval = true
end
module M = (M : S)
let x = M.succ M.bval (* run-time error *)
```

```
module type S = sig
  type t = t
  type s = s
  val succ : t -> t
  val bval : s
end
```

- Disallow non-contractive types in the sealed signature.

Features of Our Recursive Type System

- Equi-recursive types, structural type equivalence
- Type parameters, non-contractive types in the implementation, abstract types (supported in OCaml, but no sound type theory)
- Type equivalence, contractiveness defined in mixed induction and coinduction

Syntax & Additional Judgments

type name	s, t, u		
type	τ, σ	$::=$	$\mathsf{unit} \mid \alpha \mid \tau \to \sigma \mid \tau\, t$
term	e	$::=$	$() \mid a \mid x \mid \lambda a:\tau.e \mid e_1\, e_2 \mid \mathsf{fix}\, a:\tau.e \mid l$
specification	D	$::=$	$\mathsf{type}\ \alpha\ t \mid \mathsf{type}\ \alpha\ t = \tau \mid \mathsf{val}\ l : \tau$
definition	d_τ	$::=$	$\mathsf{type}\ \alpha\ t = \tau$
	d_e	$::=$	$\mathsf{let}\ l = e$
signature	S	$::=$	$\cdot \mid S, D$
structure	M	$::=$	$(\overline{d_\tau}, \overline{d_e})$
program	P	$::=$	$(M, S, e) \mid (M, e)$
value context	Γ	$::=$	$\cdot \mid \Gamma, x:\tau$
type variable set	Σ	$::=$	$\cdot \mid \{\alpha\}$
well-formedness	$S; \alpha \vdash \tau\ \mathsf{type}$		$S \vdash D\ \mathsf{ok}$ $S\ \mathsf{ok}$
membership	$S \ni \mathsf{type}\ \alpha\ t = \sigma$		
subtyping	$S_1 \le S_2$		$S \vdash D_1 \le D_2$
well-typedness	$\vdash P : (S, \tau)$		$\vdash M : S$ $S \vdash \overline{d_e} : S_e$ $S; \Gamma \vdash e : \tau$
reduction	$P \longmapsto P'$		$M \longmapsto M'$ $\overline{d_v} \vdash e \longmapsto e'$

Type Equivalence

Unfolding

$$\frac{S \ni \mathsf{type}\ \alpha\ t = \sigma}{S \vdash \tau\, t \to \{\alpha \mapsto \tau\}\sigma}\ \text{unfold} \qquad S \vdash \tau \to \sigma$$

Coinductive type equivalence

$$\frac{R \subseteq \equiv \quad S; \Sigma \vdash \tau \overset{R}{=} \sigma}{S; \Sigma \vdash \tau \equiv \sigma}\ \text{eq-ind} \qquad \frac{S \vdash \tau \to \tau' \quad S \vdash \sigma \to \sigma' \quad S; \Sigma \vdash \tau' \equiv \sigma'}{S; \Sigma \vdash \tau \equiv \sigma}\ \text{eq-coind} \qquad S; \Sigma \vdash \tau_1 = \tau_2$$

Inductive type equivalence

$$\frac{}{S; \Sigma \vdash \mathsf{unit} \overset{R}{=} \mathsf{unit}}\ \text{eq-unit} \qquad \frac{\alpha \in \Sigma}{S; \Sigma \vdash \alpha \overset{R}{=} \alpha}\ \text{eq-var}$$

$$\frac{S; \Sigma \vdash \tau_0 R\, \sigma_1 \quad S; \Sigma \vdash \tau_2 R\, \sigma_2}{S; \Sigma \vdash \tau_1 \to \tau_2 \overset{R}{=} \sigma_1 \to \sigma_2}\ \text{eq-fun} \qquad \frac{S \ni \mathsf{type}\ \alpha\ t \quad S; \Sigma \vdash \tau\, R\, \sigma}{S; \Sigma \vdash \tau\, t \overset{R}{=} \sigma\, t}\ \text{eq-abs}$$

$$\frac{\Delta \vdash \tau \to \tau' \quad \Delta; \Sigma \vdash \tau' \overset{R}{=} \sigma}{\Delta; \Sigma \vdash \tau \overset{R}{=} \sigma}\ \text{eq-lunfold} \qquad \frac{S \vdash \sigma \to \sigma' \quad S; \Sigma \vdash \tau \overset{R}{=} \sigma'}{S; \Sigma \vdash \tau \overset{R}{=} \sigma}\ \text{eq-runfold}$$

Contractive Types and Signatures

Contractive types

$$\frac{C \subseteq \Downarrow \quad S \downarrow_C \tau}{S \Downarrow \tau}\ \text{ctr-coind} \qquad \frac{}{S \downarrow_C \mathsf{unit}}\ \text{ctr-unit} \qquad \frac{}{S \downarrow_C \alpha}\ \text{ctr-var}$$

$$\frac{(S, \tau) \in C \quad (S, \sigma) \in C}{S \downarrow_C \tau \to \sigma}\ \text{ctr-fun} \qquad \frac{S \ni \mathsf{type}\ \alpha\ t \quad S \downarrow_C \tau}{S \downarrow_C \tau\, t}\ \text{ctr-abs} \qquad \frac{S \vdash \tau \to \sigma \quad S \downarrow_C \sigma}{S \downarrow_C \tau}\ \text{ctr-type}$$

Contractive signatures

$$\frac{\mathrm{BN}(S)\ \text{distinct} \quad \forall (\mathsf{type}\ \alpha\ t = \tau) \in S.\ S \Downarrow \tau}{S \Downarrow}\ \text{ctr-sig} \qquad S \Downarrow$$

Type Soundness

value	v	$::=$	$() \mid \lambda a:\tau.e$
definition value	d_v	$::=$	$\mathsf{let}\ l = v$
module value	$\overline{V}$	$::=$	$\overline{d_\tau}, \overline{d_v}$
program value	P_v	$::=$	$(\overline{V}, v)$

Theorem A.1 **(Progress)**
If $\vdash P : (S, \tau)$, then either P is a program value or there exists P' such that $P \longmapsto P'$.

Proof
By induction on $\vdash P : (S, \tau)$. □

Theorem A.2 **(Preservation)**
(1) If $\vdash \langle \overline{d_\tau}, \overline{d_e} \rangle : S, S; \cdot \vdash e : \tau$, and $\overline{d_e} \vdash e \longmapsto e'$, then $S; \cdot \vdash e' : \tau$.
(2) If $\vdash M : S$ and $M \longmapsto M'$, then $\vdash M' : S$.
(3) If $P = (M, e)$, $\vdash P : (S, \tau)$ and $P \longmapsto P'$, then $\vdash P' : (S, \tau)$.
(4) If $\vdash (M, S, e) : (S, \tau)$, then there exists S' such that $\vdash M : S'$, $S' \le S$, and $S'; \cdot \vdash e : \tau$.

Proof
(1) By induction on a derivation of $S; \cdot \vdash e : \tau$.
(2) By case analysis using (1).
(3) By case analysis using (1) and (2).
(4) By using the signature elimination lemma.
□

Key Difficulty in Soundness Proofs

$$\frac{\vdash M : S' \quad S \Downarrow \quad S' \le S \quad S \vdash e : \tau}{\vdash (M, S, e) : (S, \tau)}\ \text{typ-prog-seal} \qquad \frac{\vdash M : S \quad S; \cdot \vdash e : \tau}{\vdash (M, e) : (S, \tau)}\ \text{typ-prog}$$

Lemma A.3 **(Signature elimination)**
If $\vdash (M, S, e) : (S, \tau)$, then $\exists S'$ such that $\vdash (M, e) : (S', \tau)$ and $S' \le S$.

Lemma A.4 **(Typing is preserved by signature elimination)**
If $S_1 \le S_2$, $S_2 \Downarrow$ and $S_2; \Gamma \vdash e : \tau$, then $S_1; \Gamma \vdash e : \tau$.

Lemma A.5 **(Type equivalence is preserved by signature elimination)**
If $S_1 \le S_2$, $S_2 \Downarrow$ and $S_2; \Sigma \vdash \tau \equiv \sigma$, then $S_1; \Sigma \vdash \tau \equiv \sigma$.

Lemma A.6 **(Well-formed types are contractive)**
Suppose $S\,\mathsf{ok}$, $S \Downarrow$, and $S; \Sigma \vdash \tau$ type. Then $S \Downarrow \tau$.

Proof
The proof is by coinduction. The derivation tree below illustrates the key idea of the proof where relation C is defined as $\{(S_0, \tau_0) \mid S_0\,\mathsf{ok},\ S_0 \Downarrow,\ \text{and}\ S_0; \Sigma_0 \vdash \tau_0\ \mathsf{type}\}$.

$$\frac{\dfrac{\dfrac{}{C \subseteq \Downarrow}\ (1) \qquad \dfrac{\dfrac{}{S \ni \mathsf{type}\ \alpha_i\ t_i}\ (2) \quad S \downarrow_C \sigma_i}{S \downarrow_C \sigma_j\, t_j} \quad \vdots}{S \downarrow_C \sigma_i\, t_i} \quad \vdots}{S \downarrow_C \sigma}$$

$$\frac{S \vdash \tau \to \sigma \qquad S \downarrow_C \sigma}{S \downarrow_C \tau}$$

$$\frac{S \downarrow_C \tau}{S \Downarrow \tau}$$

□

Contributions

- First sound type system with type parameters, non-contractive types, and abstract types
- Interesting proof techniques
- Whole system and proofs are formalized in Coq

Strong Contractiveness

Strong unfolding

$$\frac{S \vdash \tau \to \sigma}{S \vdash \tau\, t \to \sigma\, t}\ \text{sunfold-abs} \qquad \frac{S \ni \mathsf{type}\ \alpha\ t = \sigma}{S \vdash \tau\, t \to \{\alpha \mapsto \tau\}\sigma}\ \text{sunfold-type} \qquad S \vdash \tau \to \sigma$$

Strong contractive types

$$\frac{C \subseteq \Downarrow^s \quad S \vdash \tau \to^* \sigma \quad S \downarrow^s_C \sigma}{S \Downarrow^s \tau}\ \text{sctr-coind} \qquad \frac{}{S \downarrow^s_C \mathsf{unit}}\ \text{sctr-unit} \qquad \frac{}{S \downarrow^s_C \alpha}\ \text{sctr-var} \qquad S \downarrow^s \tau \quad S \downarrow^s_C \tau$$

$$\frac{(S, \tau) \in C \quad (S, \sigma) \in C}{S \downarrow^s_C \tau \to \sigma}\ \text{sctr-fun} \qquad \frac{S \ni \mathsf{type}\ \alpha\ t \quad S \downarrow^s_C \tau}{S \downarrow^s_C \tau\, t}\ \text{sctr-abs}$$

Strong contractive signature

$$\frac{\mathrm{BN}(S)\ \text{distinct} \quad \forall (\mathsf{type}\ \alpha\ t = \tau) \in S.\ S \downarrow^s \tau}{S \Downarrow^s}\ \text{sctr-sig} \qquad S \Downarrow^s$$

Lemma A.7 **(Equivalence between contractiveness and strong contractiveness)**
Suppose $S\,\mathsf{ok}$. Then $S \Downarrow \tau$ if and only if $S \Downarrow^s \tau$.

Proof
The proof is by induction nested into coinduction. □

Corollary A.8
$S \Downarrow$ if and only if $S \Downarrow^s$.

Proof
Corollary of Lemma A.7 □

Type Soundness Again!

Original subtyping

$$\frac{S_1\,\mathsf{ok} \quad S_2\,\mathsf{ok} \quad \forall n \in \mathrm{dom}(S_2).\ S_1 \vdash S_1(n) \le S_2(n)}{S_1 \le S_2}\ \text{sub-sig}$$

Refined subtyping

$$\frac{S_1\,\mathsf{ok} \quad S_2\,\mathsf{ok} \quad \mathrm{dom}(S_1) = \mathrm{dom}(S_2) \quad \forall n \in \mathrm{dom}(S_2).\ S_1 \vdash S_1(n) \le S_2(n)}{S_1 \le S_2}\ \text{sub-sigeq}$$

Signature extension

$$\begin{array}{lcl} \mathit{SigExt}(S_1, S_2) & := & \langle S_1 / S_2 \rangle^\circ \cup S_2 \\ S_1 / S_2 & := & \{D_1 \mid D_1 \in S_1,\ \forall D_2 \in S_2.\ \mathrm{BN}(D_1) \neq \mathrm{BN}(D_2)\} \\ \langle D_1, \ldots, D_k \rangle^\circ & := & \langle D_1 \rangle^\circ, \ldots, \langle D_k \rangle^\circ \\ (\mathsf{type}\ \alpha\ t = \tau)^\circ & := & \mathsf{type}\ \alpha\ t \\ \langle D \rangle^\circ & := & D \quad \text{where } D \text{ is not a type equation} \end{array}$$

Lemma A.9
If $S_1 \le S_2$ then $S_1 \le \mathit{SigExt}(S_1, S_2)$.

Proof
By the definition of SigExt and subtyping. □

Lemma A.10 **(Type equivalence is preserved by signature elimination)**
If $S_1 \le S_2$, $S_2 \Downarrow^s$ and $S_2; \Sigma \vdash \tau \equiv \sigma$, then $S_1; \Sigma \vdash \tau \equiv \sigma$.

Proof
The proof is by coinduction and various properties on $S_2 \Downarrow^s$ □

Lemma A.11 **(Typing is preserved by signature elimination)**
If $S_1 \le S_2$, $S_2 \Downarrow^s$ and $S_2; \Gamma \vdash e : \tau$, then $S_1; \Gamma \vdash e : \tau$.

Proof
By rule induction on $S_2; \Gamma \vdash e : \tau$ □

Proof of Lemma A.3 (Signature elimination)

- $\vdash M : S'$, $S' \le S$, $S \Downarrow$, and $S; \cdot \vdash e : \tau$ — By inversion on $\vdash (M, S, e) : (S, \tau)$
- $S' \le \mathit{SigExt}(S', S)$ — By Lemma A.9 with $S' \le S$
- $\mathit{SigExt}(S', S) \Downarrow^s$ — From $S \Downarrow$
- $\mathit{SigExt}(S', S); \cdot \vdash e : \tau$ — By weakening with $S; \cdot \vdash e : \tau$
- $S'; \cdot \vdash e : \tau$ — By Lemma A.11
- $\vdash (M, e) : (S', \tau)$ — By the rule typ-prog with $\vdash M : S'$ and $S'; \cdot \vdash e : \tau$

□