A Recursive Type System with Type Abbreviations

Hyeonseung Im
Keiko Nakata  Sungwoo Park

PL Lab, POSTECH

ROSAEC Center 7th Workshop @ HKUST
January 19, 2012
OCaml recursive modules allow us to write mutually recursive definitions into separate modules.

```ocaml
module rec Expr : sig
  type t
end = struct
  type t = Var of string
  | LetExpr of Bind.t * t
  | ...
end

and Bind : sig
  type t
end = struct
  type t = LetBind of string * Expr.t
  | ...
end
```
Motivation: Non-contractive Type Cycles

- Contractive type cycles
  \[ \text{e.g., type } t = s \times s, \text{ type } s = t \rightarrow t \]

- Non-contractive type cycles
  \[ \text{e.g., type } t = t, \text{ type } s = u, \text{ type } u = s \]

Why non-contractive type cycles matter? Because we cannot detect them accurately.

```ocaml
module rec M : sig type 'a t end =
  struct type 'a t = 'a N.t end
and N : sig type 'a t end =
  struct type 'a t = 'a M.t end
```

To show soundness of OCaml Recursive Modules, we should be able to deal with non-contractive type cycles.
We prove soundness of a simple typed language with OCaml-style parameterized recursive type definitions, possibly non-contractive.

**Type Language**

- **Type names**: $s, t, u$
- **Kinds**: $\kappa ::= 1 | 1 \to 1$
- **Types**: $\tau, \sigma ::= \text{unit} | \alpha | \beta | \gamma | \cdots | \tau \to \sigma | \tau_1 \times \tau_2 | t | \tau \; t$

**Type definitions**: $D ::= \text{type} \; t :: 1 = \tau | \text{type} \; \alpha \; t :: 1 \to 1 = \tau$

**Challenge**: To define a type equivalence relation being able to handle non-contractive type cycles.
Given a set of mutually recursive type definitions, let us define well-formedness rules and a type equivalence relation.

<table>
<thead>
<tr>
<th>Set of type definitions</th>
<th>$\Delta ::= \cdot \mid \Delta, D$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type contexts</td>
<td>$\Sigma ::= \cdot \mid \Sigma, \alpha$ type</td>
</tr>
<tr>
<td>Well-formed types</td>
<td>$\Delta; \Sigma \vdash \tau :: 1$</td>
</tr>
</tbody>
</table>

- $\Delta; \Sigma \vdash \text{unit} :: 1$
- $\Delta; \Sigma \vdash \alpha :: 1$
- $\Delta; \Sigma \vdash \tau :: 1\quad \Delta; \Sigma \vdash \sigma :: 1$
- $\Delta; \Sigma \vdash \tau \rightarrow \sigma :: 1$
- $\Delta \ni \text{type } t :: 1 = \tau$
- $\Delta \ni \text{type } \alpha \ t :: 1 \rightarrow 1 = \sigma$
- $\Delta; \Sigma \vdash \tau :: 1$
- $\Delta; \Sigma \ni \tau \ t :: 1$
Well-formed Type Definitions

Well-formed definitions \( \Delta \vdash D \text{ wf} \)

Well-formed sets \( \vdash \Delta \text{ wf} \)

\[
\begin{align*}
\Delta; \cdot \vdash \tau :: 1 & \quad \Delta; \alpha \text{ type} \vdash \tau :: 1 \\
\Delta \vdash \text{type } t :: 1 = \tau \text{ wf} & \quad \Delta \vdash \text{type } \alpha \ t :: 1 \rightarrow 1 = \tau \text{ wf}
\end{align*}
\]

\(\text{BN}(\Delta) \text{ distinct} \quad \forall D \in \Delta, \ \Delta \vdash D \text{ wf} \)

\(\Delta \vdash D \text{ wf} \)

\(\text{BN}(\Delta, \text{type } t :: 1 = \tau) \triangleq \text{BN}(\Delta) \cup \{t\} \)

\(\text{BN}(\Delta, \text{type } \alpha \ t :: 1 \rightarrow 1 = \tau) \triangleq \text{BN}(\Delta) \cup \{t\} \)

\(\text{BN}(\cdot) \triangleq \emptyset\)
To account for infinite unfolding of type abbreviations, we use coinductive type equivalence $\Delta; \Gamma \vdash \tau \equiv \sigma :: 1$.

- e.g., type $s = t \times u$, type $t = u \times s$, type $u = s \times t$

- A wrong attempt:

  \[
  \Delta \ni \text{type } t = \tau \quad \Delta; \Sigma \vdash \tau \equiv \sigma :: 1 \\
  \Delta; \Sigma \vdash t \equiv \sigma :: 1
  \]

  This simple approach fails if $\tau = t$ and $\sigma = \text{unit}$.

  \[
  \Delta \ni \text{type } t = t \quad \Delta; \Sigma \vdash t \equiv \text{unit} :: 1 \\
  \Delta; \Sigma \vdash t \equiv \text{unit} :: 1
  \]

- Apply coinduction if both types unfold.

- Otherwise apply induction.
Inductive type equivalence  \( \Delta; \Gamma \vdash \tau = \sigma :: 1 \)

\[
\begin{align*}
\Delta; \Sigma \vdash \text{unit} = \text{unit} :: 1 & \quad \text{\( \alpha \in \Sigma \)} \\
\Delta; \Sigma \vdash \alpha = \alpha :: 1 & \\
\Delta; \Sigma \vdash \tau_1 \equiv \sigma_1 :: 1 & \quad \Delta; \Sigma \vdash \tau_2 \equiv \sigma_2 :: 1 \\
\Delta; \Sigma \vdash \tau_1 \rightarrow \tau_2 = \sigma_1 \rightarrow \sigma_2 :: 1 & \\
\Delta; \Sigma \vdash \tau_1 \equiv \sigma_1 :: 1 & \quad \Delta; \Sigma \vdash \tau_2 \equiv \sigma_2 :: 1 \\
\Delta; \Sigma \vdash \tau_1 \ast \tau_2 = \sigma_1 \ast \sigma_2 :: 1 & \\
\Delta; \Sigma \vdash t = t :: 1 & \quad \Delta; \Sigma \vdash \tau t = \tau' t :: 1 \\
\end{align*}
\]
Type Equivalence (Coinductive Interpretation)

- Unfolding type abbreviations:
  \[
  \frac{\Delta \ni \text{type } t :: 1 = \sigma}{\Delta \vdash t \rightarrow \sigma} \quad \frac{\Delta \ni \text{type } \alpha t :: 1 \rightarrow 1 = \sigma}{\Delta \vdash t \rightarrow \{t/\alpha\} \sigma}
  \]

- \( \Delta \vdash \tau \rightarrow^* \tau' \) is the reflexive and transitive closure of a single transition.

- Coinductive type equivalence:
  \[
  \frac{\Delta \vdash \tau \rightarrow \tau' \quad \Delta \vdash \sigma \rightarrow \sigma' \quad \Delta; \Sigma \vdash \tau' \equiv \sigma' :: 1}{\Delta; \Sigma \vdash \tau \equiv \sigma :: 1}
  \]
  \[
  \frac{\Delta \vdash \tau \rightarrow^* \tau' \quad \Delta \vdash \sigma \rightarrow^* \sigma' \quad \Delta; \Sigma \vdash \tau' = \sigma' :: 1}{\Delta; \Sigma \vdash \tau \equiv \sigma :: 1}
  \]
Conclusion and Future Work

- Studied a recursive type system with type abbreviations and proved its soundness.

- Am mechanizing the soundness proof using Coq.
  - Difficulty: Formalizing mixed induction-coinduction
  - Idea: Parameterizing an inductive definition by a coinductive definition

- Will extend the system with abstract types.
Thank you!
Questions?