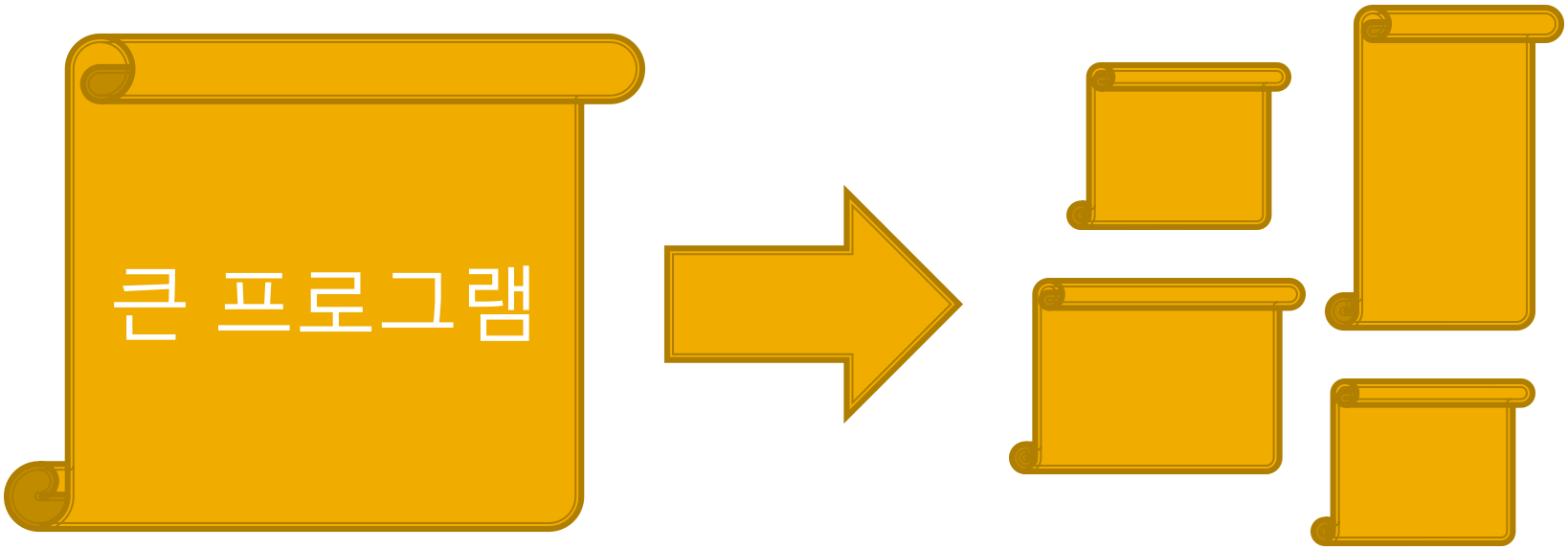
재귀 모듈을 위한 구문 기반 타입 시스템
(A syntactic type system for recursive modules)

Hyeonseung Im @ POSTECH PL 연구실
Joint work with Keiko Nakata, Jacques Garrigue, and Sungwoo Park

ROSAEC Center Workshop @ 통영, 2011-01-08 (토요일)

모듈화 프로그래밍 (Modular programming)



모듈화 프로그래밍 (Modular programming)

- 장점?
  - 컴포넌트 각각의 불변성 (invariants) 및 특성을 독립적으로 이해
  - 컴포넌트 단위 개발 및 유지 보수 용이
  - 코드 재사용

그렇다면 모듈화 프로그래밍을 어떻게 하면 잘 할 수 있을까요?

그렇다면 모듈화 프로그래밍을 어떻게 하면 잘 할 수 있을까요?

- 프로그래머가 알아서 잘 하면 됩니다.

감사합니다.
제 발표는 여기까지 입니다.

그렇다면 모듈화 프로그래밍을 어떻게 하면 잘 할 수 있을까요?

- 프로그래밍 언어 차원에서 지원해주면 용이함

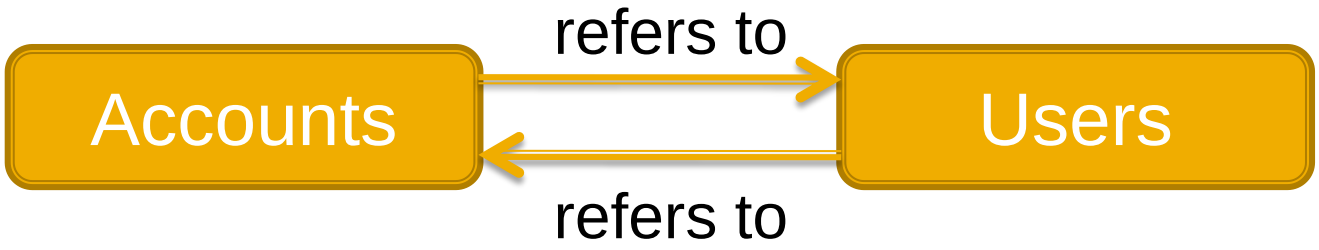
- 객체 지향 패러다임
- 모듈 시스템

모듈이란? (What is a module?)

- 스트럭처 (Structures or modules)
- 시그너처 (Signatures)
- 추상타입 (Abstract types)
- 중첩모듈 (Nested modules)
- 모듈함수 (functors)

재귀 모듈이란? (What is a recursive module?)

- 동시에 상호 참조하는 서로 다른 모듈



- 따로 컴파일 (Separate compilation)

발표 순서 (Roadmap)

- 배경지식 소개 ✓
- **연구 동기 및 목표**

연구동기 (Motivation)

- 모듈 ??년 동안 연구
- 재귀 모듈은 ??년 동안 연구

연구동기 (Motivation)

- 모듈 30년 동안 연구
- 재귀 모듈은 10년 동안 연구

- 그러나,

연구동기 (Motivation)

- 모듈 30년 동안 연구
- 재귀 모듈은 10년 동안 연구

- 그러나,
  - 문법 기반 타입 시스템
  - 복시 문제
  - 순환 타입

우리의 목표 (Our goals)

- 재귀 모듈을 위한 일반적인 뼈대 설계
  - 문법 기반 타입 시스템
  - 복시 문제 해결
  - 순환 타입 지원
  - 타입 안전성 보장

우리의 목표 (Our goals)

- 재귀 모듈을 위한 일반적인 뼈대 설계
  - 문법 기반 타입 시스템
  - 복시 문제 해결
  - 순환 타입 지원
  - 타입 안전성 보장

- 모듈함수 없이

발표 순서 (Roadmap)

- 배경지식 소개
- 연구 동기 및 목표 ✓
- **복시 문제 및 해결책**
- 순환 타입
- 결론

예제: 나무-숲 재귀 모듈 (Tree-Forest recursive modules)

```
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 = match x with
    | Leaf i -> i
    | Node (i, f) ->
      let j = Forest.max f in
        if i > j then i else j
  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 = match x with
    | [] -> 0
    | hd :: tl ->
      let i = Tree.max hd in
      let j = max tl in
        if i > j then i else j
  let combine x y = Tree.mk_tree [x; y]
end
```

복시 문제
(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 = match x with
    | Leaf i -> i
    | Node (i, f) ->
      let j = Forest.max f in
        if i > j then i else j
  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 = match x with
    | [] -> 0
    | hd :: tl ->
      let i = Tree.max hd in
      let j = max tl in
        if i > j then i else j
  let combine x y = Tree.mk_tree [x; y]
end
```

Abstract type

Expect a value of type Forest.t which is an abstract type!

'a list

복시 문제
(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 = match x with
    | Leaf i -> i
    | Node (i, f) ->
      let j = Forest.max f in
        if i > j then i else j
  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 = match x with
    | [] -> 0
    | hd :: tl ->
      let i = Tree.max hd in
      let j = max tl in
        if i > j then i else j
  let combine x y = Tree.mk_tree [x; y]
end
```

Does not typecheck!

프로그래머 관점에서의 복시 문제 (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 = match x with
    | Leaf i -> i
    | Node (i, f) ->
      let j = Forest.max f in
        if i > j then i else j
  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 = match x with
    | [] -> 0
    | hd :: tl ->
      let i = Tree.max hd in
      let j = max tl in
        if i > j then i else j
  let combine x y = Tree.mk_tree [x; y]
end
```

Abstract type

Unifiable!

Expect a value of type Forest.t

'a list

복시 문제에 대한 우리의 해법은? (Our solution to double vision)

- 패스 다시 쓰기
  - 외부에서 쓰는 이름 → 내부에서 쓰는 이름
  - Forest.t → t

복시 문제에 대한 우리의 해법 (Our solution to double vision)

```
module type S = rec(X)sig
  module Tree : ST
  module Forest : SF
end

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)
end
```

X.Tree → Y

X.Forest → Z

X.Forest.t -> X.Tree.t

Z.t -> X.Tree.t

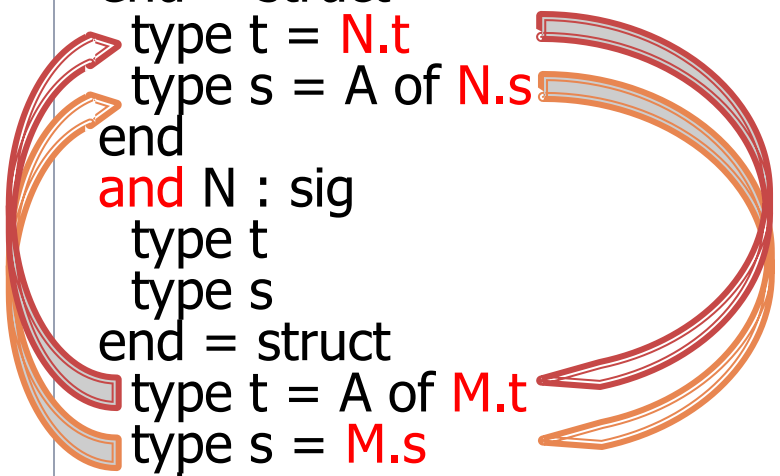
X.Tree.t list -> X.Tree.t

발표 순서 (Roadmap)

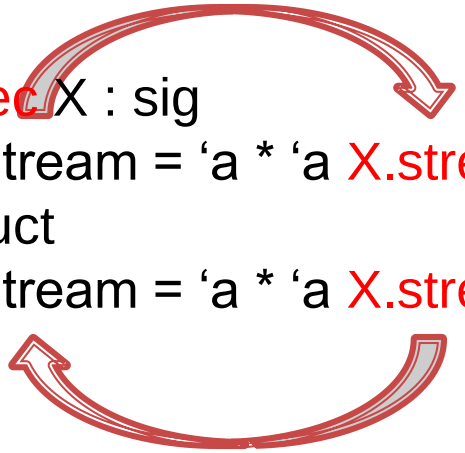
- 배경지식 소개
- 연구 동기 및 목표
- 복시 문제 및 해결책 ✓
- **순환 타입**
- 결론

순환 타입
(Cyclic type definitions, contrived)

```
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 rec X : sig
  type 'a stream = 'a * 'a X.stream
end = struct
  type 'a stream = 'a * 'a X.stream
end
```



순환 타입
(Cyclic type definitions, contrived)

```
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 rec X : sig
  type 'a stream = 'a * 'a X.stream
end = struct
  type 'a stream = 'a * 'a X.stream
end
```

Weak bisimulations!

현재까지 한 일 (What we've done)

- 문법 기반 타입 시스템 설계
- 복시 문제를 해결
- 순환 타입을 지원
- 타입 안전성 증명

현재까지 한 일

2 Abstra

3 A type

Our type system
the double vision
tying weakly bisi
bisimulation relat

For the purpo
rewriting, that is

3.1 Path ty

$\Phi;\Gamma \vdash_{\mathrm{path}} p : S$

3.2 Membe

Inspired by νObj
a corresponding e

$\Phi;\Gamma \vdash p \ni D$

3.3 Path ali

Inspired by Feath
which carries all t

$\Phi;\Gamma \vdash p \simeq q$

We can show t
taneous induction

3.4 Typin

Module expres

$\Gamma \vdash S$ wf $\Gamma,$

$\Phi;\Gamma \vdash q_1 \simeq$

$\Phi;\Gamma \vdash$

Definitions

$\Gamma;\Delta;p \vdash$

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

Core expressio

$\frac{x:\tau \in \Sigma}{\Gamma;\Delta;\Sigma \vdash x}$

$\Gamma;\Delta$

$\Gamma;\Delta;\Sigma \vdash e_1$

Programs

• T-Sub: V

3.5 Well-fo

Signatures

S is n

Specifications

$\Phi;\Gamma$

$\Phi;\Gamma \vdash \tau$
$\Phi;\Gamma \vdash$ type

Core types

$\Gamma \vdash 1$ wf Co

Our type sys
the following mo

module ty
module
module
module
end
end

rec(X : S
module
module
module
end
end

If the signatu
module abbrevia
be ill-typed. We
types is the only

3.6 Subty

$\Gamma, X : \mathrm{rec}(X)s$

$\Gamma;\Delta;p \vdash$

The rule ≤
the module is
following code

module

rec(X :
module
end

The correct sig

module

3.7 Type e

This section defi
labeled transitio
(Γ,Δ).

Labeled transi

$\Gamma \vdash p \ni ty$
$\Gamma;\Delta$

Silent transitio

$\Gamma \vdash p \ni typ$
$\Gamma;\Delta \vdash p.$

For any type,
For transition

• $p \mapsto X \in \Delta$
• If $p \mapsto X \in$

$\Gamma;\Delta \vdash \tau \to \tau'$
reflexive, transiti
$\Gamma;\Delta \vdash \tau_1 \xrightarrow{\delta}{}^{*} \tau_2$

• EQ-Alias
• EQ-Data

Weak bisimula

For a binary rela

Definition 3.1.
If, whenever $\Gamma;\Delta$
I) If $\Gamma;\Delta$
II) If $\Gamma;\Delta$
III) If $\Gamma;\Delta$
IV) If $\Gamma;\Delta$
We say that τ_1 a
bisimulation S su

Note. If $\Gamma;\Delta$
contraposition al

4 Operational semantics

This section presents a small-step call-by-value operational semantics using evaluation contexts. (For evaluation of module components, we use a call-by-name strategy.)

- We give the operational semantics with respect to a program (m, e) where $FV(\mathsf{m}) = \emptyset$.
- The evaluation of a program (m, e) begins by reducing the given expression e with respect to the top-level recursive structure m.
- We assume that a program does not include module abbreviations such as module $M = p$. This assumption eliminates the need for signature strengthening [4] in the type system and path normalization [6] in the operational semantics. Note that we can safely eliminate module abbreviations in a preprocessing step without changing the operational interpretation of a source program.

The operational semantics uses a membership judgment $\mathsf{m} \vdash p \ni d$ and a module evaluation judgment $\mathsf{m} \vdash \mathsf{m}_1 \mapsto \mathsf{m}_2$.

Membership $\boxed{\mathsf{m} \vdash p \ni d}$

$$\frac{\mathsf{m} \vdash p \mapsto \mathrm{rec}(X:S)\mathsf{struct}\ d_1\ \ldots\ d_n\ \mathsf{end}}{\mathsf{m} \vdash p \ni [X \mapsto p]\mathsf{d}_i}\ \ni\text{-Dsp}$$

Module evaluation $\boxed{\mathsf{m} \vdash \mathsf{m}_1 \mapsto \mathsf{m}_2}$

$$\frac{}{\mathrm{rec}(X:S)s \vdash X \mapsto \mathrm{rec}(X:S)s}\ \text{R-Var} \qquad \frac{\mathsf{m} \vdash p \ni \mathsf{module}\ M = \mathsf{m}_1 \quad \mathsf{m} \vdash \mathsf{m}_1 \mapsto \mathsf{m}_2}{\mathsf{m} \vdash p.M \mapsto \mathsf{m}_2}\ \text{R-Select}$$

$$\frac{}{\mathsf{m} \vdash \mathrm{rec}(X:S)s \mapsto \mathrm{rec}(X:S)s}\ \text{R-Str} \qquad \frac{}{\mathsf{m} \vdash \mathsf{functor}\ (X:S) \to m' \mapsto \mathsf{functor}\ (X:S) \to m'}\ \text{R-Str}$$

$$\frac{\mathsf{m} \vdash p_1 \mapsto \mathsf{functor}\ (X:S) \to m'}{\mathsf{m} \vdash p_1(p_2) \mapsto [X \mapsto p_2]m'}\ \text{R-App} \qquad \frac{\mathsf{m} \vdash \mathsf{m}_1 \mapsto \mathsf{m}_2}{\mathsf{m} \vdash (\mathsf{m}_1 : S) \mapsto \mathsf{m}_2}\ \text{R-Seal}$$

$$\begin{array}{lrll} \textit{Values} & v & ::= & () \mid \lambda x:\tau.e \mid (v_1, v_2) \mid p.c\ v \\ \textit{Evaluation contexts} & \kappa & ::= & \{\} \mid \kappa\, e \mid v\, \kappa \mid (\kappa, e) \mid (v, \kappa) \mid \pi_i(\kappa) \mid p.c\ \kappa \mid \mathsf{case}\ \kappa\ \mathsf{of}\ p.c\ x \Rightarrow e \end{array}$$

$$\begin{array}{rcl} (\lambda x:\tau.e)\ v & \longrightarrow_\beta & [x \mapsto v]e \\ \pi_i(v_1, v_2) & \longrightarrow_\beta & v_i \\ \mathsf{case}\ p.c\ v\ \mathsf{of}\ q.c\ x \Rightarrow e & \longrightarrow_\beta & [x \mapsto v]e \end{array}$$

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

Figure 6: Small-step call-by-value operational semantics using evaluation contexts

Theorem 4.1 (Soundness). *Let a program (m, e) is well-typed. Then the evaluation of e either returns a value or else gives rise to an infinite reduction sequence.*

Rather than proving the soundness of our system directly, we first introduce another type system which breaks type abstraction and signature sealings. Then we prove that the new type system is sound for the operational semantics, by proving the usual progress and preservation properties. We then obtain Theorem 4.1 by proving that if a program P is well-typed in our type system, so is in the new type system.

13

앞으로 할 일 (Future work)

- 고차 모듈 함수 추가
- 결정 가능한 타입 체계 설계
- 타입 유추 알고리즘 설계

발표는 이제 정말 끝입니다.
질문 있으신가요?

Backup slides

복시 문제를 요약하면 (Double vision problem, in short)

- An external name of a type (e.g., Forest.t) is considered as different from the internal implementation of the same type (e.g., Tree.t list) inside the module that defines the type.

문법구조 (Syntax)

Each module has its own recursion variable annotated with a forward reference signature.

Only forward references via recursion variables are allowed

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	::=	$1 \mid \tau_1 \rightarrow \tau_2 \mid \tau_1 * \tau_2 \mid p.t$
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 $p.c \ x \Rightarrow e' \mid p.l$

타이핑 규칙 맛보기 (Typing rules, excerpted)

$$
\begin{array}{lrcl}
\textit{Module contexts} & \Gamma & ::= & \cdot \mid \Gamma, X : S \\
\textit{Path rewritings} & \Delta & ::= & \cdot \mid \Delta, p \mapsto X \\
\textit{Core typing contexts} & \Sigma & ::= & \cdot \mid \Sigma, x : \tau
\end{array}
$$

Module expressions $\boxed{\Gamma; \Delta; p \vdash m : S}$

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

Definitions $\boxed{\Gamma; \Delta; p \vdash d : D}$

$$\frac{\Gamma; \Delta, p.M \vdash m : S}{\Gamma; \Delta; p \vdash \mathsf{module}\ M = m : \mathsf{module}\ M : S}\ \text{T-MDef} \qquad \frac{\Gamma \vdash \tau \text{ wf}}{\Gamma; \Delta; p \vdash \mathsf{type}\ t = \tau : \mathsf{type}\ t = \tau}\ \text{T-Type}$$

Core expressions $\boxed{\Gamma; \Delta; \Sigma \vdash e : \tau}$

$$\frac{x : \tau \in \Sigma}{\Gamma; \Delta; \Sigma \vdash x : \tau}\ \text{CT-Var} \qquad \frac{}{\Gamma; \Delta; \Sigma \vdash () : 1}\ \text{CT-Unit} \qquad \frac{\Gamma \vdash \tau \text{ wf} \quad \Gamma; \Delta; \Sigma, x : \tau \vdash e : \tau'}{\Gamma; \Delta; \Sigma \vdash \lambda x : \tau.e : \tau \to \tau'}\ \text{CT-Lam}$$

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

라벨 변환 시스템 (Labeled transition system)

- A type equivalence relation as the largest weak bisimulation relation on the labeled transition system of types

Labeled transition rules

$$\Gamma;\Delta \vdash 1 \xrightarrow{1} 0 \qquad \Gamma;\Delta \vdash \tau_1 \to \tau_2 \xrightarrow{\mathrm{ar}_i} \tau_i \qquad \Gamma;\Delta \vdash \tau_1 * \tau_2 \xrightarrow{\mathrm{prd}_i} \tau_i$$

$$\frac{\Gamma \vdash p \ni \mathsf{type}\ t \quad p \mapsto q \notin \Delta}{\Gamma;\Delta \vdash p.t \xrightarrow{p.t} 0}\ \text{EQ-ABS} \qquad \frac{\Gamma \vdash p \ni \mathsf{datatype}\ t = c\ of\ \tau \quad p \mapsto q \notin \Delta}{\Gamma;\Delta \vdash p.t \xrightarrow{p.t} 0}\ \text{EQ-DATA}$$

Silent transition rules

$$\frac{\Gamma \vdash p \ni \mathsf{type}\ t = \tau}{\Gamma;\Delta \vdash p.t \rightharpoonup \tau}\ \text{EQ-TYPE} \qquad \frac{p.t \text{ is an abstract type or a datatype} \quad p \mapsto q \in \Delta}{\Gamma;\Delta \vdash p.t \rightharpoonup q.t}\ \text{EQ-PATH}$$

라벨 변환 시스템 (Labeled transition system)

- A type equivalence relation as the largest weak bisimulation relation on the labeled transition system of types

Labeled transition rules

$$\Gamma;\Delta \vdash 1 \xrightarrow{1} 0 \qquad \Gamma;\Delta \vdash \tau_1 \to \tau_2 \xrightarrow{\mathrm{ar}_i} \tau_i \qquad \Gamma;\Delta \vdash$$

$$\frac{q \notin \Delta}{\quad 0} \text{ EQ-ABS} \qquad \frac{\Gamma \vdash p \ni \text{datatype}}{\Gamma;\Delta \vdash} \text{ EQ-DATA}$$

To support cyclic type definitions

To solve the double vision problem

Silent transition rules

$$\frac{\Gamma \vdash p \ni \text{type } t = \tau}{\Gamma;\Delta \vdash p.t \rightharpoonup \tau} \text{ EQ-TYPE}$$

$$\frac{p.t \text{ is an abstract type or a datatype} \quad p \mapsto q \in \Delta}{\Gamma;\Delta \vdash p.t \rightharpoonup q.t} \text{ EQ-PATH}$$

약한 상호 흉내내기 (Weak bisimulations)

For a binary relation $\mathcal{S}$, we write $\tau_1 \mathcal{S} \tau_2$ to mean $(\tau_1, \tau_2) \in \mathcal{S}$.

Definition 3.1. A binary relation $\mathcal{S}$ over types is a weak bisimulation under context (Γ, Δ) if and only if, whenever $\Gamma; \Delta \vdash \tau_1 \mathcal{S} \tau_2$,

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

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

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

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

We say that τ_1 and τ_2 are weakly bisimilar under (Γ, Δ), written $\Gamma; \Delta \vdash \tau_1 \approx \tau_2$, if there exists a weak bisimulation $\mathcal{S}$ such that $\Gamma; \Delta \vdash \tau_1 \mathcal{S} \tau_2$.

- type t = s * s, type s = t * t |- t ≈ s
- S = {(t, s), (s * s, t * t), (s, t), (t * t, s * s)} is a weak bisimulation.
- $t \rightarrow s * s \rightarrow^{left,right} s \rightarrow t * t \rightarrow^{left,right} t$
- $s \rightarrow t * t \rightarrow^{left,right} t \rightarrow s * s \rightarrow^{left,right} s$

실행과정을 드러내는 의미구조 (Operational semantics)

- A light-weight small-step call-by-value operational semantics using evaluation contexts
- The evaluation of a program (*m, e*) begins by reducing the given expression *e* with respect to the top-level recursive structure *m.*

$$\begin{array}{lrcl} \textit{Values} & v & ::= & () \mid \lambda x : \tau.e \mid (v_1, v_2) \mid p.c\ v \\ \textit{Evaluation contexts} & \kappa & ::= & \{\} \mid \kappa\ e \mid v\ \kappa \mid (\kappa, e) \mid (v, \kappa) \mid \pi_i(\kappa) \mid p.c\ \kappa \mid \mathsf{case}\ \kappa\ \mathsf{of}\ p.c\ x \Rightarrow e \end{array}$$

$$\begin{array}{rcl} (\lambda x.\tau : e)\ v & \rightarrow_\beta & [x \mapsto v]e \\ \pi_i(v_1, v_2) & \rightarrow_\beta & v_i \\ \mathsf{case}\ p.c\ v\ \mathsf{of}\ q.c\ x \Rightarrow e & \rightarrow_\beta & [x \mapsto v]e \end{array}$$

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