단계제거 변환을 통한 다단계 프로그램의 정적분석*

최원태 @ 서울대학교 프로그래밍 연구실

2011년 겨울 워크샵
소프트웨어무결점 연구센터

이 연구는 Baris Aktemur, 이광근, Makoto Tatsuta 와 공동으로 진행하였습니다

*POPL'11 에서 발표예정

한장 요약

$$\lambda_S \longmapsto \lambda_R$$

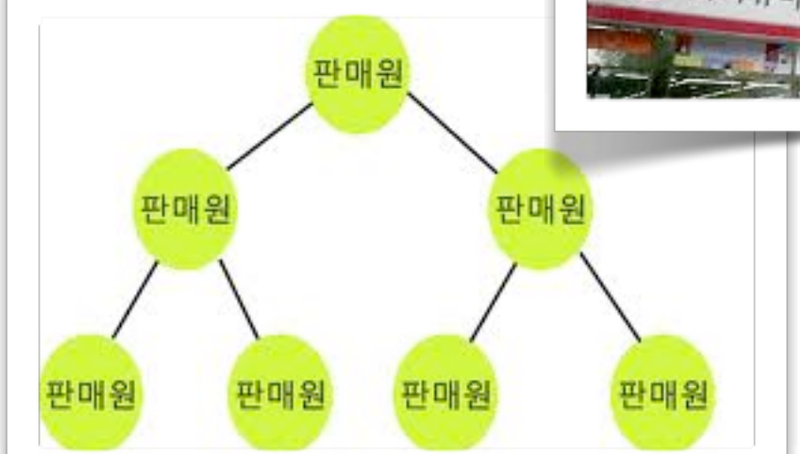
다단계 언어 → 단계 없는 언어

- 왜? 다단계 프로그램을 분석하기 위해
- 무엇을? 단계제거 변환을
- 어떻게? 정의하고 실행의미를 보존함을 증명했다

Intro

- 왜?
- 무엇을?

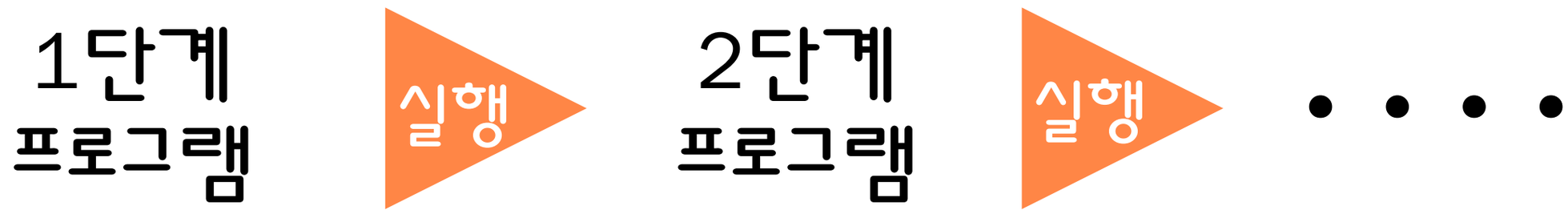
다단계 프로그램?



이런 다단계가 아니라.....

다단계 프로그램

"실행 결과가 다음단계 실행을 위한 프로그램" 인 프로그램



다단계 프로그램의 용도

특정 입력에 특화된 빠른 코드를 만들 때

```
#define Square(x) (x*x)

int main(){
  return Square(3);
}
```

실행

```
int main(){
  return 3*3;
}
```

실행

```
9
```

C macro : 2단계

다단계 프로그램 안전한가?

```c
#define Square(x) (x*x)

int main(){
  return Square(1+2);
}
```

C macro

실행

```c
int main(){
  return 1+2*1+2;
}
```

원하는 것과 다르게 파싱되는 결과물

다단계 프로그램 안전한가?

```
#define LOG(msg) printf(“%d:%s\n”,pid,msg)
void main(){
    //int pid = 0;
    LOG(“some error”);
}
```

실행

```
void main(){
    //int pid = 0;
    printf(“%d;%s”,pid,“some error”);
}
```

실행

```
error: 'pid'
 undeclared
```

변수의 바인딩이 올바르지 않음

어떻게 안전하게 만들까?

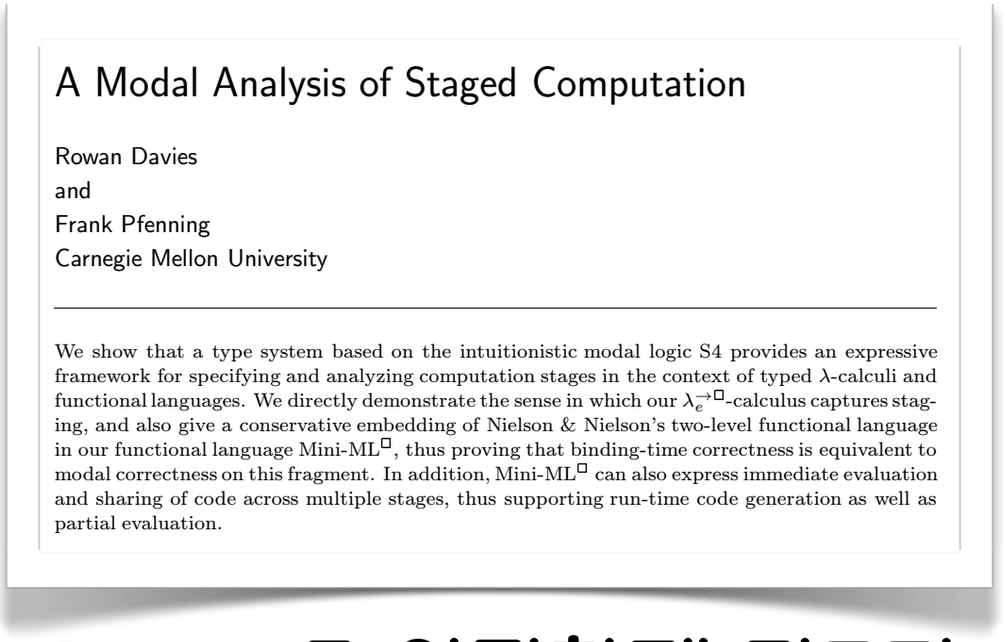
파싱이 이상하게 되지 않도록

```c
#define Square(x) ((x)*(x))

int main(){
  return Square(1+2);
}
```

입력변수를 괄호로 싼다

혹은



A Modal Analysis of Staged Computation

Rowan Davies
and
Frank Pfenning
Carnegie Mellon University

We show that a type system based on the intuitionistic modal logic S4 provides an expressive framework for specifying and analyzing computation stages in the context of typed λ-calculi and functional languages. We directly demonstrate the sense in which our $\lambda_e^{\rightarrow\Box}$-calculus captures staging, and also give a conservative embedding of Nielson & Nielson's two-level functional language in our functional language Mini-ML$^{\Box}$, thus proving that binding-time correctness is equivalent to modal correctness on this fragment. In addition, Mini-ML$^{\Box}$ can also express immediate evaluation and sharing of code across multiple stages, thus supporting run-time code generation as well as partial evaluation.

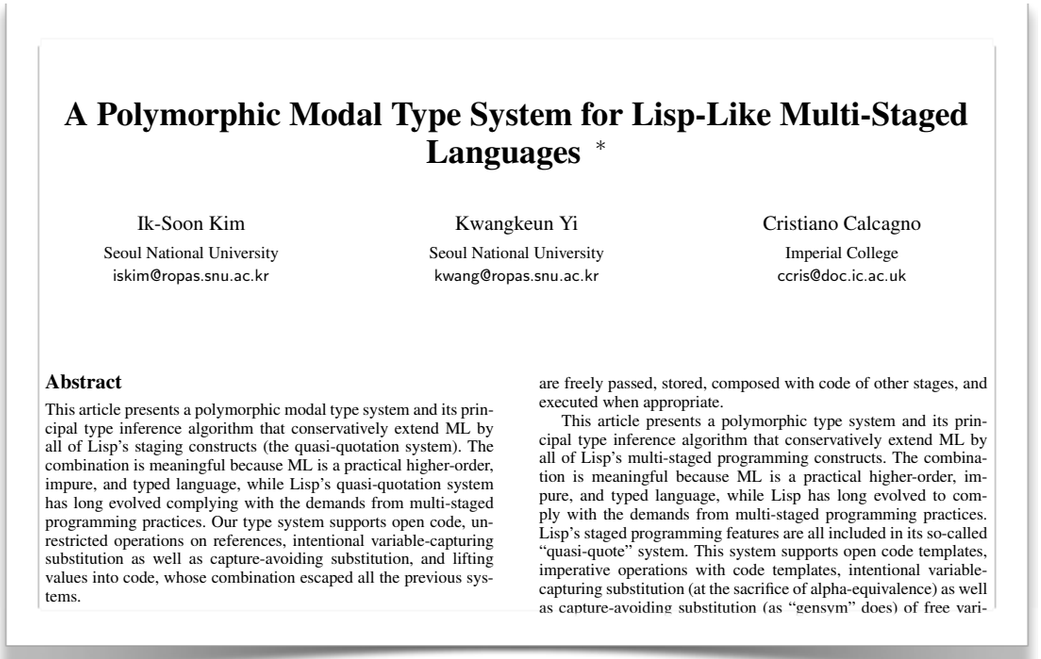
Syntax를 안전하게 만든다

어떻게 안전하게 만들까?

바인딩이 이상하게 되지 않도록

```
#define LOG(msg) printf("%d:%s\n",pid,msg)
void main(){
    int pid = 0;
    LOG("some error");
}
```

혹은



A Polymorphic Modal Type System for Lisp-Like Multi-Staged Languages *

Ik-Soon Kim
Seoul National University
iskim@ropas.snu.ac.kr

Kwangkeun Yi
Seoul National University
kwang@ropas.snu.ac.kr

Cristiano Calcagno
Imperial College
ccris@doc.ic.ac.uk

Abstract

This article presents a polymorphic modal type system and its principal type inference algorithm that conservatively extend ML by all of Lisp's staging constructs (the quasi-quotation system). The combination is meaningful because ML is a practical higher-order, impure, and typed language, while Lisp's quasi-quotation system has long evolved complying with the demands from multi-staged programming practices. Our type system supports open code, unrestricted operations on references, intentional variable-capturing substitution as well as capture-avoiding substitution, and lifting values into code, whose combination escaped all the previous systems.

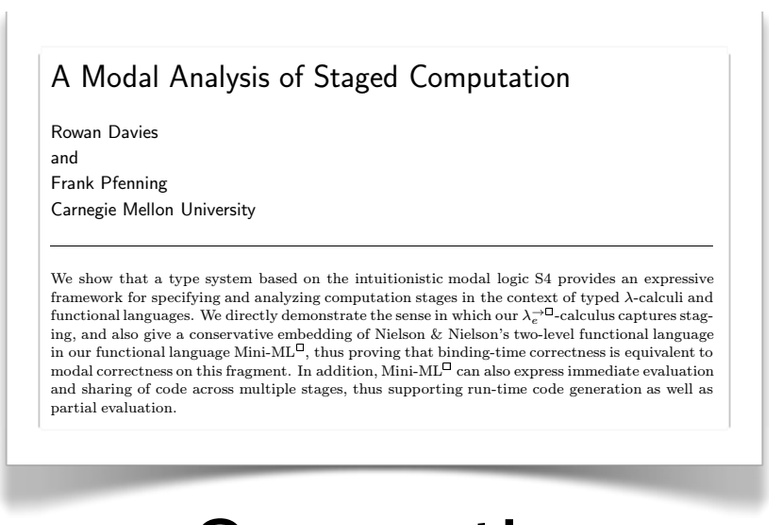
are freely passed, stored, composed with code of other stages, and executed when appropriate.

This article presents a polymorphic type system and its principal type inference algorithm that conservatively extend ML by all of Lisp's multi-staged programming constructs. The combination is meaningful because ML is a practical higher-order, impure, and typed language, while Lisp has long evolved to comply with the demands from multi-staged programming practices. Lisp's staged programming features are all included in its so-called "quasi-quote" system. This system supports open code templates, imperative operations with code templates, intentional variable-capturing substitution (at the sacrifice of alpha-equivalence) as well as capture-avoiding substitution (as "gensym" does) of free vari-

조심스럽게 프로그래밍

Type으로 (매크로 확장 전에) 미리 잡아낸다

어떻게 더욱 안전하게 만들까?

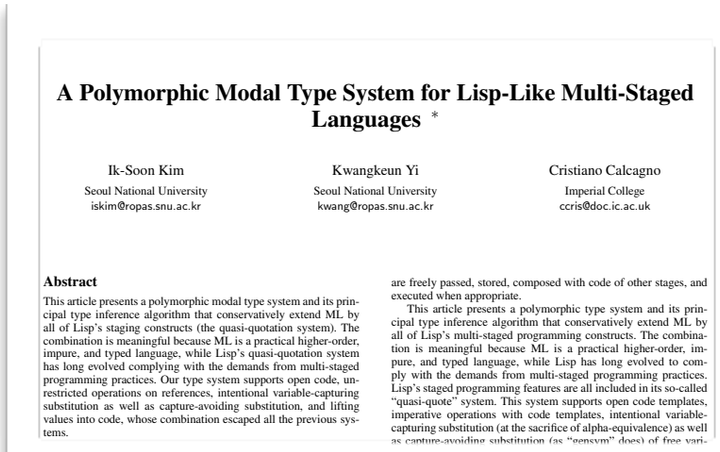


A Modal Analysis of Staged Computation

Rowan Davies
and
Frank Pfenning
Carnegie Mellon University

We show that a type system based on the intuitionistic modal logic S4 provides an expressive framework for specifying and analyzing computation stages in the context of typed λ-calculi and functional languages. We directly demonstrate the sense in which our $\lambda_c^{\to\Box}$-calculus captures staging, and also give a conservative embedding of Nielson & Nielson's two-level functional language in our functional language Mini-ML$^{\Box}$, thus proving that binding-time correctness is equivalent to modal correctness on this fragment. In addition, Mini-ML$^{\Box}$ can also express immediate evaluation and sharing of code across multiple stages, thus supporting run-time code generation as well as partial evaluation.

Semantics



A Polymorphic Modal Type System for Lisp-Like Multi-Staged Languages *

Ik-Soon Kim
Seoul National University
iskim@ropas.snu.ac.kr

Kwangkeun Yi
Seoul National University
kwang@ropas.snu.ac.kr

Cristiano Calcagno
Imperial College
ccris@doc.ic.ac.uk

Abstract

This article presents a polymorphic modal type system and its principal type inference algorithm that conservatively extend ML by all of Lisp's staging constructs (the quasi-quotation system). The combination is meaningful because ML is a practical higher-order, impure, and typed language, while Lisp's quasi-quotation system has long evolved complying with the demands from multi-staged programming practices. Our type system supports open code, unrestricted operations on references, intentional variable-capturing substitution as well as capture-avoiding substitution, and lifting values into code, whose combination escaped all the previous systems.

are freely passed, stored, composed with code of other stages, and executed when appropriate.

This article presents a polymorphic type system and its principal type inference algorithm that conservatively extend ML by all of Lisp's multi-staged programming constructs. The combination is meaningful because ML is a practical higher-order, impure, and typed language, while Lisp has long evolved to comply with the demands from multi-staged programming practices. Lisp's staged programming features are all included in its so-called "quasi-quote" system. This system supports open code templates, imperative operations with code templates, intentional variable-capturing substitution (at the sacrifice of alpha-equivalence) as well as capture-avoiding substitution (as "gensym" does) of free vari-

Types

?

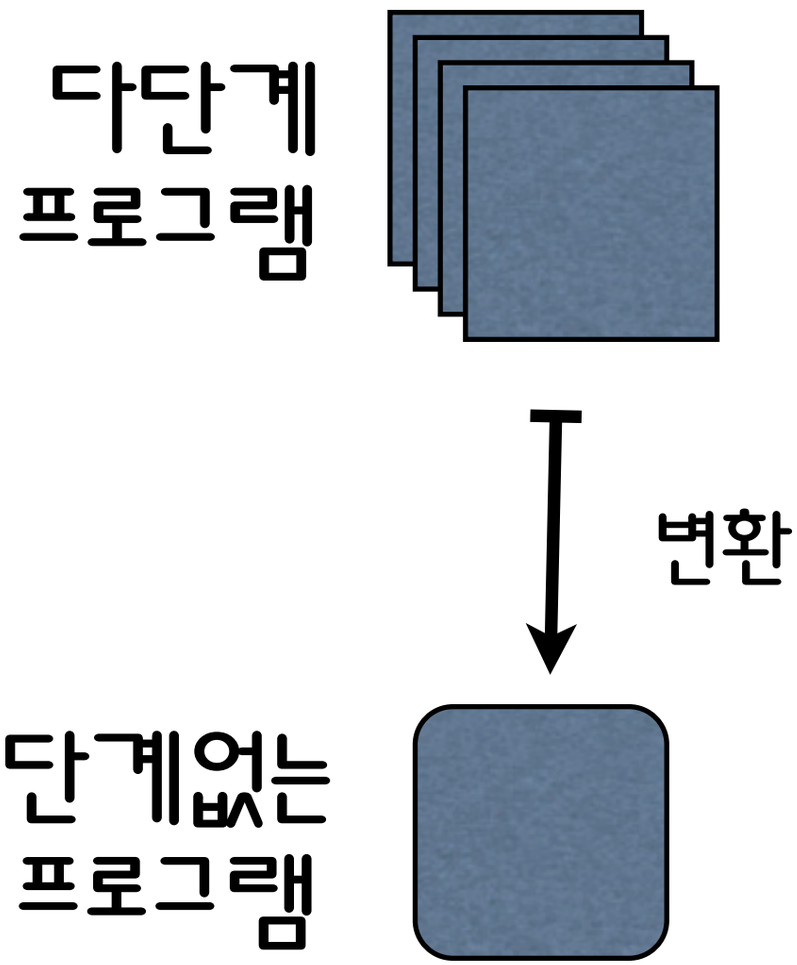
Static Analysis

그런데 직접 분석은 어렵다

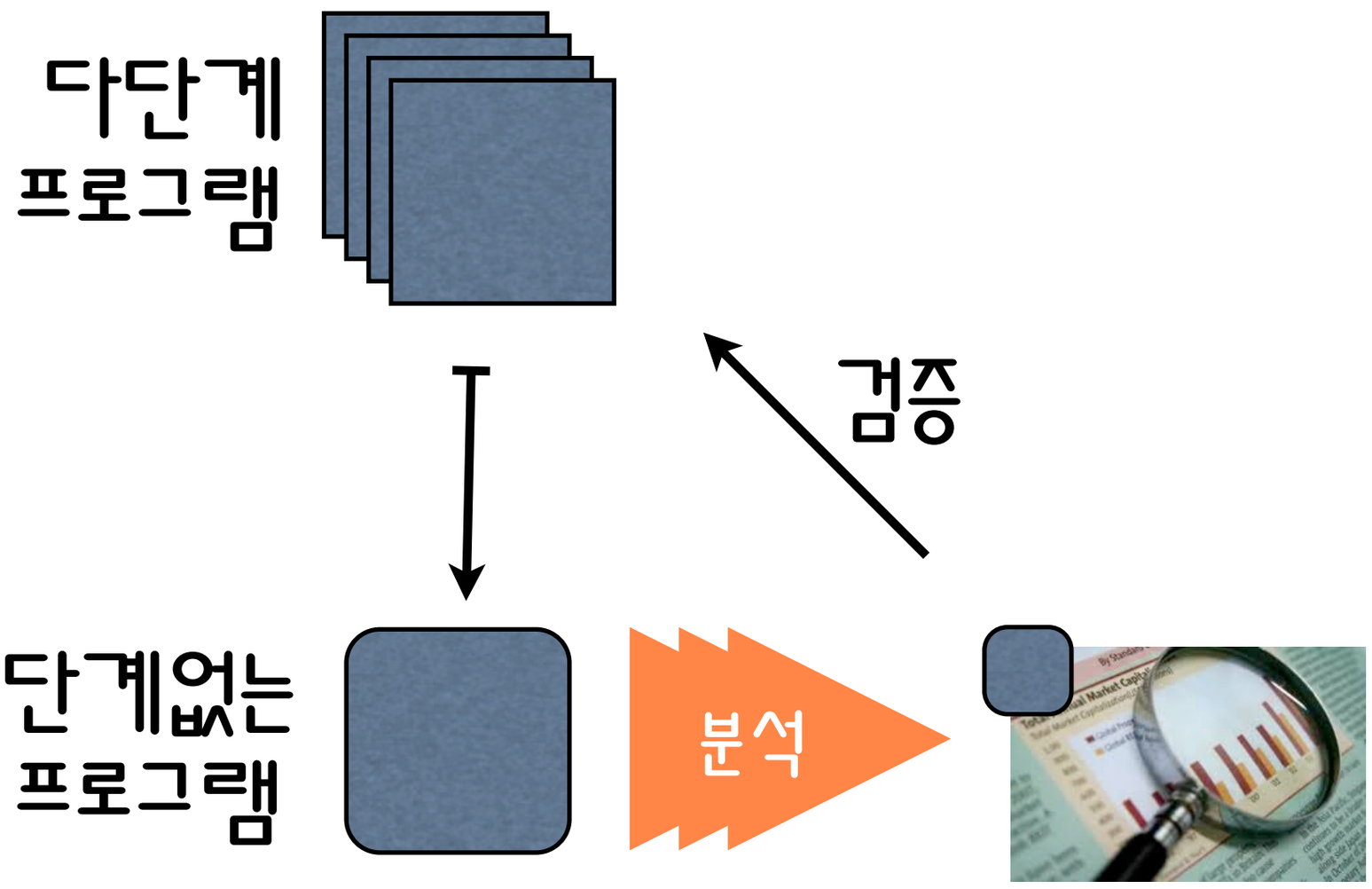
코드가 돌아다닌다

코드가 값이다 ⟹ 코드가 만들어진다 + 만들어진 코드가 실행된다 ⟹ 분석할 코드가 불분명하다

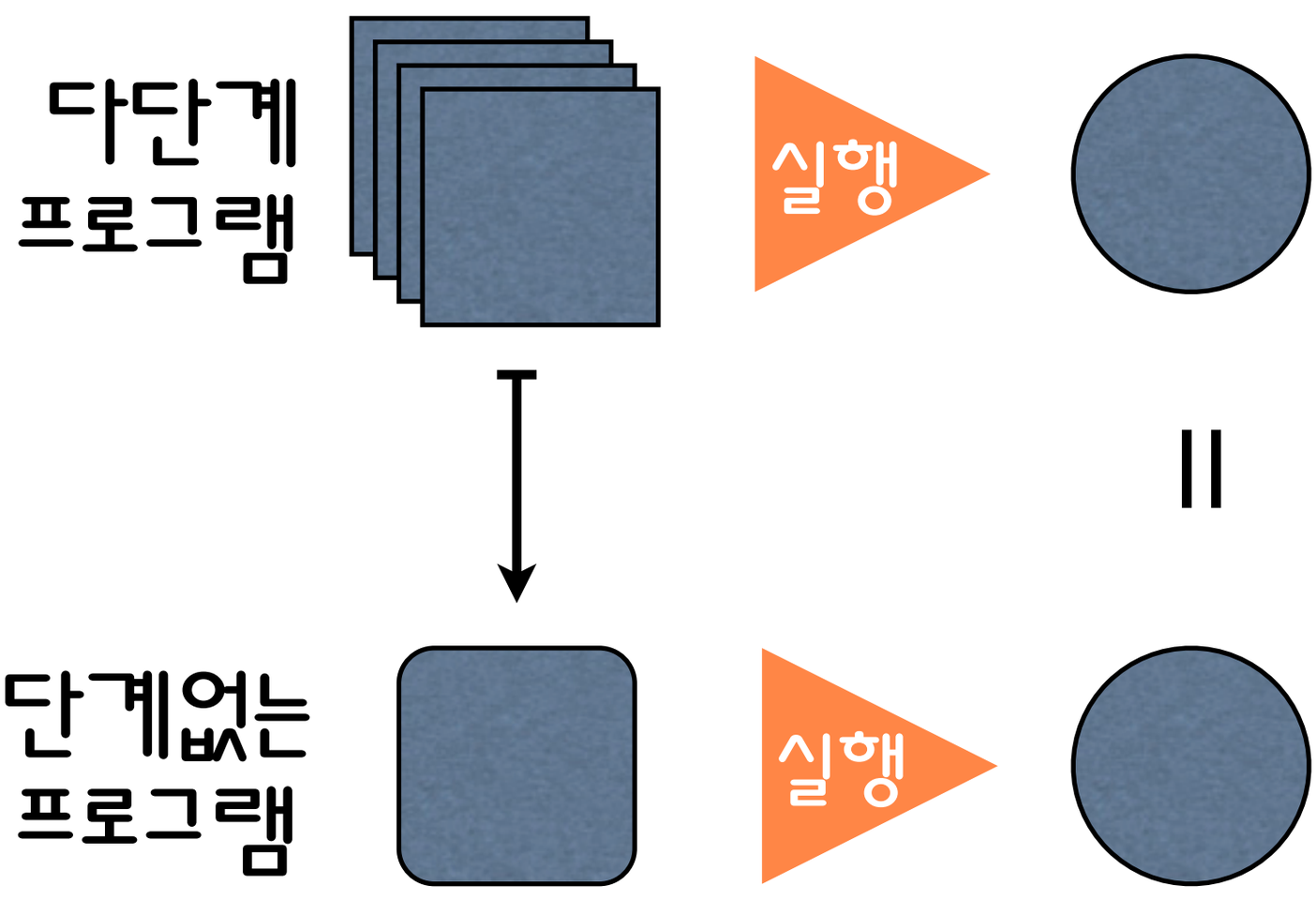
아이디어 :
변환을 통해 단계를 제거하자



좋은 변환이 있으면 :
직접 분석하지 않아도 된다



변환의 조건 :
실행의미를 보존해야



매크로 펴면 되는거 아냐?

```
#define Square(x) (x*x)

int main(){
  return Square(3);
}
```

실행

```

int main(){
  return 3*3;
}
```

단순히 매크로를 펴는 것과는 다르다

```
(define (f n)
    (if (= n 1)
        (`(x+,(f (n-1))))
        `x))

(define x 2)
(eval (f (get)))
```

n=1 => “x”
n=2 => “x+x”
n=3 => “x+x+x”

get=1 => 2
get=2 => 4
get=3 => 6

보통은 여러 단계의 실행이 서로 얽혀있음

Translation

- 무엇을?
- 어떻게?

구체적 목표

A Polymorphic Modal Type System for Lisp-Like Multi-Staged Languages *

Ik-Soon Kim
Seoul National University
iskim@ropas.snu.ac.kr

Kwangkeun Yi
Seoul National University
kwang@ropas.snu.ac.kr

Cristiano Calcagno
Imperial College
ccris@doc.ic.ac.uk

Abstract

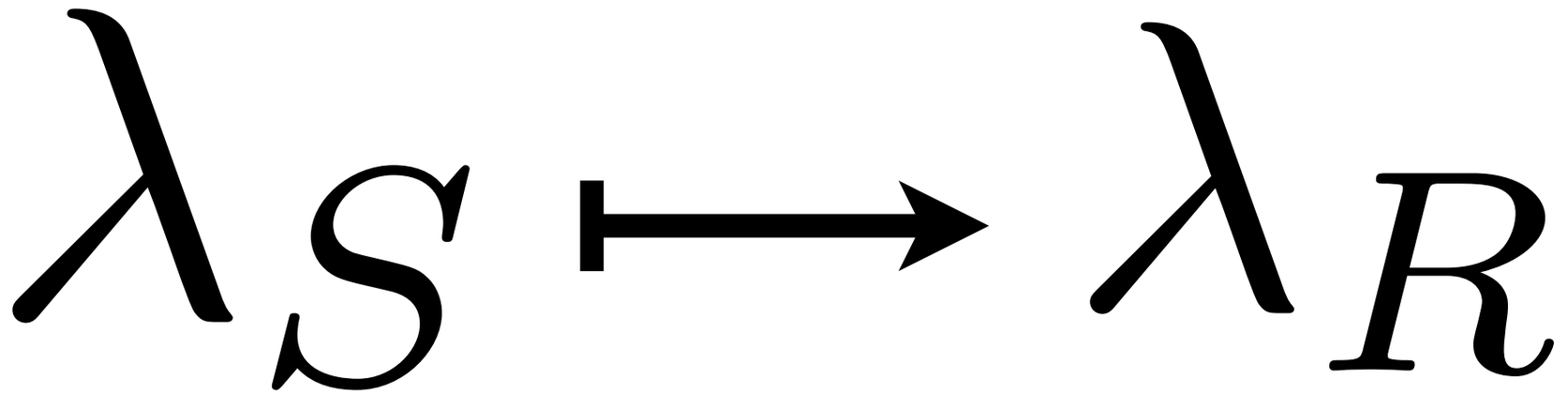
This article presents a polymorphic modal type system and its principal type inference algorithm that conservatively extend ML by all of Lisp's staging constructs (the quasi-quotation system). The combination is meaningful because ML is a practical higher-order, impure, and typed language, while Lisp's quasi-quotation system has long evolved complying with the demands from multi-staged programming practices. Our type system supports open code, unrestricted operations on references, intentional variable-capturing substitution as well as capture-avoiding substitution, and lifting values into code, whose combination escaped all the previous systems.

are freely passed, stored, composed with code of other stages, and executed when appropriate.

This article presents a polymorphic type system and its principal type inference algorithm that conservatively extend ML by all of Lisp's multi-staged programming constructs. The combination is meaningful because ML is a practical higher-order, impure, and typed language, while Lisp has long evolved to comply with the demands from multi-staged programming practices. Lisp's staged programming features are all included in its so-called "quasi-quote" system. This system supports open code templates, imperative operations with code templates, intentional variable-capturing substitution (at the sacrifice of alpha-equivalence) as well as capture-avoiding substitution (as "gensym" does) of free vari-

이 논문에서 다루는 언어를 위한 변환을 찾자

큰 그림



다단계 언어 단계 없는 언어 + 레코드

- 코드는 함수로
- 코드 실행은 함수호출로
- 변수 바인딩은 레코드로

문법

다단계 $e := \lambda x.e \mid ee \mid x \mid \text{`}e \mid \text{,}e \mid \texttt{run}\ e$

단계없는 $e := \lambda x.e \mid ee \mid x \mid \{\} \mid e\{\texttt{x} = e\} \mid e.\texttt{x}$

문법

다단계 $e := \lambda x.e \mid ee \mid x \mid \text{`}e \mid \text{,}e \mid \texttt{run}\ e$

단계없는 $e := \lambda x.e \mid ee \mid x \mid \{\} \mid e\{\texttt{x} = e\} \mid e.\texttt{x}$

특별하지 않은 레코드 연산들

문법

다단계 $e := \lambda x.e \mid ee \mid x \mid \text{`}e \mid \text{,}e \mid \texttt{run}\ e$

다단계 코드 관련된 연산들

단계없는 $e := \lambda x.e \mid ee \mid x \mid \{\} \mid e\{\texttt{x} = e\} \mid e.\texttt{x}$

실행 의미

Syntax $e := \lambda x.e \mid ee \mid x \mid \text{`}e \mid \text{,}e \mid \texttt{run}\ e$

코드를 정의한다

실행 의미

Syntax $e := \lambda x.e \mid ee \mid x \mid \text{`}e \mid \text{,}e \mid \texttt{run}\ e$

$$1 + 1 \notin \mathit{Value}$$

코드를 정의한다

실행 의미

Syntax $e := \lambda x.e \mid ee \mid x \mid \text{`}e \mid \text{,}e \mid \texttt{run}\ e$

$$1 + 1 \notin Value$$

$$\text{`}(1 + 1) \in Value$$

코드를 정의한다

실행 의미

Syntax $e := \lambda x.e \mid ee \mid x \mid \text{`}e \mid \text{,}e \mid \texttt{run}\ e$

코드를 실행한다

실행 의미

Syntax $e := \lambda x.e \mid ee \mid x \mid \text{`}e \mid \text{,}e \mid \boxed{\texttt{run}\ e}$

$\texttt{run}\ \text{`}(1+1) \blacktriangleright 2$

코드를 실행한다

변환

Syntax $e := \lambda x.e \mid ee \mid x \mid \text{`}e \mid \text{,}e \mid \texttt{run}\ e$

$\texttt{run}\ \text{`}(1+1) \quad \blacktriangleright \quad 2$

변환

Syntax $e := \lambda x.e \mid ee \mid x \mid \text{`}e \mid \text{,}e \mid \texttt{run}\ e$

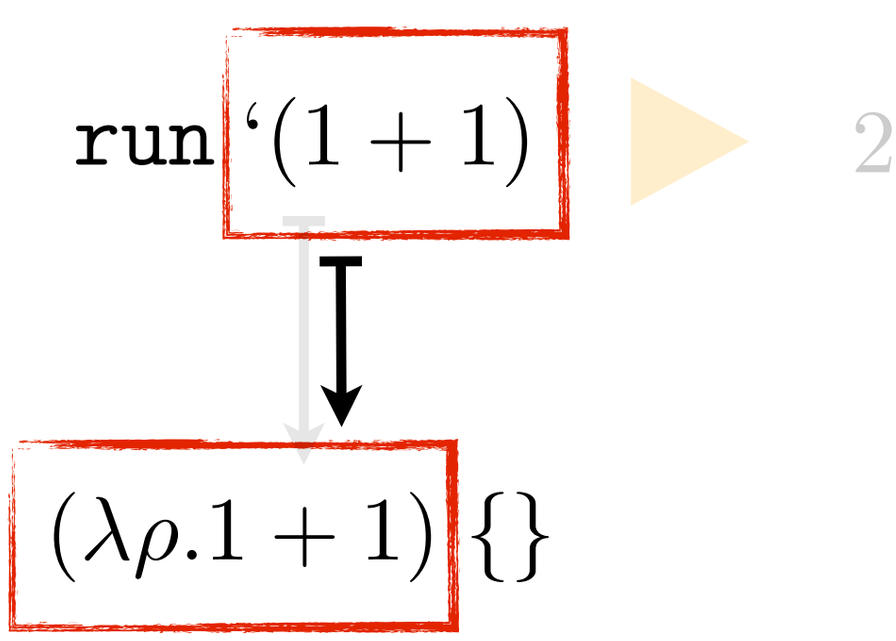
$\texttt{run}\ \text{`}(1+1) \blacktriangleright 2$

$\downarrow$

$(\lambda\rho.1+1)\ \{\}$

변환

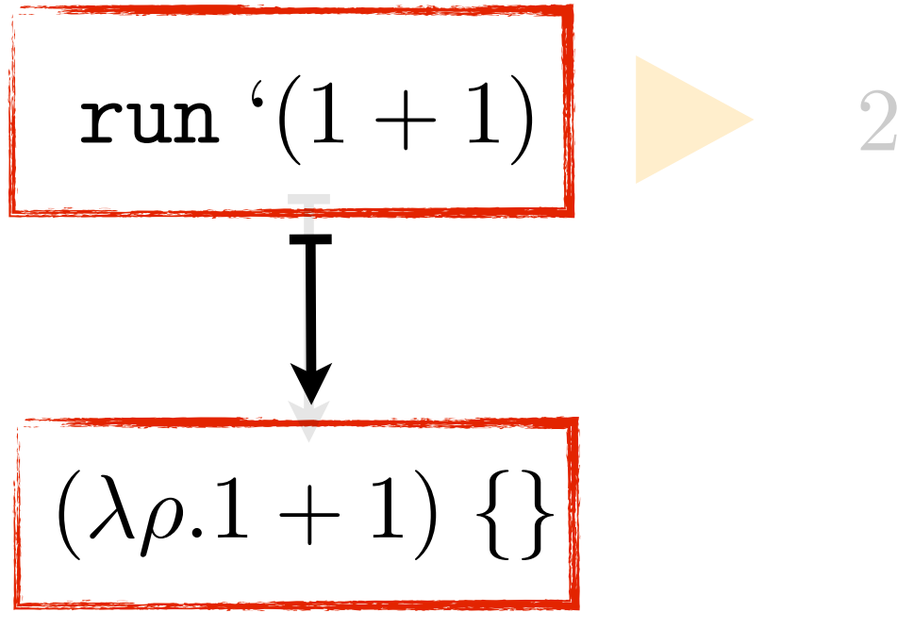
Syntax $e := \lambda x.e \mid ee \mid x \mid \text{`}e \mid \text{,}e \mid \texttt{run}\ e$



코드정의는 함수로 변환

변환

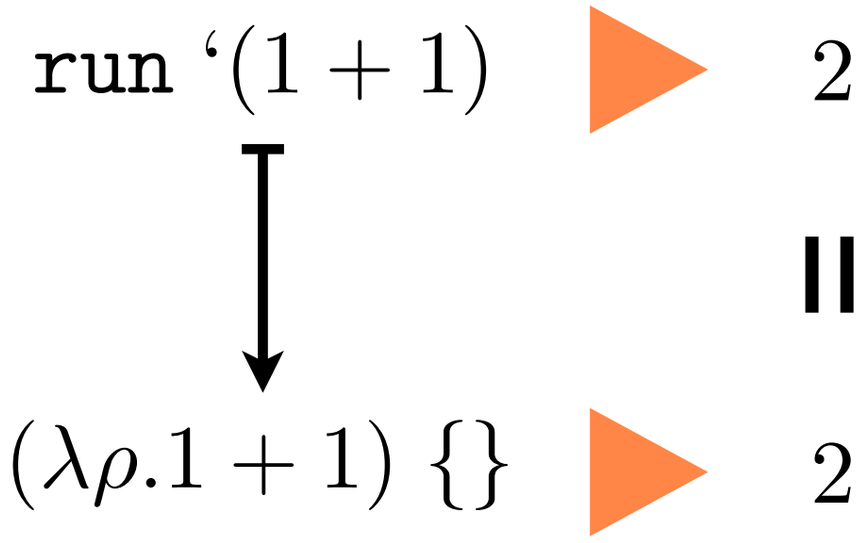
Syntax $e := \lambda x.e \mid ee \mid x \mid \text{`}e \mid \text{,}e \mid \texttt{run}\ e$



코드 실행은 함수 호출로 변환

변환

Syntax $e := \lambda x.e \mid ee \mid x \mid \text{`}e \mid \text{,}e \mid \texttt{run}\ e$



실행의미

Syntax $e := \lambda x.e \mid ee \mid x \mid \text{`}e \mid \boxed{,e} \mid \texttt{run}\ e$

$\text{`}(1+,\text{`}1)$ ▶ $\text{`}(1+1)$

코드를 다른 코드에 끼워넣는다

실행의미

Syntax $e := \lambda x.e \mid ee \mid x \mid \text{`}e \mid \boxed{,e} \mid \texttt{run}\ e$

$$\text{`}(1+, \text{`}1) \blacktriangleright \text{`}(1+1)$$

$$\text{`}(1+, ((\lambda x.x)\ \text{`}1)) \blacktriangleright \text{`}(1+1)$$

, 안의 표현식을 실행하고
결과 코드를 다른 코드에 끼워넣는다

변환

Syntax $e := \lambda x.e \mid ee \mid x \mid \text{`}e \mid \text{,}e \mid \texttt{run}\ e$

$\text{`}(1+, ((\lambda x.x)\ \text{`}1))$

$(\lambda h.\lambda p.1 + hp)\ ((\lambda x.x)\ \lambda\rho.1)$

$\text{`}(1 + 1)$

변환

Syntax $e := \lambda x.e \mid ee \mid x \mid \text{`}e \mid \text{,}e \mid \texttt{run}\ e$

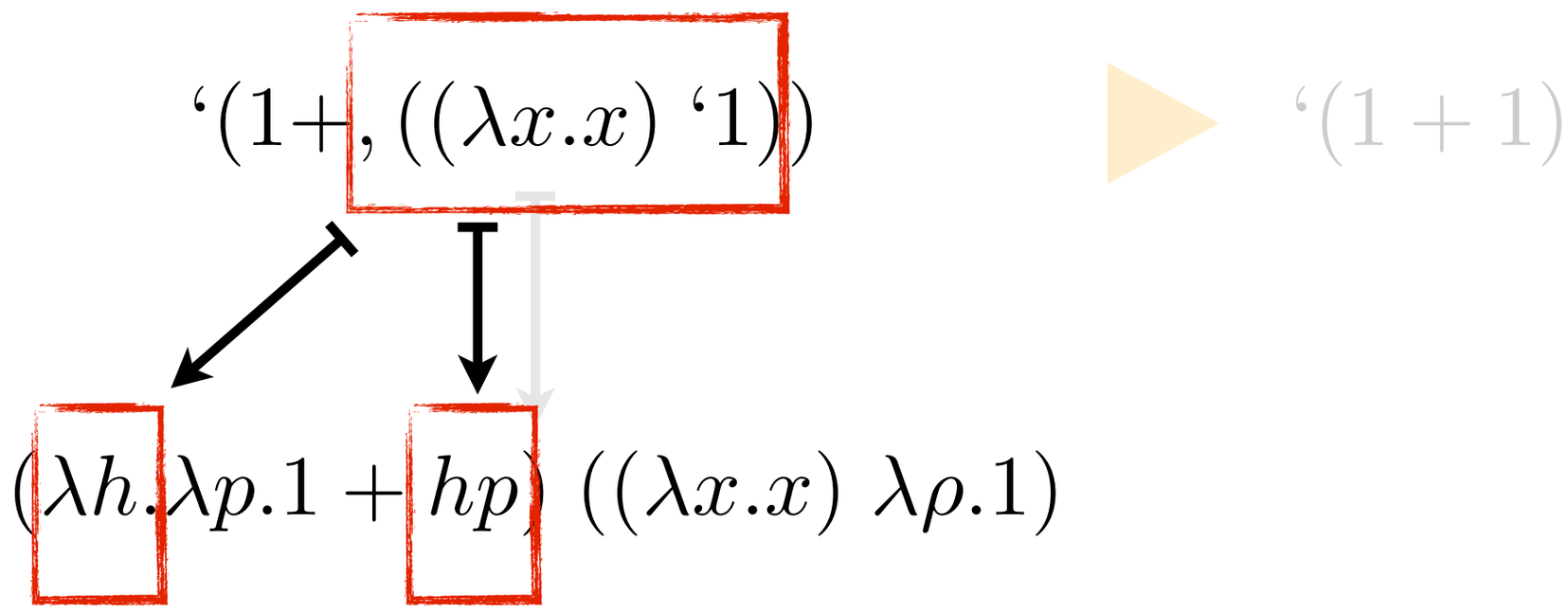
$\text{`}(1+, ((\lambda x.x)\ \text{`}1)) \blacktriangleright \text{`}(1+1)$

$(\lambda h.\lambda p.1 + hp)\ ((\lambda x.x)\ \lambda\rho.1)$

,안의 표현식은 코드 밖으로 이동시켜 실행한다

변환

Syntax $e := \lambda x.e \mid ee \mid x \mid \text{`}e \mid ,e \mid \texttt{run}\ e$



실행 결과를 제자리에 돌려놓을 장치들

변환

Syntax $e := \lambda x.e \mid ee \mid x \mid \text{`}e \mid ,e \mid \texttt{run}\ e$

$\text{`}(1+, ((\lambda x.x)\ \text{`}1))$ ▶ $\text{`}(1+1)$

$(\lambda h.\lambda p.1 + hp)\ ((\lambda x.x)\ \lambda\rho.1)$

코드자체는 여전히 함수로 변환

변환

Syntax $e := \lambda x.e \mid ee \mid x \mid \text{`}e \mid ,e \mid \texttt{run}\ e$

$$\text{`}(1+, ((\lambda x.x)\ \text{`}1)) \blacktriangleright \text{`}(1+1)$$

$$\downarrow \qquad\qquad \| $$

$$(\lambda h.\lambda p.1 + hp)\ ((\lambda x.x)\ \lambda\rho.1) \blacktriangleright \lambda\rho.1+1$$

실행의미

Syntax $e := \lambda x.e \mid ee \mid x \mid \text{`}e \mid \text{,}e \mid \texttt{run}\ e$

$\text{`}(x+1) \in Value$

코드는 자유변수를 가질 수 있다

변환

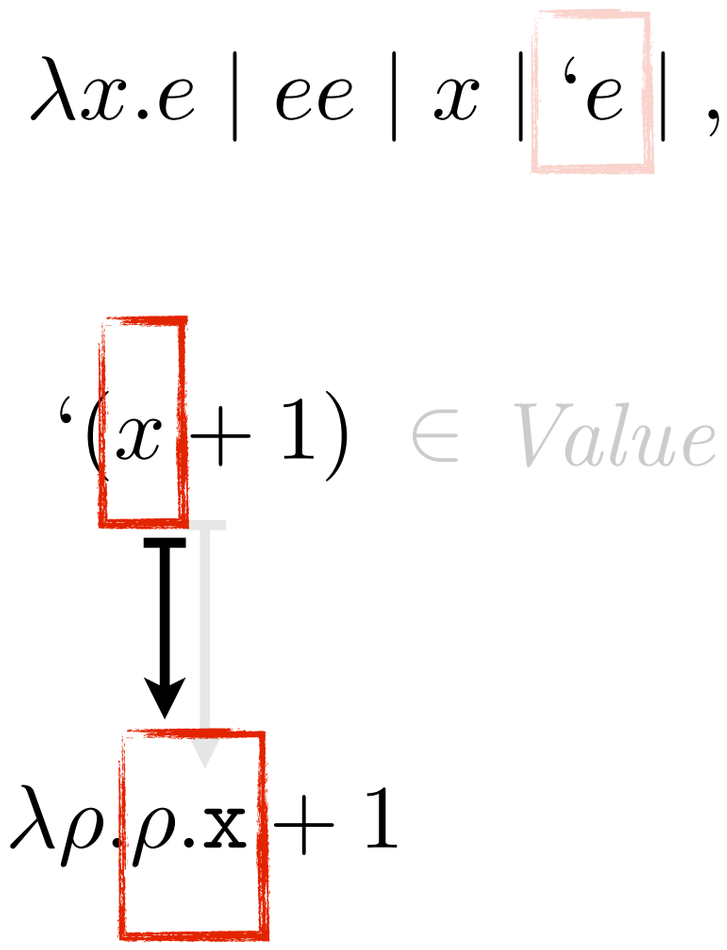
Syntax $e := \lambda x.e \mid ee \mid x \mid \text{`}e \mid \text{,}e \mid \texttt{run}\ e$

$\text{`}(x+1) \in \mathit{Value}$

$\lambda\rho.\rho.\mathtt{x}+1$

변환

Syntax $e := \lambda x.e \mid ee \mid x \mid \text{`}e \mid \text{,}e \mid \texttt{run}\ e$



자유변수는 레코드 접근으로 변환

변환

Syntax $e := \lambda x.e \mid ee \mid x \mid \text{`}e \mid \text{,}e \mid \texttt{run}\ e$

$$\text{`}(x+1) \in Value$$

$$\lambda\rho.\rho.\mathtt{x}+1 \in Value$$

실행 의미

Syntax $e := \lambda x.e \mid ee \mid x \mid \text{`}e \mid ,e \mid \texttt{run}\ e$

$\text{`}(\lambda x.,\text{`}x) \blacktriangleright \text{`}(\lambda x.x)$

자유변수는 코드가 조립될때 해소된다

변환

Syntax $e := \lambda x.e \mid ee \mid x \mid \text{`}e \mid ,e \mid \texttt{run}\ e$

$\text{`}(\lambda x.,\text{`}x)$

$(\lambda h.\lambda\rho.\lambda x.h\ (\rho\{\texttt{x} = x\}))\ \lambda\rho.\rho.\texttt{x}$

변환

Syntax $e := \lambda x.e \mid ee \mid x \mid \text{`}e \mid ,e \mid \texttt{run}\ e$

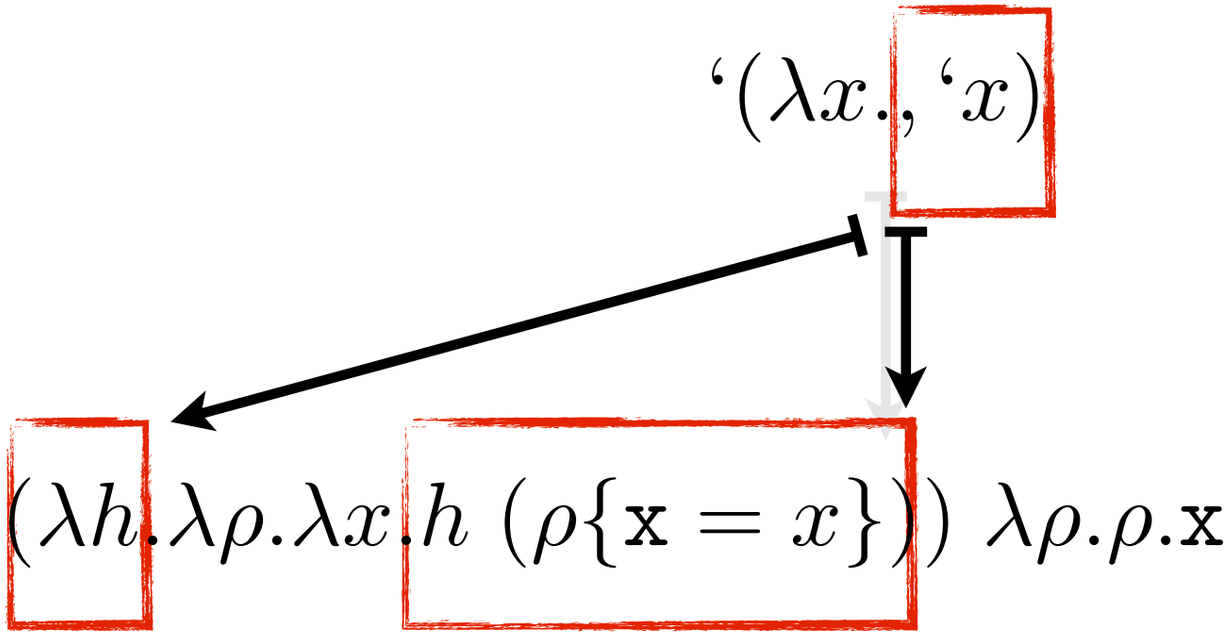
$\text{`}(\lambda x.,\text{`}x)$ ▶ $\text{`}(\lambda x.x)$

$(\lambda h.\lambda\rho.\lambda x.h\ (\rho\{\texttt{x} = x\}))\ \lambda\rho.\rho.\texttt{x}$

변환

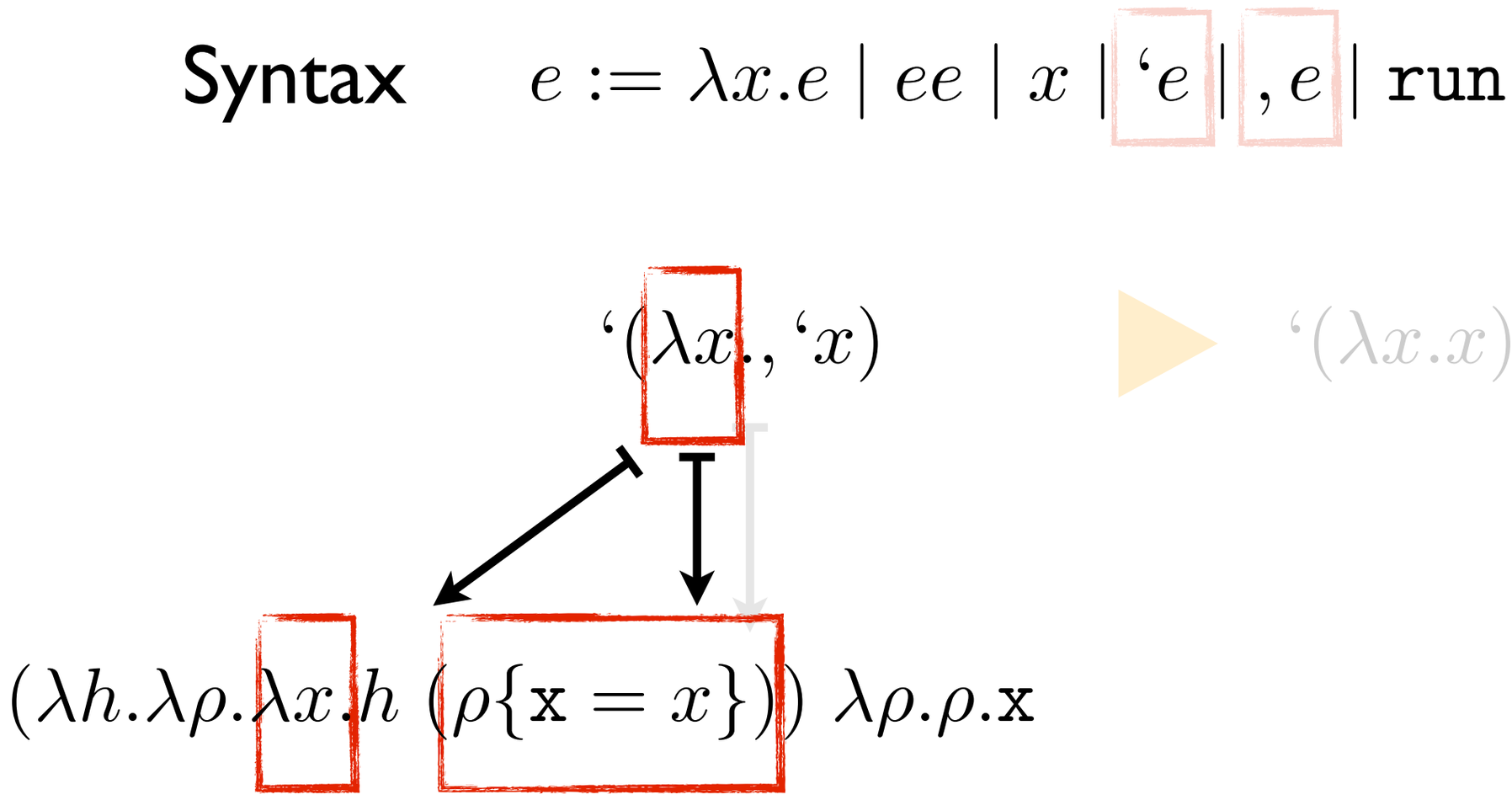
Syntax $e := \lambda x.e \mid ee \mid x \mid \text{`}e \mid ,e \mid \texttt{run}\ e$

$\text{`}(\lambda x.,\text{`}x)$ ▶ $\text{`}(\lambda x.x)$



변환

Syntax $e := \lambda x.e \mid ee \mid x \mid \text{`}e \mid ,e \mid \texttt{run}\ e$



변수의 바인딩은 레코드를 통해 전달된다

변환

Syntax $e := \lambda x.e \mid ee \mid x \mid \text{`}e \mid ,e \mid \texttt{run}\ e$

$\text{`}(\lambda x.,\text{`}x) \blacktriangleright \text{`}(\lambda x.x)$

$\downarrow$

$(\lambda h.\lambda\rho.\lambda x.h\ (\rho\{\texttt{x} = x\}))\ \lambda\rho.\rho.\texttt{x} \blacktriangleright \lambda\rho.\lambda x.x$

변환의 수학적 정의

Definitions

$$\begin{array}{lll} \textit{Environment} & r ::= & \{\} \mid \rho \mid r+\{\mathtt{x}=x\} \\ \textit{Environment Stack} & R ::= & \bot \mid R, r \end{array}$$

$$\begin{array}{lll} \textit{Context} & \kappa ::= & ((\lambda h.[\cdot])\ e) \mid ((\lambda h.\kappa)\ e) \\ \textit{Context Stack} & K ::= & \bot \mid K, \kappa \end{array}$$

Environment Lookup

$$r(\mathtt{x}) = \begin{cases} x & \text{if } r = r'+\{\mathtt{x}=x\} \\ r'(\mathtt{x}) & \text{if } r = r'+\{\mathtt{y}=\_\} \text{ and } \mathtt{x} \neq \mathtt{y} \\ \rho\cdot\mathtt{x} & \text{if } r = \rho \end{cases}$$

Context Stack Merge Operator

$$\begin{aligned} \bot \bowtie K &= K \\ K \bowtie \bot &= K \\ (K_1, \kappa_1) \bowtie (K_2, \kappa_2) &= (K_1 \bowtie K_2), (\kappa_1[\kappa_2]) \end{aligned}$$

Term Translation

(TCON) $$R \vdash i \mapsto (i, \bot)$$

(TVAR) $$R, r \vdash x \mapsto (r(\mathtt{x}), \bot)$$

(TABS) $$\frac{R, r+\{\mathtt{x}=x\} \vdash e \mapsto (\underline{e}, K)}{R, r \vdash \lambda x.e \mapsto (\lambda x.\underline{e}, K)}$$

(TFIX) $$\frac{R, r+\{\mathtt{x}=x\}+\{\mathtt{f}=f\} \vdash e \mapsto (\underline{e}, K)}{R, r \vdash \mathtt{fix}\ f\ x.e \mapsto (\mathtt{fix}\ f\ x.\underline{e}, K)}$$

(TAPP) $$\frac{R \vdash e_1 \mapsto (\underline{e_1}, K_1) \quad R \vdash e_2 \mapsto (\underline{e_2}, K_2)}{R \vdash e_1\ e_2 \mapsto (\underline{e_1}\ \underline{e_2}, K_1 \bowtie K_2)}$$

(TBOX) $$\frac{R, \rho \vdash e \mapsto (\underline{e}, (K, \kappa))}{R \vdash \mathtt{box}\ e \mapsto (\kappa[\lambda\rho.\underline{e}], K)}\ \text{new } \rho$$

$$\frac{R, \rho \vdash e \mapsto (\underline{e}, \bot)}{R \vdash \mathtt{box}\ e \mapsto (\lambda\rho.\underline{e}, \bot)}\ \text{new } \rho$$

(TUNB) $$\frac{R \vdash e \mapsto (\underline{e}, K)}{R, r \vdash \mathtt{unbox}\ e \mapsto (h\ r, (K, (\lambda h.[\cdot])\ \underline{e}))}\ \text{new } h$$

(TRUN) $$\frac{R \vdash e \mapsto (\underline{e}, K)}{R \vdash \mathtt{run}\ e \mapsto (\mathtt{let}\ \ h = \underline{e}\ \mathtt{in}\ (h\,\{\}), K)}\ \text{new } h$$

변환의 수학적 정의

Definitions

Environment $r ::= \{\} \mid \rho \mid r+\{\mathtt{x}=x\}$

Environment Stack $R ::= \bot \mid R, r$

Context $\kappa ::= ((\lambda h.[\cdot])\ e) \mid ((\lambda h.\kappa)\ e)$

Context Stack $K ::= \bot \mid K, \kappa$

Environment Lookup

$$r(\mathtt{x}) = \begin{cases} x & \text{if } r = r'+\{ \\ r'(\mathtt{x}) & \text{if } r = r'+\{ \\ \rho\cdot\mathtt{x} & \text{if } r = \rho \end{cases}$$

Context Stack Merge Operator

$$\bot \bowtie K = K$$
$$K \bowtie \bot = K$$
$$(K_1, \kappa_1) \bowtie (K_2, \kappa_2) = (K_1 \bowtie K_2), (\kappa_1[\kappa_2])$$

잘 정의되었음

Term Translation

(TCON) $$R \vdash i \mapsto (i, \bot)$$

(TVAR) $$R, r \vdash x \mapsto (r(\mathtt{x}), \bot)$$

(TABS) $$\frac{R, r+\{\mathtt{x}=x\} \vdash e \mapsto (\underline{e}, K)}{R, r \vdash \lambda x.e \mapsto (\lambda x.\underline{e}, K)}$$

(TFIX) $$\frac{R, r+\{\mathtt{x}=x\}+\{\mathtt{f}=f\} \vdash e \mapsto (\underline{e}, K)}{R, r \vdash \mathtt{fix}\ f\ x.e \mapsto (\mathtt{fix}\ f\ x.\underline{e}, K)}$$

$$\frac{(\underline{e_1}, K_1) \quad R \vdash e_2 \mapsto (\underline{e_2}, K_2)}{\ e_2 \mapsto (\underline{e_1}\ \underline{e_2}, K_1 \bowtie K_2)}$$

$$\frac{\vdash e \mapsto (\underline{e}, (K, \kappa))}{\mathtt{box}\ e \mapsto (\kappa[\lambda\rho.\underline{e}], K)}\ \text{new } \rho$$

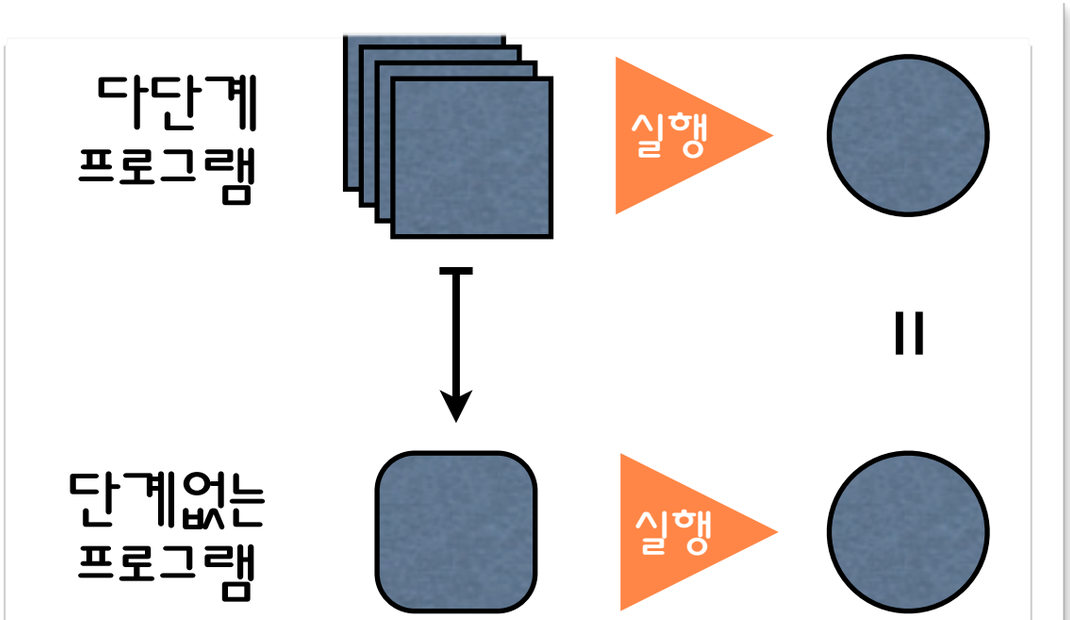
$$\frac{R, \rho \vdash e \mapsto (\underline{e}, \bot)}{R \vdash \mathtt{box}\ e \mapsto (\lambda\rho.\underline{e}, \bot)}\ \text{new } \rho$$

(TUNB) $$\frac{R \vdash e \mapsto (\underline{e}, K)}{R, r \vdash \mathtt{unbox}\ e \mapsto (h\ r, (K, (\lambda h.[\cdot])\ \underline{e}))}\ \text{new } h$$

(TRUN) $$\frac{R \vdash e \mapsto (\underline{e}, K)}{R \vdash \mathtt{run}\ e \mapsto (\mathtt{let}\ \ h = \underline{e}\ \mathtt{in}\ (h\,\{\}), K)}\ \text{new } h$$

변환은 실행의미를 보존

요구사항

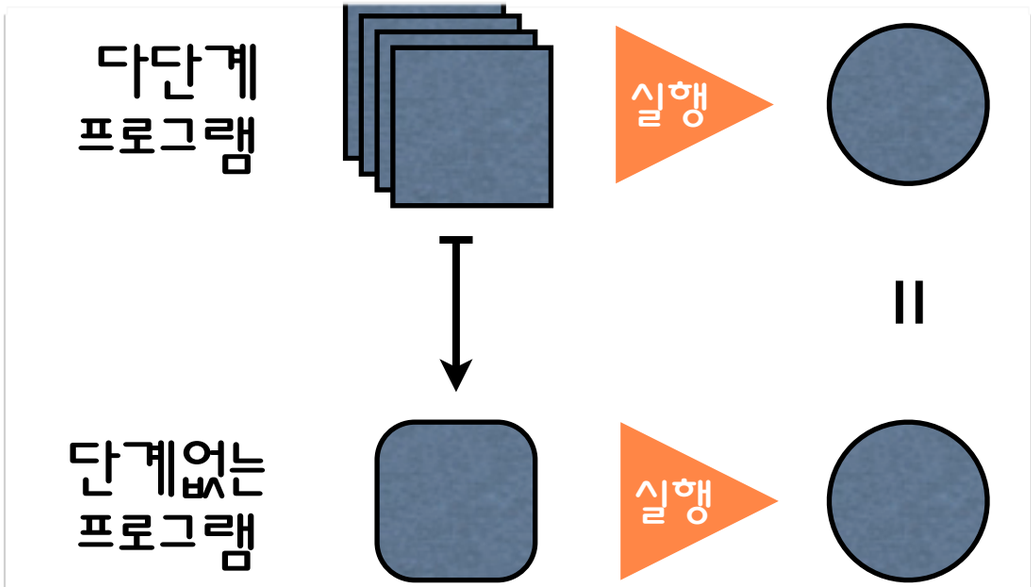


증명한 것

Theorem 1. *(Simulation) Let e be a stage-n λ_S expression with no free variables such that $e \xrightarrow{n} e'$. Let $R \vdash e \mapsto (\underline{e}, K)$ and $R \vdash e' \mapsto (\underline{e}', K')$. Then $K(\underline{e}) \xrightarrow{\mathcal{R};\mathcal{A}^*} K'(\underline{e}')$.*

변환은 실행의미를 보존

요구사항



증명한 것

"변환하고 실행 = 실행하고 변환"

변환을 이용한 분석설계하기

표현식 집합 $Exp_S\ Exp_R$
(Expressions)

$$e : Exp_S$$

$$\downarrow$$

$$\underline{e} : Exp_R$$

일단 변환했다

변환을 이용한 분석설계하기

표현식 집합 $Exp_S\ Exp_R$
(Expressions)

모듬 의미공간 $D_S\ D_R$
(Collecting Domains)

$e : Exp_S$ $\qquad$ $[\![e]\!] : D_S$

$\downarrow$

$\underline{e} : Exp_R$ $\qquad$ $[\![\underline{e}]\!] : D_R$

모듬의미* $[\![e]\!]$와 $[\![\underline{e}]\!]$를 정의한다

*모듬의미 = 구현생각하지 않고 만든 가장 정확한 분석

변환을 이용한 분석설계하기

표현식 집합 $Exp_S\ Exp_R$
(Expressions)

모듬 의미공간 $D_S\ D_R$
(Collecting Domains)

$$
\begin{array}{ccc}
e : Exp_S & & [\![e]\!] \\
\Big\downarrow & & \Big\uparrow \pi : D_R \to D_S \\
\underline{e} : Exp_R & & [\![\underline{e}]\!]
\end{array}
$$

좋은 투영 π**로** $[\![e]\!]$**와** $[\![\underline{e}]\!]$**를 연결한다**

$$[\![e]\!] \sqsubseteq \pi([\![\underline{e}]\!])$$

변환을 이용한 분석설계하기

표현식 집합 $Exp_S\ Exp_R$
(Expressions)

모듬 의미공간 $D_S\ D_R$
(Collecting Domains)

요약 의미공간 $\widehat{D}_R$
(Abstract Domains)

$$D_R \underset{\alpha_R}{\overset{\gamma_R}{\leftrightarrows}} \widehat{D}_R$$

$e : Exp_S$

$\underline{e} : Exp_R$

$$[\![e]\!]$$

$$\uparrow \pi$$

$$[\![\underline{e}]\!] \quad \sqsubseteq_{\alpha_R} \quad [\![\hat{\underline{e}}]\!] : \widehat{D}_R$$

좋은 요약의미 $[\![\hat{\underline{e}}]\!]$**를 만든다**

$$a \sqsubseteq_R b \Leftrightarrow \alpha_R(a) \sqsubseteq b$$

변환을 이용한 분석설계하기

표현식 집합 $Exp_S\ Exp_R$
(Expressions)

모듬 의미공간 $D_S\ D_R$
(Collecting Domains)

요약 의미공간 $\widehat{D}_R$
(Abstract Domains)

$$D_R \underset{\alpha_R}{\overset{\gamma_R}{\leftrightarrows}} \widehat{D}_R$$

$$\begin{array}{ccc}
e : Exp_S & [\![e]\!] & \sqsubseteq \quad \pi([\![\gamma_R([\![\hat{e}]\!])]\!) : D_S \\
\Big\downarrow & \Big\uparrow \pi & \Big\uparrow \pi \circ \gamma_R \\
\underline{e} : Exp_R & [\![\underline{e}]\!] & \sqsubseteq_{\alpha_R} \ [\![\hat{\underline{e}}]\!] : \widehat{D}_R
\end{array}$$

$[\![e]\!] \sqsubseteq \pi([\![\underline{e}]\!])$ **이므로**

변환을 이용한 분석설계하기

표현식 집합 $Exp_S\ Exp_R$
(Expressions)

모듬 의미공간 $D_S\ D_R$
(Collecting Domains)

요약 의미공간 $\widehat{D}_R$
(Abstract Domains)

$$D_R \underset{\alpha_R}{\overset{\gamma_R}{\leftrightarrows}} \widehat{D}_R$$

$$\begin{array}{lll} e : Exp_S & [\![e]\!] & \sqsubseteq \quad \pi([\gamma_R]([\![\hat{e}]\!])) : D_S \\ \Big\downarrow & \Big\uparrow \pi & \Big\uparrow \pi \circ \gamma_R \\ \underline{e} : Exp_R & [\![\underline{e}]\!] & \sqsubseteq_{\alpha_R} \ [\![\hat{\underline{e}}]\!] : \widehat{D}_R \end{array}$$

계산이 끝나지 않을수도....

변환을 이용한 분석설계하기

표현식 집합 $Exp_S\ Exp_R$
(Expressions)

모듬 의미공간 $D_S\ D_R$
(Collecting Domains)

요약 의미공간 $\widehat{D}_S\ \widehat{D}_R$
(Abstract Domains)

$D_S \underset{\alpha_S}{\overset{\gamma_S}{\leftrightarrows}} \widehat{D}_S \qquad D_R \underset{\alpha_R}{\overset{\gamma_R}{\leftrightarrows}} \widehat{D}_R$

$$
\begin{array}{ccc}
e : Exp_S & [\![e]\!] & \sqsubseteq \quad \pi([\![\gamma_R(\underline{\hat{e}})]\!]) : D_S \\
\Big\downarrow & \Big\uparrow \pi & \Big\uparrow \pi \circ \gamma_R \\
\underline{e} : Exp_R & [\![\underline{e}]\!] & \sqsubseteq_{\alpha_R} \ [\![\underline{\hat{e}}]\!] : \widehat{D}_R
\end{array}
$$

요약 의미공간을 추가해보면

변환을 이용한 분석설계하기

표현식 집합 $Exp_S\ Exp_R$
(Expressions)

모듬 의미공간 $D_S\ D_R$
(Collecting Domains)

요약 의미공간 $\widehat{D}_S\ \widehat{D}_R$
(Abstract Domains)

$D_S \underset{\alpha_S}{\overset{\gamma_S}{\leftrightarrows}} \widehat{D}_S \qquad D_R \underset{\alpha_R}{\overset{\gamma_R}{\leftrightarrows}} \widehat{D}_R$

$$
\begin{array}{ccc}
e : Exp_S & [\![e]\!] & \sqsubseteq_{\alpha_S} \alpha_S(\pi(\gamma_R([\![\hat{\underline{e}}]\!]))) : \widehat{D}_S \\
\Big\downarrow & \Big\uparrow \pi & \Big\uparrow \alpha_S \circ \pi \circ \gamma_R \\
\underline{e} : Exp_R & [\![\underline{e}]\!] & \sqsubseteq_{\alpha_R} [\![\hat{\underline{e}}]\!] : \widehat{D}_R
\end{array}
$$

요약 의미공간을 추가해보면

변환을 이용한 분석설계하기

표현식 집합 $Exp_S\ Exp_R$
(Expressions)

모듬 의미공간 $D_S\ D_R$
(Collecting Domains)

요약 의미공간 $\widehat{D}_S\ \widehat{D}_R$
(Abstract Domains)

$D_S \underset{\alpha_S}{\overset{\gamma_S}{\leftrightarrows}} \widehat{D}_S \qquad D_R \underset{\alpha_R}{\overset{\gamma_R}{\leftrightarrows}} \widehat{D}_R$

$e : Exp_S$

$\pi \uparrow \quad \downarrow$

$\underline{e} : Exp_R$

$[\![e]\!] \sqsubseteq_{\alpha_S} \alpha_S(\pi([\![\underline{e}]\!])) : \widehat{D}_S$

$\pi \uparrow \qquad \alpha_S \circ \pi \circ \gamma_R \uparrow$

$[\![\underline{e}]\!] \sqsubseteq_{\alpha_R} [\![\hat{\underline{e}}]\!] : \widehat{D}_R$

여전히 끝나지 않을수도....

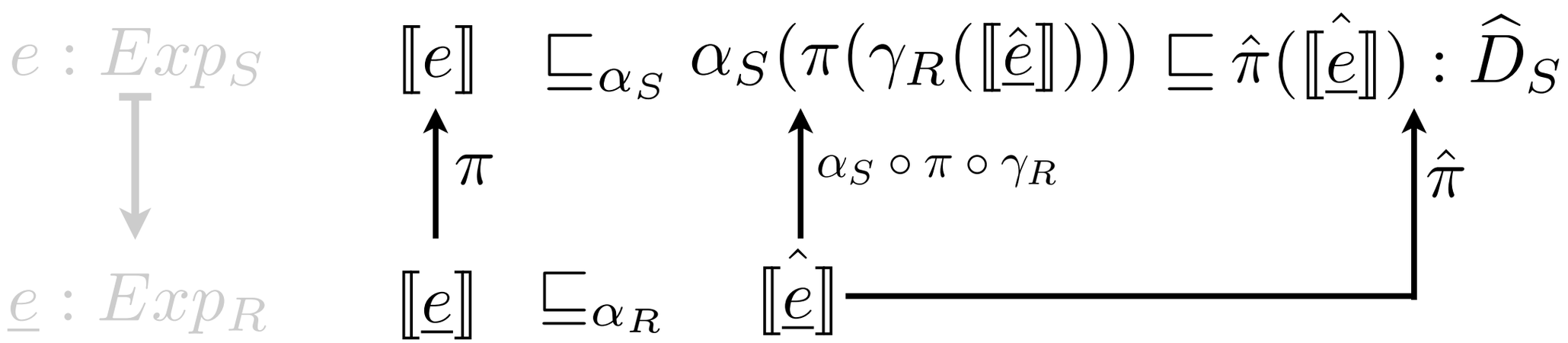
변환을 이용한 분석설계하기

표현식 집합 $Exp_S\ Exp_R$
(Expressions)

모듬 의미공간 $D_S\ D_R$
(Collecting Domains)

요약 의미공간 $\widehat{D}_S\ \widehat{D}_R$
(Abstract Domains)

$D_S \underset{\alpha_S}{\overset{\gamma_S}{\leftrightarrows}} \widehat{D}_S \qquad D_R \underset{\alpha_R}{\overset{\gamma_R}{\leftrightarrows}} \widehat{D}_R$



$\alpha_S \circ \pi \circ \gamma_R$ **대신 좋은** $\hat{\pi}$**를 사용**

$\alpha_S \circ \pi \circ \alpha_R \sqsubseteq \hat{\pi}$

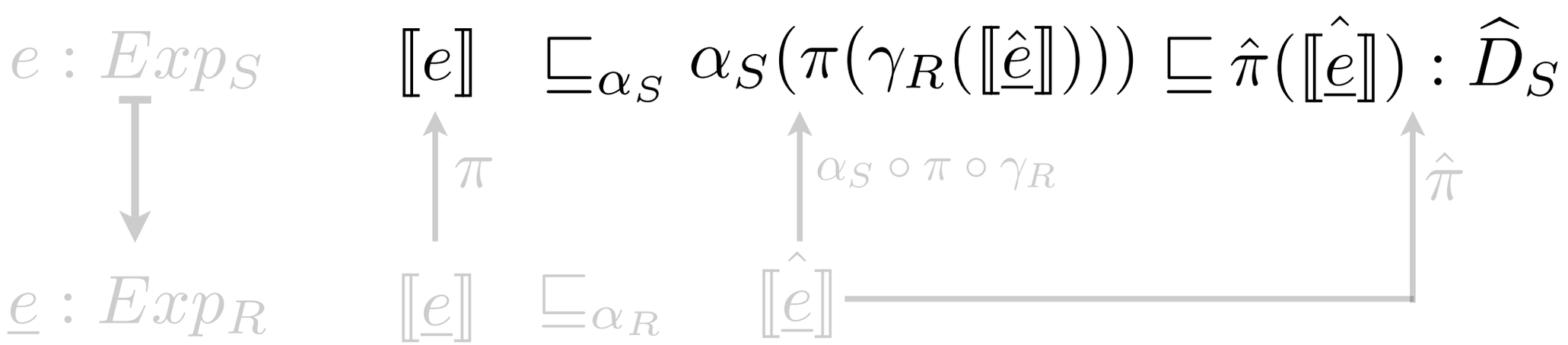
변환을 이용한 분석설계하기

표현식 집합 $Exp_S\ Exp_R$
(Expressions)

모듬 의미공간 $D_S\ D_R$
(Collecting Domains)

요약 의미공간 $\widehat{D}_S\ \widehat{D}_R$
(Abstract Domains)

$D_S \underset{\alpha_S}{\overset{\gamma_S}{\leftrightarrows}} \widehat{D}_S \qquad D_R \underset{\alpha_R}{\overset{\gamma_R}{\leftrightarrows}} \widehat{D}_R$



안전하고 끝나는 분석 완성!

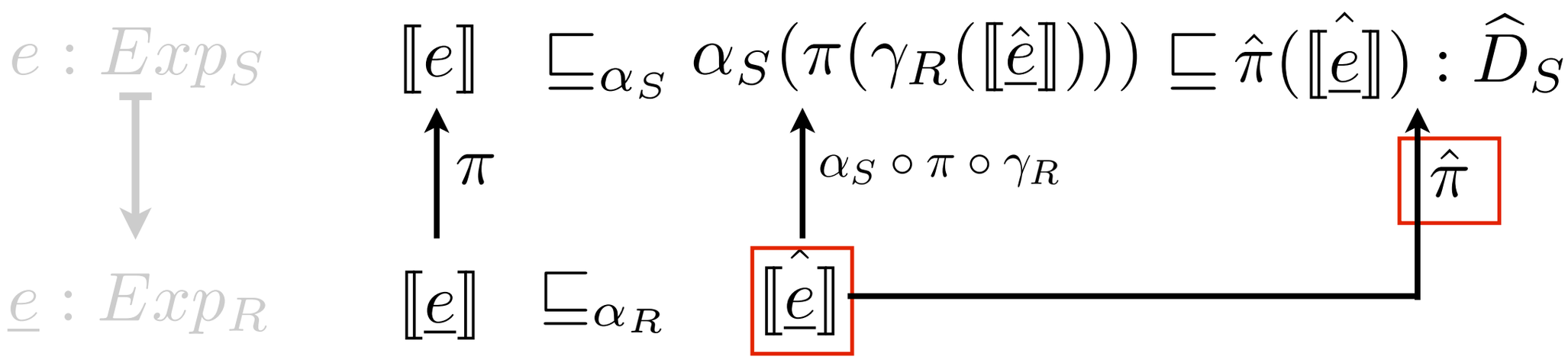
변환을 이용한 분석설계하기

표현식 집합 $Exp_S\ Exp_R$
(Expressions)

모듬 의미공간 $D_S\ D_R$
(Collecting Domains)

요약 의미공간 $\widehat{D}_S\ \widehat{D}_R$
(Abstract Domains)

$D_S \underset{\alpha_S}{\overset{\gamma_S}{\leftrightarrows}} \widehat{D}_S \qquad D_R \underset{\alpha_R}{\overset{\gamma_R}{\leftrightarrows}} \widehat{D}_R$



실제 사용할 것들

변환을 이용한 분석설계하기

표현식 집합 $Exp_S\ Exp_R$
(Expressions)

모듬 의미공간 $D_S\ D_R$
(Collecting Domains)

요약 의미공간 $\widehat{D}_S\ \widehat{D}_R$
(Abstract Domains)

$D_S \underset{\alpha_S}{\overset{\gamma_S}{\leftrightarrows}} \widehat{D}_S \qquad D_R \underset{\alpha_R}{\overset{\gamma_R}{\leftrightarrows}} \widehat{D}_R$

$e : Exp_S$

$\underline{e} : Exp_R$

$[\![e]\!] \sqsubseteq_{\alpha_S} \alpha_S(\pi([\![\underline{e}]\!])) \sqsubseteq \hat{\pi}([\![\hat{\underline{e}}]\!]) : \widehat{D}_S$

$[\![e]\!] \sqsubseteq_{\alpha_S} \alpha_S(\pi(\gamma_R([\![\hat{\underline{e}}]\!])))$

π

$\alpha_S \circ \pi \circ \gamma_R$

$\hat{\pi}$

$[\![\underline{e}]\!] \sqsubseteq_{\alpha_R} [\![\hat{\underline{e}}]\!]$

$[\![e]\!] \sqsubseteq \pi([\![\underline{e}]\!])$

안전성을 위해 필요한 장치들 #1

변환을 이용한 분석설계하기

표현식 집합 $Exp_S\ Exp_R$
(Expressions)

모듬 의미공간 $D_S\ D_R$
(Collecting Domains)

요약 의미공간 $\widehat{D}_S\ \widehat{D}_R$
(Abstract Domains)

$$D_S \underset{\alpha_S}{\overset{\gamma_S}{\leftrightarrows}} \widehat{D}_S \qquad D_R \underset{\alpha_R}{\overset{\gamma_R}{\leftrightarrows}} \widehat{D}_R$$

$$\begin{array}{ccccc}
e : Exp_S & & [\![e]\!] & \sqsubseteq_{\alpha_S} & \alpha_S(\pi(\gamma_R([\![\hat{\underline{e}}]\!]))) \sqsubseteq \hat{\pi}([\![\hat{\underline{e}}]\!]) : \widehat{D}_S \\
\downarrow & & \uparrow \pi & & \uparrow \alpha_S \circ \pi \circ \gamma_R \qquad \uparrow \hat{\pi} \\
\underline{e} : Exp_R & & [\![\underline{e}]\!] & \sqsubseteq_{\alpha_R} & [\![\hat{\underline{e}}]\!]
\end{array}$$

$$[\![e]\!] \sqsubseteq \pi([\![\underline{e}]\!])$$

$$\alpha_S \circ \pi \circ \alpha_R \sqsubseteq \hat{\pi}$$

안전성을 위해 필요한 장치들 #2

요약

- 단계를 제거하는 변환을 만들었다
- 변환은 실행의미를 보존한다
- 변환 결과물을 분석하여 원본 프로그램을 검증할 수 있다