A Semantic Approach to Multi-Language Systems

Jacob Matthews
University of Chicago
jacobm@cs.uchicago.edu

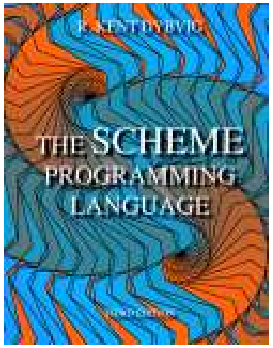
A Semantic Approach to Multi-Language Systems

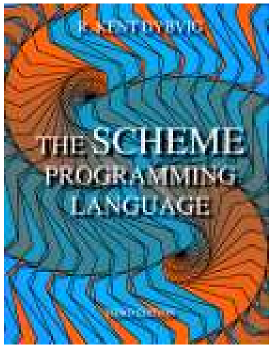
Jacob Matthews
University of Chicago
jacobm@cs.uchicago.edu

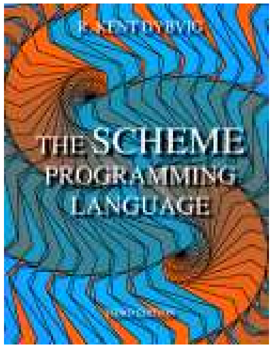

Seattle
Washington
Portland
Oregon
Montana
Idaho
Wyoming
North Dakota
South Dakota
Minnesota
Minneapolis
Wisconsin
Michigan
Milwaukee
Chicago
Detroit
Toronto
Ottawa
Québec
Montréal
New Brunswick
Maine
New York
Iowa
Omaha
Nebraska
Salt Lake City
Nevada
Utah
Denver
Colorado
Sacramento
San Francisco
California
Las Vegas
Kansas
Wichita
Kansas City
Missouri
St Louis
Illinois
Indiana
Ohio
Cleveland
Pennsylvania
Pittsburgh
New York
Cincinnati
West Virginia
Virginia
Kentucky
Tennessee
North Carolina
Los Angeles
San Diego
Arizona
Phoenix
Tucson
Albuquerque
New Mexico
El Paso
Oklahoma
Tulsa
Arkansas
Mississippi
Alabama
Atlanta
Georgia
South Carolina
Dallas
Texas
Austin
Houston
San Antonio
Louisiana
Jacksonville
Florida
Miami
Chihuahua
Monterrey
Gulf of Mexico
México
Connecticut
Rhode Island
Massachusetts
New Jersey
Delaware
Maryland
District of Columbia
A
B

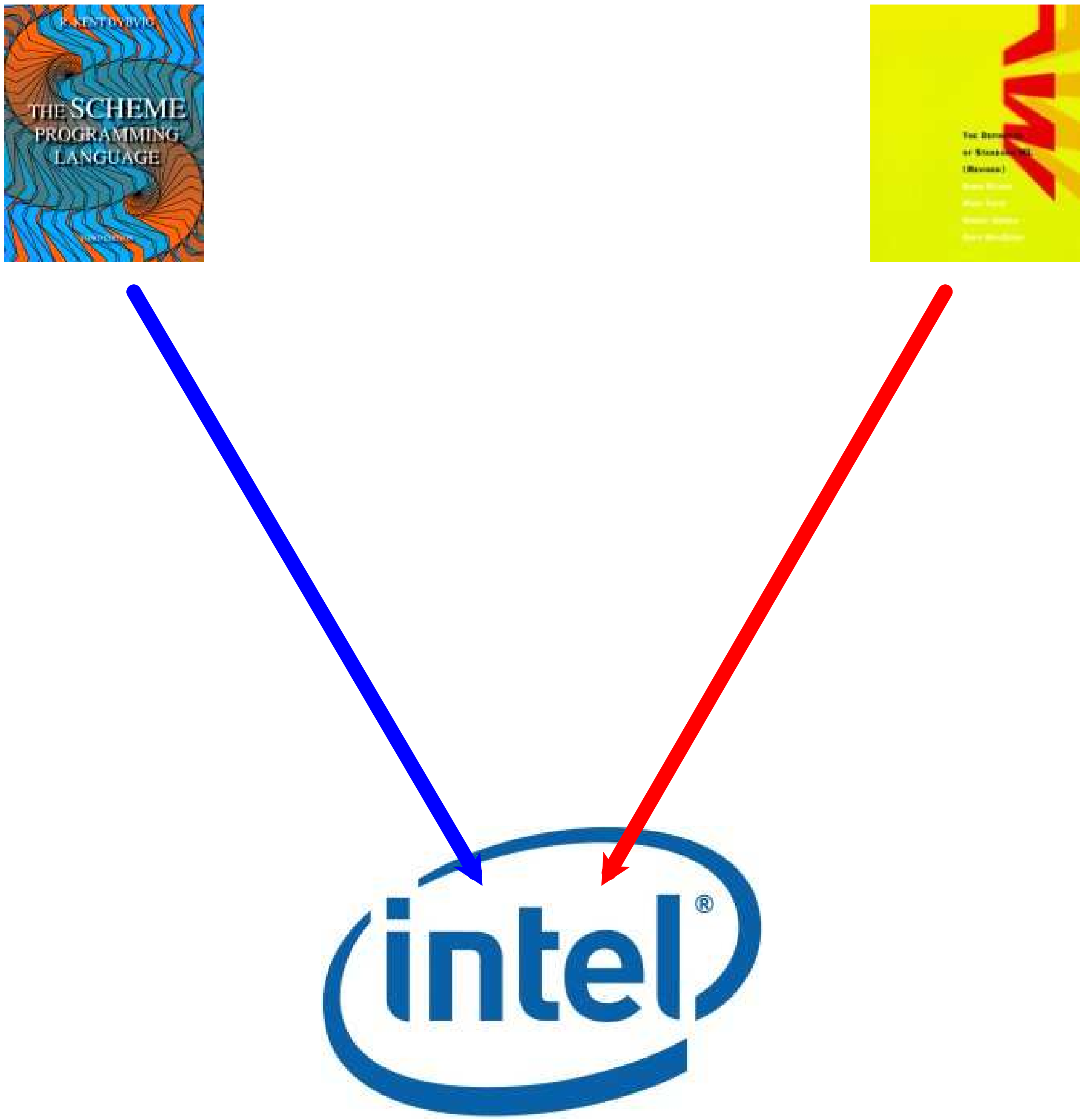
A Semantic Approach to Multi-Language Systems

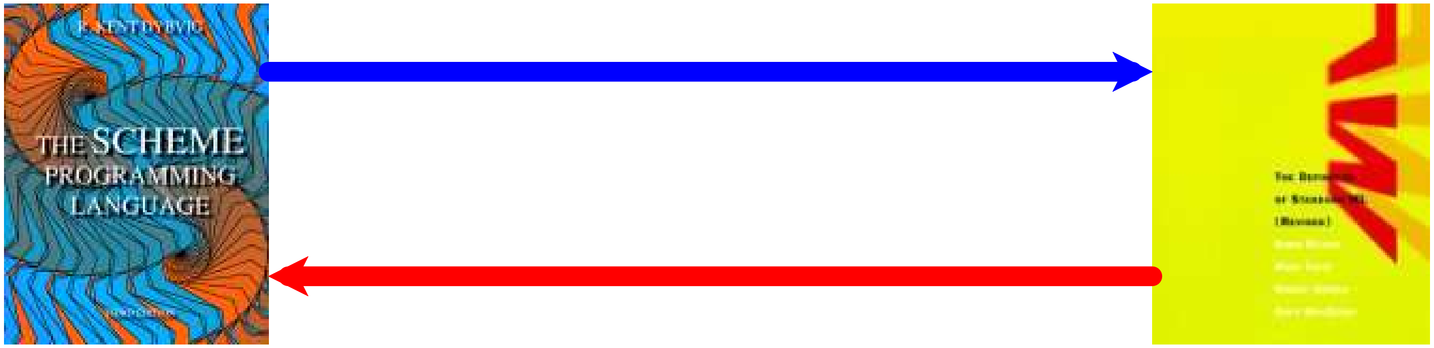
Jacob Matthews
University of Chicago - Irvine Campus
jacobm@cs.uchicago.edu


R. KENT DYBVIG
THE SCHEME
PROGRAMMING
LANGUAGE


R. KENT DYBVIG
THE SCHEME
PROGRAMMING
LANGUAGE


R. KENT DYBVIG
THE SCHEME PROGRAMMING LANGUAGE


R. KENT DYBVIG
THE SCHEME PROGRAMMING LANGUAGE
intel


R. KENT DYBVIG
THE SCHEME
PROGRAMMING
LANGUAGE

$$\mathbf{e} ::= \mathbf{v} \mid \mathtt{(e\ e)} \mid \mathtt{(+\ e\ e)}$$

$$\mathbf{v} ::= \mathtt{(}\lambda\ \mathtt{(x\ :\ }\tau\mathtt{)\ e)} \mid \mathtt{number}$$

$$\tau ::= \mathtt{int} \mid \mathtt{(}\tau \rightarrow \tau\mathtt{)}$$

$$\mathbf{E} ::= \mathtt{[]} \mid \mathtt{(v\ E)} \mid \mathtt{(E\ e)} \mid \mathtt{(+\ E\ e)} \mid \mathtt{(+\ v\ E)}$$

ML grammar

$E[((\lambda\ (x : \tau)\ e)\ v)]$ $E[e\ [x := v]]$

$E[(+\ n_1\ n_2)]$ $E[n_1+n_2]$

ML reductions

```
e := v | (e e) | (+ e e)
v := (λ (x) e) | number

E := [] | (v E) | (E e) | (+ E e) | (+ v E)
```

Scheme grammar

$\mathbf{E}[((\lambda\ (x)\ e)\ v)]$	$\mathbf{E}[e\ [x := v]]$
$\mathbf{E}[(+\ n_1\ n_2)]$	$\mathbf{E}[n_1 + n_2]$
$\mathbf{E}[(n\ v)]$	**Error:** can't apply numbers
$\mathbf{E}[(+\ (\lambda\ (x)\ e)\ n)]$	**Error:** can't add functions
$\mathbf{E}[(+\ n\ (\lambda\ (x)\ e))]$	**Error:** can't add functions

Scheme reductions

The key idea: boundaries

$$(\mathtt{MS}\ \tau\ \mathtt{TST}\ \mathtt{e})$$

An ML boundary

The key idea: boundaries

$$(\boxed{\mathbf{MS}}\ \tau\ \mathbf{TST}\ \ \mathbf{e})$$

"ML outside, Scheme inside"

The key idea: boundaries

(MS τ **TST e)**

The Scheme expression to run

The key idea: boundaries

$$\mathbf{(MS\ \boxed{\tau}\ TST\ e)}$$

The ML side's type

The key idea: boundaries

(MS τ TST e)

The Scheme side's type

The key idea: boundaries

(MS τ **e)**

... which isn't necessary to write down

The key idea: boundaries

(SM TST τ e)

A Scheme boundary

The key idea: boundaries

(SM TST τ e)

The Scheme side's type

The key idea: boundaries

$$(\mathbf{SM} \qquad\qquad \tau \quad \mathbf{e})$$

... which isn't necessary to write down

The key idea: boundaries

$$(\mathbf{SM} \qquad \boxed{\tau} \quad \mathbf{e})$$

The ML side's type

$$
\begin{aligned}
\mathbf{e} &:= \mathbf{v} \mid (\mathbf{e}\ \mathbf{e}) \mid (+\ \mathbf{e}\ \mathbf{e}) \mid (\mathbf{MS}\ \tau\ \mathbf{e}) \\
\mathbf{v} &:= (\lambda\ (\mathbf{x} : \tau)\ \mathbf{e}) \mid \mathbf{number} \\
\tau &:= \mathbf{int} \mid (\tau \to \tau)
\end{aligned}
$$

New ML grammar

```
e := v | (e e) | (+ e e) | (SM τ e)
v := (λ (x) e) | number
```

New Scheme grammar

$$E ::= [\,] \mid (v\ E) \mid (E\ e) \mid (MS\ \tau\ E)$$

$E[((\lambda\ (x : \tau)\ e)\ v)]$ $\quad E[e\ [x := v]]$

$E[(+\ n_1\ n_2)]$ $\quad E[n_1+n_2]$

New ML reductions

E[(MS int n)]   **E[n]**

E[(MS $\tau_1 \rightarrow \tau_2$ **(**λ **(x) e))]**

E[(λ **(x :** τ_1**) (MS** τ_2 **((**λ **(x) e) (SM** τ_1 **x))))]**

E[(MS int (λ **(x) e))]**   **Error:** Non-number

E[(MS $\tau_1 \rightarrow \tau_2$ **n)]**   **Error:** Non-function

```
((MS (int → int) (λ (x) (+ x 1))) 2)

((λ (y : int)
   (MS int ((λ (x) (+ x 1)) (SM int y))))
 2)

(MS int ((λ (x) (+ x 1)) (SM int 2)))

(MS int ((λ (x) (+ x 1)) 2))

(MS int (+ 2 1))

(MS int 3)

3
```

```
((MS (int → int) (λ (x) (+ x 1))) 2)

((λ (y : int)
   (MS int ((λ (x) (+ x 1)) (SM int y))))
 2)

(MS int ((λ (x) (+ x 1)) (SM int 2)))

(MS int ((λ (x) (+ x 1)) 2))

(MS int (+ 2 1))

(MS int 3)

3
```

```
((MS (int → int) (λ (x) (+ x 1))) 2)

((λ (y : int)
   (MS int ((λ (x) (+ x 1)) (SM int y))))
 2)

(MS int ((λ (x) (+ x 1)) (SM int 2)))

(MS int ((λ (x) (+ x 1)) 2))

(MS int (+ 2 1))

(MS int 3)

3
```

```
((MS (int → int) (λ (x) (+ x 1))) 2)

((λ (y : int)
    (MS int ((λ (x) (+ x 1)) (SM int y))))
 2)

(MS int ((λ (x) (+ x 1)) (SM int 2)))

(MS int ((λ (x) (+ x 1)) 2))

(MS int (+ 2 1))

(MS int 3)

3
```

```
((MS (int → int) (λ (x) (+ x 1))) 2)

((λ (y : int)
   (MS int ((λ (x) (+ x 1)) (SM int y))))
 2)

(MS int ((λ (x) (+ x 1)) (SM int 2)))

(MS int ((λ (x) (+ x 1)) 2))

(MS int (+ 2 1))

(MS int 3)

3
```

```
((MS (int → int) (λ (x) (+ x 1))) 2)

((λ (y : int)
   (MS int ((λ (x) (+ x 1)) (SM int y))))
 2)

(MS int ((λ (x) (+ x 1)) (SM int 2)))

(MS int ((λ (x) (+ x 1)) 2))

(MS int (+ 2 1))

(MS int 3)

3
```

```
((MS (int → int) (λ (x) (+ x 1))) 2)

((λ (y : int)
   (MS int ((λ (x) (+ x 1)) (SM int y))))
 2)

(MS int ((λ (x) (+ x 1)) (SM int 2)))

(MS int ((λ (x) (+ x 1)) 2))

(MS int (+ 2 1))

(MS int 3)

3
```

```
((MS (int → int) (λ (x) (λ (y) y))) 2)

((λ (y : int)
   (MS int ((λ (x) (λ (y) y)) (SM int y))))
 2)

(MS int ((λ (x) (λ (y) y)) (SM int 2)))

(MS int ((λ (x) (λ (y) y)) 2))

(MS int (λ (y) y))
```

Error: Non-number

```
((MS (int → int) (λ (x) (λ (y) y))) 2)

((λ (y : int)
    (MS int ((λ (x) (λ (y) y)) (SM int y))))
 2)

(MS int ((λ (x) (λ (y) y)) (SM int 2)))

(MS int ((λ (x) (λ (y) y)) 2))

(MS int (λ (y) y))
```

Error: Non-number

```
((MS (int → int) (λ (x) (λ (y) y))) 2)

((λ (y : int)
   (MS int ((λ (x) (λ (y) y)) (SM int y))))
 2)

(MS int ((λ (x) (λ (y) y)) (SM int 2)))

(MS int ((λ (x) (λ (y) y)) 2))

(MS int (λ (y) y))
```

Error: Non-number

```
((MS (int → int) (λ (x) (λ (y) y))) 2)

((λ (y : int)
   (MS int ((λ (x) (λ (y) y)) (SM int y))))
 2)

(MS int ((λ (x) (λ (y) y)) (SM int 2)))

(MS int ((λ (x) (λ (y) y)) 2))

(MS int (λ (y) y))
```

Error: Non-number

```
((MS (int → int) (λ (x) (λ (y) y))) 2)

((λ (y : int)
   (MS int ((λ (x) (λ (y) y)) (SM int y))))
 2)

(MS int ((λ (x) (λ (y) y)) (SM int 2)))

(MS int ((λ (x) (λ (y) y)) 2))

(MS int (λ (y) y))
```

Error: Non-number

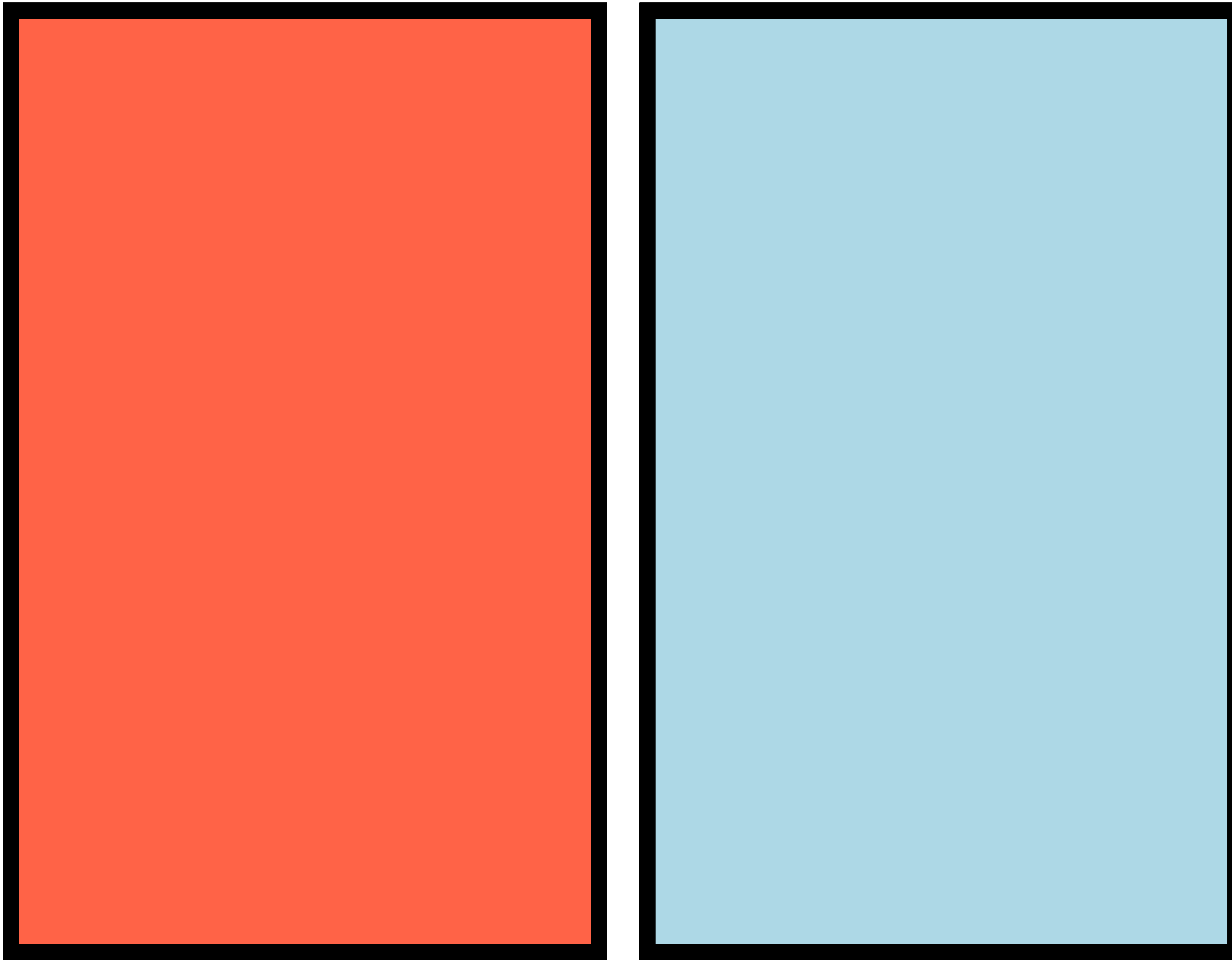
```
((MS (int → int) (λ (x) (λ (y) y))) 2)

((λ (y : int)
   (MS int ((λ (x) (λ (y) y)) (SM int y))))
 2)

(MS int ((λ (x) (λ (y) y)) (SM int 2)))

(MS int ((λ (x) (λ (y) y)) 2))

(MS int (λ (y) y))
```

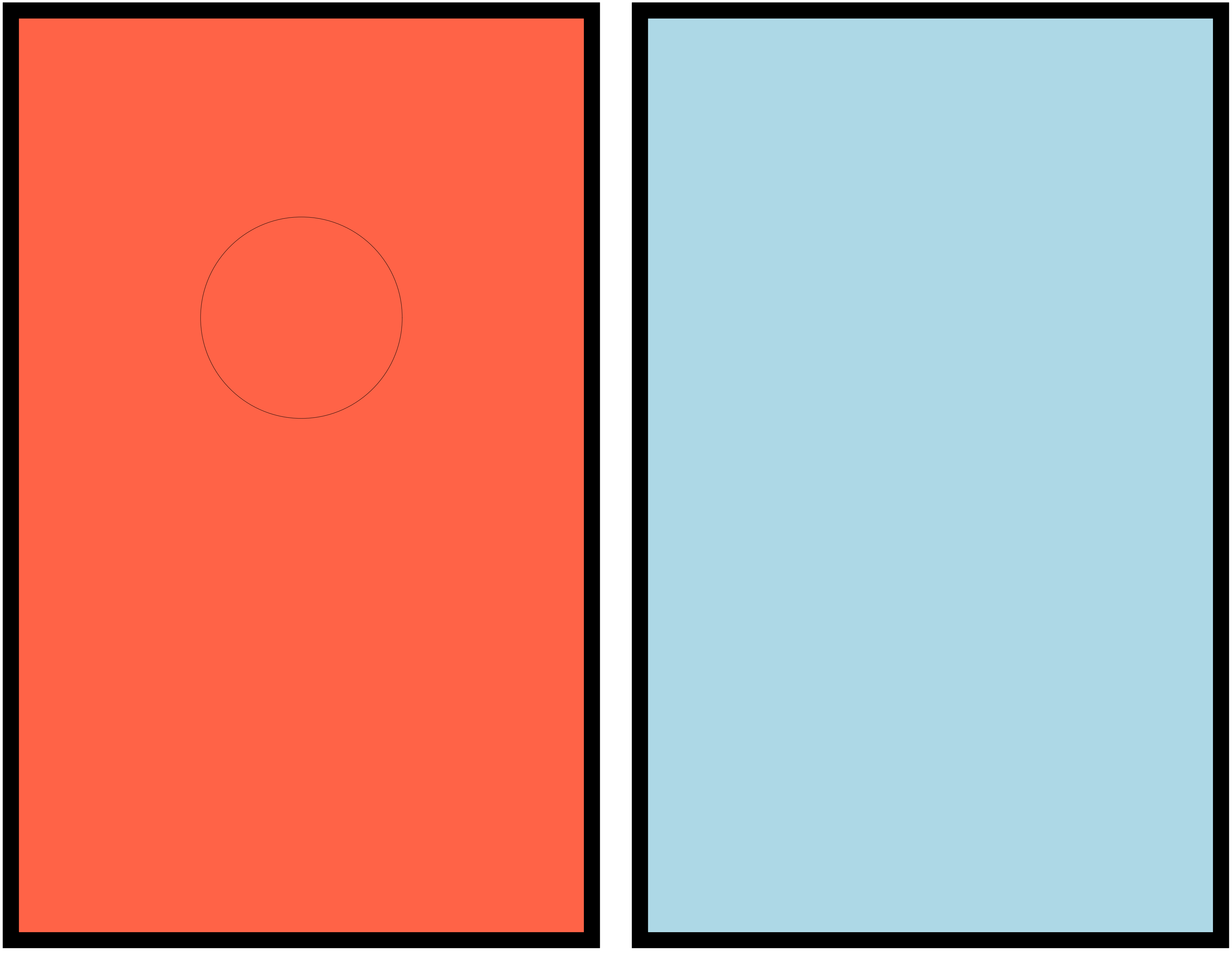
Error: Non-number

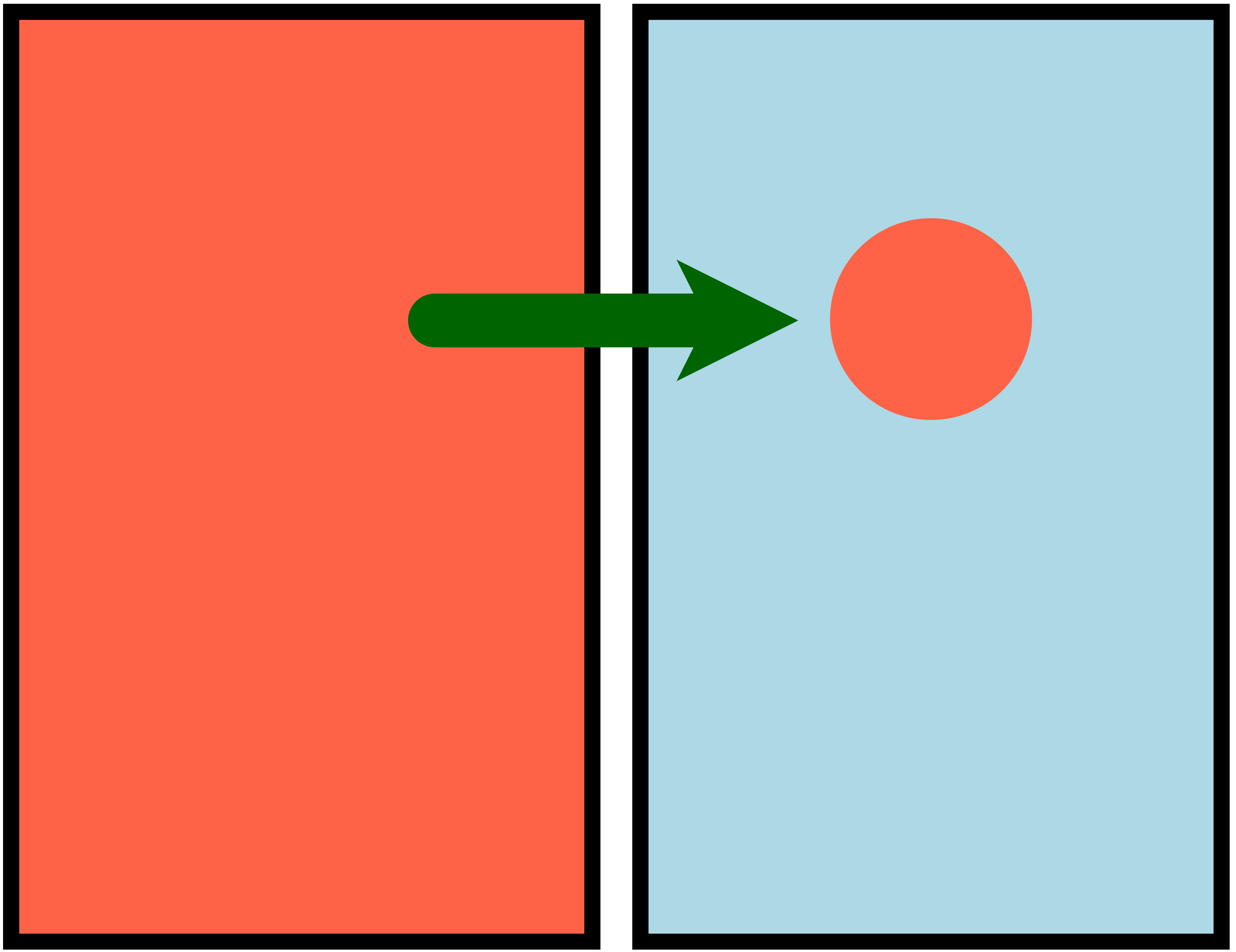
Theorem (type soundness)**:**

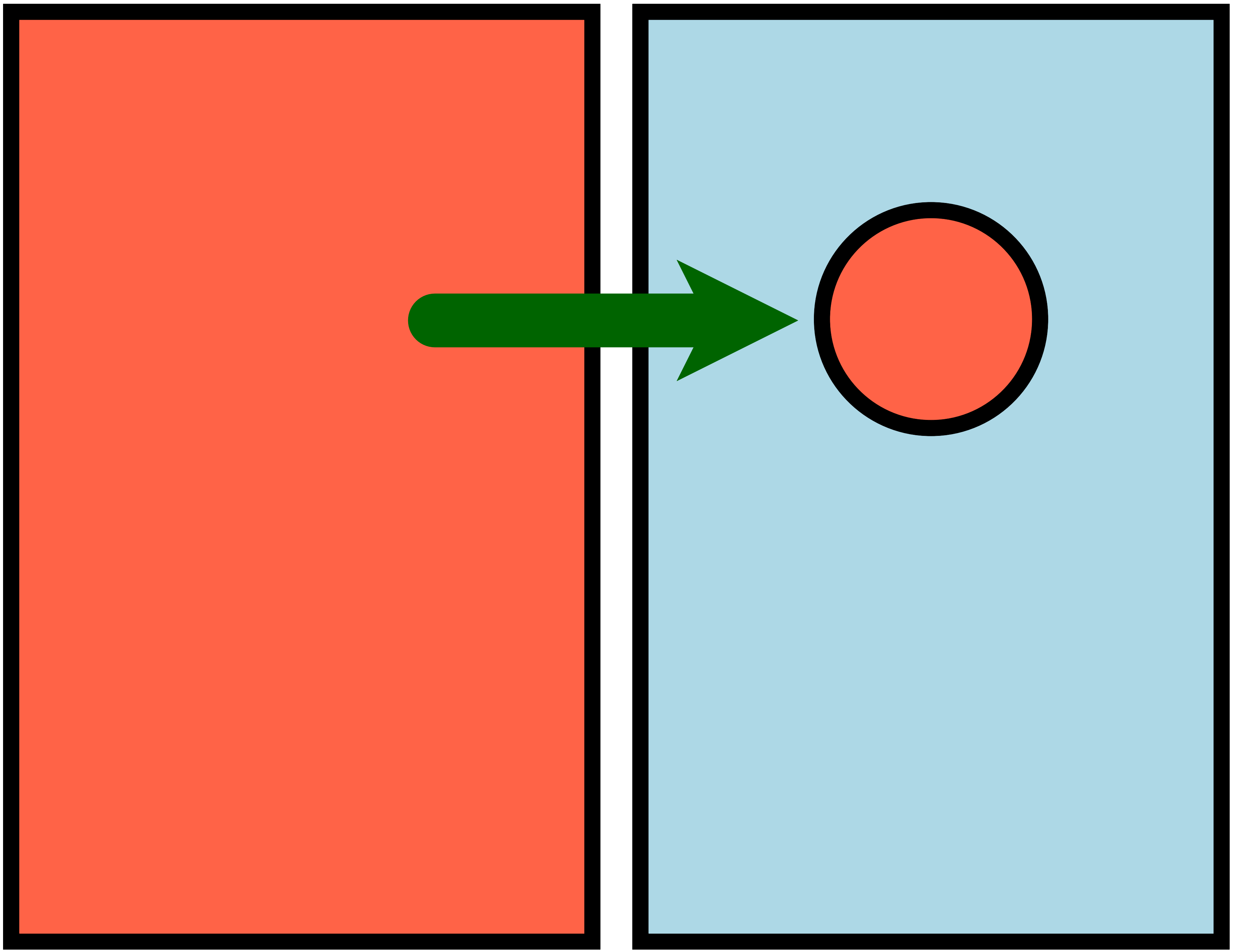
If **e** : τ, then either: **e** * **v**;
or **e** * **Error:** msg;
or **e** diverges.

Proof: Mutually-recursive variant of standard preservation and progress.









Information hiding = parametric polymorphism
[Reynolds]

versus

Information hiding = generativity + scope
[Morris]

Information hiding = parametric polymorphism

$$\forall\alpha.\alpha \to \alpha$$

```
;; → key
(define (create-key) (gensym))

;; TST key → key → TST
(define (lock v key1)
  (λ (key2)
    (if (eq? key1 key2)
        v
        (error))))
```

Information hiding = generativity + scope

Question:
But do these two strategies produce the same result?

Question:
But do these two strategies produce the same result?

Actually, that's a bad question.

Better question:

Can we use dynamic seals to combine polymorphic ML and Scheme without breaking parametricity?

Add generic types to ML:

$$
\begin{aligned}
\mathtt{e} &:= \mathtt{v} \mid \mathtt{(e\ e)} \mid \mathtt{(+\ e\ e)} \mid \mathtt{(MS}\ \tau\ \mathtt{e)} \mid \mathtt{e}\langle\tau\rangle \\
\mathtt{v} &:= \mathtt{(}\lambda\ \mathtt{(x\ :}\ \tau\mathtt{)\ e)} \mid \mathtt{number} \mid \mathtt{(}\Lambda\alpha\mathtt{.\ e)} \\
\tau &:= \mathtt{int} \mid \mathtt{(}\tau \to \tau\mathtt{)} \mid \alpha \mid \forall\alpha.\ \tau
\end{aligned}
$$

What about the new boundaries?

$$(\mathtt{SM}\ \alpha\ \mathtt{e})$$

$$(\mathtt{MS}\ \alpha\ \mathtt{e})$$

$$(\mathtt{SM}\ \forall\alpha.\ \tau\ \mathtt{e})$$

$$(\mathtt{MS}\ \forall\alpha.\ \tau\ \mathtt{e})$$

What about the new boundaries?

(SM α e)

(MS α e)

$\boxed{\begin{array}{l}\texttt{(SM } \forall\ .\tau\ \texttt{e)} \\ \texttt{MS } \forall\ .\tau\ \texttt{e}\end{array}}$

What about the new boundaries?

(SM α e)

(MS α e)

What about the new boundaries?

(SM α v)

(MS α v)

What about the new boundaries?

(SM α v)

(MS α v)

Typed value entering untyped code: Seal!

What about the new boundaries?

```
(SM α v)
(MS α v)
```

Untyped value claimed to satisfy type α: Unseal!

Theorem (parametricity)**:**

- The ML side of the embedding is parametric
- The Scheme side of the embedding is oblivious to the contents of sealed values

Proof: Simultaneously show both claims using the method of logical relations. (We need step-indexed logical relations to handle Scheme.)

Consequences:

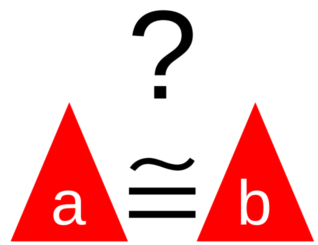
- Higher-order polymorphic contracts with dynamic sealing enforce parametricity
- At least some of System F's good properties are preserved by a well-designed embedding

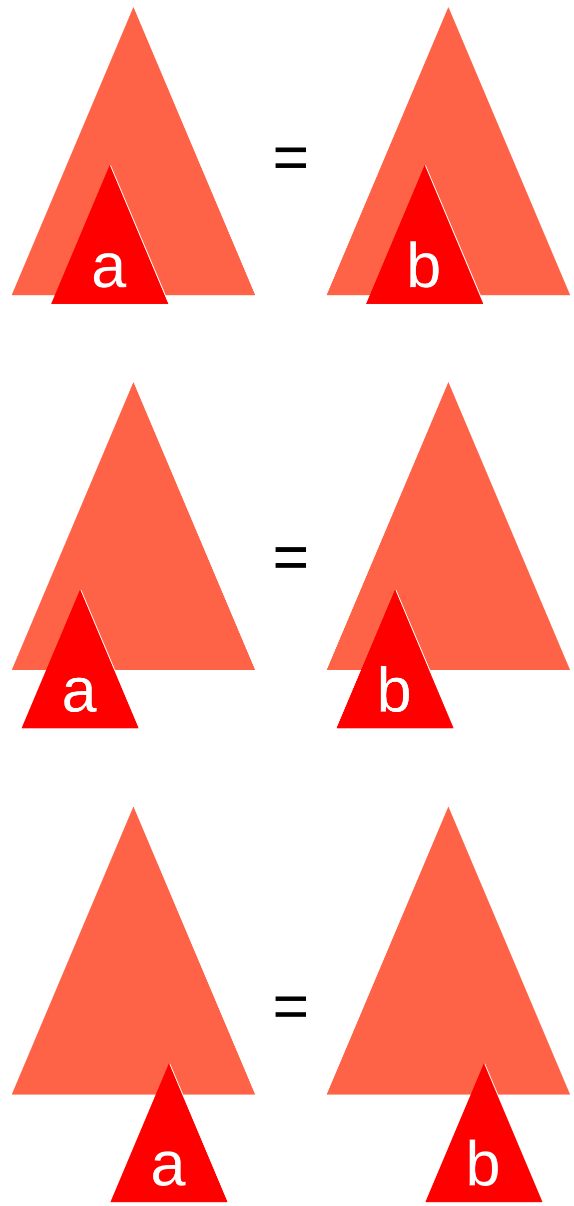
Consequences:

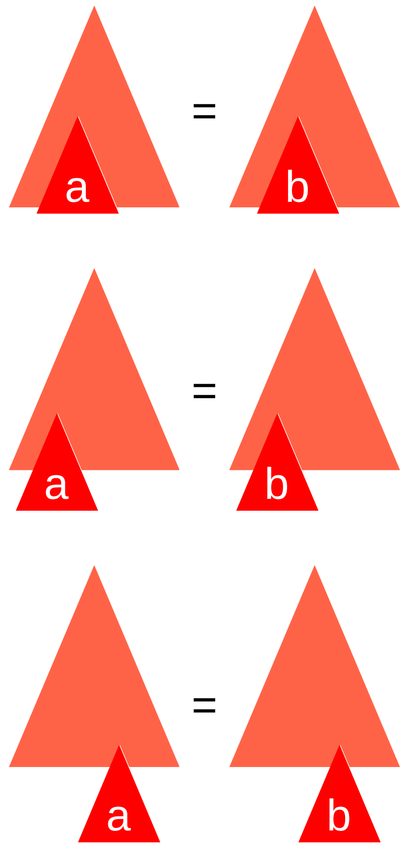
- Higher-order polymorphic contracts with dynamic sealing enforce parametricity
- At least some of System F’s good properties are preserved by a well-designed embedding

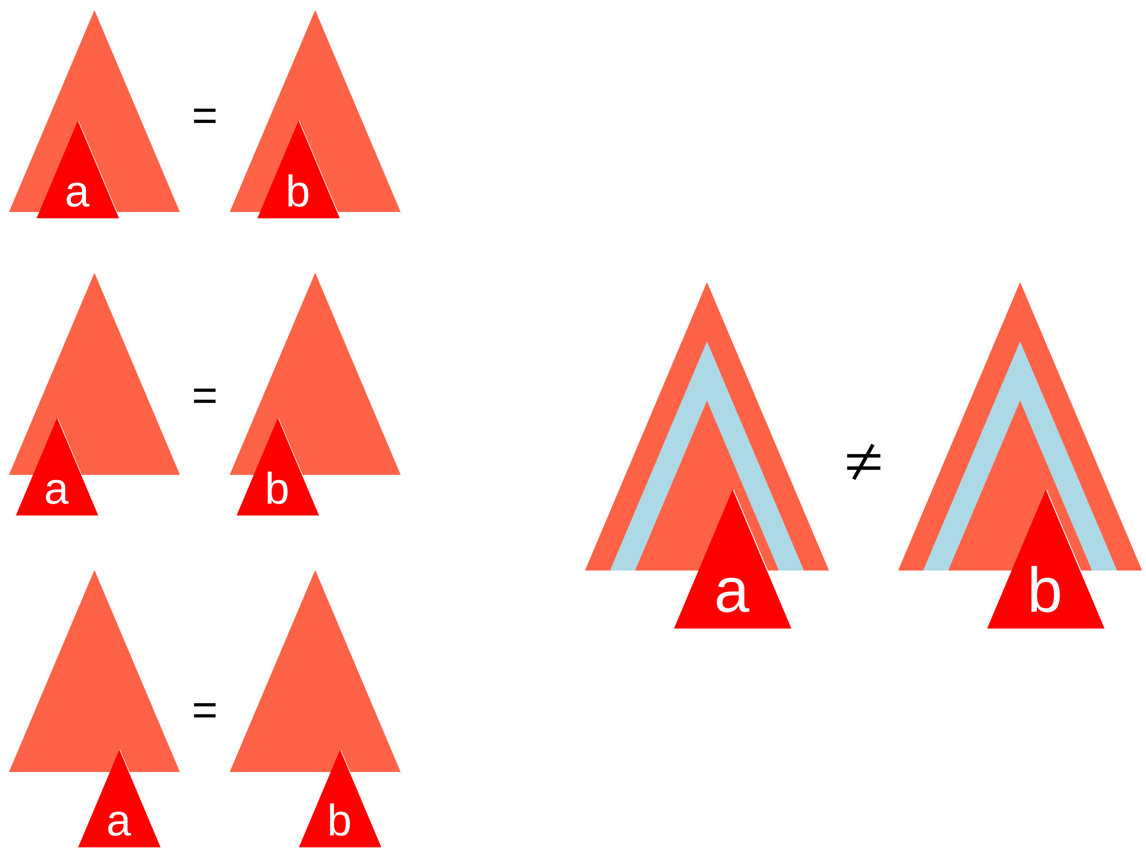
... But what about all of them?

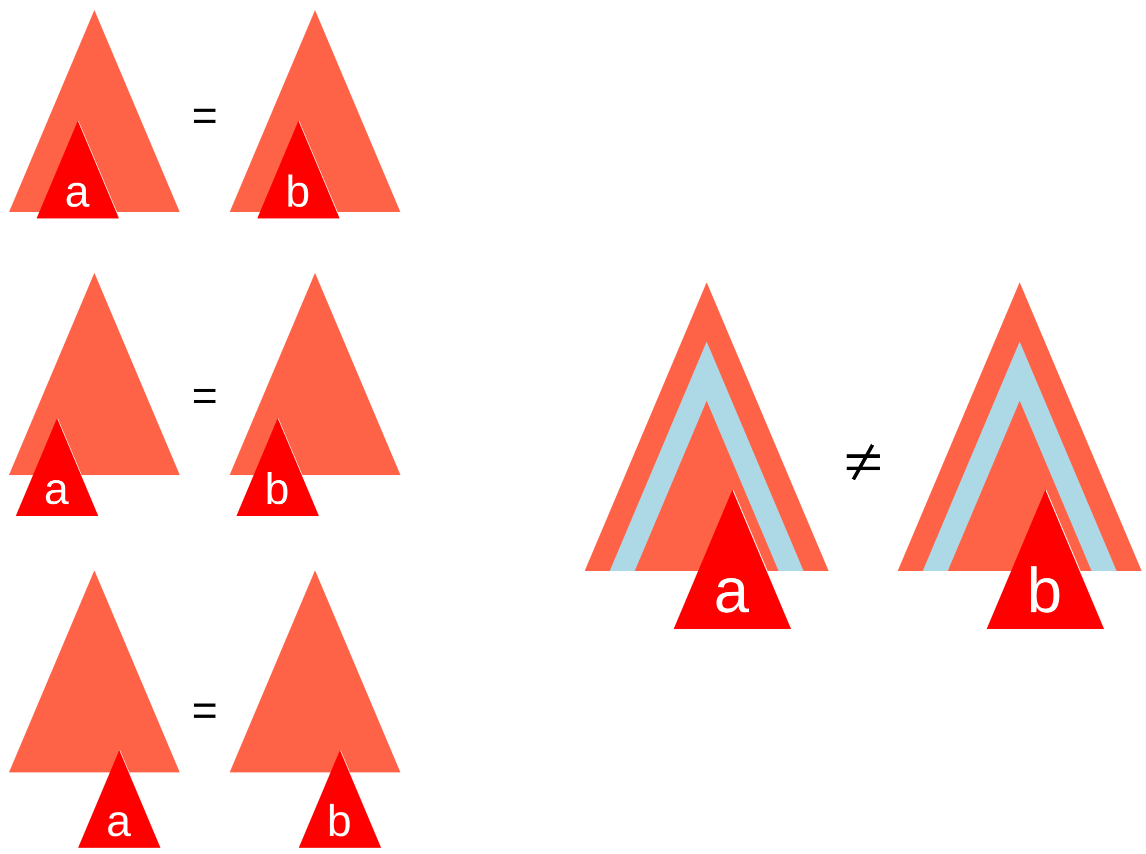
A design criterion for multi-language interfaces


?
a ≅ b


a
=
b
a
=
b
a
=
b


a
=
b
a
=
b
a
=
b


a
=
b
a
=
b
a
=
b
a
≠
b



This is strange ...

Proposed multi-language design criterion:

For any two expressions a and b,

$$a \cong_P b$$

implies

$$a \cong_{P+Q} b$$

Thanks!