Parametric Polymorphism through run-time sealing

(Theorems for low, low prices!)

Ben Greenman

September 22, 2014

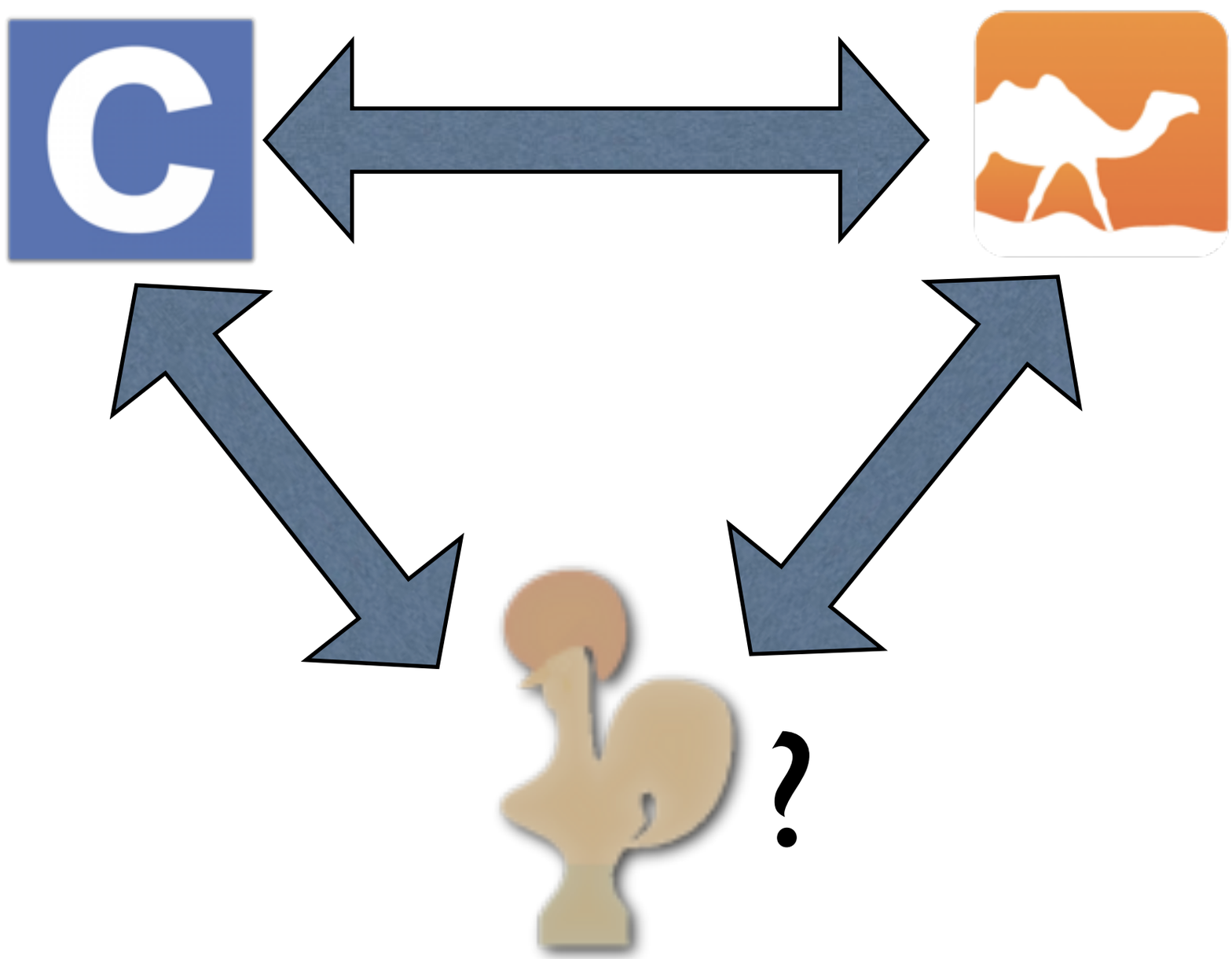
Is there a "Best" Programming Language?

Of course not!

We live in a multi-language world

Interoperability?

Interoperability?



Jacob Matthews

Amal Ahmed

Jacob Matthews

Amal Ahmed

Scheme

ML

Question

- If we mix Scheme code and ML code, what static guarantees still apply?
- Can we prove parametricity after mixing typed and untyped code?

Parametricity

- Parametricity is about invariances.
- A function is parametric if it has a "uniformly given algorithm on all types".

Parametricity

$$fst = \lambda(x,y) \; . \, x$$

Parametricity

$$\text{fst} = \lambda(x,y)\ .\ x$$

$$\forall\alpha\ .\ \alpha * \alpha \rightarrow \alpha$$

Parametricity

$$\text{fst} = \lambda(x,y)\ .\ x$$

$$\text{snd} = \lambda(x,y)\ .\ y$$

$$\forall\alpha\ .\ \alpha * \alpha \rightarrow \alpha$$

Parametricity

(1) Strachey [1967]: parametric vs. ad-hoc

(2) Reynolds [1983]: abstraction theorem

(3) Bainbridge, Freyd et. al [1988]: "parametricity"

(4) Wadler [1989]: free theorems

(1) "Fundamental Concepts in Programming Languages"
(2) "Types, Abstraction, and Parametric Polymorphism"
(3) "Functorial Polymorphism" & "Semantic Parametricity in Polymorphic Lambda Calculus"
(4) "Theorems for Free!"

Free Theorems

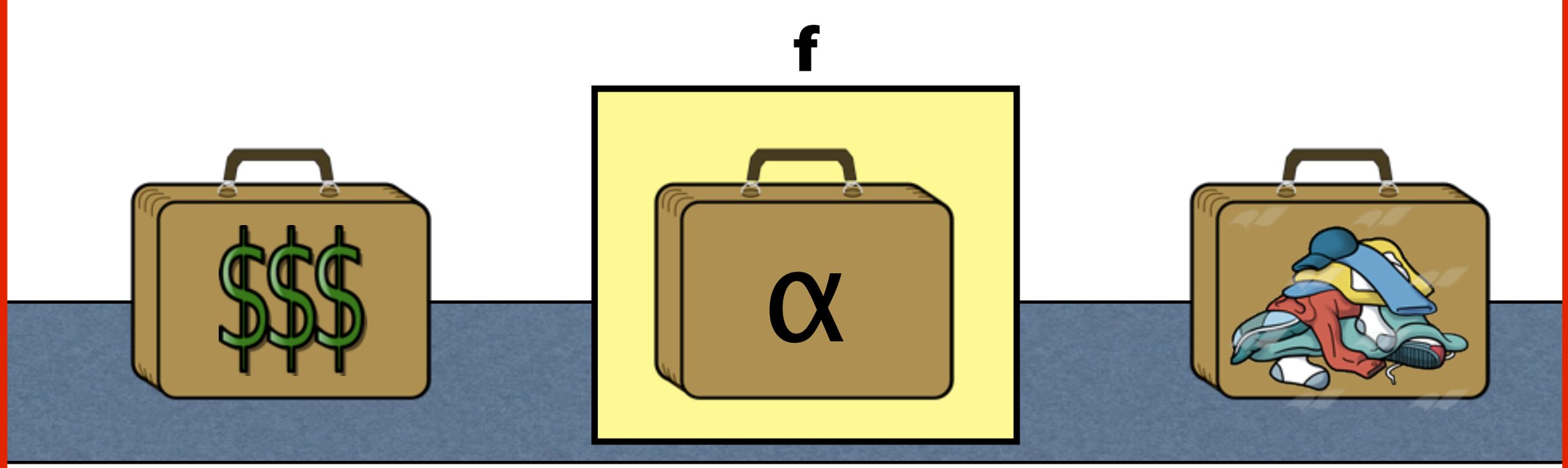
For all functions **f** : $\forall\alpha\ .\ \alpha\ \textit{list} \rightarrow \alpha\ \textit{list}$

And <u>any</u> function **g** : $\beta \rightarrow \gamma$

(map g) ∘ f = f ∘ (map g)

Free Theorems

- Why does this work?
- All values of type α are black boxes to **f**.



Free Theorems

- In System F, Haskell, ML, etc., parametricity is guaranteed statically.
- The type of a polymorphic function expresses the invariants it preserves.

What about Scheme?

Theorems for Scheme?

```
;; fst : A * A -> A
(define (fst (a,b))
    a)
```

```
;; snd : A * A -> A
(define (snd (a,b))
    b)
```

Theorems for Scheme?

```
;; fst2 : A * A -> A
(define (fst2 (a,b))
     (if (and (int? a)
              (= a 17))
 b
 a))
```

"Almost always" well-behaved is NOT good enough!

Theorems for Scheme?

```
;; min : A * A -> A
(define (min (a,b))
    (cond [(and (int? a)(int? b))
           (if (< a b) a b)]
          [(and (str? a)(str? b))
           (if (<-str a b) a b)]
          ...
          [else (error)])))
```

Theorems for Scheme?

```
;; min : A          -> A
(define (min      b))
    (cond [(an    (i      a)(int? b))
           (if        b) a b)]
          [(and      ? a)(str? b))
           (if  <-      a b) a b)]
          ..
          [    e (erro      ))
```

What happened?

```
;; fst2 : A * A -> A
(define (fst2 (a,b))
     (if (and (int? a)
              (= a 42))
         b
         a))
```

Programs should NOT be able to inspect a value with an abstract type.

Dynamic Seals

- Morris [1973]: "Types are not Sets".
- Protect exported values with secret keys.
- Crash if a sealed value is used.

```
(define (seal v s1)
     (λs2 . (if (eq? s1 s2)
                v
                (error))))
```

Can we combine static type checking and dynamic seals?

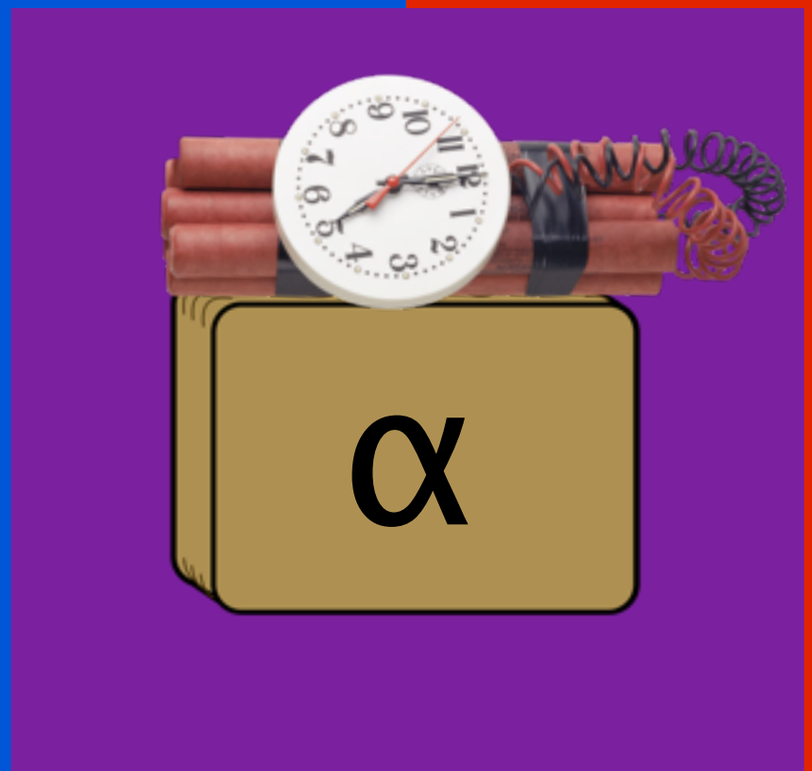
Dynamic Seals

Polymorphism

YES!

Dynamic Seals

Polymorphism



Paper Outline

I. Define the languages + conversion rules

II. Prove type safety + parametricity

III. Demonstrate applications

The Languages

v = (λx. e) | n | nil | (cons v v) | fst | rst	v = λx:τ. e | n | nil | cons v v | fst | rst
e = v | (e e) | x | (*op* e e) | (if0 e e e) | (cons e e) | (*pd* e)	e = v | (e e) | x | *op* e e | if0 e e e | cons e e
	τ = Nat | $\tau \rightarrow \tau$ | τ^*

The Languages

$v = (\lambda x.\ e) \mid n \mid nil$	$v = \lambda x{:}\tau.\ e \mid n \mid nil$
$\mid (cons\ v\ v) \mid fst \mid rst$	$\mid cons\ v\ v \mid fst \mid rst$
$e = v \mid (e\ e) \mid x \mid (op\ e\ e)$	$e = v \mid (e\ e) \mid x \mid op\ e\ e$
$\mid (if0\ e\ e\ e) \mid (cons\ e\ e)$	$\mid if0\ e\ e\ e \mid cons\ e\ e$
$\mid (pd\ e) \mid (SM^{\tau} e)$	$\mid {}^{\tau}MS\ e$
	$\tau = Nat \mid \tau \rightarrow \tau \mid \tau^*$

The Languages

(SM$^{\tau}$ e)	($^{\tau}$MS e)
Convert the ML expression e with type τ into a Scheme expression	Convert the Scheme expression e to an ML expression at type τ

The Languages

$v = (\lambda x.\ e) \mid n \mid nil$	$v = \lambda x{:}\tau.\ e \mid n \mid nil$
$\mid (cons\ v\ v) \mid fst \mid rst$	$\mid cons\ v\ v \mid fst \mid rst$
$e = v \mid (e\ e) \mid x \mid (op\ e\ e)$	$e = v \mid (e\ e) \mid x \mid op\ e\ e$
$\mid (if0\ e\ e\ e) \mid (cons\ e\ e)$	$\mid if0\ e\ e\ e \mid cons\ e\ e$
$\mid (pd\ e) \mid (SM^{\tau} e)$	$\mid {}^{\tau}MS\ e$
	$\tau = Nat \mid \tau \rightarrow \tau \mid \tau^*$

Conversion Example

$\tau \to \tau'$ MS (λx. e)

λx:τ. MS$^{\tau'}$((λx. e) SM$^{\tau}$ x

The Languages

$v = (\lambda x.\ e) \mid n \mid nil$	$v = \lambda x{:}\tau.\ e \mid n \mid nil$
$\mid (cons\ v\ v) \mid fst \mid rst$	$\mid cons\ v\ v \mid fst \mid rst$
$e = v \mid (e\ e) \mid x \mid (op\ e\ e)$	$e = v \mid (e\ e) \mid x \mid op\ e\ e$
$\mid (if0\ e\ e\ e) \mid (cons\ e\ e)$	$\mid if0\ e\ e\ e \mid cons\ e\ e$
$\mid (pd\ e) \mid (SM^{\tau} e)$	$\mid {}^{\tau}MS\ e$
	$\tau = Nat \mid \tau \rightarrow \tau \mid \tau^*$

The Languages

$v = (\lambda x.\ e) \mid n \mid nil$
$\quad \mid (cons\ v\ v) \mid fst \mid rst$

$e = v \mid (e\ e) \mid x \mid (op\ e\ e)$
$\quad \mid (if0\ e\ e\ e) \mid (cons\ e\ e)$
$\quad \mid (pd\ e) \mid (SM^{\tau} e)$

$v = \lambda x{:}\tau.\ e \mid n \mid nil$
$\quad \mid cons\ v\ v \mid fst \mid rst$
$\quad \mid {}^{L}MS\ v \mid \Lambda\tau.\ e$

$e = v \mid (e\ e) \mid x \mid op\ e\ e$
$\quad \mid if0\ e\ e\ e \mid cons\ e\ e$
$\quad \mid {}^{\tau}MS\ e \mid e\langle\tau\rangle$

$\tau = Nat \mid \tau \rightarrow \tau \mid \tau^{*}$
$\quad \mid \forall\alpha.\tau \mid \alpha \mid L$

Conversion Summary

- Seals are introduced at Scheme/ML boundaries, to protect type variables.
- $\langle \alpha, \tau \rangle$ is a seal on the variable α, which should have type τ back in ML.
- Conversion strategies κ are types that might contain seals.
- Type substitutions η map type variables to closed types.

Parametricity / Fundamental Theorem

For all seal-free terms e and e, type environments Δ and value environments Γ:

1. If $\Delta;\Gamma \vdash \mathsf{e} : \tau$ then $\Delta;\Gamma \vdash \mathsf{e} \lesssim \mathsf{e} : \tau$
2. If $\Delta;\Gamma \vdash \mathsf{e} : \mathsf{L}$ then $\Delta;\Gamma \vdash \mathsf{e} \lesssim \mathsf{e} : \mathsf{L}$

"If type checking proves that e has type τ, then our logical relation will prove that e is parametric at type τ."

Proof Strategy: Logical Relations

- Syntactically relate terms in each language.
- $e \lesssim e' : \tau$ means that a machine running e will behave no differently from a machine running e'.
- ($\lesssim$) only relates terms that are parametric at type τ.

Defining $\lesssim$

- The paper defines two logical relations, one for Scheme and one for ML.
- The relation for ML is straightforward.
  - Identical values are related.
  - Related functions map related inputs to related outputs.
- Scheme is trickier...

Nontermination

```
(define omega
    ((λx . x x) (λx . x x)))
```

"You can't do *that* in your typed languages!" -RBF

Ha

Ha

Ha

Solution: Step-Indexing

- Although all ML programs terminate, Scheme programs may fail or loop forever.
- Step-indexed logical relations guarantee some number of computational steps.
- $\lessapprox^k$ relates terms for up to *k* steps.

Ahmed [2006] "Step-indexed syntactic logical relations for recursive and quantified types".

Defining $\lesssim$

$\delta \vdash \mathsf{n} \lesssim^{k} \mathsf{n} : \mathsf{Nat}$ *Unconditionally*

Identical natural numbers are related for *k* steps. No problem!

Defining $\lesssim$

$\delta \vdash \mathsf{n} \lesssim^{k} \mathsf{n} : \mathsf{Nat}$

$\delta \vdash \mathsf{v} \lesssim^{k} \mathsf{v'} : \alpha$

Unconditionally

$(k, \mathsf{v}, \mathsf{v'}) \in \delta(\alpha)$

Two values are related at type α
if the type relation δ says so.

Defining $\lesssim$

$\delta \vdash \mathsf{n} \lesssim^{k} \mathsf{n} : \mathsf{Nat}$

$\delta \vdash \mathsf{v} \lesssim^{k} \mathsf{v'} : \alpha$

$\delta \vdash [..., \mathsf{v}_n] \lesssim^{k} [..., \mathsf{v'}_n] : \tau^*$

Unconditionally

$(k, \mathsf{v}, \mathsf{v'}) \in \delta(\alpha)$

$\forall j<k.\ \forall i \leq n.$

$\delta \vdash \mathsf{v}_i \lesssim^{j} \mathsf{v'}_i : \tau$

Two lists are related for *k* steps if you can't tell apart any pair of elements within *j*<*k* steps.

Defining $\lesssim$

$$\frac{\vdots}{\delta \vdash \lambda x{:}\tau\,.\,e \lesssim^{k} \lambda x'{:}\tau\,.\,e' : \tau \to \tau'}$$

$$\forall j{<}k.\ \forall v, v'\ .\ \delta \vdash v \lesssim^{j} v : \tau \Rightarrow \delta \vdash e[v/x] \lesssim^{j} e'[v'/x'] : \tau'$$

Two functions are related for *k* steps
if, given arguments related for *j*<*k* steps,
the outputs are related for *j* steps.

Defining $\lesssim$

$\delta \vdash \mathbf{e} \lesssim^{k} \mathbf{e'} : \tau$

$\vdots$

$\forall j<k.$
$(\mathbf{e} \rightarrow^{j} \text{error} \Rightarrow \mathbf{e'} \rightarrow^{*} \text{error})$
$\wedge$
$(\forall \mathbf{v}.\, \mathbf{e} \rightarrow^{j} \mathbf{v} \Rightarrow$
$\exists \mathbf{v'}\, .\, \mathbf{e'} \rightarrow^{*} \mathbf{v'} \wedge \delta \vdash \mathbf{v} \lesssim^{k-j} \mathbf{v'} : \tau)$

Two expressions are related for *k* steps
if they both explode
or both step to related values.

`

Bridge Lemma

For all k≥0 and type relations δ:

1. If $\delta \vdash \mathbf{e} \lesssim^{k} \mathbf{e}' : \tau$ then $\delta \vdash \mathsf{SM}^{\tau/\delta}\, \mathbf{e} \lesssim^{k} \mathsf{SM}^{\tau/\delta}\, \mathbf{e}' : \mathsf{L}$
2. If $\delta \vdash \mathbf{e} \lesssim^{k} \mathbf{e}' : \mathsf{L}$ then $\delta \vdash \mathsf{MS}^{\tau/\delta}\, \mathbf{e} \lesssim^{k} \mathsf{MS}\, \mathbf{e}' : \tau$

"Sealing respects the logical relation."

Parametricity!

- If a term has type $\forall\alpha\ .\ \alpha * \alpha \rightarrow \alpha$
- Then it is **fst** $= \lambda(x,y)\ .\ x$
  or it is **snd** $= \lambda(x,y)\ .\ y$
- (Or it always raises an **error**)
- (Or it always diverges)

Application: Contracts

- Contracts are like types, but stronger.
- Dynamically check invariants.
- Using seals, we can give an ML type to a Scheme term as its behavioral specification.
- Bridge Lemma gives a simple implementation of higher-order, polymorphic contracts.

See Findler & Felleisen [2002] "Contracts for higher-order functions" for more on contracts and Sumii & Pierce [2004] "A Bisimulation for Dynamic Sealing" for an alternate approach.

Bottom Line

- Proved parametricity in a (simple) multi-language setting.
- Keep the clean abstractions of ML by adding a little enforcement to Scheme.
- Step-indexing handles non-termination (and recursive types).
- One step towards language interoperability.

The End