Early work in partial evaluation (PE) was often carried out in the context of the lambda-calculus because it served as a clean core language in which basic techniques could be formalized and reasoned about in a rigorous way. However, now that basic partial evaluation techniques are well-understood, PEPM submissions should aim to present techniques in the context of commonly-used programming languages. While papers may present or formalize a technique in terms of a “clean core language”, submissions that attempt to handle realistic language features or convincingly argue that the proposed techniques can be applied to more challenging language features, may be more highly valued since they provide a more effective foundation for transitioning techniques into actual software development practice. Exceptions to this general rule might occur if a submission is attempting to show, e.g., the benefits of transformation/optimization on representations used in theorem-proving tools that use extensions of the lambda-calculus as an underlying representation.

- **Failure to provide examples/applications that speak to a broader community:** Early work in partial evaluation often used the “power function”, “dot product” or similar examples to illustrate a technique. Since PE concepts are now fairly well-understood in the PEPM community, such examples should be avoided in PEPM submissions and replaced with examples that could convey the utility of PE and other program transformation techniques to a larger audience. Our aim is to grow the number of people from other areas that look to PEPM for solutions relevant to their problems. People from other applications domains will likely find examples such as those listed above irrelevant and unconvincing. It’s time for PE to move beyond the power function.
Rain keeps coming, snow pounds the Sierra — many homes, businesses will be without electricity for days because of storm’s brutality

MORE THAN 50,000 STILL LACK POWER

CAMPAIGN 2008

This time, under-30 voters are showing up

By Joe Garofoli
Chronicle Staff Writer

For more than three decades, it was the hallowest of presidential campaign promises: “And we’re going to get out the youth vote!” Yeah, right. Politicians rarely talked about the issues that mattered to young people, and the under-30 crowd returned the favor by not voting.

Young people did show up four years ago — to vote against George W. Bush. But on Thursday, they showed up in record numbers to vote for somebody, helping to propel Sen. Barack Obama to victory in the Iowa Democratic caucuses.

They showed that appealing to the under-30 vote doesn’t have to be a hollow promise for candidates who know how to translate their online love into real-world votes.

The number of under-30 Iowa caucus-goers tripled compared with 2004, and more than 17 percent of young Democratic voters supported Obama. Exit polls found 32 percent of the nearly 259,000 Democratic voters were under 30. GOP caucuses were more than 60 percent older.

Following the votes: Independents are a major force this year. They comprise nearly half of New Hampshire’s registered voters, and presidential hopefuls are scrambling for their attention as that state’s primary approaches.

Democrats: “Change” is the hot topic in New Hampshire debate as a new poll shows Barack Obama and Hillary Rodham Clinton in a dead heat.

GOP: High stakes in New Hampshire’s closely contested primary.
Motivation: Typed staging with side effects

Code generation
  - partial evaluation
  - embedded domain-specific languages
  - special-purpose processors

\[ \lambda x. x \times x \times x \times x \times x \times x \times x \times x \times 1 \]
Motivation: Typed staging with side effects

Code generation
  ▶ partial evaluation
  ▶ embedded domain-specific languages
  ▶ special-purpose processors

\[ \langle \lambda x. x \times x \times x \times x \times x \times x \times x \times x \times 1 \rangle \]
Motivation: Typed staging with side effects

Code generation
- partial evaluation
- embedded domain-specific languages
- special-purpose processors

\[ \lambda x. \sim (\text{power} \ 7 \ x) \]
Motivation: Typed staging with side effects

Code generation

- partial evaluation
- embedded domain-specific languages
- special-purpose processors

\[ \text{run} \langle \lambda x. \sim (\text{power } 7 \langle x \rangle) \rangle \]
Motivation: Typed staging with side effects

Code generation
▶ partial evaluation
▶ embedded domain-specific languages
▶ special-purpose processors

Type safety
▶ generate well-typed, well-scoped code: no scope extrusion
▶ splice open code yet run closed code: keep α-equivalence

\[
\text{run} \langle \lambda x. \sim (\text{power} \ 7 \ \langle x \rangle) \rangle \\
\text{run} \langle \lambda x. \sim (\text{power} \ \langle x \rangle \ 7) \rangle \quad \times
\]
Motivation: Typed staging with side effects

Code generation
- partial evaluation
- embedded domain-specific languages
- special-purpose processors

Type safety
- generate well-typed, well-scoped code: no scope extrusion
- splice open code yet run closed code: keep α-equivalence

run $\langle \lambda x. \sim (\text{power} \ 7 \ \langle x \rangle) \rangle$
run $\langle \lambda x. \sim (\text{power} \ 7 \ \langle 2 \rangle) \rangle$
Motivation: Typed staging with side effects

Code generation
- partial evaluation
- embedded domain-specific languages
- special-purpose processors

Type safety
- generate well-typed, well-scoped code: no scope extrusion
- splice open code yet run closed code: keep $\alpha$-equivalence

\[
\text{run} \langle \lambda x. \sim (\text{power 7 } \langle x \rangle) \rangle
\]
\[
\text{run} \langle \lambda x. \sim (\text{power 7 } \langle \text{true} \rangle) \rangle \quad \times
\]
Motivation: Typed staging with side effects

Code generation
- partial evaluation
- embedded domain-specific languages
- special-purpose processors

Type safety
- generate well-typed, well-scoped code: no scope extrusion
- splice open code yet run closed code: keep \( \alpha \)-equivalence

\[
\begin{align*}
\text{run} \langle \lambda x. \sim (\text{power 7} \langle x \rangle) \rangle \\
\text{run} \langle \lambda x. \sim (\text{power 7} \langle \text{true} \rangle) \rangle & \times \\
\text{run} \langle x \rangle & \times
\end{align*}
\]
Motivation: Typed staging with side effects

Code generation
- partial evaluation
- embedded domain-specific languages
- special-purpose processors

Type safety
- generate well-typed, well-scoped code: no scope extrusion
- splice open code yet run closed code: keep α-equivalence

run \( \lambda x. \sim (\text{power } 7 \langle x \rangle) \)
run \( \lambda x. \sim (\text{power } 7 \langle \text{true} \rangle) \)
run \( x \)
Motivation: Typed staging with side effects

Code generation
- partial evaluation
- embedded domain-specific languages
- special-purpose processors

Type safety
- generate well-typed, well-scoped code: no scope extrusion
- splice open code yet run closed code: keep $\alpha$-equivalence

\[
\begin{align*}
\text{run } & \lambda x. \sim (\text{power 7 } \langle x \rangle) \\
\text{run } & \lambda x. \sim (\text{power 7 } \langle \text{true} \rangle) \times \\
\text{run } & \langle x \rangle \times \\
\text{run } & \lambda x. \sim (\ldots \text{run } \langle 2 \rangle \ldots )
\end{align*}
\]
Motivation: Typed staging with side effects

Code generation
- partial evaluation
- embedded domain-specific languages
- special-purpose processors

Type safety
- generate well-typed, well-scoped code: no scope extrusion
- splice open code yet run closed code: keep α-equivalence

\[
\begin{align*}
\text{run } \langle \lambda x. \sim (\text{power } 7 \langle x \rangle) \rangle & \quad \times \\
\text{run } \langle \lambda x. \sim (\text{power } 7 \langle \text{true} \rangle) \rangle & \quad \times \\
\text{run } \langle x \rangle & \quad \times \\
\text{run } \langle \lambda x. \sim (\ldots \text{run } \langle x \rangle \ldots) \rangle & \quad \times
\end{align*}
\]
Motivation: Typed staging with side effects

Code generation
- partial evaluation
- embedded domain-specific languages
- special-purpose processors

Type safety
- generate well-typed, well-scoped code: no scope extrusion
- splice open code yet run closed code: keep $\alpha$-equivalence

\[
\begin{align*}
gensym + \text{ binding} \\
\text{run} \langle \lambda x. \sim (\text{power} 7 \langle x \rangle) \rangle \\
\text{run} \langle \lambda x. \sim (\text{power} 7 \langle \text{true} \rangle) \rangle & \times \\
\text{run} \langle x \rangle & \times \\
\text{run} \langle \lambda x. \sim (\ldots \text{run} \langle x \rangle \ldots) \rangle & \times
\end{align*}
\]
Motivation: Typed staging with side effects

Code generation
  ▶ partial evaluation
  ▶ embedded domain-specific languages
  ▶ special-purpose processors

Type safety
  ▶ generate well-typed, well-scoped code: no scope extrusion
  ▶ splice open code yet run closed code: keep $\alpha$-equivalence

\[
gensym + \text{binding}
\]
\[
\begin{align*}
\text{run } & \langle \lambda y. \sim (\text{power 7 } \langle y \rangle) \rangle \\
\text{run } & \langle \lambda x. \sim (\text{power 7 } \langle \text{true} \rangle) \rangle \\
\text{run } & \langle x \rangle \\
\text{run } & \langle \lambda x. \sim (\ldots \text{run } \langle x \rangle \ldots) \rangle \\
\end{align*}
\]
Motivation: Typed staging with side effects

Code generation
- partial evaluation
- embedded domain-specific languages
- special-purpose processors

Type safety
- generate well-typed, well-scoped code: no scope extrusion
- splice open code yet run closed code: keep $\alpha$-equivalence

Mutable state
- let insertion, assert insertion
- count generated operations
Motivation: Typed staging with side effects

Code generation
- partial evaluation
- embedded domain-specific languages
- special-purpose processors

Type safety
- generate well-typed, well-scoped code: no scope extrusion
- splice open code yet run closed code: keep $\alpha$-equivalence

Mutable state and delimited control
- let insertion, assert insertion
- count generated operations
- partial evaluation of sum types and delimited control
Motivation: Typed staging with side effects

Code generation
- partial evaluation
- embedded domain-specific languages
- special-purpose processors

Type safety
- generate well-typed, well-scoped code: no scope extrusion
- splice open code yet run closed code: keep $\alpha$-equivalence

Mutable state and delimited control
- let insertion, assert insertion
- count generated operations
- partial evaluation of sum types and delimited control

Pick two.
We translate staging away: Simplified MetaOCaml $\Rightarrow$ System F
Closing the stage
From staged code to typed closures

Yukiyoshi Kameyama  Oleg Kiselyov  Chung-chieh Shan
University of Tsukuba  FNMOC  Rutgers University
kameyama@acm.org  oleg@pobox.com  ccshan@rutgers.edu

PEPM, January 7, 2008
let rec power n x =
    if n = 0
    then 1
    else x × (power (n - 1) x)
let power7 = λx. (power 7 x)
let rec power n x =
  if n = 0
  then 1
  else x × (power (n - 1) x)
let power7 = λx. (power 7 x)
let rec power n c =
  if \( n = 0 \)
  then \(<1>\)
  else \(<~c \times ~(power (n - 1) c)>\)
let power 7 = \(<\lambda x. ~(power 7 <x>)>\)
From code to thunks

let rec power n c =
    if n = 0
    then 〈1〉
    else 〈~ c × ~ (power (n - 1) c)〉
let power7 = 〈λx. ~ (power 7 〈x〉)〉

let rec power n c =
    if n = 0
    then \( \lambda(). 1 \)
    else \( \lambda(). c () \times \text{power} (n - 1) c () \)
let power7 = \( \lambda(). \lambda x. \text{power} 7 (\lambda(). x) () \)
let rec power n c =
  if n = 0
  then 〈1〉
  else 〈c × (power (n - 1) c)〉
let power7 = 〈λx. ~(power 7 〈x〉)〉

↓

let rec power n c =
  if n = 0
  then λ(). 1
  else λ(). c () × power (n - 1) c ()
let power7 = λ(). λx. power 7 (λ(). x) ()
From code to thunks

let rec power n c =
  if n = 0
  then <1>
  else <~c × ~(power (n - 1) c)>
let power7 = <λx. ~(power 7 <x>)}

\[
\downarrow
\]

let rec power n c =
  if n = 0
  then λ(). 1
  else
  let v = power (n - 1) c
  in
  λ(). c () × power (n - 1) c ()
let power7 =
  λ(). λx. power 7 (λ(). x) ()
let rec power n c =
    if n = 0
    then ⟨1⟩
    else ⟨~c × ~(power (n - 1) c)⟩
let power7 = ⟨λx. ~(power 7 ⟨x⟩)⟩

let rec power n c =
    if n = 0
    then λ(). 1
    else let v = power (n - 1) c in λ(). c () × v ()
let power7 = let v = power 7 (λ(). x) in λ(). λx. v ()
From code to thunks

let rec power n c =
    if n = 0
    then 〈1〉
    else 〈~c × ~(power (n - 1) c)〉
let power7 = 〈λx. ~(power 7 〈x〉)〉

↓

let rec power n c =
    if n = 0
    then λ(). 1
    else let v = power (n - 1) c in λ(). c () × v ()
let power7 = let v = power 7 (λ(). x) in λ(). λx. v ()
let rec power n c =
  if n = 0
  then 〈1〉
  else 〈~c × ~(power (n - 1) c)〉
let power7 = 〈λx. ~(power 7 〈x〉)〉

let rec power n c =
  if n = 0
  then λ(_). 1
  else let v = power (n - 1) c in λ(_). c (_)? × v (_)
let power7 = let v = power 7 (λ(_). x) in λ(). λx. v (_)

From code to thunks
From code to closures

let rec power n c =
  if n = 0
  then 〈1〉
  else 〈~c × ~(power (n - 1) c)〉
let power7 = 〈λx. ~(power 7 〈x〉)〉

↓

let rec power n c =
  if n = 0
  then λ(x). 1
  else let v = power (n - 1) c in λ(x). c (x) × v (x)
let power7 = let v = power 7 (λ(x). x) in λ(). λx. v (x)
let rec power n c =
  if n = 0
  then $\langle 1 \rangle$
  else $\langle c \times (\text{power} \ (n-1) \ c) \rangle$
let power7 = $\langle \lambda x. \ (\text{power} \ 7 \ (\langle x \rangle)) \rangle$

⇒
let rec power n c =
  if n = 0
  then $\lambda(x). 1$
  else let $v = \text{power} \ (n-1) \ c$ in $\lambda(x). c \ (x) \times v \ (x)$
let power7 = let $v = \text{power} \ 7 \ (\lambda(x). x)$ in $\lambda(). \lambda x. v \ (x)$
From code to closures

```plaintext
let rec power n c =
  if n = 0
  then ⟨1⟩
  else ⟨~c × ~(power (n - 1) c)⟩
let power7 = ⟨λx. ~(power 7 ⟨x⟩)⟩
let power7sum = ⟨λx. λy. ~(power 7 ⟨x + y⟩)⟩

let rec power n c =
  if n = 0
  then λ(x). 1
  else let v = power (n - 1) c in λ(x). c (x) × v (x)
let power7 = let v = power 7 (λ(x). x) in λ(). λx. v (x)
```
let rec power n c =
    if n = 0
    then <1>
    else <~c × ~(power (n - 1) c)>
let power7 = <λx. ~(power 7 <x>)>
let power7sum = <λx. λy. ~(power 7 <x + y>)>
let rec power n c =
    if n = 0
    then λ(x). 1
    else let υ = power (n - 1) c in λ(x). c (x) × υ (x)
let power7 = let υ = power 7 (λ(x). x) in λ(). λx. υ (x)
From code to closures

let rec power n c =
    if n = 0
    then 〈1〉
    else 〈~ c × ~ (power (n - 1) c)〉
let power7 = 〈λx. ~(power 7 〈x〉)〉
let power7sum = 〈λx. λy. ~(power 7 〈x + y〉)〉
let rec power n c =
    if n = 0
    then λr. 1
    else let v = power (n - 1) c in λr. c r × v r
let power7 = let v = power 7 (λ(x). x) in λ(). λx. v (x)
let rec power n c =
  if n = 0
  then \(1\)
  else \(<c \times (power (n - 1) c)\>
let power7 = \(\lambda x. ~ (power 7 \langle x \rangle)\)
let power7sum = \(\lambda x. \lambda y. ~ (power 7 \langle x + y \rangle)\)

let rec power n c =
  if n = 0
  then \(\lambda r \ . \ 1\)
  else let v = power (n - 1) c in \(\lambda r \ . \ c \ r \times v \ r\)
let power7 = let v = power 7 \(\lambda(x). x\) in \(\lambda(). \lambda x. v(x)\)
let power7sum = let v = power 7 \(\lambda(x,y). x + y\) in \(\lambda(). \lambda x. v(x,y)\)
let rec power n c =
    if n = 0
    then <1>
    else <~ c × ~(power (n - 1) c)>
let power7 = <λx. ~(power 7 <x>)>
let power7sum = <λx. λy. ~(power 7 <x + y>)>
let eta f = <λx. ~(f <x>)>
let rec power $n$ $c$ =
  if $n$ = 0
  then 〈1〉
  else 〈~$c$ × ~(power $(n - 1)$ $c$)〉

let power7 = 〈λ$x$. ~(power 7 〈$x$〉)〉
let power7sum = 〈λ$x$. λ$y$. ~(power 7 〈$x$ + $y$〉)〉
let eta $f$ = 〈λ$x$. ~(f 〈$x$〉)〉

let rec power'$ n$ $c$ =
  if $n$ = 0
  then 〈1〉
  else if $n$ mod 2 = 0
    then 〈let $z$ = ~$c$ × ~$c$ in ~(power' $(n ÷ 2)$ 〈$z$〉)〉
    else 〈let $z$ = ~$c$ × ~$c$ in ~$c$ × ~(power' $((n - 1) ÷ 2)$ 〈$z$〉)〉
let rec power n c =
  if n = 0
  then ⟨1⟩
  else ⟨~c × ~(power (n - 1) c)⟩
let power7 = ⟨λx. ~(power 7 ⟨x⟩)⟩
let power7sum = ⟨λx. λy. ~(power 7 ⟨x + y⟩)⟩
let eta f = ⟨λx. ~(f ⟨x⟩)⟩

⟨A⟩ ⇒ ... → A
From higher-order code to higher-rank polymorphism

```ml
let rec power n c =
  if n = 0
  then ⟨1⟩
  else ⟨~c × ~(power (n - 1) c)⟩
let power7 = ⟨λx. ~(power 7 ⟨x⟩)⟩
let power7sum = ⟨λx. λy. ~(power 7 ⟨x + y⟩)⟩
let eta f = ⟨λx. ~(f ⟨x⟩)⟩
```

\[
⟨A⟩ \Rightarrow \ldots \rightarrow A \\
⟨A⟩ \rightarrow ⟨B⟩ \Rightarrow \forall \pi. ((\ldots, \pi) \rightarrow A) \rightarrow ((\ldots, \pi) \rightarrow B)
\]
let rec power n c =
  if n = 0
  then ⟨1⟩
  else ⟨~c × ~({power (n - 1) c})⟩
let power7 = ⟨λx. ~({power 7 ⟨x⟩})⟩
let power7sum = ⟨λx. λy. ~({power 7 ⟨x + y⟩})⟩
let ηf = ⟨λx. ~({f ⟨x⟩})⟩

\[
\begin{align*}
\langle A \rangle & \Rightarrow \ldots \rightarrow A \\
\langle A \rangle \rightarrow \langle B \rangle & \Rightarrow \forall \pi. ((\ldots, \pi) \rightarrow A) \rightarrow ((\ldots, \pi) \rightarrow B) \\
(\langle A \rangle \rightarrow \langle B \rangle) \rightarrow \langle C \rangle & \Rightarrow \forall \pi. (\forall \rho. ((\ldots, \pi, \rho) \rightarrow A) \\
& \hspace{1cm} \rightarrow ((\ldots, \pi, \rho) \rightarrow B)) \\
& \hspace{1cm} \rightarrow ((\ldots, \pi) \rightarrow C)
\end{align*}
\]
Outline

Simplified MetaOCaml $\Rightarrow$ System F

Staged code $\Rightarrow$ Typed closures

Higher-order functions $\Rightarrow$ Higher-rank polymorphism

- Extension among environments $\Rightarrow$ Injection among types

  Scope extrusion $\Rightarrow$ Type error
Coercions

let rec power' n c =
    if n = 0 then \langle 1 \rangle
    else if n mod 2 = 0 then \langle let z = \sim c \times \sim c in \sim (power' (n \div 2)) \rangle
    else \langle let z = \sim c \times \sim c in \sim c \times \sim (power' \ldots) \rangle
Coercions

\[ c = \langle x \rangle \]

\[ \cdot \quad \vdash \] \[ \lambda (x). x \]

Extended

let rec power' n c =
    if n = 0
    then \langle 1 \rangle
    else if n mod 2 = 0
        then \langle let z = \sim c \times \sim c in \sim (power' (n \div 2)) \rangle
    else \langle let z = \sim c \times \sim c in \sim c \times \sim (power' \rangle
Coercions

\[ c = \langle x \rangle \]

\[ \Rightarrow \]

\[ \vdash \]

\[ x \vdash \lambda(x). x \]

\[ \Rightarrow \]

\[ \vdash \lambda(x, z_1). x \]

let rec power\(^{'}\) \( n \ c = \)

\[ \text{if } n = 0 \]

then \( \langle 1 \rangle \)

\[ \text{else if } n \mod 2 = 0 \]

then \( \langle \text{let } z = \neg c \times \neg c \text{ in } \neg (\text{power'} \ (n \div 2)) \rangle \)

\[ \text{else } \langle \text{let } z = \neg c \times \neg c \text{ in } \neg c \times \neg (\text{power'} \ (n \div 2)) \rangle \]
Coercions

\[ c = \langle x \rangle \]

\[
\Downarrow
\]

\[
\vdash
\]

\[
\text{Extend}
\]

\[
x \quad \vdash \quad \lambda(x). \, x
\]

\[
\Downarrow
\]

\[
\text{Extend}
\]

\[
x, z_1 \quad \vdash \quad \lambda(x, z_1). \, x
\]

\[
\Downarrow
\]

\[
\text{Extend}
\]

\[
x, z_1, z_2 \quad \vdash \quad \lambda(x, z_1, z_2). \, x
\]

\[
\Downarrow
\]

\[
\text{Coerce}
\]

\[
\Downarrow
\]

\[\text{power'} \, n \]

\[
\Downarrow
\]

\[
\lambda c. \, \lambda r. \ldots \, cr \times cr \ldots
\]

\[
\text{let rec power'} \, n \, c =
\]

\[
\text{if } n = 0
\]

\[
\text{then } \langle 1 \rangle
\]

\[
\text{else if } n \, \text{mod} \, 2 = 0
\]

\[
\text{then } \langle \text{let } z = \sim c \times \sim c \text{ in } \sim (\text{power'} \, (n \div 2)) \rangle
\]

\[
\text{else } \langle \text{let } z = \sim c \times \sim c \text{ in } \sim c \times \sim (\text{power'} \, n - 1) \rangle
\]
Coercions

\[
c = \langle x \rangle
\]
\[
\downarrow
data
\]
\[
\lambda c. \lambda r. \ldots. cr \times cr \ldots
\]
\[
\downarrow\text{Coerce}
data
\]
\[
\lambda c. \lambda(x, r). \ldots. c(x, r) \times c(x, r) \ldots
\]
\[
\downarrow\text{Coerce}
data
\]
\[
\lambda c. \lambda(x, z_1, r). \ldots. c(x, z_1, r) \times c(x, z_1, r) \ldots
\]
\[
\downarrow\text{Coerce}
data
\]
\[
\lambda c. \lambda(x, z_1, z_2, r). \ldots. c(x, z_1, z_2, r) \times c(x, z_1, z_2, r) \ldots
\]
Coercions

\[
\begin{aligned}
\langle \text{int} \rangle & \quad \downarrow \quad \langle \text{int} \rangle \\
. & \quad \vdash (\) \rightarrow \text{int} \\
& \quad \downarrow \quad \downarrow \\
\text{Extend} & \quad \text{Coerce} \\
\vdash (\text{int}) \rightarrow \text{int} & \quad \forall \pi.((\pi) \rightarrow \text{int}) \rightarrow ((\pi) \rightarrow \text{int}) \\
& \quad \downarrow \quad \downarrow \\
\text{Extend} & \quad \text{Coerce} \\
\vdash (\text{int}, \text{int}) \rightarrow \text{int} & \quad \forall \pi.((\text{int}, \pi) \rightarrow \text{int}) \rightarrow ((\text{int}, \pi) \rightarrow \text{int}) \\
& \quad \downarrow \quad \downarrow \\
\text{Extend} & \quad \text{Coerce} \\
\vdash (\text{int}, \text{int}, \text{int}) \rightarrow \text{int} & \quad \forall \pi.((\text{int}, \text{int}, \pi) \rightarrow \text{int}) \rightarrow ((\text{int}, \text{int}, \pi) \rightarrow \text{int}) \\
& \quad \downarrow \quad \downarrow \\
\text{Extend} & \quad \text{Coerce} \\
\vdash (\text{int}, \text{int}, \text{int}, \text{int}) \rightarrow \text{int} & \quad \forall \pi.((\text{int}, \text{int}, \text{int}, \pi) \rightarrow \text{int}) \\
& \quad \rightarrow ((\text{int}, \text{int}, \text{int}, \pi) \rightarrow \text{int})
\end{aligned}
\]
Coercions

Indexed by source type
(identity for int)

Induced by the extension
from the environment of creation
to the environment of use

We translate terms by induction on
typing derivations, yet the translation
is compositional in some senses.
Coercions elaborate environment polymorphism

In our source language
From environment classifiers (Taha, Nielsen, Calcagno, Moggi)

\(<\text{int}\)^{\alpha}\)

to contextual modal type theory (Nanevski, Pfenning, Pientka)?

\[
\begin{array}{c}
[] \text{int} \\
[x: \text{int}] \text{int} \\
[x: \text{int}, z_1: \text{int}] \text{int} \\
[x: \text{int}, z_1: \text{int}, z_2: \text{int}] \text{int}
\end{array}
\]
Coercions elaborate environment polymorphism

In our source language
From environment classifiers (Taha, Nielsen, Calcagno, Moggi)

\[ \langle \text{int} \rangle^\alpha \]

to contextual modal type theory (Nanevski, Pfenning, Pientka)?

[] int [int] int [int, int] int [int, int, int] int

Our “de Bruijn indices” maintain \( \alpha \)-equivalence and avoid the need for \( \rho \)-polymorphism and negative side conditions.
Coercions elaborate environment polymorphism

In our source language
From *environment classifiers* (Taha, Nielsen, Calcagno, Moggi)

\[ \langle \text{int} \rangle^\alpha \]

to *contextual modal type theory* (Nanevski, Pfenning, Pientka)?

\[
[\ ] \text{int} \quad [\text{int}] \text{int} \quad [\text{int}, \text{int}] \text{int} \quad [\text{int}, \text{int}, \text{int}] \text{int}
\]

Our “de Bruijn indices” maintain α-equivalence and avoid the need for ρ-polymorphism and negative side conditions.

In our target language
System F lacks environment polymorphism (weakening), so we roll our own.
Scope extrusion

How to count multiplications as we generate them?

let count = ref 0
let rec power n c =
  if n = 0
  then <1>
  else count ← !count + 1;...
Scope extrusion

How to count multiplications as we generate them?

Use environment may no longer extend creation environment.

<int> state is safe: the identity coercion is always available.

\[
\begin{align*}
\text{let } \text{count} &= \text{ref } 0 \\
\text{let rec power } n \ c &= \\
&\quad \text{if } n = 0 \\
&\quad \text{then } \langle 1 \rangle \\
&\quad \text{else } \text{count } \leftarrow !\text{count} + 1; \ldots
\end{align*}
\]

\langle int \rangle state risks scope extrusion and running open code.

\[
\begin{align*}
\text{let } x &= \text{ref } \langle 1 \rangle \text{ in} \\
&\langle \lambda y. \sim(x \leftarrow \langle y \rangle; \langle () \rangle) \rangle; \\
!x &\rightsquigarrow \langle y \rangle
\end{align*}
\]
Conclusion

Simplified MetaOCaml $\Rightarrow$ System F

Staged code $\Rightarrow$ Typed closures

Higher-order functions $\Rightarrow$ Higher-rank polymorphism

Extension among environments $\Rightarrow$ Injection among types

Scope extrusion $\Rightarrow$ Type error

Small-step operational semantics for source language
(need to show: preserved by translation)

Administrative reductions incur abstraction overhead
(eliminated by true staging) despite specialization