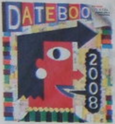
transformation/optimization on representations used in theorem-proving tools that use extensions of the lambda-calculus as an underlying representation.

- **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

# San Francisco Chronicle

★★★★★• Printed on recycled paper | SUNDAY, JANUARY 6, 2008 | sfgate.com $1.50




**ART TOMORROW**

Chronicle critics on what not to miss in theater, music, dance, art and film this year.
**Datebook, 16**

**TRAGEDY AT YEAR'S END**

A young father, knowing that gunfire was inevitable, threw himself over his 9-year-old daughter. In saving her, Albert Collins became San Francisco's final homicide victim in a deadly year.
**Bay Area, B1**

**WUNDERKIND. OR NOT.**

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

**An employee at** Nick's Restaurant in Pacifica gets light from a propane lantern as she makes coffee. Like much of the city, Nick's operated without power as heavy rains and wind kept pounding the coast.

*By John Wildermuth and Steve Rubenstein*
CHRONICLE STAFF WRITERS

**The storm**

**CAMPAIGN 2008**

# This time, under-30 voters are showing up

*By Joe Garofoli*
CHRONICLE STAFF WRITER

For more than three decades, it was the hollowest 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 57 percent of young Democratic voters supported Obama. Exit polls found 22 percent of the nearly 239,000 Democratic voters were under 30. GOP caucus winner Mike Huckabee [illegible]

▸ **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. **A5**

▸ **Democrats:** "Change" is the hot topic in a New Hampshire debate as a new poll shows Barack Obama and Hillary Rodham Clinton in a dead heat. **A4**

▸ **GOP:** High stakes kindle clashes among contenders for [illegible]

# Motivation: Typed staging with side effects

Code generation

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

$$\lambda x.\, x \times x \times x \times x \times x \times x \times x \times 1$$

# Motivation: Typed staging with side effects

Code generation

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

$$\langle \lambda x.\ x \times x \times x \times x \times x \times x \times x \times 1 \rangle$$

# Motivation: Typed staging with side effects

Code generation

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

$$\langle \lambda x.\ {\sim}(\mathtt{power}\ 7\ \langle x \rangle) \rangle$$

# Motivation: Typed staging with side effects

Code generation

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

$$\texttt{run}\langle \lambda x.\ \texttt{\~{}}(\mathrm{power}\ 7\ \langle x \rangle)\rangle$$

# 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

$$\mathtt{run}\langle\lambda x.\ \sim(\mathrm{power}\ 7\ \langle x\rangle)\rangle$$

$$\mathtt{run}\langle\lambda x.\ \sim(\mathrm{power}\ \langle x\rangle\ 7)\rangle \qquad \times$$

# 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

$$\mathtt{run}\langle\lambda x.\ \sim(\mathtt{power}\ 7\ \langle x\rangle)\rangle$$
$$\mathtt{run}\langle\lambda x.\ \sim(\mathtt{power}\ 7\ \langle 2\rangle)\rangle$$

# 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 α-equivalence

$$\mathtt{run}\langle\lambda x.\ \sim(\mathtt{power}\ 7\ \langle x\rangle)\rangle$$

$$\mathtt{run}\langle\lambda x.\ \sim(\mathtt{power}\ 7\ \langle\mathtt{true}\rangle)\rangle \qquad \times$$

# 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

$$\mathtt{run}\,\langle \lambda x.\ \sim(\mathrm{power}\ 7\ \langle x \rangle)\rangle$$

$$\mathtt{run}\,\langle \lambda x.\ \sim(\mathrm{power}\ 7\ \langle \mathrm{true} \rangle)\rangle \qquad \times$$

$$\mathtt{run}\,\langle x \rangle \qquad \times$$

# 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

$$\texttt{run}\langle\lambda x.\,\sim(\text{power } 7\ \langle x\rangle)\rangle$$

$$\texttt{run}\langle\lambda x.\,\sim(\text{power } 7\ \langle\texttt{true}\rangle)\rangle \qquad \times$$

$$\texttt{run}\langle x\rangle \qquad \times$$

# 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

| | |
|---|---|
| $\mathtt{run}\langle \lambda x.\ \char`~(\mathrm{power}\ 7\ \langle x \rangle)\rangle$ | |
| $\mathtt{run}\langle \lambda x.\ \char`~(\mathrm{power}\ 7\ \langle \mathrm{true} \rangle)\rangle$ | $\times$ |
| $\mathtt{run}\langle x \rangle$ | $\times$ |
| $\mathtt{run}\langle \lambda x.\ \char`~(\dots \mathtt{run}\langle 2 \rangle \dots)\rangle$ | |

# 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

| | |
|---|---|
| $\mathtt{run}\langle \lambda x.\ \mathord{\sim}(\mathrm{power}\ 7\ \langle x \rangle)\rangle$ | |
| $\mathtt{run}\langle \lambda x.\ \mathord{\sim}(\mathrm{power}\ 7\ \langle \mathrm{true} \rangle)\rangle$ | $\times$ |
| $\mathtt{run}\langle x \rangle$ | $\times$ |
| $\mathtt{run}\langle \lambda x.\ \mathord{\sim}(\ldots \mathtt{run}\langle x \rangle \ldots)\rangle$ | $\times$ |

# 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

$$\mathtt{run}\langle \overbrace{\lambda x.}^{\text{gensym + binding}} \sim(\mathrm{power}\ 7\ \langle x\rangle)\rangle$$

$$\mathtt{run}\langle \lambda x. \sim(\mathrm{power}\ 7\ \langle \mathrm{true}\rangle)\rangle \qquad \times$$

$$\mathtt{run}\langle x\rangle \qquad \times$$

$$\mathtt{run}\langle \lambda x. \sim(\ldots \mathtt{run}\langle x\rangle \ldots)\rangle \qquad \times$$

# 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

$$\mathtt{run}\,\langle \overbrace{\lambda y.}^{\text{gensym + binding}}\ {\sim}(\mathrm{power}\ 7\ \langle y \rangle)\rangle$$

$$\mathtt{run}\,\langle \lambda x.\ {\sim}(\mathrm{power}\ 7\ \langle \mathrm{true} \rangle)\rangle \qquad \times$$

$$\mathtt{run}\,\langle x \rangle \qquad \times$$

$$\mathtt{run}\,\langle \lambda x.\ {\sim}(\ldots \mathtt{run}\,\langle x \rangle \ldots)\rangle \qquad \times$$

# 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

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 α-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

# ?

$$
\begin{array}{l}
\text{let rec power } n\ x = \\
\quad \text{if } n = 0 \\
\quad \text{then } 1 \\
\quad \text{else } x \times (\text{power } (n-1)\ x) \\
\text{let power7} = \lambda x.\, (\text{power } 7\ x)
\end{array}
$$

let rec power $n$ $x$ =
    if $n = 0$
    then 1
    else $x \times$ (power $(n - 1)$ $x$)
let power7 = $\lambda x.$ (power 7 $x$)

$$
\begin{array}{l}
\text{let rec power } n\ c = \\
\quad \text{if } n = 0 \\
\quad \text{then } \langle 1 \rangle \\
\quad \text{else } \langle \sim c \times \sim(\text{power } (n-1)\ c) \rangle \\
\text{let power7} = \langle \lambda x.\ \sim(\text{power } 7\ \langle x \rangle) \rangle
\end{array}
$$

# From code to thunks

$$
\begin{array}{l}
\text{let rec power } n\ c = \\
\quad \text{if } n = 0 \\
\quad \text{then } \langle 1 \rangle \\
\quad \text{else } \langle \sim c \times \sim(\text{power } (n-1)\ c) \rangle \\
\text{let power7} = \langle \lambda x.\ \sim(\text{power } 7\ \langle x \rangle) \rangle
\end{array}
$$

$$\Downarrow$$

$$
\begin{array}{l}
\text{let rec power } n\ c = \\
\quad \text{if } n = 0 \\
\quad \text{then } \lambda().\ 1 \\
\quad \text{else } \lambda().\ c\ () \times \text{power } (n-1)\ c\ () \\
\text{let power7} = \lambda().\ \lambda x.\ \text{power } 7\ (\lambda().\ x)\ ()
\end{array}
$$

# From code to thunks

$$
\begin{array}{l}
\text{let rec power } n\ c = \\
\quad \text{if } n = 0 \\
\quad \text{then } \langle 1 \rangle \\
\quad \text{else } \langle \sim c \times \sim(\text{power } (n-1)\ c) \rangle \\
\text{let power7} = \langle \lambda x.\ \sim(\text{power } 7\ \langle x \rangle) \rangle
\end{array}
$$

$$\Downarrow$$

$$
\begin{array}{l}
\text{let rec power } n\ c = \\
\quad \text{if } n = 0 \\
\quad \text{then } \lambda().\ 1 \\
\quad \text{else } \lambda().\ c\ () \times \text{power } (n-1)\ c\ () \\
\text{let power7} = \lambda().\ \lambda x.\ \text{power } 7\ (\lambda().\ x)\ ()
\end{array}
$$

# From code to thunks

$$
\begin{array}{l}
\text{let rec power } n\ c = \\
\quad \text{if } n = 0 \\
\quad \text{then } \langle 1 \rangle \\
\quad \text{else } \langle \sim c \times \sim(\text{power } (n-1)\ c) \rangle \\
\text{let power7} = \langle \lambda x.\ \sim(\text{power } 7\ \langle x \rangle) \rangle
\end{array}
$$

$$\Downarrow$$

$$
\begin{array}{l}
\text{let rec power } n\ c = \\
\quad \text{if } n = 0 \\
\quad \text{then } \lambda().\ 1 \\
\quad \text{else} \qquad \lambda().\ c\ () \times \text{power } (n-1)\ c\ () \\
\text{let power7} = \qquad \lambda().\ \lambda x.\ \text{power } 7\ (\lambda().\ x)\ ()
\end{array}
$$

# From code to thunks

$$
\begin{array}{l}
\text{let rec power } n\ c = \\
\quad \text{if } n = 0 \\
\quad \text{then } \langle 1 \rangle \\
\quad \text{else } \langle \sim c \times \sim(\text{power } (n-1)\ c) \rangle \\
\text{let power7} = \langle \lambda x.\ \sim(\text{power } 7\ \langle x \rangle) \rangle
\end{array}
$$

$$\Downarrow$$

$$
\begin{array}{l}
\text{let rec power } n\ c = \\
\quad \text{if } n = 0 \\
\quad \text{then } \lambda().\ 1 \\
\quad \text{else let } v = \text{power } (n-1)\ c \text{ in } \lambda().\ c\ () \times v\ () \\
\text{let power7} = \text{let } v = \text{power } 7\ (\lambda().\ x) \text{ in } \lambda().\ \lambda x.\ v\ ()
\end{array}
$$

# From code to thunks

$$
\begin{array}{l}
\text{let rec power } n\ c = \\
\quad \text{if } n = 0 \\
\quad \text{then } \langle 1 \rangle \\
\quad \text{else } \langle \sim c \times \sim(\text{power } (n-1)\ c) \rangle \\
\text{let power7} = \langle \lambda x.\ \sim(\text{power } 7\ \langle x \rangle) \rangle
\end{array}
$$

$$\Downarrow$$

$$
\begin{array}{l}
\text{let rec power } n\ c = \\
\quad \text{if } n = 0 \\
\quad \text{then } \lambda().\ 1 \\
\quad \text{else let } v = \text{power } (n-1)\ c \text{ in } \lambda().\ c\ () \times v\ () \\
\text{let power7} = \text{let } v = \text{power } 7\ (\lambda().\ x) \text{ in } \lambda().\ \lambda x.\ v\ ()
\end{array}
$$

# From code to thunks

$$
\begin{array}{l}
\text{let rec power } n\ c = \\
\quad \text{if } n = 0 \\
\quad \text{then } \langle 1 \rangle \\
\quad \text{else } \langle \sim c \times \sim(\text{power } (n-1)\ c) \rangle \\
\text{let power7} = \langle \lambda x.\ \sim(\text{power } 7\ \langle x \rangle) \rangle
\end{array}
$$

$$\Downarrow$$

$$
\begin{array}{l}
\text{let rec power } n\ c = \\
\quad \text{if } n = 0 \\
\quad \text{then } \lambda(\ ).\ 1 \\
\quad \text{else let } v = \text{power } (n-1)\ c \text{ in } \lambda(\ ).\ c\ (\ ) \times v\ (\ ) \\
\text{let power7} = \text{let } v = \text{power } 7\ (\lambda(\ ).\ x) \text{ in } \lambda().\ \lambda x.\ v\ (\ )
\end{array}
$$

# From code to closures

$$
\begin{array}{l}
\text{let rec power } n\ c = \\
\quad \text{if } n = 0 \\
\quad \text{then } \langle 1 \rangle \\
\quad \text{else } \langle \sim c \times \sim(\text{power } (n-1)\ c) \rangle \\
\text{let power7} = \langle \lambda x.\ \sim(\text{power } 7\ \langle x \rangle) \rangle
\end{array}
$$

$$\Downarrow$$

$$
\begin{array}{l}
\text{let rec power } n\ c = \\
\quad \text{if } n = 0 \\
\quad \text{then } \lambda(x).\ 1 \\
\quad \text{else let } v = \text{power } (n-1)\ c \text{ in } \lambda(x).\ c\ (x) \times v\ (x) \\
\text{let power7} = \text{let } v = \text{power } 7\ (\lambda(x).\ x) \text{ in } \lambda().\ \lambda x.\ v\ (x)
\end{array}
$$

# From code to closures

$$\begin{array}{l}
\text{let rec power } n\ c = \\
\quad \text{if } n = 0 \\
\quad \text{then } \langle 1 \rangle \\
\quad \text{else } \langle \sim c \times \sim(\text{power } (n-1)\ c) \rangle \\
\text{let power7} = \langle \lambda x.\ \sim(\text{power } 7\ \langle x \rangle) \rangle
\end{array}$$

$$\Downarrow$$

$$\begin{array}{l}
\text{let rec power } n\ c = \\
\quad \text{if } n = 0 \\
\quad \text{then } \lambda(x).\ 1 \\
\quad \text{else let } v = \text{power } (n-1)\ c \text{ in } \lambda(x).\ c\ (x) \times v\ (x) \\
\text{let power7} = \text{let } v = \text{power } 7\ (\lambda(x).\ x) \text{ in } \lambda().\ \lambda x.\ v\ (x)
\end{array}$$

# From code to closures

let rec power $n$ $c$ =
    if $n = 0$
    then $\langle 1 \rangle$
    else $\langle \sim c \times \sim(\text{power } (n-1)\ c) \rangle$
let power7 = $\langle \lambda x.\ \sim(\text{power } 7\ \langle x \rangle) \rangle$
let power7sum = $\langle \lambda x.\ \lambda y.\ \sim(\text{power } 7\ \langle x + y \rangle) \rangle$

$$\Downarrow$$

let rec power $n$ $c$ =
    if $n = 0$
    then $\lambda(x).\ 1$
    else let $v$ = power $(n-1)$ $c$ in $\lambda(x).\ c\ (x) \times v\ (x)$
let power7 = let $v$ = power 7 $(\lambda(x).\ x)$ in $\lambda().\ \lambda x.\ v\ (x)$

# From code to closures

$$
\begin{array}{l}
\text{let rec power } n\ c = \\
\quad \text{if } n = 0 \\
\quad \text{then } \langle 1 \rangle \\
\quad \text{else } \langle \sim c \times \sim(\text{power } (n-1)\ c) \rangle \\
\text{let power7} = \langle \lambda x.\ \sim(\text{power } 7\ \langle x \rangle) \rangle \\
\text{let power7sum} = \langle \lambda x.\ \lambda y.\ \sim(\text{power } 7\ \langle x + y \rangle) \rangle
\end{array}
$$

$$\Downarrow$$

$$
\begin{array}{l}
\text{let rec power } n\ c = \\
\quad \text{if } n = 0 \\
\quad \text{then } \lambda(x).\ 1 \\
\quad \text{else let } v = \text{power } (n-1)\ c \text{ in } \lambda(x).\ c\ (x) \times v\ (x) \\
\text{let power7} = \text{let } v = \text{power } 7\ (\lambda(x).\ x) \text{ in } \lambda().\ \lambda x.\ v\ (x)
\end{array}
$$

# From code to closures

$$
\begin{array}{l}
\text{let rec power } n\ c = \\
\quad \text{if } n = 0 \\
\quad \text{then } \langle 1 \rangle \\
\quad \text{else } \langle \sim c \times \sim(\text{power } (n-1)\ c) \rangle \\
\text{let power7} = \langle \lambda x.\ \sim(\text{power } 7\ \langle x \rangle) \rangle \\
\text{let power7sum} = \langle \lambda x.\ \lambda y.\ \sim(\text{power } 7\ \langle x + y \rangle) \rangle \\
\qquad\qquad\qquad \Downarrow \\
\text{let rec power } n\ c = \\
\quad \text{if } n = 0 \\
\quad \text{then } \lambda\ r\ .\ 1 \\
\quad \text{else let } v = \text{power } (n-1)\ c \text{ in } \lambda\ r\ .\ c\ \ r\ \ \times v\ \ r \\
\text{let power7} = \text{let } v = \text{power } 7\ (\lambda(x).\ x) \text{ in } \lambda().\ \lambda x.\ v\ (x)
\end{array}
$$

# From code to closures

let rec power $n$ $c$ =
    if $n = 0$
    then $\langle 1 \rangle$
    else $\langle \sim c \times \sim(\text{power } (n-1)\ c) \rangle$
let power7 = $\langle \lambda x.\ \sim(\text{power } 7\ \langle x \rangle) \rangle$
let power7sum = $\langle \lambda x.\ \lambda y.\ \sim(\text{power } 7\ \langle x + y \rangle) \rangle$

$$\Downarrow$$

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 $\lambda().\ \lambda x.\ v\ (x, y)$

# From code to closures

$$
\begin{array}{l}
\text{let rec power } n \ c = \\
\quad \text{if } n = 0 \\
\quad \text{then } \langle 1 \rangle \\
\quad \text{else } \langle \sim c \times \sim(\text{power } (n-1) \ c) \rangle \\
\text{let power7} = \langle \lambda x.\ \sim(\text{power } 7 \ \langle x \rangle) \rangle \\
\text{let power7sum} = \langle \lambda x.\ \lambda y.\ \sim(\text{power } 7 \ \langle x + y \rangle) \rangle \\
\text{let eta } f = \langle \lambda x.\ \sim(f \ \langle x \rangle) \rangle
\end{array}
$$

# From higher-order code to higher-rank polymorphism

$$
\begin{array}{l}
\text{let rec power } n\ c = \\
\quad \text{if } n = 0 \\
\quad \text{then } \langle 1 \rangle \\
\quad \text{else } \langle {\sim}c \times {\sim}(\text{power } (n-1)\ c) \rangle \\
\text{let power7} = \langle \lambda x.\ {\sim}(\text{power } 7\ \langle x \rangle) \rangle \\
\text{let power7sum} = \langle \lambda x.\ \lambda y.\ {\sim}(\text{power } 7\ \langle x + y \rangle) \rangle \\
\text{let eta } f = \langle \lambda x.\ {\sim}(f\ \langle x \rangle) \rangle
\end{array}
$$

$$
\begin{array}{l}
\text{let rec power}'\ n\ c = \\
\quad \text{if } n = 0 \\
\quad \text{then } \langle 1 \rangle \\
\quad \text{else if } n \bmod 2 = 0 \\
\qquad \text{then } \langle \text{let } z = {\sim}c \times {\sim}c \text{ in } {\sim}(\text{power}'\ (n \div 2)\ \langle z \rangle) \rangle \\
\qquad \text{else } \langle \text{let } z = {\sim}c \times {\sim}c \text{ in } {\sim}c \times {\sim}(\text{power}'\ ((n-1) \div 2)\ \langle z \rangle) \rangle
\end{array}
$$

# From higher-order code to higher-rank polymorphism

$$
\begin{array}{l}
\text{let rec power } n\ c = \\
\quad \text{if } n = 0 \\
\quad \text{then } \langle 1 \rangle \\
\quad \text{else } \langle \sim c \times \sim(\text{power } (n-1)\ c) \rangle \\
\text{let power7} = \langle \lambda x.\ \sim(\text{power } 7\ \langle x \rangle) \rangle \\
\text{let power7sum} = \langle \lambda x.\ \lambda y.\ \sim(\text{power } 7\ \langle x + y \rangle) \rangle \\
\text{let eta } f = \langle \lambda x.\ \sim(f\ \langle x \rangle) \rangle
\end{array}
$$

$$\langle A \rangle \ \Rightarrow\ \ldots \to A$$

# From higher-order code to higher-rank polymorphism

$$
\begin{array}{l}
\text{let rec } \mathsf{power}\ n\ c = \\
\quad \text{if } n = 0 \\
\quad \text{then } \langle 1 \rangle \\
\quad \text{else } \langle {\sim} c \times {\sim}(\text{power } (n-1)\ c) \rangle \\
\text{let power7} = \langle \lambda x.\ {\sim}(\text{power } 7\ \langle x \rangle) \rangle \\
\text{let power7sum} = \langle \lambda x.\ \lambda y.\ {\sim}(\text{power } 7\ \langle x + y \rangle) \rangle \\
\text{let eta } f = \langle \lambda x.\ {\sim}(f\ \langle x \rangle) \rangle
\end{array}
$$

$$
\begin{array}{rcl}
\langle A \rangle & \Rightarrow & \ldots \to A \\
\langle A \rangle \to \langle B \rangle & \Rightarrow & \forall \pi.\, ((\ldots, \pi) \to A) \to ((\ldots, \pi) \to B)
\end{array}
$$

# From higher-order code to higher-rank polymorphism

```
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⟩)⟩
```

$$
\begin{aligned}
\langle A\rangle &\Rightarrow \ldots \to A \\
\langle A\rangle \to \langle B\rangle &\Rightarrow \forall\pi.\, ((\ldots,\pi) \to A) \to ((\ldots,\pi) \to B) \\
(\langle A\rangle \to \langle B\rangle) \to \langle C\rangle &\Rightarrow \forall\pi.\, (\forall\rho.\, ((\ldots,\pi,\rho) \to A) \\
&\qquad\qquad \to ((\ldots,\pi,\rho) \to B)) \\
&\qquad \to ((\ldots,\pi) \to C)
\end{aligned}
$$

# 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

$$\begin{array}{ll} \cdot & \vdash \\ \Big\downarrow \text{Extend} & \\ x & \vdash \\ \Big\downarrow \text{Extend} & \\ x, z_1 & \vdash \\ \Big\downarrow \text{Extend} & \\ x, z_1, z_2 & \vdash \end{array}$$

$$\begin{array}{l} \text{let rec power}' \; n \; c = \\ \quad \text{if } n = 0 \\ \quad \text{then } \langle 1 \rangle \\ \quad \text{else if } n \bmod 2 = 0 \\ \quad\quad \text{then } \langle \text{let } z = {\sim}c \times {\sim}c \text{ in } {\sim}(\text{power}' \; (n \div \\ \quad\quad \text{else } \langle \text{let } z = {\sim}c \times {\sim}c \text{ in } {\sim}c \times {\sim}(\text{power}' \end{array}$$

# Coercions

$$c = \langle x \rangle$$

$$\Downarrow$$

$$\cdot \quad \vdash$$

$\downarrow$ Extend

$$x \quad \vdash \quad \lambda(x).\, x$$

$\downarrow$ Extend

$$x, z_1 \quad \vdash$$

$\downarrow$ Extend

$$x, z_1, z_2 \quad \vdash$$

$\text{let rec power}'\ n\ c =$
$\quad \text{if } n = 0$
$\quad \text{then } \langle 1 \rangle$
$\quad \text{else if } n \bmod 2 = 0$
$\quad\quad \text{then } \langle \text{let } z = \sim c \times \sim c \text{ in } \sim(\text{power}'\ (n \div$
$\quad\quad \text{else } \langle \text{let } z = \sim c \times \sim c \text{ in } \sim c \times \sim(\text{power}'$

# Coercions

$$
\begin{tikzcd}
 & & c = \langle x \rangle \arrow[d, Rightarrow] \\
\cdot \arrow[d, "\text{Extend}"] & \vdash & {} \\
x \arrow[d, "\text{Extend}"] & \vdash & \lambda(x).\, x \arrow[d, "\text{Coerce}"] \\
{x, z_1} \arrow[d, "\text{Extend}"] & \vdash & {\lambda(x, z_1).\, x} \arrow[d, "\text{Coerce}"] \\
{x, z_1, z_2} & \vdash & {\lambda(x, z_1, z_2).\, x}
\end{tikzcd}
$$

$\mathrm{let\ rec\ power}'\ n\ c =$
$\quad \mathrm{if}\ n = 0$
$\quad \mathrm{then}\ \langle 1 \rangle$
$\quad \mathrm{else\ if}\ n\ \mathrm{mod}\ 2 = 0$
$\qquad \mathrm{then}\ \langle \mathrm{let}\ z = \sim c \times \sim c\ \mathrm{in}\ \sim(\mathrm{power}'\ (n \div$
$\qquad \mathrm{else}\ \langle \mathrm{let}\ z = \sim c \times \sim c\ \mathrm{in}\ \sim c \times \sim(\mathrm{power}'$

# Coercions

$$c = \langle x \rangle \quad \Downarrow \qquad\qquad \mathrm{power}'\ n \quad \Downarrow \quad \lambda c.\,\lambda r.\ldots cr \times cr \ldots$$

$$
\begin{array}{lcl}
\cdot & \vdash & \\
\big\downarrow \text{Extend} & & \\
x & \vdash & \lambda(x).\,x \\
\big\downarrow \text{Extend} & & \big\downarrow \text{Coerce} \\
x, z_1 & \vdash & \lambda(x, z_1).\,x \\
\big\downarrow \text{Extend} & & \big\downarrow \text{Coerce} \\
x, z_1, z_2 & \vdash & \lambda(x, z_1, z_2).\,x
\end{array}
$$

```
let rec power′ n c =
    if n = 0
    then ⟨1⟩
    else if n mod 2 = 0
        then ⟨let z = ~c × ~c in ~(power′ (n ÷
        else ⟨let z = ~c × ~c in ~c × ~(power′
```

# Coercions

$$
\begin{tikzcd}
 & & c = \langle x \rangle \arrow[d, Rightarrow] & \mathrm{power}'\ n \arrow[d, Rightarrow] \\
\cdot \arrow[d, "\text{Extend}"] & \vdash & & \lambda c.\,\lambda r.\dots cr \times cr \dots \arrow[d, "\text{Coerce}"] \\
x \arrow[d, "\text{Extend}"] & \vdash & \lambda(x).\,x \arrow[d, "\text{Coerce}"] & \lambda c.\,\lambda(x,r).\dots c(x,r) \times c(x,r) \dots \arrow[d, "\text{Coerce}"] \\
x, z_1 \arrow[d, "\text{Extend}"] & \vdash & \lambda(x,z_1).\,x \arrow[d, "\text{Coerce}"] & \lambda c.\,\lambda(x,z_1,r).\dots c(x,z_1,r) \times c(x,z_1,r) \dots \arrow[d, "\text{Coerce}"] \\
x, z_1, z_2 & \vdash & \lambda(x,z_1,z_2).\,x & \lambda c.\,\lambda(x,z_1,z_2,r).\dots c(x,z_1,z_2,r) \times c(x,z_1,z_2,r) \dots
\end{tikzcd}
$$

# Coercions

$$
\begin{array}{llll}
 & & \langle \mathtt{int} \rangle & \langle \mathtt{int} \rangle \to \langle \mathtt{int} \rangle \\
 & & \Downarrow & \Downarrow \\
\cdot & \vdash & () \to \mathtt{int} & \forall \pi.\, ((\pi) \to \mathtt{int}) \to ((\pi) \to \mathtt{int}) \\
\big\downarrow \text{Extend} & & \big\downarrow \text{Coerce} & \big\downarrow \text{Coerce} \\
x & \vdash & (\mathtt{int}) \to \mathtt{int} & \forall \pi.\, ((\mathtt{int}, \pi) \to \mathtt{int}) \to ((\mathtt{int}, \pi) \to \mathtt{int}) \\
\big\downarrow \text{Extend} & & \big\downarrow \text{Coerce} & \big\downarrow \text{Coerce} \\
x, z_1 & \vdash & (\mathtt{int}, \mathtt{int}) \to \mathtt{int} & \forall \pi.\, ((\mathtt{int}, \mathtt{int}, \pi) \to \mathtt{int}) \to ((\mathtt{int}, \mathtt{int}, \pi) \to \mathtt{int}) \\
\big\downarrow \text{Extend} & & \big\downarrow \text{Coerce} & \big\downarrow \text{Coerce} \\
x, z_1, z_2 & \vdash & (\mathtt{int}, \mathtt{int}, \mathtt{int}) \to \mathtt{int} & \begin{array}{l} \forall \pi.\, ((\mathtt{int}, \mathtt{int}, \mathtt{int}, \pi) \to \mathtt{int}) \\ \quad \to ((\mathtt{int}, \mathtt{int}, \mathtt{int}, \pi) \to \mathtt{int}) \end{array}
\end{array}
$$

# Coercions

$$\underbrace{\langle \mathtt{int} \rangle \qquad\qquad\qquad\qquad \langle \mathtt{int} \rangle \to \langle \mathtt{int} \rangle}_{\text{Indexed by source type (identity for } \mathtt{int}\text{)}}$$

$$\left.\begin{array}{ll}
\cdot & \vdash \\
\big\downarrow \text{Extend} & \\
x & \vdash \\
\big\downarrow \text{Extend} & \\
x, z_1 & \vdash \\
\big\downarrow \text{Extend} & \\
x, z_1, z_2 & \vdash
\end{array}\right\}$$

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)

$$\langle \mathtt{int} \rangle^{\alpha}$$

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

$$[\,]\,\mathtt{int} \quad [x:\mathtt{int}]\,\mathtt{int} \quad [x:\mathtt{int},\, z_1:\mathtt{int}]\,\mathtt{int} \quad [x:\mathtt{int},\, z_1:\mathtt{int},\, z_2:\mathtt{int}]\,\mathtt{int}$$

# Coercions elaborate environment polymorphism

## In our source language

From environment classifiers (Taha, Nielsen, Calcagno, Moggi)

$$\langle \mathrm{int} \rangle^{\alpha}$$

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

$[]\,\mathrm{int} \qquad [\mathrm{int}]\,\mathrm{int} \qquad [\mathrm{int}, \mathrm{int}]\,\mathrm{int} \qquad [\mathrm{int}, \mathrm{int}, \mathrm{int}]\,\mathrm{int}$

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

# Coercions elaborate environment polymorphism

## In our source language

From environment classifiers (Taha, Nielsen, Calcagno, Moggi)

$$\langle \mathrm{int} \rangle^{\alpha}$$

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

$[\,]\,\mathrm{int} \qquad [\mathrm{int}]\,\mathrm{int} \qquad [\mathrm{int}, \mathrm{int}]\,\mathrm{int} \qquad [\mathrm{int}, \mathrm{int}, \mathrm{int}]\,\mathrm{int}$

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?

# Scope extrusion

How to count multiplications as we generate them?

$$
\begin{array}{l}
\text{let count} = \text{ref } 0 \\
\text{let rec power } n\ c = \\
\quad \text{if } n = 0 \\
\quad \text{then } \langle 1 \rangle \\
\quad \text{else count} \leftarrow \text{!count} + 1; \ldots
\end{array}
$$

# Scope extrusion

How to count multiplications as we generate them?

Use environment may no longer extend creation environment.

$\mathrm{int}$ state is safe: the identity coercion is always available.

$$
\begin{array}{l}
\mathrm{let\ count} = \mathrm{ref}\ 0 \\
\mathrm{let\ rec\ power}\ n\ c = \\
\quad \mathrm{if}\ n = 0 \\
\quad \mathrm{then}\ \langle 1 \rangle \\
\quad \mathrm{else\ count} \leftarrow !\mathrm{count} + 1; \ldots
\end{array}
$$

$\langle\mathrm{int}\rangle$ state risks scope extrusion and running open code.

$$
\begin{array}{lll}
\mathrm{let}\ x = \mathrm{ref}\ \langle 1 \rangle\ \mathrm{in} & & \\
\langle \lambda y.\, {\sim}(x \leftarrow \langle y \rangle; \langle () \rangle) \rangle; & & \\
!x & \rightsquigarrow & \langle y \rangle
\end{array}
$$

# 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