Strictly Capturing Non-strict Closures

Zachary J. Sullivan, Paul Downen, and Zena M. Ariola
University of Oregon

PEPM '21, January 18–19, 2021, Virtual

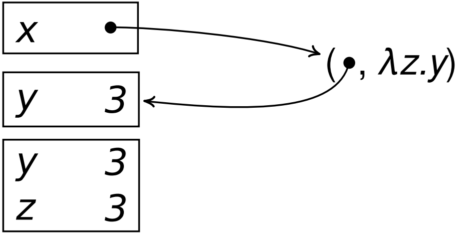
Higher-order Functions

Higher-order Functions

$$\texttt{let}\ x = (\texttt{let}\ y = 2 + 1\ \texttt{in}\ \lambda z.\, y\ )\ \texttt{in}\ (\ x\ 3\ ) + (x\ 4)$$

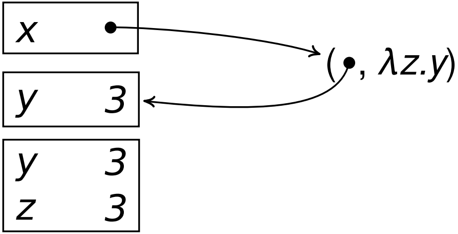
Higher-order Functions

$$\texttt{let}\ x = (\texttt{let}\ y = 2 + 1\ \texttt{in}\ \lambda z.\,y)\ \texttt{in}\ (x\ 3) + (x\ 4)$$



Higher-order Functions

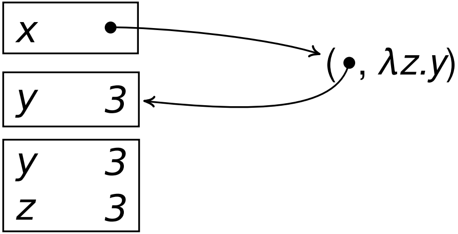
$$\texttt{let}\ x = (\texttt{let}\ y = 2 + 1\ \texttt{in}\ \lambda z.\, y\ )\ \texttt{in}\ (\ x\ 3\ ) + (x\ 4)$$



Pairs of environment and code are **closures**.

Higher-order Functions

$$\texttt{let}\ x = (\texttt{let}\ y = 2 + 1\ \texttt{in}\ \lambda z.\, y\,)\ \texttt{in}\ (\,x\ 3\,) + (x\ 4)$$



Pairs of environment and code are **closures**.
Here, they are a feature of the runtime system.

In Compilation

What if our compiler target language does not automatically create closures?

In Compilation

What if our compiler target language does not automatically create closures?

In Compilation

What if our compiler target language does not automatically create closures?

e.g. C

In Compilation

What if our compiler target language does not automatically create closures?

e.g. C

Solution: make closures explicit in the syntax

Closure-conversion

Closure-conversion

Closure-conversion transforms a language supporting open functions into one that has only closed functions.

Closure-conversion

Closure-conversion transforms a language supporting open functions into one that has only closed functions.

It is used in Scheme's Rabbit and Orbit compilers, and the SML New Jersey compiler.

Closure-conversion

Closure-conversion transforms a language supporting open functions into one that has only closed functions.

It is used in Scheme's Rabbit and Orbit compilers, and the SML New Jersey compiler. *Call-by-value compilers*

Closure-conversion

Closure-conversion transforms a language supporting open functions into one that has only closed functions.

It is used in Scheme's Rabbit and Orbit compilers, and the SML New Jersey compiler. *Call-by-value compilers*

But what about Haskell?

Contributions

Contributions

- Specify non-strict closure-conversions:
  - call-by-name
  - call-by-need

Contributions

- Specify non-strict closure-conversions:
  - call-by-name
  - call-by-need

  Strictness is an essential aspect of **useful** closure-conversion.

Contributions

- Specify non-strict closure-conversions:
  - call-by-name
  - call-by-need

  Strictness is an essential aspect of **useful** closure-conversion.

- We propose *partial closure-conversion*, which allows closures to be introduced locally instead of as a total transformation.

Closures in Strict Languages

Closures in Strict Evaluation

$$\texttt{let}\ x = (\texttt{let}\ y = 2 + 1\ \texttt{in}\ \lambda z.\, y)\ \texttt{in}\ (x\ 3) + (x\ 4)$$

Closures in Strict Evaluation

$$\texttt{let}\ x = (\texttt{let}\ y = 2 + 1\ \texttt{in}\ \lambda z.\,y\,)\ \texttt{in}\ (x\ 3) + (x\ 4)$$

Capturing a closure:

Closures in Strict Evaluation

$$\texttt{let}\ x = (\texttt{let}\ y = 2 + 1\ \texttt{in}\ \lambda z.\, y)\ \texttt{in}\ (x\ 3) + (x\ 4)$$

Capturing a closure:

$$\frac{}{\langle \Sigma \parallel \lambda x.\, M \rangle \Downarrow (\Sigma, \lambda x.\, M)}\ \mathit{Lam}$$

$$\frac{}{\langle \{3/y\} \parallel \lambda z.\, y \rangle \Downarrow (\{3/y\}, \lambda z.\, y)}$$

Closures in Strict Evaluation

$$\texttt{let } x = (\texttt{let } y = 2 + 1 \texttt{ in } \lambda z.\, y) \texttt{ in } (x\ 3) + (x\ 4)$$

Capturing a closure:

$$\frac{}{\langle \Sigma \parallel \lambda x.\, M \rangle \Downarrow (\Sigma, \lambda x.\, M)}\ \mathit{Lam}$$

$$\frac{}{\langle \{3/y\} \parallel \lambda z.\, y \rangle \Downarrow (\{3/y\}, \lambda z.\, y)}$$

Entering a closure:

Closures in Strict Evaluation

$$\texttt{let}\ x = (\texttt{let}\ y = 2 + 1\ \texttt{in}\ \lambda z.\, y)\ \texttt{in}\ (x\ 3) + (x\ 4)$$

Capturing a closure:

$$\frac{}{\langle \Sigma \parallel \lambda x.\, M \rangle \Downarrow (\Sigma, \lambda x.\, M)}\ \mathit{Lam}$$

$$\frac{}{\langle \{3/y\} \parallel \lambda z.\, y \rangle \Downarrow (\{3/y\}, \lambda z.\, y)}$$

Entering a closure:

$$\frac{\langle \Sigma \parallel M \rangle \Downarrow (\Sigma', \lambda x.\, L) \quad \langle \Sigma \parallel N \rangle \Downarrow W \quad \langle \Sigma', W/x \parallel L \rangle \Downarrow V}{\langle \Sigma \parallel M\ N \rangle \Downarrow V}\ \mathit{App}$$

Closures in Strict Evaluation

$$\texttt{let}\ x = (\texttt{let}\ y = 2 + 1\ \texttt{in}\ \lambda z.\, y)\ \texttt{in}\ (x\ 3) + (x\ 4)$$

Capturing a closure:

$$\frac{}{\langle \Sigma \parallel \lambda x.\, M \rangle \Downarrow (\Sigma, \lambda x.\, M)}\ \textit{Lam}$$

$$\frac{}{\langle \{3/y\} \parallel \lambda z.\, y \rangle \Downarrow (\{3/y\}, \lambda z.\, y)}$$

Entering a closure:

$$\frac{\langle \Sigma \parallel M \rangle \Downarrow (\Sigma', \lambda x.\, L) \quad \langle \Sigma \parallel N \rangle \Downarrow W \quad \langle \Sigma', W/x \parallel L \rangle \Downarrow V}{\langle \Sigma \parallel M\ N \rangle \Downarrow V}\ \textit{App}$$

$$\frac{\dfrac{\vdots}{\langle (\{3/y\}, \lambda z.\, y)/x \parallel x \rangle \Downarrow (\{3/y\}, \lambda z.\, y)} \quad \dfrac{\vdots}{\langle \{3/y, 3/z\} \parallel y \rangle \Downarrow 3}}{\langle (\{3/y\}, \lambda z.\, y)/x \parallel x\ 3 \rangle \Downarrow 3}$$

Closures in Strict Evaluation

$$\texttt{let}\ x = (\texttt{let}\ y = 2 + 1\ \texttt{in}\ \lambda z.\, y)\ \texttt{in}\ (x\ 3) + (x\ 4)$$

Capturing a closure:

$$\frac{}{\langle \Sigma \parallel \lambda x.\, M \rangle \Downarrow (\Sigma, \lambda x.\, M)}\ \mathit{Lam}$$

$$\frac{}{\langle \{3/y\} \parallel \lambda z.\, y \rangle \Downarrow (\{3/y\}, \lambda z.\, y)}$$

Entering a closure:

$$\frac{\langle \Sigma \parallel M \rangle \Downarrow (\Sigma', \lambda x.\, L) \quad \langle \Sigma \parallel N \rangle \Downarrow W \quad \langle \Sigma', W/x \parallel L \rangle \Downarrow V}{\langle \Sigma \parallel M\ N \rangle \Downarrow V}\ \mathit{App}$$

$$\frac{\dfrac{\vdots}{\langle (\{3/y\}, \lambda z.\, y)/x \parallel x \rangle \Downarrow (\{3/y\}, \lambda z.\, y)} \quad \dfrac{\vdots}{\langle \{3/y, 3/z\} \parallel y \rangle \Downarrow 3}}{\langle (\{3/y\}, \lambda z.\, y)/x \parallel x\ 3 \rangle \Downarrow 3}$$

Strict Closure-conversion

$$\texttt{let}\ x = (\texttt{let}\ y = 2 + 1\ \texttt{in}\ \lambda z.\, y)\ \texttt{in}\ (x\ 3) + (x\ 4)$$

Strict Closure-conversion

$$\texttt{let}\ x = (\texttt{let}\ y = 2 + 1\ \texttt{in}\ \lambda z.\, y\ )\ \texttt{in}\ (x\ 3)\ + (x\ 4)$$

is closure-converted to the following:

Strict Closure-conversion

$$\texttt{let}\ x = (\texttt{let}\ y = 2 + 1\ \texttt{in}\ \lambda z.\, y)\ \texttt{in}\ (x\ 3) + (x\ 4)$$

is closure-converted to the following:

$$\begin{array}{l}
\texttt{let}\ x = (\texttt{let}\ y = 2 + 1\ \texttt{in}\ \texttt{pack}\ (y, \lambda(y, z).\, y))\\
\texttt{in}\ (\texttt{unpack}\ x\ \texttt{as}\ (e, f)\ \texttt{in}\ f\ (e, 3)) +\\
\quad (\texttt{unpack}\ x\ \texttt{as}\ (e, f)\ \texttt{in}\ f\ (e, 4))
\end{array}$$

Strict Closure-conversion

$$\texttt{let } x = (\texttt{let } y = 2 + 1 \texttt{ in } \lambda z.\, y) \texttt{ in } (x\ 3) + (x\ 4)$$

is closure-converted to the following:

$$\begin{array}{l} \texttt{let } x = (\texttt{let } y = 2 + 1 \texttt{ in } \texttt{pack } (y, \lambda(y, z).\, y)) \\ \texttt{in } (\texttt{unpack } x \texttt{ as } (e, f) \texttt{ in } f\ (e, 3)) + \\ \quad (\texttt{unpack } x \texttt{ as } (e, f) \texttt{ in } f\ (e, 4)) \end{array}$$

How can we run this program?

Semantics of Target Language

$$\begin{array}{l}
\texttt{let}\ x = (\texttt{let}\ y = 2 + 1\ \texttt{in}\ \texttt{pack}\ (y, \lambda(y, z).\, y))\\
\texttt{in}\ (\texttt{unpack}\ x\ \texttt{as}\ (e, f)\ \texttt{in}\ f\ (e, 3)) +\\
\quad (\texttt{unpack}\ x\ \texttt{as}\ (e, f)\ \texttt{in}\ f\ (e, 4))
\end{array}$$

Semantics of Target Language

$$\begin{array}{l}
\texttt{let}\ x = (\texttt{let}\ y = 2 + 1\ \texttt{in}\ \texttt{pack}\ (y, \lambda(y, z).\, y)\,) \\
\texttt{in}\ (\texttt{unpack}\ x\ \texttt{as}\ (e, f)\ \texttt{in}\ f\ (e, 3)) + \\
\quad (\texttt{unpack}\ x\ \texttt{as}\ (e, f)\ \texttt{in}\ f\ (e, 4))
\end{array}$$

Functions do not need to capture free variables:

Semantics of Target Language

$$\begin{array}{l}\texttt{let } x = (\texttt{let } y = 2 + 1 \texttt{ in } \texttt{pack } (y, \lambda(y, z).\, y)) \\ \texttt{in } (\texttt{unpack } x \texttt{ as } (e, f) \texttt{ in } f\ (e, 3)) + \\ \quad (\texttt{unpack } x \texttt{ as } (e, f) \texttt{ in } f\ (e, 4))\end{array}$$

Functions do not need to capture free variables:

$$\frac{}{\langle \Sigma \parallel \lambda x.\, M \rangle \Downarrow \lambda x.\, M}\ \mathit{Lam}'$$

Semantics of Target Language

$$\begin{array}{l}\texttt{let}\ x = (\texttt{let}\ y = 2 + 1\ \texttt{in}\ \texttt{pack}\ (y, \lambda(y, z).\, y)\ ) \\ \texttt{in}\ (\texttt{unpack}\ x\ \texttt{as}\ (e, f)\ \texttt{in}\ f\ (e, 3)) + \\ \quad (\texttt{unpack}\ x\ \texttt{as}\ (e, f)\ \texttt{in}\ f\ (e, 4))\end{array}$$

Functions do not need to capture free variables:

$$\frac{}{\langle \Sigma \parallel \lambda x.\, M \rangle \Downarrow \lambda x.\, M}\ \mathit{Lam}'$$

Applications do not need unpack to them:

Semantics of Target Language

$$\begin{array}{l} \texttt{let } x = (\texttt{let } y = 2+1 \texttt{ in } \texttt{pack } (y, \lambda(y,z).\, y)) \\ \texttt{in } (\texttt{unpack } x \texttt{ as } (e,f) \texttt{ in } f\ (e,3)) + \\ \quad (\texttt{unpack } x \texttt{ as } (e,f) \texttt{ in } f\ (e,4)) \end{array}$$

Functions do not need to capture free variables:

$$\frac{}{\langle \Sigma \parallel \lambda x.\, M\rangle \Downarrow \lambda x.\, M}\ \mathit{Lam}'$$

Applications do not need unpack to them:

$$\frac{\langle \Sigma \parallel M\rangle \Downarrow \lambda x.\, L \qquad \langle \Sigma \parallel N\rangle \Downarrow W \qquad \langle W/x \parallel L\rangle \Downarrow V}{\langle \Sigma \parallel M\ N\rangle \Downarrow V}\ \mathit{App}'$$

Semantics of Target Language

$$\begin{array}{l} \texttt{let}\ x = (\texttt{let}\ y = 2 + 1\ \texttt{in}\ \texttt{pack}\ (y, \lambda(y, z).\, y)\,) \\ \texttt{in}\ (\texttt{unpack}\ x\ \texttt{as}\ (e, f)\ \texttt{in}\ f\ (e, 3)) + \\ \quad (\texttt{unpack}\ x\ \texttt{as}\ (e, f)\ \texttt{in}\ f\ (e, 4)) \end{array}$$

Functions do not need to capture free variables:

$$\frac{}{\langle \Sigma \parallel \lambda x.\, M \rangle \Downarrow \lambda x.\, M}\ \mathit{Lam}'$$

Applications do not need unpack to them:

$$\frac{\langle \Sigma \parallel M \rangle \Downarrow \lambda x.\, L \qquad \langle \Sigma \parallel N \rangle \Downarrow W \qquad \langle\, W / x \parallel L \rangle \Downarrow V}{\langle \Sigma \parallel M\ N \rangle \Downarrow V}\ \mathit{App}'$$

Useful Closure-conversion

Useful Closure-conversion

After closure-conversion, the program does not need a runtime that automatically creates closures.

Closures in Non-strict Languages

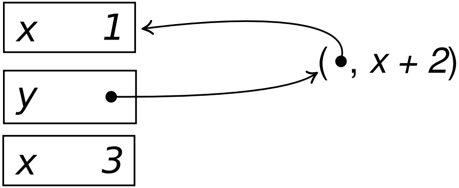
More closures for non-strict languages

More closures for non-strict languages

$$\texttt{let}\ x = 1\ \texttt{in}\ (\texttt{let}\ y = x + 2\ \texttt{in}\ (\texttt{let}\ x = 3\ \texttt{in}\ y))$$

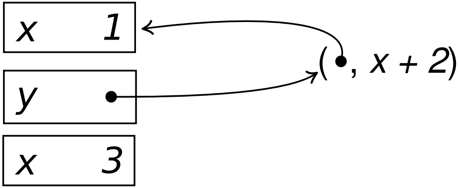
More closures for non-strict languages

$\texttt{let}\ x = 1\ \texttt{in}\ (\texttt{let}\ y = x + 2\ \texttt{in}\ (\texttt{let}\ x = 3\ \texttt{in}\ y))$



More closures for non-strict languages

$\texttt{let}\ x = 1\ \texttt{in}\ (\texttt{let}\ y = x + 2\ \texttt{in}\ (\texttt{let}\ x = 3\ \texttt{in}\ y))$



Non-strict languages create thunk closures *in addition to* function closures.

Closures in Call-by-name Languages

Closures in Call-by-name Evaluation

$$\texttt{let}\ x = 1\ \texttt{in}\ \big(\texttt{let}\ y = x + 2\ \texttt{in}\ \big(\texttt{let}\ x = 3\ \texttt{in}\ y\big)\big)$$

Closures in Call-by-name Evaluation

$$\texttt{let}\ x = 1\ \texttt{in}\ (\texttt{let}\ y = \boxed{x + 2}\ \texttt{in}\ (\texttt{let}\ x = 3\ \texttt{in}\ \boxed{y}))$$

Capturing a thunk closure:

Closures in Call-by-name Evaluation

$$\texttt{let}\ x = 1\ \texttt{in}\ (\texttt{let}\ y = \boxed{x + 2}\ \texttt{in}\ (\texttt{let}\ x = 3\ \texttt{in}\ \boxed{y}))$$

Capturing a thunk closure:

$$\frac{\langle \Sigma, \boxed{(\Sigma, M)/x} \parallel N \rangle \Downarrow R}{\langle \Sigma \parallel \texttt{let}\ x = M\ \texttt{in}\ N \rangle \Downarrow R}\ \mathit{Let}$$

Closures in Call-by-name Evaluation

$$\texttt{let}\ x = 1\ \texttt{in}\ (\texttt{let}\ y = x + 2\ \texttt{in}\ (\texttt{let}\ x = 3\ \texttt{in}\ y))$$

Capturing a thunk closure:

$$\frac{\langle \Sigma,\ (\Sigma, M)/x \parallel N \rangle \Downarrow R}{\langle \Sigma \parallel \texttt{let}\ x = M\ \texttt{in}\ N \rangle \Downarrow R}\ \mathit{Let}$$

$$\cfrac{\cfrac{\vdots}{\langle \ldots/x,\ (\{\ldots/x\},\ x + 2)/y \parallel \texttt{let}\ x = 3\ \texttt{in}\ y \rangle \Downarrow 3}}{\langle \ldots/x \parallel \texttt{let}\ y = x + 2\ \texttt{in}\ (\texttt{let}\ x = 3\ \texttt{in}\ y) \rangle \Downarrow 3}$$

Closures in Call-by-name Evaluation

$$\texttt{let}\ x = 1\ \texttt{in}\ (\texttt{let}\ y = \boxed{x + 2}\ \texttt{in}\ (\texttt{let}\ x = 3\ \texttt{in}\ \boxed{y}))$$

Entering a thunk closure:

Closures in Call-by-name Evaluation

$$\texttt{let}\ x = 1\ \texttt{in}\ (\texttt{let}\ y = x + 2\ \texttt{in}\ (\texttt{let}\ x = 3\ \texttt{in}\ y))$$

Entering a thunk closure:

$$\frac{\Sigma(x) = (\Sigma', M) \quad \langle \Sigma' \parallel M \rangle \Downarrow R}{\langle \Sigma \parallel x \rangle \Downarrow R}\ \mathit{Var}$$

Closures in Call-by-name Evaluation

$$\texttt{let}\ x = 1\ \texttt{in}\ (\texttt{let}\ y = x + 2\ \texttt{in}\ (\texttt{let}\ x = 3\ \texttt{in}\ y))$$

Entering a thunk closure:

$$\frac{\Sigma(x) = (\Sigma', M) \quad \langle \Sigma' \parallel M \rangle \Downarrow R}{\langle \Sigma \parallel x \rangle \Downarrow R}\ \mathit{Var}$$

$$\frac{\dfrac{\vdots}{\langle (\{\}, 1)/x \parallel x + 2 \rangle \Downarrow 3}}{\langle (\{\}, 1)/x, (\{(\{\}, 1)/x\}, x + 2)/y, (\{\ldots/x, \ldots/y\}, 3)/x \parallel y \rangle \Downarrow 3}$$

Call-by-name Closure-conversion

$$\texttt{let } x = 1 \texttt{ in } (\texttt{let } y = x + 2 \texttt{ in } (\texttt{let } x = 3 \texttt{ in } y))$$

Call-by-name Closure-conversion

$$\texttt{let}\ x = 1\ \texttt{in}\ (\texttt{let}\ y = \boxed{x + 2}\ \texttt{in}\ (\texttt{let}\ x = 3\ \texttt{in}\ \boxed{y}))$$

closures converts to:

Call-by-name Closure-conversion

$$\texttt{let } x = 1 \texttt{ in } (\texttt{let } y = \boxed{x + 2} \texttt{ in } (\texttt{let } x = 3 \texttt{ in } \boxed{y}))$$

closures converts to:

$$\begin{array}{l}
\texttt{let } x = \texttt{pack } ((), \lambda().\, 1) \texttt{ in} \\
\quad \texttt{let } y = \boxed{\texttt{pack } (x, \lambda x.\, (\texttt{unpack } x \texttt{ as } (e, f) \texttt{ in } f\ e) + 2)} \texttt{ in} \\
\quad\quad \texttt{let } x = \texttt{pack } ((x, y), \lambda(x, y).\, 3) \texttt{ in} \\
\quad\quad\quad \boxed{\texttt{unpack } y \texttt{ as } (e, f) \texttt{ in } f\ e}
\end{array}$$

Call-by-name Closure-conversion

$$\texttt{let}\ x = 1\ \texttt{in}\ (\texttt{let}\ y = x + 2\ \texttt{in}\ (\texttt{let}\ x = 3\ \texttt{in}\ y))$$

closures converts to:

$$\begin{array}{l}
\texttt{let}\ x = \texttt{pack}\ ((), \lambda().\, 1)\ \texttt{in} \\
\quad \texttt{let}\ y = \texttt{pack}\ (x, \lambda x.\, (\texttt{unpack}\ x\ \texttt{as}\ (e, f)\ \texttt{in}\ f\ e) + 2)\ \texttt{in} \\
\qquad \texttt{let}\ x = \texttt{pack}\ ((x, y), \lambda(x, y).\, 3)\ \texttt{in} \\
\qquad\quad \texttt{unpack}\ y\ \texttt{as}\ (e, f)\ \texttt{in}\ f\ e
\end{array}$$

How can we run this program?

Call-by-name Closure-conversion

$$\texttt{let}\ x = 1\ \texttt{in}\ (\texttt{let}\ y = x + 2\ \texttt{in}\ (\texttt{let}\ x = 3\ \texttt{in}\ y))$$

closures converts to:

$$\begin{array}{l}
\texttt{let}\ x = \texttt{pack}\ ((), \lambda().\,1)\ \texttt{in} \\
\quad \texttt{let}\ y = \texttt{pack}\ (x, \lambda x.\,(\texttt{unpack}\ x\ \texttt{as}\ (e, f)\ \texttt{in}\ f\ \ e) + 2)\ \texttt{in} \\
\quad\quad \texttt{let}\ x = \texttt{pack}\ ((x, y), \lambda(x, y).\,3)\ \texttt{in} \\
\quad\quad\quad \texttt{unpack}\ y\ \texttt{as}\ (e, f)\ \texttt{in}\ f\ \ e
\end{array}$$

How can we run this program?

The natural choice is a call-by-name language with data.

Target Language for Call-by-name Closure-conversion

Target Language for Call-by-name Closure-conversion

Non-strict data types are not evaluated until forced by their context.

Target Language for Call-by-name Closure-conversion

Non-strict data types are not evaluated until forced by their context.

For instance, existential data types:

Target Language for Call-by-name Closure-conversion

Non-strict data types are not evaluated until forced by their context.

For instance, existential data types:

$$\frac{}{\langle \Sigma \parallel \mathtt{pack}\ M \rangle \Downarrow (\Sigma, \mathtt{pack}\ M)}\ \mathit{Pack}$$

Target Language for Call-by-name Closure-conversion

Non-strict data types are not evaluated until forced by their context.

For instance, existential data types:

$$\frac{}{\langle \Sigma \parallel \texttt{pack}\ M \rangle \Downarrow (\Sigma, \texttt{pack}\ M)}\ \textit{Pack}$$

$$\frac{\langle \Sigma \parallel M \rangle \Downarrow (\Sigma', \texttt{pack}\ L) \qquad \langle \Sigma, (\Sigma', L)/x \parallel N \rangle \Downarrow R}{\langle \Sigma \parallel \texttt{unpack}\ M\ \texttt{as}\ x\ \texttt{in}\ N \rangle \Downarrow R}\ \textit{Unpack}$$

Target Language for Call-by-name Closure-conversion

Non-strict data types are not evaluated until forced by their context.

For instance, existential data types:

$$\frac{}{\langle \Sigma \parallel \texttt{pack}\ M \rangle \Downarrow \boxed{(\Sigma, \texttt{pack}\ M)}}\ \textit{Pack}$$

$$\frac{\langle \Sigma \parallel M \rangle \Downarrow \boxed{(\Sigma', \texttt{pack}\ L)} \quad \langle \Sigma, (\Sigma', L)/x \parallel N \rangle \Downarrow R}{\langle \Sigma \parallel \texttt{unpack}\ M\ \texttt{as}\ x\ \texttt{in}\ N \rangle \Downarrow R}\ \textit{Unpack}$$

Non-strict data types do not remove the need for closures in our runtime.

Target Language for Call-by-name Closure-conversion

Non-strict data types are not evaluated until forced by their context.

For instance, existential data types:

$$\frac{}{\langle \Sigma \parallel \texttt{pack}\ M \rangle \Downarrow \colorbox{lightgray}{$(\Sigma, \texttt{pack}\ M)$}}\ \textit{Pack}$$

$$\frac{\langle \Sigma \parallel M \rangle \Downarrow \colorbox{lightgray}{$(\Sigma', \texttt{pack}\ L)$} \quad \langle \Sigma, (\Sigma', L)/x \parallel N \rangle \Downarrow R}{\langle \Sigma \parallel \texttt{unpack}\ M\ \texttt{as}\ x\ \texttt{in}\ N \rangle \Downarrow R}\ \textit{Unpack}$$

Non-strict data types do not remove the need for closures in our runtime.

Neither do non-strict functions, nor let-expressions

Target Language for Call-by-name Closure-conversion

What if we simply remove the closure constructing aspect of non-strict data?

Target Language for Call-by-name Closure-conversion

What if we simply remove the closure constructing aspect of non-strict data?

$$\frac{}{\langle \Sigma \parallel \texttt{pack}\ M \rangle \Downarrow \texttt{pack}\ M}\ \textit{Pack}$$

Target Language for Call-by-name Closure-conversion

What if we simply remove the closure constructing aspect of non-strict data?

$$\frac{}{\langle \Sigma \parallel \texttt{pack}\ M \rangle \Downarrow \boxed{\texttt{pack}\ M}}\ \textit{Pack}$$

$$\frac{\langle \Sigma \parallel M \rangle \Downarrow \boxed{\texttt{pack}\ L} \qquad \langle \Sigma, L/x \parallel N \rangle \Downarrow R}{\langle \Sigma \parallel \texttt{unpack}\ M\ \texttt{as}\ x\ \texttt{in}\ N \rangle \Downarrow R}\ \textit{Unpack}$$

Target Language for Call-by-name Closure-conversion

What if we simply remove the closure constructing aspect of non-strict data?

$$\frac{}{\langle \Sigma \parallel \mathtt{pack}\ M \rangle \Downarrow \mathtt{pack}\ M}\ \textit{Pack}$$

$$\frac{\langle \Sigma \parallel M \rangle \Downarrow \mathtt{pack}\ L \qquad \langle \Sigma, L/x \parallel N \rangle \Downarrow R}{\langle \Sigma \parallel \mathtt{unpack}\ M\ \mathtt{as}\ x\ \mathtt{in}\ N \rangle \Downarrow R}\ \textit{Unpack}$$

$$\frac{\dfrac{\vdots}{\langle \mathtt{pack}\ ((), \lambda().\, 1)/x,\ \mathtt{pack}\ (x, \lambda x.\ \ldots)\ /y \parallel \mathtt{let}\ x = \mathtt{pack}\ ((x,y), \lambda(x,y).\, 3)\ \mathtt{in}\ (\ldots) \rangle \Downarrow 3}}{\langle \mathtt{pack}\ ((), \lambda x.\, 1)/x \parallel \mathtt{let}\ y = \mathtt{pack}\ (x, \lambda x.\ \ldots)\ \mathtt{in}\ (\mathtt{let}\ x = \mathtt{pack}\ ((x,y), \lambda(x,y).\, 3)\ \mathtt{in}\ (\ldots))\rangle \Downarrow 3}$$

Target Language for Call-by-name Closure-conversion

What if we simply remove the closure constructing aspect of non-strict data?

$$\frac{}{\langle \Sigma \parallel \texttt{pack}\ M \rangle \Downarrow \texttt{pack}\ M}\ \textit{Pack}$$

$$\frac{\langle \Sigma \parallel M \rangle \Downarrow \texttt{pack}\ L \qquad \langle \Sigma, L/x \parallel N \rangle \Downarrow R}{\langle \Sigma \parallel \texttt{unpack}\ M\ \texttt{as}\ x\ \texttt{in}\ N \rangle \Downarrow R}\ \textit{Unpack}$$

$$\dfrac{\dfrac{\vdots}{\langle \texttt{pack}\ ((), \lambda().\, 1)/x,\ \texttt{pack}\ (x, \lambda x.\ \ldots)\ /y \parallel \texttt{let}\ x = \texttt{pack}\ ((x, y), \lambda(x, y).\, 3)\ \texttt{in}\ (\ldots)\ \rangle \Downarrow 3}}{\langle \texttt{pack}\ ((), \lambda x.\, 1)/x \parallel \texttt{let}\ y = \texttt{pack}\ (x, \lambda x.\ \ldots)\ \texttt{in}\ (\texttt{let}\ x = \texttt{pack}\ ((x, y), \lambda(x, y).\, 3)\ \texttt{in}\ (\ldots)\ )\rangle \Downarrow 3}$$

Using non-strict data without a closure constructing target language is wrong.

Target Language for Call-by-name Closure-conversion

Target Language for Call-by-name Closure-conversion

We didn't have this problem for call-by-value closure-conversion.

Target Language for Call-by-name Closure-conversion

We didn't have this problem for call-by-value closure-conversion.
Call-by-value data types do not require closures!

Target Language for Call-by-name Closure-conversion

We didn't have this problem for call-by-value closure-conversion.
Call-by-value data types do not require closures!

$$\frac{\langle \Sigma \parallel M \rangle \Downarrow V}{\langle \Sigma \parallel \texttt{pack}\ M \rangle \Downarrow \texttt{pack}\ V}\ \mathit{Pack}$$

Target Language for Call-by-name Closure-conversion

We didn't have this problem for call-by-value closure-conversion.
Call-by-value data types do not require closures!

$$\frac{\langle \Sigma \parallel M \rangle \Downarrow V}{\langle \Sigma \parallel \texttt{pack}\ M \rangle \Downarrow \colorbox{lightgray}{$\texttt{pack}\ V$}}\ \textit{Pack}$$

$$\frac{\langle \Sigma \parallel M \rangle \Downarrow \colorbox{lightgray}{$\texttt{pack}\ V$} \qquad \langle \Sigma, V/x \parallel N \rangle \Downarrow R}{\langle \Sigma \parallel \texttt{unpack}\ M\ \texttt{as}\ x\ \texttt{in}\ N \rangle \Downarrow R}\ \textit{Unpack}$$

Target Language for Call-by-name Closure-conversion

Fortunately, the closure-conversion transformation also performed a thunking transformation.

Target Language for Call-by-name Closure-conversion

Fortunately, the closure-conversion transformation also performed a thunking transformation.

$$
\begin{array}{l}
\texttt{let}\ x = \texttt{pack}\ ((), \lambda().\ 1\ )\ \texttt{in} \\
\quad \texttt{let}\ y = \texttt{pack}\ (x, \lambda x.\ (\texttt{unpack}\ x\ \texttt{as}\ (e, f)\ \texttt{in}\ f\ \ e) + 2\ )\ \texttt{in} \\
\quad\quad \texttt{let}\ x = \texttt{pack}\ ((x, y), \lambda(x, y).\ 3\ )\ \texttt{in} \\
\quad\quad\quad \texttt{unpack}\ y\ \texttt{as}\ (e, f)\ \texttt{in}\ f\ \ e
\end{array}
$$

Target Language for Call-by-name Closure-conversion

Fortunately, the closure-conversion transformation also performed a thunking transformation.

$$\begin{array}{l}
\texttt{let } x = \texttt{pack } ((), \lambda().\ 1\ ) \texttt{ in} \\
\quad \texttt{let } y = \texttt{pack } (x, \lambda x.\ (\texttt{unpack } x \texttt{ as } (e, f) \texttt{ in } f\ e) + 2\ ) \texttt{ in} \\
\quad\quad \texttt{let } x = \texttt{pack } ((x, y), \lambda(x, y).\ 3\ ) \texttt{ in} \\
\quad\quad\quad \texttt{unpack } y \texttt{ as } (e, f) \texttt{ in } f\ e
\end{array}$$

Call-by-name closure-conversion preserves semantics in a *call-by-value* target language.

Target Language for Call-by-name Closure-conversion

Fortunately, the closure-conversion transformation also performed a thunking transformation.

$$
\begin{array}{l}
\texttt{let}\ x = \texttt{pack}\ ((), \lambda().\ 1\ )\ \texttt{in}\\
\quad \texttt{let}\ y = \texttt{pack}\ (x, \lambda x.\ (\texttt{unpack}\ x\ \texttt{as}\ (e, f)\ \texttt{in}\ f\ e) + 2\ )\ \texttt{in}\\
\quad\quad \texttt{let}\ x = \texttt{pack}\ ((x, y), \lambda(x, y).\ 3\ )\ \texttt{in}\\
\quad\quad\quad \texttt{unpack}\ y\ \texttt{as}\ (e, f)\ \texttt{in}\ f\ e
\end{array}
$$

Call-by-name closure-conversion preserves semantics in a *call-by-value* target language.

Target Language for Call-by-name Closure-conversion

Which language do we run our call-by-name closure-converted program?

Target Language for Call-by-name Closure-conversion

Which language do we run our call-by-name closure-converted program?

Runtime	Closure ignorant	Correct	Useful
call-by-name		✓	
call-by-name’	✓		
call-by-value		✓	
call-by-value’	✓	✓	✓

Target Language for Call-by-name Closure-conversion

Which language do we run our call-by-name closure-converted program?

Runtime	Closure ignorant	Correct	Useful
call-by-name		✓	
call-by-name’	✓		
call-by-value		✓	
call-by-value’	✓	✓	✓

The target for call-by-name closure-conversion
should be strict.

Call-by-name Closure-conversion Preserves Types

Call-by-name Closure-conversion Preserves Types

Strict closure-conversion preserves types by hiding environments with existential types (pack expressions).

Call-by-name Closure-conversion Preserves Types

Strict closure-conversion preserves types by hiding environments with existential types (pack expressions).

e.g. $\lambda x.\, x + x$ and $\lambda x.\, x + y$

Call-by-name Closure-conversion Preserves Types

Strict closure-conversion preserves types by hiding environments with existential types (pack expressions).

e.g. $\lambda x.\, x + x$ and $\lambda x.\, x + y$

$$[\![\texttt{int} \to \texttt{int}]\!] = \exists X.\, X \times (X \times \texttt{int} \to \texttt{int})$$

Call-by-name Closure-conversion Preserves Types

Strict closure-conversion preserves types by hiding environments with existential types (pack expressions).

e.g. $\lambda x.\, x + x$ and $\lambda x.\, x + y$

$$[\![\mathtt{int} \to \mathtt{int}]\!] = \exists X.\, X \times (X \times \mathtt{int} \to \mathtt{int})$$

Type preservation for call-by-name requires two type translations:

Call-by-name Closure-conversion Preserves Types

Strict closure-conversion preserves types by hiding environments with existential types (pack expressions).

e.g. $\lambda x.\, x + x$ and $\lambda x.\, x + y$

$$[\![\mathtt{int} \to \mathtt{int}]\!] = \exists X.\, X \times (X \times \mathtt{int} \to \mathtt{int})$$

Type preservation for call-by-name requires two type translations:

For results

Call-by-name Closure-conversion Preserves Types

Strict closure-conversion preserves types by hiding environments with existential types (pack expressions).

e.g. $\lambda x.\, x + x$ and $\lambda x.\, x + y$

$$[\![\mathtt{int} \to \mathtt{int}]\!] = \exists X.\, X \times (X \times \mathtt{int} \to \mathtt{int})$$

Type preservation for call-by-name requires two type translations:

For results

$$\mathsf{Res}[\![\tau \to \tau']\!] = \exists X.\, X \times (X \times \mathsf{Val}[\![\tau]\!] \to \mathsf{Res}[\![\tau']\!])$$

Call-by-name Closure-conversion Preserves Types

Strict closure-conversion preserves types by hiding environments with existential types (pack expressions).

e.g. $\lambda x.\, x + x$ and $\lambda x.\, x + y$

$$[\![\texttt{int} \to \texttt{int}]\!] = \exists X.\, X \times (X \times \texttt{int} \to \texttt{int})$$

Type preservation for call-by-name requires two type translations:

For results

$$\mathsf{Res}[\![\tau \to \tau']\!] = \exists X.\, X \times (X \times \mathsf{Val}[\![\tau]\!] \to \mathsf{Res}[\![\tau']\!])$$

For values, turned into thunk closures

Call-by-name Closure-conversion Preserves Types

Strict closure-conversion preserves types by hiding environments with existential types (pack expressions).

e.g. $\lambda x.\, x + x$ and $\lambda x.\, x + y$

$$[\![\texttt{int} \to \texttt{int}]\!] = \exists X.\, X \times (X \times \texttt{int} \to \texttt{int})$$

Type preservation for call-by-name requires two type translations:

For results

$$\mathsf{Res}[\![\tau \to \tau']\!] = \exists X.\, X \times (X \times \mathsf{Val}[\![\tau]\!] \to \mathsf{Res}[\![\tau']\!])$$

For values, turned into thunk closures

$$\mathsf{Val}[\![\tau]\!] = \exists X.\, X \times (X \to \mathsf{Res}[\![\tau]\!])$$

Call-by-name Closure-conversion Preserves Types

Strict closure-conversion preserves types by hiding environments with existential types (pack expressions).

e.g. $\lambda x.\, x + x$ and $\lambda x.\, x + y$

$$[\![\mathtt{int} \to \mathtt{int}]\!] = \exists X.\, X \times (X \times \mathtt{int} \to \mathtt{int})$$

Type preservation for call-by-name requires two type translations:

For results

$$\mathsf{Res}[\![\tau \to \tau']\!] = \exists X.\, X \times (X \times \mathsf{Val}[\![\tau]\!] \to \mathsf{Res}[\![\tau']\!])$$

For values, turned into thunk closures

$$\mathsf{Val}[\![\tau]\!] = \exists X.\, X \times (X \to \mathsf{Res}[\![\tau]\!])$$

Closures in Call-by-need Languages

Sharing

$$\texttt{let}\ x = (2+1)\ \texttt{in}\ x + x$$

Sharing

$$\texttt{let}\ x = \boxed{(2+1)}\ \texttt{in}\ \boxed{x} + x$$

for which call-by-name closure-conversion yields:

$$\begin{aligned}&\texttt{let}\ x = \boxed{\texttt{pack}\ ((), \lambda().\, 2+1)}\ \texttt{in}\\ &\quad \boxed{(\texttt{unpack}\ x\ \texttt{as}\ (e, f)\ \texttt{in}\ f\ \ e)} + (\texttt{unpack}\ x\ \texttt{as}\ (e, f)\ \texttt{in}\ f\ \ e)\end{aligned}$$

Sharing

$$\texttt{let}\ x = \colorbox{lavender}{$(2+1)$}\ \texttt{in}\ \colorbox{pink}{x} + x$$

for which call-by-name closure-conversion yields:

$$\begin{array}{l}\texttt{let}\ x = \colorbox{lavender}{$\texttt{pack}\ ((), \lambda().\,2+1)$}\ \texttt{in} \\ \quad \colorbox{pink}{$(\texttt{unpack}\ x\ \texttt{as}\ (e, f)\ \texttt{in}\ f\ \ e)$} + (\texttt{unpack}\ x\ \texttt{as}\ (e, f)\ \texttt{in}\ f\ \ e)\end{array}$$

The evaluation of $2+1$ will be performed twice in a call-by-value target language.

Sharing

$$\texttt{let } x = \boxed{(2+1)} \texttt{ in } \boxed{x} + x$$

for which call-by-name closure-conversion yields:

$$\begin{array}{l} \texttt{let } x = \boxed{\texttt{pack } ((), \lambda().\, 2+1)} \texttt{ in} \\ \quad \boxed{(\texttt{unpack } x \texttt{ as } (e, f) \texttt{ in } f\ e)} + (\texttt{unpack } x \texttt{ as } (e, f) \texttt{ in } f\ e) \end{array}$$

The evaluation of $2+1$ will be performed twice in a call-by-value target language.

Sharing has been lost!

Target Language for Call-by-need Closure-conversion

Thunk closures must be updatable with their evaluation result.

Target Language for Call-by-need Closure-conversion

Thunk closures must be updatable with their evaluation result.

Like call-by-value implementations of delay and force, we use:

Target Language for Call-by-need Closure-conversion

Thunk closures must be updatable with their evaluation result.

Like call-by-value implementations of delay and force, we use:

- Mutable references, to store and update

Target Language for Call-by-need Closure-conversion

Thunk closures must be updatable with their evaluation result.

Like call-by-value implementations of delay and force, we use:

- Mutable references, to store and update
- Sum types, to distinguish unevaluated thunks from their evaluation result

Target Language for Call-by-need Closure-conversion

Thunk closures must be updatable with their evaluation result.

Like call-by-value implementations of delay and force, we use:

- Mutable references, to store and update
- Sum types, to distinguish unevaluated thunks from their evaluation result

Call-by-need Closure-conversion

$$\texttt{let}\ x = \colorbox{blue!20}{$(2+1)$}\ \texttt{in}\ \colorbox{red!20}{x} + x$$

Call-by-need Closure-conversion

$$\texttt{let}\ x = (2+1)\ \texttt{in}\ x + x$$

transformed with a call-by-need closure-conversion yields:

$$\begin{aligned}&\texttt{let}\ x = \texttt{store}\ (\texttt{pack}\ ((), \lambda().\, 2+1))\ \texttt{in}\\ &\quad(\texttt{memo}\ x) + (\texttt{memo}\ x)\end{aligned}$$

Call-by-need Closure-conversion

$$\texttt{let } x = \boxed{(2+1)} \texttt{ in } \boxed{x} + x$$

transformed with a call-by-need closure-conversion yields:

$$\begin{array}{l}\texttt{let } x = \boxed{\texttt{store } (\texttt{pack } ((), \lambda(). 2+1))} \texttt{ in} \\ \quad \boxed{(\texttt{memo } x)} + (\texttt{memo } x)\end{array}$$

Where `store` and `memo` are the following macros:

Call-by-need Closure-conversion

$$\texttt{let } x = (2+1) \texttt{ in } x + x$$

transformed with a call-by-need closure-conversion yields:

$$\begin{array}{l} \texttt{let } x = \texttt{store}\ (\texttt{pack}\ ((), \lambda().\,2+1)) \texttt{ in} \\ \quad (\texttt{memo}\ x) + (\texttt{memo}\ x) \end{array}$$

Where `store` and `memo` are the following macros:

$$\begin{array}{rcl} \texttt{store}\ M & \stackrel{\text{def}}{=} & \texttt{new}\ (\texttt{inr}\ M) \\ \texttt{memo}\ x & \stackrel{\text{def}}{=} & \texttt{case}\ !x\ \texttt{of} \\ & & \quad \texttt{inl}\ v \rightarrow v \\ & & \quad \texttt{inr}\ p \rightarrow \\ & & \quad\quad \texttt{unpack}\ p\ \texttt{as}\ (e, f)\ \texttt{in} \\ & & \quad\quad\quad \texttt{let}\ v = f\ e\ \texttt{in} \\ & & \quad\quad\quad\quad \texttt{let}\ \_ = (x := \texttt{inl}\ v)\ \texttt{in}\ v \end{array}$$

Call-by-need Closure-conversion Preserves Types

The preservation argument from call-by-name transformations extends simply, because we use type preserving mutable references.

Call-by-need Closure-conversion Preserves Types

The preservation argument from call-by-name transformations extends simply, because we use type preserving mutable references.

The result translation is unchanged:

Call-by-need Closure-conversion Preserves Types

The preservation argument from call-by-name transformations extends simply, because we use type preserving mutable references.

The result translation is unchanged:

$$\mathsf{Res}[\![\tau \to \tau']\!] = \exists X.\, X \times (X \times \mathsf{Val}[\![\tau]\!] \to \mathsf{Res}[\![\tau']\!])$$

Call-by-need Closure-conversion Preserves Types

The preservation argument from call-by-name transformations extends simply, because we use type preserving mutable references.

The result translation is unchanged:

$$\mathsf{Res}[\![\tau \to \tau']\!] = \exists X.\, X \times (X \times \mathsf{Val}[\![\tau]\!] \to \mathsf{Res}[\![\tau']\!])$$

Values are turned into references to thunk closures or results:

Call-by-need Closure-conversion Preserves Types

The preservation argument from call-by-name transformations extends simply, because we use type preserving mutable references.

The result translation is unchanged:

$$\mathsf{Res}[\![\tau \to \tau']\!] = \exists X.\, X \times (X \times \mathsf{Val}[\![\tau]\!] \to \mathsf{Res}[\![\tau']\!])$$

Values are turned into references to thunk closures or results:

$$\mathsf{Val}[\![\tau]\!] = \texttt{ref}\ (\mathsf{Res}[\![\tau]\!] + (\exists X.\, X \times (X \to \mathsf{Res}[\![\tau]\!])))$$

Call-by-need Closure-conversion Preserves Types

The preservation argument from call-by-name transformations extends simply, because we use type preserving mutable references.

The result translation is unchanged:

$$\mathsf{Res}[\![\tau \to \tau']\!] = \exists X.\, X \times (X \times \mathsf{Val}[\![\tau]\!] \to \mathsf{Res}[\![\tau']\!])$$

Values are turned into references to thunk closures or results:

$$\mathsf{Val}[\![\tau]\!] = \mathtt{ref}\ (\mathsf{Res}[\![\tau]\!] + (\exists X.\, X \times (X \to \mathsf{Res}[\![\tau]\!])))$$

Call-by-need Semantic Preservation

In the paper, call-by-name closure-conversion is proved correct via a family logical relations.

Call-by-need Semantic Preservation

In the paper, call-by-name closure-conversion is proved correct via a family logical relations. *e.g.*

$$\mathrm{Val}[\![\tau]\!] \quad \subseteq \quad \textit{Value} \times \textit{Value}$$

Call-by-need Semantic Preservation

In the paper, call-by-name closure-conversion is proved correct via a family logical relations. *e.g.*

$$\begin{array}{lcl} \mathrm{Val}[\![\tau]\!] & \subseteq & \mathit{Value} \times \mathit{Value} \\ \mathrm{Val}[\![\tau]\!] & \overset{\mathrm{def}}{=} & \{((\Sigma, M), \mathtt{pack}\,(V'_e, V'_f)) \mid (\langle \Sigma \parallel M \rangle, \langle \varepsilon \parallel V'_f\ V'_e \rangle) \in [\![\tau]\!]\} \end{array}$$

Call-by-need Semantic Preservation

In the paper, call-by-name closure-conversion is proved correct via a family logical relations. *e.g.*

$$\begin{array}{lcl} \mathrm{Val}[\![\tau]\!] & \subseteq & \mathit{Value} \times \mathit{Value} \\ \mathrm{Val}[\![\tau]\!] & \stackrel{\mathrm{def}}{=} & \{((\Sigma, M), \mathtt{pack}\,(V'_e, V'_f)) \mid (\langle \Sigma \parallel M \rangle, \langle \varepsilon \parallel V'_f\ V'_e \rangle) \in [\![\tau]\!]\} \end{array}$$

Call-by-need must consider values that are in and depend on a heap Φ:

Call-by-need Semantic Preservation

In the paper, call-by-name closure-conversion is proved correct via a family logical relations. *e.g.*

$$\begin{array}{lcl} \mathrm{Val}[\![\tau]\!] & \subseteq & \mathit{Value} \times \mathit{Value} \\ \mathrm{Val}[\![\tau]\!] & \overset{\mathrm{def}}{=} & \{((\Sigma, M), \mathtt{pack}\,(V'_e, V'_f)) \mid (\langle \Sigma \parallel M \rangle, \langle \varepsilon \parallel V'_f\ V'_e \rangle) \in [\![\tau]\!]\} \end{array}$$

Call-by-need must consider values that are in and depend on a heap Φ:

$$\begin{array}{lcl} \mathrm{Val}[\![\tau]\!](\Phi, \Phi) & \overset{\mathrm{def}}{=} & \{((\Sigma, M), \mathtt{inr}\ (\mathtt{pack}\,(V'_e, V'_f))) \\ & & \quad \mid (\langle \Phi \parallel \Sigma \parallel M \rangle, \langle \Phi \parallel \varepsilon \parallel V'_f\ V'_e \rangle) \in [\![\tau]\!]\} \end{array}$$

Call-by-need Semantic Preservation

In the paper, call-by-name closure-conversion is proved correct via a family logical relations. *e.g.*

$$\begin{array}{lcl} \mathrm{Val}[\![\tau]\!] & \subseteq & \mathit{Value} \times \mathit{Value} \\ \mathrm{Val}[\![\tau]\!] & \overset{\text{def}}{=} & \{((\Sigma, M), \mathtt{pack}\,(V'_e, V'_f)) \mid (\langle \Sigma \parallel M \rangle, \langle \varepsilon \parallel V'_f\ V'_e \rangle) \in [\![\tau]\!]\} \end{array}$$

Call-by-need must consider values that are in and depend on a heap Φ:

$$\begin{array}{lcl} \mathrm{Val}[\![\tau]\!](\Phi, \Phi) & \overset{\text{def}}{=} & \{((\Sigma, M), \mathtt{inr}\ (\mathtt{pack}\,(V'_e, V'_f))) \\ & & \quad \mid (\langle \Phi \parallel \Sigma \parallel M \rangle, \langle \Phi \parallel \varepsilon \parallel V'_f\ V'_e \rangle) \in [\![\tau]\!]\} \\ & & \cup \\ & & \{((\Sigma, \lambda x.\, M), \mathtt{inl}\ V) \\ & & \quad \mid (\langle \Phi \parallel \Sigma \parallel \lambda x.\, M \rangle, \langle \Phi \parallel \varepsilon \parallel V \rangle) \in [\![\tau]\!]\} \end{array}$$

Call-by-need Semantic Preservation

In the paper, call-by-name closure-conversion is proved correct via a family logical relations. *e.g.*

$$\begin{array}{lcl}\mathrm{Val}[\![\tau]\!] & \subseteq & \mathit{Value} \times \mathit{Value} \\ \mathrm{Val}[\![\tau]\!] & \stackrel{\mathrm{def}}{=} & \{((\Sigma, M), \mathtt{pack}\,(V'_e, V'_f)) \mid (\langle \Sigma \parallel M \rangle, \langle \varepsilon \parallel V'_f\ V'_e \rangle) \in [\![\tau]\!]\}\end{array}$$

Call-by-need must consider values that are in and depend on a heap Φ:

$$\begin{array}{lcl}\mathrm{Val}[\![\tau]\!](\Phi, \Phi) & \stackrel{\mathrm{def}}{=} & \{((\Sigma, M), \mathtt{inr}\ (\mathtt{pack}\,(V'_e, V'_f))) \\ & & \quad \mid (\langle \Phi \parallel \Sigma \parallel M \rangle, \langle \Phi \parallel \varepsilon \parallel V'_f\ V'_e \rangle) \in [\![\tau]\!]\} \\ & & \cup \\ & & \{((\Sigma, \lambda x.\, M), \mathtt{inl}\ V) \\ & & \quad \mid (\langle \Phi \parallel \Sigma \parallel \lambda x.\, M \rangle, \langle \Phi \parallel \varepsilon \parallel V \rangle) \in [\![\tau]\!]\}\end{array}$$

Is this sufficient for a call-by-need language?

Call-by-need Semantic Preservation

$$\texttt{let}\ x = (2+1)\ \texttt{in}\ x + x$$

The heap is different between the first and second times that x is accessed.

Call-by-need Semantic Preservation

$$\texttt{let } x = (2+1) \texttt{ in } x + x$$

The heap is different between the first and second times that x is accessed.

For correctness, we **conjecture** that there is a notion of related future heaps $(\Phi, \Phi) \sqsubseteq (\Phi', \Phi')$ such that:

Call-by-need Semantic Preservation

$$\texttt{let}\ x = \boxed{(2+1)}\ \texttt{in}\ \boxed{x} + x$$

The heap is different between the first and second times that x is accessed.

For correctness, we **conjecture** that there is a notion of related future heaps $(\Phi, \Phi) \sqsubseteq (\Phi', \Phi')$ such that:

$$\text{If } (V, V) \in \mathrm{Val}[\![\tau]\!](\Phi, \Phi) \text{ and } (\Phi, \Phi) \sqsubseteq (\Phi', \Phi'),$$
$$\text{then } (V, V) \in \mathrm{Val}[\![\tau]\!](\Phi', \Phi').$$

Call-by-need Semantic Preservation

$$\texttt{let}\ x = \boxed{(2+1)}\ \texttt{in}\ \boxed{x} + x$$

The heap is different between the first and second times that x is accessed.

For correctness, we **conjecture** that there is a notion of related future heaps $(\Phi, \Phi) \sqsubseteq (\Phi', \Phi')$ such that:

$$\text{If } (V, V) \in \mathrm{Val}[\![\tau]\!](\Phi, \Phi) \text{ and } (\Phi, \Phi) \sqsubseteq (\Phi', \Phi'),$$
$$\text{then } (V, V) \in \mathrm{Val}[\![\tau]\!](\Phi', \Phi').$$

Such a notion of future heaps applies to a more general notion of memoization with an explicit heap.

Partial Closure-conversion

Partial Closure-conversion

Useful closure-conversion is done at the end of the compilation pipeline because of the switch to call-by-value.

Partial Closure-conversion

Useful closure-conversion is done at the end of the compilation pipeline because of the switch to call-by-value.

Partial Closure-conversion

Useful closure-conversion is done at the end of the compilation pipeline because of the switch to call-by-value.

We wish to include the lower-level notion of closures in our compiler's intermediate language

Partial Closure-conversion

Useful closure-conversion is done at the end of the compilation pipeline because of the switch to call-by-value.

We wish to include the lower-level notion of closures in our compiler's intermediate language **which is lazy**.

Capturing Closures Locally

$$\texttt{let}\ x = \colorbox{blue!20}{$y + 1$}\ \texttt{in}\ \colorbox{red!20}{x} + x$$

Capturing Closures Locally

$$\texttt{let}\ x = y + 1\ \texttt{in}\ x + x$$

To introduce the closure x into the language, we introduce a strict closure binding $\$x$:

$$\begin{aligned}
&\texttt{let}\ [\$x] = \texttt{pack}\ (y, \lambda[y].\, y + 1)\ \texttt{in} \\
&\quad \texttt{let}\ x = (\texttt{unpack}\ \$x\ \texttt{as}\ (e, f)\ \texttt{in}\ f[e])\ \texttt{in} \\
&\quad\ x + x
\end{aligned}$$

Capturing Closures Locally

$$\texttt{let}\ x = y + 1\ \texttt{in}\ x + x$$

To introduce the closure x into the language, we introduce a strict closure binding $\$x$:

$$\begin{array}{l} \texttt{let}\ [\$x] = \texttt{pack}\ (y, \lambda[y].\,y + 1)\ \texttt{in} \\ \quad \texttt{let}\ x = (\texttt{unpack}\ \$x\ \texttt{as}\ (e, f)\ \texttt{in}\ f[e])\ \texttt{in} \\ \qquad x + x \end{array}$$

The wrapper (*i.e.* $x = \ldots$) is handed off to the lazy runtime.

Capturing Closures Locally

$$\texttt{let}\ x = \boxed{y + 1}\ \texttt{in}\ \boxed{x} + x$$

To introduce the closure x into the language, we introduce a strict closure binding $\$x$:

$$\begin{aligned}
&\texttt{let}\ [\$x] = \boxed{\texttt{pack}\ (y, \lambda[y].\, y + 1)}\ \texttt{in} \\
&\quad \texttt{let}\ x = \boxed{(\texttt{unpack}\ \$x\ \texttt{as}\ (e, f)\ \texttt{in}\ f[e])}\ \texttt{in} \\
&\qquad x + x
\end{aligned}$$

The wrapper (*i.e.* $x = \dots$) is handed off to the lazy runtime.

This is the same idea as how strict unboxed types are introduced, in Haskell compiler's core.

Target Language for Partial Closure-conversion

The intermediate language must include:

Target Language for Partial Closure-conversion

The intermediate language must include:

- closed functions

Target Language for Partial Closure-conversion

The intermediate language must include:

- closed functions
- strict data

Target Language for Partial Closure-conversion

The intermediate language must include:

- closed functions
- strict data
- strict let-expressions

Summary

Summary

- Closure-conversion is not **useful** if we use non-strict data.

Summary

- Closure-conversion is not **useful** if we use non-strict data.
- Establishing the correctness of call-by-need closure-conversion depends on a notion of valid future heaps.

Summary

- Closure-conversion is not **useful** if we use non-strict data.
- Establishing the correctness of call-by-need closure-conversion depends on a notion of valid future heaps.
- Partial closure-conversion allows us to capture closures and still be lazy.

Summary

- Closure-conversion is not **useful** if we use non-strict data.
- Establishing the correctness of call-by-need closure-conversion depends on a notion of valid future heaps.
- Partial closure-conversion allows us to capture closures and still be lazy.

Future work:

Summary

- Closure-conversion is not **useful** if we use non-strict data.
- Establishing the correctness of call-by-need closure-conversion depends on a notion of valid future heaps.
- Partial closure-conversion allows us to capture closures and still be lazy.

Future work:

- Elaborate heap-based reasoning about memoization.

Summary

- Closure-conversion is not **useful** if we use non-strict data.
- Establishing the correctness of call-by-need closure-conversion depends on a notion of valid future heaps.
- Partial closure-conversion allows us to capture closures and still be lazy.

Future work:

- Elaborate heap-based reasoning about memoization.
- Explore practical benefits of partial closure-conversion.

Summary

- Closure-conversion is not **useful** if we use non-strict data.
- Establishing the correctness of call-by-need closure-conversion depends on a notion of valid future heaps.
- Partial closure-conversion allows us to capture closures and still be lazy.

Future work:

- Elaborate heap-based reasoning about memoization.
- Explore practical benefits of partial closure-conversion.

Non-strict closures are strict.