Call-By-Unboxed-Value

Paul Downen

ICFP — Tuesday, September 3, 2023

Unboxing in Practice

- Good: Unboxed values enables high-performance

- Bad: Low-level code clashes with high-level abstractions (e.g., polymorphism)

- Representation irrelevance can resolve the tension (e.g., Levity Polymorphism and Kinds Are Calling Conventions)
  - Restrictions (sometimes surprising) needed for operational meaning & compilability

  - *"If I can't compile it, the type checker must reject it"*

Unboxing in Practice

- Good: Unboxed values enables high-performance

- Bad: Low-level code clashes with high-level abstractions (e.g., polymorphism)

- Representation irrelevance can resolve the tension (e.g., Levity Polymorphism and Kinds Are Calling Conventions)
    - Restrictions (sometimes surprising) needed for operational meaning & compilability

    - *"If I can't compile it, the type checker must reject it"*

- Call-By-Unboxed-Value explains the meaning & implementation of unboxing
    - Logical & semantic foundation ensures meaningful programs

    - *"If I can write it, I can compile & run it"*

COMPILING WITH CALL-BY-UNBOXED-VALUE

- Compiling unboxed polymorphism before:

  - Only compile well-typed source programs; need typing information to generate code

  - Generate ill-typed target programs; compilation can <u>break</u> precise typing

  - *"Types describe the source, kinds describe the machine"*

COMPILING WITH CALL-BY-UNBOXED-VALUE

- Compiling unboxed polymorphism before:
  - Only compile well-typed source programs; need typing information to generate code
  - Generate ill-typed target programs; compilation can <u>break</u> precise typing
  - *"Types describe the source, kinds describe the machine"*

- Compiling unboxed polymorphism with Call-By-Unboxed-Value:
  - Can compile untyped source programs; no typing information needed
  - Compilation preserves typing if the source was well-typed
  - Low-level code can be expressed in a type-safe target language
  - Still support type erasure without changing answers

Unboxed Values

$$sumTo0 \quad :: \mathsf{Int} \to \mathsf{Int}$$
$$sumTo0 \; 0 = 0$$
$$sumTo0 \; n = n + sumTo0(n - 1)$$

Is n an integer register, or a pointer into the heap?

TO AVOID CREATING GARBAGE $\&$ CHASING POINTERS

$$sumTo0 \quad :: \mathsf{Int} \to \mathsf{Int}$$
$$sumTo0\ 0 = 0$$
$$sumTo0\ n = n + sumTo0(n-1)$$

Is n an integer register, or a pointer into the heap?

Accumulator style $\implies$ fast loop

$$sumTo0' \quad :: \mathsf{Int} \to \mathsf{Int}$$
$$sumTo0'\ n = go\ n\ 0$$
$$\textbf{where}\ go\ 0\ acc = acc$$
$$go\ n\ acc = go\ (n-1)\ (n+acc)$$

Problems with Polymorphism

Could the polymorphic *a* really be <u>any</u> type?

$$id \quad :: a \to a$$
$$id \; x = x$$

Problems with Polymorphism

Could the polymorphic *a* really be <u>any</u> type?

$$id \quad :: a \rightarrow a$$
$$id \ x = x$$

Need to know *a*'s representation to generate low-level machine code:

- Where does *x* live? (General or specialized register? Heap?)

- How many bits does *x* occupy? (32? 64? 8?)

- How to copy/move *x* from (incoming) parameter to (outgoing) return?

4

Polymorphic Ambiguity

Do we need to know *a* and *b*'s representations to compile *app*?

$$app \quad :: (a \to b) \to a \to b$$
$$app\ f\ x = f\ x$$

Polymorphic Ambiguity

Do we need to know *a* and *b*'s representations to compile *app*?

$$app \quad :: (a \to b) \to a \to b$$
$$app\, f\, x = f\, x$$

- *a*: Yes, to move *x*
- *b*: It depends...
    - Naïvely yes, to move *f*'s result to $(app\, f\, x)$'s caller
    - But with tail-call optimization, *app* never handles any *b*'s

Polymorphic Ambiguity

Do we need to know *a* and *b*'s representations to compile *app*?

$$app \quad :: (a \to b) \to a \to b$$
$$app\, f\, x = f\, x$$

- *a*: Yes, to move *x*
- *b*: It depends. . .
    - Naïvely yes, to move *f*'s result to $(app\, f\, x)$'s caller
    - But with tail-call optimization, *app* never handles any *b*'s

What about after η-reduction?

$$app' \quad :: (a \to b) \to a \to b$$
$$app'\, f = f$$

Polymorphic Ambiguity

Do we need to know *a* and *b*'s representations to compile *app*?

$$app \quad :: (a \to b) \to a \to b$$
$$app\ f\ x = f\ x$$

- *a*: Yes, to move *x*
- *b*: It depends...
    - Naïvely yes, to move *f*'s result to $(app\ f\ x)$'s caller
    - But with tail-call optimization, *app* never handles any *b*'s

What about after η-reduction?

$$app' \quad :: (a \to b) \to a \to b$$
$$app'\ f = f$$

- *a* and *b*'s representations are irrelevant!
- Only move $f :: a \to b$, always a pointer

5

Higher-Order Ambiguity

What do we need to know about *a* and *b*?

$$
\begin{aligned}
map & :: (a \to b) \to [a] \to [b] \\
map\ f\ [] &= [] \\
map\ f\ (x : xs) &= (f\ x) : (map\ f\ xs)
\end{aligned}
$$

Higher-Order Ambiguity

What do we need to know about *a* and *b*?

$$map \qquad :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]$$
$$map\ f\ [] \qquad = []$$
$$map\ f\ (x : xs) = (f\ x) : (map\ f\ xs)$$

- Representations of both *a* and *b*
    - To move $x :: a$ around
    - To store $(f\ x) :: b$ in a list

- Calling convention of *b*
    - What if $f :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$?
    - $b = \text{Int} \rightarrow \text{Int}$ is a function, needs 1 more argument
    - $(f\ x) :: \text{Int} \rightarrow \text{Int}$ might be a partial application, can't jump to *f*'s body
    - To generate code, need to distinguish partial applications from real calls

6

A First Taste of Call-By-Unboxed-Value

The Two Axes of Unboxing

- Familiar: Values versus computations
    - Values = being
    - Computations = doing

- New: Complexity versus Atomicity
    - Atomic = singular
    - Complex = multiple (parts, choices, …)

Call-By-Unboxed-Value

- Functions are called with complex unboxed values
  - Only atomic values are first class, can be named
  - Complex values are second class, <u>must</u> be matched
- Functions themselves are complex computations
  - Only atomic computations can be run directly
  - Complex computations are inert on their own, <u>must</u> match their context (η-long)

Elaborating Functional Code to Call-By-Unboxed-Value

Haskell

$$sumTo0 \;\; :: \text{Nat} \rightarrow \text{Nat}$$
$$sumTo0 \; 0 = 0$$
$$sumTo0 \; n = n + sumTo0(n - 1)$$

ELABORATING FUNCTIONAL CODE TO CALL-BY-UNBOXED-VALUE

Haskell
$$sumTo0 \quad :: \text{Nat} \to \text{Nat}$$
$$sumTo0 \; 0 = 0$$
$$sumTo0 \; n = n + sumTo0(n - 1)$$

CBPV
$$sumTo0 :: \text{Nat} \to \text{F Nat}$$
$$sumTo0 = \lambda n. \; \textbf{if } n == 0 \textbf{ then return } 0$$

$\qquad\qquad\qquad\qquad \textbf{else do } x \leftarrow n - 1; \qquad (-) :: \text{Nat} \to \text{Nat} \to \text{F Nat}$

$\qquad\qquad\qquad\qquad\qquad \textbf{do } y \leftarrow sumTo0 \; x;$

$\qquad\qquad\qquad\qquad\qquad n + y$

Elaborating Functional Code to Call-By-Unboxed-Value

Haskell

$$sumTo0 \; :: \; \mathsf{Nat} \to \mathsf{Nat}$$
$$sumTo0 \; 0 = 0$$
$$sumTo0 \; n = n + sumTo0(n-1)$$

CBPV

$$sumTo0 :: \mathsf{Nat} \to \mathsf{F} \, \mathsf{Nat}$$
$$sumTo0 = \lambda n. \; \textbf{if} \; n == 0 \; \textbf{then return} \; 0$$
$$\textbf{else do} \; x \leftarrow n - 1; \qquad (-) :: \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{F} \, \mathsf{Nat}$$
$$\textbf{do} \; y \leftarrow sumTo0 \; x;$$
$$n + y$$

CBUV

$$sumTo0 :: \mathsf{Val} \, \mathsf{Nat} \to \mathsf{Eval}(\mathsf{Ret}(\mathsf{Val} \, \mathsf{Nat}))$$
$$sumTo0 = \{\textsf{val int} \; n \cdot \textsf{eval} \to \textbf{if} \; n == 0 \; \textbf{then ret} \; 0$$
$$\textbf{else do} \; \textsf{val int} \; x \leftarrow n - 1;$$
$$\textbf{do} \; \textsf{val int} \; y \leftarrow sumTo0 \; (\textsf{val} \; x) \; . \, \textsf{eval};$$
$$n + y\}$$

9

ELABORATING FUNCTIONAL CODE TO CALL-BY-UNBOXED-VALUE

Haskell
$$sumTo0 \ :: \ \text{Nat} \rightarrow \text{Nat}$$
$$sumTo0 \ 0 = 0$$
$$sumTo0 \ n = n + sumTo0(n - 1)$$

CBPV
$$sumTo0 :: \text{Nat} \rightarrow \textcolor{orange}{\text{F}} \ \text{Nat}$$
$$sumTo0 = \lambda n. \ \textbf{if} \ n == 0 \ \textbf{then } \textbf{return} \ 0$$
$$\textbf{else do } x \leftarrow n - 1; \qquad (-) :: \text{Nat} \rightarrow \text{Nat} \rightarrow \text{F Nat}$$
$$\textbf{do } y \leftarrow sumTo0 \ x;$$
$$n + y$$

CBUV
$$sumTo0 :: \text{Val Nat} \rightarrow \text{Eval}(\textcolor{orange}{\text{Ret}}(\text{Val Nat}))$$
$$sumTo0 = \{\text{val int } n \cdot \text{eval} \rightarrow \textbf{if } n == 0 \ \textbf{then } \textbf{ret} \ 0$$
$$\textbf{else do } \text{val int } x \leftarrow n - 1;$$
$$\textbf{do } \text{val int } y \leftarrow sumTo0 \ (\text{val } x) \ . \ \text{eval};$$
$$n + y\}$$

9

Elaborating Functional Code to Call-By-Unboxed-Value

Haskell
$$sumTo0 \ :: \ \text{Nat} \to \text{Nat}$$
$$sumTo0 \ 0 = 0$$
$$sumTo0 \ n = n + sumTo0(n-1)$$

CBPV
$$sumTo0 :: \text{Nat} \to \text{F Nat}$$
$$sumTo0 = \lambda n. \ \textbf{if } n == 0 \textbf{ then return } 0$$
$$\textbf{else do } x \leftarrow n-1; \qquad (-) :: \text{Nat} \to \text{Nat} \to \text{F Nat}$$
$$\textbf{do } y \leftarrow sumTo0 \ x;$$
$$n+y$$

CBUV
$$sumTo0 :: \text{Val Nat} \to \text{Eval(Ret(Val Nat))}$$
$$sumTo0 = \{ \text{val int } n \cdot \text{eval} \to \textbf{if } n == 0 \textbf{ then ret } 0$$
$$\textbf{else do } \text{val int } x \leftarrow n-1;$$
$$\textbf{do } \text{val int } y \leftarrow sumTo0 \ (\text{val } x) \ . \ \text{eval};$$
$$n+y \}$$

9

ELABORATING FUNCTIONAL CODE TO CALL-BY-UNBOXED-VALUE

Haskell

$$sumTo0 \ :: \ \text{Nat} \to \text{Nat}$$
$$sumTo0 \ 0 = 0$$
$$sumTo0 \ n = n + sumTo0(n-1)$$

CBPV

$$sumTo0 :: \text{Nat} \to \text{F Nat}$$
$$sumTo0 = \lambda n. \ \textbf{if } n == 0 \ \textbf{then return } 0$$
$$\textbf{else do } x \leftarrow n - 1; \qquad (-) :: \text{Nat} \to \text{Nat} \to \text{F Nat}$$
$$\textbf{do } y \leftarrow sumTo0 \ x;$$
$$n + y$$

CBUV

$$sumTo0 :: \text{Val Nat} \to \text{Eval}(\text{Ret}(\text{Val Nat}))$$
$$sumTo0 = \{\text{val int } n \cdot \text{eval} \to \textbf{if } n == 0 \ \textbf{then ret } 0$$
$$\textbf{else do } \text{val int } x \leftarrow n - 1;$$
$$\textbf{do } \text{val int } y \leftarrow sumTo0 \ (\text{val } x) \ . \ \text{eval};$$
$$n + y\}$$

PASSING & RETURNING MULTIPLE ARGUMENTS

$$quotRem : \mathsf{Val\,Nat} \rightarrow \mathsf{Val\,Nat} \rightarrow \mathsf{Eval(Ret(Val\,Nat \times Val\,Nat))}$$

Complex answers must be immediately destructed in place at the call site

OK **do** (val int q, val int r) $\leftarrow$ *quotRem* (val 12) (val 5) . eval

Illegal **do** val ? qr $\leftarrow$ *quotRem* (val 12) (val 5) . eval

Passing & Returning Multiple Arguments

$$quotRem : \mathsf{Val\ Nat} \to \mathsf{Val\ Nat} \to \mathsf{Eval(Ret(Val\ Nat \times Val\ Nat))}$$

Complex answers must be immediately destructed in place at the call site

| OK | **do** (val int q, val int r) ← *quotRem* (val 12) (val 5) . eval |
| Illegal | **do** val ? qr ← *quotRem* (val 12) (val 5) . eval |

$$distance : (\mathsf{Val\ Float}, \mathsf{Val\ Float}) \to \mathsf{Eval(Ret(Val\ Float))}$$

Complex arguments must be immediately constructed in place at the call site

OK	*distance* (val 3.14, val 2.71)
OK	*distance* (val x, val y)
Illegal	*distance* xy
Illegal	*distance* (f x)

10

Polymorphic Code

id : $\forall a$: Type ref **val** . Val $a \rightarrow$ Eval(Ret(Val a))

id = { ty $a \cdot$ val ref(x : a) $\cdot$ eval $\rightarrow$ **ret** val x }

Polymorphic Code

id : $\forall a$: Type ref **val** . Val $a \rightarrow$ Eval(Ret(Val a))

id = { ty $a \cdot$ val ref$(x : a) \cdot$ eval $\rightarrow$ **ret** val x }

$id2$: $\forall a$: Type int **val** . (Val $a \times$ Val Float) $\rightarrow$ Eval(Ret(Val $a \times$ Val Float))

$id2$ = { ty $a \cdot$ (val int$(x : a)$, val flt$(y :$ Float)) $\cdot$ eval $\rightarrow$ **ret** (val x, val y) }

$$id \ : \forall a : \text{Type ref } \textbf{val} \ . \ \text{Val } a \rightarrow \text{Eval}(\text{Ret}(\text{Val } a))$$

$$id \ = \{ \ \text{ty } a \cdot \text{val ref}(x : a) \cdot \text{eval} \rightarrow \textbf{ret} \ \text{val } x \ \}$$

$$id2 : \forall a : \text{Type int } \textbf{val} \ . \ (\text{Val } a \times \text{Val Float}) \rightarrow \text{Eval}(\text{Ret}(\text{Val } a \times \text{Val Float}))$$

$$id2 = \{ \ \text{ty } a \cdot (\text{val int}(x : a), \text{val flt}(y : \text{Float})) \cdot \text{eval} \rightarrow \textbf{ret} \ (\text{val } x, \text{val } y) \ \}$$

id (Val Int $\times$ Val Float) ill-kinded, but id (Box(Val Int $\times$ Val Float)) is OK because

$$\text{Box} : \textbf{cplx val} \rightarrow \text{ref } \textbf{val}$$

Polymorphic Code

id : $\forall a$: Type ref **val** . Val $a \to$ Eval(Ret(Val a))

id = { ty $a \cdot$ val ref(x : a) $\cdot$ eval $\to$ **ret** val x }

$id2$: $\forall a$: Type int **val** . (Val $a \times$ Val Float) $\to$ Eval(Ret(Val $a \times$ Val Float))

$id2$ = { ty $a \cdot$ (val int(x : a), val flt(y : Float)) $\cdot$ eval $\to$ **ret** (val x, val y) }

id (Val Int $\times$ Val Float) ill-kinded, but id (Box(Val Int $\times$ Val Float)) is OK because

$$\text{Box} : \textbf{cplx val} \to \text{ref } \textbf{val}$$

Unboxed code still has well-defined operational meaning after type erasure!

id = { ty $a \cdot$ val ref $x \cdot$ eval $\to$ **ret** val x }

$id2$ = { ty $a \cdot$ (val int x, val flt y) $\cdot$ eval $\to$ **ret** (val x, val y) }

Fusing Values and Calling Conventions

Curried & Uncurried Functions

Unboxed tuples are flattened at compile time (a, b, c : ref **val**; $x : a, y : b, z : c$):

$$(\text{Val } a \times \text{Val } b) \times \text{Val } c \quad \approx \quad \text{Val } a \times (\text{Val } b \times \text{Val } c) \quad \approx \quad \text{Val } a \times \text{Val } b \times \text{Val } c$$
$$((\text{val } x, \text{val } y), \text{val } z) \quad \approx \quad (\text{val } x, (\text{val } y, \text{val } z)) \quad \approx \quad \text{val } x, \text{val } y, \text{val } z$$

CURRIED & UNCURRIED FUNCTIONS

Unboxed tuples are flattened at compile time (a, b, c : ref **val**; $x : a, y : b, z : c$):

$$(\text{Val } a \times \text{Val } b) \times \text{Val } c \quad \approx \quad \text{Val } a \times (\text{Val } b \times \text{Val } c) \quad \approx \quad \text{Val } a \times \text{Val } b \times \text{Val } c$$
$$((\text{val } x, \text{val } y), \text{val } z) \quad \approx \quad (\text{val } x, (\text{val } y, \text{val } z)) \quad \approx \quad \text{val } x, \text{val } y, \text{val } z$$

(Un)Curried functions are compiled to the same code (a, b : ref **val**; c : sub **comp**):

$$f : (\text{Val } a \times \text{Val } b) \to \text{Eval } c \qquad\qquad \approx \qquad g : \text{Val } a \to (\text{Val } b \to \text{Eval } c)$$
$$f = \{ \, (\text{val ref } x, \text{val ref } y) \cdot \text{eval} \to \dots \} \qquad \approx \qquad g = \{ \, \text{val ref } x \cdot (\text{val ref } y \cdot \text{eval}) \to \dots \}$$

CURRIED & UNCURRIED FUNCTIONS

Unboxed tuples are flattened at compile time (a, b, c : ref **val**; $x : a, y : b, z : c$):

$$(\mathsf{Val}\,a \times \mathsf{Val}\,b) \times \mathsf{Val}\,c \quad \approx \quad \mathsf{Val}\,a \times (\mathsf{Val}\,b \times \mathsf{Val}\,c) \quad \approx \quad \mathsf{Val}\,a \times \mathsf{Val}\,b \times \mathsf{Val}\,c$$
$$((\mathsf{val}\,x, \mathsf{val}\,y), \mathsf{val}\,z) \quad \approx \quad (\mathsf{val}\,x, (\mathsf{val}\,y, \mathsf{val}\,z)) \quad \approx \quad \mathsf{val}\,x, \mathsf{val}\,y, \mathsf{val}\,z$$

(Un)Curried functions are compiled to the same code (a, b : ref **val**; c : sub **comp**):

$$f : (\mathsf{Val}\,a \times \mathsf{Val}\,b) \to \mathsf{Eval}\,c \qquad \approx \qquad g : \mathsf{Val}\,a \to (\mathsf{Val}\,b \to \mathsf{Eval}\,c)$$
$$f = \{\,(\mathsf{val}\,\mathsf{ref}\,x, \mathsf{val}\,\mathsf{ref}\,y) \cdot \mathsf{eval} \to \dots\,\} \qquad \approx \qquad g = \{\,\mathsf{val}\,\mathsf{ref}\,x \cdot (\mathsf{val}\,\mathsf{ref}\,y \cdot \mathsf{eval}) \to \dots\,\}$$

Safe due to second-class status of complex values & computations

OK	$f\,(\mathsf{val}\,x, \mathsf{val}\,y)\,.\,\mathsf{eval}$	$\approx$	$g\,(\mathsf{val}\,x)\,(\mathsf{val}\,y)\,.\,\mathsf{eval}$	OK
Illegal	$f\,xy\,.\,\mathsf{eval}$	$\not\approx$	**let** $g' = g\,(\mathsf{val}\,x)$ **in** $\dots$	Illegal

Unboxed Sums

Invariant: all complex patterns can be fully enumerated at compile time

Unboxed sums are also flattened at compile time (a, b, c : ref **val**; $x : a, y : b, z : c$):

$$
\begin{array}{rcll}
(\mathsf{Val}\ a + \mathsf{Val}\ b) + \mathsf{Val}\ c & \approx & \mathsf{Val}\ a + (\mathsf{Val}\ b + \mathsf{Val}\ c) & \\
(0, (0, \mathsf{val}\ x)) & \approx & (0, \mathsf{val}\ x) & \text{Choice \#0} \\
(0, (1, \mathsf{val}\ y)) & \approx & (1, (0, \mathsf{val}\ y)) & \text{Choice \#1} \\
(1, \mathsf{val}\ z) & \approx & (1, (1, \mathsf{val}\ z)) & \text{Choice \#2}
\end{array}
$$

13

Unboxed Sums

Invariant: all complex patterns can be fully enumerated at compile time

Unboxed sums are also flattened at compile time $(a, b, c : \text{ref } \textbf{val}; x : a, y : b, z : c)$:

$$
\begin{array}{rcll}
(\text{Val } a + \text{Val } b) + \text{Val } c & \approx & \text{Val } a + (\text{Val } b + \text{Val } c) & \\
(0, (0, \text{val } x)) & \approx & (0, \text{val } x) & \text{Choice \#0} \\
(0, (1, \text{val } y)) & \approx & (1, (0, \text{val } y)) & \text{Choice \#1} \\
(1, \text{val } z) & \approx & (1, (1, \text{val } z)) & \text{Choice \#2}
\end{array}
$$

Unboxed tuples distribute over unboxed sums $(a, b, c : \text{ref } \textbf{val}; x : a, y : b, z : c)$:

$$
\begin{array}{rcll}
(\text{Val } a + \text{Val } b) \times \text{Val } c & \approx & (\text{Val } a \times \text{Val } c) + (\text{Val } b \times \text{Val } c) & \\
((0, \text{val } x), \text{val } z) & \approx & (0, (\text{val } x, \text{val } z)) & \text{Choice \#0} \\
((1, \text{val } y), \text{val } z) & \approx & (1, (\text{val } y, \text{val } z)) & \text{Choice \#1}
\end{array}
$$

13

$$maybeAdd \text{ Nothing } y = y \qquad\qquad maybeAdd \text{ (Just } x) \text{ } y = x + y$$

Invariant: mandatory pattern-matching on complex values

Choice Fusion

$$maybeAdd\ \text{Nothing}\ y = y \qquad\qquad maybeAdd\ (\text{Just}\ x)\ y = x + y$$

Invariant: mandatory pattern-matching on complex values

Two equivalent versions (Maybe $a = 1 + a$; Nothing $= (0, ())$; Just $x = (1, x)$):

$maybeAdd_1 : (1 + \text{Val Int}) \rightarrow \text{Val Int} \rightarrow \text{Eval}(\text{Ret}(\text{Val Int}))$
$maybeAdd_1 = \{(0, ()) \qquad \cdot \quad (\text{val int } y) \cdot \text{eval} \rightarrow \textbf{ret}\ \text{val } y \qquad$ (Choice #0)
$\qquad\qquad (1, \text{val int } x) \cdot (\text{val int } y) \cdot \text{eval} \rightarrow x + y \qquad$ (Choice #1)$\}$

$maybeAdd_2 : (\text{Val Int} \rightarrow \text{Eval}(\text{Ret}(\text{Val Int}))) \,\&\, (\text{Val Int} \rightarrow \text{Val Int} \rightarrow \text{Eval}(\text{Ret}(\text{Val Int})))$
$maybeAdd_2 = \{0 \cdot \qquad\qquad (\text{val int } y) \cdot \text{eval} \rightarrow \textbf{ret}\ \text{val } y \qquad$ (Choice #0)
$\qquad\qquad 1 \cdot (\text{val int } x) \cdot (\text{val int } y) \cdot \text{eval} \rightarrow x + y \qquad$ (Choice #1)$\}$

14

Choice Fusion

$$maybeAdd \text{ Nothing } y = y \qquad\qquad maybeAdd \text{ (Just } x) \ y = x + y$$

Invariant: mandatory pattern-matching on complex values

Two equivalent versions (Maybe $a = 1 + a$; Nothing $= (0, ())$; Just $x = (1, x)$):

$maybeAdd_1 : (1 + \text{Val Int}) \rightarrow \text{Val Int} \rightarrow \text{Eval}(\text{Ret}(\text{Val Int}))$

$maybeAdd_1 = \{(0, ()) \qquad \cdot ((\text{val int } y) \cdot \text{eval}) \rightarrow \textbf{ret val } y \qquad$ (Choice #0)
$\qquad\qquad\quad (1, \text{val int } x) \cdot ((\text{val int } y) \cdot \text{eval}) \rightarrow x + y \qquad$ (Choice #1)$\}$

$maybeAdd_2 : (\text{Val Int} \rightarrow \text{Eval}(\text{Ret}(\text{Val Int}))) \ \& \ (\text{Val Int} \rightarrow \text{Val Int} \rightarrow \text{Eval}(\text{Ret}(\text{Val Int})))$

$maybeAdd_2 = \{0 \cdot \qquad\qquad ((\text{val int } y) \cdot \text{eval}) \rightarrow \textbf{ret val } y \qquad$ (Choice #0)
$\qquad\qquad\quad 1 \cdot ((\text{val int } x) \cdot (\text{val int } y) \cdot \text{eval}) \rightarrow x + y \qquad$ (Choice #1)$\}$

$maybeAdd_1$ takes a Maybe argument; $maybeAdd_2$ gives a product of 2 functions

14

Stopping Fusion

Putting complex values in a Box pauses pattern-matching.

$$maybeAdd_3 : \mathsf{Val}(\mathsf{Box}(1 + \mathsf{Val\,Int})) \to \mathsf{Val\,Int} \to \mathsf{Eval}(\mathsf{Ret}(\mathsf{Val\,Int}))$$

$$maybeAdd_3 = \{\, \mathsf{val\,ref}\ x \cdot \mathsf{val\,int}\ y \cdot \mathsf{eval} \to$$

$$\mathbf{unbox}\ x\ \mathbf{as}\ \{$$

$$(0, ()) \qquad \to \mathbf{ret}\ \mathsf{val}\ y$$

$$(1, \mathsf{val\,int}\ x) \to x + y$$

$$\}$$

$$\}$$

Stopping Fusion

Putting complex values in a Box pauses pattern-matching.

$$maybeAdd_3 : \mathsf{Val}(\mathsf{Box}(1 + \mathsf{Val\,Int})) \to \mathsf{Val\,Int} \to \mathsf{Eval}(\mathsf{Ret}(\mathsf{Val\,Int}))$$

$$maybeAdd_3 = \{\, \mathsf{val\,ref}\,x \cdot \mathsf{val\,int}\,y \cdot \mathsf{eval} \to$$

$$\mathbf{unbox}\,x\,\mathbf{as}\,\{$$

$$(0, ()) \quad\quad \to \mathbf{ret}\,\mathsf{val}\,y$$

$$(1, \mathsf{val\,int}\,x) \to x + y$$

$$\}$$

$$\}$$

$$maybeAdd_3 \not\approx maybeAdd_1 \quad\quad\quad\quad maybeAdd_3 \not\approx maybeAdd_2$$

Foundations for Unboxing

The Usual Division of Types

Call-By-Push-Value

$$ValueType \ni A ::= A_0 \times A_1 \mid A_0 + A_1 \mid \mathsf{U}\,\underline{B}$$
$$ComputationType \ni \underline{B} ::= A \to \underline{B} \mid \underline{B}_0 \,\&\, \underline{B}_1 \mid \mathsf{F}\,A$$

Focusing & Polarity

$$PositiveType \ni P^+ ::= P_0^+ \otimes P_1^+ \mid P_0^+ \oplus P_1^+ \mid {\downarrow}Q^-$$
$$NegativeType \ni Q^- ::= P^+ \to Q^- \mid Q_0^- \,\&\, Q_1^- \mid {\uparrow}P^+$$

- *Value = Positive*
- *Computation = Negative*

16

The Usual Division of Types

Call-By-Push-Value

$$ValueType \ni A ::= A_0 \times A_1 \mid A_0 + A_1 \mid \mathsf{U}\,\underline{B}$$
$$ComputationType \ni \underline{B} ::= A \to \underline{B} \mid \underline{B}_0\ \&\ \underline{B}_1 \mid \mathsf{F}\,A$$

Focusing & Polarity

$$PositiveType \ni P^+ ::= P_0^+ \otimes P_1^+ \mid P_0^+ \oplus P_1^+ \mid {\downarrow}Q^-$$
$$NegativeType \ni Q^- ::= P^+ \to Q^- \mid Q_0^-\ \&\ Q_1^- \mid {\uparrow}P^+$$

- *Value = Positive*?
- *Computation = Negative*?
- Right?

A Distinction between the Distinctions

- In Call-By-Push-Value, "value types" are the denotable values
  - Only value types are first class, can be named

  - Computation types are second class, cannot be named unless "thunked"

- With strict focusing, pattern matching is mandatory
  - Positive types are second class, must be matched instead of named

  - Negative types are first class, cannot be matched so they are named

- Opposite sides of the complex vs atomic divide:
  - Call-By-Push-Value talks about atomic values and computations

  - Focusing talks about complex values and computations

Shifting Between Quadrants

(*Polymorphic*) *CBPV*   *Focusing*
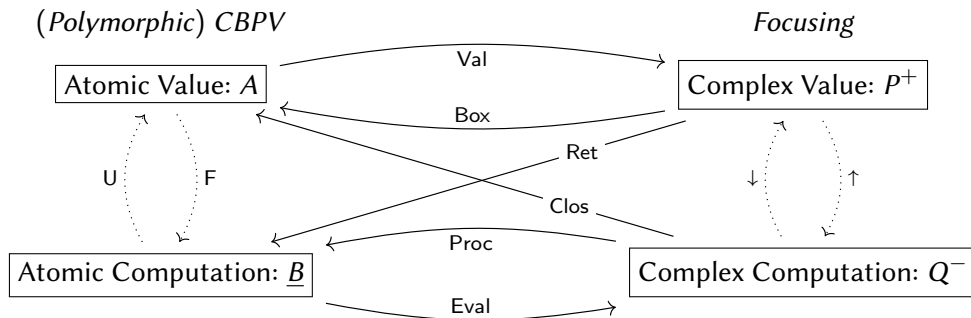
$$\mathsf{F}\,A = \mathsf{Ret}(\mathsf{Val}\,A)$$
$$\mathsf{U}\,\underline{B} = \mathsf{Clos}(\mathsf{Eval}\,\underline{B})$$

$$\uparrow P^+ = \mathsf{Eval}(\mathsf{Ret}\,P^+)$$
$$\downarrow Q^- = \mathsf{Val}(\mathsf{Clos}\,Q^-)$$

Equational theory: Sound & Complete w.r.t. Call-By-Push-Value!

Values = Are   Computations = Do

Atomic = One   Complex = Many

HIGHER-ORDER CALLING CONVENTIONS

- Default "uniform" atomic representations / calling conventions:
  - Atomic value: ref = "reference" (i.e., pointer to value)
  - Atomic computation: sub = "subroutine" (i.e., pointer to top call stack frame)

- First-class closure values built by Clos : **cplx comp** → ref **val**
  - Closure introduced by clos { . . . } around copattern-matching code
  - Closure f : Clos a eliminated with f. call operation

$app = \lambda f\ x.\ (f\ x)$

$app : \forall a : \text{Type ref } \textbf{val} . \forall b : \text{Type sub } \textbf{comp} . \downarrow(\text{Val } a \rightarrow \text{Eval } b) \rightarrow \text{Val } a \rightarrow \text{Eval } b$

$app = \{\text{ty } a \cdot \text{ty } b \cdot \text{val ref } f \cdot \text{val ref } x \cdot \text{eval sub} \rightarrow f. \text{call}(\text{val } x). \text{eval sub}\}$

HIGHER-ORDER CALLING CONVENTIONS

- Default "uniform" atomic representations / calling conventions:
  - Atomic value: ref = "reference" (i.e., pointer to value)
  - Atomic computation: sub = "subroutine" (i.e., pointer to top call stack frame)

- First-class closure values built by Clos : **cplx comp** → ref **val**
  - Closure introduced by clos { ... } around copattern-matching code
  - Closure f : Clos a eliminated with f. call operation

$app = \lambda f\ x.\ (f\ x)$

$app : \forall a :$ Type ref **val** $.\forall b :$ Type sub **comp** . $\downarrow($Val $a \to$ Eval $b) \to$ Val $a \to$ Eval b

$app = \{$ty $a \cdot$ ty $b \cdot$ val ref $f \cdot$ val ref $x \cdot$ eval sub $\to f.$ call(val x). eval sub$\}$

HIGHER-ORDER CALLING CONVENTIONS

- Default "uniform" atomic representations / calling conventions:
  - Atomic value: ref = "reference" (i.e., pointer to value)
  - Atomic computation: sub = "subroutine" (i.e., pointer to top call stack frame)

- First-class closure values built by Clos : **cplx comp** → ref **val**
  - Closure introduced by clos { . . . } around copattern-matching code
  - Closure f : Clos a eliminated with f. call operation

$app = \lambda f\ x.\ (f\ x)$

$app : \forall a : \text{Type ref } \textbf{val} . \forall b : \text{Type sub } \textbf{comp} . \downarrow(\text{Val } a \to \text{Eval } b) \to \text{Val } a \to \text{Eval } b$

$app = \{\text{ty } a \cdot \text{ty } b \cdot \text{val ref } f \cdot \text{val ref } x \cdot \text{eval sub} \to f. \text{call}(\text{val } x). \text{eval sub}\}$

$app' = \lambda f. f$

$app' : \forall a : \text{Type } \textbf{cplx val} . \forall b : \text{Type } \textbf{cplx comp} . \downarrow(a \to b) \to \uparrow\downarrow(a \to b)$

$app' = \{\text{ty } a \cdot \text{ty } b \cdot \text{val ref } f : \text{Clos}(a \to b) \cdot \text{eval sub} \to \textbf{ret } \text{val } f\}$

Higher-Order Calling Conventions

- Default "uniform" atomic representations / calling conventions:
  - Atomic value: ref = "reference" (i.e., pointer to value)
  - Atomic computation: sub = "subroutine" (i.e., pointer to top call stack frame)

- First-class closure values built by Clos : **cplx comp** → ref **val**
  - Closure introduced by clos { . . . } around copattern-matching code
  - Closure f : Clos a eliminated with f. call operation

$app = \lambda f\ x.\ (f\ x)$

$app : \forall a : \text{Type ref } \textbf{val}\ . \forall b : \text{Type sub } \textbf{comp}\ . \downarrow(\text{Val } a \to \text{Eval } b) \to \text{Val } a \to \text{Eval } b$

$app = \{\text{ty } a \cdot \text{ty } b \cdot \text{val ref } f \cdot \text{val ref } x \cdot \text{eval sub} \to f.\,\text{call(val } x).\,\text{eval sub}\}$

$app' = \lambda f.\ f$

$app' : \forall a : \text{Type } \textbf{cplx val}\ . \forall b : \text{Type } \textbf{cplx comp}\ . \downarrow(a \to b) \to \uparrow\downarrow(a \to b)$

$app' = \{\text{ty } a \cdot \text{ty } b \cdot \text{val ref } f : \text{Clos}(a \to b) \cdot \text{eval sub} \to \textbf{ret val } f\}$

Stress Test: Representation-Polymorphic Overloading

Important Application: representation-polymorphic (type class) operator overloading

$$\textbf{class } \text{Num } a \textbf{ where } (+) \quad :: a \rightarrow a \rightarrow a$$
$$negate :: a \rightarrow a$$

What can we do without explicit representation polymorphism?

STRESS TEST: REPRESENTATION-POLYMORPHIC OVERLOADING

Important Application: representation-polymorphic (type class) operator overloading

$$\textbf{class } \mathsf{Num}\ a\ \textbf{where } (+) \quad :: a \to a \to a$$
$$\mathit{negate} :: a \to a$$

What can we do without explicit representation polymorphism?

$$\textbf{type } \mathsf{Num}(a : \textbf{cplx val}) : \textbf{cplx val} = \mathsf{Clos}(a \to a \to \uparrow a) \times \mathsf{Clos}(a \to \uparrow a)$$

Stress Test: Representation-Polymorphic Overloading

Important Application: representation-polymorphic (type class) operator overloading

$$\textbf{class } \mathsf{Num}\ a\ \textbf{where } (+) \quad :: a \to a \to a$$
$$negate :: a \to a$$

What can we do without explicit representation polymorphism?

$$\textbf{type } \mathsf{Num}(a : \textbf{cplx val}) : \textbf{cplx val} = \mathsf{Clos}(a \to a \to \uparrow a) \times \mathsf{Clos}(a \to \uparrow a)$$

$$(+) \quad : \forall a : \mathsf{Type}\ \textbf{cplx val}\ .\ \mathsf{Num}\ a \to \uparrow\downarrow(a \to a \to \uparrow a)$$
$$(+) \quad = \{\ \mathsf{ty}\ a \cdot (\mathsf{val\ ref}\ f, \mathsf{val\ ref}\ g) \cdot \mathsf{eval} \to \textbf{ret}\ \mathsf{val}\ f\ \}$$

$$negate : \forall a : \mathsf{Type}\ \textbf{cplx val}\ .\ \mathsf{Num}\ a \to \uparrow\downarrow(a \to \uparrow a)$$
$$negate = \{\ \mathsf{ty}\ a \cdot (\mathsf{val\ ref}\ f, \mathsf{val\ ref}\ g) \cdot \mathsf{eval} \to \textbf{ret}\ \mathsf{val}\ g\ \}$$

Important Application: representation-polymorphic (type class) operator overloading

$$\textbf{class } \mathsf{Num}\, a \,\textbf{where } (+) \quad :: a \to a \to a$$
$$negate :: a \to a$$

What can we do without explicit representation polymorphism?

$$\textbf{type } \mathsf{Num}(a : \textbf{cplx val}) : \textbf{cplx val} = \mathsf{Clos}(a \to a \to {\uparrow}a) \times \mathsf{Clos}(a \to {\uparrow}a)$$

$$(+) \quad : \forall a : \mathsf{Type}\, \textbf{cplx val} \, . \, \mathsf{Num}\, a \to {\uparrow}{\downarrow}(a \to a \to {\uparrow}a)$$
$$(+) \quad = \{\, \mathsf{ty}\, a \cdot (\mathsf{val\,ref}\, f, \mathsf{val\,ref}\, g) \cdot \mathsf{eval} \to \textbf{ret}\, \mathsf{val}\, f \,\}$$

$$negate : \forall a : \mathsf{Type}\, \textbf{cplx val} \, . \, \mathsf{Num}\, a \to {\uparrow}{\downarrow}(a \to {\uparrow}a)$$
$$negate = \{\, \mathsf{ty}\, a \cdot (\mathsf{val\,ref}\, f, \mathsf{val\,ref}\, g) \cdot \mathsf{eval} \to \textbf{ret}\, \mathsf{val}\, g \,\}$$

STRESS TEST: REPRESENTATION-POLYMORPHIC OVERLOADING

Important Application: representation-polymorphic (type class) operator overloading

$$\textbf{class } \text{Num } a \textbf{ where } (+) \quad :: a \to a \to a$$
$$negate :: a \to a$$

What can we do without explicit representation polymorphism?

After type erasure, still get well-defined, operational code

$$(+) \quad = \{ \text{ty } a \cdot (\text{val ref } f, \text{val ref } g) \cdot \text{eval} \to \textbf{ret } \text{val } f \}$$

$$negate = \{ \text{ty } a \cdot (\text{val ref } f, \text{val ref } g) \cdot \text{eval} \to \textbf{ret } \text{val } g \}$$

COMPILING TO THE MACHINE

Complex patterns $\implies$ 1 simple switch

$x : \text{Box}((\text{Val Int} + \text{Val Float} \times \text{Val Int}) + 1)$

unbox x **as** $\{\ 0, 0, \text{val int } y \qquad\qquad \rightarrow M_1;$
$\qquad\qquad\quad 0, 1, \text{val flt } y, \text{val int } z \rightarrow M_2;$
$\qquad\qquad\quad 1, () \qquad\qquad\qquad\quad \rightarrow M_3\}$

COMPILING TO THE MACHINE

Complex patterns $\implies$ 1 simple switch

$x : \text{Box}((\text{Val Int} + \text{Val Float} \times \text{Val Int}) + 1)$

unbox x **as** $\{\ 0, 0,$ val int $y \qquad\qquad \to M_1;$
$\qquad\qquad 0, 1,$ val flt $y,$ val int $z \to M_2;$
$\qquad\qquad 1, () \qquad\qquad\qquad \to M_3\}$

```c
struct {
  char tag;
  union { // case 0 = 0, 0, val int
    int zero;
         // case 1 = 0, 1, val flt, val int
    struct { float fst; int snd; } one;
         // empty case 2 = 1, ()
  } body;
} *x;
switch (x->tag) {
  case 0:
    int y = x->body.zero; M1...; break;
  case 1:
    float y = x->body.one.fst;
    int z = x->body.one.snd;
    M2...; break;
  case 2:
    M3...
}
```

COMPLEX VARIABLES

$$\textit{and } \mathsf{True}\, x = x \qquad\qquad\qquad \textit{and } \mathsf{False}\, x = \mathsf{False}$$

Complex variables $x \in \{\, \textit{pattern} \ldots \}$ match multiple patterns

$$\mathsf{Bool} = 1 + 1 \qquad\qquad \mathsf{True} = 1, () \qquad\qquad \mathsf{False} = 0, ()$$

$$\textit{and} : \mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Eval}(\mathsf{Ret\,Bool})$$
$$\textit{and} = \{\mathsf{True} \cdot x \in \{\, \mathsf{True}; \mathsf{False} \,\} \cdot \mathsf{eval} \to \mathbf{ret}\, x \in \{\, \mathsf{True}; \mathsf{False} \,\}$$
$$\qquad\quad \mathsf{False} \cdot x \in \{\, \mathsf{True}; \mathsf{False} \,\} \cdot \mathsf{eval} \to \mathbf{ret}\, \mathsf{False} \,\}$$

is syntactic shorthand for

$$\textit{and} : \mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Eval}(\mathsf{Ret\,Bool})$$
$$\textit{and} = \{\mathsf{True} \cdot \mathsf{True} \cdot \mathsf{eval} \to \mathbf{ret}\, \mathsf{True}; \qquad \mathsf{True} \cdot \mathsf{False} \cdot \mathsf{eval} \to \mathbf{ret}\, \mathsf{False};$$
$$\qquad\quad \mathsf{False} \cdot \mathsf{True} \cdot \mathsf{eval} \to \mathbf{ret}\, \mathsf{False}; \qquad \mathsf{False} \cdot \mathsf{False} \cdot \mathsf{eval} \to \mathbf{ret}\, \mathsf{False}; \,\}$$

COMPLEX ANSWERS

Complex continuations $more \in \{\ copattern \ldots \}$ match multiple calling conventions

$app \quad : \forall a : \mathsf{Type\,ref}\ \mathbf{val}\,.\forall b : \mathsf{Type\,sub}\ \mathbf{comp}\,.\downarrow(\mathsf{Val}\,a \to \mathsf{Eval}\,b) \to \mathsf{Val}\,a \to \mathsf{Eval}\,b$

$app \ = \{\mathsf{ty}\,a \cdot \mathsf{ty}\,b \cdot \mathsf{val\,ref}\,f \cdot more \in \{\ \mathsf{val\,ref}\,x \cdot \mathsf{eval\,sub}\ \} \to f.\,\mathsf{call}\}$

$app2 : \forall a, b : \mathsf{Type\,ref}\ \mathbf{val}\,.\forall c : \mathsf{Type\,sub}\ \mathbf{comp}\,.$
$\qquad\qquad \downarrow(\mathsf{Val}\,a \to \mathsf{Val}\,b \to \mathsf{Eval}\,c) \to \mathsf{Val}\,a \to \mathsf{Val}\,b \to \mathsf{Eval}\,c$

$app2 = \{\mathsf{ty}\,a \cdot \mathsf{ty}\,b \cdot \mathsf{ty}\,c \cdot \mathsf{val\,ref}\,f \cdot more \in \{\ \mathsf{val\,ref}\,x \cdot \mathsf{val\,ref}\,y \cdot \mathsf{eval\,sub}\ \} \to f.\,\mathsf{call}\}$

is syntactic shorthand for

$$app \ = \{\mathsf{ty}\,a \cdot \mathsf{ty}\,b \cdot \mathsf{val\,ref}\,f \cdot \mathsf{val\,ref}\,x \cdot \mathsf{eval\,sub} \to f.\,\mathsf{call}(\mathsf{val}\,x).\,\mathsf{eval\,sub}\}$$

$$app2 = \{\mathsf{ty}\,a \cdot \mathsf{ty}\,b \cdot \mathsf{ty}\,c \cdot \mathsf{val\,ref}\,f \cdot \mathsf{val\,ref}\,x \cdot \mathsf{val\,ref}\,y \cdot \mathsf{eval\,sub}$$
$$\to f.\,\mathsf{call}(\mathsf{val}\,x)\,(\mathsf{val}\,y)\,.\,\mathsf{eval\,sub}\}$$