Call-By-Unboxed-Value

Paul Downen

ICFP — Tuesday, September 3, 2023

Unboxing in Practice

The Good, the Bad, and the Ugly

- 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

The Good, the Bad, and the Ugly

- 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 break 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 break 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

Holding Numbers in Registers

To avoid creating garbage & chasing pointers

$$
\begin{aligned}
&\mathit{sumTo0} \quad :: \mathsf{Int} \rightarrow \mathsf{Int} \\
&\mathit{sumTo0}\ 0 = 0 \\
&\mathit{sumTo0}\ n = n + \mathit{sumTo0}(n - 1)
\end{aligned}
$$

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

Holding Numbers in Registers

To avoid creating garbage & chasing pointers

$$
\begin{aligned}
&\mathit{sumTo0} \quad :: \mathsf{Int} \to \mathsf{Int} \\
&\mathit{sumTo0}\ 0 = 0 \\
&\mathit{sumTo0}\ n = n + \mathit{sumTo0}(n-1)
\end{aligned}
$$

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

Accumulator style $\implies$ fast loop

$$
\begin{aligned}
&\mathit{sumTo0}' \quad :: \mathsf{Int} \to \mathsf{Int} \\
&\mathit{sumTo0}'\ n = \mathit{go}\ n\ 0 \\
&\qquad \textbf{where}\ \mathit{go}\ 0\ \mathit{acc} = \mathit{acc} \\
&\qquad\qquad \mathit{go}\ n\ \mathit{acc} = \mathit{go}\ (n-1)\ (n+\mathit{acc})
\end{aligned}
$$

Problems with Polymorphism

Could the polymorphic a really be any type?

$$
\begin{aligned}
&id \quad :: a \rightarrow a \\
&id\ x = x
\end{aligned}
$$

Problems with Polymorphism

What does a compiler need to know to generate code?

Could the polymorphic a really be any type?

$$\begin{aligned} &id \quad :: a \to a \\ &id\ x = x \end{aligned}$$

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?

Polymorphic Ambiguity

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

$$
\begin{aligned}
&app \quad\; :: (a \to b) \to a \to b \\
&app\, f\; x = f\; x
\end{aligned}
$$

Polymorphic Ambiguity

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

$$\begin{aligned} app &:: (a \to b) \to a \to b \\ app\, f\ x &= f\ x \end{aligned}$$

- 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?

$$\begin{aligned} app &:: (a \to b) \to a \to b \\ app\, f\ x &= f\ x \end{aligned}$$

- 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?

$$\begin{aligned} app' &:: (a \to b) \to a \to b \\ app'\ f &= f \end{aligned}$$

Polymorphic Ambiguity

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

$$\begin{aligned} app \quad &:: (a \to b) \to a \to b \\ app\, f\ x &= f\ x \end{aligned}$$

- 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?

$$\begin{aligned} app' \quad &:: (a \to b) \to a \to b \\ app'\ f &= f \end{aligned}$$

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

Higher-Order Ambiguity

When calling unknown functions

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?

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

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

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

Having a name is a privilege, not a right

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

Haskell

$$
\begin{aligned}
&\mathit{sumTo0} \quad :: \mathsf{Nat} \to \mathsf{Nat} \\
&\mathit{sumTo0}\ 0 = 0 \\
&\mathit{sumTo0}\ n = n + \mathit{sumTo0}(n - 1)
\end{aligned}
$$

Haskell

$$\begin{aligned}
&\mathit{sumTo0} :: \mathsf{Nat} \to \mathsf{Nat}\\
&\mathit{sumTo0}\ 0 = 0\\
&\mathit{sumTo0}\ n = n + \mathit{sumTo0}(n - 1)
\end{aligned}$$

CBPV

$$\begin{aligned}
&\mathit{sumTo0} :: \mathsf{Nat} \to \mathsf{F}\,\mathsf{Nat}\\
&\mathit{sumTo0} = \lambda n.\ \mathbf{if}\ n == 0\ \mathbf{then\ return}\ 0\\
&\qquad\qquad\qquad\qquad\ \mathbf{else}\ \mathbf{do}\ x \leftarrow n - 1;\\
&\qquad\qquad\qquad\qquad\qquad\ \mathbf{do}\ y \leftarrow \mathit{sumTo0}\ x;\\
&\qquad\qquad\qquad\qquad\qquad\ n + y
\end{aligned}$$

$(-) :: \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{F}\,\mathsf{Nat}$

Haskell

$$\begin{aligned}
&\mathit{sumTo0} \quad :: \mathsf{Nat} \to \mathsf{Nat}\\
&\mathit{sumTo0}\ 0 = 0\\
&\mathit{sumTo0}\ n = n + \mathit{sumTo0}(n-1)
\end{aligned}$$

CBPV

$$\begin{aligned}
&\mathit{sumTo0} :: \mathsf{Nat} \to \mathsf{F\,Nat}\\
&\mathit{sumTo0} = \lambda n.\ \mathbf{if}\ n == 0\ \mathbf{then\ return}\ 0\\
&\qquad\qquad\qquad\qquad \mathbf{else\ \ do}\ x \leftarrow n - 1;\\
&\qquad\qquad\qquad\qquad\qquad \mathbf{do}\ y \leftarrow \mathit{sumTo0}\ x;\\
&\qquad\qquad\qquad\qquad\qquad n + y
\end{aligned}$$

$(-) :: \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{F\,Nat}$

CBUV

$$\begin{aligned}
&\mathit{sumTo0} :: \mathsf{Val\,Nat} \to \mathsf{Eval}(\mathsf{Ret}(\mathsf{Val\,Nat}))\\
&\mathit{sumTo0} = \{\mathsf{val\,int}\ n \cdot \mathsf{eval} \to \mathbf{if}\ n == 0\ \mathbf{then\ ret}\ 0\\
&\qquad\qquad\qquad\qquad \mathbf{else\ \ do}\ \mathsf{val\,int}\ x \leftarrow n - 1;\\
&\qquad\qquad\qquad\qquad\qquad \mathbf{do}\ \mathsf{val\,int}\ y \leftarrow \mathit{sumTo0}\ (\mathsf{val}\ x)\ .\ \mathsf{eval};\\
&\qquad\qquad\qquad\qquad\qquad n + y\}
\end{aligned}$$

Elaborating Functional Code to Call-By-Unboxed-Value

Haskell

$$\begin{aligned}
&\mathit{sumTo0} \;::\; \mathsf{Nat} \to \mathsf{Nat}\\
&\mathit{sumTo0}\; 0 = 0\\
&\mathit{sumTo0}\; n = n + \mathit{sumTo0}(n-1)
\end{aligned}$$

CBPV

$$\begin{aligned}
&\mathit{sumTo0} :: \mathsf{Nat} \to \mathsf{F}\,\mathsf{Nat}\\
&\mathit{sumTo0} = \lambda n.\;\mathbf{if}\; n == 0 \;\mathbf{then}\;\mathbf{return}\; 0\\
&\qquad\qquad\qquad\qquad\quad \mathbf{else}\;\mathbf{do}\; x \leftarrow n - 1;\\
&\qquad\qquad\qquad\qquad\qquad\quad \mathbf{do}\; y \leftarrow \mathit{sumTo0}\; x;\\
&\qquad\qquad\qquad\qquad\qquad\quad n + y
\end{aligned}$$

$$(-) :: \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{F}\,\mathsf{Nat}$$

CBUV

$$\begin{aligned}
&\mathit{sumTo0} :: \mathsf{Val}\,\mathsf{Nat} \to \mathsf{Eval}(\mathsf{Ret}(\mathsf{Val}\,\mathsf{Nat}))\\
&\mathit{sumTo0} = \{\mathsf{val}\,\mathsf{int}\; n \cdot \mathsf{eval} \to \mathbf{if}\; n == 0\;\mathbf{then}\;\mathbf{ret}\; 0\\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\;\mathbf{do}\;\mathsf{val}\,\mathsf{int}\; x \leftarrow n - 1;\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{do}\;\mathsf{val}\,\mathsf{int}\; y \leftarrow \mathit{sumTo0}\;(\mathsf{val}\; x)\;.\,\mathsf{eval};\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad n + y\}
\end{aligned}$$

Haskell

$$\begin{aligned}
&\mathit{sumTo0} \;::\; \mathsf{Nat} \to \mathsf{Nat}\\
&\mathit{sumTo0}\;0 = 0\\
&\mathit{sumTo0}\;n = n + \mathit{sumTo0}(n-1)
\end{aligned}$$

CBPV

$$\begin{aligned}
&\mathit{sumTo0} :: \mathsf{Nat} \to \mathsf{F}\,\mathsf{Nat}\\
&\mathit{sumTo0} = \lambda n.\ \mathbf{if}\ n == 0\ \mathbf{then\ return}\ 0\\
&\qquad\qquad\qquad\quad \mathbf{else\ do}\ x \leftarrow n - 1;\\
&\qquad\qquad\qquad\qquad\quad \mathbf{do}\ y \leftarrow \mathit{sumTo0}\ x;\\
&\qquad\qquad\qquad\qquad\quad n + y
\end{aligned}$$

$$(-) :: \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{F}\,\mathsf{Nat}$$

CBUV

$$\begin{aligned}
&\mathit{sumTo0} :: \mathsf{Val}\,\mathsf{Nat} \to \mathsf{Eval}(\mathsf{Ret}(\mathsf{Val}\,\mathsf{Nat}))\\
&\mathit{sumTo0} = \{\mathsf{val\ int}\ n \cdot \mathsf{eval} \to \mathbf{if}\ n == 0\ \mathbf{then\ ret}\ 0\\
&\qquad\qquad\qquad\qquad\qquad\quad \mathbf{else\ do}\ \mathsf{val\ int}\ x \leftarrow n - 1;\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{do}\ \mathsf{val\ int}\ y \leftarrow \mathit{sumTo0}\ (\mathsf{val}\ x)\ .\ \mathsf{eval};\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad n + y\}
\end{aligned}$$

Elaborating Functional Code to Call-By-Unboxed-Value

Haskell

$$
\begin{aligned}
&\mathit{sumTo0} \quad :: \mathsf{Nat} \to \mathsf{Nat}\\
&\mathit{sumTo0}\ 0 = 0\\
&\mathit{sumTo0}\ n = n + \mathit{sumTo0}(n-1)
\end{aligned}
$$

CBPV

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

CBUV

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

Passing & Returning Multiple Arguments

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

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

$$\begin{array}{rr} \text{OK} & \textbf{do}\ (\mathsf{val\,int}\ q, \mathsf{val\,int}\ r) \leftarrow quotRem\ (\mathsf{val}\ 12)\ (\mathsf{val}\ 5)\ .\ \mathsf{eval} \\ \text{Illegal} & \textbf{do}\ \mathsf{val}\ ?\ qr \leftarrow quotRem\ (\mathsf{val}\ 12)\ (\mathsf{val}\ 5)\ .\ \mathsf{eval} \end{array}$$

Passing & Returning Multiple Arguments

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

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

OK	$\mathbf{do}\,(\mathsf{val\,int}\ q, \mathsf{val\,int}\ r) \leftarrow \mathit{quotRem}\ (\mathsf{val}\ 12)\ (\mathsf{val}\ 5)\ .\ \mathsf{eval}$
Illegal	$\mathbf{do}\ \mathsf{val}\ ?\ qr \leftarrow \mathit{quotRem}\ (\mathsf{val}\ 12)\ (\mathsf{val}\ 5)\ .\ \mathsf{eval}$

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

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

OK	$\mathit{distance}\ (\mathsf{val}\ 3.14, \mathsf{val}\ 2.71)$
OK	$\mathit{distance}\ (\mathsf{val}\ x, \mathsf{val}\ y)$
Illegal	$\mathit{distance}\ xy$
Illegal	$\mathit{distance}\ (f\ x)$

Polymorphic Code

With type annotations...

Source
$$id \;: \forall a.\ a \to a$$
$$id \;= \Lambda a. \lambda(x : a).\ x$$

CBUV_1
$$id_1 : \forall a : \mathsf{Type}\,\mathsf{ref}\,\mathbf{val}\,.\ \mathsf{Val}\,a \to \mathsf{Eval}(\mathsf{Ret}(\mathsf{Val}\,a))$$
$$id_1 = \{\, \mathsf{ty}\,a \cdot \mathsf{val}\,\mathsf{ref}(x : a) \cdot \mathsf{eval} \to \mathbf{ret}\,\mathsf{val}\,x \,\}$$

Polymorphic Code

With type annotations...

Source $\quad id \;: \forall a.\ a \to a$

$id \;= \Lambda a.\lambda(x : a).\ x$

$\text{CBUV}_1 \quad id_1 : \forall a : \text{Type ref } \mathbf{val}\,.\ \text{Val}\, a \to \text{Eval}(\text{Ret}(\text{Val}\, a))$

$id_1 = \{\, \text{ty}\, a \cdot \text{val ref}(x : a) \cdot \text{eval} \to \mathbf{ret}\, \text{val}\, x \,\}$

$\text{CBUV}_2 \quad id_2 : \forall a : \text{Type int } \mathbf{val}\,.\ (\text{Val}\, a \times \text{Val Float}) \to \text{Eval}(\text{Ret}(\text{Val}\, a \times \text{Val Float}))$

$id_2 = \{\, \text{ty}\, a \cdot (\text{val int}(x : a), \text{val flt}(y : \text{Float})) \cdot \text{eval} \to \mathbf{ret}\, (\text{val}\, x, \text{val}\, y) \,\}$

Polymorphic Code

With type annotations...

Source $\quad id : \forall a.\ a \to a$

$id = \Lambda a.\lambda(x : a).\ x$

$\text{CBUV}_1 \quad id_1 : \forall a : \mathsf{Type\,ref}\ \mathbf{val}\,.\ \mathsf{Val}\,a \to \mathsf{Eval}(\mathsf{Ret}(\mathsf{Val}\,a))$

$id_1 = \{\ \mathsf{ty}\,a \cdot \mathsf{val\,ref}(x : a) \cdot \mathsf{eval} \to \mathbf{ret}\,\mathsf{val}\,x\ \}$

$\text{CBUV}_2 \quad id_2 : \forall a : \mathsf{Type\,int}\ \mathbf{val}\,.\ (\mathsf{Val}\,a \times \mathsf{Val\,Float}) \to \mathsf{Eval}(\mathsf{Ret}(\mathsf{Val}\,a \times \mathsf{Val\,Float}))$

$id_2 = \{\ \mathsf{ty}\,a \cdot (\mathsf{val\,int}(x : a), \mathsf{val\,flt}(y : \mathsf{Float})) \cdot \mathsf{eval} \to \mathbf{ret}\,(\mathsf{val}\,x, \mathsf{val}\,y)\ \}$

id_1 (Val Int $\times$ Val Float) ill-kinded, but id_1 (Box(Val Int $\times$ Val Float)) is OK because

$$\mathsf{Box} : \mathbf{cplx\ val} \to \mathsf{ref}\ \mathbf{val}$$

Polymorphic Code

With type annotations...and without

$$
\begin{aligned}
&\text{Source} && id : \forall a.\ a \to a \\
& && id = \Lambda a.\lambda(x : a).\ x \\[1em]
&\text{CBUV}_1 && id_1 : \forall a : \mathsf{Type}\,\mathsf{ref}\,\mathbf{val}\,.\ \mathsf{Val}\,a \to \mathsf{Eval}(\mathsf{Ret}(\mathsf{Val}\,a)) \\
& && id_1 = \{\,\mathsf{ty}\,a \cdot \mathsf{val}\,\mathsf{ref}(x : a) \cdot \mathsf{eval} \to \mathbf{ret}\,\mathsf{val}\,x\,\} \\[1em]
&\text{CBUV}_2 && id_2 : \forall a : \mathsf{Type}\,\mathsf{int}\,\mathbf{val}\,.\ (\mathsf{Val}\,a \times \mathsf{Val}\,\mathsf{Float}) \to \mathsf{Eval}(\mathsf{Ret}(\mathsf{Val}\,a \times \mathsf{Val}\,\mathsf{Float})) \\
& && id_2 = \{\,\mathsf{ty}\,a \cdot (\mathsf{val}\,\mathsf{int}(x : a), \mathsf{val}\,\mathsf{flt}(y : \mathsf{Float})) \cdot \mathsf{eval} \to \mathbf{ret}\,(\mathsf{val}\,x, \mathsf{val}\,y)\,\}
\end{aligned}
$$

id_1 $(\mathsf{Val}\,\mathsf{Int} \times \mathsf{Val}\,\mathsf{Float})$ ill-kinded, but id_1 $(\mathsf{Box}(\mathsf{Val}\,\mathsf{Int} \times \mathsf{Val}\,\mathsf{Float}))$ is OK because

$$\mathsf{Box} : \mathbf{cplx}\,\mathbf{val} \to \mathsf{ref}\,\mathbf{val}$$

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

$$id_1 = \{\,\mathsf{ty}\,a \cdot\ \mathsf{val}\,\mathsf{ref}\,x \cdot \mathsf{eval} \to \mathbf{ret}\,\mathsf{val}\,x\,\}$$

$$id_2 = \{\,\mathsf{ty}\,a \cdot (\mathsf{val}\,\mathsf{int}\,x, \mathsf{val}\,\mathsf{flt}\,y) \cdot \mathsf{eval} \to \mathbf{ret}\,(\mathsf{val}\,x, \mathsf{val}\,y)\,\}$$

Fusing Values and Calling Conventions

Curried & Uncurried Functions

Nested tuples & call stacks

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

$$
\begin{array}{rcccc}
(\mathsf{Val}\, a \times \mathsf{Val}\, b) \times \mathsf{Val}\, c & \approx & \mathsf{Val}\, a \times (\mathsf{Val}\, b \times \mathsf{Val}\, c) & \approx & \mathsf{Val}\, a \times \mathsf{Val}\, b \times \mathsf{Val}\, c \\
((\mathsf{val}\, x, \mathsf{val}\, y), \mathsf{val}\, z) & \approx & (\mathsf{val}\, x, (\mathsf{val}\, y, \mathsf{val}\, z)) & \approx & \mathsf{val}\, x, \mathsf{val}\, y, \mathsf{val}\, z
\end{array}
$$

Curried & Uncurried Functions

Nested tuples & call stacks

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

$$
\begin{array}{ccccc}
(\mathsf{Val}\, a \times \mathsf{Val}\, b) \times \mathsf{Val}\, c & \approx & \mathsf{Val}\, a \times (\mathsf{Val}\, b \times \mathsf{Val}\, c) & \approx & \mathsf{Val}\, a \times \mathsf{Val}\, b \times \mathsf{Val}\, c \\
((\mathsf{val}\, x, \mathsf{val}\, y), \mathsf{val}\, z) & \approx & (\mathsf{val}\, x, (\mathsf{val}\, y, \mathsf{val}\, z)) & \approx & \mathsf{val}\, x, \mathsf{val}\, y, \mathsf{val}\, z
\end{array}
$$

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

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

Curried & Uncurried Functions

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

$$\begin{array}{rcccr} (\mathsf{Val}\,a \times \mathsf{Val}\,b) \times \mathsf{Val}\,c & \approx & \mathsf{Val}\,a \times (\mathsf{Val}\,b \times \mathsf{Val}\,c) & \approx & \mathsf{Val}\,a \times \mathsf{Val}\,b \times \mathsf{Val}\,c \\ ((\mathsf{val}\,x, \mathsf{val}\,y), \mathsf{val}\,z) & \approx & (\mathsf{val}\,x, (\mathsf{val}\,y, \mathsf{val}\,z)) & \approx & \mathsf{val}\,x, \mathsf{val}\,y, \mathsf{val}\,z \end{array}$$

(Un)Curried functions are compiled to the same code $(a, b : \text{ref } \mathbf{val};\ c : \text{sub } \mathbf{comp})$:

$$\begin{array}{lcl} f : (\mathsf{Val}\,a \times \mathsf{Val}\,b) \to \mathsf{Eval}\,c & \approx & g : \mathsf{Val}\,a \to (\mathsf{Val}\,b \to \mathsf{Eval}\,c) \\ f = \{\, (\mathsf{val}\,\mathsf{ref}\,x, \mathsf{val}\,\mathsf{ref}\,y) \cdot \mathsf{eval} \to \ldots \} & \approx & g = \{\, \mathsf{val}\,\mathsf{ref}\,x \cdot (\mathsf{val}\,\mathsf{ref}\,y \cdot \mathsf{eval}) \to \ldots \} \end{array}$$

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	$f\ xy\ .\,\mathsf{eval}$	$\not\approx$	$h\ (g\ (\mathsf{val}\,x))$	Illegal

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

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

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

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**; $x : a, y : b, z : c$):

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

Unboxed tuples distribute over unboxed sums (a, b, c : ref **val**; $x : a, y : b, z : c$):

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

Choice Fusion

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

Invariant: mandatory pattern-matching on complex values

Choice Fusion

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

Invariant: mandatory pattern-matching on complex values

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

$$\begin{aligned}
&\mathit{maybeAdd}_1 : (1 + \mathsf{Val\,Int}) \to \mathsf{Val\,Int} \to \mathsf{Eval}(\mathsf{Ret}(\mathsf{Val\,Int}))\\
&\mathit{maybeAdd}_1 = \{(0, ()) \cdot (\mathsf{val\,int}\ y) \cdot \mathsf{eval} \to \mathbf{ret}\ \mathsf{val}\ y \quad (\text{Choice \#0})\\
&\qquad (1, \mathsf{val\,int}\ x) \cdot (\mathsf{val\,int}\ y) \cdot \mathsf{eval} \to x + y \quad (\text{Choice \#1})\}\\
&\mathit{maybeAdd}_2 : (\mathsf{Val\,Int} \to \mathsf{Eval}(\mathsf{Ret}(\mathsf{Val\,Int}))) \mathbin{\&} (\mathsf{Val\,Int} \to \mathsf{Val\,Int} \to \mathsf{Eval}(\mathsf{Ret}(\mathsf{Val\,Int})))\\
&\mathit{maybeAdd}_2 = \{0 \cdot (\mathsf{val\,int}\ y) \cdot \mathsf{eval} \to \mathbf{ret}\ \mathsf{val}\ y \quad (\text{Choice \#0})\\
&\qquad 1 \cdot (\mathsf{val\,int}\ x) \cdot (\mathsf{val\,int}\ y) \cdot \mathsf{eval} \to x + y \quad (\text{Choice \#1})\}
\end{aligned}$$

Choice Fusion

Unboxed sum parameters ≈ higher-order products

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

Invariant: mandatory pattern-matching on complex values

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

$$\begin{array}{ll} maybeAdd_1 : (1 + \mathsf{Val\,Int}) \to \mathsf{Val\,Int} \to \mathsf{Eval}(\mathsf{Ret}(\mathsf{Val\,Int})) & \\ maybeAdd_1 = \{(0, ()) \qquad \cdot ((\mathsf{val\,int}\ y) \cdot \mathsf{eval}\,) \to \mathbf{ret}\ \mathsf{val}\ y & (\text{Choice \#0}) \\ \qquad (1, \mathsf{val\,int}\ x) \cdot ((\mathsf{val\,int}\ y) \cdot \mathsf{eval}\,) \to x + y & (\text{Choice \#1})\} \\ maybeAdd_2 : (\mathsf{Val\,Int} \to \mathsf{Eval}(\mathsf{Ret}(\mathsf{Val\,Int}))) \mathbin{\&} (\mathsf{Val\,Int} \to \mathsf{Val\,Int} \to \mathsf{Eval}(\mathsf{Ret}(\mathsf{Val\,Int}))) & \\ maybeAdd_2 = \{0 \cdot \qquad ((\mathsf{val\,int}\ y) \cdot \mathsf{eval}\,) \to \mathbf{ret}\ \mathsf{val}\ y & (\text{Choice \#0}) \\ \qquad 1 \cdot ((\mathsf{val\,int}\ x) \cdot (\mathsf{val\,int}\ y) \cdot \mathsf{eval}\,) \to x + y & (\text{Choice \#1})\} \end{array}$$

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

Stopping Fusion

Putting complex values in a Box pauses pattern-matching.

$$
\begin{aligned}
&\mathit{maybeAdd}_3 : \mathsf{Val}(\mathsf{Box}(1 + \mathsf{Val\,Int})) \to \mathsf{Val\,Int} \to \mathsf{Eval}(\mathsf{Ret}(\mathsf{Val\,Int})) \\
&\mathit{maybeAdd}_3 = \{\, \mathsf{val\,ref}\ x \cdot \mathsf{val\,int}\ y \cdot \mathsf{eval} \to \\
&\qquad\qquad \textbf{unbox}\ x\ \textbf{as}\ \{ \\
&\qquad\qquad\qquad (0, ()) \qquad\quad \to \textbf{ret}\ \mathsf{val}\ y \\
&\qquad\qquad\qquad (1, \mathsf{val\,int}\ x) \to x + y \\
&\qquad\qquad \} \\
&\qquad \}
\end{aligned}
$$

Stopping Fusion

Putting complex values in a Box pauses pattern-matching.

$$
\begin{aligned}
&\mathit{maybeAdd}_3 : \mathsf{Val}(\mathsf{Box}(1 + \mathsf{Val}\,\mathsf{Int})) \to \mathsf{Val}\,\mathsf{Int} \to \mathsf{Eval}(\mathsf{Ret}(\mathsf{Val}\,\mathsf{Int}))\\
&\mathit{maybeAdd}_3 = \{\, \mathsf{val}\,\mathsf{ref}\; x \cdot \mathsf{val}\,\mathsf{int}\; y \cdot \mathsf{eval} \to\\
&\qquad\qquad \mathbf{unbox}\; x\; \mathbf{as}\, \{\\
&\qquad\qquad\qquad (0, ()) \quad\quad\;\; \to \mathbf{ret}\,\mathsf{val}\; y\\
&\qquad\qquad\qquad (1, \mathsf{val}\,\mathsf{int}\; x) \to x + y\\
&\qquad\qquad \}\\
&\qquad \}
\end{aligned}
$$

$$\mathit{maybeAdd}_3 \not\approx \mathit{maybeAdd}_1 \qquad\qquad \mathit{maybeAdd}_3 \not\approx \mathit{maybeAdd}_2$$

Foundations for Unboxing

The Usual Division of Types

A commonly-repeated refrain

Call-By-Push-Value

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

Focusing & Polarity

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

- $\mathit{Value} = \mathit{Positive}$
- $\mathit{Computation} = \mathit{Negative}$

The Usual Division of Types

A commonly-repeated refrain

Call-By-Push-Value

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

Focusing & Polarity

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

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

- 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*

$$
\begin{tikzcd}
\boxed{\text{Atomic Value: } A} \arrow[r, bend left, "\mathsf{Val}"] \arrow[d, dotted, bend left, "\mathsf{F}"] & \boxed{\text{Complex Value: } P^+} \arrow[l, bend left, "\mathsf{Box}"] \arrow[dl, "\mathsf{Ret}"] \arrow[d, dotted, bend left, "\uparrow"] \\
\boxed{\text{Atomic Computation: } \underline{B}} \arrow[u, dotted, bend left, "\mathsf{U}"] \arrow[r, bend right, "\mathsf{Eval}"'] & \boxed{\text{Complex Computation: } Q^-} \arrow[ul, "\mathsf{Clos}"] \arrow[l, bend right, "\mathsf{Proc}"'] \arrow[u, dotted, bend left, "\downarrow"]
\end{tikzcd}
$$

$$\mathsf{F}\,A = \mathsf{Ret}(\mathsf{Val}\,A) \qquad {\uparrow}P^+ = \mathsf{Eval}(\mathsf{Ret}\,P^+)$$

$$\mathsf{U}\,\underline{B} = \mathsf{Clos}(\mathsf{Eval}\,\underline{B}) \qquad {\downarrow}Q^- = \mathsf{Val}(\mathsf{Clos}\,Q^-)$$

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

$\text{Values} = \text{Are}$ $\text{Computations} = \text{Do}$

$\text{Atomic} = \text{One}$ $\text{Complex} = \text{Many}$

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 $\mathsf{Clos} : \mathbf{cplx}\ \mathbf{comp} \to \mathsf{ref}\ \mathbf{val}$
    - Closure introduced by $\mathsf{clos}\,\{\ldots\}$ around copattern-matching code
    - Closure $f : \mathsf{Clos}\,a$ eliminated with $f.\mathsf{call}$ operation

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

$$app : \forall a : \mathsf{Type\,ref}\,\mathbf{val}\,.\forall b : \mathsf{Type\,sub}\,\mathbf{comp}\,.\downarrow(\mathsf{Val}\,a \to \mathsf{Eval}\,b) \to \mathsf{Val}\,a \to \mathsf{Eval}\,b$$
$$app = \{\mathsf{ty}\,a \cdot \mathsf{ty}\,b \cdot \mathsf{val\,ref}\,f \cdot \mathsf{val\,ref}\,x \cdot \mathsf{eval\,sub} \to f.\mathsf{call}(\mathsf{val}\,x).\,\mathsf{eval\,sub}\}$$

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 $\mathsf{Clos} : \textbf{cplx}\,\textbf{comp} \to \mathsf{ref}\,\textbf{val}$
  - Closure introduced by $\mathsf{clos}\,\{\ldots\}$ around copattern-matching code
  - Closure $f : \mathsf{Clos}\,a$ eliminated with $f.\mathsf{call}$ operation

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

$$app : \forall a : \mathsf{Type}\,\mathsf{ref}\,\textbf{val}\,.\forall b : \mathsf{Type}\,\mathsf{sub}\,\textbf{comp}\,.\,{\downarrow}(\mathsf{Val}\,a \to \mathsf{Eval}\,b) \to \mathsf{Val}\,a \to \mathsf{Eval}\,b$$
$$app = \{\mathsf{ty}\,a \cdot \mathsf{ty}\,b \cdot \mathsf{val}\,\mathsf{ref}\,f \cdot \mathsf{val}\,\mathsf{ref}\,x \cdot \mathsf{eval}\,\mathsf{sub} \to f.\mathsf{call}(\mathsf{val}\,x).\,\mathsf{eval}\,\mathsf{sub}\}$$

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

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

$$app : \forall a : \mathsf{Type}\,\mathsf{ref}\,\mathbf{val}\,.\forall b : \mathsf{Type}\,\mathsf{sub}\,\mathbf{comp}\,.\downarrow(\mathsf{Val}\,a \to \mathsf{Eval}\,b) \to \mathsf{Val}\,a \to \mathsf{Eval}\,b$$
$$app = \{\mathsf{ty}\ a \cdot \mathsf{ty}\ b \cdot \mathsf{val}\,\mathsf{ref}\ f \cdot \mathsf{val}\,\mathsf{ref}\ x \cdot \mathsf{eval}\,\mathsf{sub} \to f.\,\mathsf{call}(\mathsf{val}\ x).\,\mathsf{eval}\,\mathsf{sub}\}$$

$$app' = \lambda f.\,f$$

$$app' : \forall a : \mathsf{Type}\,\mathbf{cplx\ val}\,.\forall b : \mathsf{Type}\,\mathbf{cplx\ comp}\,.\downarrow(a \to b) \to \uparrow\downarrow(a \to b)$$
$$app' = \{\mathsf{ty}\ a \cdot \mathsf{ty}\ b \cdot \mathsf{val}\,\mathsf{ref}\ f : \mathsf{Clos}(a \to b) \cdot \mathsf{eval}\,\mathsf{sub} \to \mathbf{ret}\ \mathsf{val}\ f\}$$

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

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

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

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

$$app' = \lambda f.\,f$$

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

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

Stress Test: Representation-Polymorphic Overloading

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

$$\begin{aligned}\textbf{class}\ \mathsf{Num}\ a\ \textbf{where}\ (+) &:: a \to a \to a \\ \mathit{negate} &:: a \to a\end{aligned}$$

What can we do without explicit representation polymorphism?

Stress Test: Representation-Polymorphic Overloading

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

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

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

$$\begin{aligned} \textbf{class}\ \mathsf{Num}\ a\ \textbf{where}\ (+) &:: a \to a \to a \\ \mathit{negate} &:: a \to a \end{aligned}$$

What can we do without explicit representation polymorphism?

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

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

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

Stress Test: Representation-Polymorphic Overloading

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

$$\begin{aligned} \textbf{class}\ \mathsf{Num}\ a\ \textbf{where}\ (+) &:: a \to a \to a \\ negate &:: a \to a \end{aligned}$$

What can we do without explicit representation polymorphism?

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

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

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

Stress Test: Representation-Polymorphic Overloading

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

$$\begin{aligned}\textbf{class}\ \mathsf{Num}\ a\ \textbf{where}\ (+) &:: a \to a \to a \\ \mathit{negate} &:: a \to a\end{aligned}$$

What can we do without explicit representation polymorphism?

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

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

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

Compiling to the Machine

Complex patterns $\implies$ 1 simple switch

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

$$\begin{aligned}
\textbf{unbox}\, x\, \textbf{as}\, \{\ & 0, 0, \mathsf{val\,int}\, y && \to M_1;\\
& 0, 1, \mathsf{val\,flt}\, y, \mathsf{val\,int}\, z && \to M_2;\\
& 1, () && \to M_3\}
\end{aligned}$$

Compiling to the Machine

Complex patterns $\Longrightarrow$ 1 simple switch

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

$$\begin{aligned}\textbf{unbox}\ x\ \textbf{as}\ \{\ & 0, 0, \mathsf{val\,int}\ y && \to M_1;\\ & 0, 1, \mathsf{val\,flt}\ y, \mathsf{val\,int}\ z && \to M_2;\\ & 1, () && \to M_3\}\end{aligned}$$

```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

$$\begin{aligned} \mathit{and}\ \mathsf{True}\ x &= x \\ \mathit{and}\ \mathsf{False}\ x &= \mathsf{False} \end{aligned}$$

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

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

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

is syntactic shorthand for

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

Complex answers

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

$$\begin{aligned}
\mathit{app}\ &: \forall a : \mathsf{Type}\,\mathsf{ref}\,\mathbf{val}\,.\forall b : \mathsf{Type}\,\mathsf{sub}\,\mathbf{comp}\,.\,{\downarrow}(\mathsf{Val}\,a \to \mathsf{Eval}\,b) \to \mathsf{Val}\,a \to \mathsf{Eval}\,b \\
\mathit{app}\ &= \{\mathsf{ty}\,a \cdot \mathsf{ty}\,b \cdot \mathsf{val}\,\mathsf{ref}\,f \cdot \mathsf{more} \in \{\ \mathsf{val}\,\mathsf{ref}\,x \cdot \mathsf{eval}\,\mathsf{sub}\ \} \to f\,.\,\mathsf{call}\}
\end{aligned}$$

$$\begin{aligned}
\mathit{app2} &: \forall a, b : \mathsf{Type}\,\mathsf{ref}\,\mathbf{val}\,.\forall c : \mathsf{Type}\,\mathsf{sub}\,\mathbf{comp}\,. \\
&\qquad {\downarrow}(\mathsf{Val}\,a \to \mathsf{Val}\,b \to \mathsf{Eval}\,c) \to \mathsf{Val}\,a \to \mathsf{Val}\,b \to \mathsf{Eval}\,c \\
\mathit{app2} &= \{\mathsf{ty}\,a \cdot \mathsf{ty}\,b \cdot \mathsf{ty}\,c \cdot \mathsf{val}\,\mathsf{ref}\,f \cdot \mathsf{more} \in \{\ \mathsf{val}\,\mathsf{ref}\,x \cdot \mathsf{val}\,\mathsf{ref}\,y \cdot \mathsf{eval}\,\mathsf{sub}\ \} \to f\,.\,\mathsf{call}\}
\end{aligned}$$

is syntactic shorthand for

$$\begin{aligned}
\mathit{app}\ &= \{\mathsf{ty}\,a \cdot \mathsf{ty}\,b \cdot \mathsf{val}\,\mathsf{ref}\,f \cdot \mathsf{val}\,\mathsf{ref}\,x \cdot \mathsf{eval}\,\mathsf{sub} \to f\,.\,\mathsf{call}(\mathsf{val}\,x).\,\mathsf{eval}\,\mathsf{sub}\} \\
\mathit{app2} &= \{\mathsf{ty}\,a \cdot \mathsf{ty}\,b \cdot \mathsf{ty}\,c \cdot \mathsf{val}\,\mathsf{ref}\,f \cdot \mathsf{val}\,\mathsf{ref}\,x \cdot \mathsf{val}\,\mathsf{ref}\,y \cdot \mathsf{eval}\,\mathsf{sub} \\
&\qquad\qquad \to f\,.\,\mathsf{call}(\mathsf{val}\,x)\ (\mathsf{val}\,y)\ .\,\mathsf{eval}\,\mathsf{sub}\}
\end{aligned}$$