# Compiling to categories

Conal Elliott

Target, USA

September 2017

# Overloading

- Alternative interpretation of common vocabulary.
- Laws for modular reasoning.
- Doesn't apply to lambda, variables, and application.
- Instead, *eliminate* them.

# Eliminating lambda

$$
\begin{aligned}
(\lambda p \rightarrow k) &\dashrightarrow \mathit{const}\ k \\
(\lambda p \rightarrow p) &\dashrightarrow \mathit{id} \\
(\lambda p \rightarrow u\ v) &\dashrightarrow \mathit{apply} \circ ((\lambda p \rightarrow u) \vartriangle (\lambda p \rightarrow v)) \\
(\lambda p \rightarrow \lambda q \rightarrow u) &\dashrightarrow \mathit{curry}\ (\lambda (p, q) \rightarrow u) \\
&\dashrightarrow \mathit{curry}\ (\lambda r \rightarrow u\ [p \coloneqq \mathit{fst}\ r, q \coloneqq \mathit{snd}\ r])
\end{aligned}
$$

Automate via a compiler plugin.

# Examples

$$sqr :: Num\ a \Rightarrow a \to a$$
$$sqr\ a = a * a$$

$$magSqr :: Num\ a \Rightarrow a \times a \to a$$
$$magSqr\ (a, b) = sqr\ a + sqr\ b$$

$$cosSinProd :: Floating\ a \Rightarrow a \times a \to a \times a$$
$$cosSinProd\ (x, y) = (cos\ z, sin\ z)\ \mathbf{where}\ z = x * y$$

After $\lambda$-elimination:

$$sqr = mulC \circ (id \vartriangle id)$$

$$magSqr = addC \circ (mulC \circ (exl \vartriangle exl) \vartriangle mulC \circ (exr \vartriangle exr))$$

$$cosSinProd = (cosC \vartriangle sinC) \circ mulC$$

# Abstract algebra for functions

Interface:

$$
\begin{array}{l}
\textbf{class } \mathit{Category}\ k\ \textbf{where} \\
\quad \mathit{id}\ \ :: a\ \grave{}k\grave{}\ a \\
\quad (\circ) :: (b\ \grave{}k\grave{}\ c) \rightarrow (a\ \grave{}k\grave{}\ b) \rightarrow (a\ \grave{}k\grave{}\ c) \\
\quad \textbf{infixr}\ 9\ \circ
\end{array}
$$

Laws:

$$
\begin{array}{ll}
\mathit{id} \circ f & \equiv f \\
g \circ \mathit{id} & \equiv g \\
(h \circ g) \circ f & \equiv h \circ (g \circ f)
\end{array}
$$

# Products

Interface:

$$
\begin{array}{l}
\textbf{class } \mathit{Category}\ k \Rightarrow \mathit{Cartesian}\ k\ \textbf{where} \\
\quad \textbf{type } a \times_k b \\
\quad \mathit{exl} :: (a \times_k b) \text{ `}k\text{` } a \\
\quad \mathit{exr} :: (a \times_k b) \text{ `}k\text{` } b \\
\quad (\vartriangle) :: (a \text{ `}k\text{` } c) \to (a \text{ `}k\text{` } d) \to (a \text{ `}k\text{` } (c \times_k d)) \\
\quad \textbf{infixr } 3 \vartriangle
\end{array}
$$

Laws:

$$
\begin{array}{ll}
\mathit{exl} \circ (f \vartriangle g) & \equiv f \\
\mathit{exr} \circ (f \vartriangle g) & \equiv g \\
\mathit{exl} \circ h \vartriangle \mathit{exr} \circ h & \equiv h
\end{array}
$$

# Coproducts

Dual to product.

$$
\begin{array}{l}
\textbf{class } \mathit{Category}\ k \Rightarrow \mathit{Cocartesian}\ k \textbf{ where} \\
\quad \textbf{type } a +_k b \\
\quad \mathit{inl}\ :: a \,\grave{}k\grave{}\, (a +_k b) \\
\quad \mathit{inr} :: b \,\grave{}k\grave{}\, (a +_k b) \\
\quad (\triangledown) :: (a \,\grave{}k\grave{}\, c) \to (b \,\grave{}k\grave{}\, c) \to ((a +_k b) \,\grave{}k\grave{}\, c) \\
\quad \textbf{infixr } 2\ \triangledown
\end{array}
$$

Laws:

$$
\begin{array}{ll}
(f \triangledown g) \circ \mathit{inl} & \equiv f \\
(f \triangledown g) \circ \mathit{inr} & \equiv g \\
h \circ \mathit{inl} \triangledown h \circ \mathit{inr} & \equiv h
\end{array}
$$

# Exponentials

First-class "functions" (morphisms):

$$
\begin{array}{l}
\mathbf{class}\ \mathit{Cartesian}\ k \Rightarrow \mathit{CartesianClosed}\ k\ \mathbf{where} \\
\quad \mathbf{type}\ a \Rightarrow_k b \\
\quad \mathit{apply} \quad\ \ :: ((a \Rightarrow_k b) \times_k a)\ \grave{}k\grave{}\ b \\
\quad \mathit{curry} \quad\ \ :: ((a \times_k b)\ \grave{}k\grave{}\ c) \rightarrow (a\ \grave{}k\grave{}\ (b \Rightarrow_k c)) \\
\quad \mathit{uncurry} :: (a\ \grave{}k\grave{}\ (b \Rightarrow_k c)) \rightarrow ((a \times_k b)\ \grave{}k\grave{}\ c)
\end{array}
$$

Laws:

$$
\begin{array}{ll}
\mathit{uncurry}\ (\mathit{curry}\ f) & \equiv f \\
\mathit{curry}\ (\mathit{uncurry}\ g) & \equiv g \\
\mathit{apply} \circ (\mathit{curry}\ f \circ \mathit{exl} \vartriangle \mathit{exr}) & \equiv f
\end{array}
$$

# Misc operations

**class** $NumCat\ k\ a$ **where**

$$
\begin{array}{ll}
\quad negateC & :: a \mathbin{`k`} a \\
\quad addC, sub, mulC & :: (a \times_k a) \mathbin{`k`} a \\
\quad \ldots &
\end{array}
$$

...

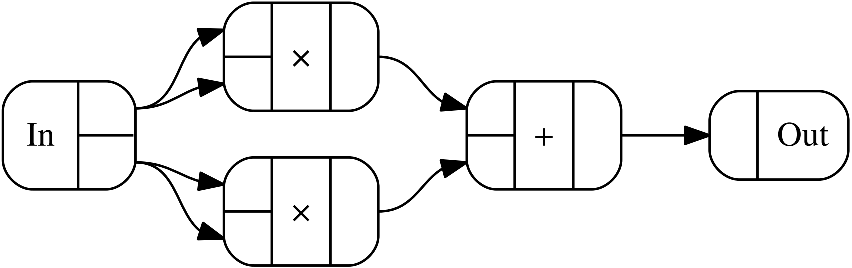
# Changing interpretations

- We've eliminated lambdas and variables
- and replaced them with an algebraic vocabulary.
- What happens if we *replace* $(\rightarrow)$ *with other instances?*
  (Via compiler plugin.)

# Computation graphs — example

$$magSqr\ (a, b) = sqr\ a + sqr\ b$$

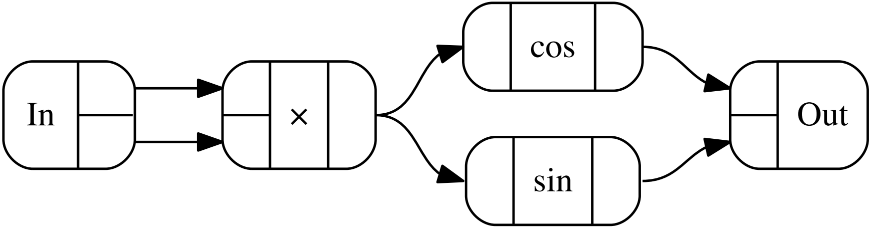
$$magSqr = addC \circ (mulC \circ (exl \vartriangle exl) \vartriangle mulC \circ (exr \vartriangle exr))$$



# Computation graphs — example

$$cosSinProd\ (x, y) = (cos\ z, sin\ z)\ \textbf{where}\ z = x * y$$

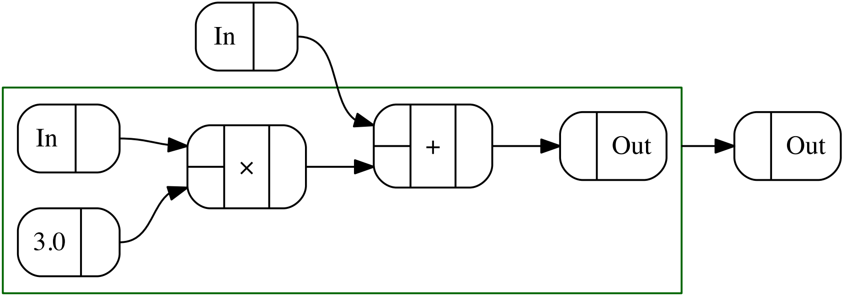
$$cosSinProd = (cosC \vartriangle sinC) \circ mulC$$



# Computation graphs — example

$\lambda x\ y \rightarrow x + 3 * y$

$curry\ (addC \circ (exl \vartriangle mulC \circ (const\ 3.0 \vartriangle exr)))$



# Computation graphs — implementation sketch

$\textbf{newtype}\ \mathit{Graph}\ a\ b = \mathit{Graph}\ (\mathit{Ports}\ a \to \mathit{GraphM}\ (\mathit{Ports}\ b))$

$\textbf{type}\ \mathit{GraphM} = \mathit{State}\ (\mathit{PortNum}, [\mathit{Comp}])$

$\textbf{data}\ \mathit{Comp} = \forall a\ b.\mathit{Comp}\ (\mathit{Template}\ a\ b)\ (\mathit{Ports}\ a)\ (\mathit{Ports}\ b)$

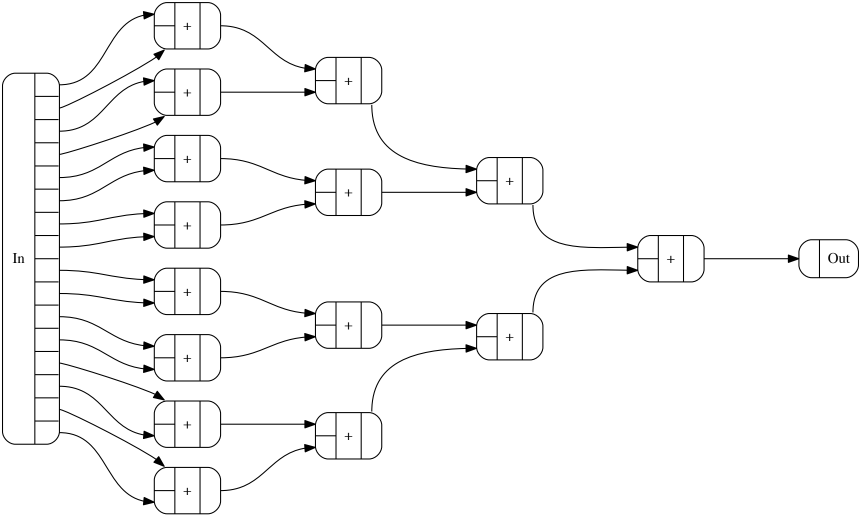
$$
\begin{array}{l}
\textbf{data}\ \mathit{Template} :: * \to * \to *\ \textbf{where} \\
\quad \mathit{Prim} \qquad :: \mathit{String} \to \mathit{Template}\ a\ b \\
\quad \mathit{Subgraph} :: \mathit{Graph}\ a\ b \to \mathit{Template}\ ()\ (a \to b)
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Category}\ \mathit{Graph}\ \textbf{where} \\
\quad \mathit{id} = \mathit{Graph}\ \mathit{return} \\
\quad \mathit{Graph}\ g \circ \mathit{Graph}\ f = \mathit{Graph}\ (g <\!\!=\!\!< f)
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance}\ \mathit{BoolCat}\ \mathit{Graph}\ \textbf{where} \\
\quad \mathit{notC} \ = \mathit{genComp}\ \texttt{"}\neg\texttt{"} \\
\quad \mathit{andC} = \mathit{genComp}\ \texttt{"}\wedge\texttt{"} \\
\quad \mathit{orC} \ \ \ = \mathit{genComp}\ \texttt{"}\vee\texttt{"}
\end{array}
$$

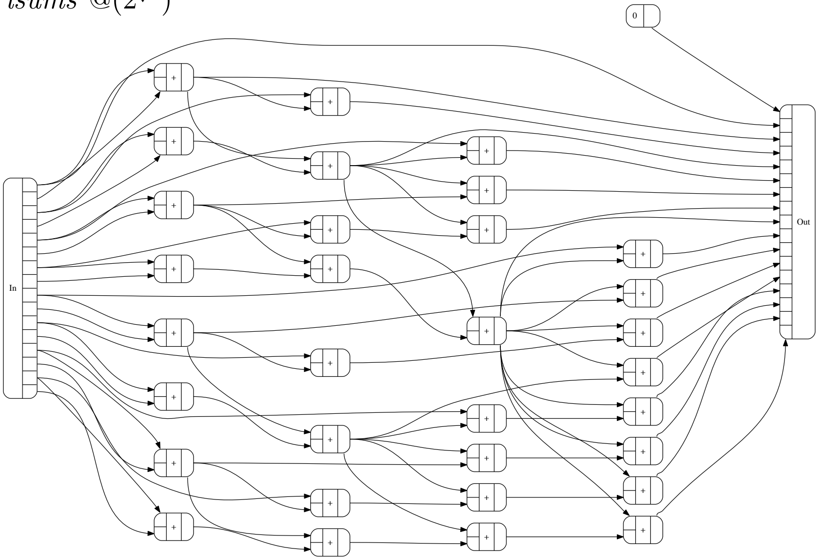
# Computation graphs — fold

$sum \ @(2^{\downarrow 4})$



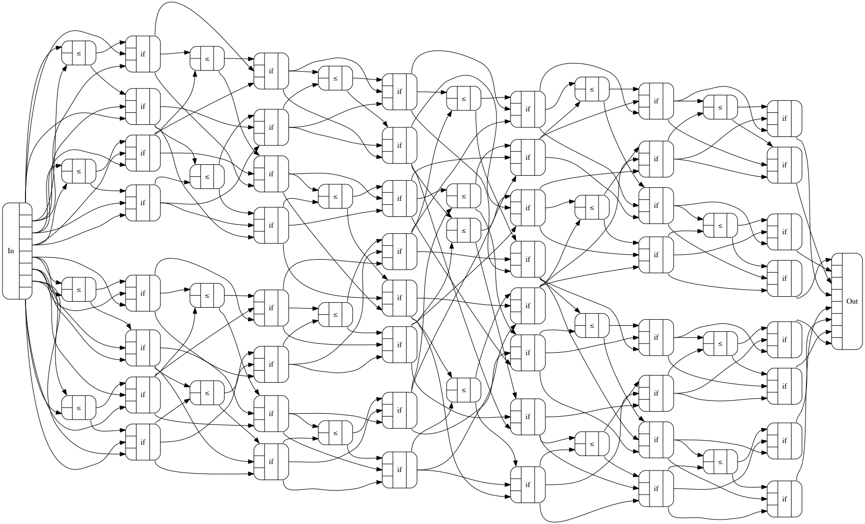
# Computation graphs — scan

$lsums\ @(2^{\downarrow 4})$



# Computation graphs — bitonic sort

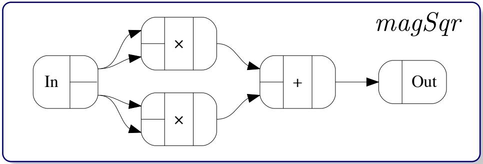
*bitonic* $@(2^{\downarrow 3})$



# Compiling to hardware

Convert graphs to Verilog:

```
module magSqr (In_0, In_1, Out);
  input [31:0] In_0;
  input [31:0] In_1;
  output [31:0] Out;
  wire [31:0] Plus_I0;
  wire [31:0] Times_I3;
  wire [31:0] Times_I4;
  assign Plus_I0 = Times_I3 + Times_I4;
  assign Out = Plus_I0;
  assign Times_I3 = In_0 * In_0;
  assign Times_I4 = In_1 * In_1;
endmodule
```

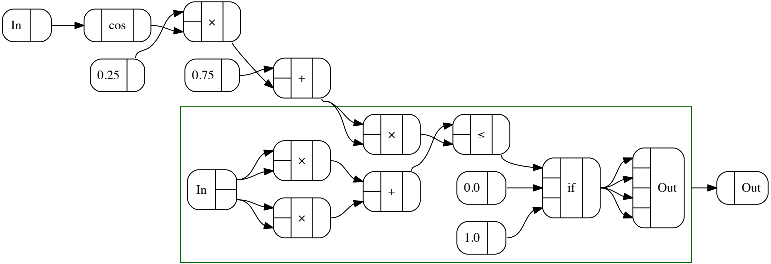


# Example — graphics

$$disk :: \mathbb{R} \to Region$$
$$disk\ r\ p = magSqr\ p \leqslant sqr\ r$$

$$anim\ t = disk\ (3/4 + 1/4 * cos\ t)$$

$$\textbf{type}\ Region = \mathbb{R} \times \mathbb{R} \to Bool$$




```
uniform float in0;
vec4 animA (float in1, float in2)
{ float v5 = 0.75 + 0.25 * cos (in0);  // TODO: hoist
  float v24 = in1 * in1 + in2 * in2 <= v5 * v5 ? 0.0 : 1.0;
  return vec4 (v24, v24, v24, v24);
}
```

# Differentiable functions

$\mathbf{newtype}\ D\ a\ b = D\ (a \rightarrow (b \times (a \multimap b)))$ -- derivative as linear map

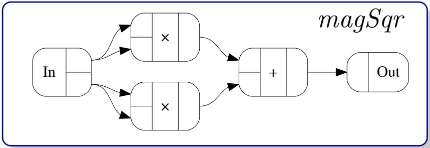
$linearD\ f = D\ (\lambda a \rightarrow (f\ a, linear\ f))$

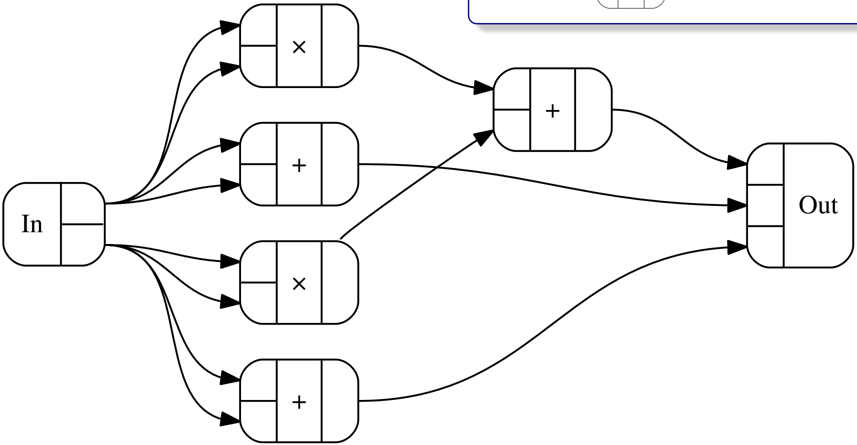
$\mathbf{instance}\ Category\ D\ \mathbf{where}$
$\quad id = linearD\ id$
$\quad D\ g \circ D\ f = D\ (\lambda a \rightarrow \mathbf{let}\ \{(b, f') = f\ a; (c, g') = g\ b\}\ \mathbf{in}\ (c, g' \circ f'))$

$\mathbf{instance}\ Cartesian\ D\ \mathbf{where}$
$\quad exl = linearD\ exl$
$\quad exr = linearD\ exr$
$\quad D\ f \vartriangle D\ g = D\ (\lambda a \rightarrow \mathbf{let}\ \{(b, f') = f\ a; (c, g') = g\ a\}\ \mathbf{in}\ ((b, c), f' \vartriangle g'))$

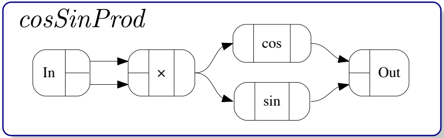
$\mathbf{instance}\ NumCat\ D\ \mathbf{where}$
$\quad negateC = linearD\ negateC$
$\quad addC = linearD\ addC$
$\quad mulC = D\ (mulC \vartriangle \lambda(a, b) \rightarrow linear\ (\lambda(da, db) \rightarrow da * b + db * a))$

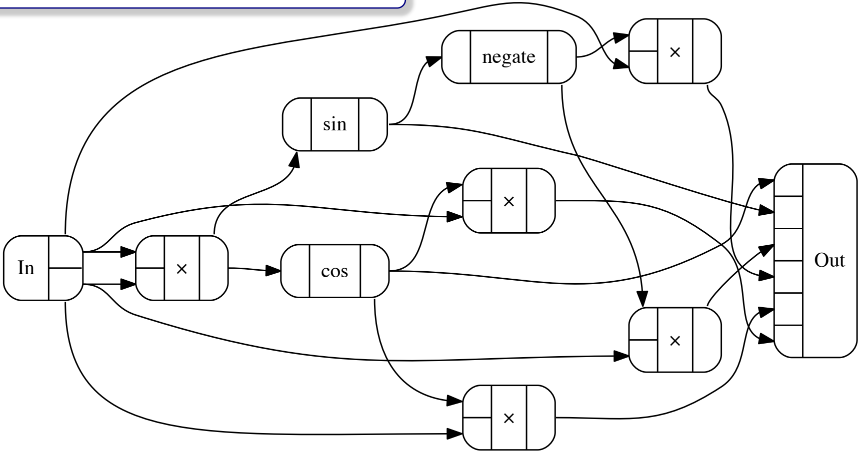
# Composing interpretations (*Graph* and *D*)



# Composing interpretations (*Graph* and *D*)



# Interval analysis

$$\textbf{data}\ IFun\ a\ b = IFun\ (Interval\ a \to Interval\ b)$$

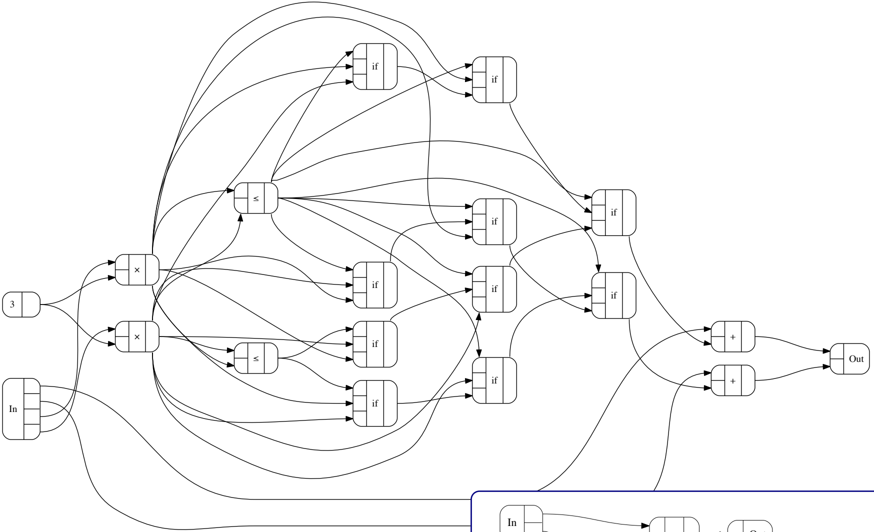
$$\begin{array}{l}
\textbf{type}\ family\ Interval\ a \\
\textbf{type instance}\ Interval\ Double\ \ = Double \times Double \\
\textbf{type instance}\ Interval\ (a \times b) = Interval\ a \times Interval\ b \\
\textbf{type instance}\ Interval\ (a \to b) = Interval\ a \to Interval\ b
\end{array}$$

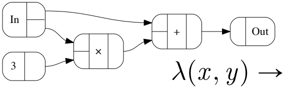
$$\begin{array}{l}
\textbf{instance}\ Category\ IFun\ \textbf{where} \\
\quad id = IFun\ id \\
\quad IFun\ g \circ IFun\ f = IFun\ (g \circ f) \\
\ldots
\end{array}$$

$$\begin{array}{l}
\textbf{instance}\ Cartesian\ IFun\ \textbf{where} \\
\quad exl = IFun\ exl \\
\quad exr = IFun\ exr \\
\quad IFun\ f \vartriangle IFun\ g = IFun\ (f \vartriangle g)
\end{array}$$

$$\begin{array}{l}
\textbf{instance}\ (Interval\ a \sim (a \times a), Num\ a, Ord\ a) \Rightarrow NumCat\ IFun\ a\ \textbf{where} \\
\quad addC = IFun\ (\lambda((a_{lo}, a_{hi}), (b_{lo}, b_{hi})) \to (a_{lo} + b_{lo}, a_{hi} + b_{hi})) \\
\quad mulC = IFun\ (\lambda((a_{lo}, a_{hi}), (b_{lo}, b_{hi})) \to \\
\qquad minmax\ [a_{lo} * b_{lo}, a_{lo} * b_{hi}, a_{hi} * b_{lo}, a_{hi} * b_{hi}] \\
\quad \ldots
\end{array}$$

# Interval analysis — example




$\lambda(x, y) \to x + 3 * y$

# Other examples

- Constraint solving via SMT (with John Wiegley)
- Linear maps
- Incremental evaluation
- Polynomials
- Nondeterministic and probabilistic programming

# Shallow embedding

- “Just a library”, but with a suitable host language.
- Easy to implement; but restricts optimization.
- Inherits host language & compiler *limitations*, e.g., no
  - differentiation or integration
  - incremental evaluation
  - optimization
  - constraint solving
  - novel back-ends, e.g., GPU, circuits, JavaScript

# Deep embedding

- Syntactic representation.
- More room for analysis and optimization.
- Harder to implement; redundant with host compiler.
- Requires some vocabulary changes.

# Compiling to categories

- Just a library.
- Easy to implement.
- Analysis, optimization, non-standard target architectures.
- Non-standard operations on functions.