Teaching new tricks to old programs

Conal Elliott

Target Data Sciences

May 2017

Domain-specific embedded languages

New vocabularies, not new languages.

The Next 700 Programming Languages

P. J. Landin

Univac Division of Sperry Rand Corp., New York, New York

"... today ... 1,700 special programming languages used to 'communicate' in over 700 application areas."—*Computer Software* Issues, an American Mathematical Association Prospectus, July 1965.

Domain-specific embedded languages

New vocabularies, not new languages.

The Next 700 Programming Languages

P. J. Landin

Univac Division of Sperry Rand Corp., New York, New York

"... today ... 1,700 special programming languages used to 'communicate' in over 700 application areas."—*Computer Software* Issues, an American Mathematical Association Prospectus, July 1965.

Can we create fewer new vocabularies as well?

What does it mean?

What does it mean?

$$x + 3 * y$$

What does it mean?

$$x + 3 * y$$

It depends on x and y.

What does it mean?

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

What does it mean?

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

It depends on $+$, $*$, and 3.

What does it mean?

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

It depends on $+$, $*$, and 3

What does it mean?

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

It depends on $+$, $*$, and 3:

- *Int*, *Float*, *Double*
- $\mathbb{Z}$, $\mathbb{N}$, $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{C}$

What does it mean?

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

It depends on $+$, $*$, and 3:

- *Int*, *Float*, *Double*
- $\mathbb{Z}$, $\mathbb{N}$, $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{C}$
- Vectors
- Polynomials
- Functions
- Regular expressions/languages
- Arbitrary rings, semirings, ...

Organizing interpretations

- Abstract algebra: interfaces and laws, e.g.,
  - Monoid, group, ring
  - Vector space
  - Functor, applicative, monad, foldable, traversable
  - Category, with products, with coproducts/sums
- Refactor and repurpose proofs and programs. (More with less.)

Example,

$$\mathit{fold} :: (\mathit{Foldable}\ f, \mathit{Monoid}\ m) \Rightarrow f\ m \to m$$

What does it mean?

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

What does it mean?

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

- The most basic “operations”: λ, variables, and application.
- We can’t re-interpret/overload.

What does it mean?

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

- The most basic “operations”: λ, variables, and application.
- We can’t re-interpret/overload.
- What if there were a way?

Why overload lambda (etc)?

Same benefits as algebraic abstraction:

- Convenient notation.
- Generalized, principled interpretation.
- Modular programming and reasoning.

Why overload lambda?

- Convenient notation for functions.

Why overload lambda?

- Convenient notation for functions.
- Alternative function implementations:
  - GPU code
  - Circuits
  - Javascript

Why overload lambda?

- Convenient notation for functions.
- Alternative function implementations:
  - GPU code
  - Circuits
  - Javascript
- Enhanced functions:
  - Derivatives and integrals
  - Incremental evaluation
  - Interval analysis
  - Optimization
  - Root-finding
  - Constraint solving

How to overload lambda?

How to overload lambda?

- Idea: eliminate it, and overload as usual.

How to overload lambda?

- Idea: eliminate it, and overload as usual.
- How?

Eliminating lambda

Introducing lambda

Introducing lambda

$$const :: b \rightarrow (a \rightarrow b)$$
$$const\ b = \lambda a \rightarrow b$$

$$id :: a \rightarrow a$$
$$id = \lambda a \rightarrow a$$

$$(\circ) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)$$
$$g \circ f = \lambda a \rightarrow g\ (f\ a)$$

$$(\vartriangle) :: (a \rightarrow c) \rightarrow (a \rightarrow d) \rightarrow (a \rightarrow c \times d)$$
$$f \vartriangle g = \lambda a \rightarrow (f\ a, g\ a)$$

$$curry :: (a \times b \rightarrow c) \rightarrow (a \rightarrow b \rightarrow c)$$
$$curry\ f = \lambda a \rightarrow \lambda b \rightarrow f\ (a, b)$$

$$apply :: (a \rightarrow b) \times a \rightarrow b$$
$$apply = \lambda (f, a) \rightarrow f\ a$$
$$= uncurry\ id$$

Eliminating lambda

Systematically *un-inline:*

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

Automate via a compiler plugin.

Examples

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

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

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

After λ-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:

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

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:

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

Laws:

$$\mathit{exl} \circ (f \vartriangle g) \equiv f$$
$$\mathit{exr} \circ (f \vartriangle g) \equiv g$$
$$\mathit{exl} \circ h \vartriangle \mathit{exr} \circ h \equiv h$$

Coproducts

Dual to product.

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

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):

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

Laws:

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

Misc operations

class *NumCat k a* **where**

$$
\begin{aligned}
&negateC && :: a \ \grave{}k\grave{} \ a \\
&addC, sub, mulC && :: (a \times_k a) \ \grave{}k\grave{} \ a \\
&\ldots
\end{aligned}
$$

...

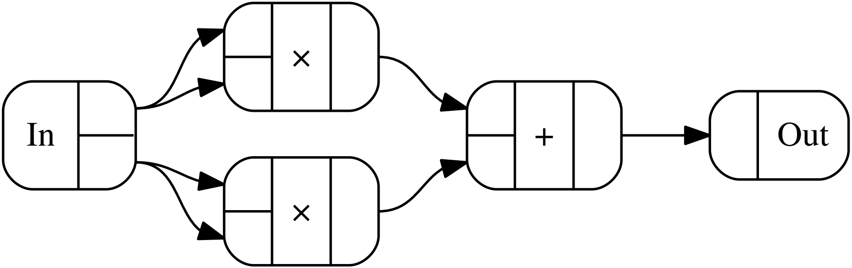
Changing interpretations

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

Computation graphs — example

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

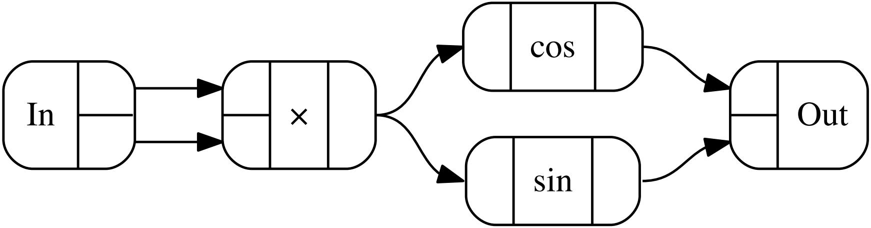
$$magSqr = addC \circ (mulC \circ (exl \vartriangle exl) \vartriangle mulC \circ (exr \vartriangle exr))$$



Computation graphs — example

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

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



Computation graphs — example

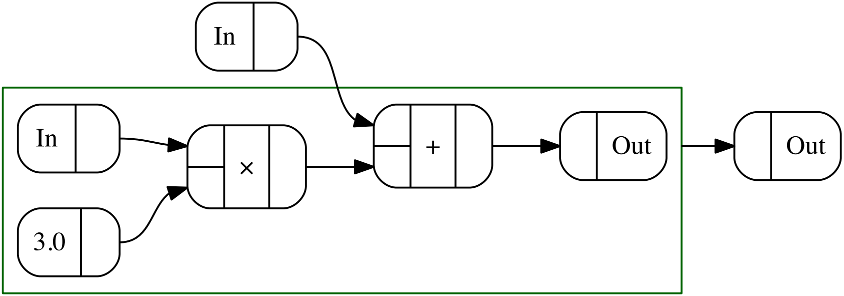
$\lambda x\ y \rightarrow x + 3 * y$

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

Computation graphs — example

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

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



Computation graphs — implementation sketch

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

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

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

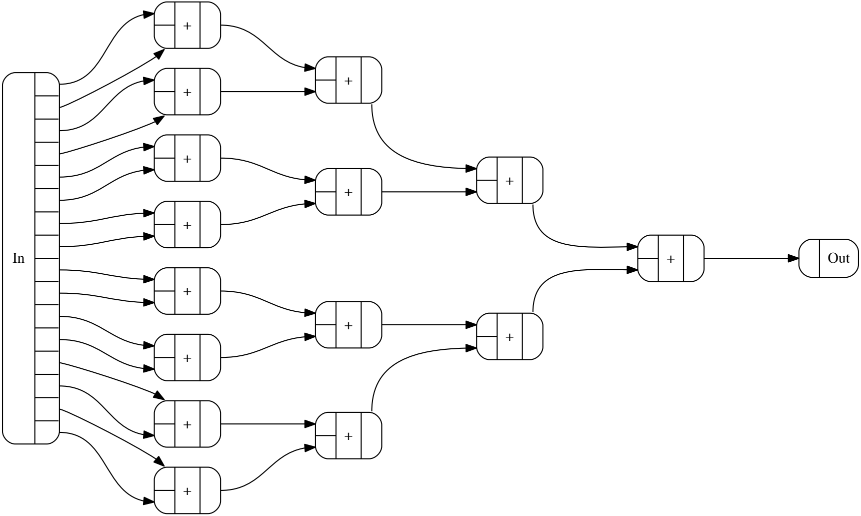
$$\begin{array}{l}\textbf{data}\ \mathit{Template} :: * \rightarrow * \rightarrow *\ \textbf{where} \\ \quad \mathit{Prim} \quad\quad :: \mathit{String} \rightarrow \mathit{Template}\ a\ b \\ \quad \mathit{Subgraph} :: \mathit{Graph}\ a\ b \rightarrow \mathit{Template}\ ()\ (a \rightarrow b)\end{array}$$

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

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

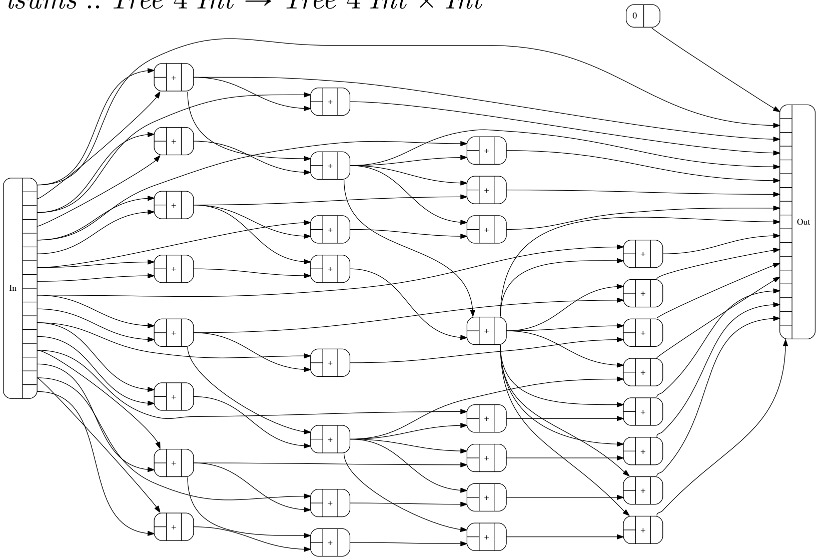
Computation graphs — fold

$sum :: Tree\ 4\ Int \rightarrow Int$



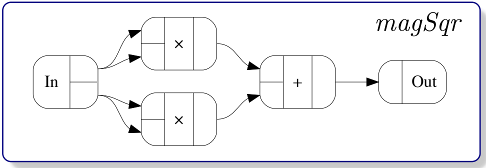
Computation graphs — scan

$lsums :: Tree\ 4\ Int \rightarrow Tree\ 4\ Int \times Int$



Haskell to hardware

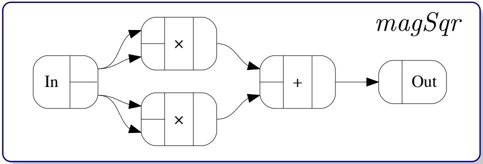
Convert graphs to Verilog:



Haskell 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

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

$woob\ t = disk\ (0.75 + 0.25 * cos\ t)$

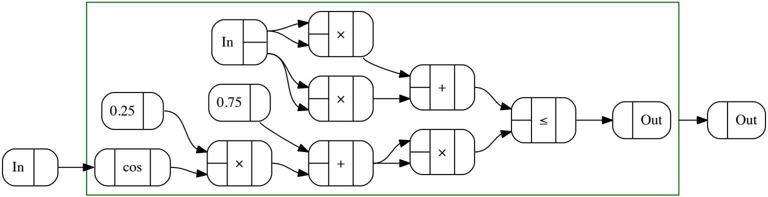
type $Region = \mathbb{R} \times \mathbb{R} \to Bool$

Example — graphics

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

$woob\ t = disk\ (0.75 + 0.25 * cos\ t)$

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

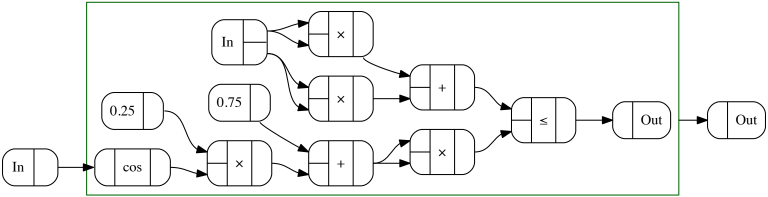


Example — graphics

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

$$woob\ t = disk\ (0.75 + 0.25 * cos\ t)$$

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




```
bool uwoob (float in0, float in1, float in2)  // Generated GLSL
{ float v17 = 1.0;
  float v23 = v17 / (0.75 + 0.25 * cos (in0));
  float v24 = in1 * v23;
  float v26 = in2 * v23;
  return v24 * v24 + v26 * v26 <= v17;
}
vec4 effect (vec2 p) { return bw(uwoob(time,p.x,p.y)); }
```

Automatic differentiation

$\mathbf{data}\ D\ a\ b = D\ (a \rightarrow b \times (a \multimap b))$ -- Derivatives are linear maps.

Automatic differentiation

$\mathbf{data}\ D\ a\ b = D\ (a \to b \times (a \multimap b))$ -- Derivatives are linear maps.

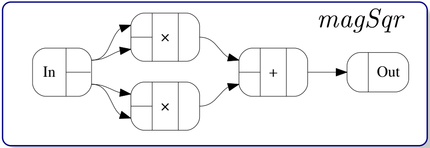
$linearD\ f = D\ (\lambda a \to (f\ a, linear\ f))$

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

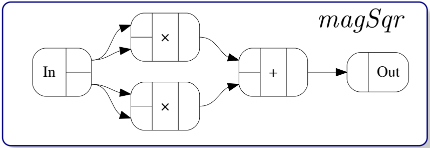
$\mathbf{instance}\ Cartesian\ D\ \mathbf{where}$
$\quad exl = linearD\ exl$
$\quad exr = linearD\ exr$
$\quad D\ f \vartriangle D\ g = D\ (\lambda a \to \mathbf{let}\ \{(b, f') = f\ a; (c, g') = g\ a\}\ \mathbf{in}\ ((b, c), f' \vartriangle g'))$

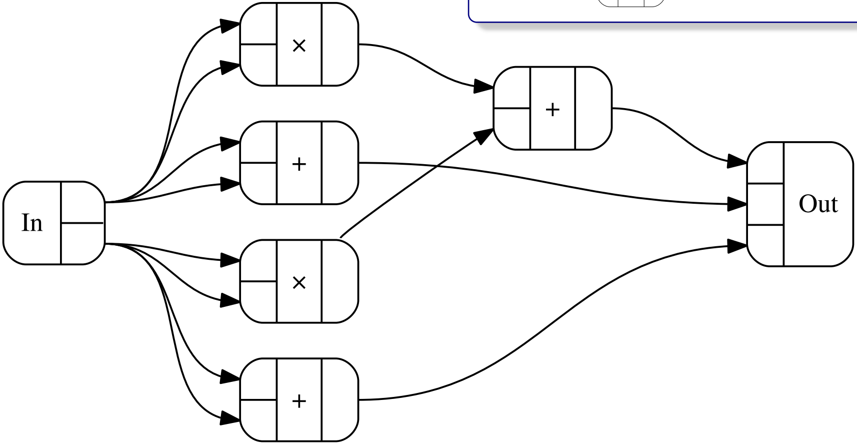
$\mathbf{instance}\ NumCat\ D\ \mathbf{where}$
$\quad negateC = linearD\ negateC$
$\quad addC = linearD\ addC$
$\quad mulC = D\ (mulC \vartriangle \lambda(a, b) \to linear\ (\lambda(da, db) \to da * b + db * a))$

Composing interpretations (*Graph* and *D*)



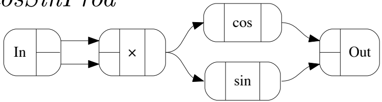
Composing interpretations (*Graph* and *D*)



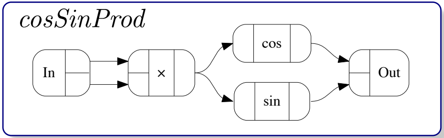


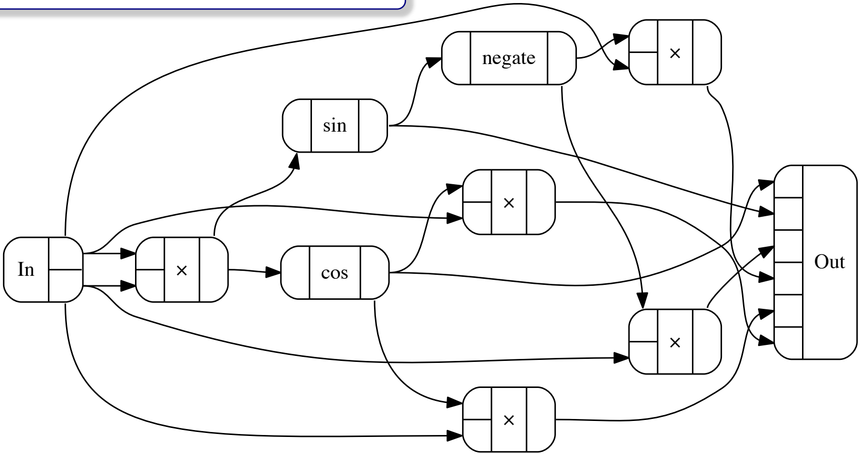
Composing interpretations (*Graph* and *D*)

cosSinProd



Composing interpretations (*Graph* and *D*)





Interval analysis

data $\mathit{IFun}\ a\ b = \mathit{IFun}\ (\mathit{Interval}\ a \rightarrow \mathit{Interval}\ b)$

Interval analysis

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

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

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

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

Interval analysis

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

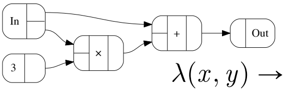
$$\begin{array}{l}
\textbf{type}\ \mathit{family}\ Interval\ a \\
\textbf{type instance}\ Interval\ Double\ \ = Double \times Double \\
\textbf{type instance}\ Interval\ (a \times b) = Interval\ a \times Interval\ b \\
\textbf{type instance}\ Interval\ (a \rightarrow b) = Interval\ a \rightarrow Interval\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

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

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

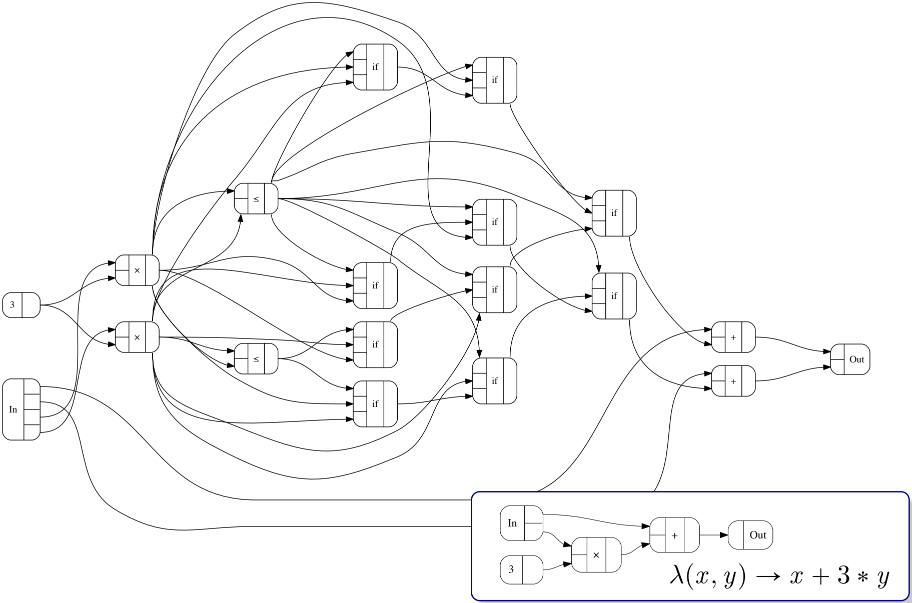
$$\begin{array}{l}
\textbf{instance}\ (Interval\ a \sim (a \times a), Num\ a, Ord\ a) \Rightarrow NumCat\ IFun\ a\ \textbf{where} \\
\quad addC = IFun\ (\lambda((a_{lo}, a_{hi}), (b_{lo}, b_{hi})) \rightarrow (a_{lo} + b_{lo}, a_{hi} + b_{hi})) \\
\quad mulC = IFun\ (\lambda((a_{lo}, a_{hi}), (b_{lo},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

Interval analysis — example



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

Interval analysis — example



Constraint solving (with John Wiegley)

$$\textbf{newtype}\ SMT\ a\ b = SMT\ (Kleisli\ Z3\ (E\ a)\ (E\ b))$$

$$
\begin{array}{l}
\textbf{data}\ E :: * \to * \ \textbf{where} \\
\quad PrimE :: AST \to E\ a \\
\quad PairE\ :: E\ a \to E\ b \to E\ (a \times b)
\end{array}
$$

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

$$
\begin{array}{l}
\textbf{instance}\ Cartesian\ SMT\ \textbf{where} \\
\quad exl\ = SMT\ (arr\ (exl \circ unpairE)) \\
\quad exr = SMT\ (arr\ (exr \circ unpairE)) \\
\quad SMT\ f \vartriangle SMT\ g = SMT\ (arr\ PairE \circ (f \vartriangle g))
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance}\ Num\ a \Rightarrow NumCat\ SMT\ a\ \textbf{where} \\
\quad negateC = liftE_1\ mkUnaryMinus \\
\quad addC \quad\ = liftE_2\ mkAdd \\
\quad subC \quad\ = liftE_2\ mkSub \\
\quad mulC \quad\ = liftE_2\ mkMul
\end{array}
$$

Constraint solving (with John Wiegley)

$$
\begin{array}{l}
\mathit{pred} :: (\mathit{Num}\ a, \mathit{Ord}\ a) \Rightarrow a \times a \rightarrow \mathit{Bool} \\
\mathit{pred}\ (x, y) = \\
\quad x < y\ \wedge \\
\quad y < 100\ \wedge \\
\quad 0 \leqslant x - 3 + 7 * y\ \wedge \\
\quad (x \equiv y\ \vee\ y + 20 \equiv x + 30)
\end{array}
$$

Solution: $(-8, 2)$.

Other examples

- Linear maps
- Incremental evaluation
- Polynomials
- Nondeterministic and probabilistic programming

Domain-specific embedded languages (DSELs)

- *Shallow* (just a library):
  - Great fit with host language.
  - Easy to implement and use.
  - Hard to optimize.
  - Good choice for *expressing ideas*.
- *Deep* (syntactic representation):
  - More room for analysis and optimization.
  - Harder to implement; redundant with host compiler.
  - Less semantic guidance.
  - Syntactically awkward in places.
  - Good choice for *efficient implementation*.

Domain-specific embedded languages (DSELs)

- *Shallow* (just a library):
  - Great fit with host language.
  - Easy to implement and use.
  - Hard to optimize.
  - Good choice for *expressing ideas*.
- *Deep* (syntactic representation):
  - More room for analysis and optimization.
  - Harder to implement; redundant with host compiler.
  - Less semantic guidance.
  - Syntactically awkward in places.
  - Good choice for *efficient implementation*.
- *Compiling to categories*:
  - Great fit with host language.
  - Semantic guidance.
  - Easy to implement.
  - Analysis, optimization, non-standard target architectures.

For more details

- The paper *Compiling to categories* (February 2017)
- GitHub project page