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.

Can we create fewer new vocabularies as well?
What does it mean?

\[ x + 3 \times y \]

It depends on \( x \) and \( y \).
What does it mean?

\[ \lambda x \; y \rightarrow x + 3 \ast y \]

It depends on \( +, \ast, \) and \( 3. \)
What does it mean?

\[ \lambda x \ y \rightarrow x + 3 \ast y \]

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,

\[
\text{fold} :: (\text{Foldable } f, \text{Monoid } m) \Rightarrow f \; m \rightarrow m
\]
What does it mean?

\[ \lambda x \ y \rightarrow x + 3 \times y \]

- The most basic “operations”: \( \lambda \), 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.

- 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?

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

\[\text{const} :: b \rightarrow (a \rightarrow b)\]
\[\text{const } b = \lambda a \rightarrow b\]

\[\text{id} :: a \rightarrow a\]
\[\text{id} = \lambda a \rightarrow a\]

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

\[(\triangle) :: (a \rightarrow c) \rightarrow (a \rightarrow d) \rightarrow (a \rightarrow c \times d)\]
\[f \triangle g = \lambda a \rightarrow (f \ a, g \ a)\]

\[\text{curry} :: (a \times b \rightarrow c) \rightarrow (a \rightarrow b \rightarrow c)\]
\[\text{curry } f = \lambda a \rightarrow \lambda b \rightarrow f (a, b)\]

\[\text{apply} :: (a \rightarrow b) \times a \rightarrow b\]
\[\text{apply} = \lambda (f, a) \rightarrow f \ a\]
\[= \text{uncurry } \text{id}\]
Eliminating lambda

Systematically *un-inline*:

\[(\lambda p \to k) \rightarrow \text{const } k\]

\[(\lambda p \to p) \rightarrow \text{id}\]

\[(\lambda p \to u \; v) \rightarrow \text{apply } \circ ((\lambda p \to u) \triangleright (\lambda p \to v))\]

\[(\lambda p \to \lambda q \to u) \rightarrow \text{curry } (\lambda (p, q) \to u)\]

\[\rightarrow \text{curry } (\lambda r \to u [p := \text{fst } r, q := \text{snd } r])\]

Automate via a compiler plugin.
Examples

\[\text{sqr} :: \text{Num } a \Rightarrow a \to a\]
\[\text{sqr } a = a \ast a\]

\[\text{magSqr} :: \text{Num } a \Rightarrow a \times a \to a\]
\[\text{magSqr } (a, b) = \text{sqr } a + \text{sqr } b\]

\[\text{cosSinProd} :: \text{Floating } a \Rightarrow a \times a \to a \times a\]
\[\text{cosSinProd } (x, y) = (\cos z, \sin z) \text{ where } z = x \ast y\]

After \(\lambda\)-elimination:

\[\text{sqr} = \text{mulC } \circ (\text{id } \triangle \text{id})\]

\[\text{magSqr} = \text{addC } \circ (\text{mulC } \circ (\text{exl } \triangle \text{exl}) \triangle \text{mulC } \circ (\text{exr } \triangle \text{exr}))\]

\[\text{cosSinProd} = (\text{cosC } \triangle \text{sinC}) \circ \text{mulC}\]
Abstract algebra for functions

Interface:

```haskell
class Category k where
  id :: a `k` a
  (○) :: (b `k` c) → (a `k` b) → (a `k` c)

infixr 9 ○
```

Laws:

\[
\begin{align*}
  \text{id} \circ f & \equiv f \\
  g \circ \text{id} & \equiv g \\
  (h \circ g) \circ f & \equiv h \circ (g \circ f)
\end{align*}
\]
Products

Interface:

```haskell
class Category k ⇒ Cartesian k where
    type a \times_k b
    exl :: (a \times_k b) \k\ a
    exr :: (a \times_k b) \k\ b
   (\triangle) :: (a \k\ c) → (a \k\ d) → (a \k\ (c \times_k d))
    infixr 3 \triangle
```

Laws:

```
   exl ∘ (f \triangle g) ≡ f
   exr ∘ (f \triangle g) ≡ g
   exl ∘ h \triangle exr ∘ h ≡ h
```
Coproducts

Dual to product.

```
class Category k ⇒ Cocartesian k where
  type a +_k b
  inl :: a `k` (a +_k b)
  inr :: b `k` (a +_k b)
  (\n) :: (a `k` c) → (b `k` c) → ((a +_k b) `k` c)
  infixr 2 \n
Laws:

(f \n g) o inl  ≡ f
(f \n g) o inr  ≡ g
h o inl \n h o inr ≡ h
```
Exponentials

First-class “functions” (morphisms):

```haskell
class Cartesian k ⇒ Closed k where
    type a ⇒k b
    apply :: ((a ⇒k b) ×k a) `k` b
    curry :: ((a ×k b) `k` c) → (a `k` (b ⇒k c))
    uncurry :: (a `k` (b ⇒k c)) → ((a ×k b) `k` c)
```

Laws:

```
uncurry (curry f) ≡ f
curry (uncurry g) ≡ g
apply ○ (curry f ○ exl △ exr) ≡ f
```
class NumCat $k \ a$ where
  negateC :: $a \ ackslash k \ a$
  addC, sub, mulC :: $(a \times_k a) \ ackslash k \ a$

...
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

\[ \text{magSqr} (a, b) = \text{sqr} \ a + \text{sqr} \ b \]

\[ \text{magSqr} = \text{addC} \circ (\text{mulC} \circ (\text{exl} \triangle \text{exl}) \triangle \text{mulC} \circ (\text{exr} \triangle \text{exr})) \]
Computation graphs — example

\[ \text{cosSinProd} (x, y) = (\cos z, \sin z) \textbf{ where } z = x \times y \]

\[ \text{cosSinProd} = (\cos C \triangle \sin C) \circ \text{mulC} \]
Computation graphs — example

\[ \lambda x \ y \rightarrow x + 3 \times y \]

\[ \text{curry} \ (\text{addC} \circ (\text{exl} \triangle \text{mulC} \circ (\text{const} \ 3.0 \triangle \text{exr}))) \]
newtype Graph a b = Graph (Ports a → GraphM (Ports b))

type GraphM = State (PortNum, [Comp])

data Comp = ∀a b. Comp (Template a b) (Ports a) (Ports b)

data Template :: * → * → * where
  Prim :: String → Template a b
  Subgraph :: Graph a b → Template () (a → b)

instance Category Graph where
  id = Graph return
  Graph g ◦ Graph f = Graph (g ◦< f)

instance BoolCat Graph where
  notC = genComp "¬"
  andC = genComp "∧"
  orC = genComp "∨"
Computation graphs — fold

\[
\text{\textit{sum}} :: \text{Tree 4 Int} \rightarrow \text{Int}
\]
Computation graphs — scan

\[ \text{lsums} :: \text{Tree 4 Int} \rightarrow \text{Tree 4 Int} \times \text{Int} \]
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

\[
disk :: \mathbb{R} \rightarrow Region
\]
\[
disk \ r \ p = \text{magSqr} \ p \leq \text{sqr} \ r
\]
\[
\text{woob} \ t = disk (0.75 + 0.25 \ast \cos \ t)
\]

\[
\text{type Region} = \mathbb{R} \times \mathbb{R} \rightarrow \text{Bool}
\]

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 \ast v24 + v26 \ast v26 \leq v17;
}

vec4 effect (vec2 p) { return bw(uwoob(time,p.x,p.y)); }

Conal Elliott (Target)  Teaching new tricks to old programs  May 2017  28 / 38
Automatic differentiation

\[ \textbf{data} \quad D \ a \ b = D \ (a \to b \times (a \to b)) \quad -- \text{Derivatives are linear maps.} \]

\[ \text{linear}D \ f = D \ (\lambda a \to (f \ a, \text{linear} \ f)) \]

\textbf{instance} \ Category \ D \ \textbf{where}

\[ \text{id} = \text{linear}D \ \text{id} \]

\[ D \ g \circ D \ f = D \ (\lambda a \to \text{let} \ ((b, f') = f \ a; (c, g') = g \ b) \ \text{in} \ (c, g' \circ f')) \]

\textbf{instance} \ Cartesian \ D \ \textbf{where}

\[ \text{exl} = \text{linear}D \ \text{exl} \]

\[ \text{exr} = \text{linear}D \ \text{exr} \]

\[ D \ f \triangle D \ g = D \ (\lambda a \to \text{let} \ ((b, f') = f \ a; (c, g') = g \ a) \ \text{in} \ ((b, c), f' \triangle g')) \]

\textbf{instance} \ NumCat \ D \ \textbf{where}

\[ \text{negate}C = \text{linear}D \ \text{negate}C \]

\[ \text{add}C = \text{linear}D \ \text{add}C \]

\[ \text{mul}C = D \ (\text{mul}C \triangle \lambda(a, b) \to \text{linear} \ (\lambda(da, db) \to da \ast b + db \ast a)) \]
Composing interpretations (\textit{Graph} and \textit{D})

\[
\begin{array}{c}
\text{In} \\
\times \\
+ \\
\times \\
+ \\
\times \\
\times \\
\times \\
\times \\
\end{array}
\]

\[
\text{Out}
\]
Composing interpretations \((\text{Graph and } D)\)

\[
\cos \sin \prod
\]

\[
\text{In} \times \cos\sin \times \text{Out}
\]

\[
\text{In} \times \cos \text{Out}
\]

\[
\text{In} \times \cos \times \text{Out}
\]

Conal Elliott (Target)  
Teaching new tricks to old programs  
May 2017  
31 / 38
Interval analysis

data IFun a b = IFun (Interval a → Interval b)

type family Interval a

type instance Interval Double = Double × Double

type instance Interval (a × b) = Interval a × Interval b

type instance Interval (a → b) = Interval a → Interval b

instance Category IFun where
  id = IFun id
  IFun g ∘ IFun f = IFun (g ∘ f)
  ...

instance Cartesian IFun where
  exl = IFun exl
  expr = IFun expr
  IFun f ∘ IFun g = IFun (f ∘ g)

instance (Interval a ~ (a × a), Num a, Ord a) ⇒ NumCat IFun a where
  addC = IFun (λ((a_lo, a_hi), (b_lo, b_hi)) → (a_lo + b_lo, a_hi + b_hi))
  mulC = IFun (λ((a_lo, a_hi), (b_lo, b_hi)) →
              minmax [a_lo * b_lo, a_lo * b_hi, a_hi * b_lo, a_hi * b_hi]
  ...

Conal Elliott (Target)  Teaching new tricks to old programs  May 2017  32 / 38
Interval analysis — example

\[ \lambda(x, y) \rightarrow x + 3 \times y \]
newtype SMT a b = SMT (Kleisli Z3 (E a) (E b))

data E :: * → * where
  PrimE :: AST → E a
  PairE :: E a → E b → E (a × b)

instance Category SMT where
  id = SMT id
  SMT g ∘ SMT f = SMT (g ∘ f)

instance Cartesian SMT where
  exl = SMT (arr (exl ∘ unpairE))
  exr = SMT (arr (exr ∘ unpairE))
  SMT f ∘ SMT g = SMT (arr PairE ∘ (f ∘ g))

instance Num a ⇒ NumCat SMT a where
  negateC = liftE₁ mkUnaryMinus
  addC = liftE₂ mkAdd
  subC = liftE₂ mkSub
  mulC = liftE₂ mkMul
pred :: (Num a, Ord a) ⇒ a × a → Bool
pred (x, y) =
  x < y ∧
  y < 100 ∧
  0 ≤ x − 3 + 7 * y ∧
  (x ≡ y ∨ y + 20 ≡ x + 30)

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*.

- **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