Beyond Polarity

A Multi-Discipline Intermediate Language with Sharing

Paul Downen   Zena M. Ariola

September 4, 2018

Minimalism in Programming Languages

Virtues of Minimalism

Fewer concepts; fewer details

Only the essence remains

Decomposes big ideas into smaller ones

Parts are composable, modular, orthogonal

Benefits both programmers and implementors

- Gives a small but powerful toolset
- Gives a small but powerful core language

E.g., a "universal" intermediate language for both eager and lazy languages

Perils of Minimalism

Deceptively simple

Easy to get close, but difficult to get right

“Obvious” encodings don’t quite work, leaky abstractions

Small incongruences get in the way, break reasoning

End result can be impractical, unfaithful representation

Overview

Logical encodings, and what goes wrong

Polarity in languages, and how it comes to the rescue

Sharing and memoization, and how it can be included

Type isomorphisms, and their application in a compiler

Encodings

Encoding Complex Types

"Programming language paper problem"

Goal: a minimum, finite basis of type constructors

Should be capable of encoding all desired types

All encodings should be faithful

- Represent the complex type exactly
- Everything known about complex type holds for encoding
- Nothing is lost by only using encoding instead

I.o.w., an isomorphism between types and their encodings

CURRYING

$$\begin{aligned} \mathit{curry} &: ((a, b) \to c) \to (a \to b \to c) \\ \mathit{curry}\, f &= \lambda x. \lambda y.\, f(x, y) \end{aligned}$$

$$\begin{aligned} \mathit{uncurry} &: (a \to b \to c) \to ((a, b) \to c) \\ \mathit{uncurry}\, f &= \lambda (x, y).\, f\ x\ y \end{aligned}$$

Is *curry* and *uncurry* an isomorphism between binary functions $(a, b) \to c$ and unary functions $a \to (b \to c)$?

Currying

Consider partial application in CBV λ-calculus

$$
\begin{aligned}
&\mathit{loop} \quad : \mathsf{Int} \to \mathsf{Bool} \to \mathsf{String} \\
&\mathit{loop}\; x = \mathit{loop}\; x
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{what}_1 = \textbf{let}\; g = \mathit{loop}\; 1\; \textbf{in}\; 0 \\
&\mathit{what}_2 = \textbf{let}\; g = (\mathit{curry}\; (\mathit{uncurry}\; \mathit{loop}))\; 1\; \textbf{in}\; 0
\end{aligned}
$$

Currying — Oops

Consider partial application in CBV λ-calculus

$$\begin{aligned} &loop \quad : \mathsf{Int} \to \mathsf{Bool} \to \mathsf{String} \\ &loop\ x = loop\ x \end{aligned}$$

$$\begin{aligned} what_1 &= \mathbf{let}\ g = loop\ 1\ \mathbf{in}\ 0 \\ what_2 &= \mathbf{let}\ g = (curry\ (uncurry\ loop))\ 1\ \mathbf{in}\ 0 \end{aligned}$$

$what_1$ diverges because $loop\ 1$ diverges

$what_2 = 0$ because $(curry\ (uncurry\ loop))\ 1 = \lambda y.loop\ 1\ y$

So currying is not an isomorphism in CBV

The culprit?

Currying — Oops

Consider partial application in CBV λ-calculus

$$\begin{aligned} &\mathit{loop} \quad : \mathsf{Int} \to \mathsf{Bool} \to \mathsf{String} \\ &\mathit{loop}\ x = \mathit{loop}\ x \end{aligned}$$

$$\begin{aligned} &\mathit{what}_1 = \mathbf{let}\, g = \mathit{loop}\ 1\ \mathbf{in}\ 0 \\ &\mathit{what}_2 = \mathbf{let}\, g = (\mathit{curry}\ (\mathit{uncurry}\ \mathit{loop}))\ 1\ \mathbf{in}\ 0 \end{aligned}$$

what_1 diverges because loop 1 diverges

$\mathit{what}_2 = 0$ because $(\mathit{curry}\ (\mathit{uncurry}\ \mathit{loop}))\ 1 = \lambda y.\mathit{loop}\ 1\ y$

So currying is not an isomorphism in CBV

The culprit? Eagerness

Nested Pairs

$$\begin{aligned} nest &: (a, b, c) \to (a, (b, c)) \\ nest\ (x, y, z) &= (x, (y, z)) \end{aligned}$$

$$\begin{aligned} unnest &: (a, (b, c)) \to (a, b, c) \\ unnest\ (x, (y, z)) &= (x, y, z) \end{aligned}$$

Is *nest* and *unnest* an isomorphism between triples (a, b, c) and nested pairs $(a, (b, c))$?

Nested Pairs

Consider pattern matching in CBN λ-calculus

$$\mathit{undefined} = \mathit{undefined}$$

$$\mathit{partial} : (\mathsf{Int}, (\mathsf{Bool}, \mathsf{String}))$$
$$\mathit{partial} = (0, \mathit{undefined})$$

$$\mathit{what}_1 = \mathbf{case}\ \mathit{partial}\ \mathbf{of}\ (x, y) \to x$$
$$\mathit{what}_2 = \mathbf{case}\ \mathit{nest}\ (\mathit{unnest}\ \mathit{partial})\ \mathbf{of}\ (x, y) \to x$$

Nested Pairs — Oops

Consider pattern matching in CBN λ-calculus

$$undefined = undefined$$

$$partial : (\text{Int}, (\text{Bool}, \text{String}))$$

$$partial = (0, undefined)$$

$$what_1 = \textbf{case}\ partial\ \textbf{of}\ (x, y) \to x$$

$$what_2 = \textbf{case}\ nest\ (unnest\ partial)\ \textbf{of}\ (x, y) \to x$$

$what_1$ returns 0

$what_2$ diverges because $nest\ (unnest\ partial)$ does

So nesting pairs is not an isomorphism in CBN

The culprit?

Nested Pairs — Oops

Consider pattern matching in CBN λ-calculus

$$undefined = undefined$$

$$partial : (\text{Int}, (\text{Bool}, \text{String}))$$

$$partial = (0, undefined)$$

$$what_1 = \textbf{case}\ partial\ \textbf{of}\ (x, y) \rightarrow x$$

$$what_2 = \textbf{case}\ nest\ (unnest\ partial)\ \textbf{of}\ (x, y) \rightarrow x$$

$what_1$ returns 0

$what_2$ diverges because $nest\ (unnest\ partial)$ does

So nesting pairs is not an isomorphism in CBN

The culprit? Laziness

Possible fix with even more laziness, but…

Nested Sums

$$\textbf{data}\ \mathrm{Either}\ a\ b = \mathsf{L}\ a \mid \mathsf{R}\ b$$

$$\textbf{data}\ \mathrm{Either3}\ a\ b\ c = \mathrm{Choice1}\ a \mid \mathrm{Choice2}\ b \mid \mathrm{Choice3}\ c$$

$$\begin{aligned} &\mathit{nest}\ (\mathrm{Choice1}\ x) = \mathsf{L}\ x & \qquad &\mathit{unnest}\ (\mathsf{L}\ x) &&= \mathrm{Choice1}\ x \\ &\mathit{nest}\ (\mathrm{Choice2}\ y) = \mathsf{R}\ (\mathsf{L}\ y) & &\mathit{unnest}\ (\mathsf{R}\ (\mathsf{L}\ y)) &&= \mathrm{Choice2}\ y \\ &\mathit{nest}\ (\mathrm{Choice3}\ z) = \mathsf{R}\ (\mathsf{R}\ z) & &\mathit{unnest}\ (\mathsf{R}\ (\mathsf{R}\ z)) &&= \mathrm{Choice3}\ z \end{aligned}$$

Is *nest* and *unnest* an isomorphism between ternary sums Either3 $a\ b\ c$ and binary sums Either a (Either $b\ c$)?

Nested Sums — Oops

$$\textbf{data}\ \mathrm{Either}\ a\ b = \mathsf{L}\ a \mid \mathsf{R}\ b$$

$$\textbf{data}\ \mathrm{Either3}\ a\ b\ c = \mathrm{Choice1}\ a \mid \mathrm{Choice2}\ b \mid \mathrm{Choice3}\ c$$

$$\begin{aligned}
&\mathit{nest}\ (\mathrm{Choice1}\ x) = \mathsf{L}\ x && \mathit{unnest}\ (\mathsf{L}\ x) &&= \mathrm{Choice1}\ x \\
&\mathit{nest}\ (\mathrm{Choice2}\ y) = \mathsf{R}\ (\mathsf{L}\ y) && \mathit{unnest}\ (\mathsf{R}\ (\mathsf{L}\ y)) &&= \mathrm{Choice2}\ y \\
&\mathit{nest}\ (\mathrm{Choice3}\ z) = \mathsf{R}\ (\mathsf{R}\ z) && \mathit{unnest}\ (\mathsf{R}\ (\mathsf{R}\ z)) &&= \mathrm{Choice3}\ z
\end{aligned}$$

Is *nest* and *unnest* an isomorphism between ternary sums $\mathrm{Either3}\ a\ b\ c$ and binary sums $\mathrm{Either}\ a\ (\mathrm{Either}\ b\ c)$?

Not in CBN, for the same reason as before; consider:

$$\mathit{what}_1 = \textbf{case}\ \mathsf{R}\ \mathit{undefined}\ \textbf{of}\ \mathsf{L}\ x \to x; \mathsf{R}\ y \to 0$$

$$\mathit{what}_2 = \textbf{case}\ \mathit{nest}\ (\mathit{unnest}\ (\mathsf{R}\ \mathit{undefined}))\ \textbf{of}\ \mathsf{L}\ x \to x; \mathsf{R}\ y \to 0$$

Connectives and Evaluation are Connected

We've seen several encodings that <u>should</u> work but <u>don't</u>

Culprit: wrong evaluation strategy (CBV vs CBN)

Each type connective has a strategy it works "best" in

They don't all agree, so someone has to be unhappy

Idea: a heterogenous language where each connective uses its "favorite" strategy

Other connective-strategy options can still be recovered

Polarity

A (Very) Brief History of Polarity

First in logic (Andreoli 1992, Girard 1991); specifies efficient proof search among other reasons

Rediscovered in computation (Levy 2001 "call-by-push-value"); decompose denotational semantics, combine functional and imperative

Later, both the logic and computation were connected (Zeilberger 2008, Munch-Maccagnoni 2009)

Types and Evaluation Order

Two different kinds of types, $+$ and $-$

Evaluation order connected to these two kinds:

$M : A : +$ means M is call-by-value

$M : A : -$ means M is call-by-name

a.k.a. $A : +$ is a <u>value type</u> and $B : -$ a <u>computation type</u>

Connectives are given their "best-case scenario"

Positive Sums

binary $\oplus : + \to + \to +$

nullary $0 : +$

Positive Pairs

binary $\otimes : + \to + \to +$

nullary $1 : +$

Negative Products

binary $\& : - \to - \to -$

nullary $\top : -$

Polar Functions

$(\to) : + \to - \to -$

Strong Type Isomorphisms

The counter examples to logical encodings are now gone

Can faithfully represent complex types

Have many strong type isomorphisms, like associativity:

$$(A \otimes B) \to C \approx A \to (B \to C)$$

$$(A \oplus B) \oplus C \approx A \oplus (B \oplus C)$$

$$(A \otimes B) \otimes C \approx A \otimes (B \otimes C)$$

$$(A \,\&\, B) \,\&\, C \approx A \,\&\, (B \,\&\, C)$$

$$\dots$$

The Missing Ingredient: Polarity Shift

But as is, this language is <u>incredibly</u> weak!

Doesn't even have the identity function type $A \to A$

> If $A : +$, then return type wrong, should be negative
>
> If $A : -$, then input type wrong, should be positive

Need a way to shift polarity between positive/negative

Identity function must use a shift to assign calling convention

> Delayed input $A : -$, call-by-name: $\downarrow A \to A$
>
> Strict output $A : +$, call-by-value: $A \to \uparrow A$

Data versus Co-data

Connectives can be either data or co-data (Zeilberger 2009)

Data types are defined by what their values look like

Sums ($A \oplus B$) are data; values are left/right injections

Tuples ($A \otimes B$) are data; values are pairs of values

Co-data types are defined by their interface

Products ($A \,\&\, B$) are co-data; with first/second projections

Functions ($A \to B$) are co-data; objects follow the call-return interface

Think: foreign functions are still functions, even though they don't look like λs.

Two Ways to Shift

There are two different descriptions of polarity shifts based on the data/co-data distinction

Zeilberger (2008)

Negative-to-positive shift $\downarrow : - \to +$ is data

Positive-to-negative shift $\uparrow : + \to -$ is co-data

Levy (2001)

Negative-to-positive shift $\Downarrow : - \to +$ is co-data

Positive-to-negative shift $\Uparrow : + \to -$ is data

Turns out two views are isomorphic, for shifts between $+$ and $-$

Polarized Encodings

Call-by-value sums and functions, $[\![A]\!]^+ : +$

$$[\![A \oplus B]\!]^+ = [\![A]\!]^+ \oplus [\![B]\!]^+$$
$$[\![A \to B]\!]^+ = \Downarrow([\![A]\!]^+ \to \uparrow[\![B]\!]^+)$$

Call-by-name sums and functions, $[\![A]\!]^- : -$

$$[\![A \oplus B]\!]^- = \Uparrow(\downarrow[\![A]\!]^- \oplus \downarrow[\![B]\!]^-)$$
$$[\![A \to B]\!]^- = \downarrow[\![A]\!]^- \to [\![B]\!]^-$$

Note, shifts show where type isomorphisms are intentionally broken

Beyond Polarity: Multi-disciplinary Computation

Polarity mixes both CBV and CBN computation

Brings out the best of every connective

- Both data types (like sums) and co-data types (like functions) are fully extensional

But by definition, it's binary; leaving out other possibilities

Idea: go beyond polarity, with as many disciplines (i.e., calling conventions) as you want

Sharing

Another Kind of Computation: Call-by-Need

Like call-by-name, results are only computed on-demand

Like call-by-value, results are only computed once

New meaning for a binding

$$\textbf{let}\, x = M\, \textbf{in}\, N$$

N is computed first, but any work done to compute M is shared throughout N (a.k.a. memoization)

The Extent of Sharing

Work is shared, but values can be copied

Sharing is preserved by data constructors

A tuple is a value only when its components are

A sum injection is a value only when its payload is

Sharing is ended by co-data objects

Any λ is a value

Any product is a value

Extending the Language: The Extra Shifts

Denote call-by-need with a third kind of type, $\star$

Need shifts between new kind ($\star$) and old ($+$ and $-$)

Shifts must correctly model the extent of sharing

Shifts between $\star$ and $+$ are data types; preserve sharing

$$\downarrow_{\star} : \star \to +$$

$$_{\star}\Uparrow : + \to \star$$

Shifts between $\star$ and $-$ are co-data types; end sharing

$$\uparrow_{\star} : \star \to -$$

$$_{\star}\Downarrow : - \to \star$$

Extending the Language: What Else?

That's it!

Four extra shifts is all that's needed to extend polarity with call-by-need

Large collection of user-defined types faithfully encoded with just shifts and polarized connectives

- <u>All</u> algebraic data types
- Also user-defined co-data types; generalizes functions and products

Polarized Encodings of Sharing

Call-by-value sums and functions, $\llbracket A \rrbracket^{+} : +$

$$\llbracket A \oplus B \rrbracket^{+} = \llbracket A \rrbracket^{+} \oplus \llbracket B \rrbracket^{+}$$
$$\llbracket A \to B \rrbracket^{+} = \Downarrow(\llbracket A \rrbracket^{+} \to \uparrow\llbracket B \rrbracket^{+})$$

Call-by-name sums and functions, $\llbracket A \rrbracket^{-} : -$

$$\llbracket A \oplus B \rrbracket^{-} = \Uparrow(\downarrow\llbracket A \rrbracket^{-} \oplus \downarrow\llbracket B \rrbracket^{-})$$
$$\llbracket A \to B \rrbracket^{-} = \downarrow\llbracket A \rrbracket^{-} \to \llbracket B \rrbracket^{-}$$

Call-by-need sums and functions, $\llbracket A \rrbracket^{\star} : \star$

$$\llbracket A \oplus B \rrbracket^{\star} = {}_{\star}\Uparrow(\downarrow_{\star}\llbracket A \rrbracket^{\star} \oplus \downarrow_{\star}\llbracket B \rrbracket^{\star})$$
$$\llbracket A \to B \rrbracket^{\star} = {}_{\star}\Downarrow(\downarrow_{\star}\llbracket A \rrbracket^{\star} \to \uparrow_{\star}\llbracket B \rrbracket^{\star})$$

Isomorphisms

Translation Isn't Enough

Can encode types with polar connectives (e.g., in a compiler)

Running the program is the "same"; but that's not enough!

Remember the counter-examples (currying, nesting)

Encoding should be robust, not a leaky abstraction

Every fact that holds before encoding must hold after

There's a reason compiler optimize programs in an intermediate core language, not assembly

Type Isomorphisms

Definition (Isomorphism)
$A \approx B : \mathcal{S}$ (for $\mathcal{S}$ ranging over $+$, $-$, and $\star$) when there are terms $x{:}A \vdash N : B$ and $y{:}B \vdash M : A$ such that

$$y{:}B \vdash (\mathbf{let}\, x = M\, \mathbf{in}\, N) = y : B \qquad x{:}A \vdash (\mathbf{let}\, y = N\, \mathbf{in}\, M) = x : A$$

Note that syntactically-defined equational theory used

Ensures that simple program rewrites can justify isomorphism

Can be implemented in an optimizing compiler

Theorem
For any $A : +$ and $B : -$, both $\uparrow A \approx \Uparrow A$ and $\downarrow B \approx \Downarrow B$.

Isomorphism-based Worker/Wrapper

Lemma (Worker/Wrapper)
If $A \approx B : \mathcal{S}$ then there are contexts C, C' such that, for any $\Gamma \vdash M : A$ and $\Gamma \vdash N : B$, we have $\Gamma \vdash C[N] : A$ and $\Gamma \vdash C'[M] : B$ and:

$$\Gamma \vdash C'[C[N]] = N : B \qquad\qquad \Gamma \vdash C[C'[M]] = M : A$$

Relies on reassociation of like-kinded bindings:

$$\begin{array}{ll} \textbf{let}\, y{:}B = (\textbf{let}\, x{:}A = M\, \textbf{in}\, N)\, \textbf{in}\, P & \\ \qquad\qquad\qquad = & (\text{if } A{:}\mathcal{S} \text{ and } B{:}\mathcal{S} \text{ for some } \mathcal{S}) \\ \textbf{let}\, x{:}A = M\, \textbf{in}\, (\textbf{let}\, y{:}B = N\, \textbf{in}\, P) & \end{array}$$

Corollary
If $A \approx B$ then terms of type A and B are in equational correspondence.

Encoding User-defined Types

Every user-defined data and co-data type constructor F can be encoded into only polarized connectives, $[\![\mathrm{F}]\!]$

Extend this encoding to full types, $[\![A]\!]$, homomorphically

> This enables a worker/wrapper-style of local transformation

Likewise encode terms M using any user-defined (co-)data types into one using only polarized types, $[\![M]\!]$

> This enables a complete form of global translation

Faithfulness of the Encoding

Theorem (Encoding Isomorphism)
For every A, we have $A \approx [\![A]\!]$.

Corollary (Local Correspondence)
Terms of type A and $[\![A]\!]$ are in equational correspondence.

Theorem (Global Correspondence)
$\Gamma \vdash M = N : A$ *if and only if* $[\![\Gamma]\!] \vdash [\![M]\!] = [\![N]\!] : [\![A]\!]$.

Corollary (Intermediate Language)
The core polarized language (with $+$, $-$, and $\star$) is in equational correspondence with its extension with user-defined (co-)data types.

Conclusion

More in the Paper

Polymorphism and type functions (a.k.a. system F_ω)

Interesting consequences for type isomorphism

Computational effects (divergence and first-class control)

A multi-discipline equational theory; conservative extension of call-by-push-value

Restoring the missing duality (appendix)

Restoring the duality

Based on sequent calculus (Curien & Herbelin 2000)

Dual to call-by-need: sharing control vs sharing information

Fully dual data and co-data types

Fully dual polar basis of primitive connectives à la linear logic

- No function type $\rightarrow$
- Negative disjunction $\parr$ and $\perp$
- Involutive pair of positive $\ominus$ and negative $\neg$ negations (Munch-Maccagnoni & Scherer 2015)

Common algebraic and logical laws as type isomorphisms

- Two dual commutative semirings from positive and negative conjunction/disjunction
- Two dual sets of De Morgan laws

Future Work

Connections with unboxed values and types in GHC (Peyton Jones & Launchbury 1991)

GHC core intermediate language has types that distinguish

- Lazy evaluation (ordinary Haskell values)
- Eager evaluation, (unboxed values, e.g., machine integers)

Perhaps the first implementation of a multi-discipline language

Idea: see if remaining polar connectives are useful for compilers

Potential application with curried function call arity

Summary

Minimalism is desirable, but requires care

Different types have different needs to bring out their best

Diversity of computation, rather than conformity, is a virtue

Multi-discipline evaluation goes beyond binary polarity

Sharing is possible, with just some shifts