Posit: A Core Calculus for Computing with Positive Types

Chris Martens

January 20, 2025

1 Introduction

This development explores the question of what happens if we strictly separate positive data from functions, adopting a "first-order" treatment of data transformations rather than functions-in-general.

Positive types characterize observable data:

$A^+, B^+ \quad ::= \quad p^+ \mid \mathbf{1} \mid A^+ \times B^+ \mid A^+ + B^+$

We refuse to pollute these types even with *suspended* negative propositions (as one might find in, say, CBPV [3]).

As a consequence, instead of having a general function space, we create a class of terms called *transformations*[1] from positive types A to B, which transform values of type A to values of type B. Transformations have syntactic forms for every positive type connective corresponding to left rules in sequent calculus. So for example, a transformation from $A \times B$ to C is

$$\mathsf{split}(x.y.\, e)$$

where e computes a value of type C when given values of type A and B to substitute for x and y. Note that unlike the elimination form in STLC/natural deduction, we don't supply the "scrutinee" of the split as part of its syntax: we just describe what it should do when it encounters a value of the appropriate type. A *computation* is a value of type A^+ together with a transformation from A^+ to some C^+.

One might note that this precludes higher-order functions—*in general*. The spoiler is that we'll be able to recover some particular higher-order function schema by way of inductive type constructions and their well-founded recursors. In the meantime, we might try and see how much we can get away with *without* including arbitrary higher-order functions in our language.

[1] TODO: decide whether to call them transformations or transformers.

2 Finite Types

2.1 Values, Transformations, and Computations

Our syntactic categories are:

Value types	A^+, B^+	$::=$	$p^+ \mid \mathbf{1} \mid A^+ \times B^+ \mid A^+ + B^+$
Values	v	$::=$	$() \mid (v_1, v_2) \mid \mathsf{in}_1\, v \mid \mathsf{in}_2\, v \mid x$
Transformations	δ	$::=$	$\mathsf{ignore}(e) \mid \mathsf{split}(x.y.\, e) \mid \mathsf{case}(x.\, e_1, y.\, e_2) \mid \mathsf{id}_{A^+}$
Computations	e	$::=$	$v \triangleright \delta \mid \{v\} \mid \mathsf{bind}(e; x.e')$
Contexts	Γ	$::=$	$\cdot \mid x{:}A^+, \Gamma$

Typing implicitly occurs in contexts $\Gamma \vdash J$ (where J is the typing judgment written explicitly); the turnstile is only written when the context Γ is referenced.

Evidence typing $v : A^+$ corresponds to right rules in sequent calculus, or introduction rules in natural deduction.

$$\frac{}{x{:}A^+ \vdash x : A^+}\ \mathsf{var} \qquad \frac{}{() : \mathbf{1}}\ \mathbf{1}R \qquad \frac{v_1 : A^+ \quad v_2 : B^+}{(v_1, v_2) : A^+ \times B^+}\ \times R$$

$$\frac{v : A^+}{\mathsf{in}_1\, v : A^+ + B^+}\ +R_1 \qquad \frac{v : B^+}{\mathsf{in}_2\, v : A^+ + B^+}\ +R_2$$

Transformation typing $\delta : A^+ \to C^+$ corresponds to left rules in sequent calculus, using a syntax meant to evoke elimination rules in natural deduction. Note that $\to$ here is part of the *typing judgment* for transformations, and that we do not have function types in the syntax in general. Likewise, we have computation typing $e \div C^+$ to denote e as an expression that "eventually computes" a value of type C^+ (as a distinct notion from $v : A^+$ meaning v *is* a value of the appropriate type).

$$\frac{}{\mathsf{id} : A^+ \to A^+}\ \mathsf{id} \qquad \frac{e \div C^+}{\mathsf{ignore}(e) : \mathbf{1} \to C^+}\ \mathbf{1}L$$

$$\frac{x{:}A^+, y{:}B^+ \vdash e \div C^+}{\mathsf{split}(x.y.\, e) : A^+ \times B^+ \to C^+}\ \times L \qquad \frac{x{:}A^+ \vdash e_1 \div C^+ \quad y{:}B^+ \vdash e_2 \div C^+}{\mathsf{case}(x.\, e_1, y.\, e_2) : A^+ + B^+ \to C^+}\ +L$$

$$\frac{v : A^+ \quad \delta : A^+ \to C^+}{v \triangleright \delta \div C^+}\ \mathsf{comp/apply} \qquad \frac{v : A^+}{\{v\} \div A^+}\ \mathsf{comp/return}$$

$$\frac{e \div A^+ \quad x{:}A^+ \vdash e' \div C^+}{\mathsf{bind}(e; x.\, e') \div C^+}\ \mathsf{comp/bind}$$

2.2 Evaluating Computations

Computations are things that we can run, so let's describe how to run them.

$e \Downarrow v$:

$$\frac{}{\{v\} \Downarrow v}\ \textsf{eval/ret} \qquad \frac{e \Downarrow v \quad [v/x]e' \Downarrow v'}{\textsf{bind}(e; x.\, e') \Downarrow v'}\ \textsf{eval/bind}$$

$$\frac{}{v \triangleright \textsf{id} \Downarrow v}\ \textsf{eval/id} \qquad \frac{e \Downarrow v}{() \triangleright \textsf{ignore}(e) \Downarrow v}\ \textsf{eval/ignore}$$

$$\frac{[v_1/x, v_2/y]e \Downarrow v}{(v_1, v_2) \triangleright \textsf{split}(x.y.\, e) \Downarrow v}\ \textsf{eval/split}$$

$$\frac{[v/x]e_1 \Downarrow v'}{\textsf{in}_1\, v \triangleright \textsf{case}(x.\, e_1, y.\, e_2) \Downarrow v'}\ \textsf{eval/inl} \qquad \frac{[v/y]e_2 \Downarrow v'}{\textsf{in}_2\, v \triangleright \textsf{case}(x.\, e_1, y.\, e_2) \Downarrow v'}\ \textsf{eval/inr}$$

2.3 Example

Here's an evidence transformer distrib that witnesses $\times$ distributing over $+$:

$$\textsf{distrib} : A \times (B + C) \to (A \times B) + (A \times C)$$

$$\textsf{distrib} = \textsf{split}(x.y.\, y \triangleright \textsf{case}(z.\, \{\textsf{in}_1\,(x, z)\}, w.\, \{\textsf{in}_2\,(x, w)\}))$$

Here's a value that can be transformed by it (assuming base values $a : A$ etc.):

$$\textsf{input} : A \times (B + C)$$

$$\textsf{input} = (a, \textsf{in}_1\, b)$$

We can check that the computation $\textsf{input} \triangleright \textsf{distrib}$ evaluates to $\textsf{in}_1\,(a, b)$ as expected.

3 Inductive Types

We now add recursive definitions to the language, starting with inductive types (μ). To do this we need a notion of type functor $\alpha.A^+$, where α stands for another value type.

3.1 Statics

Value types	A^+, B^+	$::=$	$\cdots \mid \alpha \mid \mu\alpha.\, A^+$
Values	v	$::=$	$\cdots \mid \textsf{fold}(v)$
Transformations	δ	$::=$	$\cdots \mid \textsf{rec}_{\{\alpha.A^+\}}(x.\, e)$

Computations and contexts are as before.

Typing:

$$\frac{v : [\mu\alpha.A^+/\alpha]A^+}{\textsf{fold}(v) : \mu\alpha.A^+}\ \mu R$$

$$\frac{x : [C^+/\alpha]A^+ \vdash e \div C^+}{\mathsf{rec}_{\{\alpha.A^+\}}(x.\, e) : (\mu\alpha.A^+) \to C^+}\ \mu L$$

3.2 Evaluation

Per PFPL [2], we can define evaluation with a meta-level map on transformations, defined in a type-directed way according to the structure of the polynomial functor describing the shape we are mapping over. map over a polynomial $\alpha.\, C^+$ takes transformations of type $A^+ \to B^+$ to transformations of type $[A^+/\alpha]C^+ \to [B^+/\alpha]C^+$, and is defined inductively on the structure of the polynomial. We define map in the next subsection.

Computation rules for inductive types:

$$\frac{\mathsf{map}_{\{\alpha.A^+\}}(\mathsf{rec}_{\{\alpha.A^+\}}(x.e)) = \delta \quad v \triangleright \delta \Downarrow v' \quad [v'/x]e \Downarrow v''}{\mathsf{fold}(v) \triangleright \mathsf{rec}_{\{\alpha.A^+\}}(e) \Downarrow v''}\ \mathsf{eval/rec}$$

3.3 Defining map

TODO add remaining rules. This definition is adapted directly from PFPL ([2]).

$$\frac{}{\mathsf{map}_{\{\alpha.\alpha\}}(\delta) = \delta}\ \mathsf{map/id} \qquad \frac{\alpha \notin C^+}{\mathsf{map}_{\{\alpha.\, C^+\}}(\delta) = \mathsf{id}_{C^+}}\ \mathsf{map/const}$$

$$\frac{\mathsf{map}_{\{\alpha.A^+\}}(\delta) = \delta_1 \quad \mathsf{map}_{\{\alpha.B^+\}}(\delta) = \delta_2}{\mathsf{map}_{\{\alpha.A^+ \times B^+\}}(\delta) = \mathsf{split}(x.y.\, \mathsf{bind}(x \triangleright \delta_1; z.\, \mathsf{bind}(y \triangleright \delta_2; w.\, \{(z, w)\})))}\ \mathsf{map}/\times$$

According to the Containers paper [1], we should expect to be able to extend this definition (and therefore our definition of positive types) to inductive and coinductive types.

References

[1] M. Abbott, T. Altenkirch, and N. Ghani. Containers: Constructing strictly positive types. *Theoretical Computer Science*, 342(1):3–27, 2005.

[2] R. Harper. *Practical foundations for programming languages.* Cambridge University Press, 2016.

[3] P. B. Levy. Call-by-push-value: A subsuming paradigm. In *International Conference on Typed Lambda Calculi and Applications*, pages 228–243. Springer, 1999.