Normalization by evaluation for call-by-push-value and polarized lambda calculus

Citation for the original published paper (version of record):

Abel, A., Sattler, C. (2019). Normalization by evaluation for call-by-push-value and polarized lambda calculus. ACM International Conference Proceeding Series.
http://dx.doi.org/10.1145/3354166.3354168

Normalization by Evaluation for Call-By-Push-Value and Polarized Lambda Calculus

Andreas Abel
Department of Computer Science and Engineering, Chalmers and Gothenburg University
Göteborg, Sweden

Christian Sattler
School for Computer Science, University of Nottingham
Nottingham, United Kingdom

ABSTRACT

We observe that normalization by evaluation for simply-typed lambda-calculus with weak coproducts can be carried out in a weak bi-cartesian closed category of presheaves equipped with a monad that allows us to perform case distinction on neutral terms of sum type. The placement of the monad influences the normal forms we obtain: for instance, placing the monad on coproducts gives us eta-long beta-pi normal forms where pi refers to permutation of case distinctions out of elimination positions. We further observe that placing the monad on every coproduct is rather wasteful, and an optimal placement of the monad can be determined by considering polarized simple types inspired by focalization. Polarization classifies types into positive and negative, and it is sufficient to place the monad at the embedding of positive types into negative ones. We consider two calculi based on polarized types: pure call-by-push-value (CBPV) and polarized lambda-calculus, the natural deduction calculus corresponding to focalized sequent calculus. For these two calculi, we present algorithms for normalization by evaluation. We further discuss different implementations of the monad and their relation to existing normalization proofs for lambda-calculus with sums. Our developments have been partially formalized in the Agda proof assistant.

CCS CONCEPTS

• **Theory of computation** → **Type theory**; **Type structures**; *Functional constructs*; *Proof theory*; *Categorical semantics*; Operational semantics.

KEYWORDS

Evaluation, Intuitionistic Propositional Logic, Lambda-Calculus, Monad, Normalization, Polarized Logic, Semantics

ACM Reference Format:
Andreas Abel and Christian Sattler. 2019. Normalization by Evaluation for Call-By-Push-Value and Polarized Lambda Calculus. In *PPDP 2019: 21st International Symposium on Principles and Practice of Declarative Programming, 7-9 October 2019, Porto, Portugal*. ACM, New York, NY, USA, 12 pages.

PPDP 2019, 7-9 October 2019, Porto, Portugal

ACM ISBN 978-1-4503-9999-9/18/06…$0.00
DOI: 10.1145/3354166.3354168

1 INTRODUCTION

The idea behind *normalization by evaluation* (NbE) is to utilize a standard interpreter that evaluates closed terms to compute the normal form of an open term, i. e., a term that may contain free variables. The normal form is obtained by a type-directed *reification* procedure after evaluating the open term to a semantic value, mapping (*reflecting*) the free variables to corresponding *unknowns* in the semantics. The literal use of a standard interpreter can be achieved for the pure simply-typed lambda-calculus [9, 14] by modelling uninterpreted base types as sets of neutral (aka atomic) terms. More precisely, base types are interpreted as presheaves, or sets of neutral term families, in order to facilitate the generation of fresh variables of base type during reification of functions to lambdas. Thanks to η-equality at function types, free variables of *function* type can be reflected into the semantics as functions applying the variable to their reified argument, forming a neutral term. This mechanism provides us with unknowns of base and function type which can be faithfully reified to normal forms.

For the extension to *sum* types (logically, disjunctions, and categorically, weak coproducts), this reflection trick does no longer work. A semantic value of binary sum type is either a left or a right injection, but the decision between left or right cannot be taken at reflection time, since a variable of sum type does not provide us with such information. A literal standard interpreter for closed terms can thus no longer be used for NbE; instead, we can utilize a monadic interpreter. When the interpreter attempts a case distinction on an unknown of sum type, it asks an oracle whether the unknown is a left or a right injection. The oracle returns one of these alternatives, wrapping a new unknown in the respective injection. The communication with the oracle can be modeled in a *monad* C, which records the questions asked and the continuation of the interpreter for each of the possible answers. A monadic semantic value is thus a *case tree* where the inner nodes are labeled with unknowns and the leaves with non-monadic values [5].

In this article, we only consider weak sum types, lacking the universal property of coproducts. As a consequence, terms with the same denotation may have different normal forms under NbE. In particular, case trees are not normalized, allowing, e. g., redundant case splits (i. e., asking the same question twice). Further, the order of case splits is not normalized: changing the order in which the questions are asked (commuting case splits), alters the normal form. The model would need refinement for extensional sums (strong coproducts) with unique normal forms [3, 5, 7, 8, 28], i. e., where NbE decides equality of denotations.

Filinski [15] studied NbE for Moggi's computational lambda calculus [23], shedding light on the difference between call-by-name (CBN) and call-by-value (CBV) NbE, where Danvy's type-directed partial evaluation [13] falls into the latter class. The contribution of the computational lambda calculus is to make explicit where the monad is invoked during monadic evaluation, and this placement of the monad carries over to the NbE setting. Moggi's studies were continued by Levy [19] who designed the call-by-push-value (CBPV) lambda-calculus to embed both the CBN and CBV lambda calculus. Equational theories and reduction-based normalization for CBPV were recently studied and formalized in Coq [16, 26]

In this work, we formulate NbE for CBPV (Section 3). In contrast to the normal forms of CBN NbE—which is the algorithmic counterpart of the completeness proof for intuitionistic propositional logic (IPL) using Beth models—CBPV NbE gives us more restrained normal forms, where the production of a value via injections cannot be interrupted by more questions to the oracle. In the research field of focalization [6, 20] we speak of *chaining non-invertible introductions*. Invertible introductions are already chained in NbE thanks to extensionality (η) for function, and more generally, negative types. Non-invertible eliminations are also happening in a chain when building neutrals. What is missing from the picture is the chaining of invertible eliminations, i.e., case distinctions and, more generally, pattern matching. The picture is completed by extending NbE to *polarized lambda calculus* [11, 27, 29] in Section 4.

In our presentation of the various lambda calculi we ignore the concrete syntax, only considering the abstract syntax obtained by the Curry-Howard-Isomorphism. A term is simply a derivation tree whose nodes are rule invocations. Thus, an intrinsically typed, nameless syntax is most natural, and our syntactic classes are all presheaves over the category of typing contexts and renamings. The use of presheaves then smoothly extents to the semantic constructions [4, 12].

Concerning the presentation of polarized lambda calculus, we depart from Zeilberger [29] who employs *a priori* infinitary syntax, modelling a case tree as a meta-level function mapping well-typed patterns to branches. Instead, we use a graded monad representing complete pattern matching over a newly added hypothesis, which is in spirit akin to Filinski's [15, Section 4] and Krishnaswami's [18] treatment of eager pattern matching using a separate context of variables to be matched on.

Our design choices were guided by an Agda formalization of sections 2 (complete, see https://andreasabel.github.io/ipl/html/NfModelMonad.html), 3 (in a variantion, see https://andreasabel.github.io/ipl/html-cbpv/NfCBPV.html) and 4 (partial, see https://andreasabel.github.io/ipl/html-focusing/Polarized.html). Agda was particularly helpful to correctly handle the renamings abundantly present when working with presheaves.

To summarize, in this article we make the following contributions:

- We retell the story of normalization by evaluation (NbE) for intrinsically typed lambda terms with weak sum through a generic monadic interpreter. We isolate the services the monad needs to offer in order to facilitate reification.
- We observe that the placement of the monad in the process of the type interpretation can be determined by polarization, i. e., separating types into positive and negative types. The most economic placement of the monad is achieved with deep polarization such as in focused calculi.
- As a first focused calculus, we consider call-by-push-value (CBPV) which identifies positive types with value types and negative types with computation types. To our best knowledge, we are the first to study NbE for pure, effect-free CBPV. Therein, we observe that the structures of CBPV to control effects naturally structure the NbE process as well.
- Finally, we define NbE for a fully focused calculus, the polarized lambda calculus. We use a factored presentation of terms and normal forms, which uses advanced features of Agda (nested and sized types) in our partial formalization.

2 NORMALIZATION BY EVALUATION FOR THE SIMPLY-TYPED LAMBDA CALCULUS WITH SUMS

In this section, we review the normalization by evaluation (NbE) argument for the simply-typed lambda calculus (STLC) with weak sums, setting the stage for the later sections. We work in a constructive type-theoretic meta-language, with the basic judgement $t : T$ meaning that object t is an inhabitant of type T. However, to avoid confusion with object-level types such as the simple types of lambda calculus, we will refer to meta-level types as *sets*. Consequently, the colon : takes the role of elementhood $\in$ in set theory, and we are free to reuse the symbol $\in$ for other purposes.

2.1 Contexts and indices

We adapt a categorical aka de Bruijn style for the abstract syntax of terms, which we conceive as intrinsically well-typed. In de Bruijn style, a context Γ is just a snoc list of simple types A, meaning we write context extension as $\Gamma.A$, and the empty context as ε. Membership $\boxed{A \in \Gamma}$ and sublist relations $\boxed{\Gamma \subseteq \Delta}$ are given inductively by the following rules:

$$\mathsf{zero}\ \frac{}{A \in \Gamma.A} \qquad \mathsf{suc}\ \frac{A \in \Gamma}{A \in \Gamma.B}$$

$$\varepsilon\ \frac{}{\varepsilon \subseteq \varepsilon} \qquad \mathsf{lift}\ \frac{\Gamma \subseteq \Delta}{\Gamma.A \subseteq \Delta.A} \qquad \mathsf{weak}\ \frac{\Gamma \subseteq \Delta}{\Gamma \subseteq \Delta.A}$$

We consider the rules as introductions of the indexed types _∈_ and _⊆_ and the rule names as constructors. For instance, suc zero : $A \in \Gamma.A.B$ for any Γ, A, and B; and if we read suc^n zero as unary number n, then $x : A \in \Gamma$ is exactly the (de Bruijn) index of A in Γ.

We can define

$$\begin{array}{lcl} \mathsf{id} & : & \Gamma \subseteq \Gamma \\ \_\mathbin{⨾}\_ & : & \Gamma \subseteq \Delta \to \Delta \subseteq \Phi \to \Gamma \subseteq \Phi \end{array}$$

by recursion, meaning that the (proof-relevant) sublist relation is reflexive and transitive. Thus, lists Γ form a category Cxt with morphisms $\tau : \Gamma \subseteq \Delta$, and the category laws hold propositionally, e.g., we have $\mathsf{id} \mathbin{⨾} \tau \equiv \tau$ in propositional equality for all morphisms τ. The singleton weakening $\mathsf{wk}^A_\Gamma : \Gamma \subseteq \Gamma.A$, also written wk^A or wk, is defined by wk = weak id.

In category theory, a presheaf $\mathcal{A}$ over a category $\mathbb{C}$ is a contravariant functor from $\mathbb{C}$ into the category Set of sets and functions, i. e., a (covariant) functor from the category $\mathbb{C}^{\mathsf{op}}$ with flipped morphisms to Set. In this paper, we are only interested in (covariant) functors from Cxt to Set, thus, by a *presheaf* we will always mean a presheaf over the category $\mathsf{Cxt}^{\mathsf{op}}$. For example, given any type A, we can consider $A\in\_$ as such a presheaf, with the action on objects mapping Γ to the set $A \in \Gamma$ of indices of A in Γ, and the action on morphisms being $\mathsf{reindex} : \Gamma \subseteq \Delta \to A \in \Gamma \to A \in \Delta$. The associated functor laws $\mathsf{reindex}\ \mathsf{id}\ x \equiv x$ and $\mathsf{reindex}\ \tau_2\ (\mathsf{reindex}\ \tau_1\ x) \equiv \mathsf{reindex}\ (\tau_1 \mathbin{⨾} \tau_2)\ x$ hold propositionally.

2.2 STLC and its normal forms

Simple types shall be distinguished into positive types P and negative types N, depending on their root type former (for now). Function ($\Rightarrow$) and product types ($\times$ and 1) are negative, while base types (o) and sum types (+ and 0) are positive.

$$\begin{array}{lcll} A, B, C & ::= & P \mid N & \text{simple types} \\ P & ::= & 0 \mid A + B \mid o & \text{positive types} \\ N & ::= & 1 \mid A \times B \mid A \Rightarrow B & \text{negative types} \end{array}$$

Intrinsically well-typed lambda-terms, in abstract syntax, are just inhabitants t of the indexed set $\boxed{A \dashv \Gamma}$, inductively defined by the following rules.

$$\mathsf{var}\ \frac{A \in \Gamma}{A \dashv \Gamma} \quad \mathsf{abs}\ \frac{B \dashv \Gamma.A}{A \Rightarrow B \dashv \Gamma} \quad \mathsf{app}\ \frac{A \Rightarrow B \dashv \Gamma \quad A \dashv \Gamma}{B \dashv \Gamma}$$

$$\mathsf{unit}\ \frac{}{1 \dashv \Gamma} \quad \mathsf{pair}\ \frac{A_1 \dashv \Gamma \quad A_2 \dashv \Gamma}{A_1 \times A_2 \dashv \Gamma} \quad \mathsf{prj}_i\ \frac{A_1 \times A_2 \dashv \Gamma}{A_i \dashv \Gamma}$$

$$\mathsf{inj}_i\ \frac{A_i \dashv \Gamma}{A_1 + A_2 \dashv \Gamma} \quad \mathsf{case}\ \frac{A_1 + A_2 \dashv \Gamma \quad B \dashv \Gamma.A_1 \quad B \dashv \Gamma.A_2}{B \dashv \Gamma}$$

$$\mathsf{abort}\ \frac{0 \dashv \Gamma}{B \dashv \Gamma}$$

The skilled eye of the reader will immediately recognize the proof rules of intuitionistic propositional logic (IPL) under the Curry-Howard isomorphism, where $A \dashv \Gamma$ is to be read as "A follows from Γ". Using shorthand $\boxed{\mathsf{v}_n = \mathsf{var}\,(\mathsf{suc}^n\ \mathsf{zero})}$ for the nth variable, a term such as

$$\mathsf{abs}\,(\mathsf{abs}\,(\mathsf{pair}\ \mathsf{v}_1\ (\mathsf{abs}\,(\mathsf{app}\ \mathsf{v}_1\ \mathsf{v}_0))))$$

could in concrete syntax be rendered as $\lambda x.\,\lambda y.\,(x, \lambda z.\,y\,z)$. We leave the exact connection to a printable syntax of the STLC to the imagination of the reader, as we shall not be concerned with considering *concrete* terms in this article.

Terms of type A form a presheaf $A\dashv\_$ as witnessed by the standard weakening operation[1]

$$\mathsf{ren} : \Gamma \subseteq \Delta \to A \dashv \Gamma \to A \dashv \Delta$$

defined by recursion over $t : A \dashv \Gamma$, and functor laws for ren analogously to $\mathsf{reindex}$.

Normal forms[2] are logically characterized as those fulfilling the *subformula* property [17, 25]. Normal forms $\boxed{n : \mathsf{Nf}\ A\ \Gamma}$ are mutually defined with *neutral* normal forms $\boxed{u : \mathsf{Ne}\ A\ \Gamma}$. In the following inductive definition, we reuse the rule names from the term constructors.

$$\mathsf{var}\ \frac{A \in \Gamma}{\mathsf{Ne}\ A\ \Gamma} \qquad \mathsf{prj}_i\ \frac{\mathsf{Ne}\ (A_1 \times A_2)\ \Gamma}{\mathsf{Ne}\ A_i\ \Gamma}$$

$$\mathsf{app}\ \frac{\mathsf{Ne}\ (A \Rightarrow B)\ \Gamma \quad \mathsf{Nf}\ A\ \Gamma}{\mathsf{Ne}\ B\ \Gamma} \qquad \mathsf{ne}\ \frac{\mathsf{Ne}\ o\ \Gamma}{\mathsf{Nf}\ o\ \Gamma}$$

$$\mathsf{unit}\ \frac{}{\mathsf{Nf}\ 1\ \Gamma} \qquad \mathsf{pair}\ \frac{\mathsf{Nf}\ A_1\ \Gamma \quad \mathsf{Nf}\ A_2\ \Gamma}{\mathsf{Nf}\ (A_1 \times A_2)\ \Gamma}$$

$$\mathsf{abs}\ \frac{\mathsf{Nf}\ B\ (\Gamma.A)}{\mathsf{Nf}\ (A \Rightarrow B)\ \Gamma} \qquad \mathsf{inj}_i\ \frac{\mathsf{Nf}\ A_i\ \Gamma}{\mathsf{Nf}\ (A_1 + A_2)\ \Gamma}$$

$$\mathsf{case}\ \frac{\mathsf{Ne}\ (A_1 + A_2)\ \Gamma \quad \mathsf{Nf}\ P\ (\Gamma.A_1) \quad \mathsf{Nf}\ P\ (\Gamma.A_2)}{\mathsf{Nf}\ P\ \Gamma}$$

$$\mathsf{abort}\ \frac{\mathsf{Ne}\ 0\ \Gamma}{\mathsf{Nf}\ P\ \Gamma}$$

These rules only allow the elimination of *neutrals*; this restriction guarantees the subformula property and prevents any kind of computational (β) redex. The new rule ne embeds Ne into Nf, but only at base types o [4, Section 3.3]. Further, case distinction via case and abort is restricted to positive types P. As a consequence, our normal forms are *η-long*, meaning that any normal inhabitant of a negative type is a respective introduction (abs, unit, or pair). This justifies the attribute *negative* for these types: the construction of their inhabitants proceeds mechanically, without any choices. In contrast, constructing an inhabitant of a *positive* type involves choice: whether case distinction is required, and which introduction to pick in the end (inj_1 or inj_2).

As expected, $\mathsf{Ne}\ A$ and $\mathsf{Nf}\ A$ are presheaves, i. e., support reindexing with ren just as terms do. From a normal form we can extract the term via an overloaded function

$$\begin{array}{lcl} \ulcorner\_\urcorner & : & \mathsf{Nf}\ A\ \Gamma \to A \dashv \Gamma \\ \ulcorner\_\urcorner & : & \mathsf{Ne}\ A\ \Gamma \to A \dashv \Gamma \end{array}$$

that discards constructor ne but keeps all other constructors. This erasure function naturally commutes with reindexing, making it a natural transformation between the presheaves $\mathsf{Nf}\ A$ ($\mathsf{Ne}\ A$, resp.) and $A\dashv\_$. We shall simply write, for instance, $\mathsf{Nf}\ A \dot\to A\dashv\_$ for such presheaf morphisms. (The point on the arrow is mnemonic for *pointwise*.) Slightly abusive, we shall extend this notation to n-ary morphisms, e. g., write $\mathcal{A} \dot\to \mathcal{B} \dot\to \mathcal{C}$ for $\forall\Gamma.\ \mathcal{A}\,\Gamma \to (\mathcal{B}\,\Gamma \to \mathcal{C}\,\Gamma)$.

We use the $\boxed{\forall}$ quantifier as implicit dependent function type former in our meta-language, similar to the polymorphic quantifier in functional programming languages such as Haskell. We may instantiate a polymorphic function such as $f : \mathcal{A} \dot\to \mathcal{B}$ silently, e. g., when $a : \mathcal{A}\,\Gamma$ then $f\ a : \mathcal{B}\,\Gamma$. For clarity, we may sometimes provide the instantiation explicitly via subscript, e. g., $f_\Gamma : \mathcal{A}\,\Gamma \to \mathcal{B}\,\Gamma$ and $f_\Gamma\ a : B\,\Gamma$. Similarly, to introduce a polymorphic function we may provide the implicit abstraction as subscript, such as in $\lambda_\Gamma\, a.\ b : \mathcal{A} \dot\to \mathcal{B}$ or $\lambda_\Gamma\,(a : \mathcal{A}\,\Gamma).\ b : \mathcal{A} \dot\to \mathcal{B}$, or omit it, e. g., $\lambda\,(a : \mathcal{A}\,\Gamma).\ b : \mathcal{A} \dot\to \mathcal{B}$.

[1] Here, ren is short for *renaming*, but in a nameless calculus we should better speak of *reindexing*, which could, a bit clumsily, be also abbreviated to ren.

[2] There is also a stronger notion of normal form, requiring that two *extensionally equal* lambda-terms, i. e., those that denote the same set-theoretical function, have the same normal form [3, 22, 28]. Such normal forms do not have a simple inductive definition, and we shall not consider them in this article.

Remark 1 (Positive eliminations into negative types). *While the coproduct eliminations* case *and* abort *are limited to normal forms of positive types P, their extension* case^B *and* abort^B *to negative types is admissible, for instance:*

$$\begin{array}{lll}
\mathsf{abort}^B & : & \mathsf{Ne}\,0 \dot{\to} \mathsf{Nf}\,B \\
\mathsf{abort}^1 \quad u & = & \mathsf{unit} \\
\mathsf{abort}^P \quad u & = & \mathsf{abort}\,u \\
\mathsf{abort}^{A_1 \times A_2}\, u & = & \mathsf{pair}\,(\mathsf{abort}^{A_1}\,u)\,(\mathsf{abort}^{A_2}\,u) \\
\mathsf{abort}^{A \Rightarrow B}\, u & = & \mathsf{abs}\,(\mathsf{abort}^B\,(\mathsf{ren}\,\mathsf{wk}^A\,u))
\end{array}$$

case *generalizes analogously, with a bit of care when weakening the branches.*

2.3 Normalization

Normalization is concerned with finding a normal form $n : \mathsf{Nf}\ A\,\Gamma$ for each term $t : A \dashv \Gamma$. The normal form should be *sound*, i. e., $\ulcorner n \urcorner \cong t$ with respect to an equational theory $\cong$ on terms. Further, normalization should decide $\cong$, i. e., terms t, t' with $t \cong t'$ should have the same normal form n. In this article, we present only the normalization function $\boxed{\mathsf{norm} : A \dashv \Gamma \to \mathsf{Nf}\ A\,\Gamma}$ without proving its soundness and completeness.[3] From a logical perspective, we will compute for each derivation of $A \dashv \Gamma$ a normal derivation $\mathsf{Nf}\ A\,\Gamma$.

The method *normalization by evaluation* (NbE)

$$\mathsf{norm}\,(t : A \dashv \Gamma) = \downarrow^A (\!|\, t \,|\!)_{\mathsf{fresh}^\Gamma}$$

decomposes normalization into *evaluation*

$$(\!|\_|\!) : (t : A \dashv \Gamma) \to [\![\Gamma]\!] \dot{\to} [\![A]\!]$$

in the identity environment

$$\mathsf{fresh}^\Gamma : [\![\Gamma]\!]\Gamma$$

followed by *reification*

$$\downarrow^A : [\![A]\!] \dot{\to} \mathsf{Nf} A$$

(aka quoting). The role of evaluation is to produce from a term the corresponding semantic (i. e., meta-theoretic) function, which is finally reified to a normal form. Since we are evaluating open terms t, we need to supply an environment fresh^Γ which will map the free indices of t to corresponding *unknowns*. To accommodate unknowns in the semantics, types A are mapped to presheaves $[\![A]\!]$ (rather than just sets), and in particular each base type o is mapped to the presheaf $\mathsf{Ne}\ o$ with the intention that the neutrals take the role of the unknowns. The mapping $\uparrow^A : \mathsf{Ne}\ A \dot{\to} [\![A]\!]$ from neutrals to unknowns is called *reflection* (aka unquoting), and defined mutually with reification by induction on type A.

At this point, let us fix some notation for sets to prepare for some constructions of presheaves. Let $\mathbf{1}$ denote the unit set and $()$ its unique inhabitant, $\mathbf{0}$ the empty set and $\mathsf{magic} : \mathbf{0} \to T$ the *ex falsum quod libet* elimination into any set T. Given sets S_1 and S_2, their Cartesian product is written $S_1 \times S_2$ with projections $\pi_i : S_1 \times S_2 \to S_i$, and their disjoint sum $S_1 + S_2$ with injections $\iota_i : S_i \to S_1 + S_2$ and elimination $[f_1, f_2] : S_1 + S_2 \to T$ for arbitrary $f_i : S_i \to T$.

Presheaf (co)products $\hat{0}$, $\hat{1}$, $\hat{+}$, and $\hat{\times}$ are constructed pointwise, e. g., $\hat{0}\,\Gamma = \mathbf{0}$, and given two presheaves $\mathcal{A}$ and $\mathcal{B}$, $(\mathcal{A} \mathbin{\hat{+}} \mathcal{B})\,\Gamma = \mathcal{A}\,\Gamma + \mathcal{B}\,\Gamma$. For the exponential of presheaves, however, we need the *Kripke function space* $(\mathcal{A} \mathbin{\hat{\Rightarrow}} \mathcal{B})\,\Gamma = \forall \Delta.\, \Gamma \subseteq \Delta \to \mathcal{A}\,\Delta \to \mathcal{B}\,\Delta$.

We will interpret simple types A as corresponding presheaves $[\![A]\!]$. Let us start with the negative types, defining reflection $\uparrow^A : \mathsf{Ne}\ A \dot{\to} [\![A]\!]$ and reification $\downarrow^A : [\![A]\!] \dot{\to} \mathsf{Nf}\ A$ along the way.

$$\begin{array}{lll}
[\![1]\!] & = & \hat{1} \\
\uparrow^1_\Gamma u & = & () \\
\downarrow^1_\Gamma () & = & \mathsf{unit} \\
\\
[\![A \times B]\!] & = & [\![A]\!] \mathbin{\hat{\times}} [\![B]\!] \\
\uparrow^{A \times B}_\Gamma u & = & (\uparrow^A_\Gamma(\mathsf{prj}_1\ u),\ \uparrow^B_\Gamma(\mathsf{prj}_2\ u)) \\
\downarrow^{A \times B}_\Gamma (a, b) & = & \mathsf{pair}\,(\downarrow^A_\Gamma a)\,(\downarrow^B_\Gamma b) \\
\\
[\![A \Rightarrow B]\!] & = & [\![A]\!] \mathbin{\hat{\Rightarrow}} [\![B]\!] \\
\uparrow^{A \Rightarrow B}_\Gamma u & = & \lambda_\Delta\,(\tau : \Gamma \subseteq \Delta)\,a.\ \uparrow^B_\Delta(\mathsf{app}\ (\mathsf{ren}\,\tau\,u)\ (\downarrow^A_\Delta a)) \\
\downarrow^{A \Rightarrow B}_\Gamma f & = & \mathsf{abs}\,(\downarrow^B_{\Gamma.A}(f\ \mathsf{wk}^A_\Gamma\ \mathsf{fresh}^A_\Gamma))
\end{array}$$

In the reification at function types $\downarrow^{A \Rightarrow B}$, the renaming $\mathsf{wk}^A_\Gamma : \Gamma \subseteq \Gamma.A$ makes room for a new variable of type A, which is reflected into $[\![A]\!]$ by $\boxed{\mathsf{fresh}^A_\Gamma = \uparrow^A_{\Gamma.A} \mathsf{v}_0 : [\![A]\!](\Gamma.A)}$. The ability to introduce fresh variables into a context, and to use semantic objects such as $f : [\![A \Rightarrow B]\!]\Gamma$ in such an extended context, is the reason for utilizing presheaves instead of just sets as semantic types.

Note also that in the equation for $\uparrow^{A \Rightarrow B}$, the neutral $u : \mathsf{Ne}\ (A \Rightarrow B)\ \Gamma$ is transported into $\mathsf{Ne}\ (A \Rightarrow B)\ \Delta$ via reindexing with $\tau : \Gamma \subseteq \Delta$, in order to be applicable to the normal form $\downarrow^A_\Delta a$ reified from the semantic value $a : [\![A]\!]\Delta$.

A direct extension of our presheaf semantics to positive types cannot work. For instance, with $[\![0]\!] = \hat{0}$, simply $\mathsf{fresh}^0_\varepsilon : \mathbf{0}$ would give us an inhabitant of the empty set, which means that reflection at the empty type would not be definable. Similarly, the setting $[\![A + B]\!] = [\![A]\!] \mathbin{\hat{+}} [\![B]\!]$ is refuted by $\mathsf{fresh}^{A+B}_\varepsilon : [\![A]\!](A{+}B) + [\![B]\!](A{+}B)$, which would require us to make a decision of whether A holds or B holds while only be given a hypothesis of type $A + B$. Not even the usual interpretation of base types $[\![o]\!] = \mathsf{Ne}\ o$ works in the presence of sums, as we would not be able to interpret the term $\mathsf{abs}\,(\mathsf{case}\,\mathsf{v}_0\,\mathsf{v}_0\,\mathsf{v}_0) : (o + o) \Rightarrow o$ in our semantics, because $\mathsf{Ne}\ \ o\ \ (o + o)$ is empty. What is needed are case distinctions on neutrals in the semantics, allowing us the elimination of positive hypotheses before producing a semantic value, and we shall capture this capability in a strong monad $\mathcal{C}$ which can *cover* the cases.

Recall that a monad $\mathcal{C}$ on presheaves is first an endofunctor, i.e., it maps any presheaf $\mathcal{A}$ to the presheaf $\mathcal{C}\,\mathcal{A}$ and any presheaf morphism $f : \mathcal{A} \dot{\to} \mathcal{B}$ to the morphism $\mathsf{map}^{\mathcal{C}} f : \mathcal{C}\,\mathcal{A} \dot{\to} \mathcal{C}\ \mathcal{B}$ satisfying the functor laws for identity and composition. Then, there are natural transformations $\mathsf{return}^{\mathcal{C}} : \mathcal{A} \dot{\to} \mathcal{C}\,\mathcal{A}$ (unit) and $\mathsf{join}^{\mathcal{C}} : \mathcal{C}\,(\mathcal{C}\,\mathcal{A}) \dot{\to} \mathcal{C}\,\mathcal{A}$ (multiplication) satisfying the monad laws.

We are looking for a monad $\mathcal{C}$, called a *cover monad*, that takes the role of the *oracle* alluded to in the introduction, and offers us

[3] An Agda formalization of NbE for IPL with soundness-by-construction is listed at https://andreasabel.github.io/ipl/html/NbeModel.html. It is demonstrated that normalization does not change the interpretation of a term as function in the meta-language. Instead of presheaves, we work with Kripke predicates there. The main contribution of our work, the utilization of a cover monad for NbE, is however more clearly presented by just constructing the normalization function, as we do in this article.

the following services:

$$\begin{array}{ll}\mathsf{runNf}^{\mathcal{C}} & : \mathcal{C}\,(\mathsf{Nf}\,P) \dot{\to} \mathsf{Nf}\,P \\ \mathsf{abort}^{\mathcal{C}} & : \mathsf{Ne}\;0 \dot{\to} \mathcal{C}\,\mathcal{B} \\ \mathsf{case}^{\mathcal{C}}_{\Gamma} & : \mathsf{Ne}\;(A_1 + A_2)\;\Gamma \\ & \quad \to \mathcal{C}\,\mathcal{B}\,(\Gamma.A_1) \to \mathcal{C}\,\mathcal{B}\,(\Gamma.A_2) \to \mathcal{C}\,\mathcal{B}\,\Gamma\end{array}$$

Method $\mathsf{runNf}^{\mathcal{C}}$ enables us to *run* a monadic computation of a normal form, $\mathcal{C}\,(\mathsf{Nf}\,P)\,\Gamma$, and delivers a normal form $\mathsf{Nf}\;P\;\Gamma$. While $\mathsf{runNf}^{\mathcal{C}}$ is a left inverse of $\mathsf{return}^{\mathcal{C}}$, the converse is not true. Instead, the information contained in $\mathcal{C}$ is reified and becomes part of the normal form delivered by $\mathsf{runNf}^{\mathcal{C}}$. Technically, $(\mathsf{Nf}\,P, \mathsf{runNf}^{\mathcal{C}})$ is a *monad algebra* for monad $\mathcal{C}$. Besides $\mathsf{runNf}^{\mathcal{C}} \circ \mathsf{return}^{\mathcal{C}} = \mathsf{id}_{\mathsf{Nf}\,P}$, the square $\mathsf{runNf}^{\mathcal{C}} \circ \mathsf{join}^{\mathcal{C}} = \mathsf{runNf}^{\mathcal{C}} \circ \mathsf{map}^{\mathcal{C}}\mathsf{runNf}^{\mathcal{C}}$ stands. This means that given a case tree whose leaves are again case trees ending in normal forms, it does not matter whether we first join the nested case trees and then assemble the normal form, or whether we first turn the inner case trees into normal forms via $\mathsf{map}^{\mathcal{C}}\mathsf{runNf}^{\mathcal{C}}$, and then the outer one.

Method $\mathsf{abort}^{\mathcal{C}}_{\Gamma}$ allows us to end a computation by exhibiting an inconsistency witnessed by a neutral term $u : \mathsf{Ne}\;0\;\Gamma$. Remember that, ultimately, computations in $\mathcal{C}$ produce a normal form extracted by $\mathsf{runNf}^{\mathcal{C}}$. With $\mathsf{abort}^{\mathcal{C}}$ we notify the oracle that we are in an absurd case and the desired normal form can be trivially constructed by the abort-function of Remark 1.

Method $\mathsf{case}^{\mathcal{C}}_{\Gamma}$ allows us, in a computation $\mathcal{C}\,\mathcal{B}\,\Gamma$, to ask the oracle about a neutral term $\mathsf{Ne}\;(A_1+A_2)\,\Gamma$ of sum type. We have to supply handlers for both possible answers: a computation $\mathcal{C}\,\mathcal{B}\,(\Gamma.A_1)$ that can utilize an additional hypothesis of type A_1 in the case of a left injection, and analogously a computation $\mathcal{C}\,\mathcal{B}\,(\Gamma.A_2)$ for the case of a right injection.

$$\begin{array}{lcl}
[\![o]\!] & = & \mathcal{C}\,(\mathsf{Ne}\;o) \\
\uparrow^{o} & = & \mathsf{return}^{\mathcal{C}} \\
\downarrow^{o} & = & \mathsf{runNf}^{\mathcal{C}} \circ \mathsf{map}^{\mathcal{C}}\mathsf{ne} \\
 & & \\
[\![0]\!] & = & \mathcal{C}\,\hat{0} \\
\uparrow^{0} & = & \mathsf{abort}^{\mathcal{C}} \\
\downarrow^{0} & = & \mathsf{runNf}^{\mathcal{C}} \circ \mathsf{map}^{\mathcal{C}}\mathsf{magic} \\
 & & \\
[\![A+B]\!] & = & \mathcal{C}\,([\![A]\!] \mathbin{\hat{+}} [\![B]\!]) \\
\uparrow^{A+B}_{\Gamma}\,u & = & \mathsf{case}^{\mathcal{C}}\,u\,(\mathsf{return}^{\mathcal{C}}\,(\iota_1\,\mathsf{fresh}^{A}_{\Gamma})) \\
 & & \qquad (\mathsf{return}^{\mathcal{C}}\,(\iota_2\,\mathsf{fresh}^{B}_{\Gamma})) \\
\downarrow^{A+B} & = & \mathsf{runNf}^{\mathcal{C}} \circ \mathsf{map}^{\mathcal{C}}[\mathsf{inj}_1 \circ \downarrow^{A},\ \mathsf{inj}_2 \circ \downarrow^{B}]
\end{array}$$

Figure 1: Interpretation of positive types.

The similarity of the monad services $\mathsf{abort}^{\mathcal{C}}$ and $\mathsf{case}^{\mathcal{C}}$ to the corresponding constructors for normal forms, or their generalization of Remark 1, is hard to miss. Unsurprisingly, just *case trees* are a first instance of a cover monad: the free cover monad Cov defined as an inductive family with constructors $\mathsf{return}^{\mathsf{Cov}}$, $\mathsf{abort}^{\mathsf{Cov}}$, and $\mathsf{case}^{\mathsf{Cov}}$. One can visualize an element $c : \mathsf{Cov}\,\mathcal{A}\,\Gamma$ as a binary case tree whose inner nodes (case) are labeled by neutral terms of sum type $A_1 + A_2$ and its two branches by the context extensions A_1 and A_2, resp. Leaves are either labeled by a neutral term of empty type 0 (see abort), or by an element of $\mathcal{A}$ (see return). Functoriality amounts to replacing the labels of the return-leaves, and the monadic bind (aka Kleisli extension) replaces these leaves by further case trees. Bind is realized via $\mathsf{join}^{\mathsf{Cov}}$, which flattens a 2-level case tree, i. e., a case tree with case trees as leaves, into a single one. Finally, $\mathsf{runNf}^{\mathsf{Cov}}$ is a simple recursion on the tree, replacing $\mathsf{case}^{\mathsf{Cov}}$ and $\mathsf{abort}^{\mathsf{Cov}}$ by the case and abort constructions on normal forms, and $\mathsf{return}^{\mathsf{Cov}}$ by the identity.

Using the services of a generic cover monad $\mathcal{C}$, we can complete our semantics, see Figure 1.

Since positive types P have a monadic interpretation, there is a monad algebra $\mathcal{C}\,[\![P]\!] \dot{\to} [\![P]\!]$, which is simply $\mathsf{join}^{\mathcal{C}}$. It can be extended to a monad algebra $\mathsf{run}^{A} : \mathcal{C}\,[\![A]\!] \dot{\to} [\![A]\!]$ for any simple type A, meaning we can *run* the monad, [4] pushing its effects into $[\![A]\!]$.We proceed by induction on A. At negative types we can recurse pointwise at a smaller type, exploiting that values of negative types are essentially (finite or infinite) tuples.

$$\begin{array}{ll}
\mathsf{run}^{A} & : \ \mathcal{C}[\![A]\!] \dot{\to} [\![A]\!] \\
\mathsf{run}^{P} & \ c = \mathsf{join}^{\mathcal{C}}\ c \\
\mathsf{run}^{1} & \ c = () \\
\mathsf{run}^{A\times B} & \ c = (\mathsf{run}^{A}\,(\mathsf{map}^{\mathcal{C}}\ \pi_1\ c),\ \mathsf{run}^{B}\,(\mathsf{map}^{\mathcal{C}}\ \pi_2\ c)) \\
\mathsf{run}^{A\Rightarrow B} & \ c = \lambda_{\Delta}\,\tau\ a.\ \mathsf{run}^{B}\,(\widehat{\mathsf{map}}^{\mathcal{C}}\ (\lambda_{\Phi}\ \tau'\,f.\ f\ \mathsf{id}\ (\mathsf{ren}\ \tau'\ a)) \\
 & \qquad\qquad (\mathsf{ren}\ \tau\ c))
\end{array}$$

For the case of function types $A \Rightarrow B$, we require the monad $\mathcal{C}$ to be *strong*, which amounts to having $\widehat{\mathsf{map}}^{\mathcal{C}}_{\Gamma}\ \ell : \mathcal{C}\mathcal{A}\,\Gamma \to \mathcal{C}\,\mathcal{B}\,\Gamma$ already for a "local" presheaf morphism $\ell : (\mathcal{A} \mathbin{\hat{\Rightarrow}} \mathcal{B})\Gamma$. The typings are $c : \mathcal{C}[\![A \Rightarrow B]\!]\Gamma$ and $\tau : \Gamma \subseteq \Delta$ and $a : [\![A]\!]\Delta$, and now we want to apply every function $f : [\![A \Rightarrow B]\!]$ in the cover c to argument a. Clearly, $\mathsf{map}^{\mathcal{C}}$ is not applicable since it would expect a global presheaf morphism $[\![A \to B]\!] \dot{\to} [\![B]\!]$, i. e., something that works in *any* context. However, applying to $a : [\![A]\!]\Delta$ can only work in context Δ or any extension $\tau' : \Delta \subseteq \Phi$, since we can transport a to such a Φ via $a' := \mathsf{ren}\ \tau'\ a : [\![A]\!]\Phi$, but not to a context unrelated to Δ. We obtain our input to run^{B} of type $\mathcal{C}[\![B]\!]\Gamma$ as an instance of $\widehat{\mathsf{map}}^{\mathcal{C}}_{\Gamma}$ applied to the local presheaf morphism $(\lambda_{\Phi}\ \tau'\,f.\ f\ \mathsf{id}\ a') : \Delta \subseteq \Phi \to [\![A \Rightarrow B]\!]\Phi \to [\![B]\!]\Phi$ and the transported cover $\mathsf{ren}\ \tau\ c : \mathcal{C}[\![A \Rightarrow B]\!]\Delta$.

We extend the type interpretation pointwise to contexts, i. e., $[\![\varepsilon]\!] = \hat{1}$ and $[\![\Gamma.A]\!] = [\![\Gamma]\!] \mathbin{\hat{\times}} [\![A]\!]$, and obtain a natural projection function $\mathsf{lookup}\,(x : A \in \Gamma) : [\![\Gamma]\!] \dot{\to} [\![A]\!]$ from the semantic environments. The evaluation function $\boxed{(\!|\,t : A \dashv \Gamma\,|\!) : [\![\Gamma]\!] \dot{\to} [\![A]\!]}$ can now be defined by recursion on t, see Figure 2. Herein, the environment γ lives in $[\![\Gamma]\!]\Delta$, thus, $(\!|\,t\,|\!)_{\gamma} : [\![A]\!]\Delta$. For the interpretation of the binders abs and case we use the mutually defined $\lambda(\!|\,\_\,|\!)$. The coproduct eliminations $(\!|\,\mathsf{abort}\,|\!)$ and $(\!|\,\mathsf{case}\,|\!)$ targeting an arbitrary semantic type $[\![B]\!]$ are definable thanks to the weak sheaf property, i. e., the presence of pasting via run^{B} for any type B, and strong functoriality of $\mathcal{C}$. It is noteworthy that the interpreter $(\!|\,\_\,|\!)$ does not use any of the services of the cover monad; in fact, it only requires the monad $\mathcal{C}$ to be strong. The interpreter is completely *generic* for the CBN monadic semantics of types. This restores the spirit of STLC NbE for sum types: Take an *off-the-shelf* interpreter for

[4] In categorical terminology, the existence of run^{A} means that all semantic types $[\![A]\!]$ fulfill the *weak sheaf condition*, aka *weak pasting*.

$$
\begin{array}{lcl}
(\!| t : A \dashv \Gamma |\!) & : & [\![\Gamma]\!] \mathrel{\dot\to} [\![A]\!] \\
(\!| \mathsf{var}\, x |\!)_\gamma & = & \mathsf{lookup}\, x\, \gamma \\
(\!| \mathsf{abs}\, t |\!)_\gamma & = & \lambda (\!| t |\!)_\gamma \\
(\!| \mathsf{app}\, t\, u |\!)_\gamma & = & (\!| t |\!)_\gamma\ \mathsf{id}\ (\!| u |\!)_\gamma \\
(\!| \mathsf{unit} |\!)_\gamma & = & () \\
(\!| \mathsf{pair}\, t_1\, t_2 |\!)_\gamma & = & ((\!| t_1 |\!)_\gamma,\ (\!| t_2 |\!)_\gamma) \\
(\!| \mathsf{prj}_i\ t |\!)_\gamma & = & \pi_i (\!| t |\!)_\gamma \\
(\!| \mathsf{inj}_i\ t |\!)_\gamma & = & \iota_i (\!| t |\!)_\gamma \\
(\!| \mathsf{case}\ u\ t_1\ t_2 |\!)_\gamma & = & (\!| \mathsf{case} |\!)\ (\!| u |\!)_\gamma\ \lambda (\!| t_1 |\!)_\gamma\ \lambda (\!| t_2 |\!)_\gamma \\
(\!| \mathsf{abort}\, u |\!)_\gamma & = & (\!| \mathsf{abort} |\!)\ (\!| u |\!)_\gamma
\end{array}
$$

$$
\begin{array}{l}
\lambda (\!|\ t : B \dashv \Gamma.A\ |\!) : [\![\Gamma]\!] \mathrel{\dot\to} [\![A \Rightarrow B]\!] \\
\quad = \lambda_{\,\Delta}\, (\gamma : [\![\Gamma]\!]\Delta)_{\,\Phi}\, (\tau : \Delta \subseteq \Phi)\, (a : [\![A]\!]\Phi).\ (\!| t |\!)_{(\mathsf{ren}\,\tau\,\gamma,\ a)}
\end{array}
$$

$$
\begin{array}{l}
(\!| \mathsf{abort} |\!)^B\ :\ [\![0]\!] \mathrel{\dot\to} [\![B]\!] \\
(\!| \mathsf{abort} |\!)^B = \mathsf{run}^B \circ \mathsf{map}^C\ \mathsf{magic} \\[1ex]
(\!| \mathsf{case} |\!)^B\ \ :\ [\![A_1 + A_2]\!] \mathrel{\dot\to} [\![A_1 \Rightarrow B]\!] \mathrel{\dot\to} [\![A_2 \Rightarrow B]\!] \mathrel{\dot\to} [\![B]\!] \\
(\!| \mathsf{case} |\!)^B\ \ = \lambda\, c\, f_1\, f_2.\ \mathsf{run}^B(\widehat{\mathsf{map}}^C\ (\lambda\,\tau.\ [f_1\,\tau,\ f_2\,\tau])\ c)
\end{array}
$$

Figure 2: Evaluation of terms.

terms, implement reflection and reification after interpreting base types as sets (presheaves) of neutrals, plug them together with the interpreter, and *voila!*, there is your normalizer! Our insight here is that *off-the-shelf interpreter* means *monadic* in the presence of positive types.

To complete the normalization function $\mathsf{norm}\,(t : A \dashv \Gamma) = \downarrow^A_\Gamma (\!| t |\!)_{\mathsf{fresh}^\Gamma}$ we define the identity environment $\mathsf{fresh}^\Gamma : [\![\Gamma]\!]\Gamma$, which maps each free index to its corresponding unknown in the semantics, by recursion on Γ:

$$
\begin{array}{lcl}
\mathsf{fresh}^\varepsilon & = & () \\
\mathsf{fresh}^{\Gamma.A} & = & (\mathsf{ren}\ \mathsf{wk}^A\ \mathsf{fresh}^\Gamma,\ \mathsf{fresh}^A_\Gamma)
\end{array}
$$

We have obtained a computable function norm that yields for any derivation $A \dashv \Gamma$ a normal derivation $\mathsf{Nf}\ A\ \Gamma$. This amounts to proving consistency and canonicity of intuitionistic propositional logic, the Curry-Howard counterpart of our STLC with products and sums. The normalization function is parametrized by an arbitrary cover monad C, which can be implemented via case trees (Cov). However, different implementations are possible. In the next section, we exhibit another canonical choice for C: the parameterized continuation monad on presheafs.

2.4 Continuation monad

Besides the free cover monad Cov, there is another canonical implementation of its interface, the continuation monad. Filinski [14, Section 5.4] [15, Section 3.2] already utilized the continuation monad for normalization by evaluation. In our setting, we use a continuation monad CC on *presheaves* $\mathcal{B}$ defined by

$$\mathsf{CC}\ \mathcal{B} = \forall P.\ (\mathcal{B} \mathrel{\hat\Rightarrow} \mathsf{Nf}\,P) \mathrel{\hat\Rightarrow} \mathsf{Nf}\,P.$$

The answer type of this continuation monad is always Nf, however, we are polymorphic in the positive type P of normal forms we emit in the end.

Agda has been really helpful to produce the rather technical but straightforward evidence that CC is a strong monad. The method $\mathsf{runNf}^{\mathsf{CC}} : \mathsf{CC}\,(\mathsf{Nf}\,P) \mathrel{\dot\to} \mathsf{Nf}\,P$ exists by definition, using the identity continuation $\mathsf{Nf}\,P \mathrel{\hat\Rightarrow} \mathsf{Nf}\,P$. In the following, we demonstrate that CC enables matching on neutrals:

$$
\begin{array}{l}
\mathsf{abort}^{\mathsf{CC}}_\Gamma\ (u : \mathsf{Ne}\ 0\ \Gamma)\ :\ \mathsf{CC}\,\mathcal{B}\,\Gamma \\
\mathsf{abort}^{\mathsf{CC}}_\Gamma\ u\,{}_\Delta\, (\tau : \Gamma \subseteq \Delta)\, (k : (\mathcal{B} \mathrel{\hat\Rightarrow} \mathsf{Nf}\,P)\,\Delta)\ = \\
\qquad \mathsf{abort}\,(\mathsf{ren}\,\tau\,u)
\end{array}
$$

$$
\begin{array}{l}
\mathsf{case}^{\mathsf{CC}}_\Gamma\ (u : \mathsf{Ne}\,(A_1 + A_2)\,\Gamma)_{\,\Delta} \\
\qquad (c_1 : \mathsf{CC}\,\mathcal{B}\,(\Gamma.A_1))\ (c_2 : \mathsf{CC}\,\mathcal{B}\,(\Gamma.A_2))\ :\ \mathsf{CC}\,\mathcal{B}\,\Gamma \\
\mathsf{case}^{\mathsf{CC}}_\Gamma\ u\ c_1\ c_2\ (\tau : \Gamma \subseteq \Delta)\ (k : (\mathcal{B} \mathrel{\hat\Rightarrow} \mathsf{Nf}\,P)\,\Delta)\ = \\
\qquad \mathsf{case}\ (\mathsf{ren}\,\tau\,u)\ n_1\ n_2 \\
\qquad \textbf{where} \\
\qquad\quad n_i\ :\ \mathsf{Nf}\ P\ (\Delta.A_i) \\
\qquad\quad n_i = c_i\ (\mathsf{lift}^{A_i}\ \tau : \Gamma.A_i \subseteq \Delta.A_i) \\
\qquad\qquad\quad (\lambda_{\,\Phi}\,(\tau' : \Delta.A_i \subseteq \Phi)\ (j : \mathcal{B}\,\Phi).\ k\ (\mathsf{wk}^{A_i} \mathbin{⨾} \tau')\ j)
\end{array}
$$

It is noteworthy (yet unsurprising) that $\mathsf{abort}^{\mathsf{CC}}$ discards continuation k, while $\mathsf{case}^{\mathsf{CC}}$ uses it twice, once in each case. Thus, normal forms can be of exponential size; for example, the normal form of the identity function over n-tuples of booleans has a case tree of height n, hence, size $\Theta(2^n)$.

The NbE algorithm using CC is comparable to Danvy's type-directed partial evaluation [13, Figure 8]. However, Danvy uses shift-reset style continuations, which can be expressed via the continuation monad, and relies on Scheme's `gensym` to produce fresh variables names rather than presheaves and the Kripke function space.

3 NORMALIZATION TO CALL-BY-PUSH VALUE

The placement of the monad C in the type semantics of the previous section is a bit wasteful: Each positive type is prefixed by C. In our grammar of normal forms, this corresponds to the ability to perform case distinctions (case, abort) at any positive type P. In fact, our type interpretation $[\![A]\!]$ corresponds to the translation of call-by-name (CBN) lambda-calculus into Moggi's monadic meta-language [19, 23].

It would be sufficient to perform all necessary case distinctions when transitioning from a negative type to a positive type. Introduction of the function type adds hypotheses to the context, providing material for case distinctions, but introduction of positive types does not add anything in that respect. Thus, we could *focus* on positive introductions until we transition back to a negative type. Such focusing is present in the call-by-value (CBV) lambda-calculus, where positive introductions only operate on values, and variables stand only for values. This structure is even more clearly spelled out in Levy's call-by-push-value (CBPV) [19], as it comes with a deep classification of types into positive and negative ones. In the following, we shall utilize pure (i. e., effect-free) CBPV to achieve chaining of positive introductions.

3.1 Types and polarization

CBPV calls positive types P *value types* A and negative types N *computation types* $\underline{B}$, yet we shall stick to our terminology which is common in publications on focalization. However, we shall use *Thunk* for switch $\downarrow$ and *Comp* for switch $\uparrow$, to avoid confusion with our notation for reflection and reification.

$$\begin{array}{l} \mathsf{Ty}^+ \ni P ::= o^+ \mid 1 \mid P_1 \times P_2 \mid 0 \mid P_1 + P_2 \mid \mathit{Thunk}\, N \\ \mathsf{Ty}^- \ni N ::= o^- \mid \top \mid N_1 \,\&\, N_2 \mid P \Rightarrow N \mid \mathit{Comp}\, P \end{array}$$

CBPV uses U for *Thunk* and F for *Comp*, however, we find these names uninspiring unless you have good knowledge of the intended model. Further, CBPV [19] employs labeled sums $\Sigma_I(P_i)_{i:I}$ and labeled records $\Pi_I(P_i)_{i:I}$ for up to countably infinite label sets I while we only have finite sums $(0, +)$ and records $(\top, \&)$. However, this difference is not essential, our treatment extends directly to the infinite case, since we are working in type theory, which allows infinitely branching inductive types. As a last difference, CBPV does not consider uninterpreted base types; in anticipation of the next section, we add them as both positive atoms (o^+) and negative atoms (o^-).

Getting a bit ahead of ourselves, let us consider the mutually defined interpretations $[\![P]\!]$ and $[\![N]\!]$ of positive and negative types as presheaves (see Figure 3). This interpretation is parametrized by a strong monad C on presheaves. Semantically, we do not distinguish

$$\begin{array}{lcl} [\![1]\!] & = & \hat{1} \\ [\![P_1 \times P_2]\!] & = & [\![P_1]\!] \mathbin{\hat{\times}} [\![P_2]\!] \\ [\![0]\!] & = & \hat{0} \\ [\![P_1 + P_2]\!] & = & [\![P_1]\!] \mathbin{\hat{+}} [\![P_2]\!] \\ [\![\mathit{Thunk}\, N]\!] & = & [\![N]\!] \\ [\![o^+]\!] & = & o^+ \in \_ \\ \\ [\![\top]\!] & = & \hat{1} \\ [\![N_1 \,\&\, N_2]\!] & = & [\![N_1]\!] \mathbin{\hat{\times}} [\![N_2]\!] \\ [\![P \Rightarrow N]\!] & = & [\![P]\!] \mathbin{\hat{\Rightarrow}} [\![N]\!] \\ [\![\mathit{Comp}\, P]\!] & = & C[\![P]\!] \\ [\![o^-]\!] & = & C(\mathsf{Ne}\; o^-) \end{array}$$

Figure 3: Interpretation of polarized types

between positive and negative products. Notably, sum types can now be interpreted as plain (pointwise) presheaf sums. The *Thunk* marker is ignored, yet *Comp*, marking the switch from the negative to the positive type interpretation, places the monad C. Positive atoms o^+, standing for value types without constructors, are only inhabited by variables $x : o^+ \in \Gamma$. Negative atoms o^- stand for computation types without own eliminations. Thus, their inhabitants stem only from eliminations of more complex types. They are built from positive eliminations captured in C and negative eliminations chained together as neutral $\mathsf{Ne}\; o^-$, which we shall define below. The multiplication join^C of the strong monad C can be extended to a monad algebra $\mathsf{run}^N : C[\![N]\!] \mathrel{\dot{\to}} [\![N]\!]$ for negative types N. The construction proceeds by recursion on N, exactly as in Section 2.3. This makes the effects of C available at any negative type, in other words, makes all negative types *monadic*.

Contexts are lists of *positive* types since in CBPV variables stand for values. Interpretation of contexts $[\![\Gamma]\!]$ is again defined pointwise $[\![\varepsilon]\!] = \hat{1}$ and $[\![\Gamma.P]\!] = [\![\Gamma]\!] \mathbin{\hat{\times}} [\![P]\!]$.

3.2 Terms and evaluation

Value terms $\boxed{v : \mathsf{Val}\, P\, \Gamma}$, or short, *values*, and computation terms $\boxed{t : \mathsf{Tm}\, N\, \Gamma}$, or short, *terms*, are defined mutually by the rules in Figure 4. Values inhabit positive types P and are given by introduction rules only, whereas terms inhabit negative types N and comprise both introduction rules for negative types as well as elimination rules for both positive (split, case, abort) and negative types. Note that function application is restricted to value arguments. Values of type *Thunk* N are embedded by force. Further, values of type P can be embedded via ret, producing a term of type *Comp* P. Such terms are eliminated by bind which is, unlike the usual monadic bind, not only available for *Comp*-types but for arbitrary negative types N. This is justified by the monadic character of negative types, by virtue of run^N.

$$\mathsf{var}\ \frac{P \in \Gamma}{\mathsf{Val}\, P\, \Gamma} \qquad \mathsf{thunk}\ \frac{\mathsf{Tm}\, N\, \Gamma}{\mathsf{Val}\, (\mathit{Thunk}\, N)\, \Gamma}$$

$$\mathsf{unit}^+\ \frac{}{\mathsf{Val}\, 1\, \Gamma} \qquad \mathsf{pair}^+\ \frac{\mathsf{Val}\, P_1\, \Gamma \qquad \mathsf{Val}\, P_2\, \Gamma}{\mathsf{Val}\, (P_1 \times P_2)\, \Gamma}$$

$$\mathsf{inj}_i\ \frac{\mathsf{Val}\, P_i\, \Gamma}{\mathsf{Val}\, (P_1 + P_2)\, \Gamma}$$

$$\mathsf{ret}\ \frac{\mathsf{Val}\, P\, \Gamma}{\mathsf{Tm}\, (\mathit{Comp}\, P)\, \Gamma} \qquad \mathsf{abs}\ \frac{\mathsf{Tm}\, N\, (\Gamma.P)}{\mathsf{Tm}\, (P \Rightarrow N)\, \Gamma}$$

$$\mathsf{unit}^-\ \frac{}{\mathsf{Tm}\, \top\, \Gamma} \qquad \mathsf{pair}^-\ \frac{\mathsf{Tm}\, N_1\, \Gamma \qquad \mathsf{Tm}\, N_2\, \Gamma}{\mathsf{Tm}\, (N_1 \,\&\, N_2)\, \Gamma}$$

$$\mathsf{force}\ \frac{\mathsf{Val}\, (\mathit{Thunk}\, N)\, \Gamma}{\mathsf{Tm}\, N\, \Gamma}$$

$$\mathsf{app}\ \frac{\mathsf{Tm}\, (P \Rightarrow N)\, \Gamma \qquad \mathsf{Val}\, P\, \Gamma}{\mathsf{Tm}\, N\, \Gamma} \qquad \mathsf{prj}_i\ \frac{\mathsf{Tm}\, (N_1 \,\&\, N_2)\, \Gamma}{\mathsf{Tm}\, N_i\, \Gamma}$$

$$\mathsf{bind}\ \frac{\mathsf{Tm}\, (\mathit{Comp}\, P)\, \Gamma \qquad \mathsf{Tm}\, N\, (\Gamma.P)}{\mathsf{Tm}\, N\, \Gamma}$$

$$\mathsf{split}\ \frac{\mathsf{Val}\, (P_1 \times P_2)\, \Gamma \qquad \mathsf{Tm}\, N\, (\Gamma.P_1.P_2)}{\mathsf{Tm}\, N\, \Gamma} \qquad \mathsf{abort}\ \frac{\mathsf{Val}\, 0\, \Gamma}{\mathsf{Tm}\, N\, \Gamma}$$

$$\mathsf{case}\ \frac{\mathsf{Val}\, (P_1 + P_2)\, \Gamma \qquad \mathsf{Tm}\, N\, (\Gamma.P_1) \qquad \mathsf{Tm}\, N\, (\Gamma.P_2)}{\mathsf{Tm}\, N\, \Gamma}$$

Figure 4: Value and computation terms (CBPV)

Figure 5 defines interpretation of values $(\!| v : \mathsf{Val}\, P\, \Gamma |\!) : [\![\Gamma]\!] \mathrel{\dot{\to}} [\![P]\!]$ and terms $(\!| t : \mathsf{Tm}\, N\, \Gamma |\!) : [\![\Gamma]\!] \mathrel{\dot{\to}} [\![N]\!]$. It is straightforward, thanks to the pioneering work of Moggi [23] and the design of CBPV by Levy [19]. Since *Thunk* serves only as an embedding of negative into positive types in our semantics, we interpret thunking and forcing by the identity. The eliminations split, case and abort for positive types deal now only with values, thus, need not reference

$$
\begin{array}{lcl}
(\!|\mathsf{var}\,x|\!)_\gamma &=& \mathsf{lookup}\,x\,\gamma \\
(\!|\mathsf{unit}^+|\!)_\gamma &=& () \\
(\!|\mathsf{pair}^+\,v_1\,v_2|\!)_\gamma &=& ((\!|v_1|\!)_\gamma,\ (\!|v_2|\!)_\gamma) \\
(\!|\mathsf{inj}_i\ v|\!)_\gamma &=& \iota_i(\!|v|\!)_\gamma \\
(\!|\mathsf{thunk}\,t|\!)_\gamma &=& (\!|t|\!)_\gamma \\
\hdashline
(\!|\mathsf{abs}\,t|\!)_\gamma &=& \lambda(\!|t|\!)_\gamma \\
(\!|\mathsf{unit}^-|\!)_\gamma &=& () \\
(\!|\mathsf{pair}^-\ t_1\,t_2|\!)_\gamma &=& ((\!|t_1|\!)_\gamma,\ (\!|t_2|\!)_\gamma) \\
\hdashline
(\!|\mathsf{app}\,t\,v|\!)_\gamma &=& (\!|t|\!)_\gamma\ \mathsf{id}\ (\!|v|\!)_\gamma \\
(\!|\mathsf{prj}_i\ t|\!)_\gamma &=& \pi_i(\!|t|\!)_\gamma \\
(\!|\mathsf{force}\,v|\!)_\gamma &=& (\!|v|\!)_\gamma \\
\hdashline
(\!|\mathsf{split}\,v\,t|\!)_\gamma &=& (\lambda\,(a_1,a_2).\ (\!|t|\!)_{(\gamma,a_1,a_2)})\ (\!|v|\!)_\gamma \\
(\!|\mathsf{case}\,v\,t_1\,t_2|\!)_\gamma &=& [\lambda a_1.\ (\!|t_1|\!)_{(\gamma,a_1)},\ \lambda a_2.\ (\!|t_2|\!)_{(\gamma,a_2)}]\ (\!|v|\!)_\gamma \\
(\!|\mathsf{abort}\,v|\!)_\gamma &=& \mathsf{magic}\ (\!|v|\!)_\gamma \\
\hdashline
(\!|\mathsf{ret}\,v|\!)_\gamma &=& \mathsf{return}^C\ (\!|v|\!)_\gamma \\
(\!|\mathsf{bind}\,u\,t|\!)_\gamma &=& \mathsf{run}^C\ (\widehat{\mathsf{map}}^C\ \lambda(\!|t|\!)_\gamma\ (\!|u|\!)_\gamma)
\end{array}
$$

Figure 5: Evaluation (CBPV).

the monad operations. The use of the monad is confined to ret and bind. Note the availability of $\mathsf{run}^C : C[\![N]\!] \to [\![N]\!]$ at any negative type N for the interpretation of bind.

3.3 Normal forms and normalization

The design of normal forms for pure CBPV follows the same principles as for the STLC with sums in Section 2.2. We prevent β-redexes, i.e., eliminations of a type immediately following its introduction, by restricting the terms in elimination positions to neutrals. Due to the focused nature of CBPV, neutrals of positive type are just variables. We achieve η-long forms by only embedding neutrals of negative base type o^- into normal forms. Maximal focusing for positive types is achieved by only admitting variables of base type o^+ in values. In the following, we present the grammar for normal forms in detail.

Positive normal forms are values $v : \mathsf{Vnf}\,P\,\Gamma$ referring only to atomic variables and whose thunks only contain negative normal forms.

$$
\mathsf{var}\ \frac{o^+ \in \Gamma}{\mathsf{Vnf}\ o^+\ \Gamma} \qquad \mathsf{thunk}\ \frac{\mathsf{Nf}\ N\ \Gamma}{\mathsf{Vnf}\ (\mathit{Thunk}\ N)\ \Gamma}
$$

$$
\mathsf{unit}^+\ \frac{}{\mathsf{Vnf}\ 1\ \Gamma} \qquad \mathsf{pair}^+\ \frac{\mathsf{Vnf}\ P_1\ \Gamma \qquad \mathsf{Vnf}\ P_2\ \Gamma}{\mathsf{Vnf}\ (P_1 \times P_2)\ \Gamma}
$$

$$
\mathsf{inj}_i\ \frac{\mathsf{Vnf}\ P_i\ \Gamma}{\mathsf{Vnf}\ (P_1 + P_2)\ \Gamma}
$$

Neutral normal forms $\boxed{\mathsf{Ne}\ N\ \Gamma}$ are negative eliminations starting from a forced *Thunk* rather than from variables of negative types (as those do not exist in CBPV). However, due to normality the *Thunk* cannot be a thunk, but only a variable $\mathit{Thunk}\,N \in \Gamma$.

$$
\mathsf{force}\ \frac{\mathit{Thunk}\ N \in \Gamma}{\mathsf{Ne}\ N\ \Gamma} \qquad \mathsf{prj}_i\ \frac{\mathsf{Ne}\ (N_1\ \&\ N_2)\ \Gamma}{\mathsf{Ne}\ N_i\ \Gamma}
$$

$$
\mathsf{app}\ \frac{\mathsf{Ne}\ (P \Rightarrow N)\ \Gamma \qquad \mathsf{Vnf}\ P\ \Gamma}{\mathsf{Ne}\ N\ \Gamma}
$$

Variables are originally introduced by either abs or the binding of a neutral of type $\mathit{Comp}\,P$ to a new variable of type P. Variables of composite value type can be broken down by pattern matching, introducing variables of smaller type. These positive eliminations plus bind are organized in the inductively defined strong monad $\boxed{\mathsf{Cov}}$. In the following rules, parameter $\mathcal{J}$ (for "judgement") stands for an arbitrary presheaf.

$$
\mathsf{return}\ \frac{\mathcal{J}\ \Gamma}{\mathsf{Cov}\ \mathcal{J}\ \Gamma} \qquad \mathsf{bind}\ \frac{\mathsf{Ne}\ (\mathit{Comp}\ P)\ \Gamma \qquad \mathsf{Cov}\ \mathcal{J}\ (\Gamma.P)}{\mathsf{Cov}\ \mathcal{J}\ \Gamma}
$$

$$
\mathsf{split}\ \frac{P_1 \times P_2 \in \Gamma \qquad \mathsf{Cov}\ \mathcal{J}\ (\Gamma.P_1.P_2)}{\mathsf{Cov}\ \mathcal{J}\ \Gamma} \qquad \mathsf{abort}\ \frac{0 \in \Gamma}{\mathsf{Cov}\ \mathcal{J}\ \Gamma}
$$

$$
\mathsf{case}\ \frac{P_1 + P_2 \in \Gamma \qquad \mathsf{Cov}\ \mathcal{J}\ (\Gamma.P_1) \qquad \mathsf{Cov}\ \mathcal{J}\ (\Gamma.P_2)}{\mathsf{Cov}\ \mathcal{J}\ \Gamma}
$$

Finally, normal forms of negative types are defined as inductive family $\boxed{\mathsf{Nf}\ N\ \Gamma}$. They are generated by maximal negative introduction (abs, pair^-, unit^-) until a negative atom or $\mathit{Comp}\,P$ is reached. Then, elimination of neutrals and variables is possible through the Cov monad until an answer can be given in form of a base neutral ($\mathsf{Ne}\ o^-$) or a normal value.

$$
\mathsf{ne}\ \frac{\mathsf{Cov}\ (\mathsf{Ne}\ o^-)\ \Gamma}{\mathsf{Nf}\ o^-\ \Gamma} \qquad \mathsf{ret}\ \frac{\mathsf{Cov}\ (\mathsf{Vnf}\ P)\ \Gamma}{\mathsf{Nf}\ (\mathit{Comp}\ P)\ \Gamma}
$$

$$
\mathsf{unit}^-\ \frac{}{\mathsf{Nf}\ \top\ \Gamma} \qquad \mathsf{pair}^-\ \frac{\mathsf{Nf}\ N_1\ \Gamma \qquad \mathsf{Nf}\ N_2\ \Gamma}{\mathsf{Nf}\ (N_1\ \&\ N_2)\ \Gamma}
$$

$$
\mathsf{abs}\ \frac{\mathsf{Nf}\ N\ \Gamma.P}{\mathsf{Nf}\ (P \Rightarrow N)\ \Gamma}
$$

Note that we do not have a function $\mathsf{runNf}^{\mathsf{Cov}}$ this time,[5] instead, we directly employ Cov in the definition of Nf at the base cases o^- and $\mathit{Comp}\,P$.

Reification $\downarrow^P : [\![P]\!] \dot{\to} \mathsf{Vnf}\,P$ at positive types P produces a normal value, and $\downarrow^N : [\![N]\!] \dot{\to} \mathsf{Nf}\,N$ at negative types N a normal term. During reification of function types $P \Rightarrow N$ in context Γ we need to embed a fresh variable $x : P \in (\Gamma.P)$ into $[\![P]\!]$, breaking down P to positive atoms o^+ and negative remainders $\mathit{Thunk}\,N$. However, in $[\![P]\!]$ we do not have case analysis available, thus, positive reflection $\uparrow^P_\Gamma : P \in \Gamma \to \mathsf{Cov}\ [\![P]\!]\ \Gamma$ needs to run in the monad. Luckily, we can unwind the monad using run^N before we recursively reify at type N. Negative reflection $\uparrow^N : \mathsf{Ne}\,N \dot{\to} [\![N]\!]$ is, as before, generalized from variables to neutrals, to handle the breaking down of

[5] A trivial $\mathsf{runNf}^{\mathsf{Cov}} : \mathsf{Cov}\ (\mathsf{Nf}\ N_0) \dot{\to} \mathsf{Nf}\ N_0$ for $N_0 ::= o^- \mid \mathit{Comp}\,P$ could be given by induction on the cover, essentially being a join. However its extension to arbitrary negative types N would fail for the case of pushing a binder (bind, split or case) under an abstraction. This is because our base category Cxt of order-preserving embeddings does not permit swapping of variables in the context.

N via eliminations. In the following definition of reflection we use the abbreviation $\boxed{\mathsf{fresh}^P_\Gamma = \uparrow^P_{\Gamma.P} \mathsf{v}_0 : \mathsf{Cov}\,[\![P]\!]\,(\Gamma.P)}$.

$$
\begin{array}{lcll}
\uparrow^P_\Gamma & & : & P \in \Gamma \to \mathsf{Cov}\,[\![P]\!]\,\Gamma \\
\uparrow^{o^+}_\Gamma & x & = & x \\
\uparrow^{1}_\Gamma & x & = & \mathsf{return}\,() \\
\uparrow^{P_1\times P_2}_\Gamma & x & = & \mathsf{split}\,x\,\left((\uparrow^{P_1}_{\Gamma.P_1.P_2}\,\mathsf{v}_1)\star(\uparrow^{P_2}_{\Gamma.P_1.P_2}\,\mathsf{v}_0)\right) \\
\uparrow^{0}_\Gamma & x & = & \mathsf{abort}\,x \\
\uparrow^{P_1+P_2}_\Gamma & x & = & \mathsf{case}\,x\,(\mathsf{map}\,\iota_1\,\mathsf{fresh}^{P_1}_\Gamma)\,(\mathsf{map}\,\iota_2\,\mathsf{fresh}^{P_2}_\Gamma) \\
\uparrow^{\mathit{Thunk}\,N}_\Gamma & x & = & \mathsf{return}\,(\uparrow^N_\Gamma(\mathsf{force}\,x))
\end{array}
$$

$$
\begin{array}{lcll}
\downarrow^P & & : & [\![P]\!] \dot\to \mathsf{Vnf}\,P \\
\downarrow^{o^+} & & = & \mathsf{var} \\
\downarrow^{1}_\Gamma & () & = & \mathsf{unit}^+ \\
\downarrow^{P_1\times P_2}_\Gamma & (a_1,a_2) & = & \mathsf{pair}^+\,(\downarrow^{P_1}_\Gamma a_1)\,(\downarrow^{P_2}_\Gamma a_2) \\
\downarrow^{0} & & = & \mathsf{magic} \\
\downarrow^{P_1+P_2} & & = & [\mathsf{inj}_1\circ\downarrow^{P_1},\ \mathsf{inj}_2\circ\downarrow^{P_2}] \\
\downarrow^{\mathit{Thunk}\,N} & & = & \mathsf{thunk}\circ\downarrow^N
\end{array}
$$

Reflection at positive pairs uses monoidal functoriality $C\,\mathcal{A}_1 \dot\to C\,\mathcal{A}_2 \dot\to C\,(\mathcal{A}_1 \,\hat\times\, \mathcal{A}_2)$ called $\star$ by McBride and Paterson [21, Section 7], which in our case can be defined by e. g. $c_1 \star c_2 = \mathsf{join}\,(\widehat{\mathsf{map}}\,(\lambda\,\tau_1\,a_1.\,\widehat{\mathsf{map}}\,(\lambda\,\tau_2\,a_2.\,(\mathsf{ren}\,\tau_2\,a_1,\ a_2))\,(\mathsf{ren}\,\tau_1\,c_2))\,c_1)$.

For negative types, reflection and reification works as before:

$$
\begin{array}{lcll}
\uparrow^N & & : & \mathsf{Ne}\,N \dot\to [\![N]\!] \\
\uparrow^{\mathit{Comp}\,P}_\Gamma & u & = & \mathsf{bind}\,u\,\mathsf{fresh}^P_\Gamma \\
\uparrow^{o^-}_\Gamma & u & = & \mathsf{return}\,u \\
\uparrow^{\top}_\Gamma & u & = & () \\
\uparrow^{N_1\&N_2}_\Gamma & u & = & (\uparrow^{N_1}_\Gamma\,(\mathsf{prj}_1\,u),\ \uparrow^{N_2}_\Gamma\,(\mathsf{prj}_2\,u))
\end{array}
$$

$$
\begin{array}{lcll}
\downarrow^N & & : & [\![N]\!] \dot\to \mathsf{Nf}\,N \\
\downarrow^{\mathit{Comp}\,P}_\Gamma & c & = & \mathsf{map}\,(\downarrow^P)\,c \\
\downarrow^{o^-}_\Gamma & c & = & \mathsf{ne}\,c \\
\downarrow^{\top}_\Gamma & () & = & \mathsf{unit}^- \\
\downarrow^{N_1\&N_2}_\Gamma & (b_1,b_2) & = & \mathsf{pair}^-\,(\downarrow^{N_1}_\Gamma\,b_1)\,(\downarrow^{N_2}_\Gamma\,b_2)
\end{array}
$$

Reflection for function types is also unchanged, except that app expects a value argument now.

$$
\begin{array}{l}
\uparrow^{P\Rightarrow N}_\Gamma u = \lambda_\Delta\,\tau\,a.\,\uparrow^N_\Delta(\mathsf{app}\,(\mathsf{ren}\,\tau\,u)\,(\downarrow^P_\Delta a)) \\
\downarrow^{P\Rightarrow N}_\Gamma f = (\mathsf{abs}\circ\downarrow^N_{\Gamma.P}\circ\mathsf{run}^N_{\Gamma.P} \\
\qquad\qquad \circ\,\widehat{\mathsf{map}}\,(\lambda\,\tau\,a.\,f\,(\mathsf{wk}^P \fatsemi \tau)\,a))\,\mathsf{fresh}^P_\Gamma
\end{array}
$$

For reification of a (Kripke) function $f : [\![P \Rightarrow N]\!]\Gamma$ we extend context Γ to $\Gamma.P$ and create a case tree $\mathsf{fresh}^P_\Gamma : \mathsf{Cov}\,[\![P]\!]\,(\Gamma.P)$ representing the new variable. Using strong functoriality $\widehat{\mathsf{map}}$, we then apply f to all leaves a of that case tree, which may live in further extended contexts Δ reachable by $\tau : \Gamma.P \subseteq \Delta$. The resulting case tree $\mathsf{Cov}\,[\![N]\!]\,(\Gamma.P)$ is then run giving us a value in $[\![N]\!](\Gamma.P)$, which can be reified to a $\mathsf{Nf}\;N\;(\Gamma.P)$. Finally, abstraction gives the desired normal form in $\mathsf{Nf}\,(P \Rightarrow N)\,\Gamma$.

This time, the identity environment $\mathsf{fresh}^\Gamma : \mathsf{Cov}\,[\![\Gamma]\!]\,\Gamma$ can only be generated in the monad, due to monadic positive reflection.

$$
\begin{array}{lcl}
\mathsf{fresh}^\varepsilon & = & \mathsf{return}\,() \\
\mathsf{fresh}^{\Gamma.P} & = & (\mathsf{ren}\,\mathsf{wk}^P\,\mathsf{fresh}^\Gamma)\star\mathsf{fresh}^P_\Gamma
\end{array}
$$

Putting things together, we obtain the normalization function

$$\mathsf{norm}\,(t : \mathsf{Tm}\,N\,\Gamma) = \left(\downarrow^N_\Gamma \circ \mathsf{run}^N_\Gamma \circ \mathsf{map}\,(\!|t|\!)\right)\mathsf{fresh}^\Gamma.$$

Taking stock, we have arrived at normal forms that eagerly introduce (Nf) and eliminate (Ne) negative types and also eagerly introduce positive types (Vnf). However, the elimination of positive types is still rather non-deterministic: It is possible to only partially break up a composite positive type and leave smaller, but still composite positive types for later pattern matching. The last refinement of normal forms, chaining also the positive eliminations, will be discussed in the following section.

4 FOCUSED INTUITIONISTIC PROPOSITIONAL LOGIC

Polarized lambda-calculus [27, 29] is a focused calculus, it eagerly employs so-called *invertible* rules: the introduction rules for negative types and the elimination rules for positive types. As a consequence of the latter, variables are either of atomic or negative type. Types in contexts Γ, Δ are of one of the forms o^+ or N.

To add a variable of positive type P to the context, we need to break it apart until only atoms and negative bits remain. This is performed by maximal pattern matching, called the left-invertible phase of focalization.[6] We express maximal pattern matching on P as a strong functor $[P]$ in the category of presheaves, mapping a presheaf $\mathcal{J}$ ("judgement") to $[P]\mathcal{J}$ and a presheaf morphism $f : (\mathcal{J} \mathrel{\hat\Rightarrow} \mathcal{K})\Gamma$ to $\widehat{\mathsf{map}}^{[P]}_\Gamma f : [P]\,\mathcal{J}\,\Gamma \to [P]\,\mathcal{K}\,\Gamma$. For arbitrary $\mathcal{J}$ and Γ, the family $\boxed{[P]\,\mathcal{J}\,\Gamma}$ is inductively constructed by the following rules:

$$
\mathsf{hyp}^+\,\frac{\mathcal{J}\,(\Gamma.o^+)}{[o^+]\,\mathcal{J}\,\Gamma}
\qquad
\mathsf{hyp}^-\,\frac{\mathcal{J}\,(\Gamma.N)}{[\mathit{Thunk}\,N]\,\mathcal{J}\,\Gamma}
$$

$$
\mathsf{branch}_0\,\frac{}{[0]\,\mathcal{J}\,\Gamma}
\qquad
\mathsf{branch}_2\,\frac{[P_1]\,\mathcal{J}\,\Gamma \qquad [P_2]\,\mathcal{J}\,\Gamma}{[P_1+P_2]\,\mathcal{J}\,\Gamma}
$$

$$
\mathsf{split}_0\,\frac{\mathcal{J}\,\Gamma}{[1]\,\mathcal{J}\,\Gamma}
\qquad
\mathsf{split}_2\,\frac{[P_1]\,([P_2]\,\mathcal{J})\,\Gamma}{[P_1\times P_2]\,\mathcal{J}\,\Gamma}
$$

Note the recursive occurrence of $[P_2]$ as argument to $[P_1]$ in split_2, which makes $[P]$ a nested datatype [10]. Agda supports such nested inductive types; but note that $[P]$ is uncontroversial, since it could also be defined by recursion on P. It is tempting to name split_2 "join" and split_0 "return" since $[P]$ is a graded monad on the monoid $(1, \times)$ of product types; however, this coincidence shall not matter for our further considerations.

Remark 2. *The notation $[P]$ is chosen in analogy of the next-time operator of dynamic logic [24]. The deeper reason is that the category* Cxt *can be seen as branching time where a context represents a point in time and an extension a possible future. The modality $[P]$ picks a*

[6] Filinski [15, Section 4] achieves maximal pattern matching through an additional, ordered context Θ for positive variables which are eagerly split.

certain future, the "proposition" $[P]\,\mathcal{J}$ *states that* $\mathcal{J}$ *should hold in the future determined by* P.

While in dynamic logic P *would be a regular expression, our positive types represent the fragment featuring choice* $(0, +)$ *and sequence* $(1, \times)$. *The introduction rules for* $[P]\,\mathcal{J}$ *resemble the axioms of choice* $(\mathsf{branch}_0, \mathsf{branch}_2)$ *and sequence* $(\mathsf{split}_0, \mathsf{split}_2)$ *in dynamic logic.*

Focalization is a technique to remove don't-care non-determinism from proof search, and as such, polarized lambda calculus is foremost a calculus of normal forms. These normal forms are given by four mutually defined inductive families of presheaves $\mathsf{Vnf}\,P$, $\mathsf{Ne}\,N$, $\mathsf{Cov}\,\mathcal{J}$, and $\mathsf{Nf}\,N$. As they are very similar to the CBPV normal forms given in the last section, we only report the differences. Values $\boxed{v : \mathsf{Val}\,P\,\Gamma}$ are unchanged, they can refer to atomic positive hypotheses (var^+) and normal thunks. Neutrals $\boxed{\mathsf{Ne}\ N\ \Gamma}$ start with a negative variable instead of with force, as forcing thunks is already performed in hyp^- when adding hypotheses of *Thunk* type. The normal forms $\boxed{\mathsf{Nf}\ N\ \Gamma}$ of negative type are unchanged with the exception that pattern matching happens eagerly in abs, by virtue of $[P]$.

$$\mathsf{var}^-\ \frac{N \in \Gamma}{\mathsf{Ne}\ N\ \Gamma} \qquad\qquad \mathsf{abs}\ \frac{[P]\,(\mathsf{Nf}\,N)\,\Gamma}{\mathsf{Nf}\,(P \Rightarrow N)\,\Gamma}$$

The Cover monad $\boxed{\mathsf{Cov}\,\mathcal{J}\,\Gamma}$ lacks constructors split, case and abort since the pattern matching is taken care of by $[P]$.

$$\mathsf{return}\ \frac{\mathcal{J}\,\Gamma}{\mathsf{Cov}\,\mathcal{J}\,\Gamma} \quad \mathsf{bind}\ \frac{\mathsf{Ne}\,(\mathit{Comp}\,P)\,\Gamma \quad [P]\,(\mathsf{Cov}\,\mathcal{J})\,\Gamma}{\mathsf{Cov}\,\mathcal{J}\,\Gamma}$$

All these inductive families are presheaves, however, due to the factored presentation using $[P]$ and Cov, the proof is not a simple mutual induction. Yet, in Agda, the generic proof goes through using a sized typing for these inductive families. Similarly, defining the join for monad Cov relies on sized typing [1].

$$\begin{array}{lcl}
\mathsf{join} & : & \forall i.\ \mathsf{Cov}^i\,(\mathsf{Cov}^\infty\,\mathcal{J}) \dot\to \mathsf{Cov}^\infty\,\mathcal{J} \\
\mathsf{join}^{i+1}\,(\mathsf{return}^i\ c) & = & c \\
\mathsf{join}^{i+1}\,(\mathsf{bind}^i\ t\ k) & = & \mathsf{bind}^\infty\ (t : \mathsf{Ne}\ P\ \Gamma) \\
\quad (\widehat{\mathsf{map}}^{[P]}_\Gamma\ \mathsf{join}^i\ (k : [P]\,(\mathsf{Cov}^i\,\mathcal{J})\,\Gamma)) & &
\end{array}$$

Herein, we used the sized typing of the constructors of Cov:

$$\begin{array}{ll}
\mathsf{return} & : \forall i.\ \mathcal{J} \dot\to \mathsf{Cov}^{i+1}\,\mathcal{J} \\
\mathsf{bind} & : \forall i.\ \mathsf{Ne}\,(\mathit{Comp}\,P) \dot\to [P]\,(\mathsf{Cov}^i\,\mathcal{J}) \dot\to \mathsf{Cov}^{i+1}\,\mathcal{J}
\end{array}$$

Due to the eager splitting of positive hypotheses, reflection at type P now lives in the graded monad $[P]$ rather than Cov. Further, as pattern matching may produce $n \geq 0$ cases, reflection cannot simply produce a single positive semantic value; instead, one such value is needed for every branch. We implement $\mathsf{reflect}^P : ([\![P]\!] \dot\Rightarrow \mathcal{J}) \dot\to [P]\mathcal{J}$ as a higher-order function expecting a continuation k which is invoked for each generated branch with the semantic value of type P constructed for this branch, see Figure 6.

Reflecting at a positive atomic type o^+ is the regular ending of a reflection pass: we call continuation k with a fresh variable var^+ zero of type o^+, making space for the variable using wk^{o^+}. In case we end at type *Thunk* N, we add a new variable var^- zero of type N and pass it to k, after full η-expansion via $\uparrow^N$. Two more endings are possible: At type 0, we have reached an absurd case, meaning that no continuation is necessary since we can conclude with *ex falso quodlibet*. At type 1, there is no need to add a new variable, as values of type 1 contain no information. We simply pass the unit value () to k in this case. Reflecting at $P_1 + P_2$ generates two branches, which may result in several uses of the continuation k. In the first branch, we recursively reflect at P_1. Its continuation will receive a semantic value in $[\![P_1]\!]$, which we inject via ι_1 into $[\![P_1 + P_2]\!]$ to pass it to k. The second branch proceeds analogously. Finally reflecting at $P_1 \times P_2$ means we first have to analyze P_1, and in each of the generated branches we continue to analyze P_2. Thus $\mathsf{reflect}^{P_2}$ is passed as a continuation to $\mathsf{reflect}^{P_1}$. Each reflection phase gives us a semantic value a_i of type P_i, which we combine to a tuple before passing it to k. Note also that the context extension τ_1 created in the first phase needs to be composed with the context extension τ_2 of the second phase to transport k into the final context. Further, the value a_1 was constructed relative to the target of τ_1 and still needs to be transported with τ_2 before being paired up with a_2.

$$\begin{array}{llcl}
\mathsf{reflect}^P & & : & ([\![P]\!] \dot\Rightarrow \mathcal{J}) \dot\to [P]\mathcal{J} \\
\mathsf{reflect}^{o^+} & k & = & \mathsf{hyp}^+(k\ \mathsf{wk}^{o^+}\ (\mathsf{var}^+\ \mathsf{zero})) \\
\mathsf{reflect}^{\mathit{Thunk}\,N} & k & = & \mathsf{hyp}^-(k\ \mathsf{wk}^N(\uparrow^N(\mathsf{var}^-\ \mathsf{zero}))) \\
\mathsf{reflect}^0 & k & = & \mathsf{branch}_0 \\
\mathsf{reflect}^{P_1+P_2} & k & = & \mathsf{branch}_2\,(\mathsf{reflect}^{P_1}\ (\lambda\,\tau.\ k\,\tau \circ \iota_1)) \\
 & & & \qquad\qquad (\mathsf{reflect}^{P_2}\ (\lambda\,\tau.\ k\,\tau \circ \iota_2)) \\
\mathsf{reflect}^1 & k & = & \mathsf{split}_0\ (k\ \mathsf{id}\ ()) \\
\mathsf{reflect}^{P_1\times P_2} & k & = & \mathsf{split}_2\,(\mathsf{reflect}^{P_1}(\lambda\,\tau_1\,a_1. \\
 & & & \qquad \mathsf{reflect}^{P_2}(\lambda\,\tau_2\,a_2. \\
 & & & \qquad k\,(\tau_1 \mathbin{⨾} \tau_2)\,(\mathsf{ren}\ \tau_2\,a_1,\ a_2))))
\end{array}$$

Figure 6: Positive reflection (polarized lambda calculus).

The method $\mathsf{reflect}^P$ replaces previous uses of fresh^P in reflection and reification at negative types.

$$\begin{array}{l}
\downarrow^{P\Rightarrow N}_\Gamma\ f = \mathsf{abs}\left(\mathsf{reflect}^P_\Gamma\left(\lambda_\Delta\,(\tau : \Gamma \subseteq \Delta)\ a.\ \downarrow^P_\Delta(f\ \tau\ a)\right)\right) \\
\uparrow^{\mathit{Comp}\,P}_\Gamma\ u = \mathsf{bind}\ u\left(\mathsf{reflect}^P_\Gamma\ (\lambda\ \tau\ a.\ \mathsf{return}\ a)\right)
\end{array}$$

Due to the absence of composite positive types in contexts, the identity environment fresh^Γ can be built straightforwardly using negative reflection.

$$\begin{array}{lcl}
\mathsf{fresh}^\Gamma & : & [\![\Gamma]\!]\Gamma \\
\mathsf{fresh}^\varepsilon & = & () \\
\mathsf{fresh}^{\Gamma.o^+} & = & (\mathsf{ren}\ \mathsf{wk}^{o^+}\ \mathsf{fresh}^\Gamma,\ \mathsf{var}^+\ \mathsf{zero}) \\
\mathsf{fresh}^{\Gamma.N} & = & (\mathsf{ren}\ \mathsf{wk}^N\ \mathsf{fresh}^\Gamma,\ \uparrow^N_\Gamma(\mathsf{var}^-\ \mathsf{zero}))
\end{array}$$

The terms $\boxed{\mathsf{Tm}\,N\,\Gamma}$ of the polarized lambda calculus are the ones of CBPV minus the positive eliminations (split, case, abort), the added negative variable rule (var^-), and the necessary changes to the binders abs and bind.

$$\mathsf{var}^-\ \frac{N \in \Gamma}{\mathsf{Tm}\,N\,\Gamma} \qquad \mathsf{abs}\ \frac{[P]\,(\mathsf{Tm}\,N)\,\Gamma}{\mathsf{Tm}\,(P \Rightarrow N)\,\Gamma}$$

$$\mathsf{bind}\ \frac{\mathsf{Tm}\,(\mathit{Comp}\,P)\,\Gamma \qquad [P]\,(\mathsf{Tm}\,N)\,\Gamma}{\mathsf{Tm}\,N\,\Gamma}$$

Term interpretation $\boxed{(\!|\_|\!) : \mathsf{Tm}\, N\, \Gamma \to [\![\Gamma]\!] \dot{\to} [\![N]\!]}$ shall be as for CBPV except that we need to exchange the interpretation function for binders

$$\lambda(\!|\_|\!) : \mathsf{Tm}\, N\, (\Gamma.P) \to [\![\Gamma]\!] \dot{\to} [\![P \Rightarrow N]\!].$$

Since a binder for P performs a maximal splitting on P and takes the form of a function defined by a case (and split) tree, applying it to a value v of type P amounts to a complete matching of v against the case tree and binding the remaining atomic and negative crumbs. This matching can be defined for a generic evaluation function of type $\mathsf{Ev}\, \mathcal{J}\, \Delta\, \Gamma = [\![\Gamma]\!]\Delta \to \mathcal{J}\, \Delta$.

$$\begin{array}{llcl}
\mathsf{match} & : [\![P]\!]\Delta \to [P]\,(\mathsf{Ev}\,\mathcal{J}\,\Delta) \dot{\to} \mathsf{Ev}\,\mathcal{J}\,\Delta & & \\
\mathsf{match}\ x & (\mathsf{hyp}^+\ e) \quad\quad \gamma & = & e\,(\gamma,\ \mathsf{return}\ x) \\
\mathsf{match}\ b & (\mathsf{hyp}^-\ e) \quad\quad \gamma & = & e\,(\gamma,\ b) \\
\mathsf{match}\ a & \mathsf{branch}_0 \quad\quad\quad \gamma & = & \mathsf{magic}\ a \\
\mathsf{match}\ (\iota_1\ a_1) & (\mathsf{branch}_2\ e_1\ e_2)\ \gamma & = & \mathsf{match}\ a_1\ e_1\ \gamma \\
\mathsf{match}\ (\iota_2\ a_2) & (\mathsf{branch}_2\ e_1\ e_2)\ \gamma & = & \mathsf{match}\ a_2\ e_2\ \gamma \\
\mathsf{match}\ () & (\mathsf{split}_0\ e) \quad\quad \gamma & = & e\,\gamma \\
\mathsf{match}\ (a_1, a_2) & (\mathsf{split}_2\ e) \quad\quad \gamma & = & \\
\multicolumn{4}{l}{\quad \mathsf{match}\ a_1\ (\mathsf{map}^{[P]}\,(\mathsf{match}\ a_2)\ e)\ \gamma}
\end{array}$$

With instantiations $\mathcal{J} = [\![N]\!]$ and $(\!|\_|\!) : \mathsf{Tm}\, N \dot{\to} \mathsf{Ev}\,[\![N]\!]\,\Delta$, the interpretation $\lambda(\!|\, t\,|\!)$ of binder $t : [P]\,(\mathsf{Tm}\, N)\,\Gamma$ is defined as follows:

$$\lambda(\!|t|\!)_{(\gamma:[\![\Gamma]\!]\Delta)}\,(\tau : \Delta \subseteq \Phi)\,(a : [\![P]\!]\Phi) \ = \\ \mathsf{match}\ a\ (\mathsf{map}^{[P]}\ (\!|\_|\!)\ t)\ (\mathsf{ren}\,\tau\,\gamma)$$

This completes the definition of the normalization function $\mathsf{norm}\,(t : \mathsf{Tm}\, N\, \Gamma) = \downarrow^N (\!|t|\!)_{\mathsf{fresh}^\Gamma}$.

5 CONCLUSION AND FURTHER WORK

We have defined NbE for CBPV and polarized lambda calculus, formulated with intrinsically well-typed syntax and presheaf semantics. At the heart of our development stands the notion of a cover monad, as alternative to the more common sheaf semantics, to handle sum types.

As a side result, we have proven semantically that the normal forms of both systems are logically complete, i. e., each derivable judgement $\Gamma \vdash N$ has a normal derivation. It remains to show that NbE for these calculi is also computationally sound and complete, i.e., the computational behavior of term and normal form should agree, and normalization should decide a suitable equational theory on terms. To this end, we may switch from the category of presheaves to the category of Kripke predicates, as investigated through an Agda formalization.[7]

Additionally, a natural question to investigate is whether known CBN and CBV NbE algorithms can be obtained from our NbE algorithms by embedding simply-typed lambda calculus into our polarized calculi, using known CBN and CBV translations. Further, we would like to study the NbE algorithm for STLC arising from the optimal translation, i.e., the one inserting a minimal amount of *Thunk* and *Comp* transitions. Finally, we might revisit our interpretation of *Thunk* as a monoidal comonad as suggested by the categorical models of modal logic S4 [2].

[7] https://andreasabel.github.io/ipl/html/NbeModel.html

ACKNOWLEDGMENTS

The first author took initial inspiration from some unpublished notes by Thorsten Altenkirch titled *Another topological completeness proof for intuitionistic logic* received by email on 16th March 2000. Thorsten in turn credits his inspiration to an ALF proof by Thierry Coquand. Thanks to Hugo Herbelin for encouraging feedback and spotting some typos. Thanks to the anonymous referees of previous versions of this paper for the valuable feedback and remarks that helped improving the text of this article.

The first author acknowledges financial support from VR Grant 2014-04864 *Termination Certificates for Dependently-Typed Programs and Proofs via Refinement Types* and the EU Cost Action CA15123 EUTYPES *Types for Programming and Verification*.

REFERENCES

[1] Andreas Abel. 2006. *A Polymorphic Lambda-Calculus with Sized Higher-Order Types*. Ph.D. Dissertation. Ludwig-Maximilians-Universität München.

[2] Natasha Alechina, Michael Mendler, Valeria de Paiva, and Eike Ritter. 2001. Categorical and Kripke Semantics for Constructive S4 Modal Logic. In *Computer Science Logic, 18th International Workshop, CSL 2004, 13th Annual Conference of the EACSL, Karpacz, Poland, September 20-24, 2004, Proceedings (Lecture Notes in Computer Science)*, Laurent Fribourg (Ed.), Vol. 2142. Springer, 292–307. https://doi.org/10.1007/3-540-44802-0_21

[3] Thorsten Altenkirch, Peter Dybjer, Martin Hofmann, and Philip J. Scott. 2001. Normalization by Evaluation for Typed Lambda Calculus with Coproducts. In *16th IEEE Symposium on Logic in Computer Science (LICS 2001), 16-19 June 2001, Boston University, USA, Proceedings*. IEEE Computer Society Press, 303–310. https://doi.org/10.1109/LICS.2001.932506

[4] Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. 1995. Categorical Reconstruction of a Reduction Free Normalization Proof. In *Category Theory and Computer Science, 6th International Conference, CTCS '95, Cambridge, UK, August 7-11, 1995, Proceedings (Lecture Notes in Computer Science)*, David H. Pitt, David E. Rydeheard, and Peter Johnstone (Eds.), Vol. 953. Springer, 182–199. https://doi.org/10.1007/3-540-60164-3_27

[5] Thorsten Altenkirch and Tarmo Uustalu. 2004. Normalization by Evaluation for $\lambda^{\to 2}$. In *Functional and Logic Programming, 7th International Symposium, FLOPS 2004, Nara, Japan, April 7-9, 2004, Proceedings (Lecture Notes in Computer Science)*, Yukiyoshi Kameyama and Peter J. Stuckey (Eds.), Vol. 2998. Springer, 260–275. https://doi.org/10.1007/978-3-540-24754-8_19

[6] Jean-Marc Andreoli. 1992. Logic Programming with Focusing Proofs in Linear Logic. *Journal of Logic and Computation* 2, 3 (1992), 297–347. https://doi.org/10.1093/logcom/2.3.297

[7] Vincent Balat, Roberto Di Cosmo, and Marcelo P. Fiore. 2004. Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, Neil D. Jones and Xavier Leroy (Eds.). ACM Press, 64–76. https://doi.org/10.1145/964001.964007

[8] Freiric Barral. 2008. *Decidability for non-standard conversions in lambda-calculus*. Ph.D. Dissertation. Ludwig-Maximilians-University Munich.

[9] Ulrich Berger and Helmut Schwichtenberg. 1991. An Inverse to the Evaluation Functional for Typed λ-calculus. In *Sixth Annual Symposium on Logic in Computer Science (LICS '91), July, 1991, Amsterdam, The Netherlands, Proceedings*. IEEE Computer Society Press, 203–211. https://doi.org/10.1109/LICS.1991.151645

[10] Richard S. Bird and Lambert G. L. T. Meertens. 1998. Nested Datatypes. In *Mathematics of Program Construction, MPC'98, Proceedings (Lecture Notes in Computer Science)*, Johan Jeuring (Ed.), Vol. 1422. Springer, 52–67. https://doi.org/10.1007/BFb0054285

[11] Taus Brock-Nannestad and Carsten Schürmann. 2010. Focused Natural Deduction. In *Logic for Programming, Artificial Intelligence, and Reasoning - 17th International Conference, LPAR-17, Yogyakarta, Indonesia, October 10-15, 2010. Proceedings (Lecture Notes in Computer Science)*, Christian G. Fermüller and Andrei Voronkov (Eds.), Vol. 6397. Springer, 157–171. https://doi.org/10.1007/978-3-642-16242-8_12

[12] Catarina Coquand. 1993. From Semantics to Rules: A Machine Assisted Analysis. In *Computer Science Logic, 7th Workshop, CSL '93, Swansea, United Kingdom, September 13-17, 1993, Selected Papers (Lecture Notes in Computer Science)*, Egon Börger, Yuri Gurevich, and Karl Meinke (Eds.), Vol. 832. Springer, 91–105. https://doi.org/10.1007/BFb0049326

[13] Olivier Danvy. 1996. Type-Directed Partial Evaluation. In *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, Hans-Juergen Boehm and Guy L. Steele Jr. (Eds.). ACM

Press, 242–257. https://doi.org/10.1145/237721.237784

[14] Andrzej Filinski. 1999. A Semantic Account of Type-Directed Partial Evaluation. In *Principles and Practice of Declarative Programming, International Conference, PPDP'99, Paris, France, September 29 - October 1, 1999, Proceedings (Lecture Notes in Computer Science)*, Gopalan Nadathur (Ed.), Vol. 1702. Springer, 378–395. https://doi.org/10.1007/10704567_23

[15] Andrzej Filinski. 2001. Normalization by Evaluation for the Computational Lambda-Calculus. In *Typed Lambda Calculi and Applications, 5th International Conference, TLCA 2001, Krakow, Poland, May 2-5, 2001, Proceedings (Lecture Notes in Computer Science)*, Samson Abramsky (Ed.), Vol. 2044. Springer, 151–165. https://doi.org/10.1007/3-540-45413-6_15

[16] Yannick Forster, Steven Schäfer, Simon Spies, and Kathrin Stark. 2019. Call-By-Push-Value in Coq: Operational, Equational, and Denotational Theory. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, Assia Mahboubi and Magnus O. Myreen (Eds.). ACM Press, 118–131. https://doi.org/10.1145/3293880.3294097

[17] Felix Joachimski and Ralph Matthes. 2003. Short proofs of normalization for the simply-typed lambda-calculus, permutative conversions and Gödel's T. *Archive of Mathematical Logic* 42, 1 (2003), 59–87. https://doi.org/10.1007/s00153-002-0156-9

[18] Neelakantan R. Krishnaswami. 2009. Focusing on pattern matching. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, Zhong Shao and Benjamin C. Pierce (Eds.). ACM Press, 366–378. https://doi.org/10.1145/1480881.1480927

[19] Paul Blain Levy. 2006. Call-by-push-value: Decomposing call-by-value and call-by-name. *Journal of Higher-Order and Symbolic Computation* 19, 4 (2006), 377–414. https://doi.org/10.1007/s10990-006-0480-6

[20] Chuck Liang and Dale Miller. 2007. Focusing and Polarization in Intuitionistic Logic. In *Computer Science Logic, 21st International Workshop, CSL 2007, 16th Annual Conference of the EACSL, Lausanne, Switzerland, September 11-15, 2007, Proceedings (Lecture Notes in Computer Science)*, Jacques Duparc and Thomas A. Henzinger (Eds.), Vol. 4646. Springer, 451–465. https://doi.org/10.1007/978-3-540-74915-8_34

[21] Conor McBride and Ross Paterson. 2008. Applicative programming with effects. *Journal of Functional Programming* 18, 1 (2008), 1–13. https://doi.org/10.1017/S0956796807006326

[22] John Mitchell. 1996. *Foundations of Programming Languages.* MIT Press.

[23] Eugenio Moggi. 1991. Notions of Computation and Monads. *Information and Computation* 93, 1 (1991), 55–92. https://doi.org/10.1016/0890-5401(91)90052-4

[24] Vaughan R. Pratt. 1976. Semantical Considerations on Floyd-Hoare Logic. In *17th Annual Symposium on Foundations of Computer Science, Houston, Texas, USA, 25-27 October 1976.* IEEE Computer Society Press, 109–121. https://doi.org/10.1109/SFCS.1976.27

[25] Dag Prawitz. 1965. *Natural Deduction.* Almqvist & Wiksell, Stockholm.

[26] Christine Rizkallah, Dmitri Garbuzov, and Steve Zdancewic. 2018. A Formal Equational Theory for Call-By-Push-Value. In *Interactive Theorem Proving - 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings (Lecture Notes in Computer Science)*, Jeremy Avigad and Assia Mahboubi (Eds.), Vol. 10895. Springer, 523–541. https://doi.org/10.1007/978-3-319-94821-8_31

[27] José Espírito Santo. 2017. The Polarized λ-calculus. *Electronic Notes in Theoretical Computer Science* 332 (2017), 149–168. https://doi.org/10.1016/j.entcs.2017.04.010

[28] Gabriel Scherer. 2017. Deciding equivalence with sums and the empty type. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM Press, 374–386. https://doi.org/10.1145/3009837

[29] Noam Zeilberger. 2009. *The Logical Basis of Evaluation Order and Pattern-Matching.* Ph.D. Dissertation. Carnegie Mellon University.