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

Downloaded from: https://research.chalmers.se, 2021-11-06 04:48 UTC

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

N.B. When citing this work, cite the original published paper.
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

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 NBø 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 NBø for CBPV (Section 3). In contrast to the normal forms of CBV NBø— which is the algorithmic counterpart of the completeness proof for intuitionistic propositional logic (IPL) using Beth models— CBPV NBø 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 NBø thanks to extensionality ($\eta$) 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 NBø 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/NModelMonad.html), 3 (in a variation, see https://andreasabel.github.io/ipl/html-cbpv/NCBPU.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 (NBø) 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 NBø for pure, effect-free CBPV. Therein, we observe that the structures of CBPV to control effects naturally structure the NBø process as well.
- Finally, we define NBø 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 $:\vdash$ 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 $\Gamma$ is just a snoc list of simple types $A$, meaning we write context extension as $\Gamma.A$, and the empty context as $\varepsilon$. Membership $\left[ A \in \Gamma \right]$ and sublists relations $\Gamma \subseteq \Delta$ are given inductively by the following rules:

\[
\begin{align*}
\text{zero} & : A \in \varepsilon.A \\
\text{suc} & : A \in \Gamma.A \quad \Rightarrow \quad A \in \Gamma.B
\end{align*}
\]

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

We can define

\[
\begin{align*}
\text{id} & : \Gamma \subseteq \Gamma \\
\text{id} & : \Gamma \subseteq A \Rightarrow A \subseteq \Phi \Rightarrow \Gamma \subseteq \Phi
\end{align*}
\]

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

In category theory, a presheaf $\mathcal{A}$ over a category $\mathcal{C}$ is a contravariant functor from $\mathcal{C}$ into the category Set of sets and functions, i.e., a (covariant) functor $\mathcal{C}$ 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 Cxt$^{\text{op}}$. For example, given any type $A$,...
we can consider $A \epsilon_\_\_\_$ as such a presheaf, with the action on objects mapping $\Gamma$ to the set $A \in \Gamma$ of indices of $A$ in $\Gamma$, and the action on morphisms being reindex: $\Gamma \subseteq \Delta \rightarrow A \in \Gamma \rightarrow A \in \Delta$. The associated functor laws reindex id $x \equiv x$ and reindex $t_2$ (reindex $t_1$ $x$) $\equiv$ reindex $(t_1 \downarrow t_2)$ $x$ 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.

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

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

$$
\begin{align*}
& \text{var} \quad \frac{A \in \Gamma}{A \in \Gamma} \\
& \text{abs} \quad \frac{B + 1 \Rightarrow A \in \Gamma}{A \Rightarrow B + 1 \Rightarrow A + \Gamma} \\
& \text{app} \quad \frac{A \Rightarrow B + 1 \Rightarrow A + \Gamma}{B + 1 \Rightarrow A + \Gamma} \\
& \text{unit} \quad \frac{1 \{A \Rightarrow \Gamma\}}{1 \{A + \Gamma\}} \\
& \text{pair} \quad \frac{A_1 \times A_2 + \Gamma}{A_1 + \Gamma \times A_2 + \Gamma} \\
& \text{prj}_1 \quad \frac{A_1 + \Gamma \times A_2 + \Gamma}{A_1 + \Gamma} \\
& \text{prj}_2 \quad \frac{A_1 + \Gamma \times A_2 + \Gamma}{A_2 + \Gamma} \\
& \text{inj}_1 \quad \frac{A_1 + \Gamma}{A_1 + A_2 + \Gamma} \\
& \text{inj}_2 \quad \frac{A_2 + \Gamma}{A_1 + A_2 + \Gamma} \\
& \text{case} \quad \frac{A_1 + A_2 + \Gamma}{A_1 + \Gamma} \\
& \text{abort} \quad \frac{0 + \Gamma}{B + 1 \Rightarrow A + \Gamma}
\end{align*}
$$

The skilled eye of the reader will immediately recognize the proof rules of intuitionistic propositional logic (IPL) under the Curry-Howard isomorphism, where $A + \Gamma$ is to be read as "$A$ follows from $\Gamma$". Using shorthand $\nu n = \text{var}(\text{succ}^n \text{zero})$ for the $n$th variable, a term such as

$$
\text{abs} \left( \text{abs} \left( \text{pair} \right) (\text{abs} (\text{pair} \nu_1 \text{abs} (\text{pair} \nu_1 \nu_0))) \right)
$$

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 $\text{A}^\_\_\_$ as witnessed by the standard weakening operation$^1$

$$
\begin{align*}
& \text{ren} \quad \frac{\Gamma \subseteq \Delta \rightarrow A + \Gamma \rightarrow A + \Delta}{\Gamma \rightarrow A + \Gamma \rightarrow A + \Delta}
\end{align*}
$$

defined by recursion over $\tau : A + \Gamma$, and functor laws for ren analogously to reindex.

Normal forms$^2$ are logically characterized as those fulfilling the subformula property $[17, 25]$. Normal forms $\begin{bmatrix} n : \text{Na} \ \text{A} \ \text{Gamma} \end{bmatrix}$ are mutually defined with neutral normal forms $\begin{bmatrix} u : \text{Ne} \ \text{A} \ \text{Gamma} \end{bmatrix}$ in the following inductive definition, we reuse the rule names from the term constructors.

$$
\begin{align*}
\text{var} \quad \frac{A \in \Gamma}{\text{Ne} (A \Rightarrow \Gamma)} \\
& \text{app} \quad \frac{\text{Ne} (A \Rightarrow B) \Gamma \ \text{Ne} A \ \text{Gamma} \Gamma}{\text{Ne} B \ \text{Gamma} \Gamma} \\
& \text{unit} \quad \frac{\text{Ne} 1 \ \text{Gamma} \Gamma}{\text{Ne} (A \Rightarrow B) \Gamma} \\
& \text{pair} \quad \frac{\text{Ne} A_1 \ \text{Gamma} \Gamma \ \text{Ne} A_2 \ \text{Gamma} \Gamma}{\text{Ne} (A_1 + A_2) \Gamma} \\
& \text{prj}_1 \quad \frac{\text{Ne} A_1 \ \text{Gamma} \Gamma \ \text{Ne} (A_1 + A_2) \Gamma}{\text{Ne} A_1 \ \text{Gamma} \Gamma} \\
& \text{prj}_2 \quad \frac{\text{Ne} A_2 \ \text{Gamma} \Gamma \ \text{Ne} (A_1 + A_2) \Gamma}{\text{Ne} A_2 \ \text{Gamma} \Gamma}
\end{align*}
$$

These rules only allow the elimination of neutrals; this restriction guarantees the subformula property and prevents any kind of computational (beta) redex. The new rule $\text{ne}$ embeds $\text{Ne}$ into $\text{Na}$, 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 $\eta$-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 ($\text{inj}_1$ or $\text{inj}_2$).

As expected, $\text{Ne} A$ and $\text{Na} A$ 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

$$
\begin{align*}
\begin{bmatrix} n : \text{Na} \ \text{A} \ \text{Gamma} \end{bmatrix} & \rightarrow \begin{bmatrix} n : \text{Ne} \ \text{A} \ \text{Gamma} \end{bmatrix} \\
\begin{bmatrix} u : \text{Ne} \ \text{A} \ \text{Gamma} \end{bmatrix} & \rightarrow \begin{bmatrix} u : \text{Na} \ \text{A} \ \text{Gamma} \end{bmatrix}
\end{align*}
$$

that discards constructor $\text{ne}$ but keeps all other constructors. This erasure function naturally commutes with reindexing, making it a natural transformation between the presheaves $\text{Na} A$ ($\text{Ne} A$, resp.) and $\text{A} \downarrow \_$. We shall simply write, for instance, $\text{Na} A \rightarrow \text{A} \downarrow \_$ 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 $\text{A} \rightarrow \text{B} \rightarrow \text{C}$ for $\forall \text{A} . \text{A} \rightarrow (\text{B} \rightarrow \text{C} \rightarrow \text{A})$.

We use the $\left[ \bigvee \right]$ 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 $f : \text{A} \rightarrow \text{B}$ silently, e.g., when $a : \text{A} \text{Gamma}$ then $f a : \text{B} \text{Gamma}$. For clarity, we may sometimes provide the instantiation explicitly via subscript, e.g., $f_a : \text{A} \text{Gamma} \rightarrow \text{B} \text{Gamma}$ and $f_{a b} : \text{B} \text{Gamma}$. Similarly, to introduce a polymorphic function we may provide the implicit abstraction as subscript, such as in $\lambda a . b : \text{A} \rightarrow \text{B}$ or $\lambda a : \text{A} \text{Gamma} , b : \text{A} \rightarrow \text{B}$, or omit it, e.g., $\lambda (a : \text{A} \text{Gamma}) , b : \text{A} \rightarrow \text{B}$.
The role of evaluation is to produce from a term unique inhabitant, mapped to the presheaf reification evaluation. Instead of presheaves, we work with Kripke predicates there. The main contribution of reification does not change the interpretation of a term as function in the meta-language. Defined mutually with reification by induction on type from neutrals to unknowns is called unknowns in the semantics, types the free indices of is finally reified to a normal form. Since we are evaluating open branches, case generalizes analogously, with a bit of care when weakening the negative types is admissible, for instance:

\[
\text{abs}(A \Rightarrow B) u = (\text{abs}(B)(\text{ren \ wk}^A u))
\]

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

2.3 Normalization

Normalization is concerned with finding a normal form \( n : \text{Nf} A \Gamma \) for each term \( t : A + \Gamma \). The normal form should be sound, i.e., \( \forall t \in T \) with respect to an equational theory \( \equiv \) on terms. Further, normalization should decide \( \equiv \), i.e., terms \( t, t' \) with \( t \equiv t' \) should have the same normal form \( n \). In this article, we present only the normalization function \( \text{norm} : A + \Gamma \rightarrow \text{Nf} A \Gamma \) without proving its soundness and completeness. From a logical perspective, we will compute for each derivation of \( A + \Gamma \) a normal derivation \( \text{Nf} A \Gamma \). The method normalization by evaluation (NbE)

\[
\text{norm}(t : A + \Gamma) = [\Gamma](t) \text{fresh}^t
\]

decomposes normalization into evaluation

\[
[\Gamma] : (t : A + \Gamma) \rightarrow [[\Gamma]] \rightarrow [[A]]
\]

in the identity environment

\[
\text{fresh}^\Gamma : [[\Gamma]]
\]

followed by reification

\[
\downarrow^A : [[A]] \rightarrow \text{Nf} A
\]

(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 \( ^t \) 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 \( \text{Ne} o \) with the intention that the neutrals take the role of the unknowns. The mapping \( \uparrow^A : \text{Ne} A \rightarrow [[A]] \) 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 \( I \) denote the unit set and \( (.) \) its unique inhabitant, \( 0 \) the empty set and magic \( 0 \rightarrow T \) the ex-falso-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 \( \pi_i : S_1 \times S_2 \rightarrow S_i \), and their disjoint sum \( S_1 + S_2 \) with injections \( i_j : S_j \rightarrow S_1 + S_2 \) and elimination \( [f_1, f_2] : S_1 + S_2 \rightarrow T \) for arbitrary \( f_i : S_i \rightarrow T \).

Presheaf (co)products \( 0, 1, +, \) and \( \times \) are constructed pointwise, e.g., \( \Gamma = \emptyset \), and given two presheaves \( A \) and \( B \), \((A + B)\Gamma = A\Gamma + B\Gamma \). For the exponential of presheaves, however, we need the Kripke function space \((A \Rightarrow B) \Gamma = \forall A. \Gamma \subseteq A \rightarrow A \Rightarrow B \Gamma \).

We will interpret simple types \( A \) as corresponding presheaves \([A]\). Let us start with the negative types, defining reflection \( \uparrow^A : \text{Ne} A \rightarrow [[A]] \) and reification \( \downarrow^A : [[A]] \rightarrow \text{Nf} A \) along the way.

\[
\uparrow^1 = \hat{1} \\
\uparrow^A u = (\lambda \Delta (r : \Gamma \subseteq \Delta) a. \uparrow^B \text{app}(\text{ren } r \text{ } u) (\downarrow^A a)) \\
\downarrow^A f = \text{abs}(B \text{ } \text{wk}^A \text{ fresh}^t)
\]

In the reification at function types \( \downarrow^A \Rightarrow B \), the renaming \( \text{wk}^A \text{ fresh}^t \) gives us an inhabitant of the empty set, which means that reflection at the empty type would not be definable. Similarly, the setting \( [[A + B]] = [[A]] + [[B]] \) is refuted by fresh\( ^t\) \( A \Rightarrow B \) \( + \) \( B \Rightarrow (A+B) \), 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 \( [0] = \text{Ne} 0 \) works in the presence of sums, as we would not be able to interpret the term abs (case \( v_0 \) \( \emptyset \) \( \emptyset \) \( : \) \( o + o \) ) \( \Rightarrow o \) in our semantics, because \( o \) \( o + o \) 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 \( C \) which can cover the cases.

A direct extension of our presheaf semantics to positive types cannot work. For instance, with \([0] = \emptyset \), simply \( \text{fresh}^0 : 0 \) would give us an inhabitant of the empty set, which means that reflection at the empty type would not be definable. Similarly, the setting \( [[A + B]] = [[A]] + [[B]] \) is refuted by fresh\( ^t\) \( A \Rightarrow B \) \( + \) \( B \Rightarrow (A+B) \), 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 \( [0] = \text{Ne} 0 \) works in the presence of sums, as we would not be able to interpret the term abs (case \( v_0 \) \( \emptyset \) \( \emptyset \) \( : \) \( o + o \) ) \( \Rightarrow o \) in our semantics, because \( o \) \( o + o \) 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 \( C \) which can cover the cases.

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

We are looking for a monad \( C \), called a cover monad, that takes the role of the oracle alluded to in the introduction, and offers us

\( S_1 \), and their disjoint sum \( S_1 + S_2 \) with injections \( i_j : S_j \rightarrow S_1 + S_2 \) and elimination \( [f_1, f_2] : S_1 + S_2 \rightarrow T \) for arbitrary \( f_i : S_i \rightarrow T \).
the following services:

\[
\begin{align*}
\text{runNf}^C : C(NF P) & \rightarrow NF P \\
\text{abort}^C : Ne 0 & \rightarrow C B \\
\text{case}^C_\bot : Ne (A_1 + A_2) & \Gamma \\
& \rightarrow C B(\Gamma, A_1) \rightarrow C B(\Gamma, A_2) \rightarrow C B \Gamma
\end{align*}
\]

Method runNf^C enables us to run a monadic computation of a normal form, C(NF P) \Gamma, and delivers a normal form NF P \Gamma. While runNf^C is a left inverse of return^C, the converse is not true. Instead, the information contained in C is reified and becomes part of the normal form delivered by runNf^C. Technically, (NF P, runNf^C) is a monad algebra for monad C. Besides runNf^C \circ return^C = id_{NF P}, the square runNf^C \circ \text{join}^C = runNf^C \circ \text{map}^C runNf^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 map^C runNf^C, and then the outer one.

Method abort^C allows us to end a computation by exhibiting an inconsistency witnessed by a neutral term u : Ne 0 \Gamma. Remember that, ultimately, computations in C produce a normal form extracted by runNf^C. With abort^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 case^C_\bot allows us in a computation C B \Gamma, to ask the oracle about a neutral term Ne (A_1 + A_2) \Gamma of sum type. We have to supply handlers for both possible answers: a computation C 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 C B (\Gamma, A_2) for the case of a right injection.

\[
\begin{align*}
\llbracket 0 \rrbracket &= C(Ne 0) \\
\llbracket 1 \rrbracket &= \text{return}^C \\
\llbracket 1^* \rrbracket &= \text{runNf}^C \circ \text{map}^C \text{ne} \\
\llbracket 0 \rrbracket &= C \hat 0 \\
\llbracket 0^* \rrbracket &= \text{return}^C \\
\llbracket 0^0 \rrbracket &= \text{runNf}^C \circ \text{map}^C \text{magic} \\
\llbracket A + B \rrbracket &= C \llbracket A \rrbracket \oplus \llbracket B \rrbracket \\
\llbracket \text{case}^C \theta u \text{return}^C (t_1 \text{fresh}^A) \rrbracket &= \text{runNf}^C \circ \text{map}^C \text{inj}_1 \circ \downarrow^A, \text{inj}_2 \circ \downarrow^B \\
\end{align*}
\]

Figure 1: Interpretation of positive types.

The similarity of the monad services abort^C and case^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 return^{Cov}, abort^{Cov}, and case^{Cov}. One can visualize an element e : Cov 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 \emptyset (see abort), or by an element of A (see return). Functionality 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 join^{Cov}, which flattens a 2-level case tree, i.e., a case tree with case trees as leaves, into a single one. Finally, runNf^{Cov} is a simple recursion on the tree, replacing case^{Cov} and abort^{Cov} by the case and abort constructions on normal forms, and return^{Cov} by the identity.

Using the services of a generic cov monad C, we can complete our semantics, see Figure 1.

Since positive types P have a monadic interpretation, there is a monad algebra C [P] \rightarrow [P], which is simply join^C. It can be extended to a monad algebra run^P : C [A] \rightarrow [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.

\[
\begin{align*}
\text{run}^A & : C[A] \rightarrow [A] \\
\text{run}^B & : c = \text{join}^C c \\
\text{run}^1 & : c = () \\
\text{run}^{A \times B} & : c = (\text{run}^A(\text{map}^C \pi_1 c), \text{run}^B(\text{map}^C \pi_2 c)) \\
\text{run}^{A \rightarrow B} & : c = \lambda \Delta \tau a. \text{run}^B(\text{map}^C (\lambda \phi \tau' f. f \text{ id ren } \tau')) (\text{ren } \tau c)
\end{align*}
\]

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

We extend the type interpretation pointwise to contexts, i.e., \llbracket e \rrbracket = \hat 1 and \llbracket \Gamma.A \rrbracket = \llbracket \Gamma \rrbracket \times \llbracket A \rrbracket, and obtain a natural projection function lookup (x : A \in \Gamma) : \llbracket \Gamma \rrbracket \Rightarrow \llbracket A \rrbracket from the semantic environments. The evaluation function (\ell : A + \Gamma) : \llbracket \Gamma \rrbracket \Rightarrow \llbracket A \rrbracket can now be defined by recursion on \ell, see Figure 2. Herein, the environment \ell lives in \llbracket \Gamma \Delta \rrbracket, thus, (\ell \gamma) : \llbracket \Gamma \Delta \rrbracket. For the interpretation of the binders abs and case we use the mutually defined \lambda \ell \gamma. The coproduct eliminations (abort) and (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 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 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

4In categorical terminology, the existence of run^A means that all semantic types \llbracket A \rrbracket fulfill the weak sheaf condition, aka weak pasting.
We have obtained a computable function \( P \) weighted monad is a strong monad. The method \( \text{runNF} \) : \( \text{CC} (\text{NF} P) \rightarrow \text{NF} P \) exists by definition, using the identity continuation \( \text{NF} P \Rightarrow \text{NF} P \). In the following, we demonstrate that \( \text{CC} \) enables matching on neutrals:

\[
\begin{align*}
\text{abort}_{\text{CC}}^\Gamma (u : \text{Ne} 0 \Gamma) &= \text{CC} \, B \Gamma \\
\text{case}_{\text{CC}}^\Gamma u_A (\tau : \Gamma \leq \Delta) (k : (B \Rightarrow \text{NF} P) \Delta) &= \text{case} (\text{ren} \, \tau \, u) \\
\text{where} \quad n_1 : \text{NF} P (\Delta, A_1) \\
n_2 = c_i (\text{lift}^{A_1} \tau : \Gamma, A_1 \leq \Delta, A_1) \\
(\lambda \phi (\tau' : \Delta, A_1 \leq \Phi) (j : B \Phi). k (\text{wk}^{A_1} \tau' j))
\end{align*}
\]

It is noteworthy (yet unsurprising) that \( \text{abort} \) \( \text{CC} \) discards continuation \( k \), while case \( \text{CC} \) 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 \( \text{NbE} \) algorithm using \( \text{CC} \) 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 \( B \). yet we shall stick to our terminology which
is common in publications on focalization. However, we shall use
\textit{Thunk} for switch $\downarrow$ and \textit{Comp} for switch $\uparrow$, to avoid confusion with our notation for reflection and reification.

\[
\begin{align*}
\text{Ty}^+ & \ni P ::= \sigma^+ \mid 1 \mid P_1 \times P_2 \mid 0 \mid P_1 + P_2 \mid \text{Thunk} N \\
\text{Ty}^- & \ni N ::= \sigma^- \mid \top \mid N_1 \& N_2 \mid P \Rightarrow N \mid \text{Comp} P
\end{align*}
\]

CBPV uses $U$ for \textit{Thunk} and $F$ for \textit{Comp}, however, we find these names uninspiring unless you have good knowledge of the intended model. Further, CGBP [19] employs labeled sums $\Sigma_i(P_i)_{i \in I}$ and labeled records $\Pi_i(P_i)_{i \in I}$ 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, CGBP does not consider uninterpreted base types; in anticipation of the next section, we add them as both positive atoms ($\sigma^+$) and negative atoms ($\sigma^-$).

Getting a bit ahead of ourselves, let us consider the mutually defined interpretations $\llbracket P \rrbracket$ and $\llbracket N \rrbracket$ of positive and negative types as presheaves (see Figure 3). This interpretation is parameterized by a strong monad $C$ on presheaves. Semantically, we do not distinguish between positive and negative products. Notably, sum types can now be interpreted as plain (pointwise) presheaf sums. The \textit{Thunk} marker is ignored, yet \textit{Comp}, marking the switch from the negative to the positive type interpretation, places the monad $C$. Positive atoms $\sigma^+$, standing for value types without constructors, are only inhabited by variables $x : \sigma^+ \in \Gamma$. Negative atoms $\sigma^-$ 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 $\text{Ne} \sigma^-$, which we shall define below. The multiplication $\text{join}^C$ of the strong monad $C$ can be extended to a monad algebra $\text{run}^N : C[\llbracket N \rrbracket] \to [\llbracket N \rrbracket]$ 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 \textit{monadic}.

Contexts are lists of positive types since in CBPV variables stand for values. Interpretation of contexts $\llbracket \Gamma \rrbracket$ is again defined pointwise $\llbracket \varepsilon \rrbracket = \hat{\varepsilon}$ and $\llbracket \Gamma.P \rrbracket = \llbracket \Gamma \rrbracket \times \llbracket P \rrbracket$.

\subsection{Terms and evaluation}

Value terms $\llbracket v : \text{Val} P \Gamma \rrbracket$ or short, \textit{values}, and computation terms $\llbracket t : \text{Tm} N \Gamma \rrbracket$ or short, \textit{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 $\text{Thunk} N$ are embedded by force. Further, values of type $P$ can be embedded via $\text{ret}$, producing a term of type $\text{Comp} P$. Such terms are eliminated by $\text{bind}$ which is, unlike the usual monadic bind, not only available for $\text{Comp}$-types but for arbitrary negative types $N$. This is justified by the monadic character of negative types, by virtue of $\text{run}^N$.

\begin{figure}[h]
\centering
\begin{align*}
\text{var} & \quad P \in \Gamma \quad & \text{thunk} & \quad \text{Tm} N \Gamma \\
\text{Val} P \Gamma & \quad \text{Val} (\text{Thunk} N) \Gamma \\
\text{unit}^+ & \quad \text{Val} (\text{Thunk} N) \Gamma \\
\text{inj}^+_1 & \quad \text{Val} P_1 \Gamma \\
\text{inj}^+_2 & \quad \text{Val} (P_1 \times P_2) \Gamma \\
\text{ret} & \quad \text{Val} P \Gamma \\
\text{Val} (\text{Comp} P) \Gamma & \quad \text{Val} (P \Rightarrow N) \Gamma \\
\text{force} & \quad \text{Val} (\text{Thunk} N) \Gamma \\
\text{app} & \quad \text{Val} P \Gamma \\
\text{Val} (\text{Thunk} N) \Gamma & \quad \text{Val} (N \Rightarrow N) \Gamma \\
\text{bind} & \quad \text{Tm} (\text{Comp} P) \Gamma \\
\text{split} & \quad \text{Tm} (P \Rightarrow N) \Gamma \\
\text{case} & \quad \text{Tm} (N \Rightarrow N) \Gamma \\
\end{align*}
\caption{Interpretation of polarized types}
\end{figure}

\begin{figure}[h]
\centering
\begin{align*}
\text{var} & \quad \text{Val} P \Gamma \\
\text{unit}^+ & \quad \text{Val} (P_1 \times P_2) \Gamma \\
\text{inj}^+_1 & \quad \text{Val} P_1 \Gamma \\
\text{inj}^+_2 & \quad \text{Val} (P_1 \times P_2) \Gamma \\
\text{split} & \quad \text{Val} P_1 \Gamma \\
\text{app} & \quad \text{Tm} (P \Rightarrow N) \Gamma \\
\text{bind} & \quad \text{Tm} (\text{Comp} P) \Gamma \\
\text{force} & \quad \text{Tm} (\text{Thunk} N) \Gamma \\
\text{case} & \quad \text{Tm} (N \Rightarrow N) \Gamma \\
\end{align*}
\caption{Value and computation terms (CBPV)}
\end{figure}

Figure 5 defines interpretation of values $\llbracket v : \text{Val} P \Gamma \rrbracket$ and terms $\llbracket t : \text{Tm} N \Gamma \rrbracket$ in CBPV and terms $\llbracket t : \text{Tm} N \Gamma \rrbracket$ and terms $\llbracket t : \text{Tm} N \Gamma \rrbracket$. It is straightforward, thanks to the pioneering work of Moggi [23] and the design of CBPV by Levy [19]. Since \textit{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
the monad operations. The use of the monad is confined to ret and bind. Note the availability of run\(^C\) : \(\mathbb{C}[N] \rightarrow \mathbb{N}[N]\) at any negative type \(N\) for the interpretation of bind.

3.3 Normal forms and normalization

The design of normal forms for pure CBPV is based on the same principles as for the STLC with sums in Section 2.2. We prevent \(\beta\)-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 \(\eta\)-long forms by only embedding neutrals of negative base type \(\sigma^+\) into normal forms. Maximal focusing for positive types is achieved by only admitting variables of base type \(\sigma^+\) in values. In the following, we present the grammar for normal forms in detail.

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

\[
\begin{align*}
\text{var} \quad &\sigma^+ \in \Gamma \\
\text{Vnf} \quad &\text{thunk} \\
\text{unit}^+ \quad &\text{pair}^+ \\
\text{inj}_i \quad &\text{Vnf} \ P_i \ \Gamma
\end{align*}
\]

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

\[
\begin{align*}
\text{for} &\quad \text{Thunk} \ N \in \Gamma \\
\text{Ne} &\quad \text{Nf} \ N \ \Gamma \\
\text{prj}_i &\quad \text{Ne} \ (N_1 \ &\ N_2) \ \Gamma \\
\text{app} &\quad \text{Ne} \ (P \Rightarrow \ N) \ \Gamma \\
\end{align*}
\]

Variables are originally introduced by either abs or the binding of a neutral of type \(\text{Comp} P\) 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 \(\text{Cov}\).

In the following rules, parameter \(\mathcal{J}\) (for “judgement”) stands for an arbitrary presheaf.

\[
\begin{align*}
\text{return} &\quad \mathcal{J} \ \Gamma \quad \text{bind} \\
\text{Cov} &\quad \text{Ne} \ (\text{Comp} P \ \Gamma) \quad \text{Cov} \ (\mathcal{J}.P) \\
\text{split} &\quad \text{Cov} \ (\mathcal{J}.P_1.P_2) \quad \text{abort} \\
\text{case} &\quad \text{Cov} \ (\mathcal{J}.P_1) \quad \text{Cov} \ (\mathcal{J}.P_2)
\end{align*}
\]

Finally, normal forms of negative types are defined as inductive family \(\text{Nf} \ N \ \Gamma\). They are generated by maximal negative introduction (abs, pair\(^-\), unit\(^-\)) until a negative atom or \(\text{Comp} P\) is reached.

Then, elimination of neutrals and variables is possible through the \(\text{Cov}\) monad until an answer can be given in form of a base neutral \(\text{Ne} \sigma^-\) or a normal value.

\[
\begin{align*}
\text{ne} &\quad \text{Cov} \ (\text{Ne} \sigma^-) \ \Gamma \\
\text{ret} &\quad \text{Cov} \ (\text{Vnf} \ P) \ \Gamma \\
\text{unit}^- &\quad \text{pair}^- \\
\text{abs} &\quad \text{Ne} \ N \ \Gamma \ P \\
\end{align*}
\]

Note that we do not have a function \(\text{runNf}^\text{Cov}\) this time,\(^5\) instead, we directly employ \(\text{Cov}\) in the definition of \(\text{Nf}\) at the base cases \(\sigma^-\) and \(\text{Comp} P\).

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

\(^5\)A trivial \(\text{runNf}^\text{Cov}\) : \(\text{Cov} \ (\text{Ne} \ N)\) \rightarrow \text{Nf} \ N\) for \(N_0 := \sigma^-\) \(\Rightarrow \text{Comp} P\) 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 \(\text{Cov}\) 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 \( \text{fresh}^P \Gamma = \tau^P \Gamma \circ V_0 : \text{Cov} \Gamma (\Gamma.P) \).

\[
\begin{align*}
\tau^P \Gamma & : P \in \Gamma \rightarrow \text{Cov} \Gamma (\Gamma.P) \\
\tau^P \Gamma x & = x \\
\tau^P \Gamma P \times P & : \text{split} x \left( (\tau^P \Gamma.P.P_1) \star (\tau^P \Gamma.P.P_2) \right) \\
\tau^P \Gamma x & = \text{return} () \\
\tau^P \Gamma P_1 + P_2 & : \text{case} x \left( \text{map} \ i_1 \text{fresh}^P \Gamma \right) \left( \text{map} \ i_2 \text{fresh}^P \Gamma \right) \\
\tau^P \Gamma \text{Thunk} N & = \text{return} (\tau^N \Gamma (\text{force} x))
\end{align*}
\]

Reflection at positive pairs uses monoidal functoriality \( C \mathcal{A}_1 \rightarrow C \mathcal{A}_2 \rightarrow C (\mathcal{A}_1 \times \mathcal{A}_2) \) called \( \star \) by McBride and Paterson [21, Section 7], which in our case can be defined by e.g. \( c_1 \star c_2 = \text{join} (\text{map} (\lambda \tau \alpha_1. \text{map} (\lambda \tau \alpha_2. \text{run} \tau \alpha_1 \alpha_2)) (\text{run} \tau \alpha_1 \alpha_2)) (\text{run} \tau \alpha_1 \alpha_2)). \)

For negative types, reflection and reification works as before:

\[
\begin{align*}
\downarrow^N \Gamma \text{Comp} P & : \text{Ne} \in \Gamma \rightarrow \text{Vnf} P \\
\downarrow^N \Gamma \text{u} & = \text{bind} \text{u} \text{fresh}^\Gamma \Gamma \\
\downarrow^N \Gamma \text{c} & = \text{return} \text{c} \\
\downarrow^N \Gamma \text{u} & = () \\
\downarrow^N \Gamma \text{u} & = (\text{map} (\lambda \tau \alpha. \text{map} (\lambda \tau \alpha. \text{run} \tau \alpha))) (\text{run} \tau \alpha)) \\
\downarrow^N \Gamma P & : \text{Ne} \in \Gamma \rightarrow \text{Vnf} P \\
\downarrow^N \Gamma \text{c} & = \text{map} (\downarrow^N \Gamma \text{c}) \\
\downarrow^N \Gamma \text{c} & = \text{c} \\
\downarrow^N \Gamma \text{u} & = () \\
\downarrow^N \Gamma \text{u} & = \text{pair}^- (\downarrow^N \Gamma \text{b}_1) (\downarrow^N \Gamma \text{b}_2)
\end{align*}
\]

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

\[
\begin{align*}
\uparrow^P \Gamma \Rightarrow \text{Ne} u & = \lambda \alpha \tau. \text{a}. \uparrow^N \Gamma (\text{app} (\text{run} \tau \alpha u) (\downarrow^P \Gamma \text{a})) \\
\uparrow^P \Gamma \Rightarrow \text{Ne} f & = (\text{abs} (\downarrow^N \Gamma \text{P} \circ \text{run} \tau \alpha) \\
& \circ \text{map} (\lambda \alpha \tau. f (\text{map} \text{k} \text{r} \alpha)) \text{fresh}^P \Gamma)
\end{align*}
\]

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

This time, the identity environment \( \text{fresh}^\Gamma \Gamma = \text{Cov} \Gamma (\Gamma.P) \) can only be generated in the monad, due to monadic positive reflection.

\[
\begin{align*}
\text{fresh}^\Gamma & = \text{return} () \\
\text{fresh}^\Gamma \Gamma & = (\text{run} \text{wk} \text{fresh}^\Gamma \Gamma) \star \text{fresh}^\Gamma \Gamma
\end{align*}
\]

Putting things together, we obtain the normalization function

\[
\text{norm} (t : \text{Ne} \in \Gamma) = (\text{map} \text{run} \circ \text{map} \text{run}) \text{fresh}^\Gamma
\]

Taking stock, we have arrived at normal forms that eagerly introduce \( \text{Ne} \) and eliminate \( \text{Ne} \) negative types and also eagerly introduce positive types \( \text{Vnf} \). 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 \( \Gamma, \Delta \) are of one of the forms \( \sigma^* \) 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.\(^8\) 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 \( f : (\mathcal{J} \Rightarrow \mathcal{K}) \Gamma \rightarrow [P] \mathcal{J} \) to \( [P] f : [P] \mathcal{J} \Gamma \rightarrow [P] \mathcal{K} \Gamma \). For arbitrary \( \mathcal{J} \) and \( \Gamma \), the family \( [P] \mathcal{J} \Gamma \) is inductively constructed by the following rules:

\[
\begin{align*}
\text{hyp}^+ \mathcal{J} \Gamma (\sigma^*) & \quad \text{hyp}^- \mathcal{J} \Gamma (\tau^\Gamma \Delta) \\
\text{branch}_0 [0] \mathcal{J} \Gamma & \quad \text{branch}_2 [P_1] \mathcal{J} \Gamma [P_2] \mathcal{J} \Gamma (\tau^P \Gamma \Delta) \\
\text{split}_0 [1] \mathcal{J} \Gamma & \quad \text{split}_2 [P_1] \mathcal{J} \Gamma (\tau^P \Gamma \Delta) [P_2] \mathcal{J} \Gamma (\tau^P \Gamma \Delta)
\end{align*}
\]

Note the recursive occurrence of \( [P] \) as argument to \( [P_1] \) in \( \text{split}_2 \), 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 \( \text{split}_2 \) “join” and \( \text{split}_0 \) “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 \( \text{Cxt} \) 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

\( ^8 \) Filinski [15, Section 4] achieves maximal pattern matching through an additional, ordered context \( \Theta \) for positive variables which are eagerly split.
certain future, the "proposition" [P] J states that 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, X). The introduction rules for [P] J resemble the axioms of choice (branch0, branch2) and sequence (split0, split2) in dynamic logic.

Focalization is a technique to remove non-care non-determinism from proof search, and as such, polarized lambda calculus is foremost a calculus of normal forms. These normal forms are formed by four mutually defined inductive families of presheaves VnP, NeN, Cov J, and NfN. As they are very similar to the CBPV normal forms given in the last section, we only report the differences. Values [v : Val P Γ] are unchanged; they can refer to atomic positive types (var+) and thunks.

NeN Γ start with a negative variable instead of with force, as forcing thunks are already performed in hyp when adding hypotheses of Thunk type. The normal forms [NfN Γ] of negative type are unchanged with the exception that pattern matching happens eagerly in abs, by virtue of [P].

var+ N ∈ Γ NeN Γ
abs [P](NfN Γ)
Ne(P ⇒ N)Γ

The Cover monad Cov J Γ lacks constructors split, case and abort since the pattern matching is taken care of by [P].

return Cov J Γ
bind Cov J Γ
Cov J Γ

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].

join : ∀i. Cov i (Cov oo J) ⇒ Cov oo J
join i+1 (return i e) = e
join i+1 (bind i k t) = bind oo (t : Ne P Γ)
(map P i) join i (k : [P](Cov J Γ))

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

return : ∀i. J ⇒ Cov i+1 J
bind : ∀i. Ne(Comp P) ⇒ [P](Cov i J) ⇒ Cov i+1 J

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 ≥ 0 cases, reflection cannot simply produce a single positive semantic value; instead, one such value is needed for every branch. We implement reflect P : ([P] ⇒ J) ⇒ [P] 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 β-expansion via ↑ 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 false 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 P1 + P2 generates two branches, which may result in several uses of the continuation k. In the first branch, we recursively reflect at P1. Its continuation will receive a semantic value in [P1], which we inject via ↓ into [P1 + P2] to pass it to k. The second branch proceeds analogously. Finally reflecting at P1 × P2 means we first have to analyze P1, and in each of the generated branches we continue to analyze P2. Thus reflect P2 is passed as a continuation to reflect P1. Each reflection phase gives us a semantic value a1 of type P1, which we combine to a tuple before passing it to k. Note also that the context extension τ1 created in the first phase needs to be composed with the context extension τ2 of the second phase to transport k into the final context. Further, the value a1 was constructed relative to the target of τ1 and still needs to be transported with τ2 before being paired up with a2.

The method reflect P replaces previous uses of freshP in reflection and reification at negative types.

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

\[
\text{fresh}^\Gamma = \{[P] \Rightarrow [P]\} \\
\text{fresh}^\epsilon = () \\
\text{fresh}^{\Gamma,N} = \text{ren wko}^N \text{fresh}^\Gamma, \text{var}^+ \text{zero} \\
\text{fresh}^{\Gamma,N} = \text{ren wkn}^N \text{fresh}^\Gamma, \text{var}^- \text{zero} \\
\]

The terms \([\text{Tim N}]\Gamma\) 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.

var− N ∈ Γ \text{abs}[P](\text{Tim N})Γ \text{abs}[P](\text{Tim N})Γ \text{abs}[P](\text{Tim N})Γ

bind \text{Tim (Comp P)Γ} \text{bind}[P](\text{Tim N})Γ \text{bind}[P](\text{Tim N})Γ

**Figure 6:** Positive reflection (polarized lambda calculus).
With instantiations we would like to study the NbE algorithm for STLC arising from
This completes the definition of the normalization function
This matching can be defined for a generic evaluation function of
term interpretation \( \lambda (\_): Tm \vdash N \Gamma \rightarrow [\Gamma] \rightarrow [N] \) shall be as for
CBPV except that we need to exchange the interpretation function for binders
\( \lambda (\_): Tm \vdash N (\Gamma, P) \rightarrow [\Gamma] \rightarrow [P \Rightarrow N] \).
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
type \( \nu \) of type \( P \) amounts to a complete matching of \( \nu \) against the
case tree and binding the remaining atomic and negative cubes. This
matching can be defined for a generic evaluation function of
type \( Ev \mathcal{J} \mathcal{A} \mathcal{M} = \sum [\Gamma] \rightarrow [\mathcal{J}] \mathcal{A} \).
match \( t \) (map \( P \) \( \Gamma \) \( \Delta \)) (\( \Delta \) \( \subseteq \) \( \Phi \)) \( (a: [\Phi] \Phi) = \)
mismatch \( a \) (map \( P \) \( \Gamma \) \( \Delta \)) \( \mathcal{R} \) \( \Gamma \).

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,
judgement \( \Gamma \vdash N \) is a normal derivation. At the heart of our development stands the notion of a

\[ match \ a \ (map \ P \ \Gamma) \ (\_ \Gamma) \ (\mathcal{R} \Gamma) \]

This completes the definition of the normalization function \( \text{norm}(t: Tm \vdash N \Gamma) = \downarrow N (t)(\text{fresh} \Gamma) \).

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 EUVERTYPE Types for Programming and Verification.

REFERENCES

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


