


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

Andreas Abel 

Department of Computer Science and Engineering, Chalmers and Gothenburg University, Sweden
www.cse.chalmers.se/~abela
andreas.abel@gu.se

Christian Sattler

Department of Computer Science and Engineering, Chalmers and Gothenburg University, Sweden

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.

2012 ACM Subject Classification Theory of computation → Type theory; Theory of computation → Type structures; Theory of computation → Functional constructs; Theory of computation → Proof theory; Theory of computation → Categorical semantics; Theory of computation → Operational semantics

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

Supplement Material <https://andreasabel.github.io/ipl/html/NfModelMonad.html>

Funding *Andreas Abel:* VR Grant 2014-04864 *Termination Certificates for Dependently-Typed Programs and Proofs via Refinement Types*

Acknowledgements The first author took inspiration from some unpublished notes by Thorsten Altenkirch titled *Another topological completeness proof for intuitionistic logic* received by email on 16th March 2000.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.

1 Introduction

The idea behind *normalization by evaluation* (NbE) is to utilize a standard interpreter, usually evaluating closed terms, to compute the normal form of an open term. 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 [8, 13] by modelling uninterpreted base types as sets of neutral

(aka atomic) terms, or more precisely, as presheaves or sets of neutral term families, in order to facilitate fresh bound variable generation 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 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 not work anymore. **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* \mathcal{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 leaves are occupied by non-monadic values** [4]. In this article we only consider weak sum types, producing non-unique normal forms, where it does not matter in which order the questions are asked (commuting case splits), and whether the same question is asked several times (redundant case splits). The model would need refinement for strong, extensional sums [2, 4, 6, 7, 24].

Filinski [14] studied NbE for Moggi’s computational lambda calculus [21], shedding light on the difference between call-by-name (CBN) and call-by-value (CBV) NbE, where Danvy’s type-directed partial evaluation [12] 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 [17] who designed the call-by-push-value (CBPV) lambda-calculus to embed both the CBN and CBV lambda calculus.

In this work, we formulate NbE for CBPV (Section 3), with the aim to investigate later whether CBN and CBV NbE can be recovered from CBPV NbE via the standard translations of the CBN and CBV calculi into CBPV.

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 [5, 18] 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** [25, 10, 23] in Section 4.

In our presentation of the various lambda calculi we ignore the concrete syntax, only consider the abstract syntax obtained by the Curry-Howard-Isomorphism. A term is simply a derivation tree whose nodes are rule invocations. Thus, a 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 extends to the semantic constructions [11, 3].

Concerning the presentation of polarized lambda calculus, we depart from Zeilberger

[25] 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 [14, Section 4] and Krishnaswami's [16] 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) and 4 (partial), available at <https://github.com/andreasabel/ipl>. Agda was particularly helpful to correctly handle the renamings abundantly present when working with presheaves.

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 $A \in \Gamma$ and sublist relations $\Gamma \subseteq \Delta$ are given inductively by the following rules:

$$\begin{array}{ccccc} \text{zero} & \frac{}{A \in \Gamma.A} & \text{suc} & \frac{A \in \Gamma}{A \in \Gamma.B} & \varepsilon & \frac{}{\varepsilon \subseteq \varepsilon} & \text{lift} & \frac{\Gamma \subseteq \Delta}{\Gamma.A \subseteq \Delta.A} & \text{weak} & \frac{\Gamma \subseteq \Delta}{\Gamma \subseteq \Delta.A} \end{array}$$

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

We can define $\text{id} : \Gamma \subseteq \Gamma$ and $\text{id} ; _ : \Gamma \subseteq \Delta \rightarrow \Delta \subseteq \Phi \rightarrow \Gamma \subseteq \Phi$ 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 $\text{id} ; \tau \equiv \tau$ in propositional equality for all morphisms τ . The singleton weakening $\text{wk}_\Gamma^A : \Gamma \subseteq \Gamma.A$, also written wk^A or wk , is defined by $\text{wk} = \text{weak id}$.

The category Cxt allows us to consider $A \in _$ as a presheaf over Cxt^{op} for any A , witnessed by $\text{reindex} : \Gamma \subseteq \Delta \rightarrow A \in \Gamma \rightarrow A \in \Delta$, which is the morphism part of functor $A \in _$ from Cxt to Set , mapping object Γ to the set $A \in \Gamma$ of the indices of A in Γ . The associated functor laws $\text{reindex id } x \equiv x$ and $\text{reindex } \tau_2 (\text{reindex } \tau_1 x) \equiv \text{reindex } (\tau_1 ; \tau_2) x$ 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. Function (\Rightarrow) and product types (\times and 1) are negative, while

base types (o) and sum types ($+$ and 0) are positive.

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

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.

$$\begin{array}{lll} \text{var} \frac{A \in \Gamma}{A \dashv \Gamma} & \text{abs} \frac{B \dashv \Gamma.A}{A \Rightarrow B \dashv \Gamma} & \text{app} \frac{A \Rightarrow B \dashv \Gamma \quad A \dashv \Gamma}{B \dashv \Gamma} \\ \\ \text{unit} \frac{}{1 \dashv \Gamma} & \text{pair} \frac{A_1 \dashv \Gamma \quad A_2 \dashv \Gamma}{A_1 \times A_2 \dashv \Gamma} & \text{prj}_i \frac{A_1 \times A_2 \dashv \Gamma}{A_i \dashv \Gamma} \\ \\ \text{inj}_i \frac{A_i \dashv \Gamma}{A_1 + A_2 \dashv \Gamma} & \text{case} \frac{A_1 + A_2 \dashv \Gamma \quad B \dashv \Gamma.A_1 \quad B \dashv \Gamma.A_2}{B \dashv \Gamma} & \text{abort} \frac{0 \dashv \Gamma}{B \dashv \Gamma} \end{array}$$

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 $\boxed{v_n = \text{var}(\text{suc}^n \text{zero})}$ for the n th variable, a term such as $\text{abs}(\text{abs}(\text{pair } v_1 (\text{abs}(\text{app } v_1 v_0))))$ 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 operation¹ $\boxed{\text{ren} : \Gamma \subseteq \Delta \rightarrow A \dashv \Gamma \rightarrow A \dashv \Delta}$ defined by recursion over $t : A \dashv \Gamma$, and functor laws for ren analogously to reindex .

Normal forms² are logically characterized as those fulfilling the *subformula* property [22, 15]. **Normal** forms $\boxed{n : \text{Nf } A \Gamma}$ are mutually defined with **neutral** normal forms $\boxed{u : \text{Ne } A \Gamma}$. In the following inductive definition, we reuse the rule names from the term constructors.

$$\begin{array}{lll} \text{var} \frac{A \in \Gamma}{\text{Ne } A \Gamma} & \text{abs} \frac{\text{Nf } B (\Gamma.A)}{\text{Nf } (A \Rightarrow B) \Gamma} & \text{app} \frac{\text{Ne } (A \Rightarrow B) \Gamma \quad \text{Nf } A \Gamma}{\text{Ne } B \Gamma} \quad \text{ne} \frac{\text{Ne } o \Gamma}{\text{Nf } o \Gamma} \\ \\ \text{unit} \frac{}{\text{Nf } 1 \Gamma} & \text{pair} \frac{\text{Nf } A_1 \Gamma \quad \text{Nf } A_2 \Gamma}{\text{Nf } (A_1 \times A_2) \Gamma} & \text{prj}_i \frac{\text{Ne } (A_1 \times A_2) \Gamma}{\text{Ne } A_i \Gamma} \quad \text{P : Positive types} \\ \\ \text{inj}_i \frac{\text{Nf } A_i \Gamma}{\text{Nf } (A_1 + A_2) \Gamma} & \text{case} \frac{\text{Ne } (A_1 + A_2) \Gamma \quad \text{Nf } \underline{P} (\Gamma.A_1) \quad \text{Nf } \underline{P} (\Gamma.A_2)}{\text{Nf } \underline{P} \Gamma} & \text{abort} \frac{\text{Ne } 0 \Gamma}{\text{Nf } P \Gamma} \end{array}$$

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 [3, 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).

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 [20, 2, 24]. Such normal forms do not have a simple inductive definition, and we shall not consider them in this article.

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).

Needless to say, $\text{Ne } A$ and $\text{Nf } A$ 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 $\ulcorner _ \urcorner : \text{Nf } A \Gamma \rightarrow A \dashv \Gamma$ and $\ulcorner _ \urcorner : \text{Ne } A \Gamma \rightarrow A \dashv \Gamma$ that discards constructor ne but keeps all other constructors. This erasure function naturally commutes with reindexing, making it a natural transformation between the presheaves $\text{Nf } A$ ($\text{Ne } A$, resp.) and $A \dashv _$. We shall simply write, for instance, $\text{Nf } A \dot{\rightarrow} A \dashv _$ 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 $\mathcal{A} \dot{\rightarrow} \mathcal{B} \dot{\rightarrow} \mathcal{C}$ for $\forall \Gamma. \mathcal{A} \Gamma \rightarrow (\mathcal{B} \Gamma \rightarrow \mathcal{C} \Gamma)$.

► **Remark.** 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:

$$\begin{array}{ll} \text{abort}^B : \text{Ne } 0 \dot{\rightarrow} \text{Nf } B & \\ \text{abort}^1 u = \text{unit} & \text{abort}^{A \times B} u = \text{pair}(\text{abort}^A u)(\text{abort}^B u) \\ \text{abort}^P u = \text{abort } u & \text{abort}^{A \Rightarrow B} u = \text{abs}(\text{abort}^B(\text{ren wk}^A u)) \end{array}$$

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

2.3 Normalization

Normalization is concerned with finding a normal form $n : \text{Nf } A \Gamma$ for each term $t : A \dashv \Gamma$. The normal form should be *sound*, i. e., $\ulcorner n \urcorner \cong t$ with respect to a equational theory \cong on terms (see Appendix A). 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 implement only the normalization function $\text{norm} : A \dashv \Gamma \rightarrow \text{Nf } A \Gamma$ with proving its soundness and completeness. From a logical perspective, we will compute for each derivation of $A \dashv \Gamma$ a normal derivation $\text{Nf } A \Gamma$.

Normalization by evaluation (NbE) $\text{norm}(t : A \dashv \Gamma) = \downarrow^A(\llbracket t \rrbracket_{\text{fresh}^\Gamma})$ decomposes normalization into *evaluation* $\llbracket _ \rrbracket : (t : A \dashv \Gamma) \rightarrow \llbracket \Gamma \rrbracket \dot{\rightarrow} \llbracket A \rrbracket$ in the identity environment $\text{fresh}^\Gamma : \llbracket \Gamma \rrbracket \Gamma$ followed by *reification* $\downarrow^A : \llbracket A \rrbracket \dot{\rightarrow} \text{Nf } A$ (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^Γ which will map the free indices of t to corresponding *unknowns*. To accommodate unknowns in the semantics, types A are mapped to presheaves $\llbracket A \rrbracket$ (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 $\uparrow^A : \text{Ne } A \dot{\rightarrow} \llbracket A \rrbracket$ 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 $\text{magic} : \mathbf{0} \rightarrow T$ 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 $\pi_i : S_1 \times S_2 \rightarrow S_i$, and their disjoint sum $S_1 + S_2$ with injections $\iota_i : S_i \rightarrow S_1 + S_2$ and elimination $[f_1, f_2] : S_1 + S_2 \rightarrow T$ for arbitrary $f_i : S_i \rightarrow T$.

Presheaves (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} , $(\mathcal{A} \hat{+} \mathcal{B}) \Gamma = \mathcal{A} \Gamma + \mathcal{B} \Gamma$. For the exponential of presheaves, however, we need the **Kripke function space** $(\mathcal{A} \Rightarrow \mathcal{B}) \Gamma = \forall \Delta. \Gamma \subseteq \Delta \rightarrow \mathcal{A} \Delta \rightarrow \mathcal{B} \Delta$.

We will interpret simple types A as corresponding presheaves $\llbracket A \rrbracket$. Let us start with the negative types, defining reflection $\uparrow^A : \mathbf{Ne} A \rightarrow \llbracket A \rrbracket$ and reification $\downarrow^A : \llbracket A \rrbracket \rightarrow \mathbf{Nf} A$ along the way.

$$\begin{array}{ll}
\llbracket 1 \rrbracket &= \hat{1} \\
\uparrow_\Gamma^1 u &= () \\
\downarrow_\Gamma^1 () &= \text{unit} \\
\llbracket A \times B \rrbracket &= \llbracket A \rrbracket \hat{\times} \llbracket B \rrbracket \\
\uparrow_\Gamma^{A \times B} u &= (\uparrow_\Gamma^A(\text{prj}_1 u), \uparrow_\Gamma^B(\text{prj}_2 u)) \\
\downarrow_\Gamma^{A \times B}(a, b) &= \text{pair}(\downarrow_\Gamma^A a, \downarrow_\Gamma^B b) \\
\llbracket A \Rightarrow B \rrbracket &= \llbracket A \rrbracket \Rightarrow \llbracket B \rrbracket \\
\uparrow_\Gamma^{A \Rightarrow B} u (\tau : \Gamma \subseteq \Delta) (a : \llbracket A \rrbracket \Delta) &= \uparrow_\Delta^B(\text{app}(\text{ren } \tau u) (\downarrow_\Delta^A a)) \\
\downarrow_\Gamma^{A \Rightarrow B} f &= \text{abs}(\downarrow_{\Gamma.A}^B(f \text{ wk}_\Gamma^A \text{ fresh}_\Gamma^A))
\end{array}$$

In the reification at function types $\downarrow^{A \Rightarrow B}$, the renaming $\text{wk}_\Gamma^A : \Gamma \subseteq \Gamma.A$ makes room for a new variable of type A , which is reflected into $\llbracket A \rrbracket$ by $\text{fresh}_\Gamma^A = \uparrow_{\Gamma.A}^A v_0 : \llbracket A \rrbracket(\Gamma.A)$. The ability to introduce fresh variables into a context, and to use semantic objects such as $f : \llbracket A \Rightarrow B \rrbracket \Gamma$ in a such 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 $u : \mathbf{Ne} A \Gamma$ is transported into $\mathbf{Ne} A \Gamma$ via reindexing with $\tau : \Gamma \subseteq \Delta$, in order to be applicable to the normal form $\downarrow_\Delta^A a$ reified from the semantic value a .

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

To recapitulate, 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 $f : \mathcal{A} \rightarrow \mathcal{B}$ to the morphism $\text{map}^{\mathcal{C}} f : \mathcal{C} \mathcal{A} \rightarrow \mathcal{C} \mathcal{B}$ satisfying the functor laws for identity and composition. Then, there are natural transformations $\text{return}^{\mathcal{C}} : \mathcal{A} \rightarrow \mathcal{C} \mathcal{A}$ (unit) and $\text{join}^{\mathcal{C}} : \mathcal{C}(\mathcal{C} \mathcal{A}) \rightarrow \mathcal{C} \mathcal{A}$ (multiplication) satisfying the monad laws.

We are looking for a cover monad \mathcal{C} that offers us these services:

$$\begin{array}{ll}
\text{abort}^{\mathcal{C}} &: \mathbf{Ne} 0 \rightarrow \mathcal{C} \mathcal{B} & \text{case on absurd neutral} \\
\text{case}_\Gamma^{\mathcal{C}} &: \mathbf{Ne} (A_1 + A_2) \Gamma \rightarrow \mathcal{C} \mathcal{B}(\Gamma.A_1) \rightarrow \mathcal{C} \mathcal{B}(\Gamma.A_2) \rightarrow \mathcal{C} \mathcal{B} \Gamma & \text{case on neutral} \\
\text{runNf}^{\mathcal{C}} &: \mathcal{C}(\mathbf{Nf} A) \rightarrow \mathbf{Nf} A & \text{run the monad (Nf only)}
\end{array}$$

To make things concrete, we shall immediately construct an instance of such a cover monad: the **free cover monad** Cov defined as an inductive family with constructors $\text{return}^{\text{Cov}}$, $\text{abort}^{\text{Cov}}$, and case^{Cov} . One can visualize an element $c : \text{Cov } \mathcal{A} \Gamma$ as binary case tree whose inner nodes (case) are labeled by a neutral term 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. (The uninspiring join^{Cov} flattens a 2-level case tree, i.e., a case tree with case trees as leaves, into a single one.) Finally $\text{runNf}^{\text{Cov}}$ is a simple recursion on the tree,

replacing case^{Cov} and $\text{abort}^{\text{Cov}}$ by the case and abort constructions on normal forms, and $\text{return}^{\text{Cov}}$ by the identity.

Using the services of a generic cover monad \mathcal{C} , we can complete our semantics:

$$\begin{aligned} \llbracket o \rrbracket &= \mathcal{C}(\text{Ne } o) & \llbracket 0 \rrbracket &= \mathcal{C} \hat{0} \\ \uparrow^o &= \text{return}^{\mathcal{C}} & \uparrow^0 &= \text{abort}^{\mathcal{C}} \\ \downarrow^o &= \text{runNf}^{\mathcal{C}} \circ \text{map}^{\mathcal{C}} \text{ne} & \downarrow^0 &= \text{runNf}^{\mathcal{C}} \circ \text{map}^{\mathcal{C}} \text{magic} \\ \llbracket A + B \rrbracket &= \mathcal{C}(\llbracket A \rrbracket \hat{+} \llbracket B \rrbracket) \\ \uparrow_{\Gamma}^{A+B} u &= \text{case}^{\mathcal{C}} u (\text{return}^{\mathcal{C}} (\iota_1 \text{fresh}_{\Gamma}^A)) (\text{return}^{\mathcal{C}} (\iota_2 \text{fresh}_{\Gamma}^B)) \\ \downarrow_{\Gamma}^{A+B} &= \text{runNf}^{\mathcal{C}} \circ \text{map}^{\mathcal{C}} [\text{inj}_1 \circ \downarrow^A, \text{inj}_2 \circ \downarrow^B] \end{aligned}$$

All semantic types fulfill the weak sheaf condition aka weak pasting, meaning there is a natural transformation $\text{run}^A : \mathcal{C} \llbracket A \rrbracket \rightarrow \llbracket A \rrbracket$ for any simple type A . In other words, we can *run* the monad, pushing its effects into $\llbracket A \rrbracket$. We proceed by induction on A . Positive types P are already monadic, and run^P is simply the join of the monad \mathcal{C} . At negative types we can recurse pointwise at a smaller type, exploiting that values of negative types are essentially (finite or infinite) tuples.

$$\begin{aligned} \text{run}^A &: \mathcal{C} \llbracket A \rrbracket \rightarrow \llbracket A \rrbracket \\ \text{run}^1 c &= () & \text{run}^0 &= \text{join}^{\mathcal{C}} \\ \text{run}^{A \times B} c &= (\text{run}^A (\text{map}^{\mathcal{C}} \pi_1 c), \text{run}^B (\text{map}^{\mathcal{C}} \pi_2 c)) & \text{run}^{A+B} &= \text{join}^{\mathcal{C}} \\ \text{run}^{A \Rightarrow B} c \tau a &= \text{run}^B (\widehat{\text{map}}^{\mathcal{C}} (\lambda \tau' f. f \text{id} (\text{ren } \tau' a)) (\text{ren } \tau c)) & \text{run}^o &= \text{join}^{\mathcal{C}} \end{aligned}$$

For the case of function types $A \Rightarrow B$, we require the monad \mathcal{C} to be *strong*, which amounts to having $\widehat{\text{map}}_{\Gamma}^{\mathcal{C}} \ell : \mathcal{C} A \Gamma \rightarrow \mathcal{C} B \Gamma$ already for a “local” presheaf morphism $\ell : (\mathcal{A} \Rightarrow \mathcal{B}) \Gamma$. The typings are $c : \mathcal{C} \llbracket A \Rightarrow B \rrbracket \Gamma$ and $\tau : \Gamma \subseteq \Delta$ and $a : \llbracket A \rrbracket \Delta$, and now we want to apply every function $f : \llbracket A \Rightarrow B \rrbracket$ in the cover c to argument a . Clearly, $\text{map}^{\mathcal{C}}$ 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 $a : \llbracket A \rrbracket \Delta$ can only work in context Δ or any extension $\tau' : \Delta \subseteq \Phi$, since we can transport a to such a Φ via $a' := \text{ren } \tau' a : \llbracket A \rrbracket \Phi$ but not to a context unrelated to Δ . We obtain our input to run^B of type $\mathcal{C} \llbracket B \rrbracket \Gamma$ as an instance of $\widehat{\text{map}}_{\Gamma}^{\mathcal{C}}$ applied to the local presheaf morphism $(\lambda \tau' f. f \text{id } a') : \Delta \subseteq \Phi \rightarrow \llbracket A \Rightarrow B \rrbracket \Phi \rightarrow \llbracket B \rrbracket \Phi$ and the transported cover $\text{ren } \tau c : \mathcal{C} \llbracket A \Rightarrow B \rrbracket \Delta$.

We extend the type interpretation pointwise to contexts, i.e., $\llbracket \varepsilon \rrbracket = \hat{1}$ and $\llbracket \Gamma.A \rrbracket = \llbracket \Gamma \rrbracket \hat{\times} \llbracket A \rrbracket$ and obtain a natural projection function $\text{lookup}(x : A \in \Gamma) : \llbracket \Gamma \rrbracket \rightarrow \llbracket A \rrbracket$ from the semantic environments. The evaluation function $\llbracket t : A \vdash \Gamma \rrbracket : \llbracket \Gamma \rrbracket \rightarrow \llbracket A \rrbracket$ can now be defined by recursion on t . Herein, the environment γ lives in $\llbracket \Gamma \rrbracket \Delta$, thus, $\llbracket t \rrbracket_{\gamma} : \llbracket A \rrbracket \Delta$.

$$\begin{aligned} \llbracket \text{unit} \rrbracket_{\gamma} &= () & \llbracket \text{var } x \rrbracket_{\gamma} &= \text{lookup } x \gamma \\ \llbracket \text{pair } t_1 t_2 \rrbracket_{\gamma} &= (\llbracket t_1 \rrbracket_{\gamma}, \llbracket t_2 \rrbracket_{\gamma}) & \llbracket \text{prj}_i t \rrbracket_{\gamma} &= \pi_i \llbracket t \rrbracket_{\gamma} \\ \llbracket \text{abst } t \rrbracket_{\gamma} &= \lambda \llbracket t \rrbracket_{\gamma} & \llbracket \text{app } t u \rrbracket_{\gamma} &= \llbracket t \rrbracket_{\gamma} \text{id } \llbracket u \rrbracket_{\gamma} \\ \llbracket \text{inj}_i t \rrbracket_{\gamma} &= \iota_i \llbracket t \rrbracket_{\gamma} & \llbracket \text{case } u t_1 t_2 \rrbracket_{\gamma} &= \llbracket \text{case} \rrbracket \llbracket u \rrbracket_{\gamma} \lambda \llbracket t_1 \rrbracket_{\gamma} \lambda \llbracket t_2 \rrbracket_{\gamma} \\ & & \llbracket \text{abort } u \rrbracket_{\gamma} &= \llbracket \text{abort} \rrbracket \llbracket u \rrbracket_{\gamma} \end{aligned}$$

For the interpretation of the binders **abs** and **case** we use the mutually defined $\lambda(_)$.

$$\begin{aligned} \lambda \llbracket t : B \vdash \Gamma.A \rrbracket &: \llbracket \Gamma \rrbracket \rightarrow \llbracket A \Rightarrow B \rrbracket \\ &= \lambda (\gamma : \llbracket \Gamma \rrbracket \Delta) (\tau : \Delta \subseteq \Phi) (a : \llbracket A \rrbracket \Phi). \llbracket t \rrbracket_{(\text{ren } \tau \gamma, a)} \end{aligned}$$

The coproduct eliminations $\llbracket \text{abort} \rrbracket$ and $\llbracket \text{case} \rrbracket$ targeting an arbitrary semantic type $\llbracket B \rrbracket$ 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} .

$$\begin{aligned} \langle \text{abort} \rangle^B & : \llbracket 0 \rrbracket \dot{\rightarrow} \llbracket B \rrbracket \\ \langle \text{abort} \rangle^B & = \text{run}^B \circ \text{map}^{\mathcal{C}} \text{ magic} \\ \langle \text{case} \rangle^B & : \llbracket A_1 + A_2 \rrbracket \dot{\rightarrow} \llbracket A_1 \Rightarrow B \rrbracket \dot{\rightarrow} \llbracket A_2 \Rightarrow B \rrbracket \dot{\rightarrow} \llbracket B \rrbracket \\ \langle \text{case} \rangle^B c f_1 f_2 & = \text{run}^B(\widehat{\text{map}}^{\mathcal{C}} (\lambda \tau. [f_1 \tau, f_2 \tau]) c) \end{aligned}$$

To complete the normalization function $\text{norm}(t : A \dashv \Gamma) = \downarrow_{\Gamma}^A(t)_{\text{fresh}^{\Gamma}}$ we define the identity environment $\text{fresh}^{\Gamma} : \llbracket \Gamma \rrbracket \Gamma$, which maps each free index to its corresponding unknown in the semantics, by recursion on Γ :

$$\begin{aligned} \text{fresh}^{\varepsilon} & = () \\ \text{fresh}^{\Gamma.A} & = (\text{ren wk}^A \text{ fresh}^{\Gamma}, \text{fresh}_{\Gamma}^A) \end{aligned}$$

2.4 Continuation monad

As already observed by Filinski [13, Section 5.4] [14, Section 3.2], normalization by evaluation can be carried out in the continuation monad. In our setting, we use a continuation monad CC on presheaves defined as

$$\text{CC } \mathcal{J} = \forall A. (\mathcal{J} \Rightarrow \text{Nf } A) \Rightarrow \text{Nf } A.$$

The answer type of this continuation monad is always Nf , however, we are polymorphic in the simple type A of normal forms we produce.

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

$$\begin{aligned} \text{abort}^{\text{CC}}(u : \text{Ne } 0 \Gamma) & : \text{CC } \mathcal{J} \Gamma \\ \text{abort}^{\text{CC}} u (\tau : \Gamma \subseteq \Delta) (k : (\mathcal{J} \Rightarrow \text{Nf } B) \Delta) & = \text{abort}(\text{ren } \tau u) \\ \text{case}^{\text{CC}}(u : \text{Ne } (A_1 + A_2) \Gamma) (c_1 : \text{CC } \mathcal{J} (\Gamma.A_1)) (c_2 : \text{CC } \mathcal{J} (\Gamma.A_2)) & : \text{CC } \mathcal{J} \Gamma \\ \text{case}^{\text{CC}} u c_1 c_2 (\tau : \Gamma \subseteq \Delta) (k : (\mathcal{J} \Rightarrow \text{Nf } B) \Delta) & = \text{case}(\text{ren } \tau u) n_1 n_2 \text{ where} \\ n_i & : \text{Nf } B (\Delta.A_i) \\ n_i & = c_i (\text{lift}^{A_i} \tau : \Gamma.A_i \subseteq \Delta.A_i) \left(\lambda (\tau' : \Delta.A_i \subseteq \Phi) (j : \mathcal{J} \Phi). k (\text{wk}^{A_i} ; \tau') j \right) \end{aligned}$$

The NbE algorithm using CC is comparable to Danvy's type-directed partial evaluation [12, Figure 8]. However, he uses shift-reset style continuations which can be programmed in the continuation monad, and relies on Scheme's **gensym** to produce fresh variables names rather than using Kripke function space / presheaves.

3 Normalization to Call-By-Push Value

The placement of the monad \mathcal{C} in the type semantics of the previous section is a bit wasteful: Each positive type is prefixed by \mathcal{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 $\llbracket A \rrbracket$ corresponds to the translation of call-by-name (CBN) lambda-calculus into Moggi's monadic meta-language [21, 17].

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) [17], 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 .

$$\begin{array}{ll} \mathsf{Ty}^+ \ni P, Q & ::= o^+ \mid 1 \mid P_1 \times P_2 \mid 0 \mid P_1 + P_2 \mid \mathsf{Thunk} N & \text{positive type} \\ \mathsf{Ty}^- \ni N, M & ::= o^- \mid \top \mid N_1 \& N_2 \mid P \Rightarrow N \mid \mathsf{Comp} P & \text{negative type} \end{array}$$

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 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 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 $\llbracket P \rrbracket$ and $\llbracket N \rrbracket$ of positive and negative types as presheaves.

$$\begin{array}{ll} \llbracket 1 \rrbracket & = \hat{1} & \llbracket \top \rrbracket & = \hat{1} \\ \llbracket P_1 \times P_2 \rrbracket & = \llbracket P_1 \rrbracket \hat{\times} \llbracket P_2 \rrbracket & \llbracket N_1 \& N_2 \rrbracket & = \llbracket N_1 \rrbracket \hat{\times} \llbracket N_2 \rrbracket \\ \llbracket 0 \rrbracket & = \hat{0} & & \\ \llbracket P_1 + P_2 \rrbracket & = \llbracket P_1 \rrbracket \hat{+} \llbracket P_2 \rrbracket & \llbracket P \Rightarrow N \rrbracket & = \llbracket P \rrbracket \hat{\Rightarrow} \llbracket N \rrbracket \\ \llbracket \mathsf{Thunk} N \rrbracket & = \llbracket N \rrbracket & \llbracket \mathsf{Comp} P \rrbracket & = \mathcal{C} \llbracket P \rrbracket \\ \llbracket o^+ \rrbracket & = o^+ \in _ & \llbracket o^- \rrbracket & = \mathcal{C}(\mathsf{Ne} o^-) \end{array}$$

Semantically, we do not distinguish 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 cover monad. Positive atoms, standing for value types without constructors, are only inhabited by variables $x : o^+ \in \Gamma$. Negative atoms stand for computation types without own eliminations, thus, their inhabitants stem only from eliminations of more complex types, made from positive eliminations captured in \mathcal{C} and negative eliminations chained together as neutral $\mathsf{Ne} o^-$, which we shall define below. The method $\mathsf{runNf}^{\mathcal{C}}$ of cover monad \mathcal{C} can be extended to $\mathsf{run}^N : \mathcal{C} \llbracket N \rrbracket \rightarrow \llbracket N \rrbracket$ for negative types N , by recursion on N . Informally speaking, this makes all negative types *monadic*.

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

3.2 Terms and evaluation

Assuming a family $\text{Tm } N \Gamma$ of terms of negative type N in context Γ , values $\boxed{v : \text{Val } P \Gamma}$ of positive type P shall be constructed by the following rules:

$$\begin{array}{c} \text{var} \frac{P \in \Gamma}{\text{Val } P \Gamma} \quad \text{thunk} \frac{\text{Tm } N \Gamma}{\text{Val } (\text{Thunk } N) \Gamma} \\ \\ \text{unit}^+ \frac{}{\text{Val } 1 \Gamma} \quad \text{pair}^+ \frac{\text{Val } P_1 \Gamma \quad \text{Val } P_2 \Gamma}{\text{Val } (P_1 \times P_2) \Gamma} \quad \text{inj}_i \frac{\text{Val } P_i \Gamma}{\text{Val } (P_1 + P_2) \Gamma} \end{array}$$

The terms of pure CBPV are given by the inductive family $\boxed{\text{Tm } N \Gamma}$. It repeats the introductions and eliminations of negative types, except that application is restricted to values. Values of type $\text{Thunk } N$ are embedded via **force**. Further, values of type P can be embedded via **ret**, producing a term of type $\text{Comp } P$. 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 . Finally, there are eliminators (**split**, **case**, **abort**) for values of positive product and sum types.

$$\begin{array}{c} \text{ret} \frac{\text{Val } P \Gamma}{\text{Tm } (\text{Comp } P) \Gamma} \quad \text{abs} \frac{\text{Tm } N (\Gamma.P)}{\text{Tm } (P \Rightarrow N) \Gamma} \quad \text{pair}^- \frac{\text{Tm } N_1 \Gamma \quad \text{Tm } N_2 \Gamma}{\text{Tm } (N_1 \& N_2) \Gamma} \quad \text{unit}^- \frac{}{\text{Tm } \top \Gamma} \\ \\ \text{force} \frac{\text{Val } (\text{Thunk } N) \Gamma}{\text{Tm } N \Gamma} \quad \text{app} \frac{\text{Tm } (P \Rightarrow N) \Gamma \quad \text{Val } P \Gamma}{\text{Tm } N \Gamma} \quad \text{prj}_i \frac{\text{Tm } (N_1 \& N_2) \Gamma}{\text{Tm } N_i \Gamma} \\ \\ \text{bind} \frac{\text{Tm } (\text{Comp } P) \Gamma \quad \text{Tm } N (\Gamma.P)}{\text{Tm } N \Gamma} \quad \text{split} \frac{\text{Val } (P_1 \times P_2) \Gamma \quad \text{Tm } N (\Gamma.P_1.P_2)}{\text{Tm } N \Gamma} \\ \\ \text{case} \frac{\text{Val } (P_1 + P_2) \Gamma \quad \text{Tm } N (\Gamma.P_1) \quad \text{Tm } N (\Gamma.P_2)}{\text{Tm } N \Gamma} \quad \text{abort} \frac{\text{Val } 0 \Gamma}{\text{Tm } N \Gamma} \end{array}$$

Interpretation of values $\llbracket v : \text{Val } P \Gamma \rrbracket : \llbracket \Gamma \rrbracket \rightarrow \llbracket P \rrbracket$ and terms $\llbracket t : \text{Tm } N \Gamma \rrbracket : \llbracket \Gamma \rrbracket \rightarrow \llbracket N \rrbracket$ is straightforward, thanks to the pioneering work of Moggi [21] and Levy [17] put into the design of CBPV.

$$\begin{array}{llll} \llbracket \text{var } x \rrbracket_\gamma & = & \text{lookup } x \gamma & \llbracket \text{abs } t \rrbracket_\gamma & = & \lambda \llbracket t \rrbracket_\gamma \\ \llbracket \text{unit}^+ \rrbracket_\gamma & = & () & \llbracket \text{app } t v \rrbracket_\gamma & = & \llbracket t \rrbracket_\gamma \text{ id } \llbracket v \rrbracket_\gamma \\ \llbracket \text{pair}^+ v_1 v_2 \rrbracket_\gamma & = & (\llbracket v_1 \rrbracket_\gamma, \llbracket v_2 \rrbracket_\gamma) & \llbracket \text{unit}^- \rrbracket_\gamma & = & () \\ \llbracket \text{inj}_i v \rrbracket_\gamma & = & \iota_i \llbracket v \rrbracket_\gamma & \llbracket \text{pair}^- t_1 t_2 \rrbracket_\gamma & = & (\llbracket t_1 \rrbracket_\gamma, \llbracket t_2 \rrbracket_\gamma) \\ \llbracket \text{thunk } t \rrbracket_\gamma & = & \llbracket t \rrbracket_\gamma & \llbracket \text{prj}_i t \rrbracket_\gamma & = & \pi_i \llbracket t \rrbracket_\gamma \\ & & & \llbracket \text{force } v \rrbracket_\gamma & = & \llbracket v \rrbracket_\gamma \end{array}$$

Since *Thunk* serves only as an embedding of negative into positive types and has no semantic effect, we interpret thunking and forcing by the identity. The eliminations for positive types deal now only with values, thus, need not reference the monad operations.

$$\begin{array}{ll} \llbracket \text{split } v t \rrbracket_\gamma & = \quad (\lambda (a_1, a_2). \llbracket t \rrbracket_{(\gamma, a_1, a_2)}) \llbracket v \rrbracket_\gamma \\ \llbracket \text{case } v t_1 t_2 \rrbracket_\gamma & = \quad [\lambda a_1. \llbracket t_1 \rrbracket_{(\gamma, a_1)}, \lambda a_2. \llbracket t_2 \rrbracket_{(\gamma, a_2)}] \llbracket v \rrbracket_\gamma \\ \llbracket \text{abort } v \rrbracket_\gamma & = \quad \text{magic } \llbracket v \rrbracket_\gamma \end{array}$$

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

$$\begin{aligned} \llbracket \text{ret } v \rrbracket_\gamma &= \text{return}^C \llbracket v \rrbracket_\gamma \\ \llbracket \text{bind } u \ t \rrbracket_\gamma &= \text{run}^C (\widehat{\text{map}}^C \lambda \llbracket t \rrbracket_\gamma \llbracket u \rrbracket_\gamma) \end{aligned}$$

3.3 Normal forms and normalization

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

$$\begin{aligned} \text{var } \frac{o^+ \in \Gamma}{\text{Vnf } o^+ \Gamma} \quad \text{thunk } \frac{\text{Nf } N \Gamma}{\text{Vnf } (\text{Thunk } N) \Gamma} \\ \text{unit}^+ \frac{}{\text{Vnf } 1 \Gamma} \quad \text{pair}^+ \frac{\text{Vnf } P_1 \Gamma \quad \text{Vnf } P_2 \Gamma}{\text{Vnf } (P_1 \times P_2) \Gamma} \quad \text{inj}_i \frac{\text{Vnf } P_i \Gamma}{\text{Vnf } (P_1 + P_2) \Gamma} \end{aligned}$$

Neutral normal forms $\boxed{\text{Ne } N \Gamma}$ 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 $\text{Thunk } N \in \Gamma$.

$$\text{force } \frac{\text{Thunk } N \in \Gamma}{\text{Ne } N \Gamma} \quad \text{prj}_i \frac{\text{Ne } (N_1 \ \& \ N_2) \Gamma}{\text{Ne } N_i \Gamma} \quad \text{app } \frac{\text{Ne } (P \Rightarrow N) \Gamma \quad \text{Vnf } P \Gamma}{\text{Ne } N \Gamma}$$

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 $\boxed{\text{Cov}}$.

$$\begin{aligned} \text{return } \frac{\mathcal{J} \Gamma}{\text{Cov } \mathcal{J} \Gamma} \quad \text{bind } \frac{\text{Ne } (\text{Comp } P) \Gamma \quad \text{Cov } \mathcal{J} (\Gamma.P)}{\text{Cov } \mathcal{J} \Gamma} \\ \text{split } \frac{P_1 \times P_2 \in \Gamma \quad \text{Cov } \mathcal{J} (\Gamma.P_1.P_2)}{\text{Cov } \mathcal{J} \Gamma} \\ \text{case } \frac{P_1 + P_2 \in \Gamma \quad \text{Cov } \mathcal{J} (\Gamma.P_1) \quad \text{Cov } \mathcal{J} (\Gamma.P_2)}{\text{Cov } \mathcal{J} \Gamma} \quad \text{abort } \frac{0 \in \Gamma}{\text{Cov } \mathcal{J} \Gamma} \end{aligned}$$

Finally, normal forms of negative types are defined as inductive family $\boxed{\text{Nf } N \Gamma}$. 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 `Cov` monad until an answer can be given in form of a base neutral ($\text{Ne } o^-$) or a normal value.

$$\begin{aligned} \text{ne } \frac{\text{Cov } (\text{Ne } o^-) \Gamma}{\text{Nf } o^- \Gamma} \quad \text{ret } \frac{\text{Cov } (\text{Vnf } P) \Gamma}{\text{Nf } (\text{Comp } P) \Gamma} \\ \text{unit}^- \frac{}{\text{Nf } \top \Gamma} \quad \text{pair}^- \frac{\text{Nf } N_1 \Gamma \quad \text{Nf } N_2 \Gamma}{\text{Nf } (N_1 \ \& \ N_2) \Gamma} \quad \text{abs } \frac{\text{Nf } N \Gamma.P}{\text{Nf } (P \Rightarrow N) \Gamma} \end{aligned}$$

We again can run the cover monad on normal forms, i. e., have $\text{runNf} : \text{Cov } (\text{Nf } N) \rightarrow \text{Nf } N$, which extends to negative semantic values $\text{run} : \text{Cov } \llbracket N \rrbracket \rightarrow \llbracket N \rrbracket$.

Reification $\downarrow^P : \llbracket P \rrbracket \dot{\rightarrow} \text{Vnf } P$ at positive types P produces a normal value, and $\downarrow^N : \llbracket N \rrbracket \dot{\rightarrow} \text{Nf } N$ 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 $\llbracket P \rrbracket$, breaking down P to positive atoms o^+ and negative remainders $\text{Thunk } N$. However, in $\llbracket P \rrbracket$ we do not have case analysis available, thus, positive reflection $\uparrow_\Gamma^P : P \in \Gamma \rightarrow \text{Cov } \llbracket P \rrbracket \Gamma$ needs to run in the monad. Luckily, as \downarrow^N produces a normal form, monadic intermediate computations are permitted under a final runNf . Negative reflection $\uparrow^N : \text{Ne } N \dot{\rightarrow} \llbracket N \rrbracket$ is as before generalized from variables to neutrals, to handle the breaking down of N via eliminations. In the following definition of reflection we use the abbreviation $\boxed{\text{fresh}_\Gamma^P = \uparrow_{\Gamma.P}^P \mathbf{v}_0 : \text{Cov } \llbracket P \rrbracket (\Gamma.P)}$.

\uparrow_Γ^P	$: P \in \Gamma \rightarrow \text{Cov } \llbracket P \rrbracket \Gamma$	\downarrow^P	$: \llbracket P \rrbracket \dot{\rightarrow} \text{Vnf } P$
$\uparrow_\Gamma^{o^+}$	$x = x$	\downarrow^{o^+}	$= \text{var}$
\uparrow_Γ^1	$x = \text{return } ()$	$\downarrow_\Gamma^1()$	$= \text{unit}^+$
$\uparrow_\Gamma^{P_1 \times P_2}$	$x = \text{split } x \left((\uparrow_{\Gamma.P_1.P_2}^{P_1} \mathbf{v}_1) \star (\uparrow_{\Gamma.P_1.P_2}^{P_2} \mathbf{v}_0) \right)$	$\downarrow_\Gamma^{P_1 \times P_2}(a_1, a_2)$	$= \text{pair}^+ (\downarrow_\Gamma^{P_1} a_1) (\downarrow_\Gamma^{P_2} a_2)$
\uparrow_Γ^0	$x = \text{abort } x$	\downarrow^0	$= \text{magic}$
$\uparrow_\Gamma^{P_1 + P_2}$	$x = \text{case } x \left(\text{map } \iota_1 \text{ fresh}_\Gamma^{P_1} \right) \left(\text{map } \iota_2 \text{ fresh}_\Gamma^{P_2} \right)$	$\downarrow^{P_1 + P_2}$	$= [\text{inj}_1 \circ \downarrow^{P_1}, \text{inj}_2 \circ \downarrow^{P_2}]$
$\uparrow_\Gamma^{\text{Thunk } N}$	$x = \text{return } (\uparrow_\Gamma^N (\text{force } x))$	$\downarrow^{\text{Thunk } N}$	$= \text{thunk} \circ \downarrow^N$

Reflection at positive pairs uses monoidal functoriality $\mathcal{C} \mathcal{A}_1 \dot{\rightarrow} \mathcal{C} \mathcal{A}_2 \dot{\rightarrow} \mathcal{C} (\mathcal{A}_1 \hat{\times} \mathcal{A}_2)$ called \star by McBride and Paterson [19, Section 7] which for monads \mathcal{C} can be defined by $c_1 \star c_2 = \text{join } (\text{map } (\lambda a_1. \text{map } (\lambda a_2. (a_1, a_2)) c_2) c_1)$.

For negative types, reflection and reification works as before:

\uparrow^N	$: \text{Ne } N \dot{\rightarrow} \llbracket N \rrbracket$	\downarrow^N	$: \llbracket N \rrbracket \dot{\rightarrow} \text{Nf } N$
$\uparrow_\Gamma^{\text{Comp } P} u$	$= \text{bind } u \text{ fresh}_\Gamma^P$	$\downarrow_\Gamma^{\text{Comp } P} c$	$= \text{map } (\downarrow^P) c$
$\uparrow_\Gamma^{o^-} u$	$= \text{return } u$	$\downarrow_\Gamma^{o^-} c$	$= \text{ne } c$
$\uparrow_\Gamma^\top u$	$= ()$	$\downarrow_\Gamma^\top ()$	$= \text{unit}^-$
$\uparrow_\Gamma^{N_1 \& N_2} u$	$= \left(\uparrow_\Gamma^{N_1} (\text{prj}_1 u), \uparrow_\Gamma^{N_2} (\text{prj}_2 u) \right)$	$\downarrow_\Gamma^{N_1 \& N_2} (b_1, b_2)$	$= \text{pair}^- (\downarrow_\Gamma^{N_1} b_1) (\downarrow_\Gamma^{N_2} b_2)$

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

$$\begin{aligned} \uparrow_\Gamma^{P \Rightarrow N} u &= \lambda (\tau : \Gamma \subseteq \Delta) (a : \llbracket P \rrbracket \Delta). \uparrow_\Delta^N (\text{app } (\text{ren } \tau u) (\downarrow_\Delta^P a)) \\ \downarrow_\Gamma^{P \Rightarrow N} f &= \text{abs } \left(\text{runNf } \left(\widehat{\text{map}} \left(\lambda (\tau : (\Gamma.P) \subseteq \Delta) (a : \llbracket P \rrbracket \Delta). f (\text{wk}^P \circ \tau) a \right) \text{fresh}_\Gamma^P \right) \right) \end{aligned}$$

The identity environment $\text{fresh}^\Gamma : \text{Cov } \llbracket \Gamma \rrbracket \Gamma$ can only be generated in the monad, due to monadic positive reflection.

$$\begin{aligned} \text{fresh}^\varepsilon &= \text{return } () \\ \text{fresh}^{\Gamma.P} &= (\text{ren } \text{wk}^P \text{ fresh}^\Gamma) \star \text{fresh}_\Gamma^P \end{aligned}$$

Putting things together, we obtain the normalization function

$$\text{norm } (t : \text{Tm } N \Gamma) = \text{runNf } \left(\text{map } (\downarrow^N \circ \langle t \rangle) \text{fresh}^\Gamma \right).$$

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, chaining also the positive eliminations, will be discussed in the following section.

4 Focused Intuitionistic Propositional Logic

Polarized lambda-calculus [25, 23] 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 $H ::= o^+ \mid N$. Contexts Γ, Δ are lists of H s.

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.³ We express maximal pattern matching on P as a strong functor $\langle\langle P \rangle\rangle$ in the category of presheaves, mapping a presheaf \mathcal{J} (“judgement”) to $\langle\langle P \rangle\rangle \mathcal{J}$ and a presheaf morphism $f : (\mathcal{J} \rightrightarrows \mathcal{K}) \Gamma$ to $\widehat{\text{map}}_{\Gamma}^{\langle\langle P \rangle\rangle} f : \langle\langle P \rangle\rangle \mathcal{J} \Gamma \rightarrow \langle\langle P \rangle\rangle \mathcal{K} \Gamma$. For arbitrary \mathcal{J} and Γ , the family $\boxed{\langle\langle P \rangle\rangle \mathcal{J} \Gamma}$ is inductively constructed by the following rules:

$$\begin{array}{lll} \text{hyp}^+ \frac{\mathcal{J}(\Gamma.o^+)}{\langle\langle o^+ \rangle\rangle \mathcal{J} \Gamma} & \text{branch}_0 \frac{}{\langle\langle 0 \rangle\rangle \mathcal{J} \Gamma} & \text{branch}_2 \frac{\langle\langle P_1 \rangle\rangle \mathcal{J} \Gamma \quad \langle\langle P_2 \rangle\rangle \mathcal{J} \Gamma}{\langle\langle P_1 + P_2 \rangle\rangle \mathcal{J} \Gamma} \\ \text{hyp}^- \frac{\mathcal{J}(\Gamma.N)}{\langle\langle \text{Thunk } N \rangle\rangle \mathcal{J} \Gamma} & \text{split}_0 \frac{\mathcal{J} \Gamma}{\langle\langle 1 \rangle\rangle \mathcal{J} \Gamma} & \text{split}_2 \frac{\langle\langle P_1 \rangle\rangle (\langle\langle P_2 \rangle\rangle \mathcal{J}) \Gamma}{\langle\langle P_1 \times P_2 \rangle\rangle \mathcal{J} \Gamma} \end{array}$$

Note the recursive occurrence of $\langle\langle P_2 \rangle\rangle$ as argument to $\langle\langle P_1 \rangle\rangle$ in split_2 , which makes $\langle\langle P \rangle\rangle$ a nested datatype [9]. Agda supports such nested inductive types; but note that $\langle\langle P \rangle\rangle$ 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 $\langle\langle P \rangle\rangle$ is a graded monad on the monoid $(1, \times)$ of product types; however, this coincidence shall not matter for our further considerations.

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 $\text{Vnf } P$, $\text{Ne } N$, $\text{Cov } \mathcal{J}$, and $\text{Nf } N$. As they are very similar to the CBPV normal forms given in the last section, we only report the differences. Values $\boxed{v : \text{Val } P \Gamma}$ are unchanged, they can refer to atomic positive hypotheses (var^+) and normal thunks. Neutrals $\boxed{\text{Ne } N \Gamma}$ 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 $\boxed{\text{Nf } N \Gamma}$ of negative type are unchanged with the exception that pattern matching happens eagerly in abs , by virtue of $\langle\langle P \rangle\rangle$.

$$\text{var}^- \frac{N \in \Gamma}{\text{Ne } N \Gamma} \quad \text{abs} \frac{\langle\langle P \rangle\rangle (\text{Nf } N) \Gamma}{\text{Nf } (P \Rightarrow N) \Gamma}$$

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

$$\text{return} \frac{\mathcal{J} \Gamma}{\text{Cov } \mathcal{J} \Gamma} \quad \text{bind} \frac{\text{Ne } (\text{Comp } P) \Gamma \quad \langle\langle P \rangle\rangle (\text{Cov } \mathcal{J}) \Gamma}{\text{Cov } \mathcal{J} \Gamma}$$

³ Filinski [14, Section 4] achieves maximal pattern matching through an additional, ordered context Θ for positive variables which are eagerly split.

All these inductive families are presheaves, due to factored presentation using $\langle\langle P \rangle\rangle$ 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].

$$\begin{aligned} \text{join} & : \quad \forall i. \text{Cov}^i (\text{Cov}^\infty \mathcal{J}) \rightarrow \text{Cov}^\infty \mathcal{J} \\ \text{join}^{i+1} (\text{return}^i c) & = c \\ \text{join}^{i+1} (\text{bind}^i t k) & = \text{bind}^\infty (t : \text{Ne } P \Gamma) \left(\widehat{\text{map}}_\Gamma^{\langle\langle P \rangle\rangle} \text{join}^i (k : \langle\langle P \rangle\rangle (\text{Cov}^i \mathcal{J}) \Gamma) \right) \end{aligned}$$

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

$$\begin{aligned} \text{return} & : \quad \forall i. \mathcal{J} \rightarrow \text{Cov}^{i+1} \mathcal{J} \\ \text{bind} & : \quad \forall i. \text{Ne } (\text{Comp } P) \rightarrow \langle\langle P \rangle\rangle (\text{Cov}^i \mathcal{J}) \rightarrow \text{Cov}^{i+1} \mathcal{J} \end{aligned}$$

Due to the eager splitting of positive hypotheses, reflection at type P now lives in the graded monad $\langle\langle P \rangle\rangle$ 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 $\text{reflect}^P : (\llbracket P \rrbracket \Rightarrow \mathcal{J}) \rightarrow \langle\langle P \rangle\rangle \mathcal{J}$ 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.

$$\begin{aligned} \text{reflect}^{o^+} k & = \text{hyp}^+ \left(k \text{ wk}^{o^+} (\text{var}^+ \text{zero}) \right) \\ \text{reflect}^{\text{Thunk } N} k & = \text{hyp}^- \left(k \text{ wk}^N (\uparrow^N (\text{var}^- \text{zero})) \right) \\ \text{reflect}^0 k & = \text{branch}_0 \\ \text{reflect}^{P_1 + P_2} k & = \text{branch}_2 \left(\text{reflect}^{P_1} (\lambda \tau. k \tau \circ \iota_1) \right) \left(\text{reflect}^{P_2} (\lambda \tau. k \tau \circ \iota_2) \right) \\ \text{reflect}^1 k & = \text{split}_0 (k \text{ id } ()) \\ \text{reflect}^{P_1 \times P_2} k & = \text{split}_2 \left(\text{reflect}^{P_1} \left(\lambda \tau_1 a_1. \text{reflect}^{P_2} (\lambda \tau_2 a_2. k (\tau_1 \circ \tau_2) (\text{ren } \tau_2 a_1, a_2)) \right) \right) \end{aligned}$$

Reflecting at a positive atomic type o^+ is the regular ending of a reflection pass: we call continuation k with a fresh variable $\text{var}^+ \text{zero}$ of type o^+ , making space for the variable using wk^{o^+} . In case we end at type $\text{Thunk } N$, we add a new variable $\text{var}^- \text{zero}$ 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 falsum quod libet*. 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 $\llbracket P_1 \rrbracket$ which we inject via ι_1 into $\llbracket P_1 + P_2 \rrbracket$ 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 reflect^{P_2} is passed as a continuation to 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 paired up with a_2 .

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

$$\begin{aligned} \downarrow_\Gamma^{P \Rightarrow N} f & = \text{abs} \left(\text{reflect}_\Gamma^P (\lambda (\tau : \Gamma \subseteq \Delta) a. \downarrow_\Delta^P (f \tau a)) \right) \\ \uparrow_\Gamma^{\text{Comp } P} u & = \text{bind } u \left(\text{reflect}_\Gamma^P (\lambda \tau a. \text{return } a) \right) \end{aligned}$$

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

$$\begin{aligned}\text{fresh}^\varepsilon &= () \\ \text{fresh}^{\Gamma.o^+} &= (\text{ren wk}^{o^+} \text{fresh}^\Gamma, \text{var}^+ \text{zero}) \\ \text{fresh}^{\Gamma.N} &= (\text{ren wk}^N \text{fresh}^\Gamma, \uparrow_\Gamma^N(\text{var}^- \text{zero}))\end{aligned}$$

The terms $\boxed{\text{Tm } N \Gamma}$ of the focused 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`.

$$\text{var}^- \frac{N \in \Gamma}{\text{Tm } N \Gamma} \quad \text{abs} \frac{\langle\langle P \rangle\rangle (\text{Tm } N) \Gamma}{\text{Tm } (P \Rightarrow N) \Gamma} \quad \text{bind} \frac{\text{Tm } (\text{Comp } P) \Gamma \quad \langle\langle P \rangle\rangle (\text{Tm } N) \Gamma}{\text{Tm } N \Gamma}$$

Term interpretation $\boxed{\llbracket _ \rrbracket : \text{Tm } N \Gamma \rightarrow \llbracket \Gamma \rrbracket \dot{\rightarrow} \llbracket N \rrbracket}$ shall be as for CBPV except that we need

to exchange the interpretation function for binders $\boxed{\lambda \llbracket _ \rrbracket : \text{Tm } N (\Gamma.P) \rightarrow \llbracket \Gamma \rrbracket \dot{\rightarrow} \llbracket P \Rightarrow N \rrbracket}$. 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 bind the remaining atomic and negative crumbs. This matching can be defined for a generic evaluation function of type $\text{Ev } \mathcal{J} \Delta \Gamma = \llbracket \Gamma \rrbracket \Delta \rightarrow \mathcal{J} \Delta$.

$$\begin{aligned}\text{match} &: \llbracket P \rrbracket \Delta \rightarrow \langle\langle P \rangle\rangle \text{Ev } \mathcal{J} \Delta \dot{\rightarrow} \text{Ev } \mathcal{J} \Delta \\ \text{match } x & \quad (\text{hyp}^+ e) \quad \gamma = e(\gamma, \text{return } x) \\ \text{match } b & \quad (\text{hyp}^- e) \quad \gamma = e(\gamma, b) \\ \text{match } () & \quad (\text{split}_0 e) \quad \gamma = e \gamma \\ \text{match } (a_1, a_2) & \quad (\text{split}_2 e) \quad \gamma = \text{match } a_1 (\text{map}^{\langle\langle P \rangle\rangle} (\text{match } a_2) e) \gamma \\ \text{match } a & \quad \text{branch}_0 \quad \gamma = \text{magic } a \\ \text{match } (\iota_1 a_1) & \quad (\text{branch}_2 e_1 e_2) \gamma = \text{match } a_1 e_1 \gamma \\ \text{match } (\iota_2 a_2) & \quad (\text{branch}_2 e_1 e_2) \gamma = \text{match } a_2 e_2 \gamma\end{aligned}$$

With instantiations $\mathcal{J} = \llbracket N \rrbracket$ and $\llbracket _ \rrbracket : \text{Tm } N \dot{\rightarrow} \text{Ev } \llbracket N \rrbracket \Delta$, the interpretation $\lambda \llbracket t \rrbracket$ of binder $t : \langle\langle P \rangle\rangle (\text{Tm } N) \Gamma$ is defined as follows:

$$\lambda \llbracket t \rrbracket_{(\gamma : \llbracket \Gamma \rrbracket \Delta)} (\tau : \Delta \subseteq \Phi) (a : \llbracket P \rrbracket \Phi) = \text{match } a (\text{map}^{\langle\langle P \rangle\rangle} \llbracket _ \rrbracket t) (\text{ren } \tau \gamma)$$

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

5 Conclusion

We have implemented NbE for CBPV and polarized lambda calculus formulated with intrinsically well-typed syntax and presheaf semantics. 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.

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.

References

- 1 Andreas Abel. *A Polymorphic Lambda-Calculus with Sized Higher-Order Types*. PhD thesis, Ludwig-Maximilians-Universität München, 2006.
- 2 Thorsten Altenkirch, Peter Dybjer, Martin Hofmann, and Philip J. Scott. Normalization by evaluation for typed lambda calculus with coproducts. In *LICS'01*. IEEE CS Press, 2001. doi:10.1109/LICS.2001.932506.
- 3 Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. **Categorical reconstruction of a reduction free normalization proof**. In *CTCS'95*, volume 953 of *LNCS*. Springer, 1995. doi:10.1007/3-540-60164-3_27.
- 4 Thorsten Altenkirch and Tarmo Uustalu. Normalization by evaluation for $\lambda^{\rightarrow 2}$. In *FLOPS'04*, volume 2998 of *LNCS*. Springer, 2004. doi:10.1007/978-3-540-24754-8_19.
- 5 Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *JLC*, 2(3), 1992. doi:10.1093/logcom/2.3.297.
- 6 Vincent Balat, Roberto Di Cosmo, and Marcelo P. Fiore. Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums. In *POPL'04*. ACM, 2004. doi:10.1145/964001.964007.
- 7 Freiric Barral. *Decidability for non-standard conversions in lambda-calculus*. PhD thesis, Ludwig-Maximilians-University Munich, 2008.
- 8 Ulrich Berger and Helmut Schwichtenberg. An inverse to the evaluation functional for typed λ -calculus. In *LICS'91*. IEEE CS Press, 1991. doi:10.1109/LICS.1991.151645.
- 9 Richard S. Bird and Lambert G. L. T. Meertens. Nested datatypes. In *MPC'98*, volume 1422 of *LNCS*. Springer, 1998. doi:10.1007/BFb0054285.
- 10 Taus Brock-Nannestad and Carsten Schürmann. Focused natural deduction. In *LPAR'10*, volume 6397 of *LNCS*. Springer, 2010. doi:10.1007/978-3-642-16242-8_12.
- 11 Catarina Coquand. From semantics to rules: A machine assisted analysis. In *CSL'93*, volume 832 of *LNCS*. Springer, 1993. doi:10.1007/BFb0049326.
- 12 Olivier Danvy. Type-directed partial evaluation. In *POPL'96*. ACM, 1996. doi:10.1145/237721.237784.
- 13 Andrzej Filinski. A semantic account of type-directed partial evaluation. In *PPDP'99*, volume 1702 of *LNCS*. Springer, 1999. doi:10.1007/10704567_23.
- 14 Andrzej Filinski. Normalization by evaluation for the computational lambda-calculus. In *TLCA'01*, volume 2044 of *LNCS*. Springer, 2001. doi:10.1007/3-540-45413-6_15.
- 15 Felix Joachimski and Ralph Matthes. Short proofs of normalization for the simply-typed lambda-calculus, permutative conversions and Gödel's T. *AML*, 42(1), 2003. doi:10.1007/s00153-002-0156-9.
- 16 Neelakantan R. Krishnaswami. Focusing on pattern matching. In *POPL'09*. ACM, 2009. doi:10.1145/1480881.1480927.
- 17 Paul Blain Levy. Call-by-push-value: Decomposing call-by-value and call-by-name. *HOSC*, 19(4), 2006. doi:10.1007/s10990-006-0480-6.
- 18 Chuck Liang and Dale Miller. Focusing and polarization in intuitionistic logic. In *CSL'07*, volume 4646 of *LNCS*. Springer, 2007. doi:10.1007/978-3-540-74915-8_34.
- 19 Conor McBride and Ross Paterson. Applicative programming with effects. *JFP*, 18(1), 2008. doi:10.1017/S0956796807006326.
- 20 John Mitchell. *Foundations of Programming Languages*. Foundation of computing series. MIT Press, 1996.
- 21 Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1), 1991. doi:10.1016/0890-5401(91)90052-4.
- 22 Dag Prawitz. *Natural Deduction*. Almqvist & Wiksell, Stockholm, 1965.
- 23 José Espírito Santo. The polarized λ -calculus. *ENTCS*, 332, 2017. doi:10.1016/j.entcs.2017.04.010.
- 24 Gabriel Scherer. Deciding equivalence with sums and the empty type. In *POPL'17*. ACM, 2017. doi:10.1145/3009837.

- 25 Noam Zeilberger. *The Logical Basis of Evaluation Order and Pattern-Matching*. PhD thesis, Carnegie Mellon University, 2009.

A Equational Theory for STLC with Weak Sums

Term equality $t \cong t'$ for $t, t' : A \dashv \Gamma$ is the least congruence over the following axioms, which we have grouped into β (computation), η (extensionality), and π (permutation) rules. We may refer to subgroups, like η^N for the negative η -rules ($\eta^{\Rightarrow}, \eta^{\times}, \eta^1$). Note that all typing restrictions are implicit in the well-typedness assumption, e.g., $t \cong \text{unit}$ presupposes $t : 1 \dashv \Gamma$ since $\text{unit} : 1 \dashv \Gamma$.

$$\begin{array}{c}
\beta^{\Rightarrow} \frac{}{\text{app}(\text{abs } t) u \cong t[u]} \quad \beta^{\times} \frac{}{\text{prj}_i(\text{pair } t_1 t_2) \cong t_i} \quad \beta^{+} \frac{}{\text{case}(\text{inj}_i t) t_1 t_2 \cong t_i} \\
\\
\eta^{\Rightarrow} \frac{}{t \cong \text{abs}(\text{app}(\text{ren wk } t) v_0)} \quad \eta^{\times} \frac{}{t \cong \text{pair}(\text{prj}_1 t) (\text{prj}_2 t)} \quad \eta^1 \frac{}{t \cong \text{unit}} \\
\\
\eta^{+} \frac{}{t \cong \text{case } t (\text{inj}_1 v_0) (\text{inj}_2 v_0)} \quad \eta^0 \frac{}{t \cong \text{abort } t} \\
\\
\pi^{\Rightarrow,0} \frac{}{\text{app}(\text{abort } t) u \cong \text{abort } t} \quad \pi^{\Rightarrow,+} \frac{t'_i = \text{app } t_i (\text{ren wk } u)}{\text{app}(\text{case } t t_1 t_2) u \cong \text{case } t t'_1 t'_2} \\
\\
\pi^{\times,0} \frac{}{\text{prj}_i(\text{abort } t) \cong \text{abort } t} \quad \pi^{\times,+} \frac{}{\text{prj}_i(\text{case } t t_1 t_2) \cong \text{case } t (\text{prj}_i t_1) (\text{prj}_i t_2)} \\
\\
\pi^{+,0} \frac{}{\text{case}(\text{abort } t) t_1 t_2 \cong \text{abort } t} \quad \pi^{+,+} \frac{t'_i = \text{case } t_i u'_1 u'_2 \quad u'_j = \text{ren}(\text{lift wk}) u_j}{\text{case}(\text{case } t t_1 t_2) u_1 u_2 \cong \text{case } t t'_1 t'_2} \\
\\
\pi^{0,0} \frac{}{\text{abort}(\text{abort } t) \cong \text{abort } t} \quad \pi^{0,+} \frac{}{\text{abort}(\text{case } t t_1 t_2) \cong \text{case } t (\text{abort } t_1) (\text{abort } t_2)}
\end{array}$$

The different law classes contribute to the shape and completeness of the normal forms in the following way: Thanks to the η -laws, a normal form of negative type is always an introduction. A normal form of sum type $(+, 0)$ is always a case tree whose leaves are injections. A normal form of atomic type (o) is always case tree whose leaves are neutral. The β and π laws determine the shape of neutrals. Thanks to the β -laws, the principal argument of normal eliminations cannot be an introduction. Because of the negative permutation rules $\pi^{N,P}$ it cannot be positive elimination (**case**, **abort**) either, which leaves only negative eliminations in neutrals (**app**, **prj**). Finally the positive permutation rules $\pi^{P,P}$ guarantee that having neutral scrutinees in case trees is sufficient.