Exponential Elimination for Bicartesian Closed Categorical Combinators

Nachiappan Valliappan
Chalmers University
Sweden
nacval@chalmers.se

Alejandro Russo
Chalmers University
Sweden
russo@chalmers.se

ABSTRACT

Categorical combinators offer a simpler alternative to typed lambda calculi for static analysis and implementation. Since categorical combinators are accompanied by a rich set of conversion rules which arise from categorical laws, they also offer a plethora of opportunities for program optimization. It is unclear, however, how such rules can be applied in a systematic manner to eliminate intermediate values such as *exponentials*, the categorical equivalent of higher-order functions, from a program built using combinators. Exponential elimination simplifies static analysis and enables a simple closure-free implementation of categorical combinators—reasons for which it has been sought after.

In this paper, we prove exponential elimination for *bicartesian closed* categorical (BCC) combinators using normalization. We achieve this by showing that BCC terms can be normalized to normal forms which obey a weak subformula property. We implement normalization using Normalization by Evaluation, and also show that the generated normal forms are correct using logical relations.

CCS CONCEPTS

• **Software and its engineering** → **Functional languages**; • **Theory of computation** → *Proof theory*; *Type theory*.

KEYWORDS

normalization by evaluation, categorical combinators, defunctionalization, subformula property

ACM Reference Format:
Nachiappan Valliappan and Alejandro Russo. 2019. Exponential Elimination for Bicartesian Closed Categorical Combinators. In *Principles and Practice of Programming Languages 2019 (PPDP '19), October 7–9, 2019, Porto, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3354166.3354185

1 INTRODUCTION

Categorical combinators are combinators designed after arrows, or *morphisms*, in category theory. Although originally introduced to present the connection between lambda calculus and cartesian closed categories (CCCs) [Curien 1986], categorical combinators have attracted plenty of attention in formal analysis and implementation of various lambda calculi. For example, they are commonly used to formulate an evaluation model based on abstract machines [Cousineau et al. 1987; Lafont 1988]. Abadi et al. [1991] observe that categorical combinators "make it easy to derive machines for the λ-calculus and to show the correctness of these machines". This ease is attributed to the absence of variables in combinators, which avoids the difficulty with variable names, typing contexts, substitution, etc. Recently, categorical combinators have also been used in practical applications for programming *smart contracts* on the blockchain [O'Connor 2017] and compiling functional programs [Elliott 2017].

Since categorical combinators are based on categorical models, they are accompanied by a rich set of *conversion* rules (between combinator terms) which emerge from the equivalence between morphisms in the model. These conversion rules form the basis for various correct program transformations and optimizations. For example, Elliott [2017] uses conversion rules from CCCs to design various rewrite rules to optimize the compilation of Haskell programs to CCC combinators. The availability of these rules raises a natural question for optimizing terms in categorical combinator languages: can intermediate values be eliminated by applying the conversion rules whenever possible?

The ability to eliminate intermediate values in a categorical combinator language has plenty of useful consequences, just as in functional programming. For example, the elimination of *exponentials*, the equivalent of high-order functions, from BCC combinators solves problems created by exponentials in static analysis [Valliappan et al. 2018], and has also been sought after for interpreting functional programs in categories without exponentials ([Elliott 2017], Section 10.2). It has been shown that normalization erases higher-order functions from a program with first-order input and output types in the simply typed lambda calculus (STLC) with products and sums [Najd et al. 2016]—also known as *defunctionalization* [Reynolds 1998]. Similarly, can we erase exponentials and other intermediate values by normalizing programs in the equally expressive *bicartesian closed* categorical (BCC) combinators?

In this paper, we implement normalization for BCC combinators towards eliminating intermediate values, and show that it yields exponential elimination. We first recall the term language and conversion rules for BCC combinators (Section 2), and provide a brief overview of the normalization procedure (Section 3). Then, we identify normal forms of BCC terms which obey a *weak subformula* property and prove exponential elimination by showing that these normal forms can be translated to an equivalent first-order combinator language without exponentials (Section 4 and Section 5).

To assign a normal form to every term in the language, we implement a normalization procedure using *Normalization by Evaluation* (NbE) [Berger et al. 1998; Berger and Schwichtenberg 1991] (Section 6). We then prove, using *Kripke logical relations* [Mitchell and

PPDP '19, October 7–9, 2019, Porto, Portugal

Moggi 1991], that normal forms of terms are consistent with the conversion rules by showing that they are inter-convertible. (Section 7). Furthermore, we show that exponential elimination can be used to simplify static analysis—while retaining expressiveness—of a combinator language called Simplicity (Section 8). Finally, we conclude by discussing related work (Section 9) and final remarks (Section 10).

Although we only discuss the elimination of exponentials in this paper, the elimination of intermediate values of other types can also be achieved likewise—except for *products*. The reason for this becomes apparent when we discuss the weak subformula property (in Section 5.1).

We implement normalization and mechanize the correctness proof in the dependently-typed language Agda [Bove et al. 2009; Norell 2007]. This paper is also written in literate Agda since dependent types provide a uniform framework for discussing both programs and proofs. We use category theoretic terminology to organize the implementation based on the categorical account of NbE by Altenkirch et al. [1995]. However, all the definitions, algorithms, and proofs here are written in vanilla Agda, and the reader may view them as regular programming artifacts. Hence, we do not require that the reader be familiar with advanced categorical concepts. We discuss the important parts of the implementation here, and encourage the curious reader to see the complete implementation[1] for further details.

2 BCC COMBINATORS

A BCC combinator has an input and an output type, which can be one of the following: **1** (for unit), **0** (for empty), ∗ (for product), + (for sum), ⇒ (for exponential) and base (for base types). The Agda data type BCC (see Figure 1) defines the term language for BCC combinators. In the definition, the type Ty denotes a BCC type, and Set denotes a type definition in Agda (like ∗ in Haskell). Note that the type variables a, b and c are implicitly quantified and hidden here. The combinators are self-explanatory and behave like their functional counterparts. Unlike functions, however, these combinators do not have a notion of variables or typing contexts.

```
data BCC : Ty → Ty → Set where
  id    : BCC a a
  _•_   : BCC b c → BCC a b → BCC a c
  unit  : BCC a 1
  init  : BCC 0 a
  exl   : BCC (a ∗ b) a
  exr   : BCC (a ∗ b) b
  pair  : BCC a b → BCC a c → BCC a (b ∗ c)
  inl   : BCC a (a + b)
  inr   : BCC b (a + b)
  match : BCC a c → BCC b c → BCC (a + b) c
  curry : BCC (c ∗ a) b → BCC c (a ⇒ b)
  apply : BCC (a ⇒ b ∗ a) b
```

Figure 1: BCC Combinators

```
data _≈_ : BCC a b → BCC a b → Set where
  -- categorical rules
  idr         : f • id ≈ f
  idl         : id • f ≈ f
  assoc       : f • (g • h) ≈ f • (g • h)
  -- elimination rules
  exl-pair    : (exl • pair f g) ≈ f
  exr-pair    : (exr • pair f g) ≈ g
  match-inl   : (match f g • inl) ≈ f
  match-inr   : (match f g • inr) ≈ g
  apply-curry : apply • (curry f ⊗ id) ≈ f
  -- uniqueness rules
  uniq-init   : init ≈ f
  uniq-unit   : unit ≈ f
  uniq-pair   : exl • h ≈ f → exr • h ≈ g → pair f g ≈ h
  uniq-curry  : apply • h ⊗ id ≈ f → curry f ≈ h
  uniq-match  : h • inl ≈ f → h • inr ≈ g → match f g ≈ h
  -- equivalence and congruence rules
  refl        : f ≈ f
  sym         : f ≈ g → g ≈ f
  trans       : f ≈ g → g ≈ h → f ≈ h
  congl       : x ≈ y → f • x ≈ f • y
  congr       : x ≈ y → x • f ≈ y • f
```

Figure 2: Conversion rules for BCC

The BCC combinators are accompanied by a number of conversion rules which emerge from the equational theory of bicartesian closed categories [Awodey 2010]. These rules can be formalized as an equivalence relation _≈_ : BCC a b → BCC a b → Set (see Figure 2). In the spirit of categorical laws, the type-specific conversion rules can be broadly classified as *elimination* and *uniqueness* (or *universality*) rules. The elimination rules state when the composition of two terms can be eliminated, and uniqueness rules state the unique structure of a term for a certain type. For example, the conversion rules for products include two elimination rules (exl-pair, exr-pair) and a uniqueness rule (uniq-pair):

Note that the operator ⊗ used in the exponential elimination rule (apply-curry) is defined below. It pairs two BCC terms using pair and applies them on each component of a product. The components are projected using exl and exr respectively.

```
_⊗_ : BCC a b → BCC c d → BCC (a ∗ c) (b ∗ d)
f ⊗ g = pair (f • exl) (g • exr)
```

The standard $\beta\eta$ conversion rules of STLC [Altenkirch et al. 2001; Balat et al. 2004] can be derived from the conversion rules specified here. This suggests that we can perform β and η conversion for BCC terms, and normalize them as in STLC. Let us look at a few simple examples.

Example 1. For a term f : BCC a $(b * c)$, pair (exl • f) (exr • f) can be converted to f as follows.

```
eta∗ : pair (exl • f) (exr • f) ≈ f
eta∗ = uniq-pair refl refl
```

[1] https://github.com/nachivpn/expelim

The constructor refl states that the relation ≈ is reflexive. The conversion above corresponds to η conversion for products in STLC.

Example 2. Suppose that we define a combinator uncurry as follows.

```
uncurry : BCC a (b ⇒ c) → BCC (a * b) c
uncurry f = apply • f ⊗ id
```

Given this definition, a term curry (uncurry f) can be converted to f, by unfolding the definition of uncurry—as curry (apply • f ⊗ id)—and then using uniq-curry refl.

```
eta⇒ : curry (uncurry f) ≈ f
eta⇒ = uniq-curry refl
```

Note that Agda unfolds the definition of uncurry automatically for us. The conversion above corresponds to η conversion for functions in STLC.

Example 3. Given a term t : BCC a $(b * c)$ such that $t \approx$ (pair f g) • h : BCC a $(b * c)$, t can be converted to the term pair (f • h) (g • h) using equational reasoning such as the following.

```
t
≈ (pair f g) • h                            By definition
≈ pair (exl • pair f g • h) (exr • pair f g • h)    By example 1
≈ pair (f • h) (exr • pair f g •h)          By exl-pair
≈ pair (f • h) (g • h)                      By exr-pair
```

Example 4. Given f : BCC a $(b \Rightarrow c)$ and g : BCC a b, if f can be converted to curry f', then the term (apply • pair f g) : BCC a c can be converted to f' • pair id g (the implementation is left as an exercise for the reader). Notice that the combinators curry and apply are eliminated in the result of the conversion. This conversion corresponds to β conversion for functions in STLC, and forms the basis for exponential elimination.

3 OVERVIEW OF NORMALIZATION

Our goal is to implement a normalization algorithm for BCC terms and show that normalization eliminates exponentials. We will achieve the latter using a syntactic property of normal forms called the weak subformula property. To make this property explicit, we define normal forms as a separate data type Nf as follows.

```
data Nf : Ty → Ty → Set where
```

Normal forms are not themselves BCC terms, but they can be embedded into BCC terms using a quotation function q which has the following type.

```
q : Nf a b → BCC a b
```

To prove that normalization eliminates exponentials, we show that normal forms with first-order types can be quoted into a first-order combinator language, called DBC, as follows.

```
qD : firstOrd a → firstOrd b → Nf a b → DBC a b
```

The data type DBC is defined syntactically identical to BCC without the exponential combinators curry and apply, and with an additional distributivity combinator distr (see Section 5).

Normalization based on rewriting techniques performs syntactic transformations of a term to produce a normal form. NbE, on the other hand, normalizes a term by evaluating it in a suitable semantic model, and extracting a normal form from the resulting value. Evaluation is implemented as an interpreter function eval, and extraction of normal forms—also called *reification*—is implemented as a function reify (see Section 6). These functions have the following types.

```
eval : BCC a b → (⟦ a ⟧ ↠ ⟦ b ⟧)
reify : (⟦ a ⟧ ↠ ⟦ b ⟧) → Nf a b
```

The type ⟦ a ⟧ is an interpretation of a BCC type a in the model, and similarly for b. The type ⟦ a ⟧ ↠ ⟦ b ⟧, on the other hand, is a function between interpretations (to be defined later) and denotes the interpretation of a BCC term of type BCC a b.

Normalization is achieved by evaluating a term and then reifying it, and is thus implemented as a function norm defined as follows.

```
norm : BCC a b → Nf a b
norm t = reify (eval t)
```

To ensure that the normal form generated for a term is correct, we must ensure that it is convertible to the original term. This correctness theorem is stated by quoting the normal form as follows.

```
correct-nf : (t : BCC a b) → t ≈ q (norm t)
```

We prove this theorem using logical relations between BCC terms and values in the semantic model (see Section 7).

4 SELECTIONS

The evaluation of a term requires an input of the appropriate type. During normalization, since we do not have the input, we must assign a reference to the unknown input value and use this reference to represent the value. In lambda calculus, these references are simply variables. Since BCC combinators lack the notion of variables, we must identify the subset of BCC terms which (intuitively) play the counterpart role—which is the goal of this section.

If we think of the typing context as the "input type" of a lambda term, then variables are essentially indices which *project* an unknown value from the input (a substitution). This is because typing contexts enforce a product-like structure on the input. For example, the variable x in the body of lambda term $\Gamma, x : a \vdash x : a$ projects a value of type a from the context $\Gamma, x : a$. The BCC equivalent of $\Gamma, x : a \vdash x : a$ is the term exl : $(\Gamma * a)$ a. Unlike lambda terms, however, BCC terms do not enforce a specific type structure on the input, and may also return the input entirely as id : $(\Gamma * a)$ $(\Gamma * a)$. Hence, as opposed to projections, we need a notion of *selections*.

Specific BCC terms can be used to *select* an unknown value from the input, and these terms can be defined explicitly by the data type Sel (see Figure 3). A term of type Sel a b denotes a selection of b from the input a. When the input is a product, the constructor drop drops the second component, and applies a given selection to the first component. The constructor keep, on the other hand, keeps the second component unaltered and applies a selection to the first component. We cannot select further from the input if it is not a product, and hence the remaining constructors, with the prefix end, state that we must simply return the input as is—thereafter referred to as end- constructors.

```
data Sel : Ty → Ty → Set where
  endu : Sel 1 1
  endi  : Sel 0 0
  endb : Sel base base
  ends  : Sel (a + b) (a + b)
  ende  : Sel (a ⇒ b) (a ⇒ b)
  drop  : Sel a b → Sel (a ∗ c) b
  keep  : Sel a b → Sel (a ∗ c) (b ∗ c)
```

Figure 3: Selections

Note that the four end- constructors enable the definition of a unique *identity* selection[2], iden : Sel a a. This selection can be defined by induction on the type a, where the only interesting case of products is defined as below. The remaining cases can be defined using the appropriate end- constructor.

```
iden : {a : Ty} → Sel a a
iden {a₁ ∗ a₂} = keep iden
-- end- for remaining cases
```

Figure 4 illustrates the use of selections by examples.

```
          drop iden : Sel ((a + b) ∗ c) (a + b)
       keep (drop iden) : Sel (a ∗ b ∗ c) (a ∗ c)
drop (keep (drop iden)) : Sel (a ∗ b ∗ c ∗ d) (a ∗ c)
```

Figure 4: Examples of selections

Selections form the basis for the semantic interpretation of BCC terms, and hence enable the implementation of NbE. To this extent, they have the following properties.

Property 4.1 (Category of selections)**.** Selections define a category where the objects are types and a morphism between two types a and b is a selection of type Sel a b. The identity morphisms are defined by iden, and morphism composition can be defined by straight-forward induction on the morphisms as a function of type _∘_ : Sel b c → Sel a b → Sel a c. The identity and associativity laws of a category (sel-idl, sel-idr and sel-assoc below) can be proved using Agda's built-in syntactic equality ≡ by induction on the morphisms. These laws have the following types in Agda.

```
sel-idl : iden ∘ s ≡ s
sel-idr : s ∘ iden ≡ s
sel-assoc : (s₁ ∘ s₂) ∘ s₃ ≡ s₁ ∘ (s₂ ∘ s₃)
```

Property 4.2 (Faithful embedding)**.** Selections can be faithfully embedded into BCC terms since they are simply a subset of BCC terms. This embedding can be implemented by induction on the selection, as follows.

```
embSel : Sel a b → BCC a b
embSel (drop e) = embSel e • exl
embSel (keep e) = pair (embSel e • exl) exr
-- id for remaining cases
```

5 NORMAL FORMS

In this section, we present normal forms for BCC terms, and prove exponential elimination using them. It is important to note that these normal forms are *not* normal forms of the conversion rules specified by the relation ≈, but rather are a convenient syntactic restriction over BCC terms for proving exponential elimination. Precisely, they are normal forms of BCC terms which obey a weak subformula property—defined later in this section. This characterization is based on normal forms of proofs in logic, as opposed to normal forms of terms in lambda calculus.

Normal forms are defined mutually with *neutral* forms (see Figure 5). Roughly, neutral forms are eliminators applied to selections, and they represent terms which are blocked during normalization due to unavailability of the input. The neutral form constructor sel embeds a selection as a base case of neutrals; while fst, snd and app represent the composition of the eliminators exl, exr and apply (respectively) to neutrals.

The normal form constructors unit, pair and curry represent their BCC term counterparts; ne-0 and ne-b embed neutrals which return values of type 0 and base (respectively) into normal forms; left and right represent the composition of the injections inl and inr respectively; and case represents the BCC term Case below, which is an eliminator of sums defined using distributivity of products over sums. Note that the BCC term Distr implements this distributivity requirement, and can be derived using exponentials—see Appendix A.2.

```
-- Distr : BCC (a ∗ (b + c)) ((a ∗ b) + (a ∗ c))
Case : BCC a (b + c) → BCC (a ∗ b) d → BCC (a ∗ c) d → BCC a d
Case x f g = match f g • Distr • pair id x
```

The quotation functions are implemented as a simple syntax-directed translation by mapping neutrals and normal forms to their BCC counterparts as discussed above. For example, the quotation of the neutral form fst x—where x has the type Ne a $(b * c)$—is simply exl : $(b * c)$ b composed with the quotation of x. Similarly, the quotation of left x is inl composed with the quotation of its argument x. We use the derived term Case to quote the normal form case.

Note that the normal forms resemble $\beta\eta$ long forms of STLC with products and sums [Abel and Sattler 2019], but differ with respect to the absence of typing contexts and variables. In place of variables, we use selections in neutral forms—this is an important difference since it allows us to implement *reflection*, a key component of reification (discussed later in Section 6).

In the rest of this section, we will define the weak subformula property, show that all normal forms obey it, and prove exponential elimination as a corollary.

5.1 Weak Subformula Property

To understand the need for a subformula property, let us suppose that we are given a term t : BCC $(1 * 1)$ 1. Does t use exponentials? Unfortunately, we cannot say much about the presence of curry and apply in the subterms without inspecting the body of the term

[2] We prefer to derive the identity selection as opposed to adding it as a constructor, to avoid ambiguity which could be created between selections iden and (keep iden), both of the type Sel $(a_1 * a_2)$ $(a_1 * a_2)$. The derived identity avoids this ambiguity by definition.

```
data Nf (a : Ty) : Ty → Set where
  unit  : Nf a 1
  ne-0  : Ne a 0 → Nf a b
  ne-b  : Ne a base → Nf a base
  left  : Nf a b → Nf a (b + c)
  right : Nf a c → Nf a (b + c)
  pair  : Nf a b → Nf a c → Nf a (b ∗ c)
  curry : Nf (a ∗ b) c → Nf a (b ⇒ c)
  case  : Ne a (b + c) → Nf (a ∗ b) d → Nf (a ∗ c) d → Nf a d

data Ne (a : Ty) : Ty → Set where
  sel  : Sel a b → Ne a b
  fst  : Ne a (b ∗ c) → Ne a b
  snd  : Ne a (b ∗ c) → Ne a c
  app  : Ne a (b ⇒ c) → Nf a b → Ne a c

q : Nf a b → BCC a b
q unit          = unit
q (ne-b x)      = qNe x
q (ne-0 x)      = init • qNe x
q (left n)      = inl • q n
q (right n)     = inr • q n
q (pair m n)    = pair (q m) (q n)
q (curry n)     = curry (q n)
q (case x m n)  = Case (qNe x) (q m) (q n)

qNe : Ne a b → BCC a b
qNe (sel x)     = embSel x
qNe (fst x)     = exl • qNe x
qNe (snd x)     = exr • qNe x
qNe (app x n)   = apply • pair (qNe x) (q n)
```

Figure 5: Normal forms and quotation

itself. Term t could be something as simple as exl or it could be:

```
apply • (pair (curry unit • exl) exr) : BCC (1 ∗ 1) 1
```

But with an appropriate subformula property, however, this becomes an easy task. Let us suppose that t : BCC (**1** ∗ **1**) **1** has a property that the input and output types of all its subterms occur in t's input (**1** ∗ **1**) and/or output (**1**) type. In this case, what can we say about the presence of curry and/or apply in t? Well, it would not contain any! The input and output types of *all* the subterms would be **1** and/or products of it, and hence it is impossible to find a curry or an apply in a subterm. Let us define this property precisely and show that normal forms obey it by construction.

The occurrence of a type in another is defined as follows.

Definition 5.1 (Weak subformula)**.** A type b is a weak subformula of a if $b \lhd a$, where $\lhd$ is defined as follows.

```
data _⊲_ : Ty → Ty → Set where
  self  : a ⊲ a
  subl  : a ⊲ b → a ⊲ (b ⊗ c)
  subr  : a ⊲ c → a ⊲ (b ⊗ c)
  subp  : a ⊲ c → b ⊲ d → (a ∗ b) ⊲ (c ∗ d)
```

For a binary type operator ⊗ which ranges over ∗, + or ⇒, this definition states that:

- a is a weak subformula of a (self)
- a is a weak subformula of $b \otimes c$ if a is a weak subformula of b (subl) or a is a weak subformula of c (subr)
- $a * b$ is a weak subformula of $c * d$ if a is a weak subformula of c and b is a weak subformula of d (subp).

The constructors self, subl and subr define precisely the concept of a subformula in proof theory [Troelstra and Schwichtenberg 2000]. For BCC terms, however, we also need subp which weakens the subformula definition by relaxing it *up to products*. To understand this requirement, we must first define the following property for normal forms.

Definition 5.2 (Weak subformula property)**.** A normal form of type Nf a b obeys the weak subformula property if, for all its subterms of type Nf i o, we have that $i \lhd a * b$ and $o \lhd a * b$.

Do all normal forms obey this property? It is easy to see that the normal forms constructed using unit, left, right and pair obey the weak subformula property given their subterms do the same. For instance, the constructor left returns a normal form of type Nf a $(b + c)$, where the input type (a) and output type (b) of its subterm Nf a b occur in a and $(b + c)$. Hence, if a subterm t : Nf a b obeys the weak subformula property, then so does left t.

To understand why curry satisfies the weak subformula property, recall its definition as a normal form constructor of type BCC $(c * a)$ b → BCC c $(a \Rightarrow b)$. The input type $c * a$ of its subterm argument is evidently not a subformula—as usually defined in proof theory—of the types c or $a \Rightarrow b$. However, by subp, we have that the type $c * a$ is a weak subformula of the *product* of the input and output types $c * (a \Rightarrow b)$. This is precisely the need for weakening the definition of a subformula with subp[3]. Specifically, the proof of $(c * a) \lhd c * (a \Rightarrow b)$ is given by subp (self) (subl self).

On the other hand, the definition of the constructor case looks a bit suspicious since it allows the types b and c which do not occur in final type Nf a d. To understand why case also satisfies the weak subformula property, we must establish the following property about neutral forms, which we shall call *neutrality*.

Property 5.1. Given a neutral form Ne a b, we have that b is a weak subformula of a, i.e., neutrality : Ne a b → $b \lhd a$.

Proof. By induction on neutral forms. For the base case sel, we need a lemma about neutrality of selections, which can be implemented by an auxiliary function neutrality-sel : Sel a b → $b \lhd a$ by induction on the selection. For the other cases, we simply apply the induction hypothesis on the neutral subterm. □

Due to neutrality of the neutral form Ne a $(b + c)$ in the definition of case, we have that $(b + c) \lhd a$, and hence $(b + c) \lhd (a * d)$. As a result, case also obeys the weak subformula property. Similarly, ne-**0** and ne-b also obey the weak subformula property as a consequence of neutrality. Thus, we have the following theorem.

[3] In logic, however, the requirement for weakening a subformula by products is absent, since an equivalent definition of curry as $\Gamma, a \vdash b \rightarrow \Gamma \vdash a \Rightarrow b$ uses context extension (,) instead of products (∗)

Theorem 5.1. All normal forms, as defined by the data type Nf, satisfy the weak subformula property.

Proof. By induction on normal forms, as discussed above. □

Notice that, unlike normal forms, arbitrary BCC terms need not satisfy the weak subformula property. The term apply • (pair (curry unit • exl) exr) discussed above is already an example of such a term. More specifically, its subterm apply has the input type $(\mathbf{1} \Rightarrow \mathbf{1}) * \mathbf{1}$, which does not occur in $(\mathbf{1} * \mathbf{1}) * \mathbf{1}$—i.e., $(\mathbf{1} \Rightarrow \mathbf{1}) * \mathbf{1} \ntriangleleft (\mathbf{1} * \mathbf{1}) * \mathbf{1}$. However, all BCC terms, including the ones which do not satisfy the weak subformula property, can be converted to terms which satisfy this property. This conversion is precisely the job of normalization. For instance, the previous example can be converted to unit : BCC $(\mathbf{1} * \mathbf{1})$ $\mathbf{1}$ using uniq-unit. A normalization algorithm performs such conversions automatically whenever possible.

Since neutral forms offer the intuition of an "eliminator", it might be disconcerting to see case, an eliminator of sums, oddly defined as a normal form. But suppose that it was defined in neutrals as follows.

```
case? : Ne a (b + c) → Nf (a ∗ b) d → Nf (a ∗ c) d → Ne a d
```

Such a definition breaks neutrality (Property 5.1) since we cannot prove that $d \triangleleft a$, and subsequently breaks the weak subformula property of normal forms (Theorem 5.1). But what about the following definition where the first argument to case is normal, instead of neutral?

```
case? : Nf a (b + c) → Nf (a ∗ b) d → Nf (a ∗ c) d → Nf a d
```

Such a definition also breaks the weak subformula property—for the exact same reason which caused our suspicion in the first place: b and c do not occur in a, d or $a * d$.

5.2 Syntactic Elimination of Exponentials

Exponential elimination can be proved as a simple corollary of the weak subformula property of normal forms. If a and b are first-order types, i.e., if the type constructor ⇒ does not occur in types a or b, then we can be certain that the subterms of Nf a b do not use curry (from Nf) or app (from Ne). This follows directly from the weak subformula property (Theorem 5.1). To show this explicitly, let us quote such normal forms to a first-order combinator language based on *distributive bicartesian categories* (DBC) [Awodey 2010].

The DBC term language is defined by the data type DBC, which includes all the BCC term constructors except Curry and Apply—although most of them have been left out here for brevity. Additionally, it also has a distributivity constructor distr which distributes products over sums. The constructor distr is needed to implement the BCC term Case, which is in turn needed to quote the normal form case (as earlier). This is because distributivity can no longer be derived in the absence of exponentials.

To restrict the input and output to first-order types, suppose that we define a predicate on types, firstOrd : Ty → Set, which disallows the occurrence of exponentials in a type. Given this predicate, we can now define quotation functions qNeD and qD as below. The implementation of the function qNeD is similar to that of the function qNe (discussed earlier) for most cases, and similarly for qD. The only interesting cases are that of the exponentials, and these can be implemented as follows.

```
qNeD : firstOrd a → Ne a b → DBC a b
qNeD p (app n _) = ⊥-elim (expNeutrality p n)

qD : firstOrd a → firstOrd b → Nf a b → DBC a b
qD p q (curry n) = ⊥-elim q
```

For neutrals, we escape having to quote app because such a case is impossible: We have a proof p : firstOrd a which states that input type a does not contain any exponentials. However, the exponential return type of n, say $b \Rightarrow c$, must occur in a by neutrality of n : Ne a $(b \Rightarrow c)$—which contradicts the proof p. Hence, such a case is not possible. This reasoning is implemented by applying the function ⊥-elim with a proof of impossibility produced using an auxiliary function expNeutrality : firstOrd a → Ne a b → firstOrd b. Similarly, we escape the quotation of the normal form curry since Agda automatically inferred that such a case is impossible. This is because a proof q which states that the output b is not an exponential, is contradicted by the definition of curry which states that it must be—hence q must be impossible.

Although we have shown the syntactic elimination of exponentials using normal forms, we are yet to show that there exists an equivalent normal form for every term. For this, we must implement normalization and prove its correctness.

```
data DBC : Ty → Ty → Set where
  id   : DBC a a
  _•_ : DBC b c → DBC a b → DBC a c
  -- exl, exr, pair, init
  -- inl, inr, match, unit
  distr : DBC (a ∗ (b + c)) ((a ∗ b) + (a ∗ c))
```

Figure 6: DBC combinators

6 NORMALIZATION FOR BCC

To implement evaluation and reification, we must first define an appropriate interpretation for types and terms. A naive Set-based interpretation (such as ⟦_⟧n below) which maps BCC types to their Agda counterparts fails quickly.

```
⟦ 1       ⟧n  = ⊤
⟦ 0       ⟧n  = ⊥
⟦ base    ⟧n  = ??
⟦ t₁ ∗ t₂ ⟧n  = ⟦ t₁ ⟧n × ⟦ t₂ ⟧n
⟦ t₁ + t₂ ⟧n  = ⟦ t₁ ⟧n ⊎ ⟦ t₂ ⟧n
⟦ t₁ ⇒ t₂ ⟧n = ⟦ t₁ ⟧n → ⟦ t₂ ⟧n
```

What should be the correct interpretation of the type base? The naive interpretation also makes it impossible to implement reflection for the empty and sum types, since their inhabitants cannot be faithfully represented in such an interpretation (see Section 6.3). To address this problem, we must first define an appropriate semantic model.

6.1 Interpretation in Presheaves

To implement NbE, our choice of semantic model for interpretation of BCC types must allow us to implement both evaluation and reification. NbE for STLC can be implemented by interpreting it in *presheaves* over the category of *weakenings* [Altenkirch et al. 1995] [Abel and Sattler 2019]. The semantic equivalence of BCC combinators and STLC suggests that it should be possible to interpret BCC terms in presheaves as well. The difference, however, is that we will interpret BCC in presheaves over the category of selections (instead of weakenings). Such a presheaf, for our purposes, is simply the following record definition:

```
record Pre : Set₁ where
  field
    In : Ty → Set
    lift : {i j : Ty} → Sel j i → (In i → In j)
```

Intuitively, an occurrence In i can be understood as a Set interpretation indexed by an input type i. The function lift can be understood as a utility function which converts a semantic value for the input i to a value for a "larger" input j, for a given selection of i from j.

For the category theory-aware reader, notice that Pre matches the expected definition of a presheaf as a functor which maps objects (using In) and morphisms (using lift) in the opposite category of the category of selections to the Set-category. We skip the functor laws of the presheaf in the Pre record to avoid cluttering the normalization procedure, and instead prove them separately as needed for the correctness proofs later.

With the definition of a presheaf, we can now implement the desired interpretation of types as ⟦_⟧ : Ty → Pre. Intuitively, a presheaf model allows us to interpret a BCC type as an Agda type *for* a given input type—or equivalently for a given typing context. To implement the function ⟦_⟧, we will need various presheaf constructions (instances of Pre)—defining these is the goal of this section. Note that all names ending with ' denote a presheaf.

```
1' : Pre                    0' : Pre
1' .In _     = ⊤            0' .In _ = ⊥
1' .lift _ _ = tt           0' .lift _ ()
```

Figure 7: Unit and Empty presheaves

The unit presheaf maps all input types to the type ⊤ (unit type in Agda) and empty presheaf maps it to ⊥ (empty type in Agda) (see Figure 7). The implementation of lift is trivial in both cases since the only inhabitant of ⊤ is tt, and ⊥ has no inhabitants.

The product of two presheaves A and B is defined component-wise as follows.

```
_*'_ : Pre → Pre → Pre
(A *' B) .In i          = A .In i × B .In i
(A *' B) .lift s (x , y) = (A .lift s x , B .lift s y)
```

The function lift is implemented component-wise since s has the type Sel j i, x has the type A .In i, y has the type B .In i, and the result must be a value of type A .In j × B .In j. Similarly, the sum of two presheaves is also defined component-wise as follows.

```
_+'_ : Pre → Pre → Pre
(A +' B) .In i = A .In i ⊎ B .In i
(A +' B) .lift s (inj₁ x) = inj₁ (A .lift s x)
(A +' B) .lift s (inj₂ x) = inj₂ (B .lift s x)
```

It is tempting to implement an exponential presheaf _⇒'_ component wise (like _x'_), but this fails at the implementation of lift: given Sel j i, we can not lift a function (A .In i → B .In i) to (A .In j → B .In j) directly. To solve this, we must implement a slightly more general version which allows for lifting as follows.

```
_⇒'_ : Pre → Pre → Pre
(A ⇒' B) .In i = {i₁ : Ty} → Sel i₁ i → A .In i₁ → B .In i₁
(A ⇒' B) .lift s f s' = f (s ∘ s')
```

Recall that the operator ∘ implements composition of selections. The interpretation of the exponential presheaf is defined for a given input type i, as a function (space) for all selections of the type i_1 from i [MacLane and Moerdijk 1992]—which gives us the required lifting by composition of the selections.

BCC terms also define presheaves when indexed by the output type.

```
BCC' : Ty → Pre
BCC' o .In i = BCC i o
BCC' o .lift s t = liftBCC s t
```

To implement liftBCC, recollect that selections can be embedded into BCC terms using the embSel function (from Section 4). Hence, lifting BCC terms can be implemented easily using composition, as follows.

```
liftBCC : Sel j i → BCC i a → BCC j a
liftBCC s t = t ● embSel s
```

Similarly, normal forms and neutral forms also define presheaves when indexed by the output type (see Figure 8). The implementation of lift for normal forms (litfNf) can be defined by straight-forward induction on the normal form—and similarly for liftNe.

```
                            Ne' : Ty → Pre
Nf' : Ty → Pre              Ne' o .In i = Ne i o
Nf' o .In i = Nf i o        Ne' o .lift s n = liftNe s n
Nf' o .lift s n = liftNf s n
```

Figure 8: Normal and Neutral form presheaves

For notational convenience, let us define a type alias Sem for values in the interpretation:

```
Sem : Ty → Pre → Set
Sem x P = P .In x
```

For example, a value of type Sem a ⟦ b ⟧ denotes a "semantic value" in the interpretation ⟦ b ⟧ indexed by the input type a. When the input is irrelevant, we simply skip mentioning it and say "value in the interpretation".

A BCC term is interpreted as a *natural transformation* between presheaves, which is defined as follows.

```
_↠_ : Pre → Pre → Set
A ↠ B = {i : Ty} → Sem i A → Sem i B
```

Intuitively, this function maps semantic values in A to semantic values in B (for the same input type i).

6.2 NbE for CCC Fragment

NbE for the fragment of BCC which excludes the empty and sum types, namely the CCC fragment, is rather simple—let us implement this first in this section. The presheaves defined in the previous section allow us to address the issue from earlier for interpreting the type base. The interpretation for types in the CCC fragment is defined as follows.

```
⟦_⟧ : Ty ⟶ Pre
⟦ 1 ⟧       = 1'
⟦ base ⟧    = Nf' base
⟦ a ∗ b ⟧   = ⟦ a ⟧ ∗' ⟦ b ⟧
⟦ a ⇒ b ⟧ = ⟦ a ⟧ ⇒' ⟦ b ⟧
```

The unit, product and exponential types are simply interpreted as their presheaf counterparts. The base type, on the other hand, is interpreted as the presheaf of normal forms indexed by base. This is because the definition of BCC has no combinators specifically for base types, which means that a term BCC i base must depend on its input for producing a base value. Hence, we interpret it as a family of normal forms which return base for any input i—which is precisely the definition of the Nf' base presheaf. Note that this interpretation of base types is fairly standard [Lindley 2005].

Having defined the interpretation of types, we can now define the interpretation of BCC terms, i.e., evaluation, as follows.

```
eval : BCC a b ⟶ (⟦ a ⟧ ↠ ⟦ b ⟧)
eval id        x       = x
eval (f • g) x        = eval f (eval g x)
eval unit    x         = tt
eval exl     (x1 , _) = x1
eval exr     (_ , x2) = x2
eval (pair t1 t2) x  = eval t1 x , eval t2 x
eval apply (f , x)    = f iden x
eval {a} (curry t) x = λ s y ⟶ eval t (lift ⟦ a ⟧ s x , y)
```

The function eval interprets the term id as the the identity function, term composition • as function composition, exl as the first projection, and so on for the other simple cases. Let us take a closer look at the exponential fragment.

To interpret apply for a given function f (of type Sem i ⟦$a_1 \Rightarrow a_2$⟧) and its argument x (of type Sem i ⟦ a_1 ⟧), we must return a value for its application (of type Sem i ⟦ a_2 ⟧). Recollect from the definition of the exponential presheaf that an exponential is interpreted as a generalized function for a given selection. In this case, we do not need this generality since the function and its argument are both semantic values for the same input type i. Hence, we simply use the identity selection iden : Sel i i, to obtain a suitable function which accepts the argument y .

The interpretation of a term curry t (of type BCC a ($b_1 \Rightarrow b_2$)) for a given x (of type Sem i ⟦ a ⟧) must be a function (of type Sem i_1 ⟦ $b_1 \Rightarrow b_2$ ⟧) for a given selection s (of type Sel i_1 i). We achieve this by recursively evaluating t (of type BCC ($a * b_1$) b_2), with a pair of arguments (of type Sem i_1 ⟦ a ⟧ and Sem i_1 ⟦ b_1 ⟧). For the first component, we could use x, but since it is a semantic value for the input i instead of i_1, we must first lift it to i_1 using the selection s—which explains the occurrence of lift.

To implement the reification function reify : (⟦ a ⟧ ↠ ⟦ b ⟧) ⟶ Nf a b, we need two natural transformations: reflect : Ne' a ↠ ⟦ a ⟧ and reifyVal : ⟦ b ⟧ ↠ Nf' b. The former converts a neutral to a semantic value, and the latter extracts a normal form from the semantic value. Using these functions, we can implement reification as follows.

```
reify : (⟦ a ⟧ ↠ ⟦ b ⟧) ⟶ Nf a b
reify {a} f = let y = reflect {a} (sel iden)
               in reifyVal ( f y )
```

The main idea here is the use of reflection to produce a value of type Sem a a. This value enables us to apply the function f to produce a result of type Sem a ⟦ b ⟧. The resulting value is then used to apply reifyVal and return a normal form of type Nf a b.

The natural transformations used in reification are implemented as follows.

```
reflect : {a : Ty} ⟶ Ne' a ↠ ⟦ a ⟧
reflect {1}          x = tt
reflect {base}       x = ne-b x
reflect {a1 ∗ a2}   x = reflect {a1} (fst x) , reflect {a2} (snd x)
reflect {a1 ⇒ a2} x = λ s y ⟶
  reflect {a2} (app (liftNe s x) (reifyVal y))
```

```
reifyVal : {b : Ty} ⟶ ⟦ b ⟧ ↠ Nf' b
reifyVal {1}          x = unit
reifyVal {base}       x = x
reifyVal {b1 ∗ b2}   x = pair (reifyVal (proj1 x)) (reifyVal (proj2 x))
reifyVal {b1 ⇒ b2} x =
  curry (reifyVal (x (drop iden) (reflect {b1} (snd (sel iden)))))
```

Reflection is implemented by performing a type-directed translation of neutral forms to semantic values. For example, in the product case, a pair is constructed by recursively reflecting the components of the neutral. For the exponential case, the reflection of a neutral x must return a function which accepts a selection s, an argument y, and returns a semantic value for the application of the neutral x with the argument y. In other words, the body of the function needs to be constructed somehow by applying x (a neutral function) with argument y (a semantic value). The neutral application constructor app has two requirements: the function and the argument must accept the same input, and the argument must be in normal form. To satisfy the first requirement, we lift the neutral x using the selection s, and for the latter requirement we reify the argument value y. Finally, we reflect the neutral application to produce the desired semantic value.

The implementation of the function reifyVal is similar to reflection, but performs the dual action: producing a normal form from a semantic value. Like reification, we implement this by type-directed translation of semantic values to normal forms. Notice that the case of base type is trivial for both functions. This is because we defined the interpretation of base types as normal forms (Nf' base), and a semantic value is already in normal form. Hence, reifyVal simply returns the semantic value, and reflection applies ne-b on the neutral to construct a normal form.

6.3 NbE for Sums and Empty Type

Let us suppose that we interpret **0** as ⟦ **0** ⟧ = **0**'. Now consider extending the implementation of reflection for the following case:

```
reflect {0} y = ??
```

How should we handle this case? The types tell us that we need to construct a semantic value of the type ⊥ (recollect the definition of **0**'). Since ⊥ is an empty type, this is an impossible task! A similar problem arises for sums when we interpret them as ⟦ a + b ⟧ = ⟦ a ⟧ +' ⟦ b ⟧. Reflection requires us to make a choice over a returning a semantic value of ⟦ a_1 ⟧ or ⟦ a_2 ⟧. Which is the right choice? Unfortunately, we cannot make a decision with the given information since it could be either of the cases.

We cannot construct the impossible or decide over the component of a sum to reflect, hence we will simply build up a tree of decisions that we do not wish to make. A decision tree is defined inductively by the following data type:

```
data Tree (i : Ty) (P : Pre) : Set where
  leaf    : Sem i P → Tree i P
  dead    : Ne i 0 → Tree i P
  branch : Ne i (a + b) → Tree (i ∗ a) P → Tree (i ∗ b) P → Tree i P
```

A leaf in a decision tree can be leaf, in which case it contains a semantic value in P. Alternatively, a leaf can also be dead, in which case it contains a neutral which returns **0**. A branch of the tree is constructed by branch, and represents the choice over a neutral form which returns a coproduct.

Intuitively, a tree represents a suspended computation for a value in the interpretation P. For example, Tree i **0**' represents a suspended computation for a value in Sem i **0**'—which is ⊥. Since values of this type are impossible, all the leaves of such a tree must be dead. Similarly, a tree Tree i ⟦ a + b ⟧ represents a suspended computation for a value of type Sem i ⟦ a + b ⟧—which is a sum of Sem i ⟦ a ⟧ and Sem i ⟦ b ⟧.

Trees define a monad Tree' on presheaves as follows.

```
Tree' : Pre → Pre
(Tree' A) .In i = Tree i A
(Tree' a) .lift = liftTree
```

The function liftTree is defined by induction on the tree. The standard monadic operations return, map and join are defined by the following natural transformations:

```
return : P ↠ Tree' P
join : Tree' (Tree' P) ↠ Tree' P
map : (P ↠ Q) → Tree' P ↠ Tree' Q
```

The natural transformation return is defined as leaf, while join and map can be defined by straight-forward induction on the tree. The monadic structure of trees are precisely the reason they allow us to represent suspended computation.

With the tree monad, we can now complete the interpretation of types **0** and + as follows.

```
⟦ 0 ⟧      = Tree' 0'
⟦ a + b ⟧ = Tree' (⟦ a ⟧ +' ⟦ b ⟧)
```

By interpreting the empty and sum types in the Tree' monad, we are able to handle the problematic cases of reflection by returning a value in the monad, as follows.

```
reflect {0}       x = dead x
reflect {a + b} x = branch x
                      (leaf (inj₁ (reflect {a} (snd (sel iden)))))
                      (leaf (inj₂ (reflect {b} (snd (sel iden)))))
```

In addition to general monadic operations, the monad Tree' also supports the following special "run" operations:

```
runTree : Tree' ⟦ a ⟧ ↠ ⟦ a ⟧
runTreeNf : Tree' (Nf' a) ↠ Nf' a
```

These natural transformations allow us to run a monadic value to produce a regular semantic value, and are required to implement eval and reifyVal. The implementation of these natural transformations is mostly mechanical: runTreeNf can be defined by induction on the tree, and runTree can be defined by induction on the type a using an "applicative functor" map Tree c ⟦ a ⇒ b ⟧ ⟶ Tree c ⟦ a ⟧ ⟶ Tree c ⟦ b ⟧ for the exponential case.

The remaining cases of evaluation are implemented as follows.

```
eval              inl          x = return (inj₁ x)
eval              inr          x = return (inj₂ x)
eval {0}       {b} init        x = runTree {b} (map cast x)
eval {a + b} {c} (match f g) x = runTree {c} (map match' x)
  where
    match' : (⟦ a ⟧ +' ⟦ b ⟧) ↠ ⟦ c ⟧
    match' (inj₁ y) = eval f y
    match' (inj₂ y) = eval g y
```

For the case of inl, we have a semantic value x in the interpretation ⟦ a ⟧, and we need a monadic value Tree' (⟦ a ⟧ +' ⟦ b ⟧). To achieve this, we simply return the value in the monad by applying the injection inj$_1$. The case of inr is very similar.

For the case of init, we have a value x in the interpretation Tree' **0**', and we need a value in ⟦ b ⟧. Since x is a tree, we can map over it using a function cast : **0**' ↠ ⟦ b ⟧ to get a value in Tree' ⟦ b ⟧. The resulting tree can then be run using runTree to return the desired result in ⟦ b ⟧. The function cast has a trivial implementation with an empty body since a value in the interpretation by **0**' has type ⊥. The implementation of match is also similar, and we use a natural transformation match' instead of cast to map over x.

The implementation of reification for the remaining fragment resembles evaluation:

```
reifyVal {0}       x = runTreeNf (map cast x)
reifyVal {a + b} x = runTreeNf (map matchNf x)
  where
    matchNf : (⟦ a ⟧ +' ⟦ b ⟧) ↠ Nf' (a + b)
    matchNf (inj₁ y) = left (reifyVal y)
    matchNf (inj₂ y) = right (reifyVal y)
```

We use the natural transformation runTreeNf instead of runTree and matchNf instead of match.

7 CORRECTNESS OF NORMAL FORMS

A normal form is *correct* if it is convertible to the original term when quoted. The construction of the proof for this theorem is strikingly similar to the implementation of normalization. Although the details of the proof are equally interesting, we will only discuss the required definitions and sketch the proof of the main theorems to keep this section concise. We encourage the curious reader to see the implementation of the full proof for further details (see A.1 for link). We will prove the correctness of normalization by showing that evaluation and reification are correct. To enable the definition of correctness for these functions, we must first relate terms and semantic values using logical relations.

7.1 Kripke Logical Relations

We will prove the correctness theorem using Kripke logical relations à la Coquand [1993]. In this section, we define these logical relations.

Definition 7.1 (Logical relation R)**.** A relation R between terms and semantic values, indexed by a type b, is defined by induction on b:

```
R : {b : Ty} → BCC a b → Sem a ⟦ b ⟧ → Set
R {1}        t v = ⊤
R {base}     t v = t ≈ q v
R {b₁ * b₂}  t v = R (exl ● t) (proj₁ v) × R (exr ● t) (proj₂ v)
R {b₁ ⇒ b₂}  t v = (s : Sel c _)
  → R t' x → R (apply ● pair (liftBCC s t) t') (v s x)
R {0}        t v = R₀ t v
R {b + c}    t v = R₊ t v
```

Intuitively, the relation R establishes a notion of equivalence between terms and semantic values, but we will say *related* instead of equivalent to be pedantic. For example, for the case of products, it states that composing the combinator exl with a term is related to applying the projection proj_1 on a value—and similarly for exr and proj_2. In the unit case, it states that terms and values are trivially related. For base types, it states that terms must be convertible to the quotation of values, since values are normal forms by definition of ⟦_⟧. For the case of exponentials, the definition states that t, which returns an exponential, is related to a functional value v, if for all related "arguments" t' and x, the resulting values of the application are related. As usual, since v is a function generalized over selections, the relation also states that it must hold for all appropriate selections.

For the case of empty and sum types, we need a relation between terms and trees—which is defined by Rt as follows.

Definition 7.2 (Logical relation Rt)**.** A relation Rt between terms and trees, indexed by another relation *Rl* between terms and values in the leaves, is defined by induction on the tree:

```
Rt : (Rl : BCC a₁ b → Sem a₁ B' → Set)
  → BCC a b → Tree a B' → Set
Rt Rl t (leaf a)   = Rl t a
Rt Rl t (dead x) = t ≈ init ● qNe x
Rt Rl t (branch x v₁ v₂) = ∃₂ λ t₁ t₂
  → (Rt Rl t₁ v₁) × (Rt Rl t₂ v₂) × (t ≈ Case (qNe x) t₁ t₂)
```

Intuitively, the relation Rt states that a term is related to a tree if the term is related to the values in the leaves. The key idea in the definition of Rt for the leaf case is to parameterize the definition by a relation *Rl* between terms and leaf values. Note that the relation R cannot be used here (instead of a parameterized relation *Rl*) since its type is more specific than the relation needed for leaves. For the case of dead leaves with a neutral returning **0**, the definition states that the t must be convertible to elimination of **0** using init. In the branch case, it states the inductive step: t is related to a decision branch in the tree, if t is convertible to a decision over the neutral x (implemented by Case) for some t_1 and t_2 related to subtrees v_1 and v_2.

Using the relation Rt, we can now define the remaining relations for the empty and sum types as follows.

Definition 7.3 (Logical relations R_0 and R_+)**.** Logical relations R_0 and R_+ are defined as special cases of Rt using the below defined relations Rl_0 and Rl_+ respectively:

```
Rl₀ : BCC a 0 → Sem a 0' → Set
Rl₀ _ ()

R₀ : BCC a 0 → Tree a 0' → Set
R₀ t v = Rt Rl₀ t v

Rl₊ : BCC a (b + c) → Sem a (⟦ b ⟧ +' ⟦ c ⟧) → Set
Rl₊ t (inj₁ x) = ∃ λ t' → R t' x × (inl ● t' ≈ t)
Rl₊ t (inj₂ y) = ∃ λ t' → R t' y × (inr ● t' ≈ t)

R₊ : BCC a (b + c) → Tree a (⟦ b ⟧ +' ⟦ c ⟧) → Set
R₊ t c = Rt Rl₊ t c
```

The relation Rl_0 is simply a type cast since a value of type Sem *a* **0**' does not exist. On the other hand, the relation Rl_+, states that t is related to an injection $\text{inj}_1\ x$, if t is convertible to inl ● t' for some t' related to x—and similarly for inj_2 and inr.

7.2 Proof of Correctness

We prove the main correctness theorem (Theorem 7.3) using two intermediate theorems, namely the *fundamental theorem* of logical relations (Theorem 7.1) and the correctness of reification (Theorem 7.2), and various lemmata. In all the cases, we either perform induction on the return type of a term or on a tree. The main idea here is that the appropriate induction triggers the definition of the relations, hence enabling Agda to refine the proof goal for a specific case.

Lemma 7.1 (Invariance under conversion)**.** *If a term t is convertible to t' and t' is related to a semantic value v, then t is related to v.*

```
invariance : t ≈ t' → R t' v → R t v
```

Proof. By induction on the return type of t and t'. The proof is fairly straight-forward equational reasoning using the conversion rules (≈). The empty and sum types can be handled by induction on the tree. □

Lemma 7.2 (Lifting preserves relations)**.** If a term $t : \mathrm{BCC}\ a\ b$ is related to a value $v : \mathrm{Sem}\ a\ [\![\ b\ ]\!]$, then lifting the term is related to lifting the value, for any applicable selection s.

```
liftPresR : R t v ⟶ R (liftBCC s t) (lift ⟦ b ⟧ s v)
```

Proof. By induction on the return type of t. As in the previous lemma, the empty and sum types can be handled by induction on the tree. □

Definition 7.4 (Fundamental theorem)**.** If a term t' is related to a semantic value v, then the composition $t \bullet t'$ is related to the evaluation of t with the input v, for all terms t. That is, the fundamental theorem holds if Fund t (defined below) holds for all t.

```
Fund : (t : BCC a b) ⟶ Set
Fund {a} {b} t = {c : Ty} {t' : BCC c a} {v : Sem c ⟦ a ⟧}
        ⟶ R t' v ⟶ R (t • t') (eval t v)
```

Theorem 7.1 (Correctness of evaluation)**.** The fundamental theorem holds, or equivalently, evaluation is correct.

```
correctEval : (t : BCC a b) ⟶ Fund t
```

Proof. By induction on the term t. Most cases are proved by the induction hypothesis and some equational reasoning . To enable equational reasoning, we must use the invariance lemma (Lemma 7.1). For the case of curry, the key step is to make use of the β rule for functions (from Section 2).

For the sum and empty types, recall that evaluation uses the natural transformation runTree : Tree' ⟦ a ⟧ ↠ ⟦ a ⟧. Hence, to prove correctness of evaluation for these cases, we need a lemma correctRunTree : Rt R t v ⟶ R t v—which can be proved by induction on the return type of t. The proof of this lemma also requires us to prove correctness of all the natural transformations used by runTree, which can be achieved in similar fashion to correctRunTree. Note that we must use the lifting preservation lemma (Lemma 7.2), wherever lifting is involved, for example, in the curry case. □

Lemma 7.3 (Correctness of reflect and reifyVal)**.** i) The quotation of a neutral form n is related to its reflection. ii) If a term t is related to a value v, then t must be convertible to the normal form which results from the quotation of reification of v.

```
correctReflect : {n : Ne a b} ⟶ R (qNe n) (reflect n)
correctReifyVal : R t v ⟶ t ≈ q (reifyVal v)
```

Proof. Implemented mutually by induction on the return type of the neutral / term and using the invariance lemma (Lemma 7.1) to do equational reasoning. Appropriate eta conversion rules are needed for products, exponentials and sums. □

Theorem 7.2 (Correctness of reification)**.** The fundamental theorem proves that t is convertible to quotation of the value obtained by evaluating and reifying t.

```
correctReify : (Fund t) ⟶ t ≈ q (reify (eval t))
```

Proof. By induction on the return type of term t. This theorem follows from Lemma 7.3 and the other lemmata discussed above. □

Theorem 7.3 (**Correctness of normal forms**)**.** A term is convertible to the quotation of its normal form.

Proof. Since normalization is defined as the composition of reification and evaluation, the correctness of normal forms follows from the correctness of reification and evaluation:

```
correctNf : (t : BCC a b) ⟶ t ≈ q (norm t)
correctNf t = correctReify (correctEval t)
```

□

7.3 Exponential Elimination Theorem

Using the syntactic elimination of exponentials illustrated earlier using normal forms (Section 5.2), and the normalization procedure which converts BCC terms to normal forms (Section 5.2), we finally have the following exponential elimination theorem for BCC terms.

Theorem 7.4 (Exponential elimination)**.** Given that a and b are first-order types, every term $f : \mathrm{BCC}\ a\ b$ can be converted to an equivalent term $f' : \mathrm{BCC}\ a\ b$ which does not use any exponentials.

Proof. From the normalization function norm implemented in Section 6, and the correctness of normal forms by Theorem 7.3, we know that there exists a normal form $n : \mathrm{Nf}\ a\ b$ resulting from the application norm f such that $f \approx q\ n$. Since a and b are first-order types, we have a DBC term qD n : DBC a b which does not use exponentials. However, since the function qD is syntactically identical to the quotation function q without the exponentials, it must be he case that q n : BCC a b does not use any exponentials. Thus we have an equivalent term f' = q n. □

8 SIMPLICITY, AN APPLICATION

Simplicity is a typed combinator language for programming smart contracts in blockchain applications [O'Connor 2017]. It was designed as an alternative to Bitcoin Script, especially to enable static analysis and estimation of execution costs. The original design of Simplicity only allows unit, product and sum types. It does not allow exponentials, the empty type or base types. The simple nature of these types enables calculation of upper bounds on the time and memory requirements of executing a Simplicity program in an appropriate execution model. For example, the bit-size of a value is computed using its type as follows.

```
size 1         = 0
size (t₁ * t₂) = size t₂ ‘+ size t₂
size (t₁ + t₂) = 1 ‘+ max (size t₁) (size t₂)
```

Note that the operator ‘+ is simply addition for natural numbers renamed to avoid name clash with the constructor +. The additional bit is need in the sum case to represent the appropriate injection.

Despite Simplicity's ability to express any finite computation between the allowed types, its low-level nature makes it cumbersome to actually write programs since it lacks common programming abstractions such as functions and loops. Even as a compilation target, Simplicity is too low-level. For example, compiling functional programs to Simplicity burdens the compiler with the task of defunctionalization since Simplicity does not have a corresponding notion of functions. To solve this issue, Valliappan et al. [2018] note that Simplicity can be modeled in (distributive) bicartesian categories, and propose extending Simplicity with exponentials, and hence to bicartesian closed categories without the empty type.

Although extending Simplicity with exponentials makes it more expressive, it complicates matters for static analysis. For example, the extension of the size function is already a matter of concern:

```
size (t₁ ⇒ t₂) = size t₂ ^ size t₁
```

Valliappan et al. [2018] avoid this problem by extending the bit machine with the ability to implement closures, but the problem of computing an upper bound on execution time and memory consumption remains open. Exponential elimination provides a solution for this: Simplicity programs with exponentials can be compiled by eliminating exponentials to programs without exponentials, hence providing a more expressive higher-order target language—while also retaining the original properties of static analysis.

Since Simplicity resembles BCC and DBC combinators, they can be translated to BCC, and from DBC in a straight-forward manner [Valliappan et al. 2018]:

```
SimplToBCC : Simpl a b → BCC a b
DBCToSimpl : DBC a b → Simpl a b
```

Exponential elimination bridges the gap between BCC and DBC terms:

```
elimExp : firstOrd a → firstOrd b → BCC a b → DBC a b
elimExp p q t = qD p q (norm t)
```

Thus, we can implement an exponential elimination algorithm for Simplicity programs:

```
elimExpS : firstOrd a → firstOrd b → Simpl a b → Simpl a b
elimExpS p q t = DBCToSimpl (elimExp p q (SimplToBCC t))
```

The difference between the input and output programs is of course that the input may have exponentials, but the output *will not*. The requirements that the input and output of the entire program be first-order types is a harmless one since such programs must have an observable input and output anyway.

Note that we have overlooked the empty type and the combinator init in the translation of DBCToSimpl here. However, this can be mitigated easily by adding an additional predicate nonEmpty : Ty → Set to discharge this case —as in Section 5.2, thanks to the weak subformula property!

Although our work shows that it is possible to eliminate exponentials from Simplicity programs, the implementation provided here might not be the most practical one. Normal forms are in η-expanded form, which means that the generated programs may be much larger than necessary, hence leading to code explosion. Moreover, the translation to BCC and from DBC is also an unnecessary overhead. It may be possible to tame code explosion by normalizing without η expansion [Lindley 2005]. The latter problem, on the other hand, can be solved easily by implementing exponential elimination directly on Simplicity programs. We leave these improvements as suggestions for future work.

9 RELATED WORK

Selections resemble *weakenings* (also called *order preserving embeddings*) in lambda calculus [Altenkirch et al. 1995]. Weakenings are defined for typing contexts such that a weakening $\Gamma \sqsubseteq \Delta$ selects a "subcontext" Δ from the context Γ [McBride 2018]. Selections, on the other hand, are simply a subset of BCC terms that select components of the input. Conceptually, selections are the BCC-equivalent of weakenings and they have properties (discussed in Section 4) similar to weakenings. Most importantly, selections unify the notion of weakenings and variables—since they are used in neutrals (as "variables") and for lifting (as "weakenings").

Altenkirch et al. [2001] implement NbE to solve the decision problem for STLC with all simple types except the empty type ($\lambda_{\Rightarrow \mathbf{1} * +}$). Balat et al. [2004] solve the *extensional* normalization problem using NbE for the STLC including the empty type ($\lambda_{\Rightarrow \mathbf{1} * + \mathbf{0}}$). Abel and Sattler [2019] provide an account of NbE for $\lambda_{\Rightarrow \mathbf{1} * + \mathbf{0}}$ using decision trees—the techniques of which they go on to use for more advanced calculi. They in turn attribute the idea of decision trees for normalizing sum types to Altenkirch and Uustalu [2004]. Our interpretation model is based on that of Abel and Sattler [2019] and the generated normal forms are not unique—caused by commuting case conversions and the overlap between selections and projections. The primary difference between earlier efforts and or work is that we implement NbE for a combinator language.

Altenkirch and Uustalu [2004] also prove correctness of normal forms using logical relations, but only for closed lambda terms. Our logical relations have a much more general applicability since they are indexed by the input (or equivalently by the typing context). Moreover, we prove correctness for interpreting sums using decision trees by the means of logical relations generalized over arbitrary presheaf interpretations. Since the decision tree monad Tree' is a *strong monad* [Moggi 1991], it should be possible to further extend this proof technique to normalization of calculi with *computational effects* [Filinski 2001] [Abel and Sattler 2019].

10 FINAL REMARKS

In this paper, we have shown that BCC terms of first-order types can be normalized to terms in a sub-language without exponentials based on distributive bicartesian categories. To this extent, we have implemented normalization using normalization by evaluation, and shown that normal forms are convertible to the original term in the equational theory of BCCs. Moreover, we have also shown the applicability of our technique to erase exponentials from a combinator language called Simplicity. Our work enables a closure-free implementation of BCC combinators and answers previously open questions about the elimination of exponentials.

As noted earlier, the normal forms of BCC combinators presented here are not normal forms of the equational theory specified by the conversion relation ≈. This is because the syntax of normal forms does not enforce normal forms of equivalent terms to be unique. For example, the normal forms ne-b (sel (drop endb)) and ne-b (fst (sel (keep endb))) are syntactically different, but interconvertible when quoted. Hence, the normalization procedure does not derive the conversion relation ≈, and cannot be used to decide it. Instead, our notion of normal forms is characterized by the weak subformula property, and aimed at the eliminating intermediate values by restricting the unruly composition which allows introduction and elimination of arbitrary values.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their comments on earlier drafts of this paper. We thank Andreas Abel for suggesting NbE and for the many discussions about implementing and proving the correctness of NbE. The first author took inspiration for this work from Andreas’ lecture notes on NbE for intuitionistic propositional logic at the Initial Types Club. We would also like to thank Thierry Coquand, Fabian Ruch and Sandro Stucki for the insightful discussions on the topic of exponential elimination. This work was funded by the Swedish Foundation for Strategic Research (SSF) under the projects Octopi (Ref. RIT17-0023) and WebSec (Ref. RIT17-0011), as well as the Swedish research agency Vetenskapsrådet.

REFERENCES

Martin Abadi, Luca Cardelli, P-L Curien, and J-J Lévy. 1991. Explicit substitutions. *Journal of functional programming* 1, 4 (1991), 375–416.

Andreas Abel and Christian Sattler. 2019. Normalization by Evaluation for Call-by-Push-Value and Polarized Lambda-Calculus. *arXiv preprint arXiv:1902.06097* (2019).

Thorsten Altenkirch, Peter Dybjer, Martin Hofmann, and Phil Scott. 2001. Normalization by evaluation for typed lambda calculus with coproducts. In *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*. IEEE, 303–310.

Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. 1995. Categorical reconstruction of a reduction free normalization proof. In *International Conference on Category Theory and Computer Science*. Springer, 182–199.

Thorsten Altenkirch and Tarmo Uustalu. 2004. Normalization by evaluation for $\lambda{\to}2$. In *International Symposium on Functional and Logic Programming*. Springer, 260–275.

Steve Awodey. 2010. *Category theory*. Oxford University Press.

Vincent Balat, Roberto Di Cosmo, and Marcelo Fiore. 2004. Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums. In *POPL*, Vol. 4. 49.

Ulrich Berger, Matthias Eberl, and Helmut Schwichtenberg. 1998. Normalization by evaluation. In *Prospects for Hardware Foundations*. Springer, 117–137.

Ulrich Berger and Helmut Schwichtenberg. 1991. An inverse of the evaluation functional for typed lambda-calculus. In *[1991] Proceedings Sixth Annual IEEE Symposium on Logic in Computer Science*. IEEE, 203–211.

Ana Bove, Peter Dybjer, and Ulf Norell. 2009. A brief overview of Agda–a functional language with dependent types. In *International Conference on Theorem Proving in Higher Order Logics*. Springer, 73–78.

Catarina Coquand. 1993. From semantics to rules: A machine assisted analysis. In *International Workshop on Computer Science Logic*. Springer, 91–105.

Guy Cousineau, P-L Curien, and Michel Mauny. 1987. The categorical abstract machine. *Science of computer programming* 8, 2 (1987), 173–202.

P-L Curien. 1986. Categorical combinators. *Information and Control* 69, 1-3 (1986), 188–254.

Conal Elliott. 2017. Compiling to categories. *Proceedings of the ACM on Programming Languages* 1, ICFP (2017), 27.

Andrzej Filinski. 2001. Normalization by evaluation for the computational lambda-calculus. In *International Conference on Typed Lambda Calculi and Applications*. Springer, 151–165.

Yves Lafont. 1988. The linear abstract machine. *Theoretical computer science* 59, 1-2 (1988), 157–180.

Sam Lindley. 2005. Normalisation by evaluation in the compilation of typed functional programming languages. (2005).

Saunders MacLane and Ieke Moerdijk. 1992. Sheaves in geometry and logic: a first introduction to topos theory. (1992).

Conor McBride. 2018. Everybody’s got to be somewhere. *Electronic Proceedings in Theoretical Computer Science* 275 (2018), 53–69.

John C Mitchell and Eugenio Moggi. 1991. Kripke-style models for typed lambda calculus. *Annals of Pure and Applied Logic* 51, 1-2 (1991), 99–124.

Eugenio Moggi. 1991. Notions of computation and monads. *Information and computation* 93, 1 (1991), 55–92.

Shayan Najd, Sam Lindley, Josef Svenningsson, and Philip Wadler. 2016. Everything old is new again: quoted domain-specific languages. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. ACM, 25–36.

Ulf Norell. 2007. *Towards a practical programming language based on dependent type theory*. Vol. 32. Citeseer.

Russell O’Connor. 2017. Simplicity: A new language for blockchains. In *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security*. ACM, 107–120.

John C Reynolds. 1998. Definitional interpreters for higher-order programming languages. *Higher-order and symbolic computation* 11, 4 (1998), 363–397.

Anne Sjerp Troelstra and Helmut Schwichtenberg. 2000. *Basic proof theory*. Number 43. Cambridge University Press.

Nachiappan Valliappan, Solène Mirliaz, Elisabet Lobo Vesga, and Alejandro Russo. 2018. Towards Adding Variety to Simplicity. In *International Symposium on Leveraging Applications of Formal Methods*. Springer, 414–431.

A APPENDIX

A.1 Agda Implementation

The complete Agda implementation of the normalization procedure and mechanization of the proofs in this paper can be found at the URL https://github.com/nachivpn/expelim

A.2 Implementation of distributivity in BCC

```
Distr : BCC (a ∗ (b + c)) ((a ∗ b) + (a ∗ c))
Distr = apply • (pair
  (match
    (curry (inl • pair exr exl))
    (curry (inr • pair exr exl)) • exr)
  exl)
```