Computational effects, algebraic theories and normalization by evaluation

Danel Ahman
Hughes Hall

UNIVERSITY OF
CAMBRIDGE

A dissertation submitted to the University of Cambridge in partial fulfilment of the requirements for the degree of Master of Philosophy in Advanced Computer Science

University of Cambridge
Computer Laboratory
William Gates Building
15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom

Email: danel.ahman@eesti.ee

Friday 15th June, 2012

Declaration

I Danel Ahman of Hughes Hall, being a candidate for the M.Phil in Advanced Computer Science, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.

Total word count: 14 921

Signed:

Date:

Abstract

This dissertation is concerned with modeling and reasoning about impure ML-like higher-order programs. Our work is based on the algebraic theories of computational effects proposed by Plotkin and Power. In particular, we present an extension from the algebraic value and effect theories to a fine-grained call-by-value intermediate language. Whilst this extension has a straightforward definition and is intuitively correct, the proof of its conservativeness requires extensive work. Before one is able to correctly reason about the terms in the value and effect theories and the corresponding terms in the intermediate language, it is necessary to effectively decide provable equality in the intermediate language. As a result, we spend a significant proportion of this dissertation on developing a suitable normalization by evaluation (NBE) algorithm to compute canonical normal forms in the intermediate language. The comparison of these normal forms provides us with the necessary decision procedure for proving the conservativity theorem. The NBE algorithm we define is a generalization of the usual presentations of NBE where normal forms are identified up to equality rather than modulo the given value and effect theories. However, the usual normalization results arise as special cases when the value and effect theories do not contain equations. We have also formalized the syntax of the intermediate language together with the formally verified NBE algorithm in the interactive theorem prover and functional programming language Agda.

Acknowledgments

I would like to thank my project supervisor Dr. Sam Staton for his guidance and helpful feedback. I am also grateful to Prof. Andrew M. Pitts for interesting and fruitful discussions about normalization by evaluation. I would also like to thank the staff of the Computer Laboratory for providing me with a desk and a computer and, altogether, with a wonderful working environment.

Contents

Chapter 1

Introduction

Nowadays, an increasing amount of our everyday lives is governed by computers and computer programs. It is therefore important to realize that these programs do not run in isolation but instead interact with other programs, with their environment and with the human users. As a result, the behavior of such programs is usually complex and tends to be erroneous. Therefore, it is essential to develop rigorous mathematical tools to reason about *computational effects* these programs have on their environment. Some notable examples are reading from and writing to memory, throwing and catching exceptions, performing input/output operations, deterministic and non-deterministic control flow.

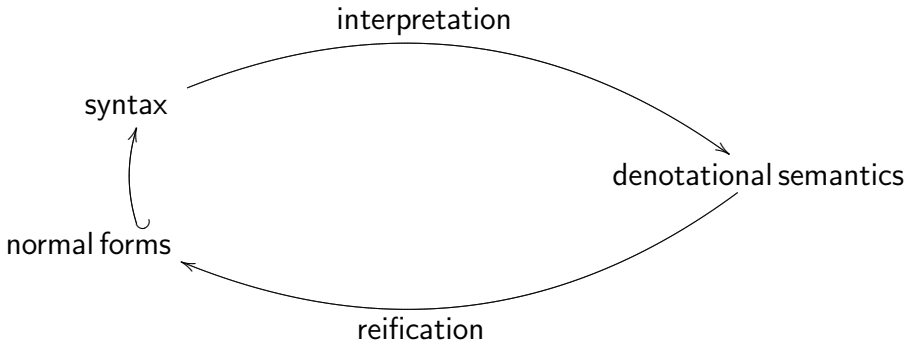
The presence of such computational effects is especially subtle in the realm of functional programming languages where, historically, programs have been modeled as pure mathematical functions. For example, consider the following program written in ML-like syntax

```
function read_and_write f loc1 loc2 =
  let a = (read loc1) in
    let b = (f a) in
      let c = (write loc2 zero) in
        return a
```

which cannot be modeled as a pure function as it performs reading from and writing to global memory. However, the last two decades have seen immense achievements in developing mathematical theories to accommodate the impurity arising from computational effects in functional programming languages. In his seminal work, Moggi [28, 29] demonstrated how monads describe computational effects and the mathematics behind them. Later, Plotkin and Power [36, 37] discovered that there is a more natural way of identifying computational effects, namely, algebraic theories. They also showed that these algebraic theories of computational effects give rise to the monads Moggi proposed.

The specific notion of algebraic theories of computational effects we investigate in this dissertation, called value and effect theories, were proposed by Plotkin and Pretnar [33]. We discuss how these theories can be extended to an intermediate language suitable for giving a mathematical account of full ML-like impure programs, not just the individual effects. The intermediate language that we investigate is a fine-grained call-by-value language first introduced by Levy, Power and Thielecke [21]. It has a syntax-level separation between values and computations and, therefore, is suitable for extending both value and effect theories.

Whilst the extension itself may seem straightforward, one still needs to formally prove that it is conservative. This means that the extension must preserve the behavior determined by the algebraic theory while not introducing any new impure behavior previously not stated in the algebraic theory. To prove the conservativity theorem, one first needs to effectively decide provable equality in the intermediate language. For this reason, we spend a considerable proportion of this dissertation on defining a normalization algorithm. We use a semantic reduction-free normalization method, called normalization by evaluation (NBE) [7, 8], that computes normal forms (i.e., a subset of terms that are canonical representatives of provably equal terms) by inverting the interpretation of syntax into a suitable denotational semantics.



The NBE algorithm we define is a generalization of the usual presentations of NBE (e.g., [8, 14]) allowing us to normalize the terms modulo the given value and effect theories. We have developed a formally verified implementation [1] of this algorithm in a functional programming language and interactive theorem prover Agda. We chose to work in Agda because (i) an interactive theorem prover provides a convenient working environment, (ii) Agda's strong type theory gives an additional correctness guarantee to our work and (iii) the end-result is a functional program. To our knowledge, this is the first formalization of NBE for the fine-grained call-by-value intermediate language.

The denotational semantics we define follows the usual categorical approach [32], namely, given a suitable category, we interpret types as its objects and programs as its morphisms.

We have chosen to work in the language of category theory [23] rather than conventional set theory to develop this dissertation in a uniform algebraic setting. The particular category we work with is the category of covariant presheaves $\mathbf{Set}^{\mathbf{Ctx}}$ (i.e., a category of functors from the category **Ctx** of typing contexts to the category **Set** of sets and functions). It has been shown [14, 15, 39] that $\mathbf{Set}^{\mathbf{Ctx}}$ provides appropriate algebraic structure for characterizing abstract syntax with variable binding and defining NBE.

1.1 Contributions

Our work is based on the previous developments in the areas of using semantic methods of normalization, e.g., NBE, and modeling computational effects with monads and algebraic theories. Our own contributions in this dissertation are

- an Agda formalization of the intermediate language,
- an Agda formalization of the value and effect theories we consider together with an extension to the intermediate language,
- an Agda formalization of the NBE algorithm we define for the intermediate language together with its correctness proofs,
- a proof of the conservativity theorem for the extension to the intermediate language.

1.2 Structure

In Chapter 2, we give a brief overview of modeling and reasoning about computational effects. We also survey some different presentations of NBE and give a concise overview of the meta-language Agda. We continue by defining the fine-grained call-by-value intermediate language in Chapter 3 and the value and effect theories together with the extension to this language in Chapter 4. The main contributions of this dissertation are presented in Chapter 5 and 6. Chapter 5 contains the NBE algorithm together with its correctness proofs. Chapter 6 is devoted to proving the conservativity of the extension of value and effect theories.

Chapter 2

Background

We now elaborate more on the background of our work. We begin by outlining previous developments in giving a mathematical account of computational effects using monads and algebraic theories. We also describe the fundamental ideas behind NBE and briefly overview the meta-language Agda we use to formalize the theory we present.

2.1 Impure programs and computational effects

Our motivation is to investigate how to reason about ML-like impure functional programs. Recall the example program from Introduction that reads from and writes to global memory.

```
function read_and_write f loc1 loc2 =
  let a = (read loc1) in
    let b = (f a) in
      let c = (write loc2 zero) in
        return a
```

This program exhibits many of the intriguing features we are interested in. First, this program causes computational effects by interacting with its environment, namely, the global memory. Therefore, it cannot be modeled as a pure mathematical function because this impure behavior also needs to be taken into account. Second, it is worth noticing that this program takes a higher-order argument f of function type. As a result, it is possible to apply f to the value a and possibly cause additional, currently unknown, computational effects. This becomes especially important when manipulating program code, e.g., finding normal forms of given programs to decide their intensional equality. Therefore, it is important to develop mathematically rigorous methods for reasoning about the behavior

of such programs. However, before modeling full programs, we first discuss two approaches to developing suitable mathematical theory to capture computational effects.

2.2 Modeling computational effects

In his seminal work [28, 29], Moggi introduced the idea of using monads, more precisely strong monads, on cartesian-closed categories to give a categorical model of computational effects which included global and local store, exceptions, input/output, control, non-determinism and continuations. Moggi's work builds on the usual categorical approach for giving denotational semantics to programming languages [32]. For example, if we would work in the category **Set** whose objects are sets and morphisms are functions, we would interpret types as sets of values and programs as functions from values to values.

Following Moggi [28], we define monads as Kleisli triples and illustrate how they are used to give a mathematical account of the aforementioned impure behavior.

Definition 2.2.1. A *Kleisli triple* $(T, \eta, \_^*)$ on a category $\mathcal{C}$ is given by a mapping $T : ob(\mathcal{C}) \to ob(\mathcal{C})$ of objects of $\mathcal{C}$ together with

- the unit $\eta_X : X \to TX$,
- given $f : X \to TY$, the Kleisli extension $f^* : TX \to TY$,
- Kleisli exponentials $\langle (X \to TY)\,,\, \varepsilon_{X,Y} \colon ((X \to TY) \times X) \to TY \rangle$ for every pair of objects $\langle X, Y \rangle$,

satisfying

- $\eta_X^* = \mathrm{id}_{TX}$,
- $f^* \circ \eta = f$,
- $g^* \circ f^* = (g^* \circ f)^*$,

with the equivalent monad structure defined by

- $Tf = (\eta_Y \circ f)^*$ given $f : X \to Y$,
- $\mu_X = \mathrm{id}_{TX}^*$.

This definition illustrates how monads give a semantic account of computations. In categorical semantics, we can think of such computations as morphisms of type $f : X \to TY$ that, informally, are functions that can cause computational effects while computing their values. The effects are captured by the monad T wrapped around the return values Y. For example, *non-determinism* is modeled as a monad given by the covariant powerset

functor $T(X) = \mathcal{P}(X)$, singleton sets $\eta_X(x) = \{x\}$ as unit and big union $\mu_X(X) = \cup X$ as multiplication. We also see that the Kleisli extension operation enables us to construct functions from computations to computations and, therefore, is an ideal candidate for modeling the sequence of multiple impure computations.

Such semantic presentations of computational effects have also found their way into the syntax of functional programming languages. For example, Haskell [31] uses Kleisli triples in its syntax with the operations called *return* and *bind* (also expressed with the shorthand *do-notation*). Moreover, the syntax of the example ML-like program we presented above is also based on Kleisli triples with the operations written *return* and *let*.

Together with this semantic approach, Moggi [28] also presented a formal calculus, called the computational lambda calculus (λ_c-calculus), to reason about impure programs. This language is a coarse-grained version of the intermediate language we use. However, it turned out that the definition above is not sufficient to give a model of the λ_c-calculus (i.e., a monad model). As a result, Moggi identified that one needs the monad to be strong to transform pairs of values and computations (or computations and computations) to computations whose values are pairs.

Definition 2.2.2. A *strength* of a monad on a cartesian-closed category is a natural transformation

- $t_{X,Y} : X \times TY \to T(X \times Y)$

satisfying

- $T(\lambda_X) \circ t_{1,X} = \lambda_{TX}$,
- $T(\alpha_{X,Y,Z}) \circ t_{X\times Y,Z} = t_{X,Y\times Z} \circ (id_X \times t_{Y,Z}) \circ \alpha_{X,Y,TZ}$,
- $t_{X,Y} \circ (id_X \times_Y) = \eta_{X\times Y}$,
- $t_{X,Y} \circ (id_X \times \mu_Y) = \mu_{X\times Y} \circ T(t_{X,Y}) \circ t_{X,TY}$,

where λ and α are the following isomorphisms arising from the product-monoidal structure of $\mathcal{C}$

- $\lambda_X : 1 \times X \to X$,
- $\alpha_{X,Y,Z} : (X \times Y) \times Z \to X \times (Y \times Z)$.

A decade after Moggi's work, Plotkin and Power [36, 37] observed that computational effects actually *determine* monads rather than are *identified* by them. They showed that identifying computational effects with operations and equations gives a mathematical theory closely corresponding to their computational nature. The monads Moggi proposed can then be recovered as the corresponding free algebra monads. This notion of *algebraic*

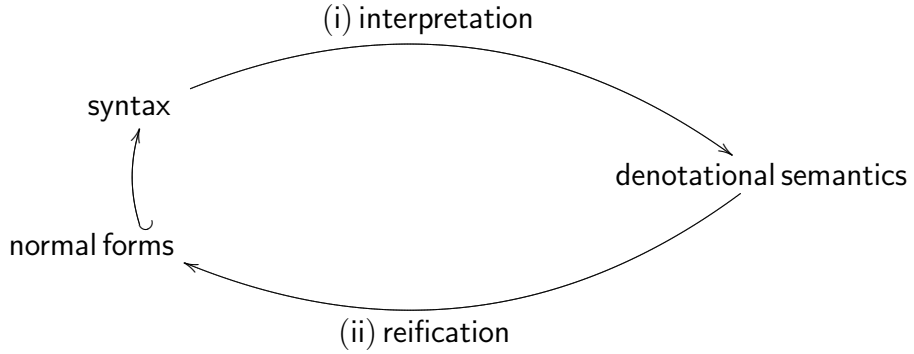
effects captures all the aforementioned computational effects with the exception of continuations that are of different character [36]. More recently, it has been investigated how tensor products and sums can be used to combine algebraic effects [18] and how they can be combined with continuations [17].

We make the notion of algebraic *effect theories* explicit in Chapter 4 together with an extension to the intermediate language suitable for reasoning about ML-like programs. In this section, we limit ourselves to a more informal description by illustrating the theory of non-determinism that is defined as a signature consisting of one binary operation symbol **or** together with the equations of the theory of semilattices. These equations require **or** to be idempotent $x \,\mathbf{or}\, x = x$, commutative $x \,\mathbf{or}\, y = y \,\mathbf{or}\, x$ and associative $(x \,\mathbf{or}\, y) \,\mathbf{or}\, z = x \,\mathbf{or}\, (y \,\mathbf{or}\, z)$.

2.3 Normalization by evaluation

Normalization is often used to simplify equational reasoning by reducing the question of equality to the comparison of normal forms. Normalization by evaluation (NBE), also known as reduction-free normalization, first appeared in the work of Martin-Löf [25] for intuitionistic type theory and was later made precise by Berger and Schwichtenberg [8] for the simply typed lambda calculus. Multiple authors have later extended NBE to more advanced type systems and given it more abstract characterizations. For example, Berger and Schwichtenberg themselves extended it to simply typed lambda calculus with products and constants [7] while Altenkirch and Uustalu [5] showed how to capture lambda calculus with booleans. Another extension was given by Filinski [13] who successfully applied NBE to call-by-name and call-by-value paradigms. A category-theoretical account of NBE was first suggested by Altenkirch, Hofmann and Streicher [4] and was later extended to simply typed lambda calculus with binary coproducts by Altenkirch, Dybjer, Hofmann and Scott [2]. The NBE algorithm we develop in this dissertation has been most influenced by these categorical approaches together with the thorough semantic analysis carried out by Fiore [14] together with Balat and Di Cosmo [6].

NBE is based on (i) defining an interpretation map taking syntax to a suitable denotational semantics and (ii) constructing an inverse, called reification, of this map that extracts syntactic normal forms from the given semantics. This rather elegant idea is summarized on the following diagram from Introduction



where the normalization algorithm is given by a straightforward composition of the interpretation and reification maps.

$$\mathsf{nf} = \mathsf{reification} \circ \mathsf{interpretation}$$

Various authors have considered different, suitably modified, denotational semantics, e.g., complete partial orders [13], Scott-domains [7], sets [5], presheaf categories [2, 4, 14] and categories of Grothendieck relations [6]. The unifying theme in these different models is their intensional nature one needs to construct the reification map. In particular, all of the aforementioned approaches interpret base types as objects carrying syntactic structure.

The correctness of NBE algorithms is usually expressed as the following conditions.

(i) The normalization algorithm preserves normal forms.

(ii) Every term is provably equal to its normal form.

(iii) Provably equal terms have equal normal forms.

These correctness results, especially (iii), rely on the soundness of interpretation, i.e., that the interpretations of provably equal terms are equal. We will make these rather informal definitions and correctness results precise when we present our NBE algorithm in Chapter 5.

2.4 Programming and theorem proving in Agda

One of our contributions is the complete verified formalizations [1] of the intermediate language and our NBE algorithm. This formalization was carried out in the interactive theorem prover and functional programming language Agda which is based on Martin-Löf's intuitionistic type theory using the Curry-Howard isomorphism and identifying propositions as types and proofs as programs.

The definitions and theorems we present in this dissertation are given in syntax very similar to Agda. Therefore, we illustrate programming and theorem proving in Agda with a central example of the *heterogeneous equality* (also known as John Major equality) we use as propositional equality in our formalization. It first appeared in McBride's PhD thesis [26] and allows us to form equations between terms of different types. However, two terms can only be considered actually equal (and hence the equation is provable) if they are of same type. Therefore, heterogeneous equality enables us to avoid proving the equality of types in function signatures and incorporate these proofs into proof terms. We define heterogeneous equality as an inductive data type in Agda with one unique constructor refl.

```
data _≅_ {A : Set} (a : A) : {A' : Set} → A' → Set where
  refl : a ≅ a
```

We use Agda's mixfix syntax _≅_ to follow the usual presentation of equality, e.g., x ≅ y. The curly braces denote implicit arguments which can be inferred by Agda and omitted in definitions. In addition, this definition tells us that refl is the only canonical inhabitant of the equality type _≅_. However, _≅_ also has the usual properties of symmetry and transitivity.

```
sym : {A : Set} → {a a' : A} → a ≅ a' → a' ≅ a
sym refl = refl


trans : {A : Set} → {a b c : A} → a ≅ b → b ≅ c → a ≅ c
trans refl p = p
```

These proof terms show Agda's ability to pattern match on function arguments. More precisely, in the proof of symmetry, we pattern match on the given equality proof of type a ≅ a' which gives us the canonical inhabitant refl and unifies a and a'. Similarly, we pattern match on the proof of type a ≅ b and unify a and b when showing transitivity.

To conclude our overview, we make use of Agda's postulation mechanism to postulate an assumption about functional extensionality for _≅_ which cannot be proved in Agda's intensional type theory. Such postulates do not have a computational behavior and, therefore, can render the type theory inconsistent if used carelessly.

```
postulate
  ext : {A : Set} {B B' : A → Set} {f : (a : A) → B a} {g : (a : A) → B' a} → ((a : A) → f a ≅ g a) → f ≅ g
```

Notice that ext works on dependently typed functions f and g. More precisely, the types of the codomains of f and g depend on the values a passed to the functions. A more thorough overview of Agda can be found in Norell's PhD thesis [30].

Chapter 3

The intermediate language

In this section, we discuss the syntax and semantics of the intermediate language we use as a basis for the extension of value and effect theories in the next chapter. In particular, we study the fine-grained call-by-value language (FGCBV) defined by Levy, Power and Thielecke [21], which is a refinement of Moggi's λ_c-calculus [28] and the monadic metalanguage λ_{ml} [29]. We also spend a considerable proportion of this chapter on presenting the Agda formalization of FGCBV syntax which is later used for defining the extension of value and effect theories and the NBE algorithm. Our formalization is inspired by some previous Agda formalizations of simply typed lambda calculus [3, 9].

3.1 Types

Definition 3.1.1. The FGCBV *type signature* is given by the following grammar

$$\sigma ::= \alpha \mid \beta \mid 1 \mid \sigma \times \sigma \mid \sigma \rightharpoonup \sigma$$

where σ ranges over types and α and β range over base types. In addition, $\rightharpoonup$ is the usual call-by-value function space where functions take values and perform, possibly impure, computations before returning values. We only consider finite products and leave the treatment of finite coproducts as a future work outlined in Section 7.1.

The definition gives rise to an inductively defined Agda data type denoting the set of FGCBV types. Notice that we use $\wedge$ for binary products of types and One for the unit type.

```
data Ty : Set where
  α : Ty
```

```
β : Ty
One : Ty
_∧_ : Ty → Ty → Ty
_⇀_ : Ty → Ty → Ty
```

3.2 Contexts

Definition 3.2.1. *Typing contexts* are given as lists of typed variables. For example, we write a context Γ comprised of n variables as

$$\Gamma = x_n : \sigma_n, x_{n-1} : \sigma_{n-1}, ..., x_1 : \sigma_1$$

where we call $x_1 : \sigma_1$ the outermost variable.

In Agda, we define contexts as snoc-lists of nameless variables represented by their types

```
Ctx : Set
Ctx = List Ty
```

where the snoc-lists are defined inductively.

```
data List (A : Set) : Set where
  []  : List A
  _::_ : List A → A → List A
```

In our formalization, we use de Bruijn indices [11] to encode the variables in contexts Γ with the following definition in Agda.

```
data _∈_ : Ty → Ctx → Set where
  Hd : {Γ : Ctx} {σ : Ty} → σ ∈ (Γ :: σ)
  Tl : {Γ : Ctx} {σ τ : Ty} → σ ∈ Γ → σ ∈ (Γ :: τ)
```

These indices give an intuitive numeric encoding to the variables in context. The constructor Hd points to the variable at the head of the given context, i.e., the outermost variable. Similarly, Tl points to the variables in the tail of the given context. As a result, we do not refer to variables with explicit names but instead with their position in contexts.

Definition 3.2.2. A *morphism* between contexts Γ and Γ' is a *type-indexed family of injective renaming functions* from de Bruijn indices of Γ to de Bruijn indices of Γ'. This gives rise to the following definition in Agda

```
Ren : Ctx → Ctx → Set
Ren Γ Γ' = {σ : Ty} → σ ∈ Γ → σ ∈ Γ'
```

together with an identity renaming

```
id-ren : {Γ : Ctx} → Ren Γ Γ
id-ren = id
```

and the composition of two renamings.

```
comp-ren : {Γ Γ' Γ'' : Ctx} → Ren Γ' Γ'' → Ren Γ Γ' → Ren Γ Γ''
comp-ren f g = f · g
```

We later became aware that instead of defining the morphisms as injective renaming functions, one could also give them a first-order representation using *order-preserving embeddings* (OPEs) [10, Section 4.5]. However, we continue using the functional presentation and leave OPEs as future work in Section 7.1.

We often need to weaken both contexts and renamings to introduce new variables. We can weaken a given context by defining a renaming that introduces a new outermost variable.

```
wk₁ : {Γ : Ctx} {σ : Ty} → Ren Γ (Γ :: σ)
wk₁ x = Tl x
```

In addition, we can also define a weakening of other renamings.

```
wk₂ : {Γ Γ' : Ctx} {σ : Ty} → Ren Γ Γ' → Ren (Γ :: σ) (Γ' :: σ)
wk₂ f Hd = Hd
wk₂ f (Tl v) = wk₁ (f v)
```

These weakenings correspond to the objects wk_1 and wk_2 of the category of weakenings considered by Altenkirch, Hofmann and Streicher [4]. In addition, we sometimes also need to exchange two outermost variables in a given context which can be formalized as the following renaming.

```
exchange : {Γ : Ctx} {σ τ : Ty} → Ren (Γ :: τ :: σ) (Γ :: σ :: τ)
exchange Hd = Tl Hd
exchange (Tl Hd) = Hd
exchange (Tl (Tl x)) = Tl (Tl x)
```

Notice that we have not formally enforced the injectivity of these renamings. However, it is straightforward to see that all the considered renamings satisfy this condition. More precisely, the identity renaming is injective and the composition, weakenings and exchange preserve injectivity.

Proposition 3.2.3. It is also well known [14] that contexts and renamings form a category **Ctx**. □

3.3 Value and producer terms

FGCBV has two separate typing judgments, one for value terms and one for producer terms (i.e. computations) respectively written as $\Gamma \vdash_v V : \sigma$ and $\Gamma \vdash_p M : \sigma$ (for a given context Γ and type σ). The well-typed value and producer terms are defined by the following typing rules.

$$\textsf{var}\quad \frac{}{\Gamma, x : \sigma, \Gamma' \vdash_v x : \sigma} \qquad \star\quad \frac{}{\Gamma \vdash_v \star : 1}$$

$$\textsf{pair}\quad \frac{\Gamma \vdash_v V_1 : \sigma_1 \quad \Gamma \vdash_v V_2 : \sigma_2}{\Gamma \vdash_v \langle V_1, V_2 \rangle : \sigma_1 \times \sigma_2} \qquad \textsf{proj}_\textsf{i}\quad \frac{\Gamma \vdash_v V : \sigma_1 \times \sigma_2}{\Gamma \vdash_v \pi_i(V) : \sigma_i}$$

$$\textsf{lam}\quad \frac{\Gamma, x : \sigma \vdash_p N : \tau}{\Gamma \vdash_v \lambda\, x : \sigma . N : \sigma \rightharpoonup \tau} \qquad \textsf{app}\quad \frac{\Gamma \vdash_v V : \sigma \rightharpoonup \tau \quad \Gamma \vdash_v W : \sigma}{\Gamma \vdash_p VW : \tau}$$

$$\textsf{return}\quad \frac{\Gamma \vdash_v V : \sigma}{\Gamma \vdash_p \mathbf{return}\, V : \sigma} \qquad \textsf{to}\quad \frac{\Gamma \vdash_p M : \sigma \quad \Gamma, x : \sigma \vdash_p N : \tau}{\Gamma \vdash_p M\, \mathbf{to}\, x . N : \tau}$$

These terms are very similar to the simply typed lambda calculus except for the separate typing judgments and two specific producer terms **return** and **to**. Intuitively, $\mathbf{return}\, V$ is the trivial producer that takes a value term V and wraps an effect-free computation around it. $M\, \mathbf{to}\, x . N$ is a sequencing operation for producers. It takes a producer term M, computes its value, binds this value to variable x in N and then computes N. In addition, lambda abstraction gives us call-by-value functions by binding the outermost free-variable in a producer term while application of two value terms results in a producer term.

In Agda, we represent *value and producer terms* using two mutually inductively defined data types where the constructors correspond to the labeled typing rules above.

```
data _⊢v_ (Γ : Ctx) : Ty → Set where
  var : {σ : Ty} → σ ∈ Γ → Γ ⊢v σ
  proj₁ : {σ₁ σ₂ : Ty} → Γ ⊢v σ₁ ∧ σ₂ → Γ ⊢v σ₁
  proj₂ : {σ₁ σ₂ : Ty} → Γ ⊢v σ₁ ∧ σ₂ → Γ ⊢v σ₂
  pair : {σ₁ σ₂ : Ty} → Γ ⊢v σ₁ → Γ ⊢v σ₂ → Γ ⊢v σ₁ ∧ σ₂
  ⋆ : Γ ⊢v One
  lam : {σ τ : Ty} → (Γ :: σ) ⊢p τ → Γ ⊢v σ ⇀ τ

data _⊢p_ (Γ : Ctx) : Ty → Set where
  return : {σ : Ty} → Γ ⊢v σ → Γ ⊢p σ
```

```
_to_ : {σ τ : Ty} → Γ ⊢p σ → (Γ :: σ) ⊢p τ → Γ ⊢p τ
app : {σ τ : Ty} → Γ ⊢v σ ⇀ τ → Γ ⊢v σ → Γ ⊢p τ
```

Here, var x denotes a variable in the given context encoded by the de Bruijn index x. Other term constructors carry their usual meaning discussed earlier modulo some naming conventions (e.g., proj_1 instead of π_1). Later, we write $\Gamma \vdash \mathsf{v}\ \sigma$ and $\Gamma \vdash \mathsf{p}\ \sigma$ to denote the set of all well-typed value and producer terms of type σ in context Γ.

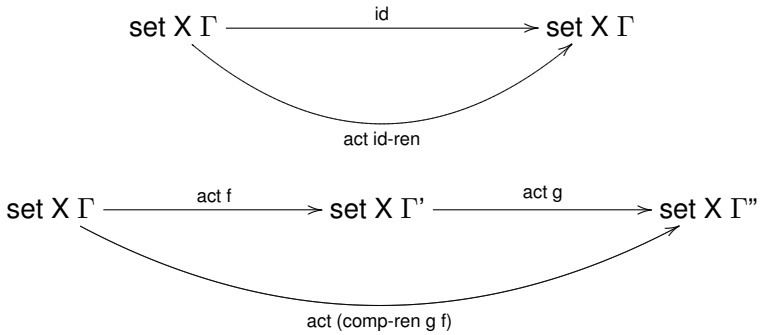
3.4 Algebraic view of value and producer terms

We follow the work of Fiore [14] in using a category of covariant presheaves $\mathbf{Set}^{\mathbf{Ctx}}$, i.e., the category of functors $\mathbf{Ctx} \to \mathbf{Set}$, to give an algebraic characterization of FGCBV syntax. This has been discussed in detail for untyped lambda calculus by Fiore, Plotkin and Simpson [15] and for simply typed lambda calculus by Zsidó [39, Chapter 5]. The category $\mathbf{Ctx}$ we defined above is equivalent to the category $\mathbb{F} \downarrow \mathcal{T}$ of untyped contexts $\mathbb{F}$ over types $\mathcal{T}$ considered by the other authors.

Definition 3.4.1. We denote the *presheaves* in $\mathbf{Set}^{\mathbf{Ctx}}$ as instances of the following record type in Agda

```
record Set^Ctx : Set₁ where
  field set : Ctx → Set
        act : {Γ Γ' : Ctx} → Ren Γ Γ' → set Γ → set Γ'
```

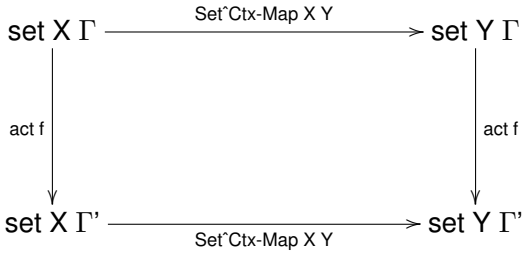
which must satisfy the following identity and composition laws.



Definition 3.4.2. *Maps between presheaves* are natural transformations between functors $\mathbf{Ctx} \to \mathbf{Set}$. In Agda, we use the following short-hand syntax to present the components of these maps

```
Set^Ctx-Map : Set^Ctx → Set^Ctx → Set
Set^Ctx-Map X Y = {Γ : Ctx} → set X Γ → set Y Γ
```

which have to satisfy the following naturality square.



Due to space restrictions, we omit the explicit proofs of these three diagrams for the presheaves we consider. However, these proofs can be found in our Agda formalization [1]. We will also need to construct *products of presheaves* for defining the strength of monads. We write $\mathsf{X} \otimes \mathsf{Y}$ for the binary product of two presheaves X and Y and define them point-wise using the binary products in **Set**. A more thorough introduction to presheaves has been given by Mac Lane and Moerdijk [24, § II].

Following Fiore [14], we know that the category $\mathbf{Set}^{\mathbf{Ctx}}$ contains the following type-indexed family of presheaves of variables.

$$V_\sigma(\Gamma) \cong \{x \mid (x : \sigma) \in \Gamma\}$$

The algebraic notion of value and producer terms can therefore be given as an initial algebra for the signature endofunctor

$$\langle F_v, F_p \rangle : (\mathbf{Set}^{\mathbf{Ctx}})^{\mathbf{Ty}} \times (\mathbf{Set}^{\mathbf{Ctx}})^{\mathbf{Ty}} \to (\mathbf{Set}^{\mathbf{Ctx}})^{\mathbf{Ty}} \times (\mathbf{Set}^{\mathbf{Ctx}})^{\mathbf{Ty}}$$

with the following components.

$$\begin{aligned}
(F_v(X,Y))_\alpha &= V_\alpha + (E_v(X,Y))_\alpha \\
(F_v(X,Y))_1 &= V_1 + 1 + (E_v(X,Y))_1 \\
(F_v(X,Y))_{\sigma_1 \times \sigma_2} &= V_{\sigma_1 \times \sigma_2} + (X_{\sigma_1} \times X_{\sigma_2}) + (E_v(X,Y))_{\sigma_1 \times \sigma_2} \\
(F_v(X,Y))_{\sigma \to \tau} &= V_{\sigma \to \tau} + (Y_\tau)^{V_\sigma} + (E_v(X,Y))_{\sigma \to \tau} \\
(F_p(X,Y))_\sigma &= X_\sigma + (E_p(X,Y))_\sigma \\
\\
(E_v(X,Y))_\tau &= \amalg_{\sigma \in \mathbf{Ty}}\ X_{\tau \times \sigma} + X_{\sigma \times \tau} \\
(E_p(X,Y))_\tau &= \amalg_{\sigma \in \mathbf{Ty}}\ (X_{\sigma \to \tau} \times X_\sigma) + (Y_\sigma \times (Y_\tau)^{V_\sigma})
\end{aligned}$$

Definition 3.4.3. The *initial* $\langle F_v, F_p \rangle$*-algebra* giving us a term model can be explicitly represented by *type-indexed families of presheaves*

- $\mathsf{VTerms}_\sigma(\Gamma) = \{\mathsf{t} \mid \mathsf{t} \in (\Gamma \vdash \mathsf{v}\ \sigma)\}$
- $\mathsf{PTerms}_\sigma(\Gamma) = \{\mathsf{t} \mid \mathsf{t} \in (\Gamma \vdash \mathsf{p}\ \sigma)\}$

describing well-typed value and producer terms of a given type. These families of presheaves have a straightforward definition in Agda with set assigning a set of well-typed terms of given type to a given context, e.g.,

```
VTerms : Ty → Set^Ctx
VTerms σ = record {
  set Γ = Γ ⊢v σ;
  act f t = ⊢v-rename f t
 }
```

The *actions of renaming* on value and producer terms

```
⊢v-rename : {σ : Ty} {Γ Γ' : Ctx} → Ren Γ Γ' → Γ ⊢v σ → Γ' ⊢v σ
⊢p-rename : {σ : Ty} {Γ Γ' : Ctx} → Ren Γ Γ' → Γ ⊢p σ → Γ' ⊢p σ
```

are defined by straightforward structural recursion.

```
⊢v-rename f (var x) = var (f x)
⊢v-rename f (proj₁ t) = proj₁ (⊢v-rename f t)
⊢v-rename f (proj₂ t) = proj₂ (⊢v-rename f t)
⊢v-rename f (pair t u) = pair (⊢v-rename f t) (⊢v-rename f u)
⊢v-rename f ⋆ = ⋆
⊢v-rename f (lam t) = lam (⊢p-rename (wk₂ f) t)
⊢p-rename f (return t) = return (⊢v-rename f t)
⊢p-rename f (t to u) = ⊢p-rename f t to ⊢p-rename (wk₂ f) u
⊢p-rename f (app t u) = app (⊢v-rename f t) (⊢v-rename f u)
```

We see that, except for three cases, the renaming is pushed unchanged inside the term constructors. However, for variables, we use the renaming to compute a new de Bruijn index in context Γ'. For lambda abstractions and sequenced producers, we use a weakened renaming to avoid renaming the outermost free variable that the term constructor binds.

3.5 Substitutions

To present the equational theory of FGCBV, we first need to define substitutions. We define a notion of parallel substitutions in which all free variables in a given context are substituted with value terms.

Definition 3.5.1. Given two contexts Γ and Γ', a *parallel substitution* is defined as a family of functions from de Bruijn indices in Γ to well-typed value terms in Γ'.

This gives rise to a straightforward definition in Agda

```
Sub : Ctx → Ctx → Set
Sub Γ Γ' = {σ : Ty} → σ ∈ Γ → Γ' ⊢v σ
```

together with the *identity substitution*

```
id-subst : {Γ : Ctx} → Sub Γ Γ
id-subst x = var x
```

and the *composition of two substitutions*

```
comp-subst : {Γ Γ' Γ'' : Ctx} → Sub Γ' Γ'' → Sub Γ Γ' → Sub Γ Γ''
comp-subst f g = subst-v f · g
```

and the *extension of a substitution* with a given value term.

```
ext-subst : {Γ Γ' : Ctx} {σ : Ty} → Sub Γ Γ' → Γ' ⊢v σ → Sub (Γ :: σ) Γ'
ext-subst f t Hd = t
ext-subst f t (Tl x) = f x
```

Similarly to the weakening of renamings, we define a corresponding notion, called *lifting*, for substitutions that passes the new outermost variable through unsubstituted.

```
lift : {Γ Γ' : Ctx} {σ : Ty} → Sub Γ Γ' → Sub (Γ :: σ) (Γ' :: σ)
lift f Hd = var Hd
lift f (Tl x) = ⊢v-rename wk₁ (f x)
```

In addition, we can also define the *action of substitution*

```
subst-v : {Γ Γ' : Ctx} → Sub Γ Γ' → {σ : Ty} → Γ ⊢v σ → Γ' ⊢v σ
subst-p : {Γ Γ' : Ctx} → Sub Γ Γ' → {σ : Ty} → Γ ⊢p σ → Γ' ⊢p σ
```

on both value and producer terms by straightforward structural recursion.

```
subst-v f (var x) = f x
subst-v f (proj₁ t) = proj₁ (subst-v f t)
subst-v f (proj₂ t) = proj₂ (subst-v f t)
subst-v f (pair t u) = pair (subst-v f t) (subst-v f u)
subst-v f ⋆ = ⋆
subst-v f (lam t) = lam (subst-p (lift f) t)
subst-p f (return t) = return (subst-v f t)
subst-p f (t to u) = subst-p f t to subst-p (lift f) u
subst-p f (app t u) = app (subst-v f t) (subst-v f u)
```

The one-place substitutions $M[V/x]$, we use in the more informal presentation of FGCBV equational theory, can be accommodated in this formalization by subst-p (ext-subst id-subst V) M. Intuitively, this means that all free variables in M, besides the outermost, are substituted with themselves and the outermost is substituted with V.

3.6 Equational theory

We now present the *equational theory* of FGCBV. The theory is given by two sets of equations between well-typed terms in context, written as $\Gamma \vdash_v V \equiv W : \sigma$ for well-typed value terms and $\Gamma \vdash_p M \equiv N : \sigma$ for well-typed producer terms. We begin by presenting the equations in natural-deduction style.

$$\beta\times_i \qquad \frac{\Gamma \vdash_v V_1 : \sigma_1 \quad \Gamma \vdash_v V_2 : \sigma_2}{\Gamma \vdash_v \pi_i(\langle V_1, V_2\rangle) \equiv V_i : \sigma_i}$$

$$\eta\star \qquad \frac{\Gamma \vdash_v V : 1}{\Gamma \vdash_v V \equiv \star : 1}$$

$$\eta\times \qquad \frac{\Gamma \vdash_v V : \sigma_1 \times \sigma_2}{\Gamma \vdash_v V \equiv \langle \pi_1(V), \pi_2(V)\rangle : \sigma_1 \times \sigma_2}$$

$$\eta\multimap \qquad \frac{\Gamma \vdash_v V\sigma \multimap \tau}{\Gamma \vdash_v V \equiv \lambda x : \sigma.(Vx) : \sigma \multimap \tau}$$

$$\beta\multimap \qquad \frac{\Gamma, x : \sigma \vdash_p M : \tau \quad \Gamma \vdash_v V : \sigma}{\Gamma \vdash_p (\lambda x : \sigma.M)V \equiv M[V/x] : \tau}$$

$$\beta\mathsf{to} \qquad \frac{\Gamma \vdash_v V : \sigma \quad \Gamma, x : \sigma \vdash_p N : \tau}{\Gamma \vdash_p \mathbf{return}\, V\, \mathbf{to}\, x.N \equiv N[V/x] : \tau}$$

$$\eta\mathsf{to} \qquad \frac{\Gamma \vdash_p M : \sigma \quad \Gamma, x : \sigma \vdash_v x : \sigma}{\Gamma \vdash_p M \equiv M\, \mathbf{to}\, x.\, \mathbf{return}\, x : \sigma}$$

$$\mathsf{assocto} \qquad \frac{\Gamma \vdash_p M : \sigma \quad \Gamma, x : \sigma \vdash_p N : \tau \quad \Gamma, y : \tau \vdash_p P : \rho}{\Gamma \vdash_p (M\, \mathbf{to}\, x.N)\, \mathbf{to}\, y.P \equiv M\, \mathbf{to}\, x.(N\, \mathbf{to}\, y.P) : \rho}$$

Moreover, we require this equational theory to be an equivalence relation which is also compatible with all term constructors, i.e., a congruence relation. We make this explicit in the Agda definition below. We present both sets of equations using two mutually defined inductive data types that are indexed over contexts and types and whose constructors

correspond to the named equations above.

The set of equations between value terms consists of four blocks, namely, equations of an equivalence relation, equations of a congruence relation, β-equations and η-equations.

data _⊢v_≡_ : (Γ : Ctx) → {σ : Ty} → Γ ⊢v σ → Γ ⊢v σ → Set **where**

≡-refl : {Γ : Ctx} {σ : Ty} {t : Γ ⊢v σ} → Γ ⊢v t ≡ t
≡-symm : {Γ : Ctx} {σ : Ty} {t u : Γ ⊢v σ} → Γ ⊢v t ≡ u → Γ ⊢v u ≡ t
≡-trans : {Γ : Ctx} {σ : Ty} {t u v : Γ ⊢v σ} → Γ ⊢v t ≡ u → Γ ⊢v u ≡ v → Γ ⊢v t ≡ v

cong, : {Γ : Ctx} {σ_1 σ_2 : Ty} {t t' : Γ ⊢v σ_1} {u u' : Γ ⊢v σ_2} → Γ ⊢v t ≡ t' → Γ ⊢v u ≡ u'
→ Γ ⊢v pair t u ≡ pair t' u'
congproj$_1$: {Γ : Ctx} {σ_1 σ_2 : Ty} {t u : Γ ⊢v σ_1 ∧ σ_2} → Γ ⊢v t ≡ u → Γ ⊢v proj$_1$ t ≡ proj$_1$ u
congproj$_2$: {Γ : Ctx} {σ_1 σ_2 : Ty} {t u : Γ ⊢v σ_1 ∧ σ_2} → Γ ⊢v t ≡ u → Γ ⊢v proj$_2$ t ≡ proj$_2$ u
conglam : {Γ : Ctx} {σ τ : Ty} {t u : Γ :: σ ⊢p τ} → (Γ :: σ) ⊢p t ≡ u → Γ ⊢v lam t ≡ lam u

$\beta\times_1$: {Γ : Ctx} {σ_1 σ_2 : Ty} {t : Γ ⊢v σ_1} {u : Γ ⊢v σ_2} → Γ ⊢v proj$_1$ (pair t u) ≡ t
$\beta\times_2$: {Γ : Ctx} {σ_1 σ_2 : Ty} {t : Γ ⊢v σ_1} {u : Γ ⊢v σ_2} → Γ ⊢v proj$_2$ (pair t u) ≡ u

$\eta\star$: {Γ : Ctx} {t : Γ ⊢v One} → Γ ⊢v t ≡ ⋆
$\eta\times$: {Γ : Ctx} {σ_1 σ_2 : Ty} {t : Γ ⊢v σ_1 ∧ σ_2} → Γ ⊢v t ≡ pair (proj$_1$ t) (proj$_2$ t)
$\eta\rightharpoonup$: {Γ : Ctx} {σ τ : Ty} {t : Γ ⊢v σ ⇀ τ} → Γ ⊢v lam (app ((⊢v-rename wk$_1$ t)) (var Hd)) ≡ t

The set of equations between producer terms consists of five blocks, namely, equations of an equivalence relation, equations of a congruence relation, β-equations, one η-equation and the associativity equation.

data _⊢p_≡_ : (Γ : Ctx) → {σ : Ty} → Γ ⊢p σ → Γ ⊢p σ → Set **where**

≡-refl : {Γ : Ctx} {σ : Ty} {t : Γ ⊢p σ} → Γ ⊢p t ≡ t
≡-symm : {Γ : Ctx} {σ : Ty} {t u : Γ ⊢p σ} → Γ ⊢p t ≡ u → Γ ⊢p u ≡ t
≡-trans : {Γ : Ctx} {σ : Ty} {t u v : Γ ⊢p σ} → Γ ⊢p t ≡ u → Γ ⊢p u ≡ v → Γ ⊢p t ≡ v

congapp : {Γ : Ctx} {σ τ : Ty} {t t' : Γ ⊢v σ ⇀ τ} {u u' : Γ ⊢v σ} → Γ ⊢v t ≡ t' → Γ ⊢v u ≡ u'
→ Γ ⊢p app t u ≡ app t' u'
congto : {Γ : Ctx} {σ τ : Ty} {t t' : Γ ⊢p σ} {u u' : (Γ :: σ) ⊢p τ} → Γ ⊢p t ≡ t' → (Γ :: σ) ⊢p u ≡ u'
→ Γ ⊢p t to u ≡ t' to u'
congreturn : {Γ : Ctx} {σ : Ty} {t t' : Γ ⊢v σ} → Γ ⊢v t ≡ t' → Γ ⊢p return t ≡ return t'

$\beta\rightharpoonup$: {Γ : Ctx} {σ τ : Ty} {t : (Γ :: σ) ⊢p τ} {u : Γ ⊢v σ}
→ Γ ⊢p subst-p (ext-subst id-subst u) t ≡ app (lam t) u
$\beta$to : {Γ : Ctx} {σ τ : Ty} {t : (Γ :: σ) ⊢p τ} {u : Γ ⊢v σ}
→ Γ ⊢p (return u) to t ≡ subst-p (ext-subst id-subst u) t

$\eta$to : {Γ : Ctx} {σ : Ty} {t : Γ ⊢p σ} → Γ ⊢p t ≡ t to return (var Hd)

assocto : {Γ : Ctx} {σ τ : Ty} {t : Γ ⊢p σ} {u : (Γ :: σ) ⊢p τ} {v : (Γ :: τ) ⊢p }
→ Γ ⊢p (t to u) to v ≡ t to (u to ⊢p-rename exchange (⊢p-rename wk_1 v))

As we have not changed the names of the equations, the reader can easily compare them with the presentation in natural deduction style. The most significant difference appears in assocto where we have explicitly weakened v and swapped the two outermost variables.

3.7 Denotational semantics

To be able to develop the NBE algorithm in Chapter 5, we first need to define the denotational semantics of FGCBV. The semantic model we consider for FGCBV in this dissertation is the monad model on a cartesian-closed category $\mathcal{C}$ discussed in Section 2.2. We follow the usual approach [32] of giving categorical semantics by interpreting types as objects of $\mathcal{C}$ and judgments as morphisms of $\mathcal{C}$.

We begin by explicitly defining the objects that individual FGCBV types denote.

Definition 3.7.1. The *interpretation* $[\![-]\!] : \mathsf{Ty} \to Ob(\mathcal{C})$ *of FGCBV types* is given by structural recursion on types.

- $[\![\alpha]\!]$ = a distinguished "base type object" of $\mathcal{C}$
- $[\![\beta]\!]$ = a distinguished "base type object" of $\mathcal{C}$
- $[\![1]\!]$ = the terminal object of $\mathcal{C}$
- $[\![\sigma_1 \times \sigma_2]\!] = [\![\sigma_1]\!] \times [\![\sigma_2]\!]$ *(using binary products in $\mathcal{C}$)*
- $[\![\sigma \rightharpoonup \tau]\!] = [\![\sigma]\!] \Rightarrow T[\![\tau]\!]$ *(using Kleisli exponentials in $\mathcal{C}$)*

Later, we refer to the objects that FGCBV types denote as semantic values to emphasize that they stand for values computed by programs. This interpretation is extended to contexts in the usual way by using finite products in $\mathcal{C}$.

$$[\![\Gamma]\!] = [\![x_n : \sigma_n, x_{n-1} : \sigma_{n-1}, ..., x_1 : \sigma_1]\!] = [\![\sigma_n]\!] \times [\![\sigma_{n-1}]\!] \times ... \times [\![\sigma_1]\!]$$

Definition 3.7.2. The *interpretation of value and producer terms* takes syntactic well-typed terms $\Gamma \vdash_v V : \sigma$ and $\Gamma \vdash_p M : \sigma$ to morphisms $[\![\Gamma \vdash_v V : \sigma]\!]_{\mathrm{v}} : [\![\Gamma]\!] \to [\![\sigma]\!]$ and $[\![\Gamma \vdash_p M : \sigma]\!]_{\mathrm{p}} : [\![\Gamma]\!] \to T[\![\sigma]\!]$ in $\mathcal{C}$. This gives rise to two mutually defined interpretation maps.

- $[\![\Gamma, x_i : \sigma_i, \Gamma' \vdash^v x_i : \sigma_i]\!]_v : [\![\Gamma]\!] \times [\![\sigma_i]\!] \times [\![\Gamma']\!] \to [\![\sigma_i]\!]$
  $[\![x_i]\!]_v\, e = \pi_i\, e$
- $[\![\Gamma \vdash^v \star : 1]\!]_v : [\![\Gamma]\!] \to [\![1]\!]$
  $[\![\star]\!]_v\, e = *$
- $[\![\Gamma \vdash^v \langle V_1, V_2\rangle : \sigma_1 \times \sigma_2]\!]_v : [\![\Gamma]\!] \to [\![\sigma_1]\!] \times [\![\sigma_2]\!]$
  $[\![\langle V_1, V_2\rangle]\!]_v\, e = ([\![V_1]\!]_v\, e, [\![V_2]\!]_v\, e)$
- $[\![\Gamma \vdash^v \pi_i(V) : \sigma_i]\!]_v : [\![\Gamma]\!] \to [\![\sigma_i]\!]$
  $[\![\pi_i(V)]\!]_v\, e = \pi_i([\![V]\!]_v\, e)$
- $[\![\Gamma \vdash^p \mathbf{return}\, V : \sigma]\!]_p : [\![\Gamma]\!] \to T[\![\sigma]\!]$
  $[\![\mathbf{return}\, V]\!]_p\, e = \eta_{[\![\sigma]\!]}([\![V]\!]_v\, e)$
- $[\![\Gamma \vdash^p M\, \mathbf{to}\, x.N : \tau]\!]_p : [\![\Gamma]\!] \to T[\![\tau]\!]$
  $[\![M\, \mathbf{to}\, x.N]\!]_p\, e = (\lambda\,(e,d)\,.\,[\![N]\!]_{\mathrm{p}}(e, x \mapsto d))^*\,(t\,(e, [\![M]\!]_{\mathrm{p}} e))$
- $[\![\Gamma \vdash^v \lambda x : \sigma.N : \sigma \multimap \tau]\!]_v : [\![\Gamma]\!] \to ([\![\sigma]\!] \Rightarrow T[\![\tau]\!])$
  $[\![\lambda x : \sigma.N]\!] = \Lambda_{[\![\Gamma]\!],[\![\sigma]\!],T[\![\tau]\!]}[\![N]\!]_p$
- $[\![\Gamma \vdash^p VW : \tau]\!]_p : [\![\Gamma]\!] \to T[\![\tau]\!]$
  $[\![VW]\!]_p\, e = \varepsilon_{[\![\sigma]\!],T[\![\tau]\!]}([\![V]\!]_v\, e, [\![W]\!]_v\, e)$

where

- $\Lambda_{X,Y,TZ} : \mathcal{C}(X \times Y, TZ) \cong \mathcal{C}(X, Y \Rightarrow TZ)$
- $\varepsilon_{X,TY} : (X \Rightarrow TY) \times X \to TY$

This definition makes it clear that the monad unit η is the semantic counterpart of the trivial producer **return**. Intuitively, it takes a value and returns an effect-free computation. In similar fashion, the Kleisli extension $\_^*$ is the semantic counterpart of sequencing producer terms with **to**.

Theorem 3.7.3. The defined interpretation of FGCBV into a monad model is sound and complete with respect to the equational theory defined in Section 3.6.

Proof. This has been shown by Levy, Power and Thielecke [21, Theorem 4.7]. In particular, they first formulated a sound and complete interpretation for FGCBV into closed Freyd categories and then showed that giving a closed Freyd category is equivalent to giving a monad model. □

Chapter 4

Effect theories

In this chapter, we discuss the use of algebraic theories for reasoning about computational effects. We first recall the formal definition of these value and effect theories and then define an extension to the FGCBV intermediate language.

4.1 Value and effect theories

The algebraic value and effect theories we consider in this dissertation were first defined by Plotkin and Pretnar[33]. Møgelberg and Staton [27] have defined a similar notion of value and effect theories together with an extension to FGCBV. However, they focus on generic effects [37] rather than algebraic effects discussed in this dissertation.

Definition 4.1.1. A *value signature* Σ_{val} is a signature of a many-sorted algebraic theory consisting of a set of base types β, a subset α of base types, called arity types, and function symbols $f : (\boldsymbol{\beta}) \to \beta$ where $\boldsymbol{\beta}$ denotes a vector $\beta_1, ..., \beta_n$.

Value terms are built from variables and function symbols in the usual way as illustrated by two typing rules.

$$\frac{}{\Gamma, x : \beta, \Gamma' \vdash x : \beta} \qquad \frac{\Gamma \vdash v_1 : \beta_1 \quad ... \quad \Gamma \vdash v_n : \beta_n}{\Gamma \vdash f(v_1, ..., v_n) : \beta}$$

Definition 4.1.2. A *value theory* $\mathbb{T}_{\text{val}} = (\Sigma_{\text{val}}, E_{\text{val}})$ consists of a value signature Σ_{val} and a set of equations E_{val} of the form $\Gamma \vdash v_1 \equiv v_2 : \beta$ between well-typed value terms $\Gamma \vdash v_1 : \beta$ and $\Gamma \vdash v_2 : \beta$.

Definition 4.1.3. An *effect signature* Σ_{eff} consists of operation symbols $\mathsf{op} : \boldsymbol{\beta}; \boldsymbol{\alpha}_1, ..., \boldsymbol{\alpha}_n$ where $\boldsymbol{\beta}$ is a list of base types of *parameters* and $\boldsymbol{\alpha}_1, ..., \boldsymbol{\alpha}_n$ are lists of argument *arity*

types.

We follow the usual notation by writing $\mathsf{op} : \boldsymbol{\alpha}_1, ..., \boldsymbol{\alpha}_n$ when $\boldsymbol{\beta}$ is empty and the operation does not take parameters. In addition, we write $\mathsf{op} : \boldsymbol{\beta}; n$ when all the $\boldsymbol{\alpha}_1, ..., \boldsymbol{\alpha}_n$ are empty.

Effect terms are given by the following typing judgments

$$\frac{\Gamma \vdash \mathbf{v} : \boldsymbol{\beta} \quad w : (\boldsymbol{\beta}) \in \Delta}{\Gamma; \Delta \vdash_E w(\mathbf{v})} \qquad \frac{\Gamma \vdash \boldsymbol{p} : \boldsymbol{\beta} \qquad \Gamma, \boldsymbol{x}_1 : \boldsymbol{\alpha}_1; \Delta \vdash_E t_1 \quad ... \quad \Gamma, \boldsymbol{x}_n : \boldsymbol{\alpha}_n; \Delta \vdash_E t_n}{\Gamma; \Delta \vdash_E \mathsf{op}(\boldsymbol{p}; \boldsymbol{x}_1 : \boldsymbol{\alpha}_1.t_1, ..., \boldsymbol{x}_n : \boldsymbol{\alpha}_n.t_n)}$$

where w ranges over effect variables, $\boldsymbol{p}$ ranges over parameters and the additional contexts Δ consist of lists of effect variables.

Writing $\boldsymbol{x}_i : \boldsymbol{\alpha}_i.t_i$ above means that an argument t_i of an operation is dependent on the outcome of the corresponding effect. Plotkin and Pretnar proposed this effect-dependency to finitely describe effects that could have an infinite number of outcomes. In addition, the explicit use of parameters $\mathbf{p}$ allows finite descriptions of infinite families of operations.

Definition 4.1.4. An *effect theory* $\mathbb{T}_{\text{eff}} = (\Sigma_{\text{eff}}, E_{\text{eff}})$ consists of an effect signature Σ_{eff} together with a set of equations E_{eff} of the form $\Gamma; \Delta \vdash_E t_1 \equiv t_2$ between well-typed effect terms $\Gamma; \Delta \vdash_E t_1$ and $\Gamma; \Delta \vdash_E t_2$.

4.1.1 Example effect theories

Non-determinism

To describe non-determinism, we begin by taking an empty value theory and an effect signature consisting of just one operation symbol $\mathsf{or} : 2$. By using infix notation and writing $t \,\mathsf{or}\, u$, we describe a computation that non-deterministically continues either as t or u. The effect theory is then given by this effect signature together with the equations of the theory of semilattices as mentioned at the end of Section 2.2.

Deterministic choice

The theory of deterministic choice describes the usual notion of booleans and if-conditionals while being a simple and intuitive example demonstrating the use of parametrized operations and non-empty value theories.

We begin with a value signature consisting of a base type **bool** and nullary function symbols $\mathsf{true} : \mathbf{bool}$ and $\mathsf{false} : \mathbf{bool}$ with no equations. The effect signature consists of one operation symbol $\mathsf{if} : \mathbf{bool}; 2$. To show the close correspondence with the usual

notion of if-conditionals, we use the infix notation and write the corresponding terms as $\mathsf{if}\, p\, \mathsf{then}\, t\, \mathsf{else}\, u$. The effect theory consists of the following intuitive equations.

- $\Gamma; \Delta \vdash_E \mathsf{if}\, \mathsf{true}\, \mathsf{then}\, t\, \mathsf{else}\, u \equiv t$
- $\Gamma; \Delta \vdash_E \mathsf{if}\, \mathsf{false}\, \mathsf{then}\, t\, \mathsf{else}\, u \equiv u$
- $\Gamma; \Delta \vdash_E \mathsf{if}\, \mathsf{v}\, \mathsf{then}\, (w(\mathsf{true}))\, \mathsf{else}\, (w(\mathsf{false})) \equiv w(\mathsf{v})$

Global store

The third example of an effect theory we consider is global store, i.e., reading from and writing to global memory. The value signature consists of a base type loc of memory locations and data of data together with appropriate function symbols to represent specific locations and data items, e.g., nullary functions denoting locations. One could also add specific function symbols to modify the data, e.g., binary addition function for numeric data. The effect signature consists of two operation symbols for reading data from memory $\mathsf{read} : \mathbf{loc}; \mathbf{data}$ and for updating data in memory $\mathsf{write} : \mathbf{loc}, \mathbf{data}; 1$. The effect theory consists of the seven equations described in detail by Plotkin and Power [36].

4.2 Models of value and effect theories

We now briefly summarize the models of value and effect theories defined by Plotkin and Pretnar [33].

Definition 4.2.1. A *model of a value theory* in category $\mathcal{C}$ with finite products is given by objects $[\![\alpha]\!]$, $[\![\beta]\!]$ for all base types and morphisms $[\![f]\!] : [\![\boldsymbol{\beta}]\!] \to [\![\beta]\!]$ for all function symbols with $[\![\boldsymbol{\beta}]\!] = [\![\beta_1]\!] \times ... \times [\![\beta_n]\!]$. Value terms $\Gamma \vdash: \beta$ are interpreted as morphisms $[\![\Gamma]\!] \to [\![\sigma]\!]$ that are required to satisfy all the equations in E_{val}.

Definition 4.2.2. A *model of an effect theory* in category $\mathcal{C}$ with finite products is given by an object X together with a family of morphisms $\{\mathsf{op}_X : [\![\boldsymbol{\beta}]\!] \times \Pi_i X^{[\![\boldsymbol{\alpha}_i]\!]} \to X\}_{\mathsf{op}:\boldsymbol{\beta};\boldsymbol{\alpha}_1,\ldots,\boldsymbol{\alpha}_n \in \Sigma_{\text{eff}}}$. Given $\boldsymbol{p} \in [\![\boldsymbol{\beta}]\!]$, the maps $\mathsf{op}_X(\boldsymbol{b}, -)$ have to satisfy all the equations in E_{eff}.

Such models of effect theories form a category $\mathbf{Mod}_{\text{eff}}$. If $\mathcal{C}$ is locally presentable [36], there exists a forgetful functor $U : \mathsf{Mod}_{\text{eff}} \to \mathbf{Set}$ that maps models to their underlying sets. This forgetful functor also has a left adjoint F that induces a free monad $T = UF$ corresponding to the monad proposed by Moggi to model the computational effect corresponding to the effect theory.

4.3 Extension to the intermediate language

In this section, we extend the value and effect theories of Plotkin and Pretnar to the FGCBV intermediate language. We write $\text{FGCBV}_{\text{eff}}$ for FGCBV extended with value and effect theories.

Definition 4.3.1 (Extension of value theories)**.**

- Every base type in the value signature defines a corresponding base type in $\text{FGCBV}_{\text{eff}}$.
- Every function symbol $f : \boldsymbol{\beta} \to \beta$ defines a typing judgment of value terms.

$$\textsf{fun} \quad \frac{\Gamma \vdash_v V_1 : \beta_1 \quad ... \quad \Gamma \vdash_v V_n : \beta_n}{\Gamma \vdash_v f(V_1, ..., V_n) : \beta}$$

- Every equation $\Gamma \vdash v_1 \equiv v_2 : \beta$ between well-typed value terms defines a corresponding equation between well-typed $\text{FGCBV}_{\text{eff}}$ value terms.

This definition gives rise to the following illustrative extension to the Agda data type of value terms. We define the rudiments of a value theory and present the signatures of the straightforward extension maps to the corresponding structure in $\text{FGCBV}_{\text{eff}}$. First, we define an inductive set of base types

```
data BaseTy : Set where
  α : BaseTy
  β : BaseTy
```

inducing a straightforward extension map to $\text{FGCBV}_{\text{eff}}$.

```
extend-ty : BaseTy → Ty
```

Next, we define contexts of variables of base types

```
ValCtx : Set
ValCtx = List BaseTy
```

together with a corresponding notion of de Bruijn indices

```
data _∈'_ : BaseTy → ValCtx → Set where
  Hd : {Γ : ValCtx} {σ : BaseTy} → σ ∈' (Γ :: σ)
  Tl : {Γ : ValCtx} {σ τ : BaseTy} → σ ∈' Γ → σ ∈' (Γ :: τ)
```

inducing straightforward extension maps to $\text{FGCBV}_{\text{eff}}$.

```
extend-valctx : ValCtx → Ctx
extend-valvar : {Γ : ValCtx}{σ : BaseTy} → σ ∈' Γ → (extend-ty σ) ∈ (extend-valctx Γ)
```

Next, we define value terms as an inductive data type. For illustration, we present variables and one example binary function symbol

```
data _⊢_ (Γ : ValCtx) : BaseTy → Set where
  var : {σ : BaseTy} → σ ∈' Γ → Γ ⊢ σ
  fun : Γ ⊢ β → Γ ⊢ β → Γ ⊢ β
```

that gives rise to the corresponding $\text{FGCBV}_{\text{eff}}$ value term

```
data _⊢v_ (Γ : Ctx) : Ty → Set where
  ...
  fun : Γ ⊢v β → Γ ⊢v β → Γ ⊢v β
```

and induces a straightforward extension map.

```
extend-val : {Γ : ValCtx}{σ : BaseTy} → Γ ⊢ σ → (extend-valctx Γ) ⊢v (extend-ty σ)
```

In this dissertation, we consider an extension of a slightly different version of the effect theories of Plotkin and Pretnar. More precisely, we work with effect signatures consisting of effect variables $w : [\,]$ and operations $op : \boldsymbol{\beta}; n$. Recall from Definition 4.1.3 that n corresponds to the list $\boldsymbol{\alpha}_1, ..., \boldsymbol{\alpha}_n$ of lists of argument arity types where all $\boldsymbol{\alpha}_i$ are empty. However, we conjecture that the proof of the conservativity theorem will also hold in the more general situation. We leave this as a future work and summarize the subtleties in Section 7.1.

Definition 4.3.2 (Extension of effect theories)**.**

- Given an $\text{FGCBV}_{\text{eff}}$ type σ, every effect variable $w : []$ defines an $\text{FGCBV}_{\text{eff}}$ variable $x : 1 \rightharpoonup \sigma$.
- Given an $\text{FGCBV}_{\text{eff}}$ type σ, every operation symbol $\mathsf{op} : \boldsymbol{\beta}; n$ defines a typing rule of producer terms.

$$\mathsf{op}\quad \frac{\Gamma \vdash_v p_1 : \beta_1 \ ... \ \Gamma \vdash_p p_n : \beta_n \qquad \Gamma \vdash_p M_1 : \sigma \ ... \ \Gamma \vdash_p M_n : \sigma}{\Gamma \vdash_p \ \mathrm{op}_\sigma((p_1, ..., p_m); (M_1, ...M_n)) : \sigma}$$

- Every equation $\Gamma; \Delta \vdash_E t_1 \equiv t_2$ between well-typed effect terms defines a corresponding equation between well-typed producer terms.
- We add the following *algebraicity equation* for each operation to relate operations at different types.

$$\mathsf{op\text{-}algebraicity}\quad \frac{\Gamma \vdash_p \mathbf{p} : \boldsymbol{\beta} \qquad \Gamma \vdash_p M_1 : \sigma \quad ... \quad \Gamma \vdash_p M_n : \sigma \qquad \Gamma, x : \sigma \vdash_p N : \tau}{\Gamma \vdash_p \ \mathrm{op}_\sigma(\mathbf{p}; M_1, ..., M_n)\,\mathbf{to}\,x.N \equiv \ \mathrm{op}_\tau(\mathbf{p}; M_1\,\mathbf{to}\,x.N, \ ...\, M_n\,\mathbf{to}\,x.N) : \tau}$$

Intuitively, the producer term $\mathrm{op}_\sigma((p_1, ..., p_m); (M_1 ... M_{nl}))$ now denotes a computation that causes the effect determined by op with parameters $p_1, ..., p_m$ and then continues as one of the computations denoted by producer terms $M_1, ..., M_n$.

This gives rise to a straightforward formalization in Agda. First, we define the effect contexts as lists of empty lists (denoted with the canonical element of the singleton set Unit)

```
EffCtx : Set
EffCtx = List Unit
```

together with corresponding de Bruijn indices

```
data _∈''_ : Unit → EffCtx → Set where
  Hd : {Δ : EffCtx} {w : Unit} → w ∈'' (Δ :: w)
  Tl : {Δ : EffCtx} {w w' : Unit} → w ∈'' Δ → w ∈'' (Δ :: w')
```

both inducing extensions

```
extend-effctx : Ty → EffCtx → Ctx
extend-effvar : {Γ : ValCtx}{Δ : EffCtx} {w : Unit} {σ : Ty} → w ∈'' Δ
                → (One ⇀ σ) ∈ (extend-valctx Γ) @ (extend-effctx σ Δ)
```

where @ is the concatenation of two contexts (i.e., lists).

Finally, we define effect terms as an inductive data type illustrated with one binary operation parametrized over a parameter of base type β.

```
data _,_⊢E (Γ : ValCtx) (Δ : EffCtx) : Set where
  var : {n : Unit} → (w : n ∈'' Δ) → Γ , Δ ⊢E
  op : Γ ⊢ β → Γ , Δ ⊢E σ → Γ , Δ ⊢E σ → Γ , Δ ⊢E σ
```

together with a corresponding FGCBV producer term

```
data _⊢p_ (Γ : Ctx) : Ty → Set where
  ...
  op : {σ : Ty} → Γ ⊢v β → Γ ⊢p σ → Γ ⊢p σ → Γ ⊢p σ
```

and an extension map

```
extend-eff : {Γ : ValCtx} {Δ : EffCtx} → (σ : Ty) → Γ , Δ ⊢E
             → (extend-valctx Γ) @ (extend-effctx σ Δ) ⊢p σ
```

where effect variables are extended by using application app (var (extend-effvar w)) ⋆.

To conclude this section, we show how the equations in value and effect theories can be defined in Agda as inductive data types.

```
data _⊢_≡_ : (Γ : ValCtx) → {σ : BaseTy} → Γ ⊢ σ → Γ ⊢ σ → Set where
  ≡-refl : {Γ : ValCtx} {σ : BaseTy} {t : Γ ⊢ σ} → Γ ⊢ t ≡ t
  ≡-symm : {Γ : ValCtx} {σ : BaseTy} {t u : Γ ⊢ σ} → Γ ⊢ t ≡ u → Γ ⊢ u ≡ t
  ≡-trans :  {Γ : ValCtx} {σ : BaseTy} {t u v : Γ ⊢ σ} → Γ ⊢ t ≡ u → Γ ⊢ u ≡ v → Γ ⊢ t ≡ v
  congf :  {Γ : ValCtx} {σ : BaseTy} {t u v w : Γ ⊢ σ} → Γ ⊢ t ≡ u → Γ ⊢ v ≡ w → Γ ⊢ f t u ≡ f v w

data _,_⊢E_≡_ : (Γ : ValCtx) → (Δ : EffCtx) → Γ , Δ ⊢E → Γ , Δ ⊢E → Set where
  ≡-refl : {Γ : ValCtx} {Δ : EffCtx} {t : Γ , Δ ⊢E} → Γ , Δ ⊢E t ≡ t
  ≡-symm : {Γ : ValCtx} {Δ : EffCtx} {t u : Γ , Δ ⊢E} → Γ , Δ ⊢E t ≡ u → Γ , Δ ⊢E u ≡ t
  ≡-trans : {Γ : ValCtx} {Δ : EffCtx} {t u v : Γ , Δ ⊢E} → Γ , Δ ⊢E t ≡ u
          → Γ , Δ ⊢E u ≡ v → Γ , Δ ⊢E t ≡ v
  congif : {Γ : ValCtx} {Δ : EffCtx} {b b' : Γ ⊢ Bool} {t u t' u' : Γ , Δ ⊢E} → Γ ⊢ b ≡ b'
          → Γ , Δ ⊢E t ≡ t' → Γ , Δ ⊢E u ≡ u' → Γ , Δ ⊢E if b then t else u ≡ if b' then t' else u'
```

These data types need to be extended with equations of specific value and effect theories. Every equation $\Gamma \vdash \mathsf{t} \equiv \mathsf{u}$ defines an equation (extend-valctx Γ) ⊢v (extend-val t) ≡ (extend-val u) between value terms. In addition, given an $\mathrm{FGCBV}_{\mathrm{eff}}$ type σ, every equation $\Gamma , \Delta \vdash \mathsf{E}\ \mathsf{t} \equiv \mathsf{u}$ defines an equation (extend-valctx Γ) @ (extend-effctx σ Δ) ⊢p (extend-eff σ t) ≡ (extend-eff σ u) between producer terms. For example, in case of non-determinism, we add the following three equations together with a straightforward congruence equation for or.

```
or-idempotency : {Γ : ValCtx} {Δ : EffCtx} {t : Γ , Δ ⊢E} → Γ , Δ ⊢E or t t ≡ t
or-commutativity : {Γ : ValCtx} {Δ : EffCtx} {t u : Γ , Δ ⊢E} → Γ , Δ ⊢E or t u ≡ or u t
or-associativity : {Γ : ValCtx} {Δ : EffCtx} {t u v : Γ , Δ ⊢E} → Γ , Δ ⊢E or (or t u) v ≡ or t (or u v)
```

4.3.1 Extension of example effect theories

We now show how the three example theories of non-determinism, deterministic choice and global store from Section 4.1.1 can be extended to $\mathrm{FGCBV}_{\mathrm{eff}}$.

Non-determinism

We extend $\mathrm{FGCBV}_{\mathrm{eff}}$ with a binary producer term

$$\frac{\Gamma \vdash_p M_1 : \sigma \quad \Gamma \vdash_p M_2 : \sigma}{\Gamma \vdash_p \mathbf{or}_\sigma(M_1, M_2) : \sigma}$$

together with the algebraicity equation

$$\text{or algebraicity} \quad \frac{\Gamma \vdash_p M_1 : \sigma \quad \Gamma \vdash_p M_2 : \sigma \qquad \Gamma, x : \sigma \vdash_p N : \tau}{\Gamma \vdash_p \mathbf{or}_\sigma(M_1, M_2)\,\mathbf{to}\,x.N \equiv \mathbf{or}_\tau(M_1\,\mathbf{to}\,x.N, M_2\,\mathbf{to}\,x.N) : \tau}$$

and the equations of the theory of semilattices.

$$\text{or idempotency} \quad \frac{\Gamma \vdash_p M : \sigma}{\Gamma \vdash_p \mathbf{or}_\sigma(M, M) \equiv M : \sigma}$$

$$\text{or commutativity} \quad \frac{\Gamma \vdash_p M_1 : \sigma \quad \Gamma \vdash_p M_2 : \sigma}{\Gamma \vdash_p \mathbf{or}_\sigma(M_1, M_2) \equiv \mathbf{or}_\sigma(M_2, M_1) : \sigma}$$

$$\text{or associativity} \quad \frac{\Gamma \vdash_p M_1 : \sigma \quad \Gamma \vdash_p M_2 : \sigma \quad \Gamma \vdash_p M_3 : \sigma}{\Gamma \vdash_p \mathbf{or}_\sigma(\mathbf{or}_\sigma(M_1, M_2), M_3) \equiv \mathbf{or}_\sigma(M_1, \mathbf{or}_\sigma(M_2, M_3)) : \sigma}$$

Deterministic choice

We define a new base type bool denoting booleans together with two value terms denoting true and false.

$$\frac{}{\Gamma \vdash_v \mathbf{true} : \mathsf{bool}} \qquad \frac{}{\Gamma \vdash_v \mathbf{false} : \mathsf{bool}}$$

We extend FGCBV$_{\text{eff}}$ with a binary producer term parametrized over boolean values of type bool

$$\frac{\Gamma \vdash_v B : \mathsf{bool} \quad \Gamma \vdash_p M_1 : \sigma \quad \Gamma \vdash_p M_2 : \sigma}{\Gamma \vdash_p \mathbf{if}_\sigma\, B\,\mathbf{then}\, M_1\,\mathbf{else}\, M_2 : \sigma}$$

together with the straightforward algebraicity equation and three equations making the behavior of deterministic choice explicit.

$$\text{if true} \quad \frac{\Gamma \vdash_p M_1 : \sigma \quad \Gamma \vdash_p M_2 : \sigma}{\Gamma \vdash_p \mathbf{if}_\sigma\,\mathbf{true}\,\mathbf{then}\, M_1\,\mathbf{else}\, M_2 \equiv M_1 : \sigma}$$

$$\text{if false} \quad \frac{\Gamma \vdash_p M_1 : \sigma \quad \Gamma \vdash_p M_2 : \sigma}{\Gamma \vdash_p \mathbf{if}_\sigma\,\mathbf{false}\,\mathbf{then}\, M_1\,\mathbf{else}\, M_2 \equiv M_2 : \sigma}$$

$$\text{if booleans} \quad \frac{\Gamma \vdash_v B : \mathsf{bool}}{\Gamma \vdash_p \mathbf{if}_\sigma\, B\,\mathbf{then}\,(\mathbf{return}\,\mathbf{true})\,\mathbf{else}\,(\mathbf{return}\,\mathbf{false}) \equiv (\mathbf{return}\, B) : \mathsf{bool}}$$

Global store

We define two new base types loc and data denoting a set of memory locations and a set of possible data values. For example, if we would be working with a single one-bit memory cell, we would add the following three value terms to describe the single cell and two possible data values.

$$\frac{}{\Gamma \vdash_v \mathbf{cell} : \mathsf{loc}} \qquad \frac{}{\Gamma \vdash_v \mathbf{zero} : \mathsf{data}} \qquad \frac{}{\Gamma \vdash_v \mathbf{one} : \mathsf{data}}$$

To support reading from and writing to the locations loc, we define the following two producer terms

$$\frac{\Gamma \vdash_v L : \mathsf{loc} \quad \Gamma \vdash_p M_1 : \sigma \quad ... \quad \Gamma \vdash_p M_n : \sigma}{\Gamma \vdash_p \mathbf{read}_\sigma\,(L; M_1, ..., M_n) : \sigma}$$

$$\frac{\Gamma \vdash_v L : \mathsf{loc} \quad \Gamma \vdash_v V : \mathsf{dat} \quad \Gamma \vdash_p M : \sigma}{\Gamma \vdash_p \mathbf{write}_\sigma\,(L, V; M) : \sigma}$$

where n is the number of possible data values. In addition, we add the equations given by Plotkin and Power [36].

4.4 Semantics of the extended intermediate language

We conclude the extension of value and effect theories by discussing the semantics of $\text{FGCBV}_{\text{eff}}$ which is based on the semantics of FGCBV given in Section 3.7 and the models of value and effect theories given in Section 4.2. Therefore, the model of $\text{FGCBV}_{\text{eff}}$ is a monad model with the additional restriction that it also has to be a model of the corresponding value and effect theories $\mathbb{T}_{\text{val}} = (\Sigma_{\text{val}}, E_{\text{val}})$ and $\mathbb{T}_{\text{eff}} = (\Sigma_{\text{eff}}, E_{\text{eff}})$.

The assumption about being a model of a value theory allows us to extend the interpretation of FGCBV with the interpretation of value terms corresponding to function symbols.

$$[\![f(V_1, ..., V_n)]\!]_{\text{v}}\, e = [\![f]\!]([\![V_1]\!]_{\text{v}}\, e, ..., [\![V_n]\!]_{\text{v}}\, e)$$

The assumption about being a model of an effect theory allows us to extend the interpretation of FGCBV with the interpretation of producer terms corresponding to operations.

$$[\![op_\sigma((p_1, ..., p_m); (M_{11}, ..., M_{1k}), ..., (M_{n1}, ..., M_{nl})]\!]_{\text{p}}\, e =$$

$$\mathsf{op}_{T[\![\sigma]\!]}((\![p_1]\!]_{\mathrm{v}}e, ..., [\![p_m]\!]_{\mathrm{v}}e), ([\![M_{11}]\!]_{\mathrm{p}}e, ..., [\![M_{1k}]\!]_{\mathrm{p}}e), ..., ([\![M_{n1}]\!]_{\mathrm{p}}e, ..., [\![M_{nl}]\!]_{\mathrm{p}}e))$$

4.5 Question of conservativity

Having defined the extension of value and effect theories, the natural question is whether this extension is conservative. More precisely, a theory $\mathbb{T}_2$ is said to be a *conservative extension* of $\mathbb{T}_1$ if every theorem of $\mathbb{T}_1$ is a theorem of $\mathbb{T}_2$ and every theorem of $\mathbb{T}_2$, expressible in $\mathbb{T}_1$, is also a theorem of $\mathbb{T}_1$. When we consider the equational theory of $\mathrm{FGCBV}_{\mathrm{eff}}$ as $\mathbb{T}_2$ and the value and effect theories as $\mathbb{T}_1$, we get the following correspondence. Namely, two value or effect terms are provably equal in the value or effect theory if and only if they are provably equal in $\mathrm{FGCBV}_{\mathrm{eff}}$. The intuition is that the extension must not restrict any behavior determined by value and effect theories ($\Longrightarrow$ direction) while not introducing any new unexpected behavior not determined by value and effect theories ($\Longleftarrow$ direction).

It turns out that the $\Longrightarrow$ direction is relatively straightforward as we show in Chapter 6. On the other hand, we discovered that before proving the $\Longleftarrow$ direction in Chapter 6, one first needs to be able to effectively decide provable equality in $\mathrm{FGCBV}_{\mathrm{eff}}$. For this reason, we spend the whole of Chapter 5 on developing an NBE algorithm for computing inductively defined normal forms. The comparison of these normal forms then gives us the desired method to decide provable modulo to the given value and effect theories.

It is worthwhile to note that the strong normalization of $\mathrm{FGCBV}_{\mathrm{eff}}$ can also be proved by applying the reduction-based method advocated by Girard [16] and also following Lindley and Stark [22] in defining suitable notions of reducibility and continuations for $\mathrm{FGCBV}_{\mathrm{eff}}$ terms. However, we discovered that NBE is more suited for our work and, due to space restrictions, we omit the reduction-based normalization results from this dissertation.

Chapter 5

Normalization by evaluation

Motivated by the question of conservativity of the extension of value and effect theories to the intermediate language, we spend this chapter on developing a normalization algorithm which is used in Chapter 6 for answering this question. The main technique we use is a semantic notion of normalization, called normalization by evaluation (NBE), discussed briefly in Section 2.3. However, we present and formalize a more general algorithm than the usual presentations of NBE. In particular, we do not normalize the value and effect theories and, therefore, compute normal forms modulo the given value and effect theories rather than up to equality.

We begin by inductively defining normal and atomic forms together with a suitable equational theory. Next, we define a suitable denotational semantics together with an NBE algorithm. Finally, we use Kripke logical relations to prove this algorithm correct.

5.1 Normal and atomic forms

We begin with the inductive definition of normal and atomic forms for $\mathrm{FGCBV}_{\mathrm{eff}}$.

Definition 5.1.1. *Normal and atomic forms* are given by typing judgments $\Gamma \vdash_{nv} V : \sigma$ (normal values), $\Gamma \vdash_{np} V : \sigma$ (normal producers), $\Gamma \vdash_{av} V : \sigma$ (atomic values), $\Gamma \vdash_{ap} V : \sigma$ (atomic producers) with well-typed terms given by the following mutually defined rules.

varAV $$\frac{}{\Gamma, x : \sigma, \Gamma' \vdash_{av} x : \sigma}$$

avNV $$\frac{\Gamma \vdash_{av} V : \alpha}{\Gamma \vdash_{nv} V : \alpha}$$

bavNV $$\frac{\Gamma \vdash_{av} V : \beta}{\Gamma \vdash_{nv} V : \beta}$$

unitNV $$\frac{}{\Gamma \vdash_{nv} \star : 1}$$

pairNV $$\frac{\Gamma \vdash_{nv} V_1 : \sigma_1 \quad \Gamma \vdash_{nv} V_2 : \sigma_2}{\Gamma \vdash_{nv} \langle V_1, V_2 \rangle : \sigma_1 \times \sigma_2}$$

funNV $$\frac{\Gamma \vdash_{nv} V_1 : \beta_1 \quad ... \quad \Gamma \vdash_{nv} V_n : \beta_n}{\Gamma \vdash_{nv} f(V_1, ..., V_n) : \beta}$$

proj$_i$AV $$\frac{\Gamma \vdash_{av} V : \sigma_1 \times \sigma_2}{\Gamma \vdash_{av} \pi_i(V) : \sigma_i}$$

appAP $$\frac{\Gamma \vdash_{av} V : \sigma \rightharpoonup \tau \quad \Gamma \vdash_{nv} W : \sigma}{\Gamma \vdash_{ap} VW : \tau}$$

lamNV $$\frac{\Gamma, x : \sigma \vdash_{np} N : \tau}{\Gamma \vdash_{nv} \lambda x : \sigma . N : \sigma \rightharpoonup \tau}$$

returnNP $$\frac{\Gamma \vdash_{nv} V : \sigma}{\Gamma \vdash_{np} \mathbf{return}\, V : \sigma}$$

toNP $$\frac{\Gamma \vdash_{ap} M : \sigma \quad \Gamma, x : \sigma \vdash_{np} N : \tau}{\Gamma \vdash_{np} M\, \mathbf{to}\, x . N : \tau}$$

opNP $$\frac{\Gamma \vdash_{nv} p_1 : \beta_1 \ ... \ \Gamma \vdash_{np} p_n : \beta_n \qquad \Gamma \vdash_{np} M_{11} : \sigma \ ... \ \Gamma \vdash_{np} M_{nl} : \sigma}{\Gamma \vdash_{np} \mathrm{op}_\sigma((p_1, ..., p_m); (M_{11}, ..., M_{1k}), \ ..., (M_{n1}, ..., M_{nl})) : \sigma}$$

These normal forms are reminiscent of the η-long β-normal forms of simply typed lambda calculus with the atomic forms denoting terms that cannot be normalized further. The separate definition of atomic producers is actually redundant as it is possible to give a straightforward derived typing rule for the normal form of **to**. However, we feel that the current definition gives a more intuitive and symmetrical presentation.

Similarly to value and producer terms in Section 3.3, these typing rules give rise to straightforward inductive data types in Agda where the constructors correspond to the labeled typing rules above.

```
data _⊢nv_ (Γ : Ctx) : Ty → Set where
  avNV : Γ ⊢av α → Γ ⊢nv α
  bavNV : Γ ⊢av β → Γ ⊢nv β
  unitNV : Γ ⊢nv One
  pairNV : {σ₁ σ₂ : Ty} → Γ ⊢nv σ₁ → Γ ⊢nv σ₂ → Γ ⊢nv σ₁ ∧ σ₂
  lamNV : {σ τ : Ty} → (Γ :: σ) ⊢np τ → Γ ⊢nv σ ⇁ τ
  funNV : Γ ⊢nv β → Γ ⊢nv β → Γ ⊢nv β
```

data _⊢np_ (Γ : Ctx) : Ty → Set **where**
returnNP : {σ : Ty} → Γ ⊢nv σ → Γ ⊢np σ
toNP : {σ τ : Ty} → Γ ⊢ap σ → (Γ :: σ) ⊢np τ → Γ ⊢np τ
opNP : {σ : Ty} → Γ ⊢nv β → Γ ⊢np σ → Γ ⊢np σ → Γ ⊢np σ

data _⊢av_ (Γ : Ctx) : Ty → Set **where**
varAV : {σ : Ty} → σ ∈ Γ → Γ ⊢av σ
proj$_1$AV : {σ_1 σ_2 : Ty} → Γ ⊢av $\sigma_1 \wedge \sigma_2$ → Γ ⊢av σ_1
proj$_2$AV : {σ_1 σ_2 : Ty} → Γ ⊢av $\sigma_1 \wedge \sigma_2$ → Γ ⊢av σ_2

data _⊢ap_ (Γ : Ctx) : Ty → Set **where**
appAP : {σ τ : Ty} → Γ ⊢av $\sigma \rightharpoonup \tau$ → Γ ⊢nv σ → Γ ⊢ap τ

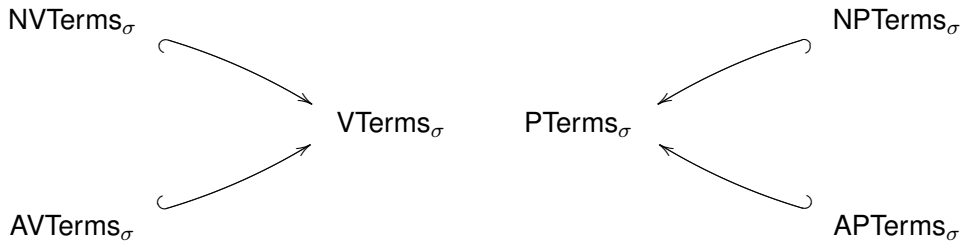
For presentation purposes, and to follow Section 4.3, we let funNV range over binary function symbols and opNP range over binary operations with one parameter.

Similarly to Section 3.4, it is possible to give an algebraic account of normal and atomic forms. This time we consider the initial algebra for a signature endofunctor $\langle F_{nv}, F_{np}, F_{av}, F_{ap}\rangle$ on $(\mathbf{Set}^{\mathbf{Ctx}})^{\mathbf{Ty}} \times (\mathbf{Set}^{\mathbf{Ctx}})^{\mathbf{Ty}} \times (\mathbf{Set}^{\mathbf{Ctx}})^{\mathbf{Ty}} \times (\mathbf{Set}^{\mathbf{Ctx}})^{\mathbf{Ty}}$. We omit the components here as they can be easily recovered from the families of presheaves describing the initial algebra, i.e.,

- $\mathsf{NVTerms}_\sigma(\Gamma) = \{\mathsf{t} \mid \mathsf{t} \in (\Gamma \vdash \mathsf{nv}\ \sigma)\}$, *(normal values)*
- $\mathsf{NPTerms}_\sigma(\Gamma) = \{\mathsf{t} \mid \mathsf{t} \in (\Gamma \vdash \mathsf{np}\ \sigma)\}$, *(normal producers)*
- $\mathsf{AVTerms}_\sigma(\Gamma) = \{\mathsf{t} \mid \mathsf{t} \in (\Gamma \vdash \mathsf{av}\ \sigma)\}$, *(atomic values)*
- $\mathsf{APTerms}_\sigma(\Gamma) = \{\mathsf{t} \mid \mathsf{t} \in (\Gamma \vdash \mathsf{ap}\ \sigma)\}$. *(atomic producers)*

that again have straightforward definitions in Agda.

It is also easy to see that every normal and atomic form defines a corresponding FGCBV$_{\text{eff}}$ value or a producer term with corresponding structure. In particular, there exist the following families of embeddings



which are defined as families of maps between presheaves.

```
⊢nv-embed : {σ : Ty} → Set^Ctx-Map (NVTerms σ) (VTerms σ)
⊢av-embed : {σ : Ty} → Set^Ctx-Map (AVTerms σ) (VTerms σ)
⊢np-embed : {σ : Ty} → Set^Ctx-Map (NPTerms σ) (PTerms σ)
⊢ap-embed : {σ : Ty} → Set^Ctx-Map (APTerms σ) (PTerms σ)
```

Finally, we can also formalize an extension of value and effect theories to normal forms. This extension is similar to Section 4.3. In short, every function symbol in a value theory defines a corresponding normal value and every operation in an effect theory defines a corresponding normal producer for a given type. This canonical structure also induces two extension maps

```
extend-val-nv : {Γ : ValCtx}{σ : BaseTy} → Γ ⊢ σ → (extend-valctx Γ) ⊢nv (extend-ty σ)
extend-eff-np : {Γ : ValCtx}{Δ : EffCtx} → (σ : Ty) → Γ , Δ ⊢E
                  → (extend-valctx Γ) @ (extend-effctx σ Δ) ⊢np σ
```

where effect variables are extended using the NBE algorithm we define in this chapter, i.e., as nf-p (app (var (extend-effvar w)) ⋆).

5.2 Equational theories of normal and atomic forms

In usual presentations of NBE, normal and atomic forms are identified up to equality. However, as we do not aim to normalize the value and effect theories, need to define a more general equivalence between them by equipping both normal and atomic forms with suitable equational theories.

We begin by identifying atomic values up to the heterogeneous equality.

```
_⊢av_≡_ : (Γ : Ctx) → {σ : Ty} → Γ ⊢av σ → Γ ⊢av σ → Set
Γ ⊢av t ≡ u = t ≅ u
```

We define the equational theories of normal values, normal producers and atomic producers as three mutually defined sets of equations similarly to $\mathrm{FGCBV}_{\mathrm{eff}}$ equational theory in Section 3.6. These equations identify normal values, normal producers and atomic producers up to equivalence relations that are compatible with all the term constructors. In Agda, we again present these sets of equations as inductive data types. For example, below is the data type for normal values.

```
data _⊢nv_≡_ : (Γ : Ctx) → {σ : Ty} → Γ ⊢nv σ → Γ ⊢nv σ → Set where
  ≡-refl : {Γ : Ctx} {σ : Ty} {t : Γ ⊢nv σ} → Γ ⊢nv t ≡ t
  ≡-sym : {Γ : Ctx} {σ : Ty} {t u : Γ ⊢nv σ} → Γ ⊢nv t ≡ u → Γ ⊢nv u ≡ t
  ≡-trans : {Γ : Ctx} {σ : Ty} {t u v : Γ ⊢nv σ} → Γ ⊢nv t ≡ u → Γ ⊢nv u ≡ v → Γ ⊢nv t ≡ v
  congav : {Γ : Ctx} {t u : Γ ⊢av α} → Γ ⊢av t ≡ u → Γ ⊢nv avNV t ≡ avNV u
```

```
congbav : {Γ : Ctx} {t u : Γ ⊢av β} → Γ ⊢av t ≡ u → Γ ⊢nv bavNV t ≡ bavNV u
congpair : {Γ : Ctx} {σ₁ σ₂ : Ty} {t t' : Γ ⊢nv σ₁} {u u' : Γ ⊢nv σ₂} → Γ ⊢nv t ≡ t' → Γ ⊢nv u ≡ u'
        → Γ ⊢nv pairNV t u ≡ pairNV t' u'
conglam : {Γ : Ctx} {σ τ : Ty} {t u : (Γ :: σ) ⊢np τ} → (Γ :: σ) ⊢np t ≡ u → Γ ⊢nv lamNV t ≡ lamNV u
congfun : {Γ : Ctx} {t t' u u' : Γ ⊢nv β} → Γ ⊢nv t ≡ t' → Γ ⊢nv u ≡ u' → Γ ⊢nv f t u ≡ f t' u'
```

In addition, the sets of equations for normal values and normal producers are extended with equations from specific value and effect theories. Similarly to Section 4.3, every equation $\Gamma \vdash \mathsf{t} \equiv \mathsf{u}$ defines an equation (extend-valctx Γ) ⊢nv (extend-val-nv t) ≡ (extend-val-nv u) between normal values. In addition, given an $\mathrm{FGCBV}_{\mathrm{eff}}$ type σ, every equation $\Gamma\,, \Delta \vdash \mathsf{E}\ \mathsf{t} \equiv \mathsf{u}$ defines an equation (extend-valctx Γ) @ (extend-effctx σ Δ) ⊢np (extend-eff-np σ t) ≡ (extend-eff-np σ u) between normal producers. For example, for the theory of non-determinism we add the following three equations together with a straightforward congruence equation for or.

```
or-idempotency : {Γ : Ctx} {σ : Ty} {t : Γ ⊢np σ} → Γ ⊢np or t t ≡ t
or-commutativity : {Γ : Ctx} {σ : Ty} {t u : Γ ⊢np σ} → Γ ⊢np or t u ≡ or u t
or-associativity : {Γ : Ctx} {σ : Ty} {t u v : Γ ⊢np σ} → Γ ⊢np or (or t u) v ≡ or t (or u v)
```

5.3 Denotational semantics

In this section, we define a suitable denotational semantics for $\mathrm{FGCBV}_{\mathrm{eff}}$ needed for our NBE algorithm. We will give a residualizing interpretation into a suitable and rather intensional monad model. By residualizing (notion borrowed from Filinski [13]) we mean an interpretation that preserves enough syntactic structure of $\mathrm{FGCBV}_{\mathrm{eff}}$ terms to construct the inverse map. For example, let us consider the following two terms.

$$(i) \quad \Gamma \vdash_{np} \lambda\,(f, g)\,.\,(f\star)\,\mathbf{to}\,x.(g\star) : (1 \rightharpoonup \alpha) \times (1 \rightharpoonup \alpha)$$
$$(ii) \quad \Gamma \vdash_{np} \lambda\,(f, g)\,.\,(g\star)\,\mathbf{to}\,x.(f\star) : (1 \rightharpoonup \alpha) \times (1 \rightharpoonup \alpha)$$

Both (i) and (ii) contain two atomic producers $(f\star)$ and $(g\star)$. As f and g are variables, we have no further information which effects $(f\star)$ and $(g\star)$ might produce. Therefore, the normal forms of (i) and (ii) must be different and this needs to be made explicit in the denotational semantics.

5.3.1 Free monad

We now construct a free monad on the signature endofunctor F_{np} we discussed in Section 5.1. This construction gives us the intensional monad model suitable for preserving enough

structure of atomic producers and operations to invert the interpretation. Later, in Section 5.5, we show how this free monad also forms a quotient monad satisfying the given effect theory.

We present the free monad as a Kleisli triple $(T, \eta, \_^*)$. The *object map* T is defined as an inductive data type to emphasize the monad's tree-like nature

```
data T (X : Ctx → Set) : Ctx → Set where
  T-return : {Γ : Ctx} → X Γ → T X Γ
  T-to : {Γ : Ctx} {σ : Ty} → Γ ⊢ap σ → T X (Γ :: σ) → T X Γ
  T-op : {Γ : Ctx} → Γ ⊢nv β → T X Γ → T X Γ → T X Γ
```

together with the *action of renaming* on it.

```
T-rename : {X : Set^Ctx} {Γ Γ' : Ctx} → (f : Ren Γ Γ') → T (set X) Γ → T (set X) Γ'
T-rename {X} f (T-return x) = T-return ((Set^Ctx.act X) f x)
T-rename {X} f (T-to x y) = T-to (⊢ap-rename f x) (T-rename (wk₂ f) y)
T-rename {X} f (T-op p x y) = T-op (⊢nv-rename f p) (T-rename f x) (T-rename f y)
```

The intuition behind this free monad is that it defines a semantic computation tree where the nodes are denoted with T-to's and T-op's and the leafs with T-return's.

Proposition 5.3.1. Given a presheaf X : Setˆ Ctx, the object map T and the action of renaming T-rename define a new presheaf T-Setˆ Ctx X. □

The *unit* and the *Kleisli extension* can now be defined as maps between presheaves.

```
η : {X : Set^Ctx} → Set^C-Map X (T-Set^Ctx X)
η X = T-return X


_* : {X Y : Set^Ctx} → (Set^C-Map X (T-Set^Ctx Y)) → Set^C-Map (T-Set^Ctx X) (T-Set^Ctx Y)
f* (T-return x) = f x
f* (T-to t x) = T-to t (f* x)
f* (T-op p x y) = T-op p (f* x) (f* y)
```

To complete the definition of our monad model, we have to define both the *strength* and *Kleisli exponentials* for T. In Agda, we write t-r for the strength and _⇒_ for the components of Kleisli exponentials with $\varepsilon_{X,Y}$ given by Agda function application. The *strength* of T is defined as a map between presheaves

```
t-r : {X Y : Set^Ctx} → Set^C-Map (X ⊗ (T-Set^Ctx Y)) (T-Set^Ctx (X ⊗ Y))
t-r (x , T-return y) = T-return (x , y)
t-r (x , T-to t y) = T-to t (t-r ((act X wk₁ x) , y))
t-r (x , T-op p y z) = T-op p (t-r (x , y)) (t-r (x , z))
```

while the components of Kleisli exponentials between presheaves X, Y : Setˆ Ctx are defined as Agda function spaces

```
_⇒_ : (X Y : Ctx → Set) → Ctx → Set
(X ⇒ Y) Γ = {Γ' : Ctx} → Ren Γ Γ' → X Γ' → T Y Γ'
```

together with the corresponding action of renaming.

```
⇒-rename : {X Y : Set^Ctx} {Γ Γ' : Ctx} → (f : Ren Γ Γ') → ((set X) ⇒ (set Y)) Γ → ((set X) ⇒ (set Y)) Γ'
⇒-rename f x g = x (comp-ren g f)
```

5.3.2 Residualizing interpetation

Following the definition given in Section 3.7, we now define the residualizing interpretation into our monad model on $\mathbf{Set}^{\mathbf{Ctx}}$. First, we define the *residualizing interpretation* $[\![\_]\!]$ *of* $FGCBV_{\text{eff}}$ *types* in $\mathbf{Set}^{\mathbf{Ctx}}$ given by the components

```
⟦_⟧ : Ty → Ctx → Set
⟦ α ⟧ Γ = Γ ⊢nv α
⟦ β ⟧ Γ = Γ ⊢nv β
⟦ One ⟧ Γ = Unit
⟦ σ₁ ∧ σ₂ ⟧ Γ = ⟦ σ₁ ⟧ Γ × ⟦ σ₂ ⟧ Γ
⟦ σ ⇀ τ ⟧ Γ = (⟦ σ ⟧ ⇒ ⟦ τ ⟧) Γ
```

together with the action of renaming on them.

```
⟦⟧-rename : {σ : Ty} {Γ Γ' : Ctx} → Ren Γ Γ' → ⟦ σ ⟧ Γ → ⟦ σ ⟧ Γ'
⟦⟧-rename {α} f t = ⊢nv-rename f t
⟦⟧-rename {β} f t = ⊢nv-rename f t
⟦⟧-rename {One} f t = ⋆
⟦⟧-rename {σ₁ ∧ σ₂} f p = ⟦⟧-rename f (fst p) , ⟦⟧-rename f (snd p)
⟦⟧-rename {σ ⇀ τ} f h = λ g d → h (g · f) d
```

Definition 5.3.2. We write Denot σ for the presheaves given by $[\![\sigma]\!]$ and ⟦⟧-rename. In addition, we write T-Denot σ for the presheaves given by T-Setˆ Ctx (Denot σ).

This interpretation of types illustrates how syntactic structure is stored in the semantics by interpreting base types as presheaves of normal values of base type. This extends the ideas in the previous section where we used the free monad to store syntactic structure of producer terms.

Next, we extend the interpretation of types to the interpretation of contexts. We do this by defining a notion of *environments* that map de Bruijn indices to semantic values

```
Env : Ctx → Ctx → Set
Env Γ Γ' = {σ : Ty} → σ ∈ Γ → ⟦ σ ⟧ Γ'
```

together with the action of renaming on them

```
env-rename : {Γ Γ' Γ'' : Ctx} → Ren Γ' Γ'' → Env Γ Γ' → Env Γ Γ''
env-rename f e x = ⟦⟧-rename f (e x)
```

These environments correspond to the interpretation of contexts $\llbracket\, \Gamma \,\rrbracket$ we discussed in Section 3.7. We decided to model the environments using two contexts to keep the presentation as symmetric as possible with the definition of parallel substitutions. Furthermore, environments can also be described by presheaves.

Definition 5.3.3. We write Env-Denot Γ for the presheaves given by Env Γ and env-rename.

Similarly to substitutions, one often needs to extend environments with new semantic values to interpret terms with free variables. This *extension* operation is defined, similarly to the extension of substitutions, by mapping a given semantic value to a new outermost variable in the given context.

```
env-extend : {Γ Γ' : Ctx} {σ : Ty} → Env Γ Γ' → ⟦ σ ⟧ Γ' → Env (Γ :: σ) Γ'
env-extend e d Hd = d
env-extend e d (Tl x) = e x
```

Finally, we define the *residualizing interpretation* of $\mathrm{FGCBV}_{\mathrm{eff}}$ value and producer terms in $\mathbf{Set}^{\mathbf{Ctx}}$ by following the interpretation in Section 3.7. The two interpretation maps now take both value and producer terms to morphisms in $\mathbf{Set}^{\mathbf{Ctx}}$.

```
⟦_⟧v : {Γ : Ctx} {σ : Ty} → Γ ⊢v σ → Set^Ctx-Map (Env-Denot Γ) (Denot σ)
⟦_⟧p : {Γ : Ctx} {σ : Ty} → Γ ⊢p σ → Set^Ctx-Map (Env-Denot Γ) (Denot-T σ)
```

These maps are defined by straightforward structural recursion on value and producer terms.

```
⟦ var x ⟧v e = e x
⟦ proj₁ t ⟧v e = fst (⟦ t ⟧v e)
⟦ proj₂ t ⟧v e = snd (⟦ t ⟧v e)
⟦ pair t u ⟧v e = ⟦ t ⟧v e , ⟦ u ⟧v e
⟦ ⋆ ⟧v e = ⋆
⟦ lam t ⟧v e = λ f u → ⟦ t ⟧p (env-extend (env-rename f e) u)
⟦ f t u ⟧v e = fNV (⟦ t ⟧v e) (⟦ u ⟧v e)
⟦ return t ⟧p e = η (⟦ t ⟧v e)
⟦ t to u ⟧p e = (λ v → ⟦ u ⟧p (env-extend (fst v) (snd v)))* (t-r (e , ⟦ t ⟧p e))
```

```
⟦ app t u ⟧p e = (⟦ t ⟧v e) id (⟦ u ⟧v e)
⟦ op p t u ⟧p e = T-op (⟦ p ⟧v e) (⟦ t ⟧p e) (⟦ u ⟧p e)
```

5.4 The normalization algorithm

Before we are able to define our normalization algorithm, we first need to invert the interpretation maps given in the previous section. This is done by defining the following families of maps between presheaves.

First, we define two families of *reification maps* (one for values and one for producers)

```
reify-v : {σ : Ty} → Set^Ctx-Map (Denot σ) (NVTerms σ)
reify-p : {σ : Ty} → Set^Ctx-Map (T-Denot σ) (NPTerms σ)
```

from semantic values to normal forms. We define these maps by structural recursion on types and the monad structure such that they extract the normal forms directly from the denotational semantics.

```
reify-v {α} d = d
reify-v {β} d = d
reify-v {One} d = unitNV
reify-v {σ₁ ∧ σ₂} d = pairNV (reify-v (fst d)) (reify-v (snd d))
reify-v {σ ⇁ τ} d = lamNV (reify-p (d wk₁ (reflect-v (varAV Hd))))
reify-p (T-return d) = returnNP (reify-v d)
reify-p (T-to t d) = toNP t (reify-p d)
reify-p (T-op p d d') = opNP p (reify-p d) (reify-p d')
```

reify-v gives us η-expanded β-normal forms according to the following intuition.

- We identify semantic and normal values of *base type* to extract their syntactic structure from the semantics.
- For semantic values of *unit type*, reification just returns the unique normal form of unit type.
- Semantic values of *product type* are expanded into syntactic pairs of reified components of these semantic values.
- Semantic values of *function type* are reified as normal lambda abstractions using reify-p to reify the application under the lambda abstraction. Notice that we also have to use reflect-v when reifying the application. The reason is that the normal lambda abstraction introduces syntactic variables that need to be reflected into the semantics before reifying the semantic application.

- On the other hand, reify-p takes us from the free monad structure to the normal producers by unrolling the tree-like structure in a straightforward way.

Next, we define two families of *reflection maps* (one for values and one for producers)

```
reflect-v : {σ : Ty} → Set^Ctx-Map (AVTerms σ) (Denot σ)
reflect-p : {σ : Ty} → Set^Ctx-Map (APTerms σ) (T-Denot σ)
```

going in the other direction from atomic forms to semantic values. We define these maps by structural recursion on types

```
reflect-v {α} t = avNV t
reflect-v {β} t = bavNV t
reflect-v {One} t = ⋆
reflect-v {σ₁ ∧ σ₂} t = reflect-v (proj₁AV t) , reflect-v (proj₂AV t)
reflect-v {σ ⇀ τ} t = λ f v → reflect-p (appAP (⊢av-rename f t) (reify-v v))
reflect-p t = T-to t (T-return (reflect-v (varAV Hd)))
```

such that they enable us to preserve enough syntactic structure of atomic forms in the semantics with the following intuition.

- Atomic values of *base types* are reflected as normal values of base types to explicitly mark where the variables (or projections of them) appeared in the syntax.
- Atomic values of *unit type* are reflected as the unique semantic value of unit type.
- Atomic values of *product type* are reflected as pairs of semantic values whose components are reflections of projections of these atomic values.
- Atomic values of *function type* are reflected as Kleisli exponentials. However, we first need to reify the semantic value given by the Kleisli exponential before reflecting the syntactic application.
- On the other hand, reflect-p preserves the atomic producers as instances of the free monad structure, i.e., T-to.

We can extend the family of reflection maps for atomic values to the reflection of contexts by reflecting every variable in a given context. We call this construction the *identity environment* to emphasize its intensional nature in representing syntactic variables in the semantics. These identity environments have a straightforward definition in Agda

```
id-env : {Γ : Ctx} → Env Γ Γ
id-env x = reflect-v (varAV x)
```

together with the following proposition showing how interpretations using identity environments respect normal and atomic forms.

Proposition 5.4.1. Given an identity environment, the reification map inverts the interpretation of normal forms

(i) $\forall$ t $\in$ ($\Gamma \vdash$nv σ) . reify-v ($[\![\vdash$nv-embed t $]\!]$v id-env) $\cong$ t

(ii) $\forall$ t $\in$ ($\Gamma \vdash$np σ) . reify-p ($[\![\vdash$np-embed t $]\!]$p id-env) $\cong$ t

and the reflection of atomic terms is equal to their interpretation.

(iii) $\forall$ t $\in$ ($\Gamma \vdash$av σ) . $[\![\vdash$av-embed t $]\!]$v id-env $\cong$ reflect-v t

(iv) $\forall$ t $\in$ ($\Gamma \vdash$ap σ) . $[\![\vdash$ap-embed t $]\!]$p id-env $\cong$ reflect-v t □

We outlined in Section 2.3 that the *NBE algorithm* can now be given by a straightforward composition of the interpretation and reification maps.

```
nf-v : {σ : Ty} → Set^Ctx-Map (VTerms σ) (NVTerms σ)
nf-v t = reify-v ([[ t ]]v id-env)

nf-p : {σ : Ty} → Set^Ctx-Map (PTerms σ) (NPTerms σ)
nf-p t = reify-p ([[ t ]]p id-env)
```

We will now spend the rest of this chapter on proving that this normalization algorithm is correct and satisfies the conditions outlined in Section 2.3. All the results we present can be found in full detail in our Agda formalization [1].

5.5 Kripke logical relations

Before proving the correctness of the NBE algorithm, it is necessary to formally relate the syntax of $\text{FGCBV}_{\text{eff}}$ and the monad model we have defined. In particular, we use the notion of Kripke logical relations to define suitable relations between $\text{FGCBV}_{\text{eff}}$ terms and semantic values. These logical relations were pioneered by Plotkin [34, 35] to characterize definable elements in the models of simply typed lambda calculi. Later, Jung and Tiuryn [19] generalized the Kripke logical relations from fixed arities to varying arities. This has been used by Fiore [14] to analyze NBE. However, in our work, it is sufficient to use binary Kripke logical relations as defined below.

Definition 5.5.1. A *world structure* $(W, \leq)$ consists of a collection of worlds W and an alternativeness relation (a preorder) $\leq$ between worlds.

It is worthwhile to notice that in our work, the world structure is given by the category of contexts. More precisely, we consider contexts as worlds and the injective renamings

as the relations between them. Using this observation, it is straightforward to define a suitable notion of Kripke relations between presheaves in $\mathbf{Set}^{\mathbf{Ctx}}$.

Definition 5.5.2. Given $X, Y \in \mathbf{Set}^{\mathbf{Ctx}}$, a *binary Kripke logical relation* $\sim$ is given on the components of X and Y by

- $\sim_\Gamma \in X_\Gamma \times Y_\Gamma$

satisfying a monotonicity condition.

- $\forall$ f $\in$ (Ren Γ Γ') . x $\sim_\Gamma$ y $\Longrightarrow$ act f x $\sim_{\Gamma'}$ act f y

Next, we relate the term model given by the initial algebra for the signature endofunctor $\langle F_v, F_p \rangle$ in Section 3.4 with our monad model.

Definition 5.5.3. The Kripke logical relations

- $\sim_v^{\Gamma;\sigma} \in (\Gamma \vdash \mathsf{v}\ \sigma) \times (\llbracket\ \sigma\ \rrbracket\ \Gamma)$
- $\sim_p^{\Gamma;\sigma} \in (\Gamma \vdash \mathsf{p}\ \sigma) \times (\mathsf{T}\ \llbracket\ \sigma\ \rrbracket\ \Gamma)$

relate value terms with semantic values of same type

- t $\sim_v^{\Gamma;\alpha}$ d $\Longleftrightarrow$ $\Gamma \vdash$v t $\equiv \vdash$nv-embed d
- t $\sim_v^{\Gamma;\mathrm{One}}$ d is always true
- t $\sim_v^{\Gamma;\sigma_1 \wedge \sigma_2}$ d $\Longleftrightarrow$ (proj$_1$ t) $\sim_v^{\Gamma;\sigma_1}$ (fst d) && (proj$_2$ t) $\sim_v^{\Gamma;\sigma_1}$ (snd d)
- t $\sim_v^{\Gamma;\sigma \rightarrow \tau}$ d $\Longleftrightarrow$ $\forall$ f $\in$ (Ren Γ Γ'), u $\in$ (Γ' $\vdash$v σ), d' $\in$ ($\llbracket\ \sigma\ \rrbracket$ Γ') . u $\sim_v^{\Gamma';\sigma}$ d' $\Longrightarrow$ $\Longrightarrow$ (app ($\vdash$v-rename f t) u)) $\sim_p^{\Gamma';\tau}$ (d f d')

and producer terms with the corresponding free monad structure.

- t $\sim_p^{\Gamma;\sigma}$ (T-return d) $\Longleftrightarrow$ $\exists$ u $\in$ ($\Gamma \vdash$v σ) . $\Gamma \vdash$p t $\equiv$ return u && u $\sim_v^{\Gamma;\sigma}$ d
- t $\sim_p^{\Gamma;\sigma}$ (T-to u d) $\Longleftrightarrow$ $\exists$ v $\in$ ((Γ :: τ) $\vdash$p σ) . $\Gamma \vdash$p t $\equiv$ ($\vdash$ap-embed u) to v && v $\sim_p^{\Gamma :: \sigma;\tau}$ d
- t $\sim_p^{\Gamma;\sigma}$ (T-op p d d') $\Longleftrightarrow$ $\exists$ u,v $\in$ ($\Gamma \vdash$p σ) . $\Gamma \vdash$p t $\equiv$ or ($\vdash$nv-embed p) u v && u $\sim_p^{\Gamma;\sigma}$ d && v $\sim_p^{\Gamma;\sigma}$ d'

The intuition behind this definition can be summarized as follows.

- A value term of *base type* is related to a semantic value of base type (i.e., a normal form of base type) if and only if they are provably equal as value terms.
- A value term of *unit type* is always related to the unique semantic value of unit type.

- A value term of *product type* is related to a semantic value of product type if and only if both projections are related.
- A value term of *function type* is related to a semantic value of function type if and only if, given related value term and semantic value in some future context, the applications are related in this future context.
- The relations relating *producer terms* with the free monad structure follow a common pattern. That is, a producer term is related to the monad if and only if the term is provably equal to a producer term corresponding to the free monad structure (e.g., op p t u and T-op p d d') and the relations hold for the corresponding substructures.

Proposition 5.5.4 (Monotonicity of $\sim_v^{\Gamma;\sigma}$ and $\sim_p^{\Gamma;\sigma}$)**.** The relations $\sim_v^{\Gamma;\sigma}$ and $\sim_p^{\Gamma;\sigma}$ are *monotone* under renamings.

(i) $\forall$ t $\in$ ($\Gamma \vdash$v σ), d $\in$ ($[\![\sigma]\!]$ Γ), f $\in$ (Ren Γ Γ') . t $\sim_v^{\Gamma;\sigma}$ d $\Longrightarrow$
(⊢v-rename f t) $\sim_v^{\Gamma';\sigma}$ ([[]]-rename f d)

(ii) $\forall$ t $\in$ ($\Gamma \vdash$p σ), d $\in$ (T $[\![\sigma]\!]$ Γ), f $\in$ (Ren Γ Γ') . t $\sim_p^{\Gamma;\sigma}$ d $\Longrightarrow$
(⊢p-rename f t) $\sim_p^{\Gamma';\sigma}$ (T-rename f d)

Proof. We prove this result by simultaneous structural induction on (i) types and (ii) the free monad structure.

- The base cases for base types are proved by using the naturality of ⊢nv-embed and the monotonicity of $\Gamma \vdash$v t $\equiv$ u under renamings.
- The inductive cases for unit and product types are proved by straightforward application of induction hypotheses.
- The inductive case for function types is proved by using (ii).
- The base case for T-return is proved by using (i).
- The inductive cases for T-to and T-op are proved by using the naturality of ⊢np-embed and the monotonicity of $\Gamma \vdash$p t $\equiv$ u under renamings.

□

In addition to satisfying the monotonicity condition, these relations exhibit another interesting invariance property. More precisely, the terms in the same equivalence class of the term model, given by the equational theories $\Gamma \vdash$v t $\equiv$ u and $\Gamma \vdash$p t $\equiv$ u, should be related to the same semantic values. This result will be important for showing that terms are provably equal to their normal forms.

Proposition 5.5.5 (Invariance under provable equality)**.** The relations $\sim_v^{\Gamma;\sigma}$ and $\sim_p^{\Gamma;\sigma}$ are *invariant* under provable equality.

(i) $\forall\ \mathsf{t}, \mathsf{u} \in (\Gamma \vdash \mathsf{v}\ \sigma),\ \mathsf{d} \in (\llbracket\ \sigma\ \rrbracket\ \Gamma)\ .\ \mathsf{t} \sim_v^{\Gamma;\sigma} \mathsf{d}\ \&\&\ \Gamma \vdash \mathsf{v}\ \mathsf{t} \equiv \mathsf{u} \Longrightarrow \mathsf{u} \sim_v^{\Gamma;\sigma} \mathsf{d}$

(ii) $\forall\ \mathsf{t}, \mathsf{u} \in (\Gamma \vdash \mathsf{p}\ \sigma),\ \mathsf{d} \in (\mathsf{T}\ \llbracket\ \sigma\ \rrbracket\ \Gamma)\ .\ \mathsf{t} \sim_p^{\Gamma;\sigma} \mathsf{d}\ \&\&\ \Gamma \vdash \mathsf{p}\ \mathsf{t} \equiv \mathsf{u} \Longrightarrow \mathsf{u} \sim_p^{\Gamma;\sigma} \mathsf{d}$

Proof. We prove these results by simultaneous structural induction on (i) types and (ii) the free monad structure.

- The base cases for base types are proved by using the provable equality of value terms and the definition of $\sim_v^{\Gamma;\sigma}$ for base types.
- The base case for T-return is proved by using (i).
- The inductive cases are proved by straightforward application of induction hypotheses with the function type case using (ii). □

It is straightforward to extend these Kripke logical relations to interpretations into the term model and the monad model given respectively by the action of parallel substitution and the residualizing interpretation.

Definition 5.5.6. Substitutions and environments are related by a Kripke logical relation

- $\sim_e^{\Gamma;\Gamma'} \in$ (Sub Γ Γ') × (Env Γ Γ')

if and only if they are related at each single variable.

- $\mathsf{s} \sim_e^{\Gamma;\Gamma'} \mathsf{e} \Longleftrightarrow \forall\ \mathsf{x} \in (\sigma \in \Gamma)\ .$ (s x) $\sim_v^{\Gamma';\sigma}$ (e x)

It is straightforward to verify that, due to Proposition 5.5.4, $\sim_e^{\Gamma;\Gamma'}$ is also monotone under renamings. However, it is more interesting to investigate whether $\sim_v^{\Gamma;\sigma}$ and $\sim_p^{\Gamma;\sigma}$ are compatible with the reification and reflection maps.

Proposition 5.5.7 (Compatibility)**.** $\sim_v^{\Gamma;\sigma}$ and $\sim_p^{\Gamma;\sigma}$ are compatible with the reification and reflection maps.

(i) $\forall\ \mathsf{t} \in (\Gamma \vdash \mathsf{av}\ \sigma)\ .$ (⊢av-embed t) $\sim_v^{\Gamma;\sigma}$ (reflect-v t)

(ii) $\forall\ \mathsf{t} \in (\Gamma \vdash \mathsf{ap}\ \sigma)\ .$ (⊢ap-embed t) $\sim_p^{\Gamma;\sigma}$ (reflect-p t)

(iii) $\forall\ \mathsf{t} \in (\Gamma \vdash \mathsf{v}\ \sigma),\ \mathsf{d} \in (\llbracket\ \sigma\ \rrbracket\ \Gamma)\ .\ \mathsf{t} \sim_v^{\Gamma;\sigma} \mathsf{d} \Longrightarrow \Gamma \vdash \mathsf{v}\ \mathsf{t} \equiv$ (⊢nv-embed (reify-v d))

(iv) $\forall\ \mathsf{t} \in (\Gamma \vdash \mathsf{p}\ \sigma),\ \mathsf{d} \in (\mathsf{T}\ \llbracket\ \sigma\ \rrbracket\ \Gamma)\ .\ \mathsf{t} \sim_p^{\Gamma;\sigma} \mathsf{d} \Longrightarrow \Gamma \vdash \mathsf{p}\ \mathsf{t} \equiv$ (⊢np-embed (reify-p d))

Proof. We prove these results by simultaneous structural induction on (i,iii) types and (ii,iv) the free monad structure.

- The base cases in (i) for base types are proved by using the definition of reflect-v.
- The base case in (ii) for T-return is proved by using (i).

- The base cases in (iii) for base types are proved by applying the given relation.
- The base case in (iv) for T-return is proved by using (iii).
- The inductive cases are proved by straightforward application of induction hypotheses. □

Corollary 5.5.8 (Identity substitutions and environments)**.** It follows from Proposition 5.5.7 (i) that the identity substitutions id-subst and identity environments id-env are related by $\sim_e^{\Gamma;\Gamma'}$. □

We conclude this section by presenting the fundamental lemma of logical relations that will be used in Section 5.7 for proving the normalization results.

Theorem 5.5.9 (Fundamental lemma of logical relations)**.** Interpretations of value and producer terms using related environments are related.

(i) $\forall$ t $\in$ ($\Gamma \vdash$v σ), s $\in$ (Sub Γ Γ'), e $\in$ (Env Γ Γ') . s $\sim_e^{\Gamma;\Gamma'}$ e $\Longrightarrow$
(subst-v s t) $\sim_v^{\Gamma';\sigma}$ ($[\![$ t $]\!]$v e)

(ii) $\forall$ t $\in$ ($\Gamma \vdash$p σ), s $\in$ (Sub Γ Γ'), e $\in$ (Env Γ Γ') . s $\sim_e^{\Gamma;\Gamma'}$ e $\Longrightarrow$
(subst-p s t) $\sim_v^{\Gamma';\sigma}$ ($[\![$ t $]\!]$p e)

Proof. We prove (i) and (ii) by straightforward structural induction on value and producer terms.

- The base case in (i) for variables is proved by using the given relation between substitutions and environments.
- The base case in (ii) for return is proved by using (i).
- The inductive cases are proved by straightforward application of induction hypotheses.
- The only non-trivial case concerns the sequenced producers t to u which is proved by using Proposition 5.5.10. □

Proposition 5.5.10. The logical relation $\sim_p^{\Gamma;\sigma}$ relates sequenced producers with the composition of Kleisli extensions and monad strength.

(i) $\forall$ t $\in$ ($\Gamma \vdash$p σ), u $\in$ ((Γ :: σ) $\vdash$p τ), d $\in$ (T $[\![$ σ $]\!]$ Γ'), s $\in$ (Sub Γ Γ'),
e $\in$ (Env Γ Γ') . t $\sim_p^{\Gamma';\sigma}$ d && s $\sim_e^{\Gamma;\Gamma'}$ e $\Longrightarrow$
(t to subst-p (lift s) u) $\sim_p^{\Gamma';\sigma}$ (λ v $\rightarrow$ $[\![$ u $]\!]$p (env-extend (fst v) (snd v)))* (t-r (e , d))

Proof. We prove this proposition by structural induction on d simultaneously with proving

the fundamental lemmas in Theorem 5.5.9. The proof also uses the invariance results from Proposition 5.5.5 for T-return and T-to. □

5.6 Soundness of the residualizing interpretation

In this section, we prove that the residualizing interpretation is sound. However, it is important to note that we consider a more general notion of soundness than the usual presentations of NBE. More precisely, we do not consider soundness up to equality but instead up to suitable Kripke logical relations we define below. This is necessary for showing that the normal forms returned by our normalization algorithm are equivalent modulo the given value and effect theories. However, we also show that the residualizing interpretation is sound up to equality if there are no equations in the value and effect theories.

We now define two partial equivalence relations on semantic values and the free monad structure showing that they are form Kripke logical relations.

Definition 5.6.1. We turn the free monad into a quotient monad by equipping it with a partial equivalence relation

- $\approx_T^{\Gamma;\sigma} \in (\mathsf{T}\ [\![\ \sigma\ ]\!]\ \Gamma) \times (\mathsf{T}\ [\![\ \sigma\ ]\!]\ \Gamma)$

generated by the following rules.

- (sym) $\forall$ d,d' $\in (\mathsf{T}\ [\![\ \sigma\ ]\!]\ \Gamma)$. d $\approx_T^{\Gamma;\sigma}$ d' $\Longrightarrow$ d' $\approx_T^{\Gamma;\sigma}$ d
- (trans) $\forall$ d,d',d" $\in (\mathsf{T}\ [\![\ \sigma\ ]\!]\ \Gamma)$. d $\approx_T^{\Gamma;\sigma}$ d' && d' $\approx_T^{\Gamma;\sigma}$ d" $\Longrightarrow$ d $\approx_T^{\Gamma;\sigma}$ d"
- (cong return) $\forall$ d,d' $\in ([\![\ \sigma\ ]\!]\ \Gamma)$. d $\approx^{\Gamma;\sigma}$ d' $\Longrightarrow$ (T-return d) $\approx_T^{\Gamma;\sigma}$ (T-return d')
- (cong to) $\forall$ t,u $\in (\Gamma \vdash \mathsf{ap}\ \tau)$, d,d' $\in (\mathsf{T}\ [\![\ \sigma\ ]\!]\ (\Gamma :: \tau))$. $\Gamma \vdash$ ap t $\equiv$ u && d $\approx_T^{\Gamma::\tau;\sigma}$ d' $\Longrightarrow$ (T-to t d) $\approx_T^{\Gamma;\sigma}$ (T-to u d')
- (cong op) $\forall$ d,d',d",d"' $\in (\mathsf{T}\ [\![\ \sigma\ ]\!]\ \Gamma)$, p,p' $\in ([\![\ \beta\ ]\!]\ \Gamma)$. d $\approx_T^{\Gamma;\sigma}$ d' && d" $\approx_T^{\Gamma;\sigma}$ d"' && $\Gamma \vdash$nv p $\equiv$ p' $\Longrightarrow$ (T-op p d d") $\approx_T^{\Gamma;\sigma}$ (T-op p' d' d"')
- and the rules representing the equations of specific value and effect theories together with straightforward rules of congruence

It is easy to see that,because the free monad carries the structure of producer terms, these rules are reminiscent of the equational theory (without $\beta\eta$-equations) of producer terms presented in Section 3.6. However, due to partiality, we have to make additional assumptions about arguments being related. In addition, this relation is parametrized over another relation $\approx^{\Gamma;\sigma}$ to relate semantic values under T-return.

Definition 5.6.2. The *partial equivalence relation* $\approx^{\Gamma;\sigma}$ on semantic values

- $\approx^{\Gamma;\sigma} \in (\llbracket\ \sigma\ \rrbracket\ \Gamma) \times (\llbracket\ \sigma\ \rrbracket\ \Gamma)$

relates semantic values of the same type.

- $\mathsf{d} \approx^{\Gamma;\alpha} \mathsf{d'} \iff \Gamma \vdash \mathsf{nv\ d} \equiv \mathsf{d'}$
- $\mathsf{d} \approx^{\Gamma;One} \mathsf{d'} \iff \mathsf{d} \cong \mathsf{d'}$
- $\mathsf{d} \approx^{\Gamma;\sigma_1 \wedge \sigma_2} \mathsf{d'} \iff \mathsf{(fst\ d)} \approx^{\Gamma;\sigma_1} \mathsf{(fst\ d')}\ \&\&\ \mathsf{(snd\ d)} \approx^{\Gamma;\sigma_1} \mathsf{(snd\ d')}$
- $\mathsf{d} \approx^{\Gamma;\sigma \to \tau} \mathsf{d'} \iff \forall\ \mathsf{f} \in \mathsf{(Ren\ \Gamma\ \Gamma')},\ \mathsf{d''}, \mathsf{d'''} \in (\llbracket\ \sigma\ \rrbracket\ \Gamma')\ .\ \mathsf{d''} \approx^{\Gamma';\sigma} \mathsf{d'''} \implies \mathsf{(d\ f\ d'')} \approx_T^{\Gamma';\tau} \mathsf{(d'\ f\ d''')}$

This definition is very similar to the relation $\sim_v^{\Gamma;\sigma}$ defined in Section 5.5.

- Semantic values of *base type* are related if and only if they are provably equal as normal forms.
- Semantic values of *unit type* are related if and only if they are equal.
- Semantic values of *product type* are related if and only if both projections are related.
- Semantic values of *function type* are related if and only if, given two related semantic values in some future context, the semantic applications are related in the future context.

It is now routine to show that $\approx^{\Gamma';\tau}$ and $\approx_T^{\Gamma';\tau}$ are Kripke logical relations satisfying the monotonicity condition.

Proposition 5.6.3 (Monotonicity)**.** The relations $\approx^{\Gamma';\tau}$ and $\approx_T^{\Gamma';\tau}$ are monotone under renamings.

(i) $\forall\ \mathsf{d,d'} \in (\llbracket\ \sigma\ \rrbracket\ \Gamma),\ \mathsf{f} \in \mathsf{(Ren\ \Gamma\ \Gamma')}\ .\ \mathsf{(}[\![]\!]\mathsf{\text{-}rename\ f\ d)} \approx^{\Gamma;\sigma} \mathsf{(}[\![]\!]\mathsf{\text{-}rename\ f\ d')}$

(ii) $\forall\ \mathsf{d,d'} \in \mathsf{(T}\ \llbracket\ \sigma\ \rrbracket\ \Gamma),\ \mathsf{f} \in \mathsf{(Ren\ \Gamma\ \Gamma')}\ .\ \mathsf{(T\text{-}rename\ f\ d)} \approx_T^{\Gamma;\sigma} \mathsf{(T\text{-}rename\ f\ d')}$

Proof. We prove these result by structural induction on (i,iii) the types and (ii,iv) the free monad structure. The proof follows similar structure to the proof of Proposition 5.5.4. □

Definition 5.6.4. Similarly to Definition 5.5.6, two environments are related by the Kripke logical relation

- $\approx_e^{\Gamma;\Gamma'} \in \mathsf{(Env\ \Gamma\ \Gamma')} \times \mathsf{(Env\ \Gamma\ \Gamma')}$

if and only if they are related at every single variable.

- $\mathsf{e} \approx_e^{\Gamma;\Gamma'} \mathsf{e'} \iff \forall\ \mathsf{x} \in (\sigma \in \Gamma)\ .\ (\mathsf{e\ x}) \approx^{\Gamma';\sigma} (\mathsf{e'\ x})$

It follows from Proposition 5.6.3 that $\approx_e^{\Gamma;\Gamma'}$ is also monotone under renamings. In addition, the Kripke logical relations $\approx^{\Gamma';\sigma}$ and $\approx_T^{\Gamma';\sigma}$ also satisfy the fundamental lemma of logical relations.

Theorem 5.6.5 (Fundamental lemma of logical relations)**.** Interpretations of value and producer terms using related environments are related.

(i) $\forall\ \mathsf{t} \in (\Gamma \vdash \mathsf{v}\ \sigma), \mathsf{e}, \mathsf{e'} \in (\mathsf{Env}\ \Gamma\ \Gamma')\ .\ \mathsf{e} \approx_e^{\Gamma;\Gamma'} \mathsf{e'} \implies (\llbracket\ \mathsf{t}\ \rrbracket \mathsf{v\ e}) \approx^{\Gamma';\sigma} (\llbracket\ \mathsf{t}\ \rrbracket \mathsf{v\ e'})$

(ii) $\forall\ \mathsf{t} \in (\Gamma \vdash \mathsf{p}\ \sigma), \mathsf{e}, \mathsf{e'} \in (\mathsf{Env}\ \Gamma\ \Gamma')\ .\ \mathsf{e} \approx_e^{\Gamma;\Gamma'} \mathsf{e'} \implies (\llbracket\ \mathsf{t}\ \rrbracket \mathsf{p\ e}) \approx_T^{\Gamma';\sigma} (\llbracket\ \mathsf{t}\ \rrbracket \mathsf{p\ e'})$

Proof. We prove the fundamental lemmas by structural induction on value and producer terms. Similarly to the proof of Theorem 5.5.9, the only non-trivial case concerns the sequenced producers which we prove by using Proposition 5.6.6. □

Proposition 5.6.6. The logical relation $\approx_T^{\Gamma;\sigma}$ relates compositions of Kleisli extensions and monad strength.

(i) $\forall\ \mathsf{t} \in (\mathsf{t} : \Gamma :: \sigma \vdash \mathsf{p}\ \tau), \mathsf{d}, \mathsf{d'} \in (\mathsf{T}\ \llbracket\ \sigma\ \rrbracket\ \Gamma'), \mathsf{e}, \mathsf{e'} \in (\mathsf{Env}\ \Gamma\ \Gamma')\ .\ \mathsf{d} \approx_T^{\Gamma';\sigma} \mathsf{d'}\ \&\&\ \mathsf{e} \approx_e^{\Gamma;\Gamma'} \mathsf{e'} \implies$
$(\lambda\ \mathsf{v} \to \llbracket\ \mathsf{t}\ \rrbracket \mathsf{p}\ (\mathsf{env\text{-}extend}\ (\mathsf{fst\ v})\ (\mathsf{snd\ v})))^*\ (\mathsf{t\text{-}r}\ (\mathsf{e}\ ,\ \mathsf{d})) \sim_p^{\Gamma';\sigma}$
$(\lambda\ \mathsf{v} \to \llbracket\ \mathsf{t}\ \rrbracket \mathsf{p}\ (\mathsf{env\text{-}extend}\ (\mathsf{fst\ v})\ (\mathsf{snd\ v})))^*\ (\mathsf{t\text{-}r}\ (\mathsf{e'}\ ,\ \mathsf{d'}))$

Proof. We prove this result by induction on the derivation of the partial equivalence relation $\mathsf{d} \approx_T^{\Gamma';\sigma} \mathsf{d'}$ simultaneously with proving the fundamental lemmas in Theorem 5.6.5. □

Following the previous section, it is interesting to investigate whether the reification and reflection maps are also compatible with these Kripke logical relations.

Proposition 5.6.7 (Compatibility)**.** $\approx^{\Gamma;\sigma}$ and $\approx_T^{\Gamma;\sigma}$ are compatible with the reification and reflection maps.

(i) $\forall\ \mathsf{d},\mathsf{d'} \in (\llbracket\ \sigma\ \rrbracket\ \Gamma)\ .\ \mathsf{d} \approx^{\Gamma;\sigma} \mathsf{d'} \implies \Gamma \vdash \mathsf{nv}\ (\mathsf{reify\text{-}v\ d}) \equiv (\mathsf{reify\text{-}v\ d'})$

(ii) $\forall\ \mathsf{d},\mathsf{d'} \in (\mathsf{T}\ \llbracket\ \sigma\ \rrbracket\ \Gamma)\ .\ \mathsf{d} \approx_T^{\Gamma;\sigma} \mathsf{d'} \implies \Gamma \vdash \mathsf{np}\ (\mathsf{reify\text{-}p\ d}) \equiv (\mathsf{reify\text{-}p\ d'})$

(iii) $\forall\ \mathsf{t},\mathsf{u} \in (\Gamma \vdash \mathsf{av}\ \sigma)\ .\ \Gamma \vdash \mathrm{av}\ \mathrm{t} \equiv \mathrm{u} \implies (\mathsf{reflect\text{-}v\ t}) \approx^{\Gamma;\sigma} (\mathsf{reflect\text{-}v\ u})$

(iv) $\forall\ \mathsf{t},\mathsf{u} \in (\Gamma \vdash \mathsf{ap}\ \sigma)\ .\ \Gamma \vdash \mathrm{ap}\ \mathrm{t} \equiv \mathrm{u} \implies (\mathsf{reflect\text{-}p\ t}) \approx_T^{\Gamma;\sigma} (\mathsf{reflect\text{-}p\ u})$

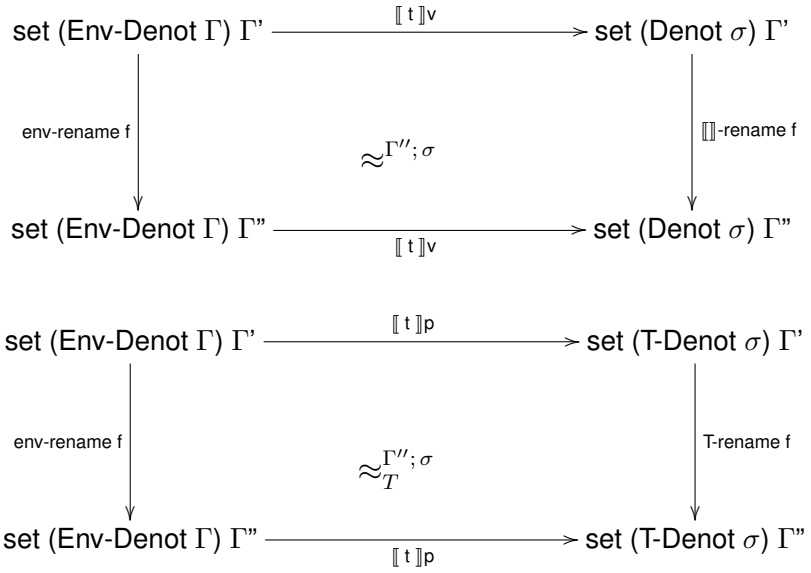
Proof. We prove these results by simultaneous induction (i, iii) on the structure of types and (ii, iv) on the derivation of the relations similarly to the proof of Proposition 5.5.7.

In addition, as the equivalence relation for atomic values is definitionally equal to the heterogeneous equality $\_ \cong \_$, the given hypothesis $\Gamma \vdash \mathsf{av}\ \mathsf{t} \equiv \mathsf{u}$ in (iii) unifies t and u. □

Corollary 5.6.8. It follows from Proposition 5.6.7 (iii) that the identity environments id-env are related by $\approx_e^{\Gamma;\Gamma'}$. □

In the soundness proof we present, we have to appeal to the naturality of the interpretation maps. However, as we identify semantic values up to the Kripke logical relations $\approx^{\Gamma;\sigma}$ and $\approx_T^{\Gamma;\sigma}$, we have to consider a more general naturality condition than given in Definition 3.4.2 of maps between presheaves.

Proposition 5.6.9. Given a value or producer term t, the resulting maps $[\![\ \mathsf{t}\ ]\!]\mathsf{v}$ and $[\![\ \mathsf{t}\ ]\!]\mathsf{p}$ are natural up to the Kripke logical relations $\approx^{\Gamma;\sigma}$ and $\approx_T^{\Gamma;\sigma}$. Explicitly this means that the following two naturality squares commute.



□

Finally, we conclude this section by proving the soundness theorem for the residualizing interpretation.

Theorem 5.6.10 (Soundness)**.** The residualizing interpretation maps $[\![\_]\!]\mathsf{v}$ and $[\![\_]\!]\mathsf{p}$ are sound up to the Kripke logical relations $\approx^{\Gamma;\sigma}$ and $\approx_T^{\Gamma;\sigma}$ with respect to the equational theories $\Gamma \vdash \mathsf{v}\ \mathsf{t} \equiv \mathsf{u}$ and $\Gamma \vdash \mathsf{p}\ \mathsf{t} \equiv \mathsf{u}$.

(i) $\forall$ t,u $\in (\Gamma \vdash \mathsf{v}\ \sigma)$, e,e' $\in$ (Env Γ Γ') . $\Gamma \vdash \mathsf{v}\ \mathsf{t} \equiv \mathsf{u}$ && e $\approx_e^{\Gamma;\Gamma'}$ e' $\Longrightarrow$
([[t]]v e) $\approx^{\Gamma';\sigma}$ ([[u]]v e')

(ii) $\forall$ t,u $\in (\Gamma \vdash \mathsf{p}\ \sigma)$, e,e' $\in$ (Env Γ Γ') . $\Gamma \vdash \mathsf{p}\ \mathsf{t} \equiv \mathsf{u}$ && e $\approx_e^{\Gamma;\Gamma'}$ e' $\Longrightarrow$

$(\llbracket\ \mathsf{t}\ \rrbracket\mathsf{p}\ \mathsf{e}) \approx_T^{\Gamma';\sigma} (\llbracket\ \mathsf{u}\ \rrbracket\mathsf{p}\ \mathsf{e'})$

Proof. We prove soundness by simultaneous induction on the derivations of $\Gamma \vdash \mathsf{v}\ \mathsf{t} \equiv \mathsf{u}$ and $\Gamma \vdash \mathsf{p}\ \mathsf{t} \equiv \mathsf{u}$. We only outline the most important inductive cases that also require the use of other results.

- conglam - by using induction hypothesis together with the monotonicity of $\approx_e^{\Gamma;\Gamma'}$
- $\eta\rightharpoonup$ - by using the naturality of $[\![\_]\!]\mathsf{v}$ from Proposition 5.6.9 and the fundamental lemma for $\approx^{\Gamma;\sigma}$ from Theorem 5.6.5
- congto - by using the induction hypothesis together with the following result about soundness of interpreting sequenced producers using Kleisli exponentials and monad strength

  (*) $\forall$ t,u $\in ((\Gamma :: \sigma) \vdash \mathsf{p}\ \tau)$, e,e' $\in$ (Env Γ Γ'), d,d' $\in$ (T $\llbracket\ \sigma\ \rrbracket$ Γ') .
  $(\Gamma :: \sigma) \vdash \mathsf{p}\ \mathsf{t} \equiv \mathsf{u}$ && d $\approx_T^{\Gamma';\sigma}$ d' && e $\approx_e^{\Gamma;\Gamma'}$ e' $\Longrightarrow$
  $(\lambda\ \mathsf{v} \to \llbracket\ \mathsf{t}\ \rrbracket\mathsf{p}$ (env-extend (fst v) (snd v)))* (t-r (e , d)) $\approx_T^{\Gamma';\tau}$
  $(\lambda\ \mathsf{v} \to \llbracket\ \mathsf{u}\ \rrbracket\mathsf{p}$ (env-extend (fst v) (snd v)))* (t-r (e' , d'))
- congreturn - by using the congruence rule for T-return in Definition 5.6.1 together with the soundness result (i)
- congop - by using the congruence rule for T-op in Definition 5.6.1
- $\beta\rightharpoonup$ - by using the fundamental lemmas for $\approx^{\Gamma;\sigma}$ and $\approx_T^{\Gamma;\sigma}$ from Theorem 5.6.5
- $\beta$to - by using the fundamental lemma for $\approx_T^{\Gamma;\sigma}$ from Theorem 5.6.5
- $\eta$to - by using the fundamental lemma for $\approx_T^{\Gamma;\sigma}$ from Theorem 5.6.5 and the following result about composing Kleisli extensions with the unit and strength of the monad

  (**) $\forall$ e,e' $\in$ (Env Γ Γ'), d,d' $\in$ (T $\llbracket\ \sigma\ \rrbracket$ Γ') . d $\approx_T^{\Gamma';\sigma}$ d' && e $\approx_e^{\Gamma;\Gamma'}$ e' $\Longrightarrow$
  $(\lambda\ \mathsf{v} \to \eta$ (snd v))* (t-r (e , d)) $\approx_T^{\Gamma';\sigma}$ d'
- opto - by using the fundamental lemma for $\approx_T^{\Gamma;\sigma}$ from Theorem 5.6.5 and the result about composing Kleisli extensions and strength from Proposition 5.6.6
- assocto - by using the fundamental lemma for $\approx_T^{\Gamma;\sigma}$ from Theorem 5.6.5 and the associativity result for composing Kleisli extensions and monad strength

  (***) $\forall$ t $\in (\Gamma :: \sigma' \vdash \mathsf{p}\ \sigma)$, u $\in ((\Gamma :: \sigma) \vdash \mathsf{p}\ \tau)$, e,e' $\in$ (Env Γ Γ'),
  d,d' $\in$ (T $\llbracket\ \sigma'\ \rrbracket$ Γ') . d $\approx_T^{\Gamma';\sigma'}$ d' && e $\approx_e^{\Gamma;\Gamma'}$ e' $\Longrightarrow$
  $(\lambda\ \mathsf{v} \to \llbracket\ \mathsf{u}\ \rrbracket\mathsf{p}$ (env-extend (fst v) (snd v)))*
  (t-r (e , (λ v' $\to \llbracket$ t $\rrbracket$p (env-extend (fst v) (snd v)))* (t-r (e , d))))
  $\approx_T^{\Gamma';\tau}$
  $(\lambda\ \mathsf{v} \to (\lambda\ \mathsf{v} \to \llbracket \vdash$p-rename exchange ($\vdash$p-rename wk_1 u) $\rrbracket$p

(env-extend (fst v)) (snd v)))*
(t-r ((env-extend (fst v') (snd v')) , ⟦ t ⟧p (env-extend (fst v) (snd v)))))*
(t-r (e' , d'))

□

Although this soundness result is proved up to the Kripke logical relations $\approx^{\Gamma;\sigma}$ and $\approx_T^{\Gamma;\sigma}$, it is possible to recover the usual notion of soundness up to equality if there are no equations in the value and effect theories.

Proposition 5.6.11. If the sets of equations E_{val} and E_{eff} are empty, the residualizing interpretations ⟦_⟧v and ⟦_⟧p are sound up to equality.

(i) ∀ t,u ∈ (Γ ⊢v σ), e ∈ (Env Γ Γ') . Γ ⊢v t ≡ u ⟹ (⟦ t ⟧v e) ≅ (⟦ u ⟧v e)

(ii) ∀ t,u ∈ (Γ ⊢p σ), e ∈ (Env Γ Γ') . Γ ⊢p t ≡ u ⟹ (⟦ t ⟧p e) ≅ (⟦ u ⟧p e)

Proof. Similarly to Theorem 5.6.10, this soundness result is again proved by simultaneous induction on the derivations of Γ ⊢v t ≡ u and Γ ⊢p t ≡ u but now using the usual notion of naturality. □

5.7 Correctness of the normalization algorithm

We conclude this chapter by proving the correctness of our NBE algorithm. We present the correctness results as the following four theorems. These results formalize the three conditions given in Section 2.3 together with showing that our normalization algorithm works modulo the given value and effect theories.

Theorem 5.7.1 (Preserving normal forms)**.** When applied to normal forms, the normalization function acts as an identity.

(i) ∀ t ∈ (Γ ⊢nv σ) . nf-v (⊢nv-embed t) ≅ t

(ii) ∀ t ∈ (Γ ⊢np σ) . nf-p (⊢np-embed t) ≅ t

Proof. We prove this theorem directly by using the definitions of nf-v and nf-p together with Proposition 5.4.1 (i)-(ii). □

Theorem 5.7.2 (Provable equality)**.** Every value and producer term is provably equal to its normal form.

(i) ∀ t ∈ (Γ ⊢v σ) . Γ ⊢v t ≡ ⊢nv-embed (nf-v t)

(ii) ∀ t ∈ (Γ ⊢p σ) . Γ ⊢p t ≡ ⊢np-embed (nf-p t)

Proof. We prove this theorem by using the fundamental lemmas of logical relations and the compatibility results from earlier.

- We first observe that the identity substitutions and identity environments are related by $\sim_e^{\Gamma;\Gamma'}$ (Corollary 5.5.8).
- We then use the fundamental lemma of Kripke logical relations between syntax and semantics from Theorem 5.5.9 on identity substitutions and identity environments and get (subst-v id-subst t) $\sim_v^{\Gamma;\sigma}$ ([[t]]v id-env) and (subst-p id-subst t) $\sim_p^{\Gamma;\sigma}$ ([[t]]p id-env).
- We show that subst-v id-subst t $\cong$ t and subst-p id-subst t $\cong$ t and get t $\sim_v^{\Gamma;\sigma}$ ([[t]]v id-env) and t $\sim_p^{\Gamma;\sigma}$ ([[t]]p id-env).
- Next, we apply Proposition 5.5.7 to these relations and get $\Gamma \vdash$v t $\equiv \vdash$nv-embed (reify-v ([[t]]v id-env)) and $\Gamma \vdash$p t $\equiv \vdash$np-embed (reify-p ([[t]]p id-env))
- Finally, we notice that the right hand sides are definitionally equal to $\vdash$nv-embed (nf-v t) and $\vdash$np-embed (nf-p t). □

Theorem 5.7.3 (Normal forms)**.** Given two provably equal terms, their normal forms are equivalent modulo the given value and effect theories.

(i) $\forall$ t, u $\in$ ($\Gamma \vdash$v σ) . $\Gamma \vdash$v t $\equiv$ u $\implies \Gamma \vdash$nv nf-v t $\equiv$ nf-v u

(ii) $\forall$ t, u $\in$ ($\Gamma \vdash$p σ) . $\Gamma \vdash$p t $\equiv$ u $\implies \Gamma \vdash$np nf-p t $\equiv$ nf-p u

Proof. We prove this theorem by using the soundness and compatibility results from earlier.

- We first observe that the identity environments are related by $\approx_e^{\Gamma;\Gamma'}$ (Corollary 5.6.8).
- We then use the soundness result in Theorem 5.6.10 and get ([[t]]v id-env) $\approx^{\Gamma;\sigma}$ ([[u]]v id-env) and ([[t]]p id-env) $\approx_T^{\Gamma;\sigma}$ ([[u]]p id-env).
- Next, we use the compatibility results from Proposition 5.6.7 (i)-(ii) and get $\Gamma \vdash$nv reify-v ([[t]]v id-env) $\equiv$ reify-v ([[u]]v id-env) and $\Gamma \vdash$np reify-p ([[t]]p id-env) $\equiv$ reify-p ([[u]]p id-env)
- Finally, we notice that these equations are definitionally equal to $\Gamma \vdash$nv nf-v t $\equiv$ nf-v u and $\Gamma \vdash$np nf-p t $\equiv$ nf-p u □

Theorem 5.7.4 (Normalization modulo the value and effect theories)**.** Normal forms of $\text{FGCBV}_{\text{eff}}$ value and producer terms containing only the structure given in the signatures Σ_{val} and Σ_{eff} are equal to the extensions of corresponding value and effect terms.

(i) $\forall$ t $\in$ ($\Gamma \vdash \sigma$) . (nf-v (extend-val t)) $\cong$ (extend-val-nv t)

(ii) $\forall\ \sigma \in \mathsf{Ty},\ \mathsf{t} \in (\Gamma\ ,\ \Delta \vdash \mathsf{E})\ .\ (\mathsf{nf\text{-}p}\ (\mathsf{extend\text{-}eff}\ \sigma\ \mathsf{t})) \cong (\mathsf{extend\text{-}eff\text{-}np}\ \sigma\ \mathsf{t})$

Proof. We prove both (i) and (ii) by structural induction on the value and effect terms. We apply (i) to prove the base case for effect variables in (ii). □

Chapter 6

Conservativity of the extension

In this chapter, we present one of our main contributions, namely, the proof that the proposed extension of value and effect theories is conservative. The proof is based on the correctness results of the NBE algorithm from the previous chapter.

6.1 Conservativity theorem

Theorem 6.1.1 (Conservativity theorem)**.** Given two value or effect terms t and u, they are provably equal in the value and effect theories if and only if they are provably equal in $\mathrm{FGCBV}_{\mathrm{eff}}$.

(i) $\forall\ \mathsf{t}, \mathsf{u} \in (\Gamma \vdash \sigma)\ .\ \Gamma \vdash \mathsf{t} \equiv \mathsf{u} \iff (\mathsf{extend\text{-}valctx}\ \Gamma) \vdash \mathsf{v}\ (\mathsf{extend\text{-}val}\ \mathsf{t}) \equiv (\mathsf{extend\text{-}val}\ \mathsf{u})$

(ii) $\forall\ \sigma \in \mathsf{BaseTy},\ \mathsf{t}, \mathsf{u} \in (\Gamma\ ,\ \Delta \vdash \mathsf{E})\ .\ \Gamma \vdash \mathsf{E}\ \mathsf{t} \equiv \mathsf{u} \iff$
$(\mathsf{extend\text{-}valctx}\ \Gamma)\ @\ (\mathsf{extend\text{-}effctx}\ \sigma\ \Delta) \vdash \mathsf{p}\ (\mathsf{extend\text{-}eff}\ \sigma\ \mathsf{t}) \equiv (\mathsf{extend\text{-}eff}\ \sigma\ \mathsf{u})$

Proof. We prove both directions separately in Theorem 6.1.2 and 6.1.4. In addition, notice that in (ii) we require the given $\mathrm{FGCBV}_{\mathrm{eff}}$ type to be a base type although we omit the extensions $\mathsf{extend\text{-}ty}\ \sigma$ for readability. As a result, we only consider producer terms that are expressible in the effect theory in the $\Longleftarrow$ direction. For the $\Longrightarrow$ direction, we can give a more general result using an arbitrary $\mathrm{FGCBV}_{\mathrm{eff}}$ type. □

Theorem 6.1.2 ($\Longrightarrow$ direction)**.** Given two value or effect terms, if they are provably equal in the value or effect theories, then they are provably equal in $\mathrm{FGCBV}_{\mathrm{eff}}$.

(i) $\forall\ \mathsf{t}, \mathsf{u} \in (\Gamma \vdash \sigma)\ .\ \Gamma \vdash \mathsf{t} \equiv \mathsf{u} \Longrightarrow (\mathsf{extend\text{-}valctx}\ \Gamma) \vdash \mathsf{v}\ (\mathsf{extend\text{-}val}\ \mathsf{t}) \equiv (\mathsf{extend\text{-}val}\ \mathsf{u})$

(ii) $\forall\ \sigma \in \mathsf{Ty},\ \mathsf{t}, \mathsf{u} \in (\Gamma\ ,\ \Delta \vdash \mathsf{E})\ .\ \Gamma \vdash \mathsf{E}\ \mathsf{t} \equiv \mathsf{u} \Longrightarrow$
$(\mathsf{extend\text{-}valctx}\ \Gamma)\ @\ (\mathsf{extend\text{-}effctx}\ \sigma\ \Delta) \vdash \mathsf{p}\ (\mathsf{extend\text{-}eff}\ \sigma\ \mathsf{t}) \equiv (\mathsf{extend\text{-}eff}\ \sigma\ \mathsf{u})$

Proof. We prove (i) and (ii) by simultaneous induction on the derivations of $\Gamma \vdash \mathsf{t} \equiv \mathsf{u}$ and $\Gamma \vdash \mathsf{E}\ \mathsf{t} \equiv \mathsf{u}$. Both the base and inductive cases are immediate because we have extended the equational theory of $\mathrm{FGCBV}_{\mathrm{eff}}$ with all the equations in the corresponding value and effect theories. □

On the other hand, before we prove the $\Longleftarrow$ direction, we need a similar result about the extension of value and effect theories to normal forms.

Proposition 6.1.3. Given two value or effect terms, if they are provably equal as normal forms, then they are provably equal in the value and effect theories.

(i) $\forall\ \mathsf{t}, \mathsf{u} \in (\Gamma \vdash \sigma)\ .\ (\mathsf{extend\text{-}valctx}\ \Gamma) \vdash \mathsf{nv}\ (\mathsf{extend\text{-}val\text{-}nv}\ \mathsf{t}) \equiv (\mathsf{extend\text{-}val\text{-}nv}\ \mathsf{u}) \Longrightarrow \Gamma \vdash \mathsf{t} \equiv \mathsf{u}$

(ii) $\forall\ \sigma \in \mathsf{BaseTy}, \mathsf{t}, \mathsf{u} \in (\Gamma\ , \Delta \vdash \mathsf{E})\ .$
$(\mathsf{extend\text{-}valctx}\ \Gamma)\ @\ (\mathsf{extend\text{-}effctx}\ \sigma\ \Delta) \vdash \mathsf{np}\ (\mathsf{extend\text{-}eff\text{-}np}\ \sigma\ \mathsf{t}) \equiv (\mathsf{extend\text{-}eff\text{-}np}\ \sigma\ \mathsf{u})$
$\Longrightarrow \Gamma \vdash \mathsf{E}\ \mathsf{t} \equiv \mathsf{u}$

Proof.[1] We prove (i) and (ii) by simultaneous induction on the derivations of the provable equality on normal forms. The proof is based on three notable observations.

- The equational theories of normal forms do not include any $\beta\eta$-equations that are present in the equational theory of $\mathrm{FGCBV}_{\mathrm{eff}}$. Moreover, if E_{val} and E_{eff} are empty then normal forms of provably equal terms are actually equal.
- $\mathsf{extend\text{-}val\text{-}nv}\ \mathsf{t}$ only returns variables and value terms corresponding to function symbols.
- $\mathsf{extend\text{-}eff\text{-}np}\ \mathsf{t}$ returns operations and the producer terms corresponding to effect variables, (*) e.g., $(\mathsf{appAP}\ (\mathsf{var}\ (\mathsf{extend\text{-}effvar}\ \mathsf{w}))\ \star)\ \mathsf{toNP}\ (\mathsf{returnNP}\ (\mathsf{avNV}\ (\mathsf{varAV}\ \mathsf{Hd})))$. This observation is possible because we consider only base types and variables of base type can be embedded into normal values.

Therefore, the induction on the derivations of the assumed proof terms involves equations for reflexivity, symmetry, transitivity, congruence of function symbols, congruence of operations, (**) congruence of toNP and the equations of the value and effect theories. These equations, except for (**), have corresponding counterparts in the value and effect theories and, therefore, we can construct all the required proofs in the value and effect theories. However, notice that the equation (**) only relates normal producers of the form (*) arising from extensions of effect variables. These normal producers can only be related if the applications are related as atomic producers. Further, the atomic producers are only related when the atomic values (i.e., effect variables) are equal allowing us to construct a corresponding proof in the effect theory. □

[1]The proof of Proposition 6.1.3 has not yet been formalized in Agda.

Theorem 6.1.4 ($\Longleftarrow$ direction)**.** Given two value or effect terms, if they are provably equal in $\mathrm{FGCBV}_{\mathrm{eff}}$, then they are provably equal in the value and effect theories.

(i) $\forall$ t, u $\in$ ($\Gamma \vdash \sigma$) . (extend-valctx Γ) $\vdash$v (extend-val t) $\equiv$ (extend-val u) $\Longrightarrow$ $\Gamma \vdash$ t $\equiv$ u

(ii) $\forall$ $\sigma \in$ BaseTy, t, u $\in$ (Γ , $\Delta \vdash$E) .
(extend-valctx Γ) @ (extend-effctx σ Δ) $\vdash$p (extend-eff σ t) $\equiv$ (extend-eff σ u)
$\Longrightarrow$ $\Gamma \vdash$E t $\equiv$ u

Proof. We do not prove (i) and (ii) directly but instead use Proposition 6.1.3 together with the correctness results of our NBE algorithm.

- We first use Proposition 6.1.3 to reduce the theorem to showing an implication from the equational theory of $\mathrm{FGCBV}_{\mathrm{eff}}$ to the equational theories of normal forms as
(extend-valctx Γ) $\vdash$v (extend-val t) $\equiv$ (extend-val u) $\Longrightarrow$
(extend-valctx Γ) $\vdash$nv (extend-val-nv t) $\equiv$ (extend-val-nv u)
and
(extend-valctx Γ) @ (extend-effctx σ Δ) $\vdash$p (extend-eff σ t) $\equiv$ (extend-eff σ u) $\Longrightarrow$
(extend-valctx Γ) @ (extend-effctx σ Δ) $\vdash$np (extend-eff-np σ t) $\equiv$ (extend-eff-np σ u).
- We apply Theorem 5.7.4 to the right hand sides above to work with normalized extensions of value and effect terms in
(extend-valctx Γ) $\vdash$v (extend-val t) $\equiv$ (extend-val u) $\Longrightarrow$
(extend-valctx Γ) $\vdash$nv (nf-v (extend-val t)) $\equiv$ (nf-v (extend-val u))
and
(extend-valctx Γ) @ (extend-effctx σ Δ) $\vdash$p (extend-eff σ t) $\equiv$ (extend-eff σ u) $\Longrightarrow$
(extend-valctx Γ) @ (extend-effctx σ Δ) $\vdash$np (nf-p (extend-eff σ t)) $\equiv$ (nf-p (extend-eff σ u)).
- We use Theorem 5.7.3 with (extend-valctx Γ) $\vdash$v (extend-val t) $\equiv$ (extend-val u) and (extend-valctx Γ) @ (extend-effctx σ Δ) $\vdash$p (extend-eff σ t) $\equiv$ (extend-eff σ u) to prove the implications above. $\square$

Chapter 7

Conclusions

We discussed the modeling of and reasoning about impure higher-order functional programs and gave an overview of two main approaches to give a rigorous mathematical theory to computational effects and the impurity they introduce. Namely, we discussed the approach of using monads that was first proposed by Moggi and also the approach of using algebraic theories as proposed by Plotkin and Power. Although the former has gained more ground in programming language semantics, the latter gives a more natural and intuitive mathematical account of the behavior of computational effects. In this dissertation, we discussed an extension of such algebraic theories of computational effects to the fine-grained call-by-value intermediate language ($\mathrm{FGCBV}_{\mathrm{eff}}$) suitable for reasoning about ML-like impure programs. Whilst the extension itself is intuitive and straightforward, the proof of its conservativity forms the major part of the work we presented here. It turned out that one first needs an effective way of deciding provable equality in $\mathrm{FGCBV}_{\mathrm{eff}}$ before the conservativity of the extension can be proved. For this reason, we used a semantic reduction-free normalization method, called normalization by evaluation (NBE), giving us the needed decision procedure. This method computes normal forms by inverting the interpretation of the syntax of $\mathrm{FGCBV}_{\mathrm{eff}}$. The NBE algorithm we present in this dissertation gives a generalization of the usual presentations of NBE. In particular, we identify both the interpretation of syntax and the resulting normal forms up to suitable equivalence relations. As a result, the NBE algorithm we present normalizes the terms in $\mathrm{FGCBV}_{\mathrm{eff}}$ modulo the given algebraic theory of computational effects. Both the normalization algorithm and its proofs of correctness have been fully formalized in an interactive theorem prover and functional programming language Agda.

7.1 Future work

7.1.1 Extending the type signature

We considered a version of $\mathrm{FGCBV}_{\mathrm{eff}}$ with a rather modest type signature. For example, although we discussed finite products and function space, we omitted sum types and the empty type (i.e., finite coproducts) and also natural numbers. We conjecture that the addition of these types to the language and the conservativity proof should follow from the earlier work in the literature. For example, Altenkirch, Dybjer, Hofmann and Scott [2] have defined an NBE algorithm for simply typed lambda calculus with sum types (i.e., binary coproducts). More recently, Balat, Di Cosmo and Fiore [6] have shown that the same can be done for both the sum types and the empty type when using Grothendieck logical relations together with corresponding categories. These results suggest that similar approaches should suffice for the $\mathrm{FGCBV}_{\mathrm{eff}}$ when the eliminators of the sum types and the empty type are considered as value terms. On the other hand, natural numbers have been discussed by Dybjer and Filinski [12] in their work on NBE and type-directed partial evaluation for Gödel's System T. In $\mathrm{FGCBV}_{\mathrm{eff}}$, we should only need to consider a category with a suitable natural number object to accommodate them in our normalization and conservativity proofs.

7.1.2 First-order representations of context renamings in Agda

In Section 3.2, we suggested that it might be desirable to use first-order representation of injective context renamings instead modeling them as higher-order functions. The representation we proposed to use is called *order-preserving embeddings* (OPEs) and illustrated in Chapman's PhD thesis [10, Section 4.5]. OPEs have also appeared as the category of weakenings in the work of Altekirch, Hofmann and Streicher [4]. The idea is to define first-order operations from one context to another keeping the old variables and explicitly marking where new variables should appear. We have high hopes that such first-order representation would free us from having to postulate functional extensionality for heterogeneous equality in Agda. In particular, as the extensionality for semantic values is encoded in the definition of the partial equivalence relation $\approx^{\Gamma;\sigma\rightharpoonup\tau}$ for function types, the only proofs that use the postulated extensionality are concerned with context renamings.

7.1.3 Substitutions for normal forms

It is worthwhile to notice that we currently can not accommodate effect-dependency in the NBE correctness proofs and in the conservativity theorem. For example, consider

another natural equation for deterministic choice

$$\frac{\Gamma \vdash_v x : \mathsf{bool} \quad \Gamma \vdash_p M_1 : \sigma \quad \Gamma \vdash_p M_2 : \sigma}{\Gamma \vdash_p \mathbf{if}_\sigma\, x\, \mathbf{then}\, M_1\, \mathbf{else}\, M_2 \equiv \mathbf{if}_\sigma\, x\, \mathbf{then}\, M_1[\mathbf{true}/x]\, \mathbf{else}\, M_2[\mathbf{false}/x] : \sigma}$$

describing that, after making the choice, the boolean variable x would be true in the first branch and false in the other branch. If we would have this equation in the equational theory of $\mathrm{FGCBV}_{\mathrm{eff}}$, we would also need to add the corresponding equation to the equational theory of normal forms to prove the correctness of our NBE algorithm and the conservativity of the extension. However, it is worth noting that we cannot use the usual notion of substitutions for normal producers. In particular, we need to use a more general notion of substitutions, e.g., hereditary substitutions [38] that are capable of preserving normal forms when applying the substitutions. Lately, Keller and Altenkirch [20] have formalized a structurally recursive version of hereditary substitutions for the simply typed lambda calculus. We propose to follow their work to develop a similar notion of substitutions for our normal and atomic forms to define the effect-dependent equations for normal forms.

References

[1] Danel Ahman. Agda formalization of the fine-grain intermediate language and the provably correct normalization by evaluation algorithm. https://github.com/danelahman/normalization-by-evaluation.

[2] T. Altenkirch, P. Dybjer, M. Hofmann, and P. Scott. Normalization by evaluation for typed lambda calculus with coproducts. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, LICS '01, pages 303–310, Washington, DC, USA, 2001. IEEE Computer Society.

[3] Thorsten Altenkirch, James Chapman, and Tarmo Uustalu. Relative monads formalised. *To appear in the Journal of Formalized Reasoning. Final version pending.*

[4] Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Categorical reconstruction of a reduction free normalization proof. In David H. Pitt, David E. Rydeheard, and Peter Johnstone, editors, *Category Theory and Computer Science*, volume 953 of *Lecture Notes in Computer Science*, pages 182–199. Springer, 1995.

[5] Thorsten Altenkirch and Tarmo Uustalu. Normalization by evaluation for lambda^{-2}. In Yukiyoshi Kameyama and Peter J. Stuckey, editors, *FLOPS*, volume 2998 of *Lecture Notes in Computer Science*, pages 260–275. Springer, 2004.

[6] Vincent Balat, Roberto Di Cosmo, and Marcelo Fiore. Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '04, pages 64–76, New York, NY, USA, 2004. ACM.

[7] Ulrich Berger, Matthias Eberl, and Helmut Schwichtenberg. Normalisation by evaluation. In Bernhard Möller and J. V. Tucker, editors, *Prospects for Hardware Foundations*, volume 1546 of *Lecture Notes in Computer Science*, pages 117–137. Springer, 1998.

[8] Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed λ–calculus. In R. Vemuri, editor, *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 203–211. IEEE Computer Society Press, Los Alamitos, 1991.

[9] John Capper and Henrik Nilsson. Towards a formal semantics for a structurally dynamic noncausal modelling language. In *Proceedings of the 8th ACM SIGPLAN workshop on Types in language design and implementation*, TLDI '12, pages 39–50, New York, NY, USA, 2012. ACM.

[10] James Maitland Chapman. *Type checking and normalisation.* PhD thesis, University of Nottingham, 2008.

[11] N.G de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381 – 392, 1972.

[12] Peter Dybjer and Andrzej Filinski. Normalization and partial evaluation. In *Applied Semantics, International Summer School, APPSEM 2000, Caminha, Portugal, September 9-15, 2000, Advanced Lectures*, pages 137–192, London, UK, UK, 2002. Springer-Verlag.

[13] Andrzej Filinski. Normalization by evaluation for the computational lambda-calculus. In *Proceedings of the 5th international conference on Typed lambda calculi and applications*, TLCA'01, pages 151–165, Berlin, Heidelberg, 2001. Springer-Verlag.

[14] Marcelo Fiore. Semantic analysis of normalisation by evaluation for typed lambda calculus. In *Proceedings of the 4th ACM SIGPLAN international conference on Principles and practice of declarative programming*, PPDP '02, pages 26–37, New York, NY, USA, 2002. ACM.

[15] Marcelo Fiore, Gordon Plotkin, and Daniele Turi. Abstract syntax and variable binding. In *Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science*, LICS '99, pages 193–202, Washington, DC, USA, 1999. IEEE Computer Society.

[16] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and types*. Cambridge University Press, New York, NY, USA, 1989.

[17] Martin Hyland, Paul Blain Levy, Gordon Plotkin, and John Power. Combining algebraic effects with continuations. *Theor. Comput. Sci.*, 375(1-3):20–40, April 2007.

[18] Martin Hyland, Gordon Plotkin, and John Power. Combining effects: sum and tensor. *Theor. Comput. Sci.*, 357(1):70–99, July 2006.

[19] Achim Jung and Jerzy Tiuryn. A new characterization of lambda definability. In *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, TLCA '93, pages 245–257, London, UK, UK, 1993. Springer-Verlag.

[20] Chantal Keller and Thorsten Altenkirch. Hereditary substitutions for simple types, formalized. In *Proceedings of the third ACM SIGPLAN workshop on Mathematically structured functional programming*, MSFP '10, pages 3–10, New York, NY, USA, 2010. ACM.

[21] Paul Blain Levy, John Power, and Hayo Thielecke. Modelling environments in call-by-value programming languages. *Information and Computation*, 185(2):182–210, September 2003.

[22] Sam Lindley and Ian Stark. Reducibility and $\top\top$-lifting for computation types. In *Typed Lambda Calculi and Applications: Proceedings of the Seventh International Conference TLCA 2005*, number 3461 in Lecture Notes in Computer Science, pages 262–277. Springer-Verlag, 2005.

[23] Saunders Mac Lane. *Categories for the working mathematician*. Springer-Verlag, New York, 1971. Graduate Texts in Mathematics, Vol. 5.

[24] Saunders Mac Lane and Ieke Moerdijk. *Sheaves in geometry and logic: A First Introduction to Topos Theory*. Springer-Verlag, 1992.

[25] Per Martin-Löf. About models for intuitionistic type theories and the notion of definitional equality. In S. Kanger, editor, *3rd Scandinavian Logic Symp.*, pages 81–109. North-Holland, 1975.

[26] Conor McBride. *Dependently Typed Programs and their Proofs.* PhD thesis, University of Edinburgh, 1999.

[27] Rasmus Ejlers Møgelberg and Sam Staton. Linearly-used state in models of call-by-value. In *Proceedings of the 4th international conference on Algebra and coalgebra in computer science*, CALCO'11, pages 298–313, Berlin, Heidelberg, 2011. Springer-Verlag.

[28] E. Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in computer science*, pages 14–23, Piscataway, NJ, USA, 1989. IEEE Press.

[29] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[30] Ulf Norell. *Towards a Practical Programming Language Based on Dependent type Theory.* PhD thesis, Chalmers University of Technology, 2007.

[31] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '93, pages 71–84, New York, NY, USA, 1993. ACM.

[32] A. M. Pitts. Categorical logic. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science, Volume 5. Algebraic and Logical Structures*, chapter 2, pages 39–128. Oxford University Press, 2000.

[33] Gordon Plotkin and Matija Pretnar. Handlers of algebraic effects. In *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, ESOP '09, pages 80–94, Berlin, Heidelberg, 2009. Springer-Verlag.

[34] Gordon D. Plotkin. Lambda definability and logical relations. Technical report, School of Artificial Intelligence, University of Edinburgh, 1973.

[35] Gordon D. Plotkin. Lambda-Definability in the Full Type Hierarchy. In J. P. Seldin and J. R. Hindley, editors, *To H.B.˜Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism.* Academic Press, London, 1980.

[36] Gordon D. Plotkin and John Power. Notions of computation determine monads. In *Proceedings of the 5th International Conference on Foundations of Software Science and Computation Structures*, FoSSaCS '02, pages 342–356, London, UK, UK, 2002. Springer-Verlag.

[37] Gordon D. Plotkin and John Power. Algebraic operations and generic effects. *Applied Categorical Structures*, 11(1):69–94, 2003.

[38] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework: The propositional fragment. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *TYPES*, volume 3085 of *Lecture Notes in Computer Science*, pages 355–377. Springer, 2003.

[39] Julianna Zsidó. *Typed Abstract Syntax*. PhD thesis, Université de Nice Sophia Antipolis, 2010.