

Contents

1	Introduction	3
1.1	Overview	3
1.2	Related Work	5
2	Metatheory	6
2.1	Type Theory	6
2.2	Agda	7
2.2.1	Basic Constructions	8
2.2.2	Formalization	10
2.2.3	Standard Library	10
3	Syntax	14
3.1	Core Syntax	14
3.2	Context Embeddings and Substitutions	16
3.2.1	Embeddings	16
3.2.2	Substitutions	18
3.3	Conversion	21
4	Normalization	22
4.1	Normal terms	22
4.2	Preliminaries: Models of STLC	23
4.3	Implementation	26
4.4	Kripke Models	29

5	Correctness	33
5.1	Categorical Notions	33
5.2	Laws for Embedding and Substitution	35
5.2.1	Embeddings	36
5.2.2	Substitutions	37
5.3	Completeness	40
5.4	Presheaf Refinement	43
5.5	Soundness	47
5.6	Stability	54
6	Variations	56
6.1	Direct Presheaf Model	56
6.2	More Efficient Evaluation	59
7	Discussion and Future Work	62

Chapter 1

Introduction

1.1 Overview

Normalization animates the syntax of typed λ -calculi, making them useful as programming languages. Also, in contrast to general terms, normal forms possess a more restricted structure which simplifies formal reasoning. Moreover, normalization allows one to decide convertibility of terms, which is required during type checking polymorphic and dependent type theories.¹

This thesis presents an efficient and verified implementation of $\beta\eta$ -normalization for simply typed λ -calculus (STLC) in Agda, an implementation of constructive type theory. In this setting, the syntax can be seen as an embedded language, and manipulating embedded terms is a limited form of metaprogramming. Normalization can be reused in the implementation of proof tactics, decision procedures or domain-specific languages, therefore efficiency is desirable. It is also preferable that the syntax remains as simple and self-contained as possible, both for pedagogical purposes and for the ease of reuse in other Agda developments. For this reason we choose an intrinsic syntax with de Bruijn indices and implicit substitutions.

¹In particular, the types of System F_ω largely correspond to simply typed λ -terms.

We are not concerned with mapping the algorithm to lower-level abstract machines or hardware. Thus we are free to make use of the most convenient evaluation mechanism at hand: that of the metalanguage. This way, we can gloss over implementation details of efficient higher-order evaluation. This is the core idea of normalization by evaluation (NbE) from an operational point of view. Also, totality of evaluation in the metatheory is always implicitly assumed from the inside. This lets us implement normalization in a structurally recursive way, making its totality trivial. NbE also naturally supports η -normalization.

If we are to trust a particular normalization algorithm, we need to prove the following properties (borrowing terminology from [3]):

- *Completeness*: normalization maps a term to a normal form which is convertible to the term.
- *Soundness*: normalization maps convertible terms to equal normal forms.

Additionally, we require stability, which establishes that normalization is aligned with the specification of normal forms:

- *Stability*: normalization acts as the identity function on normal terms.

Chapter 2 describes the metatheory we work in. In Chapter 3, we formalize the syntax of STLC along with $\beta\eta$ -conversion. Chapter 4 presents the normalization function, and Chapter 5 proves its correctness. Chapter 6 describes two alternative formalizations, one with more efficient evaluation and one with more concise correctness proofs. We discuss the results and possible future research in the final chapter.

The Agda formalization can be found at <https://github.com/AndrasKovacs/stlc-nbe>.

1.2 Related Work

Martin-Löf [27] first used NbE (though the term had not been coined yet) to show decidability of type checking for the 1975 version of his intuitionistic type theory. Berger and Schwichtenberg reintroduced NbE in 1991 [10] as a normalization method for lambda calculi. Abel’s work [1] gives a comprehensive overview of NbE, and also develops it for a range of type theories. The approach differs markedly from ours, as it uses extrinsic syntax and separate typed and untyped phases of evaluation.

Catarina Coquand’s work [18] is the most closely related to our development. She formally proves soundness and completeness of NbE for simple type theory. However, she considers a nameful syntax with explicit substitutions while we use intrinsic de Bruijn variables with implicit substitutions.

Altenkirch and Chapman [3] formalize a big-step normalization algorithm in Agda. This development uses an explicit substitution calculus and implements normalization using an environment machine with first-order closures. The algorithm works similarly to ours on the common syntactic fragment, but it is not structurally recursive and hence specifies evaluation as a inductive relation, using the Bove-Capretta method [11].

Altenkirch and Kaposi [5] use a glued presheaf model (“presheaf logical predicate”) for NbE for a minimal dependent type theory. This development provided the initial inspiration for the formalization in this thesis; it seemed plausible that “scaling down” dependently typed NbE to simple type theory would yield a formalization that is more compact than existing ones. This turned to be the case; however the discussion of resulting development is relegated to Chapter 6.1, because using a Kripke model has the advantage of clean separation of the actual algorithm and its naturality proofs, which was deemed preferable to the more involved presheaf construction. Also, our work uses a simple presheaf model rather than the glued model used in *Ibid.* or in the work of Altenkirch, Hoffman and Streicher [4].

Chapter 2

Metatheory

In this chapter the metatheory used for formalization is presented, first broadly, then its specific implementation in Agda.

2.1 Type Theory

The basic system we use is intensional Martin-Löf type theory (MLTT). For an overview and tutorial see the first chapter of the Homotopy Type Theory book [35]. However, there is no canonical definition of intensional type theory. There are numerous variations concerning type universes and the range of allowed type definitions. Our development uses the following features:

- *Strictly positive inductive definitions.* These are required for an intrinsically well-typed syntax. We do not require induction-recursion, induction-induction, higher induction or large inductive types.
- *Two predicative universes, named \mathbf{Set}_0 and \mathbf{Set}_1 , with large elimination into \mathbf{Set}_0 .* \mathbf{Set}_0 is the base universe in Agda, i. e. the universe of small types. Large elimination is needed for recursive definitions of semantic types, since inductive

definitions for them would not be positive and hence would be illegal. We only need `Set1` to type the large eliminations. In Agda `Set` is a synonym for `Set0`.

- *Function extensionality as an axiom.* This posits that two functions are equal if they take equal arguments to equal results. We could eschew it by using setoid reasoning with semantic equivalence, as C. Coquand does [18]. However, we only use function extensionality in correctness proofs, and the normalization function does not refer to it or to any other postulate. We are content with *logical* as opposed to computational content for correctness proofs. This way, the numerous congruence lemmas needed for setoid reasoning can be skipped. Also, function extensionality has computational interpretation in cubical [14] and observational [7] type theories, to which this development could be plausibly ported in the future (when practical implementations of the mentioned theories become available).

In Agda 2.5.2, there is robust support for dependent pattern matching without Streicher’s K axiom [33]. We use the `--without-K` language option for all of our code in the main development. However, we do use axiom K in Chapter 6.1 for the direct presheaf model, but only for technical convenience, as it can be avoided by additional uniqueness proofs for equalities.

2.2 Agda

Agda is a dependently typed programming language and proof assistant. Its design and core features are described in Ulf Norell’s thesis [30]. The latest documentation is available online [21]. For a book-length introduction geared towards beginners, see [34]. We summarize here the core constructions and our notational conventions and liberties.

2.2.1 Basic Constructions

Two essential artifacts in Agda code are *inductive type definitions* and *function definitions*.

Inductive definitions introduce new types. Agda has *open type universes*, which means that programmers may freely add new types as long as they preserve logical consistency¹. An inductive definition involves a type constructor declaration followed by zero or more term constructor declarations. For example, the type of natural numbers is defined the following way:

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ
```

Inductive definitions may have *parameters* and *indices*. The former are implicitly quantified over the term constructors, but must be uniform in the constructors' return types. The latter must be explicitly quantified in term constructors, but are allowed to vary. The definition for length-indexed vectors exhibits both:

```
data Vec (A : Set) : ℕ → Set where
  nil  : Vec A zero
  cons : {n : ℕ} → A → Vec A n → Vec A (suc n)
```

In a type constructor declaration, parameters are listed left to the colon, while indices are to the right. `(A : Set)` is a parameter, so it is implicitly quantified and is the same in the return types of `nil` and `cons`. In contrast, the length index is quantified in `cons`. We can use brackets to make parameters implicit; here `cons` has an implicit first parameter, and can be used like `cons zero nil` for a value of `Vec ℕ (suc zero)`. Implicit arguments are filled in by Agda's unification algorithm. Alternatively, the `∀` symbol can be used to leave the types of parameters implicit, and we could define `cons` as follows:

```
cons : ∀ {n} → A → Vec A n → Vec A (suc n)
```

¹Agda checks *strict positivity* as a sufficient condition for consistency.

Additionally, Agda has records, providing syntactic sugar and namespace management for iterated Σ types. For example, semigroups can be defined as elements of the following record type:

```
record Semigroup : Set1 where
  field
    S      : Set
    _⊗_    : M → M → M
    assoc  : ∀ m1 m2 m3 → (m1 ⊗ (m2 ⊗ m3)) ≡ ((m1 ⊗ m2) ⊗ m3)
```

Elements of `Semigroup` can be given as `record {S = ...; _⊗_ = ...; assoc = ...}`. Whenever we have some `(sg : Semigroup)` in context, we can bring all of its fields into the current namespace by `open Semigroup sg`. Also, given an `(sg : Semigroup)` in scope, we can use field names as projections, as in `(S sg)` or `(assoc sg)`. We will sometimes omit record instances and simply write `S` or `assoc` (or whatever record fields we are working with) when it is clear from where we are projecting from.

Function definitions are given by pattern matching. In Agda, pattern matching implements branching evaluation as it is usual in functional languages, but it also refines type indices based on particular selected cases. The main practical difference between parameters and indices is that we can gain information about indices by pattern matching, while we can't infer anything about parameters just by having a parameterized term.

For example, in the following definition of concatenation for `Vec`, the result type is refined depending on whether the first argument is empty:

```
_+_ : ℕ → ℕ → ℕ
zero + m = m
suc n + m = suc (n + m)

_++_ : ∀ {A n m} → Vec A n → Vec A m → Vec A (n + m)
nil    ++ ys = ys
cons x xs ++ ys = cons x (xs ++ ys)
```

In the `cons x xs ++ ys` case, the result type is refined to `Vec A (suc (n + m))`, allowing us to build a `suc`-long result on the right hand side with `cons`.

Function definitions must be total, hence pattern matching must be exhaustive and recursive calls must be structurally decreasing. Dependent pattern matching with structural recursion allows us to write the same proofs as inductive eliminators, but with a more convenient interface and less administrative burden; see e. g. [23] for a translation from patterns to eliminators.

As a notational convention, we shall take cues from the Idris [12] programming language and sometimes leave implicit parameters implicit even in type declarations of functions or constructors. In this style, declaring `_++_` looks as follows:

```
_++_ : Vec A n → Vec A m → Vec A (n + m)
```

This is not valid Agda, but we shall do this whenever the types and binding status of parameters are obvious. Sometimes we also omit type annotations from bindings when they can be inferred or are not relevant for exposition.

2.2.2 Formalization

The project can be found at <https://github.com/AndrasKovacs/stlc-nbe>. Most Agda code listings in this thesis are also included in the formal development, although the versions here may include syntactic liberties and abbreviations. We indicate the names of relevant source files in the examples as Agda comments or in expository text.

The formalization involves a considerable amount of details. We omit a significant part of them from our explanations; in particular, we often only present the types of definitions and omit the implementations. It may be useful to read the source code and this thesis at the same time, and also have an interactive Agda environment at hand.

2.2.3 Standard Library

The formal development does not have any external dependencies. We need less than a hundred lines of code from the Agda standard library [22], but we choose to include

it alongside the project in `Lib.agda`, for portability, and also make some changes.

Most of `Lib.agda` concerns equality and equational reasoning. Propositional equality is the same as in standard Agda:

```
data _≡_ {i}{A : Set i} (x : A) : A → Set i where
  refl : x ≡ x
```

We also postulate function extensionality, for functions with both implicit and explicit arguments (the “i” in `fexti` stands for “implicit”):

```
postulate
  fext  : (∀ x → f x ≡ g x) → f ≡ g
  fexti : (∀ x → f {x} ≡ g {x}) → (λ {x} → f {x}) ≡ (λ {x} → g {x})
```

Note that the types of `f` and `g` are left implicit above, in accordance with our notational liberties. In full Agda we write:

```
fext :
  ∀ {i j}{A : Set i}{B : A → Set j}{f g : (x : A) → B x}
  → ((x : A) → f x ≡ g x) → f ≡ g
```

Here the quantification noise is increased by universe indices. Universe polymorphism is only used in `Lib.agda`, but we will henceforth omit universe indices anyway.

Our style of equational reasoning deviates from the standard library. First, we use a more compact notation for symmetry and transitivity, inspired by the Homotopy Type Theory book [35]:

```
_■_ : x ≡ y → y ≡ z → x ≡ z -- transitivity
refl ■ refl = refl

_⁻¹ : x ≡ y → y ≡ x -- symmetry
refl ⁻¹ = refl
```

`_⁻¹` is postfix, so if we have `(p : x ≡ y)`, then `(p ⁻¹)` has type `(y ≡ x)`.

Second, for reasoning about congruences we mainly use two operations. The first (named `ap` in the HoTT book and `cong` in the standard library) expresses that functions respect equality:

```

_&_ : (f : A → B) → x ≡ y → f x ≡ f y
f & refl = refl
infixl 9 _&_

```

The second one expresses that *non-dependent function application* respects equality as well:

```

_⊗_ : f ≡ g → x ≡ y → f x ≡ g y
refl ⊗ refl = refl
infixl 8 _⊗_

```

Using these two operators, we can lift non-dependent functions with arbitrary arities to congruences. For example, if we have $(p : a_0 \equiv a_1)$, $(q : b_0 \equiv b_1)$ and $(f : A \rightarrow B \rightarrow C)$, then $(f \& p \otimes q)$ has type $(f \ a_0 \ b_0 \equiv f \ a_1 \ b_1)$. Note that both operators associate left, and $_ \& _$ binds more strongly. This is reminiscent of the lifting syntax with applicative functors [29] in Haskell programming.

Furthermore, we use coercion instead of transport (see HoTT book [35, pp. 72], also named `subst` in the Agda standard library):

```

coe : A ≡ B → A → B
coe refl a = a

```

`subst` can be recovered by $_ \& _$ and `coe`:

```

subst : (P : A → Set) → x ≡ y → P x → P y
subst P eq = coe (P & eq)

```

The choice for `coe` is mainly stylistic and rather subjective. `coe (P & eq)` does not look worse than `subst P eq`, but `coe` by itself is used often and is more compact than `subst id`.

In Agda, sometimes an explicit “equational reasoning” syntax is used. For example, commutativity for addition may look like the following:

```

+-comm : (m n : ℕ) → m + n ≡ n + m
+-comm zero    n = +-right-identity n -1
+-comm (suc m) n = begin
  suc m + n    ≡⟨ ⟩
  suc (m + n)  ≡⟨ suc & +-comm m n ⟩
  suc (n + m)  ≡⟨ +-suc n m -1 ⟩
  n + suc m    QED

```

In formal development we do not use this style and instead keep the intermediate expressions implicit:

```

+-comm (suc m) n = (suc & +-comm n m) ■ (+-suc n m -1)

```

This is because in an interactive environment one can step through the intermediate points fairly easily, and writing them out adds significant visual clutter. However, we will use informal equational reasoning when it aids explanation.

We also borrow Σ (dependent sum), τ (unit type), \perp (empty type) and $_ \sqcup _$ (disjoint union) from the standard library. Elements of Σ are constructed as (a, b) pairs, and the projections are proj_1 and proj_2 . $A \times B$ is defined in terms of Σ as the non-dependent pair type. τ has tt (“trivially true”) as sole element. Injection into disjoint unions is done with inj_1 and inj_2 . The *ex falso* principle is named $\perp\text{-elim}$, with type $\{A : \text{Set}\} \rightarrow A \rightarrow A$.

Chapter 3

Syntax

In this chapter we present the syntax of simply typed lambda calculus - in our case intrinsic syntax with implicit substitutions and de Bruijn indices. We take term conversion to be part of the syntax and therefore include it in this chapter.¹ Since the definition of conversion refers to substitution and weakening, they are also presented here.

3.1 Core Syntax

The complete definition is the following:

```
-- Syntax.agda
data Ty : Set where
  ι      : Ty
  _⇒_    : Ty → Ty → Ty
```

¹We take the view that the “proper” definition of the syntax would be an initial object in some suitable category of models. In this way, models include conversion as propositional equations, so the syntax is quotiented by conversion as well; see e. g. [13] and [6]. We do not use this definition because quotients are not yet natively supported in Agda, and we find value in developing a conventional presentation first.

```

data Con : Set where
  •   : Con
  _,_ : Con → Ty → Con

data _∈_ (A : Ty) : Con → Set where
  vz : A ∈ (Γ , A)
  vs : A ∈ Γ → A ∈ (Γ , B)

data Tm Γ : Ty → Set where
  var : A ∈ Γ → Tm Γ A
  lam : Tm (Γ , A) B → Tm Γ (A ⇒ B)
  app : Tm Γ (A ⇒ B) → Tm Γ A → Tm Γ B

```

There is a base type ι (Greek iota) and function types. Contexts are just lists of types. $_ \in _$ is an inductively defined membership relation on contexts. It has the same structure as de Bruijn indices; for this reason they are named **vz** for “zero” and **vs** for “suc”. Terms are parameterized by a Γ context and indexed by a type. **var** picks an entry from the contexts, **lam** is abstraction while **app** is function application.

As an example, the identity function on the base type is represented as:

```

id- $\iota$  : Tm • ( $\iota \Rightarrow \iota$ )
id- $\iota$  = lam (var vz) -- with nameful syntax:  $\lambda (x : \iota) . x$ 

```

The presented syntax is *intrinsic*. In other words, only the well-typed terms are defined. An *extrinsic* definition would first define untyped *preterms* and separately a typing relation on preterms and contexts:

```

data Tm : Set where -- omitted constructors
data _ $\vdash$ _∈_ (Γ : Con) → Tm → Ty → Set where -- omitted constructors

```

Here, $\Gamma \vdash t \in A$ expresses that the preterm t is well-formed and has type A in context Γ .

There are advantages and disadvantages to both intrinsic and extrinsic definitions. With intrinsic syntax, well-formedness and type preservation come for free. On the other hand, proofs and computations which do not depend on types are easier to formulate with preterms. In the case of STLC, the type system is simple enough so that

carrying around type information never becomes burdensome, so in this setting intrinsic definitions are all-around more convenient. For polymorphic systems such as System F , intrinsic typing introduces significant bureaucratic overhead, by tagging terms with type coercions; see Benton et al. [9] for approaches to dealing with them. For extrinsic syntax, powerful automation for reasoning about substitution is available in Coq, allowing spectacularly compact proofs [32]. It remains to be seen if it can be practically adapted to intrinsic syntax. For dependent type theories, intrinsic typing with quotient inductive-inductive definitions seems to be most promising, as it requires a relatively compact set of rules [6], in contrast to extrinsic approaches which suffer from a veritable explosion of well-formedness conditions and congruence rules.

3.2 Context Embeddings and Substitutions

In order to specify β -conversion, we need a notion of substitution. Also, the η -rule must refer to a notion of *weakening* or *embedding*: it mentions the “same” term under and over a λ binder, but the two occurrences cannot be definitionally the same, since they are in different contexts and thus have different types. We need to express the notion that whenever one has a term in a context, one can construct essentially the same term in a larger context. We use *order-preserving-embeddings* for this. Embeddings enable us to state the η -rule, but we also make use of them for defining substitutions.

3.2.1 Embeddings

We define order-preserving embeddings the following way:

```
-- Embedding.agda
data OPE : Con → Con → Set where
  •      : OPE • •
  drop  : OPE  $\Gamma$   $\Delta$  → OPE ( $\Gamma$  , A)  $\Delta$ 
  keep  : OPE  $\Gamma$   $\Delta$  → OPE ( $\Gamma$  , A) ( $\Delta$  , A)
```


Elements of $\text{OPE } \Gamma \Delta$ express that Δ can be obtained from Γ by dropping zero or more entries in order.

Some choices can be made here. C. Coquand [19] uses an explicit identity embedding constructor with type $(\forall \{\Gamma\} \rightarrow \text{OPE } \Gamma \Gamma)$ instead of our \bullet . This slightly simplifies some proofs and slightly complicates others. It does not have propositionally unique identity embeddings², but overall the choice between this definition and ours is fairly arbitrary.

An alternative solution is using *renamings*. Renamings are essentially substitutions containing only variables, alternatively definable as functions mapping variables of smaller contexts to variables of larger contexts [28, pp. 24]:

$$\begin{aligned} \text{Ren} &: \text{Con} \rightarrow \text{Con} \rightarrow \text{Set} \\ \text{Ren } \Gamma \Delta &= \{A : \text{Ty}\} \rightarrow A \in \Delta \rightarrow A \in \Gamma \end{aligned}$$

However, renamings are able to express *permutations* of contexts as well, which we have no need for in this development. Thus, renamings are larger than what is strictly necessary, which may or may not result in technical difficulties. We aim to choose the smallest workable representation, as a general principle.

Embeddings have action on variables and terms, reconstructing them in larger contexts. We denote embedding by lowercase “e” subscripts:

$$\begin{aligned} \epsilon_e &: \text{OPE } \Gamma \Delta \rightarrow A \in \Delta \rightarrow A \in \Gamma \\ \epsilon_e \bullet &= v \\ \epsilon_e (\text{drop } \sigma) v &= v \sigma (\epsilon_e \sigma v) \\ \epsilon_e (\text{keep } \sigma) v z &= v z \\ \epsilon_e (\text{keep } \sigma) (v \sigma v) &= v \sigma (\epsilon_e \sigma v) \\ \\ \text{Tm}_e &: \text{OPE } \Gamma \Delta \rightarrow \text{Tm } \Delta A \rightarrow \text{Tm } \Gamma A \\ \text{Tm}_e \sigma (\text{var } v) &= \text{var } (\epsilon_e \sigma v) \\ \text{Tm}_e \sigma (\text{lam } t) &= \text{lam } (\text{Tm}_e (\text{keep } \sigma) t) \\ \text{Tm}_e \sigma (\text{app } f a) &= \text{app } (\text{Tm}_e \sigma f) (\text{Tm}_e \sigma a) \end{aligned}$$

²Since wrapping the identity embedding with any number of `keep`-s also yields identity embeddings.

Identity embeddings keep every entry:

$$\begin{aligned} \text{id}_e &: \forall \{\Gamma\} \rightarrow \text{OPE } \Gamma \ \Gamma \\ \text{id}_e \ \{\bullet\} &= \bullet \\ \text{id}_e \ \{\Gamma, A\} &= \text{keep } (\text{id}_e \ \{\Gamma\}) \end{aligned}$$

We define wk as shorthand for the embedding that only drops the topmost entry:

$$\begin{aligned} \text{wk} &: \forall \{A \ \Gamma\} \rightarrow \text{OPE } (\Gamma, A) \ \Gamma \\ \text{wk} &= \text{drop id}_e \end{aligned}$$

3.2.2 Substitutions

A salient feature of our syntax is the lack of explicit substitution, i. e. we define substitutions separately as lists of terms, and their action on terms is given as a recursive function. We choose implicit substitutions for two reasons:

1. It is the canonical presentation of STLC in textbooks [31, 24].
2. Lighter formalization. With implicit substitutions, the definition of conversion becomes significantly smaller. With explicit substitutions, we need congruence closure of β and η conversion on terms and substitutions, plus rules for the action of substitution on terms. With implicit substitutions, only terms need to be considered. For illustration, Altenkirch and Chapman's definition [3] of conversion for explicit substitution calculus has twenty-four rules, while this development has only seven. It also follows that with our syntax, equational reasoning about substitutions involves definitional or propositional equalities, which are automatically respected by all constructions. There is minor complication involving semantic interpretation of substitutions, discussed in Chapter 5.5.

Strictly speaking, stating β -conversion only requires single substitutions. However, simultaneous substitution is easier to define and reason about. Thus, we shall define the former using the latter and identity substitutions. We have substitutions as lists of terms:

```
-- Substitution.agda
data Sub (Γ : Con) : Con → Set where
  •      : Sub Γ •
  _,_    : Sub Γ Δ → Tm Γ A → Sub Γ (Δ , A)
```

Elements of $\text{Sub } \Gamma \Delta$ contain a $\text{Tm } \Gamma A$ for each A type in Δ . In other words, an element of $\text{Sub } \Gamma \Delta$ assigns to each variable in Δ a term in Γ .

Again, there are some choices in defining **Sub**. Identity substitutions (which have the identity action on terms) could be represented as an explicit constructor, as well as **Sub** composition and weakenings. Our **Sub** can be seen as a normal form of substitutions: explicit composition and weakening forms tree-like structures, but we flatten them to just lists of terms. This eliminates redundancy, and allows us to define compositions and identities recursively, which provides us with more definitional equalities.

We could also give a functional definition (see [28, pp. 24]), similarly to how we did for renamings before:

```
Sub : Con → Con → Set
Sub Γ Δ = {A : Ty} → A ∈ Δ → Tm Γ A
```

However, this contains many definitionally distinct functions with the same action on terms, and so we dismiss it on grounds of unnecessary largeness.

Before we can define identity substitutions and the action of substitution on terms, we need an operation with type $(\forall \{A : \text{Ty}\} \rightarrow \text{Sub } \Gamma \Delta \rightarrow \text{Sub } (\Gamma , A) (\Delta , A))$. This enables pushing substitutions under λ -binders when recursing on terms. Constructing this substitution requires that we embed all terms in the input substitution into the extended (Γ , A) context. We define this as right composition with an embedding:

```
_s ° e_ : Sub Δ Σ → OPE Γ Δ → Sub Γ Σ
•      s ° e δ = •
(σ , t) s ° e δ = (σ s ° e δ) , Tm_e δ t
```

The notation $_s \circ e_$ expresses that we have a substitution on the left and an embedding on the right. Its type resembles that of ordinary function composition, if we consider both **Sub** and **OPE** as context morphisms. We will expand on the categorical

interpretation of this in Chapter 5.2.

There is a canonical injection $\ulcorner _ \urcorner_{\text{ope}}$ from **OPE** to **Sub**:

$$\begin{aligned} \text{drop}_s &: \text{Sub } \Gamma \Delta \rightarrow \text{Sub } (\Gamma, A) \Delta \\ \text{drop}_s \sigma &= \sigma \circ_{\text{e}} \text{wk} \\ \\ \text{keep}_s &: \text{Sub } \Gamma \Delta \rightarrow \text{Sub } (\Gamma, A) (\Delta, A) \\ \text{keep}_s \sigma &= \text{drop}_s \sigma, \text{var vz} \\ \\ \ulcorner _ \urcorner_{\text{ope}} &: \text{OPE } \Gamma \Delta \rightarrow \text{Sub } \Gamma \Delta \\ \ulcorner \bullet \urcorner_{\text{ope}} &= \bullet \\ \ulcorner \text{drop } \sigma \urcorner_{\text{ope}} &= \text{drop}_s \ulcorner \sigma \urcorner_{\text{ope}} \\ \ulcorner \text{keep } \sigma \urcorner_{\text{ope}} &= \text{keep}_s \ulcorner \sigma \urcorner_{\text{ope}} \end{aligned}$$

We note that keep_s is the needed operation for pushing substitutions under binders.

Now the action of substitution on terms and variables can be defined:

$$\begin{aligned} \epsilon_s &: \text{Sub } \Gamma \Delta \rightarrow A \in \Delta \rightarrow \text{Tm } \Gamma A \\ \epsilon_s (\sigma, t) \text{ vz} &= t \\ \epsilon_s (\sigma, t) (\text{vs } v) &= \epsilon_s \sigma v \\ \\ \text{Tm}_s &: \text{Sub } \Gamma \Delta \rightarrow \text{Tm } \Delta A \rightarrow \text{Tm } \Gamma A \\ \text{Tm}_s \sigma (\text{var } v) &= \epsilon_s \sigma v \\ \text{Tm}_s \sigma (\text{lam } t) &= \text{lam } (\text{Tm}_s (\text{keep}_s \sigma) t) \\ \text{Tm}_s \sigma (\text{app } f a) &= \text{app } (\text{Tm}_s \sigma f) (\text{Tm}_s \sigma a) \end{aligned}$$

ϵ_s simply looks up a term from a substitution³, while Tm_s recurses into terms to substitute all variables. Identity substitutions can be defined as well:

$$\begin{aligned} \text{id}_s &: \{\Gamma : \text{Con}\} \rightarrow \text{Sub } \Gamma \Gamma \\ \text{id}_s \{\bullet\} &= \bullet \\ \text{id}_s \{\Gamma, A\} &= \text{keep}_s \text{id}_s \end{aligned}$$

Single substitution with a $(t : \text{Tm } \Gamma A)$ term is given by (id_s, t) . This assigns

³The immediate reason for not naming ϵ_s simply “lookup” is that we will need several different lookup functions for various purposes. For any sizable formal development, naming schemes eventually emerge, for better or worse. The author has found that uniformly encoding salient information about types of operations in their names is worthwhile to do. It is also not easy to hit the right level of abstraction; too little and we repeat ourselves too much or neglect to include known structures, but too much abstraction may cause the project to be less accessible and often also less convenient to develop in the first place. Many of the operations and proofs here could be defined explicitly using categorical language, i. e. bundling together functors’ actions on objects, morphisms and category laws in records, then projecting out required components. Our choice is to keep the general level of abstraction low for this development.

the \mathbf{t} term to the zeroth de Bruijn variable and leaves all other variables unchanged.

3.3 Conversion

The conversion relation is given as:

```
-- Conversion.agda
data _~_ {Γ} : ∀ {A} → Tm Γ A → Tm Γ A → Set where
  η      : t ~ lam (app (Tme wk t) (var vz))
  β      : app (lam t) t' ~ Tms (ids , t') t

  lam    : t ~ t' → lam t ~ lam t'
  app    : f ~ f' → a ~ a' → app f a ~ app f' a'

  ~refl  : t ~ t
  _~-1_ : t ~ t' → t' ~ t
  _~■_   : t ~ t' → t' ~ t'' → t ~ t''
```

η and β are the actual conversion rules. We push \mathbf{t} under a lambda with $(\text{Tm}_e \text{ wk})$ in η , and use single substitution for β . The rest are rules for congruence (lam , app) and equivalence closure ($\sim\text{refl}$, \sim^{-1} , \sim^\bullet). The syntax for equivalence rules is the same as for propositional equality, except for the \sim prefixes.

Chapter 4

Normalization

In this chapter we specify normal forms and implement normalization. Then, we discuss Kripke models and how normalization can be expressed in terms of them.

4.1 Normal terms

Our definition of normal forms is entirely standard: they are either lambdas or *neutral* terms of base type, and neutral terms are variables applied to zero or more normal arguments:

```
-- NormalForm.agda
mutual
  data Nf (Γ : Con) : Ty → Set where
    ne  : Ne Γ ι → Nf Γ ι
    lam : Nf (Γ , A) B → Nf Γ (A ⇒ B)

  data Ne (Γ : Con) : Ty → Set where
    var : A ∈ Γ → Ne Γ A
    app : Ne Γ (A ⇒ B) → Nf Γ A → Ne Γ B
```

Agda allows us to overload `var`, `lam` and `app` for normal forms. β -normality is obvious, since only variables can be applied. Normal forms are also η -long: since `ne` only injects

neutrals of base type, all normal terms of function type must in fact be lambdas. Only requiring β -normality would be as simple as having $(ne : \forall \{A\} \rightarrow Ne \ \Gamma \ A \rightarrow Nf \ \Gamma \ A)$. Alternatively, normal forms can be given as

```
data Nf (Γ : Con) : (A : Ty) → Tm Γ A → Set
```

with the same construction as before except that it is a predicate on general terms, expressing their normality. This definition would be roughly as convenient as the one we use. We also need actions of context embeddings:

```
Nfe : OPE Γ Δ → Nf Δ A → Nf Γ A
Nee : OPE Γ Δ → Ne Δ A → Ne Γ A
```

Normal and neutral terms can be injected back to terms. Also, injection commutes with embedding (i. e. injection is natural).

```
⌊_⌋Nf : Nf Γ A → Tm Γ A
⌊_⌋Ne : Ne Γ A → Tm Γ A

⌊_⌋Ne-nat : ∀ σ n → ⌊ Nee σ n ⌋Ne ≡ Tme σ ⌊ n ⌋Ne
⌊_⌋Nf-nat : ∀ σ n → ⌊ Nfe σ n ⌋Nf ≡ Tme σ ⌊ n ⌋Nf
```

4.2 Preliminaries: Models of STLC

Clearly, STLC is a small fragment of Agda, so we should be able to interpret the syntax back to Agda types and constructions in a straightforward way. From a semantic viewpoint, the most straightforward interpretation of the syntax is called the *standard model*. From an operational viewpoint, the standard model is just a well-typed interpreter [8] for STLC as an embedded language. It is implemented as follows:

```
-- Misc/StdModel.agda
Tys : Ty → Set
Tys ⊥      = ⊥
Tys (A ⇒ B) = Tys A → Tys B
```

```

Cons : Con → Set
Cons •      = ⊤
Cons (Γ , A) = Cons Γ × Tys A

Es : ∀ {Γ A} → A ∈ Γ → (Cons Γ → Tys A)
Es vz      Γs = proj2 Γs
Es (vs v) Γs = Es v (proj1 Γs)

Tms : ∀ {Γ A} → Tm Γ A → (Cons Γ → Tys A)
Tms (var v)   Γs = Es v Γs
Tms (lam t)   Γs = λ as → Tms t (Γs , as)
Tms (app f a) Γs = Tms f Γs (Tms a Γs)

```

Traditionally, notation for semantics uses double brackets, e. g. the interpretation of types would look like this:

```

[[_]] : Ty → Set
[[ ⊥   ]] = ⊥
[[ A ⇒ B ]] = [[ A ]] → [[ B ]]

```

We instead opt for notating semantic interpretations with superscripts on type constructors (contrast *subscripts*, which we use for denoting operations for embeddings and substitutions). Syntactic overloading in Agda is not convenient for our purposes here, so we have to distinguish different constructions (terms or types, etc.) and different models in any case. Therefore it makes sense to just reuse the names of types in the syntax, and distinguish different models by different superscripts. Similarly, we annotate names of function arguments with superscripts, if they are elements of certain semantic sets. For example, for some model \mathbb{M} we use $\Gamma^{\mathbb{M}}$, $\Delta^{\mathbb{M}}$ etc. for elements of semantic contexts, and $a^{\mathbb{M}}$, $b^{\mathbb{M}}$ etc. for semantic terms.

As to the implementation of the model: we interpret the base type with the empty type, and functions as functions. Contexts are interpreted as lists of semantic values. Terms are interpreted as functions from semantic contexts to semantic types. Now, normalizing $(\text{Tm}^s (\text{lam} (\text{var} \text{vz})) \text{tt})$ yields the $(\lambda a^s \rightarrow a^s)$ Agda term, so indeed we interpret a syntactic identity function with a metatheoretic identity function.

The standard model already has a key feature of NbE: it uses metatheoretic func-

tions for evaluation. However, the standard model does not allow one to go back to syntax from semantics. If we have an $(f : A^s \rightarrow B^s)$ semantic function, all we can do with it is to apply it to an argument and extract a B^s . *Weak evaluation* of programs assumes that all variables are mapped to semantic values. This corresponds to the usual notion of program evaluation in common programming languages. In contrast, strong normalization requires reducing terms inside function bodies, under binders. Bound variables are not mapped to any semantic value, so they block evaluation. In STLC, variables block function application; in richer systems variables may block case analysis or elimination of inductively defined data as well. See [1, pp. 12] for an overview for various approaches for implementing computation under binders.

NbE adds additional structure to semantic types, internalizing computation with blocking variables, which allows one to get back to syntax, moreover, to a subset of syntax containing only normal terms.

Adding progressively more structure to models allows us to prove more properties about the syntax. The standard model yields a simple proof of consistency, namely that there is no term with base type in the empty context:

```
consistency : Tm • 1 → ⊥
consistency t = Tms t tt
```

Kripke models contain slightly more structure than the standard model, allowing us to implement normalization, which is a *completeness* theorem from a logical viewpoint [20], as per the Curry-Howard correspondence. Adding yet more structure yields *presheaf models*, which enable correctness proofs for normalization as well. We summarize the computational and logical interpretations below. Be mindful though that the table picks out specific things relevant to our interest, and there are many more computations and logical interpretations possible for each class of models.

Table 4.1: Overview of models

Model	Computation	Proof
Standard	Type-safe interpreter	Consistency
Kripke	Normalizer	Completeness
Presheaf	Normalizer	Proof-relevant completeness

4.3 Implementation

We denote the model for normalization with ⁿ superscripts. The implementation follows a similar shape as the standard model. The key difference - from which others follow - is in the interpretation of functions:

```
-- Normalization.agda
Tyn : Ty → Con → Set
Tyn ⊢      Γ = Nf Γ ⊢
Tyn (A ⇒ B) Γ = ∀ {Δ} → OPE Δ Γ → Tyn A Δ → Tyn B Δ
```

Each type is mapped to a $(\text{Con} \rightarrow \text{Set})$ predicate. Hence, semantic types are not just sets anymore, but families of sets indexed by contexts. The base type is mapped to the family of its normal forms. Functions are mapped to semantic functions which take semantic inputs to semantic outputs in *any context larger than Γ* . The additional **OPE** parameter is the key to the algorithm. As it was noted in Chapter 4.2, semantic functions must be able to take blocking variables as inputs. Blocking variables must be necessarily “fresh” with respect to the Γ context of a semantic function. Here, we can apply a semantic function to suitable embedding into a larger context, allowing to subsequently apply it to semantic terms containing variables pointing into that context. In the simplest case (and the only case we will actually use), if we have

$$f^n : \forall \{ \Delta \} \rightarrow \text{OPE } \Delta \ \Gamma \rightarrow \text{Ty}^n \ A \ \Delta \rightarrow \text{Ty}^n \ B \ \Delta$$

then we also have

$$f^N \text{ (wk } \{A\}) : \text{Ty}^N A (\Gamma, A) \rightarrow \text{Ty}^N B (\Gamma, A)$$

The interpretations of contexts and variables are analogous to those in the standard model: contexts become lists of semantic values and variables look up semantic values from semantic contexts. However, we define semantic contexts inductively rather than recursively, for technical convenience.

$$\begin{aligned} \text{data } \text{Con}^N &: \text{Con} \rightarrow \text{Con} \rightarrow \text{Set} \text{ where} \\ &\bullet : \text{Con}^N \bullet \Delta \\ _,_ &: \text{Con}^N \Gamma \Delta \rightarrow \text{Ty}^N A \Delta \rightarrow \text{Con}^N (\Gamma, A) \Delta \\ \epsilon^N &: \forall \{ \Gamma A \} \rightarrow A \in \Gamma \rightarrow \forall \{ \Delta \} \rightarrow \text{Con}^N \Gamma \Delta \rightarrow \text{Ty}^N A \Delta \\ \epsilon^N \text{ vz} & \quad (\Gamma^N, t^N) = t^N \\ \epsilon^N \text{ (vs } v) & (\Gamma^N, _) = \epsilon^N v \Gamma^N \end{aligned}$$

When interpreting terms, variables and applications are easy. In the latter case we just apply the recursive result to id_e - we can choose *not* to conjure up fresh variables if there is no need to. We run into a bit of a bump in the interpretation of lambdas:

$$\begin{aligned} \text{Tm}^N &: \forall \{ \Gamma A \} \rightarrow \text{Tm} \Gamma A \rightarrow \forall \{ \Delta \} \rightarrow \text{Con}^N \Gamma \Delta \rightarrow \text{Ty}^N A \Delta \\ \text{Tm}^N \text{ (var } v) & \quad \Gamma^N = \epsilon^N v \Gamma^N \\ \text{Tm}^N \text{ (lam } t) & \quad \Gamma^N = \lambda \sigma a^N \rightarrow \text{Tm}^N t (? , a^N) \quad \text{-- ? marks the bump} \\ \text{Tm}^N \text{ (app } f a) & \quad \Gamma^N = \text{Tm}^N f \Gamma^N \text{id}_e (\text{Tm}^N a \Gamma^N) \end{aligned}$$

Clearly, the idea is to evaluate the inner t term in (Γ^N, a^N) as we did in the standard model, but Γ^N does not have the right type. It has type $\text{Con}^N \Gamma \Delta$, but we need to return in some Σ extended context, with $(\sigma : \text{OPE } \Sigma \Delta)$ witnessing the extension. So, what we need is an action of context embedding on semantic contexts, and on semantic terms as well, since the former are just lists of the latter. To implement it, we first need *composition* for context embeddings:

$$\begin{aligned} _,\circ_e_, &: \text{OPE } \Delta \Sigma \rightarrow \text{OPE } \Gamma \Delta \rightarrow \text{OPE } \Gamma \Sigma \\ \sigma &\quad \circ_e \bullet = \sigma \\ \sigma &\quad \circ_e \text{ drop } \delta = \text{drop } (\sigma \circ_e \delta) \\ \text{drop } \sigma &\quad \circ_e \text{ keep } \delta = \text{drop } (\sigma \circ_e \delta) \\ \text{keep } \sigma &\quad \circ_e \text{ keep } \delta = \text{keep } (\sigma \circ_e \delta) \end{aligned}$$

It witnesses transitivity for the embedding relation. Then, embeddings on semantic terms is easy enough:

$$\begin{aligned}
\text{Ty}^N_e &: \text{OPE } \Delta \ \Gamma \rightarrow \text{Ty}^N \ A \ \Gamma \rightarrow \text{Ty}^N \ A \ \Delta \\
\text{Ty}^N_e \ \{\iota\} &\quad \sigma \ t^N = \text{Nf}_e \ \sigma \ t^N \\
\text{Ty}^N_e \ \{A \Rightarrow B\} &\quad \sigma \ t^N = \lambda \ \delta \ a^N \rightarrow t^N \ (\sigma \circ_e \delta) \ a^N \\
\\
\text{Con}^N_e &: \text{OPE } \Sigma \ \Delta \rightarrow \text{Con}^N \ \Gamma \ \Delta \rightarrow \text{Con}^N \ \Gamma \ \Sigma \\
\text{Con}^N_e \ \sigma \bullet &\quad = \bullet \\
\text{Con}^N_e \ \sigma \ (\Gamma^N, t^N) &= \text{Con}^N_e \ \sigma \ \Gamma^N, \text{Ty}^N_e \ \sigma \ t^N
\end{aligned}$$

For $\text{Ty}^N_e \ \{\iota\}$, we use embedding of normal forms. For $\text{Ty}^N_e \ \{A \Rightarrow B\}$, we compose the externally given σ embedding with the input embedding δ . Con^N_e is just pointwise application of Ty^N_e . Tm^N can be now given as:

$$\begin{aligned}
\text{Tm}^N &: \forall \ \{\Gamma \ A\} \rightarrow \text{Tm} \ \Gamma \ A \rightarrow \forall \ \{\Delta\} \rightarrow \text{Con}^N \ \Gamma \ \Delta \rightarrow \text{Ty}^N \ A \ \Delta \\
\text{Tm}^N \ (\text{var } v) &\quad \Gamma^N = \epsilon^N \ v \ \Gamma^N \\
\text{Tm}^N \ (\text{lam } t) &\quad \Gamma^N = \lambda \ \sigma \ a^N \rightarrow \text{Tm}^N \ t \ (\text{Con}^N_e \ \sigma \ \Gamma^N, a^N) \\
\text{Tm}^N \ (\text{app } f \ a) &\quad \Gamma^N = \text{Tm}^N \ f \ \Gamma^N \ \text{id}_e \ (\text{Tm}^N \ a \ \Gamma^N)
\end{aligned}$$

We still need a quote function to transform semantic terms to normal terms. We also need to mutually define an unquote function which sends neutral terms to semantic terms, for reasons shortly illuminated. Then, normalization is given as evaluation followed by quotation, where evaluation uses a semantic context given by unquoting each variable in the context.

$$\begin{aligned}
\text{mutual} \\
q^N &: \forall \ \{A \ \Gamma\} \rightarrow \text{Ty}^N \ A \ \Gamma \rightarrow \text{Nf} \ \Gamma \ A \\
q^N \ \{\iota\} &\quad t^N = t^N \\
q^N \ \{A \Rightarrow B\} &\quad t^N = \text{lam} \ (q^N \ (t^N \ \text{wk} \ (u^N \ (\text{var } v_z)))) \\
\\
u^N &: \forall \ \{A \ \Gamma\} \rightarrow \text{Ne} \ \Gamma \ A \rightarrow \text{Ty}^N \ A \ \Gamma \\
u^N \ \{\iota\} &\quad n = \text{ne } n \\
u^N \ \{A \Rightarrow B\} &\quad n = \lambda \ \sigma \ a^N \rightarrow u^N \ (\text{app} \ (\text{Ne}_e \ \sigma \ n) \ (q^N \ a^N)) \\
\\
u^{cN} &: \forall \ \{\Gamma\} \rightarrow \text{Con}^N \ \Gamma \ \Gamma \\
u^{cN} \ \{\bullet\} &\quad = \bullet \\
u^{cN} \ \{\Gamma, A\} &= \text{Con}^N_e \ \text{wk} \ u^{cN}, u^N \ (\text{var } v_z) \\
\\
\text{nf} &: \forall \ \{\Gamma \ A\} \rightarrow \text{Tm} \ \Gamma \ A \rightarrow \text{Nf} \ \Gamma \ A \\
\text{nf } t &= q^N \ (\text{Tm}^N \ t \ u^{cN})
\end{aligned}$$

At first, it can be difficult to build a mental model of the operation of the algorithm.

On the highest level, normalization alternates between evaluation and quoting: a term is first evaluated, then if it has a function type, its semantic value is applied to a “blocking” input and we quote the result, which can be a semantic function again. This evaluate-quote alternation proceeds until the result is not a function but a base value. Note that although \mathbf{q}^N does not directly call \mathbf{Tm}^N , it does apply semantic functions, and this application results in an *internal* call to \mathbf{Tm}^N , accordingly to the definition of semantic application.

In fact, this encapsulation of recursive \mathbf{Tm}^N calls is a crucial detail which makes this whole definition structurally recursive and thus total. In the $(\mathbf{Tm}^N (\text{lam } t) \Gamma^N = \lambda \sigma a^N \rightarrow \mathbf{Tm}^N t (\text{Con}^N_e \sigma \Gamma^N, a^N))$ clause, the recursive \mathbf{Tm}^N call is evidently structural, and because it happens under a *metatheoretic* lambda, semantic functions can be applied to *any* value in any definition without compromising structurality.

To explain the previous scare quotes around “blocking”: in the $(t^N \text{ wk } (u^N (\text{var } vz)))$ application, the $(u^N (\text{var } vz))$ term plays the role of blocking input, but it is not quite entirely blocking. u^N performs an operation best described as *semantic η -expansion*: it acts as identity on neutral base terms, and from neutral functions it produces semantic functions which build up neutral applications from their inputs. Thus, $(u^N \{i\} (\text{var } vz))$ simply reduces to $(\text{ne } (\text{var } vz))$, while $(u^N \{i \Rightarrow i\} (\text{var } vz))$ reduces to $(\lambda \sigma a^N \rightarrow \text{ne } (\text{app } (\text{var } (\text{E}_e \sigma vz)) a^N))$. In short, u^N returns semantic values which yield properly η -expanded normal forms when quoted. As convoluted this may seem, it is actually a very elegant solution.

4.4 Kripke Models

We show now how Kripke models relate to the above definition of normalization. Following the presentation of Altenkirch [2], Kripke models can be defined in Agda for the syntax of STLC as follows (omitting universe polymorphism):

```
-- Misc/Kripke.agda
record KripkeModel : Set1 where
  field
    W      : Set
    _≤_     : W → W → Set
    ≤refl   : ∀ {w} → w ≤ w
    _≤■_    : ∀ {w1 w2 w3} → w1 ≤ w2 → w2 ≤ w3 → w1 ≤ w3
    _||-ι_  : W → Set
    ι-mono  : ∀ {w1 w2} → w1 ≤ w2 → w1 ||-ι → w2 ||-ι
```

A Kripke model consists of a preordered set of “worlds”, denoted W , along with a forcing predicate for the base type, and a *monotonicity condition* ι -mono. On a concrete level, we can obtain the definition of `KripkeModel` by noticing the details in our previous `N` model which can be abstracted out, and packing them into a record. Indeed, the interpretation of function types, contexts and terms can be derived from this amount of data. Agda allow us to include them in the record as well, as definitions which depend on the record fields:

```
record KripkeModel : Set1 where
  field
    ... -- the same as before

    _||-Ty_ : W → Ty → Set
    w ||-Ty ι      = w ||-ι
    w ||-Ty (A ⇒ B) = ∀ {w'} → w ≤ w' → w' ||-Ty A → w' ||-Ty B

    _||-Con_ : W → Con → Set
    w ||-Con •      = ⊤
    w ||-Con (Γ , A) = (w ||-Con Γ) × (w ||-Ty A)

    _||-_ : Con → Ty → Set
    Γ ||- A = ∀ w → w ||-Con Γ → w ||-Ty A

    Ty-mono : w ≤ w' → w ||-Ty A → w' ||-Ty A
    Ty-mono {ι}      σ p = ι-mono σ p
    Ty-mono {A ⇒ B} σ p δ q = p (σ ≤■ δ) q

    Con-mono : w ≤ w' → w ||-Con Γ → w' ||-Con Γ
    Con-mono {•}      σ p = p
    Con-mono {Γ , A} σ p = Con-mono σ (proj1 p) , Ty-mono {A} σ (proj2 p)
```

```

||-Tm : Tm  $\Gamma$  A  $\rightarrow$   $\Gamma$  ||- A
||-Tm (var vz)      w ( _ , q) = q
||-Tm (var (vs v)) w (p , q) = ||-Tm (var v) w p
||-Tm (lam t)       w p  $\sigma$  a = ||-Tm t _ (Con-mono  $\sigma$  p , a)
||-Tm (app f a)     w p       = ||-Tm f w p  $\leq$ refl (||-Tm a w p)

```

With this, everything up to Tm^N in our previous implementation can be obtained by defining the corresponding model:

```

N : KripkeModel
N = record {
  W      = Con
; _ $\leq$ _   =  $\lambda$   $\Gamma$   $\Delta \rightarrow$  OPE  $\Delta$   $\Gamma$ 
;  $\leq$ refl = ide
; _ $\leq$ ■_   = _ $\circ_e$ _
; _||- $\iota$  =  $\lambda$   $\Gamma \rightarrow$  Nf  $\Gamma$   $\iota$ 
;  $\iota$ -mono = Nfe }

```

This is an interesting exercise in abstraction, but what is the deeper significance? STLC can be viewed as a simple intuitionistic propositional logic, with a single atomic proposition ι . What is a right notion of semantics, though? We could abstract out the interpretation of base types from the standard model as given in Chapter 4.2, and get another notion of models; would not that suffice? From a logician's point of view, the issue is that it does not give us completeness. Semantics in general is necessary because we use logic to talk about concepts about which we have underlying intuition, but we have no *a priori* reason to believe that syntactic constructions talk meaningfully about any intuitive concept.

Soundness and completeness together express that syntactic and semantic entailment are logically equivalent; the lack of completeness indicates that there is a mismatch, that semantics is larger than necessary. In Agda, we define soundness and completeness for Kripke models as follows¹:

```

sound      = Tm  $\Gamma$  A  $\rightarrow$  ( $\forall$  M  $\rightarrow$  let open KripkeModel M in  $\Gamma$  ||- A)
complete = ( $\forall$  M  $\rightarrow$  let open KripkeModel M in  $\Gamma$  ||- A)  $\rightarrow$  Tm  $\Gamma$  A

```

¹Note that these are wholly different from soundness and completeness as defined for a normalization algorithm in Chapter 1.1.

The `let open KripkeModel M` incantation makes locally available all fields and definitions of `M`. We give here a brief informal proof for soundness and completeness.

Theorem. *STLC is sound and complete with respect to Kripke models.*

Proof. Soundness follows immediately from $||\cdot||_{\mathbf{Tm}}$. For completeness, we instantiate the hypothesis with the previously defined `N` model, then define quoting and unquoting the same way as in Chapter 4.3. Using them, we can produce a $(\mathbf{Nf} \vdash \mathbf{A})$, which can be canonically injected back to $(\mathbf{Tm} \vdash \mathbf{A})$. \square

With the standard set-theoretic semantics for first-order logic, the intuitive content is clear: formulas talk about truth and falsehood. What is an intuitive meaning of Kripke semantics, though? A possible answer is that it talks about *knowledge*. `W` worlds can be understood as states of knowledge, and \leq denotes increasing knowledge. Monotonicity expresses that if something is proven in some state of knowledge, it will never be invalidated, not matter how “wiser” we get.

Note that in the formal development the direct `^` definitions are used instead of Kripke models. This is primarily to make the algorithm as transparent as possible to lay readers. In the `Normalization.agda` file, the core algorithm is laid out in about 50 lines, and together with the definition of syntax and embedding it is about a hundred lines.

Chapter 5

Correctness

In this chapter, we prove completeness, soundness and stability for the algorithm.

5.1 Categorical Notions

First, we shall introduce a modest number of concepts from category theory. The prime motivation is that they allow us to compactly describe the contents of our proofs. They also make it easier to mentally keep track of proof obligations, since often a few words of categorical definitions can be unpacked into a dozen lemmas. When proofs align with category theory, that is usually also indicative that one has the right approach. That said, we shall keep category theory to a minimum and only introduce definitions that are directly applicable to our work.

A *category* is defined as follows.

```
-- Misc/Category.agda
record Category : Set1 where
  field
    Obj      : Set                -- "objects"
    morph    : Obj → Obj → Set  -- "morphisms"
    id       : morph I I
    _◦_      : morph J K → morph I J → morph I K
```

```

idl   : f ∘ id ≡ f
idr   : id ∘ f ≡ f
ass   : f ∘ (g ∘ h) ≡ (f ∘ g) ∘ h

```

In short, categories have a set of objects and sets of morphisms between objects, with identity morphisms, composition, and associativity and identity laws for composition. Note that this is a naive definition which does not take into account size issues and subtleties arising from the ambient type theory; for a more rigorous treatment see [35, pp. 319].

In written exposition, $obj : C$ expresses that obj is an object of C , while $f : C(A, B)$ means that f is a morphism of C from A to B .

Most importantly, the syntax of STLC itself forms a category, which is sometimes called the *syntactic category*. We denote it as **STLC**. Its objects are contexts, its morphisms are substitutions with the identity substitution as identity morphism. We have not yet defined composition for substitution, which provides composition for **STLC**, nor the composition laws. Providing these will be a significant part of our proof obligations about substitutions.

Another important category is the category of sets, denoted **Set**. In our setting, **Set** has as objects Agda types and functions as morphisms (again, ignoring subtleties).

Functors are structure-preserving mappings between categories:

```

record Functor (C D : Category) : Set₁ where
  field
    Obj⇒      : Obj C → Obj D
    morph⇒    : morph I J → morph (Obj⇒ I) (Obj⇒ J)
    id⇒       : morph⇒ id ≡ id
    ◦⇒        : morph⇒ (f ∘ g) ≡ morph⇒ f ∘ morph⇒ g

```

We use $F : C \rightarrow D$ to express that F is a functor from C to D . Functors map objects to objects and morphisms to morphisms. Also, they must map identities to identities and compositions to compositions. We will use the terms *action on objects* and *action on morphisms* to denote the first two, and *functor laws* for the latter two.

A *contravariant functor* flips morphisms, i. e. it has

$$\text{morph} \Rightarrow : \text{morph } I \ J \rightarrow \text{morph } (\text{Obj} \Rightarrow J) \ (\text{Obj} \Rightarrow I)$$

The composition law changes accordingly. Contravariant functors are often denoted as $F : C^{op} \rightarrow D$ (meaning that F 's domain is the *opposite* category of C , where all morphisms are flipped). Regular non-flipping functors are sometimes called *covariant functors*.

A *presheaf* is a contravariant functor from some C category to **Set**, i. e. it maps objects to sets and morphisms to functions, with morphism arrows being flipped in the process.

Natural transformations are mappings between $F, G : C \rightarrow D$ functors. They consist of a ϕ family of morphisms for each I object of C , and a *naturality* condition, which expresses that ϕ commutes with any lifted C -morphism.

```
record Nat {C D : Category} (F G : Functor C D) : Set₁ where
  field
    Φ    : {I : Obj C} → morph (Obj ⇒ F I) (Obj ⇒ G I)
    nat  : {I J : Obj C} {f : morph I J} → ψ ∘ morph ⇒ F f ≡ morph ⇒ G f ∘ ψ
```

$F : C \rightarrow D$ functors themselves form a category, where functors are objects and natural transformations are morphisms.

This development mostly utilizes presheaves and natural transformations between presheaves. More concretely, our setting will be mostly $PSh(\mathbf{OPE})$, the category of presheaves on **OPE**, where **OPE** is the category of contexts and order-preserving context embeddings. We examine **OPE** in detail in the next section.

5.2 Laws for Embedding and Substitution

Before we attempt to prove correctness we should pin down the laws of embedding and substitution. People who set out to write normalization proofs soon find that a

jumble of twenty-odd substitution lemmas is required to make headway. The naive proving process works by repeatedly hitting roadblocks and reacting by adding more lemmas. A more prudent way is to characterize all of them beforehand in categorical terms, which lends confidence that a given set of lemmas is complete.

5.2.1 Embeddings

Contexts and order-preserving embeddings form a category, which we call **OPE**. Objects are contexts, morphisms are elements of the **OPE** type. Identity and composition are as previously defined in Chapter 3.2.1 and Chapter 4.3:

```
ide  : ∀ {Γ} → OPE Γ Γ
_◦e_ : OPE Δ Σ → OPE Γ Δ → OPE Γ Σ
```

The category laws can be proven with simple induction.

```
idle : ide ◦e σ ≡ σ           -- left identity
idre : σ ◦e ide ≡ σ           -- right identity
asse : (σ ◦e δ) ◦e ν ≡ σ ◦e (δ ◦e ν) -- associativity
```

Additionally, variables and terms of a given type are presheaves on **OPE**. $(\lambda \Gamma \rightarrow A \in \Gamma)$ maps each object of **OPE** to an object of **Set**, that is, an Agda type. Likewise does $(\lambda \Gamma \rightarrow \text{Tm } \Gamma \text{ } A)$. They also have action on morphisms: these are precisely the embedding operations on variables and terms, defined in Chapter 3.2.1.

```
εe  : OPE Γ Δ → (A ∈ Δ → A ∈ Γ)
Tme : OPE Γ Δ → (Tm Δ A → Tm Γ A)
```

Note that the input is a morphism in **OPE**, i. e. a context embedding, and the output is a morphism in **Set**, which is an Agda function. Also note the contravariance: the order of Γ and Δ is flipped in the output. Functor laws for variables and terms are as follows:

```
-- Embedding.agda
ε-ide : ∀ v → εe ide v ≡ v
ε-◦e  : ∀ σ δ ν → εe (σ ◦e δ) ν ≡ εe δ (εe σ ν)
```

```

Tm-ide : ∀ t → Tme ide t ≡ t
Tm-◦e  : ∀ σ δ t → Tme (σ ◦e δ) t ≡ Tme δ (Tme σ t)

```

Here, the functor laws are given in a pointwise form. The categorically precise type of $\mathsf{E-id}_e$ would be the following:

```

E-ide : Ee ide ≡ (λ v → v)

```

This expresses that id_e is mapped to the identity function in **Set**. Instead of using this definition, we apply both sides of the equation to a variable, and similarly transform the types for the other laws. The reason for this is that proving the categorically precise forms requires function extensionality which we would like to avoid when possible.

Normal and neutral terms are also presheaves on **OPE** in a similar manner:

```

-- NormalForm.agda
Nfe    : OPE Γ Δ → Nf Δ A → Nf Γ A
Nee    : OPE Γ Δ → Ne Δ A → Ne Γ A
Nf-◦e  : ∀ σ δ t → Nfe (σ ◦e δ) t ≡ Nfe δ (Nfe σ t)
Ne-◦e  : ∀ σ δ t → Nee (σ ◦e δ) t ≡ Nee δ (Nee σ t)
Nf-ide : ∀ t → Nfe ide t ≡ t
Ne-ide : ∀ t → Nee ide t ≡ t

```

This is to be expected, since neutral and normal terms are just terms with restricted structure.

5.2.2 Substitutions

We previously introduced **STLC** as the syntactic category, containing contexts as objects and substitutions as morphisms. However, we then presented **OPE** as yet another category with contexts as objects. Why single out **STLC** as the category which characterizes syntax? Well, the conversion relation is an integral part of the syntax, and **OPE** does not contain the morphisms needed to define β -conversion. In contrast, **STLC** contains all substitutions as well as all embeddings, since embeddings can be viewed as restricted substitutions. **OPE** is a *wide subcategory* [17] of **STLC**: it contains all the objects of **STLC** but only some of its morphisms.

We review identity substitution here (previously defined in Chapter 3.2.2). Composition is elementwise substitution of terms in a substitution.

```

ids : ∀ {Γ} → Sub Γ Γ
ids {•}      = •
ids {Γ , A} = keeps ids

_◦s_ : Sub Δ Σ → Sub Γ Δ → Sub Γ Σ
•      ◦s δ = •
(σ , t) ◦s δ = (σ ◦s δ) , Tms δ t

```

We also have substitution operations for terms and variables:

```

Es   : Sub Γ Δ → A ∈ Δ → Tm Γ A
Tms  : Sub Γ Δ → Tm Δ A → Tm Γ A

```

We aim to establish that **STLC** is a category, and that $(Tm _ A)$ and $(A \in _)$ are presheaves on it. This is significantly more complicated than what we have seen for **OPE**. Identity and composition for substitutions are actually defined using embeddings, thus proving properties of substitution involves a proving a variety of lemmas about their interaction with embedding.

The categorical structure of proof obligations is less clear here; although we have clear goals (category and presheaf laws), this author is unaware of a compact and abstract explanation of the process of building **STLC** around **OPE**. In [9], substitution laws are constructed the same way as here, but the authors of *Ibid.* also do not provide a semantic explanation. For contrast, see [5], where substitutions are given first as part of the syntax, and renamings (an alternative of embeddings) are defined later as a subcategory.

There are four different operations of composing embeddings and substitutions, since there are two choices for both arguments. We define the missing $_e \circ_s _$ operation here:

```

_◦e_ : OPE Δ Σ → OPE Γ Δ → OPE Γ Σ -- defined in 4.3
_◦s_ : Sub Δ Σ → Sub Γ Δ → Sub Γ Σ -- defined in this section
_◦s◦e_ : Sub Δ Σ → OPE Γ Δ → Sub Γ Σ -- defined in 3.2.2

```

```

_e°s_ : OPE Δ Σ → Sub Γ Δ → Sub Γ Σ
•      e°s δ      = δ
drop σ e°s (δ , t) = σ e°s δ
keep σ e°s (δ , t) = (σ e°s δ) , t

```

There are variations of the category laws where occurrences of composition can be any of the four versions, and it appears that some (not all) of these laws have to be proven. Similarly, functor laws for terms and variables have multiple versions differing in the mapped composition. All in all, we need to prove the following theorems in order:

```

-- Substitution.agda
asssee : ∀ σ δ v → (σ s°e δ) s°e v ≡ σ s°e (δ °e v)
assese : ∀ σ δ v → (σ e°s δ) s°e v ≡ σ e°s (δ s°e v)

idles : ∀ σ → ide e°s σ ≡ σ
idlse : ∀ σ → ids s°e σ ≡ Γ σ ⊔ope
idres : ∀ σ → σ e°s ids ≡ Γ σ ⊔ope

E-e°s : ∀ σ δ v → Es (σ e°s δ) v ≡ Es δ (Ee σ v)
Tm-e°s : ∀ σ δ t → Tms (σ e°s δ) t ≡ Tms δ (Tme σ t)

E-s°e : ∀ σ δ v → Es (σ s°e δ) v ≡ Tme δ (Es σ v)
Tm-s°e : ∀ σ δ t → Tms (σ s°e δ) t ≡ Tme δ (Tms σ t)

assses : ∀ σ δ v → (σ s°e δ) °s v ≡ σ °s (δ e°s v)
asssse : ∀ σ δ v → (σ °s δ) s°e v ≡ σ °s (δ s°e v)

-- Functor laws for (A ∈ _) and (Tm _ A)
E-°s : ∀ σ δ v → Es (σ °s δ) v ≡ Tms δ (Es σ v)
Tm-°s : ∀ σ δ t → Tms (σ °s δ) t ≡ Tms δ (Tms σ t)
E-ids : ∀ v → Es ids v ≡ var v
Tm-ids : ∀ t → Tms ids t ≡ t

-- Category laws for STLC
idrs : ∀ σ → σ °s ids ≡ σ
idls : ∀ σ → ids °s σ ≡ σ
asss : ∀ σ δ v → (σ °s δ) °s v ≡ σ °s (δ °s v)

```

The proofs are moderately interesting and involve straightforward induction and equational reasoning. They comprise about 110 lines of Agda.

5.3 Completeness

In this section we prove completeness of normalization. It expresses that the output of normalization is convertible to its input.

```
complete : ∀ {Γ A}(t : Tm Γ A) → t ~ ⌈ nf t ⌋Nf
```

`nf` refers to the function defined in Chapter 4.3. In the return type, the injection $\lceil _ \rceil Nf$ converts $(Nf \ \Gamma \ A)$ back to $(Tm \ \Gamma \ A)$.

Since `nf` is defined with a Kripke model, it is clear that proving properties about it has to involve similar structures: we will have interpretation of types, contexts, terms, quoting and unquoting. In general, proofs about properties of functions must have the same inductive “shape” as the functions in question. The main difference here is that types are interpreted as *relations*:

```
-- Completeness.agda
_≈_ : ∀ {A Γ} → Tm Γ A → TyN A Γ → Set
_≈_ {ι}      t tN = t ~ ⌈ tN ⌋Nf
_≈_ {A ⇒ B}{Γ} t tN =
  ∀ {Δ}(σ : OPE Δ Γ){a aN} → a ≈ aN → app (Tme σ t) a ≈ tN σ aN
```

$_ \approx _$ is a *Kripke logical relation*. Here, “logical relation” simply means that it is a relation defined by induction on types. “Kripke” indicates that the interpretation of function types is generalized to work in extended contexts. The purpose of the generalization is the same as with evaluation: it provides us with a “fresh variable” supply, allowing us to quote semantic functions and go under binders.

$_ \approx _$ essentially extends conversion to relate syntactic and semantic terms. At the base type, it is precisely conversion; at function types, it expresses that syntactic and semantic application respect $_ \approx _$.

We extend $_ \approx _$ to a relation between substitutions and semantic contexts:

```
data _≈c_ {Γ} : ∀ {Δ} → Sub Γ Δ → ConN Δ Γ → Set where
  •      : • ≈c •
  _,_    : σ ≈c δ → t ≈ t' → (σ , t) ≈c (δ , t')
```


Substitutions are lists of syntactic terms while semantic context are lists of semantic terms, so they can be elementwise related by \approx .

Mirroring N , we need monotonicity for semantic terms and contexts:

$$\begin{aligned} \approx_e & : \forall \sigma \rightarrow t \approx t^N \rightarrow Tm_e \sigma t \approx Ty^N_e \sigma t^N \\ \approx^c_e & : \forall \sigma \rightarrow \delta \approx^c \nu^N \rightarrow (\delta \circ_e \sigma) \approx^c Con^N_e \sigma \nu^N \end{aligned}$$

The only interesting case here is the base type:

$$\approx_e \{1\} \sigma t \approx t^N = ?$$

Here, the goal has type $(Tm_e \sigma t \sim \ulcorner Nf_e \sigma t^N \urcorner Nf)$, and we have $(t \approx t^N : t \sim \ulcorner t^N \urcorner Nf)$. First, we witness $(Tm_e \sigma t \sim Tm_e \sigma \ulcorner t^N \urcorner Nf)$ by showing that embedding respects conversion:

$$\begin{aligned} & \text{-- Embedding.agda} \\ \sim_e & : \forall \sigma \rightarrow t \sim t' \rightarrow Tm_e \sigma t \sim Tm_e \sigma t' \end{aligned}$$

This can be proven by induction on $(t \sim t')$, with the help of substitution laws defined in Chapter 5.2. Now, $(\sim_e \sigma t \approx t^N : Tm_e \sigma t \sim Tm_e \sigma \ulcorner t^N \urcorner Nf)$, and the type can be shown to be equal to the goal using $\ulcorner Nf\text{-nat}$.

The only needed lemma which has no direct counterpart in N is the following:

$$\begin{aligned} \sim_{\blacksquare} & : \forall \{A \Gamma\} \{t \ t'\} \{t^N : Ty^N A \Gamma\} \rightarrow t \sim t' \rightarrow t' \approx t^N \rightarrow t \approx t^N \\ \sim_{\blacksquare} \{1\} & \quad p \ q = p \sim_{\blacksquare} q \\ \sim_{\blacksquare} \{A \Rightarrow B\} & \quad p \ q = \lambda \sigma \ a \approx a^N \rightarrow \text{app } (\sim_e \sigma p) \sim_{\text{refl}} \sim_{\blacksquare} q \sigma \ a \approx a^N \end{aligned}$$

Now, the interpretation of terms can be defined. This is often called the “fundamental theorem” of a logical relation interpretation.

$$Tm_{\approx} : \forall (t : Tm \Gamma A) \{\sigma\} \{\delta^N : Con^N \Gamma \Delta\} \rightarrow \sigma \approx^c \delta^N \rightarrow Tm_s \sigma t \approx Tm^N t \delta^N$$

The proof is by induction on the input term. Witnesses for variables are simply looked up from $(\sigma \approx^c \delta^N)$, the same way as with other models. If the input term is an application, we use the inductive hypotheses the same way as in Tm^N , although first we need to rewrite the goal type with a functor law:

```

Tm≈ (app f a) {σ} σ≈δN
  rewrite Tm-ide (Tms σ f) -1 = Tm≈ f σ≈δN ide (Tm≈ a σ≈δN)

```

Only the case for `(lam t)` is non-trivial:

```

Tm≈ (lam t) {σ} {δ} σ≈δN v {a} {aN} a≈aN = ?

```

The goal is

```

app (lam (Tme (keep v) (Tms (keeps σ) t))) a ≈ TmN t (ConNe v δN , aN)

```

By using the induction hypotheses, we get the following:

```

Tm≈ t (≈ce v σ≈δN , a≈aN)
  : Tms ((σs ◦e v) , a) t ≈ TmN t (ConNe v δN , aN)

```

The left hand side of the `_≈_` does not yet match the goal. We need to use `_~■≈_` to compose it with a conversion proof on the left. The β rule can be used, but the result type does not line up on the right hand side:

```

β (Tme (keep v) (Tms (keeps σ) t)) a
  : app (lam (Tme (keep v) (Tms (keeps σ) t))) a ~
    Tms (ids , a) (Tme (keep v) (Tms (keeps σ) t))

```

Hence we need to coerce this type to the desired one, using embedding and substitution laws. We omit the equality proof here.¹

```

Tm≈ (lam t) {σ} {δN} σ≈δN v {a} {aN} a≈aN =
  coe eq (β (Tme (keep v) (Tms (keeps σ) t))) a -- eq omitted
  ~■≈
  Tm≈ t (≈ce v σ≈δN , a≈aN)

```

Now, quoting, unquoting and completeness can be defined as follows:

```

mutual
  q≈ : ∀ {A} → t ≈ tN → t ~ ⊢ qN tN ⊢Nf
  q≈ {ι}      t≈tN = t≈tN
  q≈ {A ⇒ B} t≈tN = η _ ~■ lam (q≈ (t≈tN wk (u≈ (var vz))))

```

¹Generally, explicit reasoning about substitutions is omitted from textbooks as well, because it tends to be technical and uninteresting.

```

u≈ : ∀ {A} (n : Ne Γ A) → Γ n ⊢ Ne ≈ uN n
u≈ {ι}      n = ~refl
u≈ {A ⇒ B} n σ {a} {aN} a ≈ aN
rewrite Γ ⊢ Ne-nat σ n -1 =
  app ~refl (q≈ a ≈ aN) ~▀ u≈ (app (Nee σ n) (qN aN))

uc≈ : ∀ {Γ} → ids {Γ} ≈c ucN
uc≈ {•}      = •
uc≈ {Γ , A} = ≈ce wk uc≈ , u≈ (var vz)

complete : ∀ {Γ A} (t : Tm Γ A) → t ~ Γ nf t ⊢ Nf
complete t = coe eq (q≈ (Tm≈ t uc≈)) -- eq omitted

```

Since, `nf` was defined as evaluation followed by quotation, we define `complete` this way as well. Expanding the type of `complete`, we get:

```
complete : ∀ {Γ A} (t : Tm Γ A) → t ~ Γ qN (TmN t ucN) ⊢ Nf
```

However, $(q_{\approx} (Tm_{\approx} t u^c_{\approx}))$ has type $(Tm_s id_s t \sim \Gamma q^N (Tm^N t u^{cN}) \vdash Nf)$, hence we need an extra coercion to rewrite the result type along $Tm-id_s$. This completes the proof. \square

Now, since a large part of the proof was just reiterating the structure of N , would not it be better to implement normalization and completeness in one go? For instance, the following could be implemented:

```
nf : ∀ {Γ A} (t : Tm Γ A) → Σ (Nf Γ A) λ n → t ~ Γ n ⊢ Nf
```

However, this has a disadvantage: the proof terms threaded through add considerable noise to any subsequent proof about normalization. It is also contrary to our goal of implementing the algorithm as simply as possible.

5.4 Presheaf Refinement

For soundness, one might be tempted to try directly defining another logical relation, and proceeding similarly as in the previous section. That approach would fall short.

The reason is that at some point one has to prove naturality for evaluation, quoting and unquoting, i. e. that they commute with embeddings.

$$\begin{aligned} \text{Tm}^N\text{-nat} &: \forall t \sigma \Gamma^N \rightarrow \text{Tm}^N t (\text{Con}^N_e \sigma \Gamma^N) \equiv \text{Ty}^N_e \sigma (\text{Tm}^N t \Gamma^N) \\ \text{q}^N\text{-nat} &: \forall \sigma t^N \rightarrow \text{Nf}_e \sigma (\text{q}^N t^N) \equiv \text{q}^N (\text{Ty}^N_e \sigma t^N) \\ \text{u}^N\text{-nat} &: \forall \sigma n \rightarrow \text{Ty}^N_e \sigma (\text{u}^N n) \equiv \text{u}^N (\text{Ne}_e \sigma n) \end{aligned}$$

Unfortunately, these are not actually provable, because Ty^N is too large: it contains “unnatural” semantic terms. Recall its definition:

$$\begin{aligned} \text{Ty}^N &: \text{Ty} \rightarrow \text{Con} \rightarrow \text{Set} \\ \text{Ty}^N \iota & \quad \Gamma = \text{Nf} \Gamma \iota \\ \text{Ty}^N (A \Rightarrow B) \Gamma &= \forall \{\Delta\} \rightarrow \text{OPE} \Delta \Gamma \rightarrow \text{Ty}^N A \Delta \rightarrow \text{Ty}^N B \Delta \end{aligned}$$

$(\text{Ty}^N \iota \Gamma)$ is always fine, but $(\text{Ty}^N (A \Rightarrow B) \Gamma)$ contains ill-behaved values, for instance a semantic function which returns different values depending on the length of the input context Δ . Such a function clearly does not commute with embedding. We would like to restrict semantic functions to a well-behaved subset.

A *presheaf model* of STLC interprets types and contexts as presheaves. In particular, it interprets function types as *presheaf exponentials* [15] whose definition expands to a semantic function together with the naturality condition we are looking for. There is a presheaf model over **OPE** which can be used for normalization and which is a straightforward refinement of the Kripke model ^N. Recall that a presheaf is a contravariant set-valued functor, hence the interpretation of types has to include four components: action on objects, action on morphisms and the two functor laws. Below, for the interpretation of types the action on objects is defined and we give the types of the other components:

$$\begin{aligned} &\text{-- action on objects} \\ \text{Ty}^P &: \text{Ty} \rightarrow \text{Con} \rightarrow \text{Set} \\ \text{Ty}^P \iota & \quad \Gamma = \text{Nf} \Gamma \iota \\ \text{Ty}^P (A \Rightarrow B) \Gamma &= \\ &\quad \Sigma (\forall \{\Delta\} \rightarrow \text{OPE} \Delta \Gamma \rightarrow \text{Ty}^P A \Delta \rightarrow \text{Ty}^P B \Delta) \lambda f^P \rightarrow \\ &\quad \forall \{\Delta \Sigma\} (\sigma : \text{OPE} \Delta \Gamma) (\delta : \text{OPE} \Sigma \Delta) a^P \\ &\quad \rightarrow f^P (\sigma \circ_e \delta) (\text{Ty}^P_e \delta a^P) \equiv \text{Ty}^P_e \delta (f^P \sigma a^P) \end{aligned}$$

```

-- action on morphisms
TyPe      : OPE Δ Γ → TyP A Γ → TyP A Δ

-- functor laws
TyP-ide   : ∀ tP → TyPe ide tP ≡ tP
TyP-◦e    : ∀ tP σ δ → TyPe (σ ◦e δ) tP ≡ TyPe δ (TyPe σ tP)

```

The key change is in the interpretation of function types, where we have a Σ of a function and a naturality condition.

Note that the action on morphisms was already present in the Kripke model under the name of “monotonicity condition”. In fact, every presheaf model is a Kripke model (but not vice versa). In Kripke models, types are also interpreted as functors, but from a *preordered set* rather than a general category.

Preordered sets can be viewed as categories with at most a single morphism between any two objects [16]. If $I \leq J$, then there is a morphism between I and J , otherwise there is no such morphism. Identity and composition of morphisms comes from reflexivity and transitivity of preordering.

In a Kripke model, functor laws hold by default, since the uniqueness of morphisms implies that identities necessarily map to identities and compositions to compositions. However, our **OPE** as it is defined in Agda is actually not a preorder. It can be informally considered as such if we only care about the mere existence of an embedding between two contexts. Formally, we could use *propositional truncation* [35, pp. 117] on the sets of morphisms of **OPE** to get an actual preorder, but that is beyond the native capabilities of Agda and the scope of this thesis.

In the presheaf model, terms are interpreted as morphisms from semantic contexts to semantic types, in other words, natural transformations. Concretely, this means that the evaluation function² should be mutually defined with the proof that it commutes with embeddings, i. e. its naturality. Similarly, quoting and unquoting are defined mutually with their naturality proofs.

²Recall that the evaluation function is the interpretation of terms.

In this section, we do not use the \mathcal{P} model - it is described in Chapter 6.1 in more detail. Instead, the previous definition of normalization using \mathcal{N} is kept unchanged, but we shall need additional machinery to prove the necessary naturalities. The solution is the following: first, we define a logical predicate on semantic values which restricts them to natural inhabitants. Then, we give all naturality proofs, using restricted semantic values when required, and also prove that the $\mathsf{Tm}^{\mathcal{N}}$ evaluation function outputs natural values.

The naturality predicate is defined below. We use superscript \mathcal{P} notation for definitions related to this predicate. Here, \mathcal{P} means “presheaf refinement”, not “presheaf model”.

```
-- PresheafRefinement.agda
TyP : ∀ {Γ A} → TyN A Γ → Set
TyP {Γ} {ι}      tN = τ
TyP {Γ} {A ⇒ B} tN =
  ∀ {Δ} (σ : OPE Δ Γ) {aN : TyN A Δ}
  → TyP aN
  → (∀ {Σ} (δ : OPE Σ Δ) → tN (σ ∘e δ) (TyNe δ aN) ≡ TyNe δ (tN σ aN))
  × TyP (tN σ aN)
```

At base type every semantic value is natural. A semantic function is natural if when applied to a natural argument it commutes with embedding and yields a natural result.

We extend naturality to semantic contexts and prove that it is preserved by embedding:

```
TyPe : ∀ σ → TyP tN → TyP (TyNe σ tN)

data ConP : ∀ {Γ Δ} → ConN Γ Δ → Set where
  •      : ConP •
  _',_   : ConP ΓN → TyP tN → ConP (ΓN , tN)

ConPe : ∀ σ → ConP ΓN → ConP (ConNe σ ΓN)
```

Next are functor laws for types and contexts. These are provable without reference to naturality. However, function extensionality is required.

```

TyN-ide : ∀ {A} (tN : TyN A Γ) → TyNe ide tN ≡ tN
TyN-ide {A = 1}      tN = Nf-ide tN
TyN-ide {A = A ⇒ B} tN = fexti λ Δ → fext λ δ → tN & idle δ

TyN-oe : ∀ {A} (tN : TyN A Σ) σ δ → TyNe (σ oe δ) tN ≡ TyNe δ (TyNe σ tN)
TyN-oe {A = 1}      tN σ δ = Nf-oe σ δ tN
TyN-oe {A = A ⇒ B} tN σ δ = fexti λ Ξ → fext λ ν → tN & asse σ δ ν

-- proofs for Con are just elementwise extensions as always
ConN-ide : ∀ σN → ConNe ide σN ≡ σN
ConN-oe : ∀ σ δ ν → ConNe (σ oe δ) ν ≡ ConNe δ (ConNe σ ν)

```

We proceed to naturality for evaluation, quoting and unquoting, which are provable this time, thanks to the additional ^P assumptions:

```

TmP      : ∀ t → ConP ΓN → TyP (TmN t ΓN)
TmN-nat   : ∀ t σ → ConP ΓN → TmN t (ConNe σ ΓN) ≡ TyNe σ (TmN t ΓN)
uP       : ∀ n → TyP (uN n)
qN-nat   : ∀ σ tN → TyP tN → Nfe σ (qN tN) ≡ qN (TyNe σ tN)
uN-nat   : ∀ σ n → TyNe σ (uN n) ≡ uN (Nee σ n)
uc P     : ∀ {Γ} → ConP (uc N {Γ})

```

The proofs are obtained using mutual induction and substitution laws.

5.5 Soundness

Soundness expresses that normalization maps convertible terms to the same normal form:

```

sound : t ~ t' → nf t ≡ nf t'

```

Unfolding `nf` in the return type yields $(q^N (Tm^N t) \equiv q^N (Tm^N t'))$. It is clear that $(Tm^N t)$ should be in some relation to $(Tm^N t')$ such that we can prove that q^N takes related inputs to equal outputs.

The simplest option is to define that relation as propositional equality. Since we assume function extensionality, propositional equality on semantic values is equivalent to the following logical relation:

```

_≈_ : ∀ {A Γ} → TyN A Γ → TyN A Γ → Set
_≈_ {1} {Γ} t t' = t ≡ t'
_≈_ {A ⇒ B}{Γ} t t' = ∀ {Δ}(σ : OPE Δ Γ){a a'} → a ≈ a' → t σ a ≈ t' σ a'

```

In other words, equality of semantic values is just propositional equality at base types and pointwise equality for semantic functions. However, we are again foiled by requirements of naturality: `_≈_` relates unnatural values as well, but some needed equalities can only be shown on natural domains. Hence, we amend the above definition:

```

-- Soundness.agda
_≈_ : ∀ {A Γ} → TyN A Γ → TyN A Γ → Set
_≈_ {1} {Γ} t t' = t ≡ t'
_≈_ {A ⇒ B}{Γ} t t' =
  ∀ {Δ}(σ : OPE Δ Γ){a a'} → TyP a → TyP a' → a ≈ a' → t σ a ≈ t' σ a'

```

Alternatively, `_≈_` and `TyP` could be merged into a single *partial equivalence relation*. In this definition, two functions are related if both are natural and take related inputs to related outputs. The naturality predicate can be recovered by setting `(TyP t)` to be `(t ≈ t)`. This alternative was explored in the formalization but appeared to be less convenient than separate naturality predicates.

To prove soundness, first we extend `_≈_` to semantic contexts and prove monotonicity, similarly as in the completeness proof:

```

data _≈c_ : ∀ {Γ Δ} → ConN Γ Δ → ConN Γ Δ → Set where
  • : • ≈c •
  _,_ : σ ≈c δ → t ≈ t' → (σ , t) ≈c (δ , t')

≈e : ∀ σ → t ≈ t' → TyNe σ t ≈ TyNe σ t'
≈ec : ∀ σ → δ ≈c v → ConNe σ δ ≈c ConNe σ v

```

Next, we prove that evaluating the same term in related semantic contexts yields related semantic values.

```

-- E≈ just looks up a witness, as before.
E≈ : (v : A ∈ Γ) → σ ≈c δ → EN v σ ≈ EN v δ

```


$$\begin{aligned}
\text{Tm} \approx & : (\text{t} : \text{Tm } \Gamma \text{ A}) \rightarrow \text{Con}^P \Gamma^N \rightarrow \text{Con}^P \Gamma^{N'} \rightarrow \Gamma^N \approx^c \Gamma^{N'} \rightarrow \text{Tm}^N \text{ t } \Gamma^N \approx \text{Tm}^N \text{ t } \Gamma^{N'} \\
\text{Tm} \approx (\text{var } v) & \quad \Gamma^P \Gamma^{P'} \sigma \approx \delta = \epsilon \approx v \sigma \approx \delta \\
\text{Tm} \approx (\text{lam } t) & \quad \Gamma^P \Gamma^{P'} \sigma \approx \delta = \\
& \quad \lambda v \text{ a}^P \text{ a}^{P'} \text{ a}^N \approx \text{a}^{N'} \\
& \quad \rightarrow \text{Tm} \approx t (\text{Con}^P_e v \Gamma^P, \text{a}^P) (\text{Con}^P_e v \Gamma^{P'}, \text{a}^{P'}) (\approx^c_e v \sigma \approx \delta, \text{a} \approx \text{a}') \\
\text{Tm} \approx (\text{app } f \text{ a}) & \quad \Gamma^P \Gamma^{P'} \sigma \approx \delta = \\
& \quad \text{Tm} \approx f \Gamma^P \Gamma^{P'} \sigma \approx \delta \text{ id}_e (\text{Tm}^P \text{ a } \Gamma^P) (\text{Tm}^P \text{ a } \Gamma^{P'}) (\text{Tm} \approx \text{a } \Gamma^P \Gamma^{P'} \sigma \approx \delta)
\end{aligned}$$

The proof is entirely mechanical, although it is complicated by the need to carry along Con^P proofs, in order to produce $(\text{Ty}^P \text{ a } \Gamma)$ naturality proofs for inputs in the $(\text{app } f \text{ a})$ case.

Next, we prove the more general lemma expressing that convertible terms are evaluated to related values:

$$\sim \approx : \text{t} \sim \text{t}' \rightarrow \text{Con}^P \Gamma^N \rightarrow \text{Con}^P \Gamma^{N'} \rightarrow \Gamma^N \approx^c \Gamma^{N'} \rightarrow \text{Tm}^N \text{ t } \Gamma^N \approx \text{Tm}^N \text{ t}' \Gamma^{N'}$$

This is the bulk of the work. We proceed with induction on $(\text{t} \sim \text{t}')$. First, let us consider the constructors for the equivalence closure, namely reflexivity, symmetry and transitivity. The latter two can be proven in general for $_ \approx _$ and $_ \approx^c _$:

$$\begin{aligned}
_ \approx^{-1} & : \text{t} \approx \text{t}' \rightarrow \text{t}' \approx \text{t} \\
_ \approx^{c-1} & : \sigma \approx^c \delta \rightarrow \delta \approx^c \sigma \\
_ \approx_{\blacksquare} _ & : \text{t} \approx \text{t}' \rightarrow \text{t}' \approx \text{t}'' \rightarrow \text{t} \approx \text{t}'' \\
_ \approx^c_{\blacksquare} _ & : \sigma \approx^c \delta \rightarrow \delta \approx^c v \rightarrow \sigma \approx^c v
\end{aligned}$$

Reflexivity is not generally provable, but we only need it on the codomain of Tm^N , which is already covered by $\text{Tm} \approx$. We can implement the cases for equivalence closure in $\sim \approx$:

$$\begin{aligned}
\sim \approx \{ \text{t} = \text{t}' \} \sim \text{refl } \Gamma^P \Gamma^{P'} \Gamma^N \approx \Gamma^{N'} & = \\
& \text{Tm} \approx \text{t } \Gamma^P \Gamma^{P'} \Gamma^N \approx \Gamma^{N'} \\
\sim \approx (\text{t}' \sim \text{t} \sim^{-1}) \Gamma^P \Gamma^{P'} \Gamma^N \approx \Gamma^{N'} & = \\
& \sim \approx \text{t}' \sim \text{t } \Gamma^{P'} \Gamma^P (\Gamma^N \approx \Gamma^{N'} \approx^{c-1}) \approx^{-1} \\
\sim \approx (\text{t} \sim \text{t}' \sim_{\blacksquare} \text{t}' \sim \text{t}'') \Gamma^P \Gamma^{P'} \Gamma^N \approx \Gamma^{N'} & = \\
& \sim \approx \text{t} \sim \text{t}' \Gamma^P \Gamma^P (\Gamma^N \approx \Gamma^{N'} \approx^c_{\blacksquare} (\Gamma^N \approx \Gamma^{N'} \approx^{c-1})) \approx_{\blacksquare} \sim \approx \text{t}' \sim \text{t}'' \Gamma^P \Gamma^{P'} \Gamma^N \approx \Gamma^{N'}
\end{aligned}$$

The `app` and `lam` congruences are also straightforward:

$$\begin{aligned}
& \sim (\text{lam } t \sim t') \ \Gamma^P \ \Gamma^{P'} \ \Gamma^N \approx \Gamma^{N'} = \\
& \quad \lambda \ v \ a^P \ a^{P'} \ a \approx a' \rightarrow \\
& \quad \sim t \sim t' \ (\text{Con}^P_e \ v \ \Gamma^P \ , \ a^P) \ (\text{Con}^P_e \ v \ \Gamma^{P'} \ , \ a^{P'}) \ (\approx^c_e \ v \ \Gamma^N \approx \Gamma^{N'} \ , \ a \approx a') \\
& \sim (\text{app } \{f = f'\} \{f'\} \{a = a'\} \{a'\} \ t \sim t' \ a \sim a') \ \Gamma^P \ \Gamma^{P'} \ \Gamma^N \approx \Gamma^{N'} = \\
& \quad \sim t \sim t' \ \Gamma^P \ \Gamma^{P'} \ \Gamma^N \approx \Gamma^{N'} \ \text{id}_e \ (\text{Tm}^P \ a \ \Gamma^P) \ (\text{Tm}^P \ a' \ \Gamma^{P'}) \ (\sim a \sim a' \ \Gamma^P \ \Gamma^{P'} \ \Gamma^N \approx \Gamma^{N'})
\end{aligned}$$

The real work happens with the β and η rules. Let us see η :

$$\sim (\eta \ t) \ \{\Gamma^N\} \ \{\Gamma^{N'}\} \ \Gamma^P \ \Gamma^{P'} \ \Gamma^N \approx \Gamma^{N'} \ v \ \{a\} \ \{a'\} \ a^P \ a^{P'} \ a \approx a' = ?$$

The goal has the following type:

$$\text{Tm}^N \ t \ \Gamma^N \ v \ a \approx \text{Tm}^N \ (\text{Tm}_e \ \text{wk } t) \ (\text{Con}^N_e \ v \ \Gamma^{N'} \ , \ a') \ \text{id}_e \ a'$$

Making full use of the induction hypothesis gives us the following:

$$\text{Tm} \approx t \ \Gamma^P \ \Gamma^{P'} \ \Gamma^N \approx \Gamma^{N'} \ v \ a^P \ a^{P'} \ a \approx a' : \text{Tm}^N \ t \ \Gamma^N \ v \ a \approx \text{Tm}^N \ t \ \Gamma^{N'} \ v \ a'$$

Hence, we need to rewrite (or coerce) the right hand side of the result type with an equality.

$$\text{eq} : \text{Tm}^N \ (\text{Tm}_e \ \text{wk } t) \ (\text{Con}^N_e \ v \ \Gamma^{N'} \ , \ a') \ \text{id}_e \ a' \equiv \text{Tm}^N \ t \ \Gamma^{N'} \ v \ a'$$

This should give us a moment of pause. Obviously, we need to prove something about the action of Tm^N on *embedded terms* such as $(\text{Tm}_e \ \text{wk } t)$. However, we haven't even defined an interpretation for embeddings in any model. Embedding and substitution is implicit in our syntax, therefore we are not obliged to interpret them. Note that in the presheaf model in Chapter 5.4, we interpreted *types* as presheaves, and for each type we defined its action on embeddings, with embeddings considered as **OPE** morphisms, but that is not the same as interpreting embeddings themselves, considered as **STLC** morphisms.

We interpret embeddings as natural transformations between semantic contexts:

$$\begin{aligned}
& \text{OPE}^N : \text{OPE} \ \Gamma \ \Delta \rightarrow \forall \ \{\Sigma\} \rightarrow \text{Con}^N \ \Gamma \ \Sigma \rightarrow \text{Con}^N \ \Delta \ \Sigma \\
& \text{OPE}^N \bullet \quad \Gamma^N \quad = \delta^N \\
& \text{OPE}^N \ (\text{drop } \sigma) \ (\Gamma^N \ , \ _) = \text{OPE}^N \ \sigma \ \Gamma^N \\
& \text{OPE}^N \ (\text{keep } \sigma) \ (\Gamma^N \ , \ t^N) = \text{OPE}^N \ \sigma \ \Gamma^N \ , \ t^N
\end{aligned}$$

$\text{OPE}^N\text{-nat} : \forall \sigma \delta \Gamma^N \rightarrow \text{OPE}^N \sigma (\text{Con}^N_e \delta \Gamma^N) \equiv \text{Con}^N_e \delta (\text{OPE}^N \sigma \Gamma^N)$
-- proof omitted

While an embedding encodes that one context can be obtained from another by dropping zero or more elements, a semantic embedding actually performs the dropping of elements on semantic contexts. For instance, $(\text{OPE}^N \text{wk})$ acts as a “tail” function, dropping the topmost element.

Additionally, since the identity embedding is defined recursively (as opposed to as a constructor), we have to prove that it is interpreted as the identity natural transformation:

$\text{OPE}^N\text{-id}_e : \forall \Gamma^N \rightarrow \text{OPE}^N \text{id}_e \Gamma^N \equiv \Gamma^N$

Now, we can give the action of evaluation on embedded terms:

$\epsilon_e^N : \forall \sigma v \Gamma^N \rightarrow \epsilon^N (\epsilon_e \sigma v) \quad \Gamma^N \equiv \epsilon^N v (\text{OPE}^N \sigma \Gamma^N)$
 $\text{Tm}_e^N : \forall \sigma t \Gamma^N \rightarrow \text{Tm}^N (\text{Tm}_e \sigma t) \Gamma^N \equiv \text{Tm}^N t (\text{OPE}^N \sigma \Gamma^N)$

Note that these are essentially *computation* rules, but given as propositional equalities. If the syntax had explicit substitutions (and, by extension, explicit embeddings), these rules would be actual definitional equalities in the evaluation function. It is a trade-off: we forgo these definitional equalities in favor of others stemming from the recursive definition of embedding and substitution, and in favor of the greatly reduced number of conversion rules.

We can turn our attention back to the equality proof required for the η case. By informal equational reasoning:

$\text{Tm}^N (\text{Tm}_e \text{wk } t) (\text{Con}^N_e v \Gamma^{N'}, a') \text{id}_e a' \equiv$
-- by $(\text{Tm}_e^N \text{wk } t (\text{Con}^N_e v \Gamma^{N'}, a'))$
 $\text{Tm}^N t (\text{OPE}^N \text{id}_e (\text{Con}^N_e v \Gamma^{N'})) \text{id}_e a' \equiv$
-- by $(\text{OPE}^N\text{-id}_e (\text{Con}^N_e v \Gamma^{N'}))$
 $\text{Tm}^N t (\text{Con}^N_e v \Gamma^{N'}) \text{id}_e a' \equiv$
-- by $(\text{Tm}^N\text{-nat } t v \{\Gamma^{N'}\} \Gamma^{P'})$
 $\text{Tm}^N t \Gamma^{N'} (v \circ_e \text{id}_e) a' \equiv$
-- by $(\text{idr}_e v)$
 $\text{Tm}^N t \Gamma^{N'} v a'$

In Agda, we can rewrite the goal type with the above equality, then return the induction hypothesis:

```

 $\approx$  (η t) {ΓN} {ΓN'} ΓP ΓP' ΓN≈ΓN' v {a} {a'} aP aP' a≈a'
  rewrite
    TmeN wk t (ConeN v ΓN' , a')
  | OPEN-ide (ConeN v ΓN')
  | TmN-nat t v {ΓN'} ΓP'
  | idre v
  = Tm≈ t ΓP ΓP' ΓN≈ΓN' v aP aP' a≈a'

```

Now, consider the β case:

```

 $\approx$  (β t t') {ΓN} {ΓN'} ΓP ΓP' σ≈δ = ?

```

The goal is $(\text{Tm}^N t (\text{Con}_e^N \text{id}_e \Gamma^N , \text{Tm}^N t' \Gamma^N) \approx \text{Tm}^N (\text{Tm}_s (\text{id}_s , t') t) \Gamma^N)$. Note that the right hand side is the evaluation of a substituted expression. Similarly as with embeddings, an interpretation must be given for substitutions as well as the action of evaluation on substituted expressions.

Substitutions are also interpreted as natural transformations, however, naturality only holds on natural inputs:

```

SubN : Sub Γ Δ → ∀ {Σ} → ConN Γ Σ → ConN Δ Σ
SubN •      ΓN = •
SubN (σ , t) ΓN = SubN σ ΓN , TmN t ΓN

```

```

SubN-nat : ∀ σ δ ΓN → ConP ΓN → SubN σ (ConeN δ ΓN) ≡ ConeN δ (SubN σ ΓN)

```

$(\text{Sub}^N \sigma \Gamma^N)$ evaluates each term in σ in Γ^N , thereby converting a list of terms (substitution) into a list of values (semantic context). We also need two related lemmas:

```

SubN-s◦e : ∀ σ δ ΓN → SubN (σ s◦e δ) ΓN ≡ SubN σ (OPEN δ ΓN)
SubN-ids : ∀ ΓN → SubN ids ΓN ≡ ΓN

```

There is no discernible deep reason why we need $_s \circ_e _$ but not the other variants; only this lemma is needed in the rest of the soundness proof.

The action of evaluation on substituted terms is a bit more involved. Naturality must be assumed for the input semantic context, or else the proof does not go through -

in fact, this is the reason why naturality predicates were introduced. This implies that the action of evaluation cannot be stated as propositional equality, only as $\underline{\approx}$, the previously defined equality of semantic values.

$$\text{Tm}_s^N : \forall \sigma \, t \, \Gamma^N \rightarrow \text{Con}^P \, \Gamma^N \rightarrow \Gamma^N \approx^c \Gamma^N \rightarrow \text{Tm}^N \, (\text{Tm}_s \, \sigma \, t) \, \Gamma^N \approx \text{Tm}^N \, t \, (\text{Sub}^N \, \sigma \, \Gamma^N)$$

Now, Tm_s^N can be used to prove a $\underline{\approx}$ where the right hand side matches the goal type for the β case.

$$\begin{aligned} & (\text{Tm}_s^N \, (\text{id}_s \, , \, t') \, t \, \Gamma^{N'} \, \Gamma^{P'} \, ((\sigma \approx \delta \approx^{c-1}) \approx^{c \blacksquare} \sigma \approx \delta) \approx^{-1}) \\ & : \text{Tm}^N \, t \, (\text{Sub}^N \, \text{id}_s \, \Gamma^{N'} \, , \, \text{Tm}^N \, t' \, \Gamma^{N'}) \approx \text{Tm}^N \, (\text{Tm}_s \, (\text{id}_s \, , \, t') \, t) \, \Gamma^{N'} \end{aligned}$$

We can use transitivity of $\underline{\approx}$ and erase semantic id_e actions, after which the goal becomes

$$\text{Tm}^N \, t \, (\Gamma^N \, , \, \text{Tm}^N \, t' \, \Gamma^N) \approx \text{Tm}^N \, t \, (\Gamma^{N'} \, , \, \text{Tm}^N \, t' \, \Gamma^{N'})$$

Note that the evaluation context differs on the sides, but the evaluated term is the same. Since the evaluation contexts are related by $\underline{\approx}$, $\text{Tm} \approx$ can be used to prove the goal, which completes the proof of $\sim \approx$.

$$\begin{aligned} & (\text{Tm} \approx \, t \, (\Gamma^P \, , \, \text{Tm}^P \, t' \, \Gamma^P) \, (\Gamma^{P'} \, , \, \text{Tm}^P \, t' \, \Gamma^{P'}) \, (\sigma \approx \delta \, , \, (\text{Tm} \approx \, t' \, \Gamma^P \, \Gamma^{P'} \, \sigma \approx \delta))) \\ & : \text{Tm}^N \, t \, (\Gamma^N \, , \, \text{Tm}^N \, t' \, \Gamma^N) \approx \text{Tm}^N \, t \, (\Gamma^{N'} \, , \, \text{Tm}^N \, t' \, \Gamma^{N'}) \end{aligned}$$

After this, quoting, unquoting and soundness is easily given.

mutual

$$\begin{aligned} q \approx & : \forall \{A \, \Gamma\} \{t \, t' : \text{Ty}^N \, A \, \Gamma\} \rightarrow t \approx t' \rightarrow q^N \, t \equiv q^N \, t' \\ q \approx & \{ \iota \} \quad t \approx t' = t \approx t' \\ q \approx & \{A \Rightarrow B\} \, t \approx t' = \text{lam} \, \& \, q \approx (t \approx t' \, (\text{wk} \, \{A\}) \, (\text{u}^P \, _) \, (\text{u}^P \, _) \, (\text{u} \approx \text{refl})) \end{aligned}$$

$$\begin{aligned} u \approx & : \forall \{A \, \Gamma\} \{n \, n' : \text{Ne} \, \Gamma \, A\} \rightarrow n \equiv n' \rightarrow u^N \, n \approx u^N \, n' \\ u \approx & \{ \iota \} \quad p = \text{ne} \, \& \, p \\ u \approx & \{A \Rightarrow B\} \, p = \lambda \, \sigma \, a^P \, a^{P'} \, a \approx a' \rightarrow u \approx (\text{app} \, \& \, (\text{Ne}_e \, \sigma \, \& \, p) \, \otimes \, q \approx a \approx a') \end{aligned}$$

$$\begin{aligned} u^{c \approx} & : \forall \{ \Gamma \} \rightarrow u^{c \, N} \, \{ \Gamma \} \approx^c u^{c \, N} \\ u^{c \approx} & \{ \bullet \} = \bullet \\ u^{c \approx} & \{ \Gamma \, , \, A \} = \approx^{c \, e} \, \text{wk} \, u^{c \approx} \, , \, u \approx \text{refl} \end{aligned}$$

$$\begin{aligned} \text{sound} & : \forall \{ \Gamma \, A \} \{ t \, t' : \text{Tm} \, \Gamma \, A \} \rightarrow t \sim t' \rightarrow \text{nf} \, t \equiv \text{nf} \, t' \\ \text{sound} & \, t \sim t' = q \approx (\sim \approx \, t \sim t' \, u^{c \, P} \, u^{c \, P} \, u^{c \approx}) \end{aligned}$$

□

5.6 Stability

Stability expresses that normalization has no action on normal terms:

```
stable : (n : Ne  $\Gamma$  A)  $\rightarrow$  nf  $\Gamma$  n  $\neg$  Nf  $\equiv$  n
```

It differs in several ways from the other two correctness properties. First, it is not quite as crucial; failing either soundness or completeness results in something else than a normalization algorithm, but instability is tolerable. A primary goal of normalization is to have a decision procedure for conversion, however, it is achievable without stability:

```
-- decidable Nf equality, given by simple induction
Nf $\equiv$ ? : (n n' : Nf  $\Gamma$  A)  $\rightarrow$  (n  $\equiv$  n')  $\vee$  (n  $\equiv$  n'  $\rightarrow \perp$ )

decidableConversion : (t t' : Tm  $\Gamma$  A)  $\rightarrow$  (t  $\sim$  t')  $\vee$  (t  $\sim$  t'  $\rightarrow \perp$ )
decidableConversion t t' with Nf $\equiv$ ? (nf t) (nf t')
... | inj1 p =
  inj1 (complete t  $\sim$ ■ coe (( $\lambda$  x  $\rightarrow$   $\Gamma$  x  $\neg$  Nf  $\sim$  t') & p-1) (complete t'  $\sim$ -1))
... | inj2 p =
  inj2 ( $\lambda$  q  $\rightarrow$  p (sound q))
```

Rather, stability tells us whether a normalization algorithm matches up exactly with a definition of normal forms. In our case, if we leave normalization essentially unchanged, but do not mandate η -long normal forms as we currently do, there would be η -short normal forms which are η -expanded by normalization:

```
t : Tm ( $\mathbf{1} \Rightarrow \mathbf{1}$ ) ( $\mathbf{1} \Rightarrow \mathbf{1}$ )
t = var vz

nf- $\eta$ -expands : nf t  $\equiv$  (lam (app (var (vs vz)) (var vz)))
nf- $\eta$ -expands = refl
```

Here, normalization “thinks” that normal forms are more restricted than they actually are. To regain stability, normalization would have to be modified so that it does not do any η -conversion and leaves neutral terms unchanged. That said, an unstable

algorithm is still fine - it just does “more” than what is required. One could argue that instability should be avoided because unstable algorithms perform unnecessary computation.

Stability is much simpler to prove than previous correctness properties. Simple mutual induction on normal and neutral terms suffices, making use of previously proved naturalities and straightforward equational reasoning.

```

stableE : (v : A ∈ Γ) → EN v uc N ≡ uN (var v)
stableE vz      = refl
stableE (vs v) =
  EN-nat v wk uc N
  ■ TyNe wk & stableE v
  ■ uN-nat wk (var v)
  ■ (λ x → uN (var (vs x))) & E-ide v

```

mutual

```

stable : (n : Nf Γ A) → nf Γ n ⊢ Nf ≡ n
stable (ne n) = stableNe n
stable (lam n) = lam & stable n

```

```

stableNe : (n : Ne Γ A) → TmN Γ n ⊢ Ne uc N ≡ uN n
stableNe (var v) = stableE v
stableNe (app f a) =
  (λ x → x ide (TmN Γ a ⊢ Nf uc N)) & stableNe f
  ■ (λ x → uN (app (Nee ide f) x)) & stable a
  ■ (λ x → uN (app x a)) & Ne-ide f

```

□

Chapter 6

Variations

In this chapter we look at two alternate variants of the formalization with different design choices. The first one is more compact but includes a less transparent implementation of normalization. The second one contains a more efficient evaluation function at the cost of slightly larger proofs.

6.1 Direct Presheaf Model

In Chapter 5.4 the presheaf model of STLC was described, but not pursued to its full extent. Going for a full presheaf model lets us dispense with about 100 lines of Agda, which is a significant portion of the roughly 700 lines of the whole development. The formalization can be found at <https://github.com/AndrasKovacs/stlc-nbe/tree/master>.

The \mathcal{N} model is changed to a presheaf model, therefore it includes all mutually defined naturality proofs. The interpretation of types is now the same as \mathbf{Ty}^p in Chapter 5.4, although here we also use \mathcal{N} superscripts for the presheaf model¹.

A benefit of this approach is that the \mathbf{Ty}^p logical predicate for naturality can be

¹It stands for “normalization”, which it achieves, but by a different model than previously.

entirely dropped, since naturality is already baked into Ty^N . The drawback is that functor laws and naturalities must be defined mutually with evaluation. We list here the types (only the types, for brevity) of the definitions required to implement normalization, along with brief descriptions of categorical interpretations. Below, $(f : \text{PSh}(\text{OPE})(A, B))$ means that f is a natural transformation in $\text{PSh}(\text{OPE})$ between A and B presheaves.

```
-- Presheaf.agda

-- Ty^N A : PSh(OPE)
-----
Ty^N   : Ty → Con → Set                -- action on objects
Ty^N_e : OPE Δ Γ → Ty^N A Γ → Ty^N A Δ -- action on morphisms

Ty^N-id_e : ∀ t^N → Ty^N_e id_e t^N ≡ t^N
Ty^N-∘_e  : ∀ t^N σ δ → Ty^N_e (σ ∘_e δ) t^N ≡ Ty^N_e δ (Ty^N_e σ t^N)

-- Con^N Γ : PSh(OPE)
-----
Con^N   : Con → Con → Set                -- action on objects
Con^N_e : OPE Δ Σ → Con^N Γ Σ → Con^N Γ Δ -- action on morphisms

Con^N-id_e : ∀ Γ^N → Con^N_e id_e Γ^N ≡ Γ^N
Con^N-∘_e  : ∀ σ δ Γ^N → Con^N_e (σ ∘_e δ) Γ^N ≡ Con^N_e δ (Con^N_e σ Γ^N)

-- ∈^N {Γ}{A} v : PSh(OPE)(Con^N Γ, Ty^N A)
-----
∈^N      : ∀ {Γ A} → A ∈ Γ → ∀ {Δ} → Con^N Γ Δ → Ty^N A Δ
∈^N-nat  : ∀ (v : A ∈ Γ) σ Γ^N → ∈^N v (Con^N_e σ Γ^N) ≡ Ty^N_e σ (∈^N v Γ^N)

-- Tm^N {Γ}{A} t : PSh(OPE)(Con^N Γ, Ty^N A)
-----
Tm^N     : ∀ {Γ A} → Tm Γ A → ∀ {Δ} → Con^N Γ Δ → Ty^N A Δ
Tm^N-nat : ∀ t σ Γ^N → Tm^N t (Con^N_e σ Γ^N) ≡ Ty^N_e σ (Tm^N t Γ^N)

-- q^N {A} : PSh(OPE)(Ty^N A, Nf _ A)
-----
q^N      : ∀ {A Γ} → Ty^N A Γ → Nf Γ A
q^N-nat  : ∀ σ t^N → Nf_e σ (q^N t^N) ≡ q^N (Ty^N_e σ t^N)
```

```

-- uN {A} : PSh(OPE)(Ne _ A, TyN A)
-----
uN      : ∀ {A Γ} → Ne Γ A → TyN A Γ
uN-nat  : ∀ σ n → TyNe σ (uN n) ≡ uN (Nee σ n)

-----
ucN    : ∀ {Γ} → ConN Γ Γ
nf      : ∀ {Γ A} → Tm Γ A → Nf Γ A

```

The proofs for completeness and stability are largely unchanged. For soundness, we again need to interpret embeddings and substitutions and define the action of evaluation on them.

```

-- Presheaf.agda

-- OPEN {Γ}{Δ} σ : PSh(OPE)(ConN Γ, ConN Δ)
-----
OPEN      : ∀ {Γ Δ} → OPE Γ Δ → ∀ {Σ} → ConN Γ Σ → ConN Δ Σ
OPEN-nat  : ∀ σ δ ΓN → OPEN σ (ConNe δ ΓN) ≡ ConNe δ (OPEN σ ΓN)

OPEN-ide : ∀ ΓN → OPEN ide ΓN ≡ ΓN
TmeN     : ∀ σ t ΓN → TmN (Tme σ t) ΓN ≡ TmN t (OPEN σ ΓN)

-- SubN {Γ}{Δ} σ : PSh(OPE)(ConN Γ, ConN Δ)
-----
SubN      : ∀ {Γ Δ} → Sub Γ Δ → ∀ {Σ} → ConN Γ Σ → ConN Δ Σ
SubN-nat  : ∀ σ δ ΓN → SubN σ (ConNe δ ΓN) ≡ ConNe δ (SubN σ ΓN)

SubN-ids : ∀ ΓN → SubN ids ΓN ≡ ΓN
SubN-soe : ∀ σ δ ΓN → SubN (σ soe δ) ΓN ≡ SubN σ (OPEN δ ΓN)
TmsN     : ∀ σ t ΓN → TmN (Tms σ t) ΓN ≡ TmN t (SubN σ ΓN)

```

The soundness proof is dramatically simpler this time. The reason is that propositional equality is already right for semantic values, so there is no need for any additional \approx logical relation, and propositional equality is already an equivalence relation which is respected by all constructions in the metatheory. Hence, soundness is simply given as follows:

```
-- Soundness.agda
~~~ : ∀ {t t' : Tm Γ A} → t ~ t' → (σ : ConN Γ Δ) → TmN t σ ≡ TmN t' σ

sound : ∀ {Γ A}{t t' : Tm Γ A} → t ~ t' → nf t ≡ nf t'
sound t~t' = qN & ~~~ t~t' ucN
```

Here, the proof for $\sim\sim$ still makes use of \mathbf{OPE}^N and \mathbf{Sub}^N , but it is much simpler than before and altogether takes 11 lines of Agda.

There is a slight disadvantage of the direct presheaf model: because normalization is defined mutually with naturalities, and some naturality proofs refer to function extensionality, it is not anymore trivial that normalization always computes. What if normalization gets stuck on the postulated extensionality? It is easy to prove that normalization does compute, but this proof can only be done *external to Agda*. Internally to Agda we obviously have no means to reason about postulates or distinguish them from defined proofs.

6.2 More Efficient Evaluation

We describe a more efficient evaluation function in this section. The correctness proofs are also formalized for this version, requiring only minor modifications. The formalization can be found at <https://github.com/AndrasKovacs/stlc-nbe/blob/efficient-appN>.

Let us review the definition of \mathbf{Tm}^N :

```
TmN : ∀ {Γ A} → Tm Γ A → ∀ {Δ} → ConN Γ Δ → TyN A Δ
TmN (var v)   ΓN = ∈N v ΓN
TmN (lam t)   ΓN = λ σ aN → TmN t (ConNe σ ΓN , aN)
TmN (app f a) ΓN = TmN f ΓN ide (TmN a ΓN)
```

Notice the \mathbf{id}_e in the $(\mathbf{app} \ f \ a)$ case. After each application to \mathbf{id}_e , the Γ^N context is changed to $(\mathbf{Con}^N_e \ \mathbf{id}_e \ \Gamma^N)$ during the evaluation of the internal t body of $(\mathbf{Tm}^N \ f \ \Gamma^N)$. This means that every application has an overhead proportional to the size of the

current context. Also, recall Ty^N_e for functions:

$$\text{Ty}^N_e \{A \Rightarrow B\} \sigma \ t^N = \lambda \delta \ a^N \rightarrow t^N (\sigma \circ_e \delta) \ a^N$$

$(\text{Ty}^N_e \text{ id}_e \ t^N)$ creates a new closure around t^N which has essentially no effect, and these closures accumulate on the top of semantic functions in Γ^N after each $(\text{app } f \ a)$ evaluation. This is certainly bad for performance. We would like evaluation to be essentially as efficient for weak reduction as the standard model (as defined in Chapter 4.2). This can be achieved by defining id_e and Con^N_e in a way such that $(\text{Con}^N_e \text{ id}_e \ \Gamma^N)$ immediately reduces to Γ^N .

First, id_e needs to be a primitive constructor:

```
-- Embedding.agda
data OPE : Con → Con → Set where
  id_e   : ∀ {Γ} → OPE Γ Γ
  drop   : ∀ {A Γ Δ} → OPE Γ Δ → OPE (Γ , A) Δ
  keep   : ∀ {A Γ Δ} → OPE Γ Δ → OPE (Γ , A) (Δ , A)
```

This slightly complicates the identity functor law for $(\text{Tm } _ \ A)$ and $(A \in _)$, since identity embeddings are not unique anymore. We define a predicate on embeddings which picks out the identities:

```
Id_e : ∀ {Γ} → OPE Γ Γ → Set
Id_e id_e      = τ
Id_e (drop σ) = ⊥
Id_e (keep σ)  = Id_e σ
```

Id_e embeddings are id_e -s wrapped in zero or more keep -s. We prove identity functor laws using Id_e :

```
∈-id_e : ∀ (v : A ∈ Γ) {σ : OPE Γ Γ} {_ : Id_e σ} → ∈_ σ v ≡ v
Tm-id_e : ∀ (t : Tm Γ A) {σ : OPE Γ Γ} {_ : Id_e σ} → Tm_e σ t ≡ t
```

Secondly, Con^N_e is redefined as follows:

```
Con^N_e : OPE Σ Δ → Con^N Γ Δ → Con^N Γ Σ
Con^N_e id_e      Γ^N      = Γ^N
Con^N_e (drop σ) •      = •
Con^N_e (drop σ) (Γ^N , t^N) = (Con^N_e (drop σ) Γ^N) , Ty^N_e (drop σ) t^N
```

$$\begin{aligned} \text{Con}^{\text{N}_e} (\text{keep } \sigma) \bullet &= \bullet \\ \text{Con}^{\text{N}_e} (\text{keep } \sigma) (\Gamma^{\text{N}}, \mathbf{t}^{\text{N}}) &= (\text{Con}^{\text{N}_e} (\text{keep } \sigma) \Gamma^{\text{N}}, (\text{Ty}^{\text{N}_e} (\text{keep } \sigma) \mathbf{t}^{\text{N}})) \end{aligned}$$

We get the desired definitional equality, but in turn lose two equalities which are quite useful to have. We can resurrect them as propositional equalities:

```
-- PresheafRefinement.agda
ConNe - • : ∀ σ → ConNe σ • ≡ •
ConNe -, : ∀ σ ΓN tN → ConNe σ (ΓN, tN) ≡ (ConNe σ ΓN, TyNe σ tN)
```

We must manually rewrite along these equalities when necessary. It is not a significant burden though. Altogether, the changes described here result in about 20 additional lines in the formalization.

We eliminated the overhead on the evaluation of application, but what about quoting? It also contains semantic application:

```
qN : ∀ {A Γ} → TyN A Γ → Nf Γ A
qN {ι}      tN = tN
qN {A ⇒ B} tN = lam (qN (tN wk (uN (var vz))))
```

Here, the $(\mathbf{t}^{\text{N}} \text{ wk})$ is essential, but it is far less of an overhead than the id_e -s before. Notice that \mathbf{q}^{N} is *type-directed*, and for each semantic value it is called as many times as there are function arguments in the value's type. Of course, \mathbf{q}^{N} is applied to various sub-terms of a term during normalization (via \mathbf{u}^{N}), so the overall number of \mathbf{q}^{N} calls is likely greater than the number of arguments in a term's type. Still, it is only to be expected that *normalization* is more costly than standard interpretation, since it does more work. *Evaluation* on its own is no less efficient than standard interpretation, with the current optimized implementation.

Chapter 7

Discussion and Future Work

The primary goal of this development was to formalize $\beta\eta$ -normalization for STLC as simply and compactly as possible, with the sub-goal of implementing normalization itself in a transparent and efficient manner. Overall, these goals are achieved in the main formalization and its two alternate versions as well, with slightly varying trade-offs with respect to the desiderata. In particular, our development has the following advantages:

- Our syntax is conventional and matches the common presentation in pedagogical literature. Also, the used metatheory is conservative.
- The formalization is complete: no postulates (aside from function extensionality) or holes remain.
- The formalization is compact: not counting the standard library, the main version is roughly 700 lines, while the direct presheaf version is 600 lines.
- The normalization algorithm is structurally recursive, obviating the need for termination proofs.

As to possible extensions and improvements to this work, while it seems that the current approach is uniquely well-suited to STLC and its non-polymorphic extensions, there are complications and rough edges when we attempt to scale to more powerful

type theories.

For System F with intrinsic syntax and implicit substitutions, this author has an unpublished mostly-complete formalization of an NbE algorithm [26], although there are no correctness proofs yet. It seems that System F_ω is about as far as we can take intrinsic syntax with implicit substitutions. With full dependent types, intrinsic syntax only seems feasible with higher-inductive definitions and explicit substitutions. Implicit substitutions remain only feasible with extrinsic syntax.

Another possible improvement would be proof automation for substitution and embedding laws. A significant part of the development is essentially “boilerplate” related to this. For extrinsic syntax, there are very powerful automation libraries available in the proof assistant Coq [25, 32]. For intrinsic syntax, automatic generation of substitution lemmas seems more difficult, but even if we prove categorical lemmas by hand, we could still make use of automation for proving ad hoc term equations. This seems doable in Agda, although the automation facilities of Agda are not as nearly mature as that of Coq.

Bibliography

- [1] Andreas Abel. “Normalization by Evaluation: Dependent Types and Impredicativity”. Habilitation thesis. PhD thesis. Ludwig-Maximilians-Universität München, 2013.
- [2] Thorsten Altenkirch. “Normalisation by completeness”. In: *Talk given at the Workshop on Normalization by Evaluation in Los Angeles*. 2009.
- [3] Thorsten Altenkirch and James Chapman. “Big-step normalisation”. In: *Journal of Functional Programming* 19.3-4 (2009), pp. 311–333.
- [4] Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. “Categorical reconstruction of a reduction free normalization proof”. In: *Category Theory and Computer Science*. Springer. 1995, pp. 182–199.
- [5] Thorsten Altenkirch and Ambrus Kaposi. “Normalisation by evaluation for dependent types”. In: *LIPICs-Leibniz International Proceedings in Informatics*. Vol. 52. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2016.
- [6] Thorsten Altenkirch and Ambrus Kaposi. “Type theory in type theory using quotient inductive types”. In: *ACM SIGPLAN Notices*. Vol. 51. 1. ACM. 2016, pp. 18–29.
- [7] Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. “Observational equality, now!” In: *Proceedings of the 2007 workshop on Programming languages meets program verification*. ACM. 2007, pp. 57–68.

- [8] Lennart Augustsson and Magnus Carlsson. *An exercise in dependent types: A well-typed interpreter*. 1999.
- [9] Nick Benton et al. “Strongly typed term representations in Coq”. In: *Journal of automated reasoning* 49.2 (2012), pp. 141–159.
- [10] Ulrich Berger and Helmut Schwichtenberg. “An inverse of the evaluation functional for typed lambda-calculus”. In: *Logic in Computer Science, 1991. LICS’91., Proceedings of Sixth Annual IEEE Symposium on*. IEEE. 1991, pp. 203–211.
- [11] Ana Bove and Venanzio Capretta. “Modelling general recursion in type theory”. In: *Mathematical Structures in Computer Science* 15.04 (2005), pp. 671–708.
- [12] Edwin Brady. “Idris, a general-purpose dependently typed programming language: Design and implementation”. In: *Journal of Functional Programming* 23.05 (2013), pp. 552–593.
- [13] Simon Castellan. *Dependent type theory as the initial category with families*. 2014. URL: <http://iso.mor.phis.me/archives/2011-2012/stage-2012-goteburg/report.pdf> (visited on 03/10/2017).
- [14] Cyril Cohen et al. “Cubical type theory: a constructive interpretation of the univalence axiom”. In: *arXiv preprint arXiv:1611.02108* (2016).
- [15] nLab contributors. *closed monoidal structure on presheaves*. 2017. URL: <https://ncatlab.org/nlab/show/closed+monoidal+structure+on+presheaves> (visited on 04/07/2017).
- [16] nLab contributors. *preorder*. 2017. URL: <https://ncatlab.org/nlab/show/preorder> (visited on 04/07/2017).
- [17] nLab contributors. *wide subcategory*. 2017. URL: <https://ncatlab.org/nlab/show/wide+subcategory> (visited on 04/07/2017).
- [18] Catarina Coquand. “A formalised proof of the soundness and completeness of a simply typed lambda-calculus with explicit substitutions”. In: *Higher-Order and Symbolic Computation* 15.1 (2002), pp. 57–90.

- [19] Catarina Coquand. “A proof of normalization for simply typed lambda calculus written in ALF”. In: *Proceedings of the Workshop on Types for Proofs and Programs*. Citeseer. 1992, pp. 85–92.
- [20] Thierry Coquand and Peter Dybjer. “Intuitionistic model constructions and normalization proofs”. In: *Mathematical Structures in Computer Science* 7.1 (1997), pp. 75–94.
- [21] Agda developers. *Agda documentation*. 2017. URL: <https://agda.readthedocs.io/en/v2.5.2/> (visited on 03/10/2017).
- [22] Agda developers. *Agda standard library*. 2017. URL: <https://github.com/agda/agda-stdlib> (visited on 03/10/2017).
- [23] Healfdene Goguen, Conor McBride, and James McKinna. “Eliminating dependent pattern matching”. In: *Algebra, Meaning, and Computation* (2006), pp. 521–540.
- [24] Robert Harper. *Practical foundations for programming languages*. Cambridge University Press, 2016.
- [25] Steven Keuchel, Stephanie Weirich, and Tom Schrijvers. “Needle & Knot: Binder Boilerplate Tied Up”. In: *Proceedings of the 25th European Symposium on Programming Languages and Systems - Volume 9632*. New York, NY, USA: Springer-Verlag New York, Inc., 2016, pp. 419–445. ISBN: 978-3-662-49497-4. DOI: 10.1007/978-3-662-49498-1_17. URL: http://dx.doi.org/10.1007/978-3-662-49498-1_17.
- [26] András Kovács. *Normalization by evaluation for intrinsic System F*. 2017. URL: <https://github.com/AndrasKovacs/sysF-NbE> (visited on 04/07/2017).
- [27] Per Martin-Löf. “An intuitionistic theory of types: Predicative part”. In: *Studies in Logic and the Foundations of Mathematics* 80 (1975), pp. 73–118.
- [28] Conor McBride. *Datatypes of Datatypes*. 2015. URL: <http://staff.mmcs.sfedu.ru/~ulysses/Edu/SSGEP/conor/conor.pdf>.
- [29] Conor McBride and Ross Paterson. “Applicative programming with effects”. In: *Journal of functional programming* 18.01 (2008), pp. 1–13.

- [30] Ulf Norell. *Towards a practical programming language based on dependent type theory*. Vol. 32. Citeseer, 2007.
- [31] Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.
- [32] Steven Schäfer, Tobias Tebbi, and Gert Smolka. “Autosubst: Reasoning with de Bruijn Terms and Parallel Substitutions”. In: *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015*. Ed. by Xingyuan Zhang and Christian Urban. LNAI. Springer-Verlag, Aug. 2015.
- [33] Thomas Streicher. *Investigations into intensional type theory*. <http://www.mathematik.tu-darmstadt.de/~streicher/HabilStreicher.pdf>, 1993.
- [34] Aaron Stump. *Verified Functional Programming in Agda*. New York, NY, USA: Association for Computing Machinery and Morgan & Claypool, 2016. ISBN: 978-1-97000-127-3.
- [35] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study: <https://homotopytypetheory.org/book>, 2013.