25x8X?

5555

A Machine-Checked Correctness Proof of Normalization by Evaluation for Simply Typed Lambda Calculus

Author: András Kovács Advisor: Ambrus Kaposi

Budapest, 2017

Contents

Chapter 1

Introduction

1.1 Overview

Normalization animates the syntax of typed -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 -normalization for simply typed -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.

[1] In particular, the types of System F largely correspond to simply typed -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 -normalization.

If we are to trust a particular normalization algorithm, we need to prove the following properties (we borrow terminology from [**altenkirch2009big**]):

- *Completeness*: terms are convertible to their normal forms.
- *Soundness*: normalization takes convertible terms to the same normal form.

Additionally, we may require stability, which establishes that there is no redundancy in normal forms:

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

Chapter **??** describes the metatheory we work in. In Chapter **??**, we formalize the syntax of STLC along with -conversion. Chapter **??** first presents the normalization function, then proceeds with its correctness proofs. In Chapter **??** we describe 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 is available online at *https://github.com/AndrasKovacs/stlc-nbe*.

1.2 Related Work

Martin-Löf [**martin1975intuitionistic**] first used NbE (though the term has 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 [**berger1991inverse**] as a normalization method for lambda calculi. Abel's work [**abel2013normalization**] 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 [**coquand2002formalised**] 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 [**altenkirch2009big**] 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 [**bove2005modelling**].

Altenkirch and Kaposi [**altenkirch2016normalisation**] 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 **??**, 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 [**altenkirch1995categorical**].

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 [**hottbook**]. 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* `Set₀` *and* `Set₁`*, with large elimination into* `Set₀`. `Set₀` 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 [**coquand2002formalised**]. 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 [**cohen2016cubical**] and observational [**altenkirch2007observational**] 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 [**streicher1993investigations**]. We use the `--without-K` language option for all of our code in the main development. However, we do use axiom K in Chapter **??** 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 [**Norell:2007thesis**]. The latest documentation is available online [**agdadocs**]. For a book-length introduction geared towards beginners, see [**Stump:agdabook**]. We summarize here the core constructions and our notational liberties in relation to them.

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 consistency[1]. 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)
```

[1] Agda 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 de given as `record {S = ...; _⊗_ = ...; assoc = ...}`. Also, 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 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`.

Agda 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. Also, injectivity and disjointness of term constructors is implicit in pattern matching, while they have to be separately proven when using eliminators.

As a notational convention, we shall take cues from the Idris [**brady2013idris**] 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.

2.2.2 Included Code Examples

The project can be found in *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. When a code example here corresponds to some code in the repository, we indicate the name of the source file in the examples as Agda comments or in expository text.

2.2.3 Standard Library

Our development does not have any external dependencies. We need less than a hundred lines of code from the Agda standard library [**agda-stdlib**], 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 out 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 [**hottbook**]:

```
_■_ : 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 : a₀ ≡ a₁)`, `(q : b₀ ≡ b₁)` and `(f : A → B → C)`, then `(f & p ⊗ q)` has type `(f a₀ b₀ ≡ f a₁ b₁)`. Note that both operators associate left, and `_&_` binds more strongly. This is reminiscent of the lifting syntax with applicative functors [**mcbride2008applicative**] in Haskell programming.

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

```
coe : A ≡ B → A → B
coe refl a = a
```

`transport`/`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), `⊤` (unit type), `⊥` (empty type) and `_⊎_` (disjoint union) from the standard library. Elements of `Σ` are constructed as `(a , b)` pairs, and the projections are `proj₁` and `proj₂`. `A × B` is defined in terms of `Σ` as the non-dependent pair type. `⊤` has `tt` ("trivially true") as sole element. Injection into disjoint unions is done with `inj₁` and `inj₂`. The *ex falso* principle is named `⊥-elim`, with type `{A : Set} → A → 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 Base Syntax

The complete definition is the following:

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

data Con : Set where
  •   : Con
```

[1] We take the view that the "proper" definition of the syntax would be an initial object in some suitable category of models, e. g. the initial category with families. In this way, models include conversion as propositional equations, so the syntax is quotiented by conversion as well; see e. g. [**initialCwF**] and [**tt-in-tt**]. 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.

```
  _,_ : 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. `_∈_` 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-ι : Tm • (ι ⇒ ι)
id-ι = lam (var vz) -- with nameful syntax: λ (x : ι) . x
```

Th 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 _⊢_∈_ (Γ : Con) → Tm → Ty → Set where -- omitted constructors
```

Now, `Γ ⊢ t ∈ 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. [**benton2012strongly**] for approaches to dealing with them. For extrinsic syntax, powerful automation for reasoning about substitution is available in Coq, allowing spectacularly compact proofs [**autosubst**]. 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 [**tt-in-tt**], 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 -conversion, we need a notion of substitution. Also, the -rule must refer to a notion of *weakening* or *embedding*: it mentions the "same" term under and over a binder, but the two mentions 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 -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 Γ Δ → OPE (Γ , A) Δ
  keep : OPE Γ Δ → OPE (Γ , A) (Δ , A)
```

Elements of `OPE Γ Δ` express that `Δ` can be obtained from `Γ` by dropping zero or

more entries in order.

Some choices can be made here. C. Coquand [**coquand1992proof**] uses an explicit identity embedding constructor with type `(∀ {Γ} → OPE Γ Γ)` instead of our `•`. This slightly simplifies some proofs and slightly complicates others. It does not have propositionally unique identity embeddings[2], 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 [**mcbride2015datatypes**]:

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

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:

```
∈ₑ : OPE Γ Δ → A ∈ Δ → A ∈ Γ
∈ₑ •        v      = v
∈ₑ (drop σ) v      = vs (∈ₑ σ v)
∈ₑ (keep σ) vz     = vz
∈ₑ (keep σ) (vs v) = vs (∈ₑ σ v)

Tmₑ : OPE Γ Δ → Tm Δ A → Tm Γ A
Tmₑ σ (var v)   = var (∈ₑ σ v)
Tmₑ σ (lam t)   = lam (Tmₑ (keep σ) t)
Tmₑ σ (app f a) = app (Tmₑ σ f) (Tmₑ σ a)
```

Identity embeddings keep every entry:

```
idₑ : ∀ {Γ} → OPE Γ Γ
idₑ {•}     = •
```

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

```
idₑ {Γ , A} = keep (idₑ {Γ})
```

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

```
wk : ∀ {A Γ} → OPE (Γ , A) Γ
wk = drop idₑ
```

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 [**pierce2002types**, **harper2016practical**].

2. Lighter formalization. With implicit substitutions, the definition of conversion becomes significantly smaller. With explicit substitutions, we need congruence closure of and 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 [**altenkirch2009big**] 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 **??**.

Strictly speaking, stating -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 `Sub Γ Δ` contain a `Tm Γ A` for each `A` type in `Δ`. In other words, an element of `Sub Γ Δ` 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 [**mcbride2015datatypes**]), 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 `(∀ {A Γ Δ} → Sub Γ Δ → Sub (Γ , A) (Δ , A))`. This enables pushing substitutions under -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) , Tmₑ δ t
```

The notation `_ₛ∘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 **??**.

There is a canonical injection `⌜_⌝ᵒᵖᵉ` from `OPE` to `Sub`:

```
dropₛ : Sub Γ Δ → Sub (Γ , A) Δ
dropₛ σ = σ ₛ∘ₑ wk

keepₛ : Sub Γ Δ → Sub (Γ , A) (Δ , A)
keepₛ σ = dropₛ σ , var vz

⌜_⌝ᵒᵖᵉ : OPE Γ Δ → Sub Γ Δ
⌜ •      ⌝ᵒᵖᵉ = •
⌜ drop σ ⌝ᵒᵖᵉ = dropₛ ⌜ σ ⌝ᵒᵖᵉ
⌜ keep σ ⌝ᵒᵖᵉ = keepₛ ⌜ σ ⌝ᵒᵖᵉ
```

We note that `keepₛ` is the needed operation for pushing substitutions under binders. Now the action of substitution on terms and variables can be defined:

```
∈ₛ : Sub Γ Δ → A ∈ Δ → Tm Γ A
∈ₛ (σ , t) vz     = t
∈ₛ (σ , t) (vs v) = ∈ₛ σ v

Tmₛ : Sub Γ Δ → Tm Δ A → Tm Γ A
Tmₛ σ (var v)   = ∈ₛ σ v
Tmₛ σ (lam t)   = lam (Tmₛ (keepₛ σ) t)
Tmₛ σ (app f a) = app (Tmₛ σ f) (Tmₛ σ a)
```

`∈ₛ` simply looks up a term from a substitution[3], while `Tmₛ` recurses into terms to substitute all variables. Identity substitutions can be defined as well:

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

Single substitution with a `(t : Tm Γ A)` term is given by `(idₛ , t)`. This assigns

[3]The immediate reason for not naming `∈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 `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 `t` under a lambda with `(Tmₑ wk)` in `η`, and use single substitution for `β`. The rest are rules for congruence (`lam`, `app`) and equivalence closure (`~refl`, `_~⁻¹`, `_~■_`). The syntax for equivalence rules is the same as for propositional equality, except for the `~` 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 of 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. -normality is obvious, since only variables can be applied. Normal forms are also -long: since `ne` only injects

neutrals of base type, all normal terms of function type must in fact be lambdas. Only requiring -normality would be as simple as having `(ne : ∀ {A} → Ne Γ A → Nf Γ 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 -- definitions omitted
Neₑ : OPE Γ Δ → Ne Δ A → Ne Γ A -- definitions omitted
```

These are defined by straightforward mutual recursion.

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 [**augustsson1999exercise**] for STLC as an embedded language. It is implemented as follows:

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

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

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

```
Tmˢ : ∀ {Γ A} → Tm Γ A → (Conˢ Γ → Tyˢ A)
Tmˢ (var v)   Γˢ = ∈ˢ 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
⟦ ι     ⟧ = ⊥
⟦ 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 `M` we use `Γᴹ`, `Δᴹ` etc. for elements of semantic contexts, and `aᴹ, bᴹ` 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 `(Tmˢ (lam (var vz)) tt)` yields the `(λ aˢ → aˢ)` 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 metatheoretical functions for evaluation. However, the standard model does not allow one to go back to syntax from semantics. If we have an `(f : Aˢ → Bˢ)` semantic function, all we can do with it is to apply it to an argument and extract a `Bˢ`. *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 [**abel2013normalization**] 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 • ι → ⊥
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 [**coquand1997intuitionistic**], 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.

Table 4.1: Overview of models

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

4.3 Implementation

We denote the model for normalization with capital "N" superscript. 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 ι       Γ = Nf Γ ι
Tyᴺ (A ⇒ B) Γ = ∀ {Δ} → OPE Δ Γ → Tyᴺ A Δ → Tyᴺ B Δ
```

Each type is mapped to a `Con → 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 **??**, 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

```
fᴺ : ∀ {Δ} → OPE Δ Γ → Tyᴺ A Δ → Tyᴺ B Δ
```

then we also have

```
fᴺ (wk {A}) : Tyᴺ A (Γ , A) → Tyᴺ B (Γ , 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.

```
data Conᴺ : Con → Con → Set where
  •   : Conᴺ • Δ
  _,_ : Conᴺ Γ Δ → Tyᴺ A Δ → Conᴺ (Γ , A) Δ
```

```
∈ᴺ : ∀ {Γ A} → A ∈ Γ → ∀ {Δ} → Conᴺ Γ Δ → Tyᴺ A Δ
∈ᴺ vz     (Γᴺ , tᴺ) = tᴺ
∈ᴺ (vs v) (Γᴺ , _ ) = ∈ᴺ v Γᴺ
```

When interpreting terms, variables and applications are easy. In the latter case we just apply the recursive result to `idₑ` - 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:

```
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 (? , aᴺ)  -- ? marks the bump
Tmᴺ (app f a) Γᴺ = Tmᴺ f Γᴺ idₑ (Tmᴺ a Γᴺ)
```

Clearly, the idea is to evaluate the inner `t` term in `(Γᴺ , aᴺ)` as we did in the standard model, but `Γᴺ` does not have the right type. It has type `Conᴺ Γ Δ`, but we need to return in some `Σ` extended context, with `(σ : OPE Σ Δ)` 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:

```
_∘ₑ_ : OPE Δ Σ → OPE Γ Δ → OPE Γ Σ
σ      ∘ₑ •      = σ
σ      ∘ₑ drop δ = drop (σ ∘ₑ δ)
drop σ ∘ₑ keep δ = drop (σ ∘ₑ δ)
keep σ ∘ₑ keep δ = keep (σ ∘ₑ δ)
```

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

```
Tyᴺₑ : OPE Δ Γ → Tyᴺ A Γ → Tyᴺ A Δ
Tyᴺₑ {ι}     σ tᴺ = Nfₑ σ tᴺ
Tyᴺₑ {A ⇒ B} σ tᴺ = λ δ aᴺ → tᴺ (σ ∘ₑ δ) aᴺ

Conᴺₑ : OPE Σ Δ → Conᴺ Γ Δ → Conᴺ Γ Σ
Conᴺₑ σ •         = •
Conᴺₑ σ (Γᴺ , tᴺ) = Conᴺₑ σ Γᴺ , Tyᴺₑ σ tᴺ
```

For `Tyᴺₑ {ι}`, we use embedding of normal forms. For `Tyᴺₑ {A ⇒ B}`, we compose the externally given `σ` embedding with the input embedding `δ`. `Conᴺₑ` is just pointwise

application of `Tyᴺₑ`. `Tmᴺ` can be now given as:

```
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)
```

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.

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

  uᴺ : ∀ {A Γ} → Ne Γ A → Tyᴺ A Γ
  uᴺ {ι}     n = ne n
  uᴺ {A ⇒ B} n = λ σ aᴺ → uᴺ (app (Neₑ σ n) (qᴺ aᴺ))

uᶜᴺ : ∀ {Γ} → Conᴺ Γ Γ
uᶜᴺ {•}     = •
uᶜᴺ {Γ , A} = Conᴺₑ wk uᶜᴺ , uᴺ (var vz)

nf : ∀ {Γ A} → Tm Γ A → Nf Γ A
nf t = qᴺ (Tmᴺ t uᶜᴺ)
```

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 is a function, its semantic function 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 `qᴺ` does not directly call `Tmᴺ`, it does apply semantic functions, which may *internally* call `Tmᴺ`.

In fact, this encapsulation of recursive `Tmᴺ` calls is a crucial detail which makes this whole definition structurally recursive and thus total. In the

`(Tmᴺ (lam t) Γᴺ = λ σ aᴺ → Tmᴺ t (Conᴺₑ σ Γᴺ , aᴺ))` clause, the recursive `Tmᴺ` 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 `(tᴺ wk (uᴺ (var vz)))` application, the `(uᴺ (var vz))` term plays the role of blocking input, but it is not quite entirely blocking. `uᴺ` performs an operation best described as *semantic -expansion*: it acts as identity on neutral base terms, and from neutral function it produces semantic function which build up neutral applications from their inputs. Thus, `(uᴺ {ι} (var vz))` simply reduces to `(ne (var vz))`, while `(uᴺ {ι ⇒ i} (var vz))` reduces to `(λ σ aᴺ → ne (app (var (∈ₑ σ vz)) aᴺ))`. In short, `uᴺ` returns semantic values which yield properly -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 Kirpke models relate to the above definition of normalization. Following the presentation of Altenkirch [**altenkirch2009normalisation**], 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
    ... -- 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 : ∀ {A w w'} → 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 : ∀ {Γ}{w w'} → 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 : ∀ {Γ A} → Tm Γ A → Γ ||- A
  ||-Tm (var vz)     w (_ , q) = q
  ||-Tm (var (vs v)) w (p , q) = ||-Tm (var v) w p
  ||-Tm (lam t)      w p σ a   = ||-Tm t _ (Con-mono σ p , a)
  ||-Tm (app f a)    w p       = ||-Tm f w p ≤refl (||-Tm a w p)
```

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

```
N : KripkeModel
N = record {
    W      = Con
  ; _≤_    = λ Γ Δ → OPE Δ Γ
```

```
  ; ≤refl  = idₑ
  ; _≤■_   = _°ₑ_
  ; _||-ι  = λ Γ → Nf Γ ι
  ; ι-mono = Nfₑ }
```

This has been 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 propopsition `ι`. What is a right notion of semantics, though? We could abstract out the interpetation of base types from the standard model as given in Chapter **??**, 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 follows:

```
sound    = ∀ {Γ A} → Tm Γ A → (∀ M → let open KripkeModel M in Γ ||- A)
complete = ∀ {Γ A} → (∀ M → let open KripkeModel M in Γ ||- A) → Tm Γ A
```

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 `||-Tm`. For completeness, we instantiate the hypothesis with the previously defined `N` model, then define quoting and unquoting the same way as in Chapter **??**. Using them, we can produce a `(Nf Γ A)`, which can be canonically injected back to `(Tm Γ A)`. □

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 `_≤_` 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 `ᴺ` 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; 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 [**hottbook**].

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 *syntacic 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 $F : C \quad D$ 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

```
morph⇒ : morph I J → morph (Obj⇒ J) (Obj⇒ I))
```

The composition law changes accordingly. Contravariant functors are often denoted as $F : C^{op} \quad D$ (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 $FG : C \quad D$ functors. They consist of a family of morphisms for each I object of C, and a *naturality* condition, which expresses that 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
```

$F : C \quad D$ 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 us 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 **??** and Chapter **??**:

```
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**. `(λ Γ → A ∈ Γ)` maps each object of **OPE** to an object of **Set**, that is, an Agda type. Likewise does `(λ Γ → Tm Γ A)`. They also have action on morphisms: these are precisely the embedding operations on variables and terms, defined in Chapter **??**.

```
∈ₑ  : 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
∈-idₑ  : ∀ v → ∈ₑ idₑ v ≡ v
∈-∘ₑ   : ∀ σ δ v → ∈ₑ (σ ∘ₑ δ) v ≡ ∈ₑ δ (∈ₑ σ v)
```

```
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 `Є-idₑ` would be the following:

```
Є-idₑ : Єₑ idₑ ≡ (λ v → v)
```

This expresses that `idₑ` 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 -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* of **STLC**: it contains all the objects of **STLC** but only some of its morphisms.

We review identity substitution here (previously defined in Chapter **??**). 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:

```
∈ₛ  : 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 ∈ _)` 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 [**benton2012strongly**], substitution laws are constructed the same way as here, but the authors of Ibid. also do not provide a semantic explanation. For contrast, see [**altenkirch2016normalisation**], 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 `_∘ₑₛ_` operation here:

```
_∘ₑ_  : OPE Δ Σ → OPE Γ Δ → OPE Γ Σ -- defined in Chapter 4.3
_∘ₛ_  : Sub Δ Σ → Sub Γ Δ → Sub Γ Σ -- defined in this chapter
_ₛ∘ₑ_ : Sub Δ Σ → OPE Γ Δ → Sub Γ Σ -- defined in Chapter 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 ν → (σ s∘e δ) s∘e ν ≡ σ s∘e (δ ∘e ν)
assese : ∀ σ δ ν → (σ e∘s δ) s∘e ν ≡ σ e∘s (δ s∘e ν)

idles : ∀ σ → ide e∘s σ ≡ σ
idlse : ∀ σ → ids s∘e σ ≡ ⌜ σ ⌝ope
idres : ∀ σ → σ e∘s ids ≡ ⌜ σ ⌝ope

∈-e∘s  : ∀ σ δ v → ∈s (σ e∘s δ) v ≡ ∈s δ (∈e σ v)
Tm-e∘s : ∀ σ δ t → Tms (σ e∘s δ) t ≡ Tms δ (Tme σ t)

∈-s∘e  : ∀ σ δ v → ∈s (σ s∘e δ) v ≡ Tme δ (∈s σ v)
Tm-s∘e : ∀ σ δ t → Tms (σ s∘e δ) t ≡ Tme δ (Tms σ t)

assses : ∀ σ δ ν → (σ s∘e δ) ∘s ν ≡ σ ∘s (δ e∘s ν)
assse : ∀ σ δ ν → (σ ∘s δ) s∘e ν ≡ σ ∘s (δ s∘e ν)

-- Functor laws for (A ∈ _) and (Tm _ A)
∈-∘s   : ∀ σ δ v → ∈s (σ ∘s δ) v ≡ Tms δ (∈s σ v)
Tm-∘s  : ∀ σ δ t → Tms (σ ∘s δ) t ≡ Tms δ (Tms σ t)
∈-ids  : ∀ v → ∈s 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 ν → (σ ∘s δ) ∘s ν ≡ σ ∘s (δ ∘s ν)
```

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 **??**. In the return type, the injection `⌜_⌝Nf` converts `(Nf Γ A)` back to `(Tm Γ A)`.

Since `nf` is defined with a Kripke model, it is clear that proving properties about it have to involve similar model structures. The key point is again the interpretation of types. The overall structure of the proof is very similar to evaluation itself; we will have interpretation of types, contexts, terms, and quoting and unquoting. The main difference is that types are interpreted as *relations* this time.

```
_≈_ : ∀ {A Γ} → Tm Γ A → Tyᴺ A Γ → Set
_≈_ {ι}       t tᴺ = t ~ ⌜ qᴺ tᴺ ⌝Nf
_≈_ {A ⇒ B}{Γ} t tᴺ =
    ∀ {Δ}(σ : OPE Δ Γ){a aᴺ} → a ≈ aᴺ → app (Tmₑ σ t) a ≈ tᴺ σ aᴺ
```

`_≈_` is a *Kripke logical relation*. Here, "logical relation" simply means that it is a relation defined by large recursion on types. "Kripke" indicates that the interpretation of function types is generalized to work in any extended context. 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.

5.4 Presheaf Model

TODO

5.5 Soundness

TODO

5.6 Stability

TODO

Chapter 6

Variations

6.1 Direct Presheaf Model

- Definition, proofs
- Semantic equality == propositional equality
- However: funext in definition of nf. Use external canonicity proof or metatheory with funext

6.2 More Efficient Evaluation

- efficient embedding for semantic contexts
- Agda benchmarking
- performance limitations of intrinsic syntax compared to type assignment

Chapter 7

Discussion and Future Work

- Technical overhead: Vs big-step, hereditary subst, Coquand, Abel (?)
- Linecounts vs big-step and hsubst (email C. Coquand to get sources?)
- Usability as EDSL normalizer foo
- Scaling up the technique
- With implicit substitution and conversion relation: to System F, probably