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 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.

[1] In 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.[2]

If we are to trust a particular normalization algorithm, we need to prove the following properties (CIT: bigstep or alti):

- *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.

TODO: Summary of chapters, some blahblah (at least half page worth of it)

1.2 Related Work

TODO: Not directly related background: Martin-Löf, Berger & Schwichtenberg. (one paragraph)

Abel's CIT work 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 on extrinsic syntax and separate typed and untyped phases of evaluation.

Catarina Coquand's work CIT is the most closely related to our development. She

[2]In contrast to approaches based on small-step reduction, where η requires complicated setup.

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 substitution.

Altenkirch and Chapman CIT 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 CIT.

Altenkirch and Kaposi CIT 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 sec. 5.1, 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 somewhat less transparent 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 CIT.

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 [16]. 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* CIT. 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.* We could eschew it by using setoid reasoning with semantic equivalence, as C. Coquand does CIT. 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 CIT and observational CIT 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 CIT. We use the `--without-K` language option for all of our code in the main development. However, we do use axiom K in sec. 5.1 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 [12]. The latest documentation is available online [8]. For a book-length introduction geared towards beginners, see [15]. We nontheless summarize here the core constructions.

2.2.1 Basic Constructions

NOTE: EITHER CULL THIS PART OR EXTEND IT. Ideally it should be culled, because a few page worth of Agda intr isn't realistically useful.

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)
```

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

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

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 CIT. 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 [5] 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 Standard Library

Our development does not have any external dependencies. We need less than a hundred lines of code from the Agda standard library CIT, but we choose to include it along 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 [16]:

```
_■_ : 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 [11] in Haskell programming.

Furthermore, we use coercion instead of transport (see HoTT book [16, pp. 72], also named `subst` in the Agda standard library CIT):

```
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`. `⊤` has `tt` ("trivially true") as sole element. Injection into disjoint sums is done with `inj₁` and `inj₂`. The *ex falso* principle is named `⊥-elim`, and we have `⊥-elim : {A : Set} → ⊥ → 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:

```
data Ty : Set where
  ι   : Ty
  _⇒_ : Ty → Ty → Ty

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

[1] We take the view that the "proper" definition of syntax would be an initial object in some suitable category of models, e. g. the initial category with families. Thus, models include conversion as propositional equations, so the syntax is quotiented by conversion as well; see e. g. [2]. 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 _∈_ (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. [4] for approaches to dealing with them. For extrinsic syntax, powerful automation for reasoning about substitution is available in Coq, allowing spectacularly compact proofs [14]. 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 [2], in contrast to extrinsic approaches which suffer from a veritable explosion of well-formedness conditions and congruence rules.

3.2 Context Embeddings and Substitutions

Another 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 substitution for two reasons:

1. It is the canonical presentation of STLC in textbooks [13, 9].

2. Lighter formalization. With implicit substitution, 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[1] 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 sec. 4.6.

At this point, we are concerned with conversion for terms, and substitution is introduced in order to define β-conversion. For this we need single substitution. However, simultaneous substitution is easier to define and reason about. Thus, we shall define

the former using the latter and identity substitutions.

Identity substitutions are those which have the identity action on terms. Conceptually, they assign to each free variable in a term the same variable. They can be implemented in a number of ways. One possibility is to make identity substitutions explicit constructions, in which case their action on terms can be simply defined as identity. However, we instead opt for a *normalized* representation of substitutions, as lists of terms:

```
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. Following McBride [10, pp. 24] we could give an alternative `Sub` definition, which makes the underlying assignment obvious:

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

However, this representation contains many definitionally distinct functions with the same action on terms. Thus, it is larger than what is strictly necessary, which may or may not result in technical difficulties. We aim to choose the smallest representation that does the job, as a general principle.

Before we can define identity substitutions and the action of substitution on terms, we need to expresses the notion that whenever one has a term in a context, one can construct essentially the same term in a larger context. Specifically here, we need an operation with type `(∀ {A Γ Δ} → Sub Γ Δ → Sub (Γ , A) (Δ , A))`, in order to be able to push 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, then extend the result with `var vz`.

We define *order-preserving embeddings* the following way:

```
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 [6] 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. They can fulfill the same role as order-preserving embeddings. However, they are larger than necessary, since they can represent permutations contexts as well, which we have no need for.

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 {•}     = •
idₑ {Γ , A} = keep (idₑ {Γ})
```

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

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

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

Embedding of terms can be pointwise extended to substitutions:

```
_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 it is a composition of a substitution and an embedding. Its type resembles that of ordinary function composition, if we consider `Sub` and `OPE` as context morphisms. We will expand on the categorical interpretation of this in sec. 4.3.

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
```

`keepₛ` is precisely the substitution operation we were looking for. 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 the `t` term to the zeroth de Bruijn variable and leaves all other variables unchanged.

3.3 Conversion

The conversion relation is given as:

```
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`, `_~■_`).

[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.

Chapter 4

Normalization

In this chapter we implement normalization and prove its correctness.

4.1 Preliminaries: Standard and Enriched Models

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

```
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 use 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. Agda does not yet have convenient overloading support, 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.

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`. The idea of NbE is to add extra structure to semantic types 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 about the syntax. The standard model yields a simple proof of soundness, namely that there is no term with base type in the empty context:

```
sound : Tm • ι → ⊥
```

```
sound 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 [7]. Adding yet more structure yields *presheaf models*, which enable correctness proofs for normalization as well. In this chapter, we will present Kripke and presheaf models in this order.

4.2 Normalization with a Kripke Model

TODO

4.3 Laws for Embedding and Substitution

TODO

4.4 Completeness

TODO

4.5 Presheaf Model

TODO

4.6 Soundness

TODO

4.7 Stability

TODO

Chapter 5

Variations

5.1 Direct Presheaf Model

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

5.2 More Efficient Evaluation

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

Chapter 6

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
- Scaling up the technique
- With implicit substitution and conversion relation: to System F, probably

Bibliography

[1] Thorsten Altenkirch and James Chapman. "Big-step normalisation". In: *Journal of Functional Programming* 19.3-4 (2009), pp. 311–333.

[2] 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.

[3] Lennart Augustsson and Magnus Carlsson. *An exercise in dependent types: A well-typed interpreter*. 1999.

[4] Nick Benton et al. "Strongly typed term representations in Coq". In: *Journal of automated reasoning* 49.2 (2012), pp. 141–159.

[5] Edwin Brady. "Idris, a general-purpose dependently typed programming language: Design and implementation". In: *Journal of Functional Programming* 23.05 (2013), pp. 552–593.

[6] 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.

[7] Thierry Coquand and Peter Dybjer. "Intuitionistic model constructions and normalization proofs". In: *Mathematical Structures in Computer Science* 7.1 (1997), pp. 75–94.

[8] Agda developers. *Agda documentation*. 2017. URL: https://agda.readthedocs.io/en/v2.5.2/ (visited on 03/10/2017).

[9] Robert Harper. *Practical foundations for programming languages*. Cambridge University Press, 2016.

[10] Conor McBride. "Datatypes of Datatypes". In: (2015).

[11] Conor McBride and Ross Paterson. "Applicative programming with effects". In: *Journal of functional programming* 18.01 (2008), pp. 1–13.

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

[13] Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.

[14] 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.

[15] 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.

[16] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study: https://homotopytypetheory.org/book, 2013.