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

Author: András Kovács Advisor: Ambrus Kaposi

Budapest, 2017

Abstract

We implement and prove correct normalization by evaluation for simply typed lambda calculus, using the proof assistant Agda. The correctness proof consists of soundness, completeness and stability with respect to $\beta\eta$-conversion. The algorithm uses a Kripke model over the preordered set of order-preserving context embeddings. Completeness is given by a logical relation between the term model and the Kripke model. For soundness and stability, we first need to prove that evaluation, quoting and unquoting all commute with order-preserving embeddings, which amounts to showing that the codomain of the evaluation function can be refined to a presheaf model. Then, soundness is shown by a logical relation on semantic values which fall into the refined model. Stability is proved by induction on normal and neutral terms. Decidability of $\beta\eta$-conversion for terms also follows from correctness. The main advantage of the formalization is that normalization has a concise and structurally recursive specification and is presented separately from the correctness proofs. Also, the syntax of terms remains simple, making no use of explicit substitutions or closures. Overall, the technical overhead of the development is relatively light.

Contents

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.

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]

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

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

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 summary move here?)

Catarina Coquand's work CIT 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 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 confers 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.

For an alternative approach to NbE, see Abel CIT for an overview of NbE for a range of type theories, using extrinsic syntax and separate typed and untyped phases of evaluation.

2 Metatheory and Notation

In this chapter the metatheory used for formalization is presented, first broadly, then its specific implementation and notation 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 [5]. However, there is no canonical definition of intensional type theory. There are numerous variations concerning type universes and 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, called* `Set₀` *and* `Set₁`*, with large elimination into* `Set₀`. Large elimination is needed for recursive definitions of semantic types, since inductive definitions would not positive and hence would be illegal. We only need `Set₁` to type the large eliminations.
- *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 the main development, we do not use Streicher's K axiom. We do use it in sec. 5.1

for the direct presheaf model, but the usage there is only for technical convenience, as it could be avoided by additional proofs of propositionality of equalities.

2.2 Agda

NOTE: cull this part if it gets superfluous. It's not a good idea to write an Agda tutorial in 4 pages.

Agda is a dependently typed programming language and proof assistant. Its current design and core features are described in Ulf Norell's thesis [3]. The latest documentation is available online [2]. For a book-length introduction geared towards beginners, see [4]. We nonetheless summarize the core constructions, and also our brevity-motivated deviations from legal Agda.

2.2.1 Basic Constructions

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

Inductive definitions are the facility used to introduce new types. Agda has *open type universes*, which means that programmers may freely introduce new types as long as the definitions preserve logical consistency[3]. 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

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

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 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`.

As a notational convention, we shall take cues from the Idris [1] programming language and sometimes leave implicit parameters implicit even in type declarations of functions or con. 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 status of parameters are obvious.

3 Syntax

3.1 Base Syntax

3.2 Context Embedding

TODO

3.3 Substitution

TODO

3.4 Conversion

TODO

3.5 A Note on Categorical Semantics

TODO

4 Normalization

TODO

4.1 Preliminaries: Standard Model

TODO

4.2 Normalization with a Kripke Model

TODO

4.3 Completeness

TODO

4.4 Naturality Proofs

TODO

4.5 Soundness

TODO

4.6 Stability

TODO

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

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 foo

References

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

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

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

[4] 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.

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