A Coq Formalization of Normalization by Evaluation for Martin-Löf Type Theory

Paweł Wieczorek
University of Wrocław, Poland
pawel.wieczorek@cs.uni.wroc.pl

Dariusz Biernacki
University of Wrocław, Poland
dabi@cs.uni.wroc.pl

Abstract

We present a Coq formalization of the normalization-by-evaluation algorithm for Martin-Löf dependent type theory with one universe and judgmental equality. The end results of the formalization are certified implementations of a reduction-free normalizer and of a decision procedure for term equality.

The formalization takes advantage of a graph-based variant of the Bove-Capretta method to encode mutually recursive evaluation functions with nested recursive calls. The proof of completeness, which uses the PER-model of dependent types, is formalized by relying on impredicativity of the Coq system rather than on the commonly used induction-recursion scheme which is not available in Coq. The proof of soundness is formalized by encoding logical relations as partial functions.

CCS Concepts • **Theory of computation → Proof theory**; **Type theory**;

Keywords normalization by evaluation, type theory, Coq, program certification

ACM Reference Format:
Paweł Wieczorek and Dariusz Biernacki. 2018. A Coq Formalization of Normalization by Evaluation for Martin-Löf Type Theory. In *Proceedings of 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP'18).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3167091

CPP'18, January 8–9, 2018, Los Angeles, CA, USA

ACM ISBN 978-1-4503-5586-5/18/01…$15.00
https://doi.org/10.1145/3167091

1 Introduction

Proof assistants such as Coq or Agda rely on constructive type theories, where logic is obtained through the Curry-Howard correspondence. In this paradigm propositions are identified with types, proofs are identified with terms, and proof validation is obtained by *type checking*. Thus, crucially, the type-checking in such systems has to be decidable. No less important is the *normalization* property, which states that each typable expression can be reduced to a normal form, since it plays a crucial role in a proof of the consistency of the logic. These two properties are intimately connected in the presence of *dependent types* i.e., types that depend on terms and that may themselves depend on the terms of the logic, which can be used to encode predicates. Since terms are present at the type level the type-checking algorithm must be equipped with a procedure for deciding equality over terms of the system. Furthermore, the presence of the *universe* – the type of types – requires one also to normalize types as part of the type-checking algorithm. Proving correctness of such rich type theories is hard and any seemingly innocent extensions can introduce issues that make it even harder. For instance, we may wish to equip the underlying type theory of a proof assistant with η-rules, since they allow us to equate terms like $P(f)$ and $P(\lambda x : A.\ f\,x)$, which turns out to be convenient in practice. However, such an extension may break the Church-Rosser property [Lazarom 1973] that is often used in a proof of soundness of the system. A possible approach to handle η-rules would be to extend the system with *judgmental equality*, but this could lead to other issues [Adams 2006].

Normalization by evaluation Normalization by evaluation (NbE) is a procedure for computing normal forms in lambda calculi [Abel 2010; Abel et al. 2007a,b, 2009; Aehlig and Joachimski 2004; Berger and Schwichtenberg 1991; Coquand and Dybjer 1997; Filinski and Rohde 2004]. It does not depend on a reduction relation, but instead it interprets terms in a sophisticated domain and then reads back a normal form of a given term. Therefore, the technique does not depend on the Church-Rosser property of the reduction relation. Furthermore, it can be readily used to construct a *decision procedure* for equality of terms, necessarily in a type-checking algorithm for a dependent type theory. Correctness

of the NbE procedure for dependent type theory is usually established using a model where types and their equality are interpreted as *partial equivalence relations* (PERs). These PER-models and related theories can be used to prove the consistency of the encoded logic as well as the correctness of the type-checking algorithm.

This work In this article we present the first Coq formalization of a normalization by evaluation procedure for Martin-Löf dependent type theory with one universe and judgmental equality [Martin-Löf 1998; Nordström et al. 1990]. We have formalized a formulation of the NbE developed in [Abel 2010; Abel et al. 2007a,b, 2009].

One of the biggest challenges in our work arises from the presence of partial functions in the formalized NbE procedure. Such functions cannot be directly programmed in Coq since its type system only allows us to formulate total functions. Moreover we cannot use popular techniques of encoding partial functions, since they cannot handle nested recursive calls. The usual extensions of such methods require the induction-recursion scheme [Dybjer 2000], where a function can be defined simultaneously with an inductive predicate – a technique not available in Coq. To overcome this difficulty, we use a variant of the Bove-Capretta method [Bove 2009; Bove and Capretta 2001] of encoding partial functions in constructive type theory, where an auxiliary graph-based definition of the encoded function is used.

The lack of the induction-recursion scheme in Coq leads to yet another significant challenge in our formalization. The scheme plays a key role in the construction of the PER-model, required for the usual proof of completeness of NbE. Our approach to bypass this limitation is to take advantage of the impredicativity of the `Prop` sort.

Yet another challenge worth mentioning stems from the fact that the logical relations, used to prove soundness of NbE, are indexed by semantic representation of types. Thus, our encoding of logical relations is based on partial functions defined over an inductive predicate.

An attractive and practical aspect of constructive type theories is the possibility of automatic program extraction from proofs [Letouzey 2008]. Our formalization is driven by the goal of extracting a certified implementation of the NbE procedure and many decisions have been made to make the extraction feasible.

Contributions To summarize our main contributions:

- We provide a first Coq formalization of the NbE algorithm for dependent type theory.
- We provide certified implementations of the procedures for normalizing expressions and deciding equality in type theory.
- We show how to build the PER-model for dependent type theory without involving the induction-recursion scheme.
- We show an advanced example of defining partial functions with nested recursion using a graph-based variant of the Bove-Capretta method.

Structure of the paper The structure of this paper mimics the structure of the underlying articles where NbE for dependent types has been developed [Abel 2010; Abel et al. 2007a,b, 2009], but we focus only on selected parts of our formalization. We first give a brief introduction to a formulation of the type theory under consideration (Section 2). We then explain our encoding of the normalization-by-evaluation procedure in Coq (Section 3). Next, we present a formalized PER-model used to prove completeness of NbE (Section 4) and a formalization of logical relations used to prove soundness of NbE (Section 5). Then we briefly comment on extracting certified code from the formalization (Section 6). Finally, we conclude with discussion of the closest related work and possible directions for future work (Section 7).

Coq project The complete formalization is available at

https://bitbucket.org/pl-uwr/nbe-martinlof

2 Type Theory

In this section we briefly introduce the type theory that we formalize, i.e. Martin-Löf dependent type theory with one universe and judgmental equality [Martin-Löf 1998; Nordström et al. 1990], and with explicit substitutions [Abadi et al. 1991]. The calculus we consider is additionally equipped with the unit and empty types. Furthermore, the equality contains η-rules for functions and the unit type. The grammar of the calculus is given below:

$$\begin{array}{rcl} Tm \ni t, s, A, B & ::= & v_0 \mid t\ s \mid \lambda A.\ t \mid \Pi A.\ B \mid t\ \sigma \mid \\ & & \mathtt{Unit} \mid \mathtt{()} \mid \bot \mid \epsilon_A\ t \\ Sb \ni \sigma, \delta & ::= & \mathtt{id} \mid \uparrow \mid \sigma \cdot \delta \mid (\sigma, t) \end{array}$$

Tm represents the set of terms and Sb represents the set of explicit substitutions. A term $t\ \sigma$ represents application of a term t to a substitution σ. Variables are represented by de Bruijn indices where v_0 represents the variable with the lowest index and the remaining variables are represented using the substitution $\uparrow$, i.e., $v_{n+1} = v_n\uparrow$. $\Pi A.B$ represents a dependent function space with a domain A and a dependent co-domain B where the variable v_0 is bound. The constructs $t\ s$ and $\lambda A.\ t$ denote term application and lambda abstraction with the variable v_0 bound in the function body, respectively. `Unit` denotes the unit type where `()` is its only value. The empty type is represented by $\bot$ where ϵ_A represents its eliminator. `id` is the identity substitution and $\sigma \cdot \gamma$ is a composition of substitutions. (σ, t) represents an extension of the substitution σ with the term t.

The type system, fragments of which are shown in Figure 1, contains several kinds of judgments. Most of them are

$$\textsc{EMPTY}\ \frac{}{\diamond \vdash} \qquad \textsc{EXT}\ \frac{\Gamma \vdash \quad \Gamma \vdash A}{\Gamma, A \vdash} \qquad \textsc{UNIV\_UNIT\_F}\ \frac{\Gamma \vdash}{\Gamma \vdash \mathtt{Unit} : \mathtt{Univ}} \qquad \textsc{UNIV\_E}\ \frac{\Gamma \vdash A : \mathtt{Univ}}{\Gamma \vdash A} \qquad \textsc{FUN\_F}\ \frac{\Gamma \vdash A \quad \Gamma, A \vdash B}{\Gamma \vdash \Pi A.\ B}$$

$$\textsc{HYP}\ \frac{\Gamma, A \vdash}{\Gamma, A \vdash v_0 : A\uparrow} \qquad \textsc{FUN\_E}\ \frac{\Gamma \vdash t : \Pi A.\ B \quad \Gamma \vdash s : A}{\Gamma \vdash t\ s : B\ (\mathtt{id}, s)} \qquad \textsc{SB}\ \frac{\Gamma \vdash \sigma : \Delta \quad \Delta \vdash t : A}{\Gamma \vdash t\ \sigma : A\ \sigma} \qquad \textsc{SID}\ \frac{\Gamma \vdash}{\Gamma \vdash \mathtt{id} : \Gamma}$$

$$\textsc{SUP}\ \frac{\Gamma \vdash A}{\Gamma, A \vdash \uparrow : \Gamma} \qquad \textsc{CONV}\ \frac{\Gamma \vdash t : B \quad \Gamma \vdash A = B}{\Gamma \vdash t : A} \qquad \textsc{SB\_CONV}\ \frac{\Gamma \vdash \sigma : \Delta \quad \vdash \Delta = \Psi}{\Gamma \vdash \sigma : \Psi} \qquad \textsc{EQ\_CXT\_REFL}\ \frac{\Gamma \vdash}{\vdash \Gamma = \Gamma}$$

$$\textsc{EQ\_FUN\_Eta}\ \frac{\Gamma \vdash t : \Pi A.\ B}{\Gamma \vdash t = \lambda A.(t\uparrow) v_0 : \Pi A.\ B} \qquad \textsc{EQ\_CXT\_EXT}\ \frac{\vdash \Gamma = \Delta \quad \Gamma \vdash A \quad \Delta \vdash B \quad \Gamma \vdash A = B}{\vdash \Gamma, A = \Delta, B}$$

Figure 1. Fragments of the type system

of the form $\Gamma \vdash J$ where Γ is typing context and J is a particular judgment. The typing context is an ordered sequence of types:

$$\Gamma ::= \diamond \mid \Gamma, A$$

$\diamond$ denotes empty context and Γ, A denotes a context where A is the type of the variable v_0 and the type A may refer to variables valid in the context Γ.

A judgment $\boxed{\Gamma \vdash}$ expresses that the typing context Γ is well formed, i.e., all types are well formed and the dependencies between them are valid. A judgment $\boxed{\Gamma \vdash A}$ expresses that a type is well formed in a given context; a judgment $\boxed{\Gamma \vdash t : A}$ states that the term has a given type in a given context, and a judgment $\boxed{\Gamma \vdash \sigma : \Delta}$ states that a given substitution yields terms/types valid in the context Γ when applied to terms or types valid in the context Δ. Judgments $\boxed{\Gamma \vdash A = B}$, $\boxed{\Gamma \vdash s = t : A}$, $\boxed{\Gamma \vdash \sigma = \delta : \Delta}$ represent semantic equality in the system. Judgments of the form $\boxed{\vdash \Gamma = \Delta}$ state that two given typing contexts have equal size and that the corresponding types they mention are also equal.

Judgments are formalized as a set of inductive predicates defined mutually. Most of the properties such as inversion lemmas were proved easily using the induction scheme generated by the system or mutual induction obtained using the `Combined Scheme` command. We encountered difficulties only with the *context conversion* and *syntactic validity* theorems.

The first one allows to transfer judgments from one typing context to another if only we have a proof that they are equal w.r.t. the $\boxed{\vdash \Gamma = \Delta}$ statement. The context equality could be formulated outside the typing system, but the presence of explicit substitutions makes the theorem hard to prove. The problem arises from the typing rules like `SID` or `SUP` where the typing context and the type of the substitution are strongly tied. We preferred to embed such an equality into the system and formulate the `SB_CONV` rule which allows to treat substitutions in a way analogous to other expressions. A similar observation was made in [Pagano 2012, Section 5.1]. Additionally, the proof requires a well-founded induction scheme over the derivation height which was impossible to obtain because the system was formulated in `Prop` and computing data, like height, on proofs is forbidden in Coq. Since propositions are erased during program extraction we preferred to stick to such a formulation rather than to encode derivations as data. To overcome the problem we decided to duplicate the definition of our system, and have a version of it augmented with annotations allowing to formulate the required induction scheme. Since such formulations are equivalent we jump to the augmented system whenever it is needed.

The syntactic validity theorem ensures that sub-parts of a given statement are also well formed. For example, for a judgment $\boxed{\Gamma \vdash t : T}$ it states that the well-formedness of the typing context or type are also provable. The two theorems (context conversion and syntactic validity) depend on each other and we were unable to prove them in a straightforward way. We first proved a simpler version of the context conversion theorem which allows to transfer judgments between typing context that differ only on the last position. Then we were able to prove the syntactic validity theorem and finally the full context conversion theorem.

3 Normalization by Evaluation

The NbE procedure can be seen as a composition of two phases. The first one is an evaluation of a term with a proper environment in a domain which is capable of representing open terms. In our work, we follow the definitions from [Abel 2010], where the domain is an applicative structure with functions represented by first-order closures. The second phase consists in extracting a term in $\beta\eta$-long normal form from the semantic value representing the evaluated expression. In our formalization we follow a technique developed

in [Abel et al. 2009], where the reading-back phase has been split into two steps. First, we perform η-expansion inside the domain. Then, we read-back the normal form, generating fresh variables by converting de Bruijn levels that represent variables in the domain, into de Bruijn indices that represent variables in the calculus.

For lack of space, in our presentation we focus mainly on the lambda calculus part of the system. For a more detailed and comprehensive explanation of the technique we refer the reader to [Abel et al. 2007a,b, 2009].

3.1 The Domain

Computations are carried out in an untyped domain represented by an applicative structure that is capable of representing neutral terms by *neutral-term-like values*. The presence of dependent types and of the universe leads to computations on types, thus the domain is equipped also with *type-like values*. There are two auxiliary functions, named *reflection* and *reification*, responsible for embedding the representation of neutral terms into the domain and performing η-expansion, respectively. Both functions require type information and are driven by a type-like value. The domain includes a special kind of values representing closures of the reflection and reification functions.

We present the encoded domain `D` in Figure 2. `Dclo t` η is a function closure, where `t` is a term representing a function body and η is an enclosing environment. The `Dunit`, `Dempty`, and `Duniv` are constants representing basic types. `Dfun a f` is a value representing a dependent function space, where `a` is a value representing a type of the domain and `f` is a value representing a dependent co-domain.

`DNe` represents neutral-like values where variables are encoded with de Bruijn levels. `DNf` represents closures of the reification function. `DdownX dT da` represents an invocation of the reification function with a type-like value `dT` and a value `da`. The `DdownN` (written mathematically $\Downarrow$) is an invocation of the reification function where type information is not needed, thus not represented in the closure. Similarly, `DdownN` (written mathematically $\Uparrow$) and `DdownX` represent closures, equipped with type information or not, of the reflection function. Both defunctionalized functions are evaluated by the read-back functions, as explained in the following sections.

The `DEnv` type represents environments which are encoded as a sequence of values. We use an auxiliary function `Ddown` (written mathematically $\downarrow$) which takes a type-like value along with another value, and decides if the type information has to be preserved, returning `DdownX` or `DdownN`. We also use a similar function `Dup` (written $\uparrow$).

3.2 Encoding Partial Functions

It is known that in systems based on type theory, including Coq, one cannot directly encode partial functions, and therefore we need a method of encoding partial functions as total

```
Inductive D : Set :=
| Dclo     : ∀ (t:Tm) (η :DEnv), D
| DupX     : ∀ (dT:D) (dnf:DNe), D
| DupN     : ∀ (dne:DNe), D
| Dfun     : ∀ (DA:D) (DF:D), D
| Dunit    : D
| D1       : D
| Dempty   : D
| Dabsurd  : D → D
| Duniv    : D
with DNe : Set :=
| Dvar     : ∀ (n:nat), DNe
| Dapp     : ∀ (dne:DNe) (dnf:DNf), DNe
| DneAbsurd : ∀ (da:DNf) (dne:DNe), DNe
with DNf : Set :=
| DdownX   : ∀ (dT:D) (da:D), DNf
| DdownN   : ∀ (da:D), DNf
with DEnv : Set :=
| Did      : DEnv
| Dext     : ∀ (η :DEnv) (dx:D), DEnv.
```

Figure 2. Domain

ones. A common approach is to truncate the domain of a function by introducing an additional argument – a predicate that holds only when the rest of the arguments are in the domain of the function. Additionally, we need to encode recursion properly so that Coq can verify that it is terminating. To this end, we use the Bove-Capretta method [Bove and Capretta 2001], where each function has a dedicated inductive accessibility predicate whose definition reflects the structure of recursive calls of the function. Computations (the `Set` universe) and logic (the `Prop` universe) are strictly separated in the Coq system and it is impossible to perform such recursion over proof term directly. This issue is resolved by using additional inversion functions. The method itself and its adaptation to Coq system is well described in [Bertot and Castéran 2004, Section 15.4].

Unfortunately, the method cannot be easily used in the Coq system when one deals with nested recursive calls. A general approach in such situations is to use induction-recursion scheme [Dybjer 2000], where the accessibility predicate and the function are defined simultaneously. The need of such a scheme stems from the fact that the predicate refers to the results of nested recursive calls. In Coq the induction-recursion scheme is not present, but to the rescue comes a method presented in [Bove 2009]. In this approach, one first defines the graph of a function and then the accessibility predicate is derived from it. It is this method that we use to define partial recursive functions. The difference between our encoding and one presented in [Bove 2009] is that we cannot derive the domain predicate *Dom* by putting $Dom(x) \iff \exists y.Graph(x, y)$. Aiming for effective program extraction, we need to define it from scratch. To the best of

$$\frac{}{[\![v_0]\!]_{\eta,d} \searrow d} \qquad \frac{}{[\![\lambda A.t]\!]_{\eta} \searrow Dclo(t,\eta)}$$

$$\frac{[\![\sigma]\!]_{\eta} \searrow \eta' \quad [\![t]\!]_{\eta'} \searrow d}{[\![t\,\sigma]\!]_{\eta} \searrow d}$$

$$\frac{[\![t]\!]_{\eta} \searrow d_t \quad [\![s]\!]_{\eta} \searrow d_s \quad d_t \cdot d_s \searrow d_y}{[\![t\,s]\!]_{\eta} \searrow d_y}$$

$$\frac{[\![t]\!]_{\eta,d_x} \searrow d_y}{Dclo(t,\eta) \cdot d_x \searrow d_y}$$

$$[\![t]\!]_{\eta} = d \iff [\![t]\!]_{\eta} \searrow d$$
$$d_f \cdot d_a = d_y \iff d_f \cdot d_a \searrow d_y$$

Figure 3. Graphs of evaluation functions

our knowledge our formalization provides the most complex application of the graph-based variant of Bove-Capretta method in Coq. Therefore we decided to provide complete explanation of such encoding in Section 3.3.

Encoding partial functions in Coq poses further difficulties. One needs to maintain accessibility predicates through all theorems and proofs while being able to evaluate the defined functions when needed. Hence the proofs throughout the development have to be transparent (where `Qed` is replaced by the `Defined` Vernacular command). Additionally, one must carefully design one's functions and theorems to avoid a dead-end due to mixing logic with computations which is forbidden in the Coq system. For these reasons we decided to stick to the graph representation of functions when proving NbE correct, and turn to the actual definitions of partial functions only for program extraction. We find this approach rather advantageous and convenient, since it allows us to isolate the computation-related phase of the development leading to program extraction from the proof-based phase establishing the correctness of the extracted program.

3.3 Evaluation Functions

In Figure 3 we show selected fragments of the mutually recursive definition of the graphs of the evaluation functions for terms and types as well as for substitutions, and the application function performing application of a function to an argument in the domain. Evaluation resembles standard denotational semantics, except that it is non-compositional. For example, the $[\![t\,s]\!]_{\eta} \searrow d_y$ constructor states that an application $t\,s$ in an environment η evaluates to a value d_y if only t and s within the environment η evaluate to values d_t and d_s, respectively, and applying d_t to d_s yields the final result d_y. Among other properties, we prove that the graphs are deterministic and thus represent actual functions.

```
Inductive EvalTm_Dom: Tm → DEnv → Prop :=
| evalVarD: ∀ d η , EvalTm_Dom TmVar (Dext η d)
| evalAbsD: ∀ A t η , EvalTm_Dom (TmAbs A t) η
| evalAppD: ∀ tf tx η , EvalTm_Dom tf η
   → EvalTm_Dom tx η
   → (∀ df dx, EvalTm tf η df → EvalTm tx η dx → App_Dom df dx)
   → EvalTm_Dom (TmApp tf tx) η
| evalSbD: ∀ η s t, EvalSb_Dom s η
   → (∀ η', EvalSb s η η' → EvalTm_Dom t η' )
   → EvalTm_Dom (TmSb t s) η
with EvalSb_Dom : Sb → DEnv → Prop :=
…
with App_Dom: D → D → Prop :=
| appCloD : ∀ tm η dx, EvalTm_Dom tm (Dext η dx)
   → App_Dom (Dclo tm η ) dx
…
```

Figure 4. Accessibility predicates for evaluation functions

```
Program Fixpoint evalTm (tm:Tm) ( η :DEnv) (H: EvalTm_Dom tm η )
  {struct H}: { d:D | EvalTm tm η d } :=
match tm as T return (tm = T → { d:D | EvalTm T η d }) with
| TmAbs A tm0 ⇒ fun _ ⇒ Dclo tm0 η
| TmVar ⇒ fun _ ⇒
   match η with
   | Did ⇒ _
   | Dext η ' d' ⇒ d'
   end
| TmAbs A tm0 ⇒ fun _ ⇒ Dclo tm0 η
| TmApp tf tx ⇒ fun H' ⇒
   let (df,Hdf) := evalTm (EvalTm_Dom_app_invF H H') in
    let (dx,Hdx) := evalTm (EvalTm_Dom_app_invX H H') in
     let (dy,Hdy) := app (EvalTm_Dom_app_invA H H' Hdf Hdx) in
      dy
| TmSb tm0 sb0 ⇒ fun H' ⇒
   let ( η' , Hη' ) := evalSb (EvalTm_Dom_sb_inv1 H H') in
    let (d, Hd) := evalTm (EvalTm_Dom_sb_inv2 H H' Hη' ) in
     d
…
end (eq_refl tm)
with evalSb (sb:Sb) ( η :DEnv) (H: EvalSb_Dom sb η )
  {struct H}: { η' :DEnv | EvalSb sb η η' } :=
…
with app (df dx:D) (H: App_Dom df dx)
  {struct H}: { d:D | App df dx d } :=
match df as DF return df = DF → { d:D | App DF dx d } with
| Dclo tm η ⇒ fun H' ⇒
   let (dy, Hdy) := evalTm (App_Dom_clo_inv H H') in
    let H := appClo Hdy in
     exist (fun d ⇒ App (Dclo tm η ) dx d) dy H
…
```

Figure 5. Evaluation functions

In Figure 4 we show the definition of the accessibility predicates. The `evalAppD` constructor states that `TmApp tf tx` and

$$\frac{m \vdash_{ne} d \searrow t}{m \vdash_{nf} \Downarrow (\Uparrow d) \searrow t}$$

$$\frac{m \vdash_{nf} \Downarrow D_A \searrow A \quad D_F \cdot \uparrow^{D_A} Dvar(m) \searrow D_B \quad m+1 \vdash_{nf} \Downarrow D_B \searrow B}{m \vdash_{nf} \Downarrow Dfun(D_A, D_F) \searrow \Pi A.\, B}$$

$$\frac{m \vdash_{nf} \Downarrow D_A \searrow A \quad D_F \cdot \uparrow^{D_A} Dvar(m) \searrow D_B \quad d_f \cdot \uparrow^{D_A} Dvar(m) \searrow d_b \quad m+1 \vdash_{nf} \Downarrow D_B d_b \searrow b}{m \vdash_{nf} \downarrow^{Dfun(D_A, D_F)} d_f \searrow \lambda A.b}$$

$$\frac{}{m \vdash_{ne} Dvar(j) \searrow v_{m-j-1}}$$

$$\frac{m \vdash_{ne} d_t \searrow t \quad m \vdash_{nf} d_s \searrow s}{m \vdash_{ne} Dapp(d_t, d_s) \searrow ts}$$

$$Rb^{nf}_m\, d = t \iff m \vdash_{nf} d \searrow t$$
$$Rb^{ne}_m\, d = t \iff m \vdash_{ne} d \searrow t$$

Figure 6. Graphs of read-back functions

```
Inductive RbNf_Dom : nat → DNf → Prop :=
| reifyNf_absD: ∀ f DA m F,  RbNf_Dom m (DdownN DA)
   → App_Dom F (Dup DA (Dvar m))
   → App_Dom f (Dup DA (Dvar m))
   → (∀ db DB, App F (Dup DA (Dvar m))  DB
        → App f (Dup DA (Dvar m))  db
        → RbNf_Dom (S m) (Ddown DB db))
    → RbNf_Dom m (DdownX (Dfun DA F) f)
| reifyNf_neD: ∀ m e,  RbNe_Dom m e
   → RbNf_Dom m (DdownN (DupN e))
| reifyNf_fun: ∀ m DA DF,  RbNf_Dom m (DdownN DA)
   → App_Dom DF (Dup DA (Dvar m))
   → (∀ DB, App DF (Dup DA (Dvar m))  DB
        → RbNf_Dom (S m) (DdownN DB))
   → RbNf_Dom m (DdownN (Dfun DA DF))
...
with RbNe_Dom: nat → DNe → Prop :=
| reifyNe_varD: ∀ m j,   RbNe_Dom m (Dvar j)
| reifyNe_appD: ∀ m e d,
   → RbNe_Dom m e
   → RbNf_Dom m d
   → ∀ ne nd, RbNe m e ne
   → RbNf m d nd
   → RbNe_Dom m (Dapp e d)
...
```

Figure 7. Accessibility predicates for read-back functions

environment `env` are in the function domain if any values `df` and `dx` computed by the recursive calls on `tf` and `tx`, respectively, are in the domain of the application function. It is here that the graph representation of the functions is used to refer to the values computed by the recursive calls, which eliminates the need for the induction-recursion scheme. The way how we refer to nested recursive calls also plays a key role in the encoding of partial functions. The premise is a function that returns a proof of an inductive predicate defined by mutual recursion, therefore the Coq system will consider the computed proof as structurally smaller and will allow a recursive call on it.

In Figure 5 we present the definition of the encoded evaluation functions. `evalTm` is defined by structural recursion over the additional parameter – the accessibility predicate. We use Coq with `Implicit Arguments` option that allows the user to omit those actual arguments which can be inferred from the context. In our case, the type of the accessibility predicate alone suffices to determine the remaining arguments. In order to exploit the technique of avoiding the induction-recursion scheme the function has to return not only a value, but also a proof that the arguments and the value satisfy the graph of the function.

Let us focus on the case of an application `TmApp tf tx`. Note that the only possible constructor for a proof term `H` is `evalAppD`. The function uses inversion functions to extract proofs that `tf` and `tx` within a given environment are in the domain, and then performs recursive calls on them. When it obtains the values `df` and `dx` with proofs that they belong to the function graph, the inverse function extracts the premise stating that any values belonging to the graph are also in the domain of the `app` function. We call this function on proofs returned from the recursive calls and obtain a structurally smaller proof on which we can perform the final nested recursive call. The encoded functions can also be seen as proofs that our definitions are consistent: for any argument value belonging to the function domain there exists a result value such that the pair of values belongs to the graph.

Note that the evaluation functions are suitable for extraction, as required. In Sections 4 and 5, we continue the theoretical work using the function graph instead, and without any harness related to extraction mechanism.

3.4 Read-back Functions

The next phase is reading back a computed value into a term in $\beta\eta$-normal form. The graphs of read-back functions are shown in Figure 6. The function Rb^{nf} is responsible for evaluating a closure of the reification function, which performs η-expansions, and for producing a term in $\beta\eta$-normal form. For example, the statement $m \vdash_{nf} \downarrow^{Dfun(D_A, D_F)} d_f \searrow \lambda A.b$ tells us that the closure of the reification function $\downarrow^{Dfun(D_A, D_F)} d_f$ will be extracted to a lambda-term when the value is annotated by a type-like value $Dfun(\ D_A,\ D_F)$ denoting a functional space. The function Rb^{nf} recursively extracts a term representing a type annotation from the type-like value representing the function domain. To extract a term representing the function

$$Nbe(\Gamma, t, A) = Rb^{nf}_{|\Gamma|}\left(\downarrow^{[\![A]\!]_{\eta_\Gamma}} [\![t]\!]_{\eta_\Gamma}\right)$$

Figure 8. The NbE procedure for terms

body, Rb^{nf} requires to compute a type-like value corresponding to its type. It creates a neutral-like value representing the formal argument as a function and then applies the value representing a dependent co-domain DF to such a value, obtaining a type-like value representing the type of the body. Finally, it recursively extracts the function body in normal form. Note that Rb^{nf} is type-driven and always performs η-expansion for values annotated by a functional type.

The statement $m \vdash_{nf} \Downarrow Dfun(D_A, D_F) \searrow \Pi A.\, B$ tells us how a term representing a dependent function space is extracted from an adequate type-like value. The statement $m \vdash_{nf} \Downarrow (\Uparrow d) \searrow t$ represents a computation when reification function has been called on a closure of the reflection function. It erases closures and delegates computation to the Rb^{ne} function.

The function Rb^{ne} is responsible for reading a neutral term from a neutral-term-like value. The statement $m \vdash_{ne} Dvar(j) \searrow v_{m-j-1}$ tells us how the domain representation of a free variable is converted into a term, taking care of replacing de Bruijn levels with de Bruijn indices. The statement $m \vdash_{ne} Dapp(d_t, d_s) \searrow ts$ tells us how neutral term representing an application, where computations were stopped by a free variable in a function position, is read back to a term. To this end, it recursively reads back a neutral term representing a function and a normal value representing an argument, and it concatenates them into a neutral term representing the application.

The definitions of the corresponding accessibility predicates are shown in Figure 7. The actual partial functions straightforwardly follow from the graph representations, and are not shown here.

3.5 The Normalization-by-Evaluation Procedure

The NbE procedure, shown in Figure 8, is a composition of the previously presented functions. It is a top level function that first computes a term in the domain and then creates a closure of the reification function and calls the read-back function on it. η_Γ is an environment computed from a typing context with neutral-like values representing free variables. This environment contains the meaning of free variables possibly occurring in a term and thus is suitable for evaluating the term.

3.6 Correctness

The correctness proof of the NbE procedure that we have formalized and report on in the rest of the article consists in establishing the following properties:

$$\frac{\forall m\, \exists n.\ m \vdash_{nf} e \searrow n \wedge m \vdash_{nf} e' \searrow n}{e = e' \in PerNf}$$

$$\frac{\forall m\, \exists n.\ m \vdash_{ne} e \searrow n \wedge m \vdash_{ne} e' \searrow n}{e = e' \in PerNe}$$

$$\frac{e = e' \in PerNe}{\uparrow^{Dempty} e = \uparrow^{Dempty} e' \in PerEmpty}$$

Figure 9. PER relations

- *Completeness*: for any two typable provably equal expressions, the NbE procedure returns the same canonical normal form.
- *Soundness*: any typable expression is provably equal to the canonical normal form computed by the NbE procedure.
- *Termination*: the NbE procedure is defined for any typable expression.

4 Establishing Completeness

In order to establish the completeness and termination of the normalization-by-evaluation procedure we need to build a model from Partial Equivalence Relations[1] (PER) over the domain values [Abel et al. 2007a]. In such a model each type is represented by a PER, where two values are related when they are considered as equal in the type system. There is also an interpretation function which assigns a PER to a type. To establish completeness it is shown that the evaluation of convertible terms of a given type results in values related by the PER assigned to the type. As a consequence these values will be read-back as the same term.

All relations used to model types are of type `relation D` which is defined as a function type `D → D → Prop`. We use the definition of PER from the Coq standard library, [2] defined as a type class parameterized by a given relation that contains proofs of the required properties. We find this approach quite convenient since it allows us to define a relation and then separately prove that it is a PER by instantiating the type class, which automatically makes the relation amenable to basic tactics like `transitive` and `symmetry`.

The formalization of most of the relations used to model types is straightforward and follows directly the mathematical definitions from the literature. Two auxiliary relations `PerNe` and `PerNf` relate those neutral or normal values which can be read-back to the same term regardless of the shift of de Bruijn indices in variables. It is common to use a notation $a = b \in R$ instead of $R(a, b)$ to underline that relation models equality, and we follow this idea in our formalization to increase readability.

[1] Partial Equivalence Relations are symmetric and transitive, but not necessarily reflexive.

[2] Module `Coq.Classes.RelationClasses`.

```
Definition InRel (D: Type) (R: relation D) := R.
Notation "d = d' ∈ R" := (InRel R d d')
 (at level 75, no associativity).

Definition binRelEq (D: Type) (R1 R2: relation D) :=
 (∀ x y, R1 x y ↔ R2 x y).
Notation "R1 ≈ R2" := (binRelEq R1 R2)
 (at level 75, no associativity).
Instance binRelEq_EQ D: Equivalence (·binRelEq D).

Inductive PerNe : relation DNe :=
| PerNe_intro: ∀ e e',
   (∀ m, ∃ n, RbNe m e n ∧ RbNe m e' n) → e = e' ∈ PerNe.
Instance PerNe_is_PER : PER (InRel PerNe).

Inductive PerNf : relation DNf :=
| PerNf_intro: ∀ e e',
   (∀ m, ∃ v, RbNf m e v ∧ RbNf m e' v) → e = e' ∈ PerNf.
Instance PerNf_is_PER : PER (InRel PerNf).

Inductive PerEmpty : relation D :=
| PerEmpty_ne: ∀ e e', e = e' ∈ PerNe
   → Dup Dempty e = Dup Dempty e' ∈ PerEmpty.
Instance PerEmpty_is_PER: PER (InRel PerEmpty).
```

Figure 10. PER relations and auxiliary definitions

```
Definition rel_oper (D: Type) := D → relation D → Prop.

Record ProperRelOper (A: relation D) (F: rel_oper D) :=
 Mk_ProperRelOper
  { ro_resp_arg: ∀ a0 a1 Y, a0 = a1 ∈ A → F a0 Y → F a1 Y
  ; ro_resp_ex: ∀ a, a = a ∈ A → ∃ B, F a B
  ; ro_resp_det: ∀ a0 a1 Y0 Y1, a0 = a1 ∈ A
     → F a0 Y0 → F a1 Y1 → Y0 ≈ Y1 }.

Inductive RelProd (A: relation D) (F: rel_oper D): relation D :=
| RelProd_intro: ∀ f0 f1,
   (∀ a0 a1, a0 = a1 ∈ A
     → ∃ y0 y1 Y,
        F a0 Y ∧ App f0 a0 y0 ∧ App f1 a1 y1 ∧ y0 = y1 ∈ Y)
   → f0 = f1 ∈ RelProd A F.

Record ProperPerProd (A: relation D) (F: rel_oper D) :=
 Mk_ProperPerOper
  { po_ro :> ProperRelOper A F
  ; po_per_codom: ∀ a Y, a = a ∈ A → F a Y → PER Y
  ; po_per_dom :> PER A }.

Instance ProperPerProd_is_PER A F (H:ProperPerProd A F):
   PER (RelProd A F).
```

Figure 11. PER product

In Figure 10 we show some auxiliary definitions and the PER relation used to model the empty type. Its mathematical counterpart is presented in Figure 9. Since this type is not inhabited its relation models equality only between open terms.

4.1 Dependent Function Space

Let $\mathcal{D}$ be an abstract domain suitable for evaluating expressions and let $Per(\mathcal{D})$ be a set of PERs over the domain. Let $Dom(\mathcal{A})$ be a set of all elements for which relation $\mathcal{A}$ is defined. A dependent function space is modeled by a product of PERs over a domain $\mathcal{D}$. If $\mathcal{A}$ is a PER representing a function domain and the dependent co-domain is represented by a family $\mathcal{F} : Dom(\mathcal{A}) \to Per(\mathcal{D})$ of PERs indexed by values from the function domain, then $\Pi(\mathcal{A}, \mathcal{F})$ is a PER that models the dependent type. The meaning of the relation is to relate only those values from $\mathcal{D}$ which represent functions preserving equality from the function domain w.r.t. the relation assigned to the particular co-domain:

$$\frac{\forall a = a' \in \mathcal{A}.\ f \cdot a = f' \cdot a' \in \mathcal{F}(a)}{f = f' \in \Pi(\mathcal{A}, \mathcal{F})}$$

The definition of the PER product is problematic for formalization since one of its components is a family of relations. It makes it harder to use since reasoning about equality of products is limited. To overcome this problem without any axioms like function or predicate extensionality we define an auxiliary equivalence relation `binEqRel` for binary predicates and use it instead of Coq's built-in equality.

We split the task of defining the PER product into several smaller steps. First, we formalize our representation of a family of PER relations in a way which is suitable for the rest of our formalization. Then we provide a generic definition of the relational product and define the properties that are required from a PER product. Finally, we instantiate the PER type class for any relational product that satisfies the required properties. All of these steps are shown in Figure 11.

We represent a family of relations by a relation, not by a function. We decided on such a representation because in this case using relations is more convenient than using partial functions and it makes it possible for us to overcome the lack of induction-recursion scheme in Coq, as explained in the next section. We call this relation a relation operator (`rel_oper` in the formalization). The `ProperRelOper A F` is a record containing proofs of the properties required from a relation operator `F` to represent a family of relations indexed by equivalence classes of a given binary relation `A`. The first two properties in the definition express that for each element in `A` we have an associated relation and the selection works for the entire equivalence class. The last property is more subtle, and it ensures us that all relations assigned to the elements of the same equivalence class are equivalent w.r.t. our auxiliary equivalence relation over binary predicates (`binEqRel`). This property could be seen as a weak variant of the requirement that the representation of the family of

$$\frac{}{Dunit = Dunit \in \mathcal{U}} \qquad \frac{}{Dempty = Dempty \in \mathcal{U}}$$

$$\frac{e = e' \in PerNe}{\uparrow^{Duniv} e = \uparrow^{Duniv} e' \in \mathcal{U}} \qquad \frac{d_a = d'_a \in \mathcal{U} \quad \forall x = x' \in |d_a|_{\mathcal{U}}.\ d_f \cdot x = d'_f \cdot x' \in \mathcal{U}}{Dfun(\ d_a,\ d_f) = Dfun(\ d'_a,\ d'_f) \in \mathcal{U}}$$

$$\begin{aligned} |-|_{\mathcal{U}} &: Dom(\mathcal{U}) \to Rel(\mathcal{D}) \\ |Dunit|_{\mathcal{U}} &= PerUnit \\ |Dempty|_{\mathcal{U}} &= PerEmpty \\ |\uparrow^{Duniv} e|_{\mathcal{U}} &= PerNe \quad \text{when } e = e \in PerNe \\ |Dfun(\ d_a,\ d_f)|_{\mathcal{U}} &= \Pi(|d_a|_{\mathcal{U}}, \lambda x.\ |d_f \cdot x|_{\mathcal{U}}) \end{aligned}$$

Figure 12. Universe of small types

```
Inductive InterpUniv: D → relation D → Prop :=
| InterpUniv_ne: ∀ de, de = de ∈ PerNe
    → InterpUniv (Dup Duniv de) PerNeUniv
| InterpUniv_empty: InterpUniv Dempty PerEmpty
| InterpUniv_unit: InterpUniv Dunit PerUnit
| InterpUniv_fun: ∀ DA DF PA PF, InterpUniv DA PA
    → ProperPerProd PA PF
    → (∀ a PB DB, a = a ∈ PA → App DF a DB → PF a PB
        → InterpUniv DB PB)
    → (∀ a, a = a ∈ PA → ∃ DB, App DF a DB)
    → InterpUniv (Dfun DA DF) (RelProd PA PF).

Definition InterpUnivPer (A:Tm) (η :DEnv) (PA: relation D): Prop :=
 ∃ DA, EvalTm A η DA ∧ InterpUniv DA PA.
```

Figure 13. PER interpretation function

```
Inductive PerUniv : relation D :=
| PerUniv_ne: ∀ e e', e = e' ∈ PerNe
    → Dup Duniv e = Dup Duniv e' ∈ PerUniv
| PerUniv_unit: Dunit = Dunit ∈ PerUniv
| PerUniv_empty: Dempty = Dempty ∈ PerUniv
| PerUniv_fun: ∀ DA DF DA' DF' PA, DA = DA' ∈ PerUniv
    → InterpUniv DA PA
    → (∀ a, a = a ∈ PA → ∃ DB, App DF a DB)
    → (∀ a, a = a ∈ PA → ∃ DB', App DF' a DB')
    → (∀ a0 a1 DB0 DB1 P , InterpUniv DA P
        → a0 = a1 ∈ P
        → App DF a0 DB0 → App DF' a1 DB1
        → DB0 = DB1 ∈ PerUniv)
    → Dfun DA DF = Dfun DA' DF' ∈ PerUniv.
```

Figure 14. PER model of the universe

relations is functional. We believe that using Coq's standard equality here would be too strong a requirement.

The relation `RelProd` resembles the mathematical rule given at the beginning of this section with the difference that we do not postulate or require any knowledge (like being a PER or a family of such) about its components. Then the record `ProperPerProd A F` contains proofs of the properties required to ensure that `RelProd A F` is a proper PER product. We require that the family of relations `F` is represented in a proper way, that it is a family of PERs, and that the domain of the function `A` is also a PER. All of that is sufficient to prove that a given product is actually a valid PER.

4.2 Universe of Small Types

Mathematically, the universe of small types *Univ* is represented by a structure $\langle \mathcal{U}, |.|_{\mathcal{U}} \rangle$, where $\mathcal{U}$ is a PER modeling the *Univ* type, and $|.|_{\mathcal{U}} : Dom(\mathcal{U}) \to Rel(\mathcal{D})$ is an interpretation function mapping a domain value representing a type to a PER modeling this type. This structure is similar to the universe à la Tarski [Martin-Löf 1998; Nordström et al. 1990], which is a well-known example of a structure defined with the induction-recursion scheme [Dybjer 2000]. See Figure 12.

The presence of the dependent function space requires us to use the interpretation function in the definition of the $\mathcal{U}$ relation. To say that two values representing a dependent function space are in the relation we have to make sure that for any values belonging to the *interpretation* of the value representing the function domain, the computed values representing co-domains are also in the relation. Therefore, the interpretation function and the relation have to be defined simultaneously.

Our first attempt was to use a graph of the interpretation function and define those two relations by mutual recursion, but unfortunately it does not seem possible. If we try to define an inductive predicate representing the graph of the interpretation function, then we would end up with a definition, where the predicate occurs negatively, and that is forbidden in Coq. So, instead, inspired by Girard's proof of strong normalization for System F [Girard et al. 1989], we decided to use impredicativity of `Prop` to break the mutual dependency. To this end, we first define an interpretation

relation InterpUniv without any reference to the not-yet defined PER representing the universe. The constructor for dependent function space takes any family of PERs PF that satisfies an additional property stating that any PER assigned to the element belonging to the function domain could also be obtained using InterpUniv. Later, due to impredicativity of the Prop universe, we can parameterize the constructor by the InterpUniv relation itself. Moreover, the additional property allows us to shift form reasoning about particular family of PERs to reasoning about the InterpUniv relation. Our definition is less precise than its mathematical counterpart, but it proved sufficient for our needs. The definition is shown in Figure 13.

The definition of the PerUniv relation for the universe of small types is straightforward once the interpretation relation is in place. It is shown in Figure 14.

4.3 Universe of Big Types

The universe of big types (universe of all types in the formalized system) is similar to the universe of small types, so the definition is exactly the same except that it also embeds the universe of small types. The definitions of the interpretation operator and the relation itself are presented below. We show only the single new case for embedding the universe of small types.

$$\frac{}{Duniv = Duniv \in \mathcal{T}}$$

$$|Duniv|_{\mathcal{T}} = \mathcal{U}$$

In the formalization the $\mathcal{T}$ universe is encoded as the PerType type. Its constructor for the universe takes an additional parameter which states that all values related by the PerUniv relations are also related by the PerType relation. This statement is trivial to prove, but we added it as a parameter to obtain a stronger induction hypothesis.

4.4 Characterization Theorem

The characterization theorem establishes properties required from NbE as far as terms are concerned. It ensures that all PERs used to model types are capable of relating neutral-term-like values and thus are sufficient to deal with open terms. It also ensures that the results of the reification function belong to the PerNf PER, which means that values modeled as equivalent are represented by the same term in normal form.

```
Theorem ReflectReify_Characterization: ∀ T T',
  T = T' ∈ PerType
  → ∀ PT, InterpType T PT
      → (∀ e e', e = e' ∈ PerNe
            → Dup T e = Dup T' e' ∈ PT)
        ∧ (∀ d d', d = d' ∈ PT
            → Ddown T d = Ddown T' d' ∈ PerNf)
        ∧ (DdownN T = DdownN T' ∈ PerNf).
```

```
Inductive ValEnv : Cxt → DEnv → DEnv → Prop :=
 | ValEnv_id: [ Did ] = [ Did ] ∈ nil
 | ValEnv_ext: ∀ η0 η1 Γ A d0 d1, [η0 ] = [η1 ] ∈ Γ
     → ∀ PA, InterpTypePer A η0 PA → d0 = d1 ∈ PA
     → [ Dext η0 d0 ] = [ Dext η1 d1 ] ∈ Γ ,A
where "[ d ] = [ d' ] ∈ Γ " := (ValEnv Γ d d').
Instance ValEnv_is_Per Γ (H : Γ |=) : PER(ValEnv Γ ).

Inductive ValCxt : Cxt → Prop :=
...
with ValTpEq: Cxt → Tm → Tm → Prop :=
| ValTp_intro: ∀ Γ A B, Γ |= →
   (∀ η0 η1 , [η0 ] = [η1 ] ∈ Γ
     → ∃ DA, ∃ DB, EvalTm A η0 DA ∧ EvalTm B η1 DB
         ∧ DA = DB ∈ PerType)
   → Γ |= A = B
where "Γ |= A = B" := (ValTpEq Γ A B)

with ValTmEq : Cxt → Tm → Tm → Tm → Prop :=
| ValTm_intro: ∀ Γ tm0 tm1 A, Γ |= A = A →
   (∀ η0 η1 , [η0 ] = [η1 ] ∈ Γ
     → ∃ dtm0, ∃ dtm1, ∃ PA, InterpTypePer A η0 PA
         ∧ EvalTm tm0 η0 dtm0 ∧ EvalTm tm1 η1 dtm1
         ∧ dtm0 = dtm1 ∈ PA)
   → Γ |= tm0 = tm1 : A
where "Γ |= t = t' : A" := (ValTmEq Γ t t' A)
```

Figure 15. Validity and PER for environments

4.5 Validity in the Model

To show completeness of the normalization-by-evaluation procedure we have to prove that the formalized system is adequate to build the model. To achieve this we need to define an appropriate semantics relation $\models$. We say that a given judgment is valid in a model when relations stated by the judgment are preserved by the model regardless of the interpretation of free variables.

All relations are defined simultaneously as inductive predicates and they resemble their mathematical counterparts. We use existential quantification for results of evaluation, and therefore, the validity relations additionally express termination of the evaluation procedure. Below is a fragment of the mathematical definition of the relation for terms and types. $\lceil\Gamma\rceil$ is a PER for environments containing values that belong to PERs modeling types in a given typing context. The Coq counterpart can be found in Figure 15.

$$\frac{\Gamma \models \quad \forall \eta = \eta' \in \lceil\Gamma\rceil.\ [\![A]\!]_\eta = [\![B]\!]_{\eta'} \in PerType}{\Gamma \models A = B}$$

$$\frac{\Gamma \models A = A \quad \forall \eta = \eta' \in \lceil\Gamma\rceil.\ [\![t]\!]_\eta = [\![t']\!]_{\eta'} \in \lceil [\![A]\!]_\pi \rceil}{\Gamma \models t = t' : A}$$

Finally, we prove that the syntactic judgments are semantically valid:

```
Fixpoint RelType (Γ :Cxt) (T: Tm) (DT DT': D)
  (HDT : DT = DT' ∈ PerType) : Prop :=
...
where "Γ ⊩ A ∈ HDT" := (RelType Γ A HDT)

Fixpoint RelType (Γ :Cxt) (T: Tm) (DT DT': D)
 (HDT : DT = DT' ∈ PerType) : Prop :=
match DT as _DT, DT' as _DT'
 return (DT = _DT → DT' = _DT' → Prop) with
  ...
| Dfun DA DF, Dfun DA' DF' ⇒ fun Heq Heq' ⇒
   Γ ⊢ t : T
   ∧ ∃ A F PT, InterpType (Dfun DA DF) PT
   ∧ dt = dt ∈ PT
   ∧ Γ ⊢ T = TmFun A F
   ∧ Γ ⊩ A ∈ (PerType_inv_Dfun_DA HDT Heq Heq')
   ∧ (∀ Δ i a da DB DB' PA (HPA : InterpType DA PA)
        (Hda : da = da ∈ PA) (Happ : App DF da DB)
        (Happ' : App DF' da DB'),
         CxtShift Δ i Γ
          → Δ ⊩ a : (TmSb A (Sups i)) ≈ da
              ∈ (PerType_inv_Dfun_DA HDT Heq Heq')
          → ∃ dy, App dt da dy
              ∧ Δ ⊩ TmApp (TmSb t (Sups i)) a :
                 TmSb F (Sext (Sups i) a) ≈ dy
                 ∈ (PerType_inv_Dfun_DB
                       HDT Heq Heq' HPA Hda Happ Happ'))
...
where "Γ ⊩ t : A ≈ d ∈ X" := (RelTerm Γ t A d X).
```

Figure 16. Logical relation for terms of functional type

Theorem 4.1 (Soundness of judgments). *For any judgment J we have $\Gamma \models J$ when $\Gamma \vdash J$.*

4.6 Completeness and Termination

From the semantic validity of the system, we prove completeness of NbE, stated in the following theorem:

```
Theorem NBE_Completeness: ∀ Γ t0 t1 T,
  Γ ⊢ t0 = t1 : T
  → ∃ v, Nbe T Γ t0 v ∧ Nbe T Γ t1 v.
```

The proofs are straightforward. Given two terms that are equal in the system, from the soundness of judgments we know that the evaluation procedure is defined for the given terms and the results are related by a PER assigned to the type of the terms. From the characterization theorem we know that the evaluated terms can be read-back to the same term in normal form. Therefore, the normalization-by-evaluation procedure computes the same canonical representation for equal terms.

The termination theorem could be seen as a special case of completeness for the $\Gamma \vdash t = t$ judgment:

```
Theorem NBE_Terminating: ∀ Γ t A,
  Γ ⊢ t : A → ∃ v, Nbe A Γ t v.
```

```
Inductive RelSb : Cxt → Sb → Cxt → DEnv → Prop :=
| RelSb_Sid: ∀ Γ sb, Γ ⊢ [ sb ] : nil
   → Γ ⊩ [ sb ] : nil ≈ Did
| RelSb_Sext: ∀ Γ Δ A M SB sb η dM DX DX'
   (HDX : DX = DX' ∈ PerType) PX,
    Γ ⊩ [ sb ] : Δ ≈ η
    → Γ ⊩ TmSb A sb ∈ HDX
    → Γ ⊩ M : TmSb A sb ≈ dM ∈ HDX
    → Γ ⊢[ SB ] = [Sext sb M ] : Δ ,A
    → [η ] = [η ] ∈ Δ
    → EvalTm A η DX
    → InterpType DX PX
    → dM = dM ∈ PX
    → Γ ⊩ [ SB ] : Δ ,A ≈ Dext η dM
where "Γ ⊩ [ s ] : Δ ≈ η " := (RelSb Γ s Δ η )
```

Figure 17. Logical relation for substitutions

Reasoning behind the mentioned theorems is presented in Section 6.

4.7 System Consistency

The PER model and its formalization are sufficient to prove the consistency of the system:

```
Theorem Consistency: ¬ ∃ M, nil ⊢ M : TmEmpty.
```

The technique to obtain such a proof using PER model has been shown in [Abel 2010]. It resembles similar proofs that use reasoning on term in normal form and it is a proof by contradiction: we first assume that it is possible to prove falsehood and then we use the validity theorems to obtain a corresponding value belonging to the PER modeling the empty type. But this relation contains only neutral-term-like values which cannot be obtained by evaluation in the empty context, thus such a value cannot exist.

5 Establishing Soundness

Soundness of the NbE procedure is proved using logical relations that establish a connection between the semantic and syntactic worlds. The primary role of those relations is to guarantee the following properties of reading back the canonical expression:

- $\Gamma \Vdash T \approx D_T \in \mathit{PerType}$. The expression T denoting a type is in relation with a type-like value D_T. It guarantees that $\Downarrow D_T$ reads back to an expression convertible with T.
- $\Gamma \Vdash t : T \approx d_t \in [D_T]$. The expression t is in relation with a value d_t belonging to a PER obtained by interpreting a type-like value D_T being in relation with the expression T. It guarantees that $\downarrow_{d_t}^{D_T}$ reads back to an expression convertible with t.

- $\Gamma \Vdash \gamma : \Psi \approx \eta$. The substitution γ is in relation with the environment η. It guarantees that each term occurring in the substitution is in the relation with the corresponding value in the environment.

The first two relations are defined mutually by induction over $T = T \in \mathit{PerType}$. We show here only the case for logical relations for terms of functional type:

- $\Gamma \Vdash t : T \approx f \in (\mathtt{Dfun}\ D_A\ D_F) \in \mathit{PerType}$ if
  - $\Gamma \vdash t : T$
  - there exist A and F such that
    - $\Gamma \vdash T = \Pi A.F$
    - $\Gamma \Vdash A \approx D_A \in \mathit{PerType}$
    - $\Delta \Vdash (t\uparrow^i) \cdot a : F[a] \approx f \cdot d_a \in [D_F \cdot d_a]$ for any context $\Gamma \leq^i \Delta$, term a and value d_a such that $\Delta \Vdash a : A\uparrow^i \approx d_a \in [D_A]$

where by $\Gamma \leq^i \Delta$ we denote that context Δ has been obtained by extending context Γ with i types.

It is impossible to formalize this definition as an inductive type because it would lead to a negative recursive occurrence of the defined type in the above case. Since the definition is formed by induction over the *PerType* relation we cannot provide a straightforward definition phrased inductively over the domain values or syntactic types. Thus we decided to define a partial function computing the formula by structural recursion over proof term for $T = T' \in \mathit{PerType}$. The technique used to encode such a function is similar to the one used to define partial functions in general – we have defined inversions for recursion that extracts a sub-term of a given proof term. The definition is shown in Figure 16. Note that the type-like value on the right hand side is not used, which makes the encoded definition slightly less elegant than the mathematical definitions from the literature. Figure 17 contains the definition of the logical relations for substitutions. Encoding them as an inductive type posed no problems.

5.1 Characterization of Reflection and Reification

The theorem that characterizes reflection and reification ensures us that we are able to read-back a canonical expression from a value in the model and the obtained normal form is convertible to an expression we started with. The theorem is shown in Figure 18.

5.2 Fundamental Theorem

The fundamental theorem, shown in Figure 19, states that if a judgment and a substitution which is in logical relation with an environment then the judgment will be in logical relation with evaluated expressions. Note that we used universal quantifiers only – we do not postulate that evaluation functions are terminating for judgment's expressions. We see that formulation more convenient to work with.

```
Theorem RelReify1: ∀ Γ T DT DT' (HDT : DT = DT' ∈ PerType),
 Γ ⊩ T ∈ HDT
  → (∃ A, RbNf (length Γ) (DdownN DT) A ∧ Γ ⊢ T = A)
   ∧ (∀ t dt, Γ ⊩ t : T ≈ dt ∈ HDT
       → (∃ v, RbNf (length Γ) (Ddown DT dt) v ∧ Γ ⊢ t = v : T))
   ∧ (∀ t k, k = k ∈ PerNe
       → (∀ Δ i, CxtShift Δ i Γ
           → ∃ tv, RbNe (length Δ) k tv
               ∧ Δ ⊢ TmSb t (Sups i) = tv : TmSb T (Sups i))
       → Γ ⊩ t : T ≈ (Dup DT k) ∈ HDT).
```

Figure 18. Characterization of reflection and reification

```
Theorem Rel_Fundamental: ∀ Γ ,
 (∀ Γ T, Γ ⊢ T → ∀ Δ SB η , Δ ⊩ [SB] : Γ ≈ η
   → ∀ DT DT', ∀ HDT : DT = DT' ∈ PerType, EvalTm T η DT
       → Δ ⊩ TmSb T SB ∈ HDT)
 ∧ (∀ Γ t T, Γ ⊢ t : T
     → ∀ Δ SB η , Δ ⊩ [SB] : Γ ≈ η
       → ∀ DT DT', ∀ HDT : DT = DT' ∈ PerType,
           ∀ dt, EvalTm T η DT → EvalTm t η dt
            → Δ ⊩ TmSb t SB : TmSb T SB ≈ dt ∈ HDT)
 ∧ (∀ Γ sb Ψ , Γ ⊢ [sb] : Ψ
     → ∀ Δ SB η , Δ ⊩ [SB] : Γ ≈ η
       → ∀ dsb, EvalSb sb η dsb
           → Δ ⊩ [Sseq sb SB] : Ψ ≈ dsb).
```

Figure 19. Fundamental theorem for logical relations

```
Definition nf Γ t A (H: Γ ⊢ t : A) : { v | Nf v ∧ Γ ⊢ t = v : A }.

Definition wtTm_nbetest Γ t s A (Ht: Γ ⊢ t : A) (Hs: Γ ⊢ s : A) :
 { Γ ⊢ t = s : A } + { ¬ Γ ⊢ t = s : A }.
```

Figure 20. Normalization function and decision procedure

5.3 Soundness of NbE

The soundness of normalization-by-evaluation procedure follows from the fundamental theorem. We first compute the identity environment and prove that it is in the logical relation with the identity substitution. Then the desired property follows directly from the logical relation obtained by the fundamental theorem.

```
Theorem Nbe_Soundness: ∀ Γ t T n,
  Γ ⊢ t : T → Nbe T Γ t n → Γ ⊢t = n : T.
```

6 Obtaining Certification

The decidability of equality over terms is a direct result of the properties of the normalization-by-evaluation procedure that we have proved. To obtain a certified decision procedure by program extraction we need to link the proven properties with the encoding of the partial functions of Section 3. Certified functions are present in Figure 20.

We briefly describe how we achieved totality of the presented functions. Let us focus on the given judgment $\Gamma \vdash t : A$. From Theorem 4.1 we obtain a proof that the judgment is reflected in the model, i.e., $\Gamma \models t : A$. Directly from the definition of the validity relation we obtain a proof that there exists a value d_t such that together with the identity environment η_Γ, it belongs to the graph of the evaluation function, i.e., $[\![t]\!]_{\eta_\Gamma} \searrow d_t$, and to the PER modeling the given type. From the characterization theorem presented in Section 4.4, we conclude that the reification of the value belongs to the *PerNf* relation which guarantees that there exists a term v that belongs to the graph of the read-back function, i.e., $|\Gamma| \vdash_{nf} \downarrow^{[\![A]\!]_{\eta_\Gamma}} d_t \searrow v$. Therefore, the value v belongs to the graph of the NbE function. From that we infer that the accessibility predicate for the function *nbe* holds.

The entire reasoning is carried out in `Prop` and thus it is not harnessed by the program extraction mechanism. Note that for any function, from the proof that a value belongs to the function graph we can show that the domain predicate holds for this value. Therefore, one can also directly prove that the domain predicate of the evaluator holds for any well-typed term, and use this function in other developments based on our formalization.

The two certified functions `nf` and `wtTm_nbetest` are corollaries of the fundamental theorem for logical relations presented in Section 5.2.

7 Conclusion

We have presented a formalization of Martin-Löf dependent type theory inside the Coq proof assistant which leads to a certified implementation of the procedures for normalization and deciding equality. Our work provides examples of how one can avoid the induction-recursion scheme in formalizing models of dependent type theories and how to use a graph-based variant of Bove-Capretta method to encode complex recursive functions.

Type theory with judgmental equality contains a considerable number of typing rules which had a direct impact on the size of our formalization. According to the `coqwc` command our formalization contains more than 7 thousand of specification lines and over 15 thousand lines of proof scripts. A modern desktop computer equipped with 3.2Ghz CPU takes about 6 minutes to verify the code.

Related work According to our knowledge there is no other Coq formalization of the NbE procedure for dependent type theory. However, such formalizations have been done in Agda [Buisse and Dybjer 2008; Danielsson 2006]. There is also a partial Agda formalization for dependent type theory presented in [Altenkirch and Kaposi 2016]. The main difference with our work and Agda formalizations is that we aim to obtain an extracted certified normalizer and we do not depend on the induction-recursion scheme.

Close to our work, but not related to NbE, is a Coq formalization of the Calculus of Constructions [Barras and Werner 1997]. The authors of this paper give a full formalization of the impredicative dependent type theory and have extracted not only a procedure for deciding equality, but also a proof-checker.

There exist also some successful formalizations of NbE for the simply typed λ-calculus both in Agda [Abel and Chapman 2014] and Coq [Garillot and Werner 2007], including instances of NbE obtained by program extraction from Tait-style proofs of strong normalization [Berger et al. 2006; Sozeau 2007].

Future work In our formalization we focused on certifying the normalizer and equality decision procedure only for terms. Of course, types are also taken into account by the procedures (terms contain type annotations), but there was no need to provide the corresponding top-level procedures for them. As a matter of fact, deriving such procedures in our formalization would be a straightforward exercise.

The normalization-by-evaluation procedure and its proof of correctness can play a key role in a construction of a certified type-checking algorithm for dependent types. Therefore, a natural extension of the formalization presented in this article would be to formalize such an algorithm and obtain a certified implementation of an idealized core of a proof assistant based on Martin-Löf dependent type theory. A definition and a correctness proof of a type-checker based on NbE is already present in [Abel et al. 2009].

In order to see if our approach, i.e., focusing on extraction and using impredicativity instead of induction-recursion, scales well one could extend the formalized type system with the identity types and the full hierarchy of predicative universes. Also, in terms of proof engineering, although we provided tactics to a number of fragments of our formalization, we believe that developing a richer library of dedicated tactics, that would allow for refactoring of our code, would make our formalization easier to reuse and adapt to other formulations of type theories.

In this work we did not provide a generic technique that could replace the induction-recursion scheme – we presented only an ad-hoc encoding developed specifically for our needs. It seems a worthwhile undertaking to investigate whether our approach is in any way related to the theoretical considerations of encoding the induction-recursion scheme known from the literaure [Hancock et al. 2013].

Acknowledgments

We would like to thank Małgorzata Biernacka, Filip Sieczkowski and the anonymous reviewers for their helpful comments on the presentation of this article.

This work has been supported by National Science Centre, Poland, grant no. 2014/15/B/ST6/00619.

References

Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. 1991. Explicit Substitutions. *J. Funct. Program.* 1, 4 (1991), 375–416. https://doi.org/10.1017/S0956796800000186

Andreas Abel. 2010. Towards Normalization by Evaluation for the $\beta\eta$-Calculus of Constructions. In *Functional and Logic Programming, 10th International Symposium, FLOPS 2010, Sendai, Japan, April 19-21, 2010. Proceedings (Lecture Notes in Computer Science)*, Matthias Blume, Naoki Kobayashi, and Germán Vidal (Eds.), Vol. 6009. Springer, 224–239. https://doi.org/10.1007/978-3-642-12251-4_17

Andreas Abel, Klaus Aehlig, and Peter Dybjer. 2007a. Normalization by Evaluation for Martin-Löf Type Theory with One Universe. *Electr. Notes Theor. Comput. Sci.* 173 (2007), 17–39.

Andreas Abel and James Chapman. 2014. Normalization by Evaluation in the Delay Monad: A Case Study for Coinduction via Copatterns and Sized Types. In *Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014. (EPTCS)*, Paul Levy and Neel Krishnaswami (Eds.), Vol. 153. 51–67. https://doi.org/10.4204/EPTCS.153.4

Andreas Abel, Thierry Coquand, and Peter Dybjer. 2007b. Normalization by Evaluation for Martin-Löf Type Theory with Typed Equality Judgements. In *22nd IEEE Symposium on Logic in Computer Science (LICS 2007), 10-12 July 2007, Wroclaw, Poland, Proceedings*. IEEE Computer Society, 3–12. https://doi.org/10.1109/LICS.2007.33

Andreas Abel, Thierry Coquand, and Miguel Pagano. 2009. A Modular Type-Checking Algorithm for Type Theory with Singleton Types and Proof Irrelevance. In *Typed Lambda Calculi and Applications, 9th International Conference, TLCA 2009, Brasilia, Brazil, July 1-3, 2009. Proceedings (Lecture Notes in Computer Science)*, Pierre-Louis Curien (Ed.), Vol. 5608. Springer, 5–19. https://doi.org/10.1007/978-3-642-02273-9_3

Robin Adams. 2006. Pure Type Systems with Judgemental Equality. *J. Funct. Program.* 16, 2 (2006), 219–246. https://doi.org/10.1017/S0956796805005770

Klaus Aehlig and Felix Joachimski. 2004. Operational aspects of untyped Normalisation by Evaluation. *Mathematical Structures in Computer Science* 14, 4 (2004), 587–611. https://doi.org/10.1017/S096012950400427X

Thorsten Altenkirch and Ambrus Kaposi. 2016. Normalisation by Evaluation for Dependent Types. In *1st International Conference on Formal Structures for Computation and Deduction, FSCD 2016, June 22-26, 2016, Porto, Portugal (LIPIcs)*, Delia Kesner and Brigitte Pientka (Eds.), Vol. 52. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 6:1–6:16. https://doi.org/10.4230/LIPIcs.FSCD.2016.6

Bruno Barras and Benjamin Werner. 1997. Coq in Coq. (1997). Unpublished note.

Ulrich Berger, Stefan Berghofer, Pierre Letouzey, and Helmut Schwichtenberg. 2006. Program Extraction from Normalization Proofs. *Studia Logica* 82, 1 (2006), 25–49. https://doi.org/10.1007/s11225-006-6604-5

Ulrich Berger and Helmut Schwichtenberg. 1991. An Inverse of the Evaluation Functional for Typed lambda-calculus. In *Proceedings of the Sixth Annual Symposium on Logic in Computer Science (LICS '91), Amsterdam, The Netherlands, July 15-18, 1991*, Giles Kahn (Ed.). IEEE Computer Society, 203–211. https://doi.org/10.1109/LICS.1991.151645

Yves Bertot and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Springer. https://doi.org/10.1007/978-3-662-07964-5

Ana Bove. 2009. Another Look at Function Domains. *Electr. Notes Theor. Comput. Sci.* 249 (2009), 61–74. https://doi.org/10.1016/j.entcs.2009.07.084

Ana Bove and Venanzio Capretta. 2001. Nested General Recursion and Partiality in Type Theory. In *Theorem Proving in Higher Order Logics, 14th International Conference, TPHOLs 2001, Edinburgh, Scotland, UK, September 3-6, 2001, Proceedings (Lecture Notes in Computer Science)*, Richard J. Boulton and Paul B. Jackson (Eds.), Vol. 2152. Springer, 121–135. https://doi.org/10.1007/3-540-44755-5_10

Alexandre Buisse and Peter Dybjer. 2008. Towards Formalizing Categorical Models of Type Theory in Type Theory. *Electr. Notes Theor. Comput. Sci.* 196 (2008), 137–151. https://doi.org/10.1016/j.entcs.2007.09.023

Thierry Coquand and Peter Dybjer. 1997. Intuitionistic Model Constructions and Normalization Proofs. *Mathematical Structures in Computer Science* 7, 1 (1997), 75–94. https://doi.org/10.1017/S0960129596002150

Nils Anders Danielsson. 2006. A Formalisation of a Dependently Typed Language as an Inductive-Recursive Family. In *Types for Proofs and Programs, International Workshop, TYPES 2006, Nottingham, UK, April 18-21, 2006, Revised Selected Papers (Lecture Notes in Computer Science)*, Thorsten Altenkirch and Conor McBride (Eds.), Vol. 4502. Springer, 93–109. https://doi.org/10.1007/978-3-540-74464-1_7

Peter Dybjer. 2000. A General Formulation of Simultaneous Inductive-Recursive Definitions in Type Theory. *J. Symb. Log.* 65, 2 (2000), 525–549. https://doi.org/10.2307/2586554

Andrzej Filinski and Henning Korsholm Rohde. 2004. A Denotational Account of Untyped Normalization by Evaluation. In *Foundations of Software Science and Computation Structures, 7th International Conference, FOSSACS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings (Lecture Notes in Computer Science)*, Igor Walukiewicz (Ed.), Vol. 2987. Springer, 167–181. https://doi.org/10.1007/978-3-540-24727-2_13

François Garillot and Benjamin Werner. 2007. Simple Types in Type Theory: Deep and Shallow Encodings. In *Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007, Kaiserslautern, Germany, September 10-13, 2007, Proceedings (Lecture Notes in Computer Science)*, Klaus Schneider and Jens Brandt (Eds.), Vol. 4732. Springer, 368–382. https://doi.org/10.1007/978-3-540-74591-4_27

Jean-Yves Girard, Paul Taylor, and Yves Lafont. 1989. *Proofs and Types*. Cambridge University Press.

Peter Hancock, Conor McBride, Neil Ghani, Lorenzo Malatesta, and Thorsten Altenkirch. 2013. Small Induction Recursion. In *Typed Lambda Calculi and Applications, 11th International Conference, TLCA 2013, Eindhoven, The Netherlands, June 26-28, 2013. Proceedings (Lecture Notes in Computer Science)*, Masahito Hasegawa (Ed.), Vol. 7941. Springer, 156–172. https://doi.org/10.1007/978-3-642-38946-7_13

Robert Pieter Nederpelt Lazarom. 1973. *Strong Normalization in a Typed Lambda Calculus with Lambda Structured Types*. Ph.D. Dissertation. Technische Universiteit Eindhoven.

Pierre Letouzey. 2008. Extraction in Coq: An Overview. In *Logic and Theory of Algorithms, 4th Conference on Computability in Europe, CiE 2008, Athens, Greece, June 15-20, 2008, Proceedings (Lecture Notes in Computer Science)*, Arnold Beckmann, Costas Dimitracopoulos, and Benedikt Löwe (Eds.), Vol. 5028. Springer, 359–369. https://doi.org/10.1007/978-3-540-69407-6_39

Per Martin-Löf. 1998. An Intuitionistic Theory of Types. In *Twenty-Five Years of Constructive Type Theory*, Giovanni Sambin and Jan M. Smith (Eds.). Oxford University Press.

Bengt Nordström, Kent Petersson, and Jan M. Smith. 1990. *Programming in Martin-Löf's Type Theory: An Introduction*. Oxford University Press.

Miguel Pagano. 2012. *Type-Checking and Normalisation By Evaluation For Dependent Type Systems*. Ph.D. Dissertation. Universidad Nacional De Córdoba.

Matthieu Sozeau. 2007. A Dependently-Typed Formalization of Simply-Typed Lambda-Calculus: Substitution, Denotation, Normalization. (2007). Unpublished note.