Normalization by Evaluation for Simple Types: Formalization in Coq and Overview

Yuxiang Pan

December 18, 2023

Abstract

Understanding a logical expression can sometimes be cumbersome. An expression can have a simple semantic while is represented in a complex syntax structure, or two expressions with the same semantic can be represented by different forms. These examples make the understanding of logical expression unmanageable. Presenting Normalization by Evaluation (NbE), a powerful normalization concept in the realm of type theory and functional programming that aims to establish a consistent normal form of representing typable expressions. This technique integrates normalization and evaluation into a single process and is applicable in many instances such as compiler design, providing a means to normalize expressions and produce efficient code.

1 Introduction

The principle of NbE is to bridge the gap between syntactic structures and their corresponding semantic interpretations, thereby facilitating the analysis and manipulation of programs within a formal framework. In the case of simple types, which consist of basic data types (natural numbers in this paper) and function types, NbE serves as a method for systematically reducing complex expressions to their simplest form. The process begins with the representation of types and terms within a formal system, for instance, the lambda calculus. Types are used to classify expressions, and terms denote the computational content. However, expressions may exhibit redundancies or non-essential details that do not affect their fundamental meaning, which entails inconsistent representations of the same semantic. As a way out, NbE simplifies logical constructs by evaluating their structure into a consistent semantic form.

The "evaluation" part of NbE involves interpreting terms in a computational model. This model can be thought of as a machine that simulates the behavior of the programming language. By executing the terms in this model, NbE reduces expressions to their normal form by performing reduction rules on their operational semantics. Simultaneously, the "normalization" part reads back the evaluated expressions into a canonical or simplified representation that captures their essential meaning. The result is a normalized expression that retains the same meaning as the original but in a concise and consistent form. Such a concept of conversion is introduced by Martin-Löf [ML75].

In simple types, NbE enables the reduction of higher-order functions and applications to their simplest forms. This simplification not only aids in understanding and reasoning about programs but also provides a foundation for optimizing code and ensuring that equivalent expressions are treated as such.

In terms of real-life application, NbE has found use in various areas of computer science, including proof assistants, functional programming languages, and type theory. By connecting the logical and computational aspects of a programming language, NbE contributes to the development of more robust and expressive systems, enhancing the ability to reason about programs formally and systematically.

1.1 Related Work

This paper follows closely the work of Andreas Abel [Abe13] for the simple types. However, note that there are countless other design choices for normalization and for NbE in general. For similar work, Abel [ea07] and Wieczorek [ea18] introduce NbE for Martin-Lof type theory with beta eta equality, which features dependent types instead of simple types. Dependent types are more real-life applicable than the simple types present in this paper, however, the former is more cumbersome to mechanize during the partial equivalence relation. In addition, Altenkirch [ea16] and Cubric [ea98] take another approach for dependent types NbE by using the Presheaf categories and Yoneda embedding, which focus on capturing context and environment.

Furthermore, Abel and Pientka [AA10] discuss explicit substitution approaches in the normalization with the dependent types framework. This project also chooses explicit substitution since it forces a direct manipulation of substitution in the mechanization, provides a more controlled way of representing the concept, and generally makes the lemmas more compact. The above paper [AA10] explores explicit substitution more deeply by analyzing the vanilla explicit substitution with lazy and single-pass approaches, which definitely are feasible extensions for a future project.

As for more distant extension, Berger [ea91b] presents a version of normalization that avoids explicit beta eta reductions by inverting the evaluation function for lambdas, and Altenkirch [ea95] constructs a proof of Berger's work in category theory. To sum up, the relation between NbE for the simple types and, not limited to, the mentioned others is that one can think of the former as the baseline version of the concept that can be further extended to more complicated systems. The mechanization in this paper builds on top of the baseline with the addition of de Bruijn indices for variables and explicit substitution.

The advantage of such variable representation is to avoid renaming problems, thus making substitution easier to handle. One can also use de Bruijn levels to represent variables. The difference is when using indices, one needs to shift the indices of unbound variables; while with levels, one needs to shift the levels of bound variables, which causes a duality. However, in the context of this paper, bound variables never appear as the work is around closure. Since there is nothing to shift, indices are preferred.

2 Simple Types

The mechanization of simple types in this paper consists of Gödel's System T [God58], de Bruijn indices for variables, natural numbers, function abstraction and application, primitive recursion, and explicit substitution [ea91a]. Primitive recursion installs an intuitive and consistent way to define functions over natural numbers, which facilitates the reasoning about the properties of these functions. Explicit substitution 2.1 provides a clear representation of the substitution process and simplifies the overall implementation. The Gödel's System T is defined as follows:

$\boxed{S, T \in tp}$ Types.

$$\frac{}{N \in tp} \text{ type of natural numbers} \qquad \frac{S \in tp \quad T \in tp}{S \to T \in tp} \text{ function type}$$

$\boxed{\Gamma \in context}$ Typing contexts.

$$\frac{}{() \in context} \text{ empty context} \qquad \frac{\Gamma \in context \quad x \notin \Gamma}{(\Gamma, x : T) \in context} \text{ context extension}$$

$\boxed{c \in Cst^T}$ Constants of type T.

$$\frac{}{zero : N} \text{ zero} \qquad \frac{}{succ : N \to N} \text{ successor function}$$

$$\frac{}{recc : T \to (N \to T \to T) \to N \to T} \text{ primitive recursion into type T}$$

$\boxed{r, s, t \in tm_\Gamma^T}$ Well-typed terms.

$$\frac{c : T}{\Gamma \vdash c : T} \text{ constant} \qquad \frac{(x : T) \in \Gamma}{\Gamma \vdash x : T} \text{ variable}$$

$$\frac{\Gamma, x : S \vdash t : T}{\Gamma \vdash \lambda x t : S \to T} \text{ function abstraction} \qquad \frac{\Gamma \vdash r : S \to T \quad \Gamma \vdash s : S}{\Gamma \vdash r s : T} \text{ function application}$$

The terms $t \in tm$ are given as follows:

$$tm \ni r, s, t ::= x_i \mid zero \mid succ(t) \mid \lambda t \mid r\ s \mid recc(t_z, t_s, t_n)$$

Where x_i represents the ith variable, $zero$ and $succ(t)$ are natural numbers, λt is abstracting the 0th variable in t, $r\ s$ is applying r to s, and $recc(t_z, t_s, t_n)$ is primitive recursion.

2.1 Explicit Substitution

Substitution is an indispensable component in functional programming. Instead of using the classical representation of substitution as an independent operation, the mechanization in this paper installs substitution explicitly as a term in the form of closure.

By following Abadi's definition [ea91a], substitution extends the term definition with substitution application, and the grammar extends as follows:

$$tm \ni r, s, t ::= \ldots \mid t\ \sigma$$
$$subs \ni \sigma, \tau ::= id \mid \uparrow \mid \sigma\tau \mid (\sigma, s)$$

Where $t\ \sigma$ represents substitution application, id is the identity substitution, $\uparrow$ is index shift or weakening which increases all de Bruijn indices by 1 and can be useful in, for example, variable lookup or eta-expansion. $\sigma\tau$ is substitution composition that offers an orderly way of performing multiple substitutions, and (σ, s) is substitution extension that extends σ with a term s. Typing rules extend as well:

$$\frac{\Gamma \models t : T \quad \Delta \models \sigma : \Gamma}{\Delta \models t\ \sigma : T}$$

$$\frac{}{\Gamma \models id : T} \qquad \frac{}{\Gamma, S \models \uparrow : \Gamma} \qquad \frac{\Gamma_1 \models \tau : \Gamma_2 \quad \Gamma_2 \models \sigma : \Gamma_3}{\Gamma_1 \models \sigma\tau : \Gamma_3} \qquad \frac{\Gamma \models \sigma : \Delta \quad \Gamma \models s : S}{\Gamma \models (\sigma, s) : \Delta, S}$$

2.2 Lemmas

This paper chooses to mechanize term equality with extensionality to prove term reflexivity, and presupposition as a way to show the correctness of the lambda calculus so far.

$$\Gamma \models t = t' : T \quad \text{term equality}$$
$$\Gamma \models \sigma = \sigma' : \Delta \quad \text{substitution equality}$$

Together with extensionality rules.

$$\frac{\Gamma, S \vdash t = t' : T}{\Gamma \vdash \lambda t = \lambda t' : S \to T} \text{ weak function extensionality}$$

$$\frac{\Gamma \vdash t : S \to T}{\Gamma \vdash t = \lambda.(t \uparrow) x_0 : S \to T} \text{ function extensionality}$$

$$\frac{\Gamma \vdash \sigma : \Delta \quad \Delta, S \vdash t : T}{\Gamma \vdash (\lambda t)\sigma = \lambda.t(\sigma \uparrow, x_0) : S \to T} \text{ substitution in lambda}$$

Proving reflexivity strengthens the logical consistency of the system, while presupposition supports valid reasoning, both promote the system's correctness. In the context of logical systems, there is plenty of freedom in choosing lemmas to prove the correctness, such as the law of excluded middle, contrapositive, and distributive properties. In this paper, it is equally feasible to prove symmetry and transitivity in definitional equality.

Reflexivity:

$$t \in tm_T^\Gamma \longleftrightarrow \Gamma \models t = t : T$$
$$\sigma \in subs_\Delta^\Gamma \longleftrightarrow \Gamma \models \sigma = \sigma : \Delta$$

Presupposition:

$$\Gamma \models t = s : T \longleftrightarrow t \in tm_T^\Gamma \ \wedge \ s \in tm_T^\Gamma$$
$$\Gamma \models \sigma = \tau : \Delta \longleftrightarrow \sigma \in subs_\Delta^\Gamma \ \wedge \ \tau \in subs_\Delta^\Gamma$$

In the mechanization, both lemmas are trivial to prove as they only require applying equality rules and unfolding definitions. However, presupposition may cause some slight problems as Coq's automation tactic is powerful but incomplete, hence it needs manual guidance. Another observation is that with the addition of explicit substitution and extensionality, the rules become messy and the overall readability is reduced, which will cause some mild headaches in choosing the right rule.

3 Normalization

In the context of NbE, normalization refers to rewriting or simplifying expressions according to a transformation function into a domain. The goal is to reach a form where no further reductions are possible. This process ensures that expressions are consistent and avoids issues such as non-termination or divergence.

3.1 Domain Model

The domain model consists of operational semantic values for logical expressions, which form an applicative structure. This is essential to NbE since both evaluation and readback functions depend on the domain. Then, normal and neutral terms are defined. A term is in its normal form if no further reduction is possible according to the typing rules of the system. A neutral form is a specific kind of normal form, however, it contains subterms that can not be fully reduced. In other words, a term is in its neutral form when its computation is only partially completed. Their grammar is as follows:

$$nf \ \ni \ v \ ::= \ u \mid zero \mid succ(v) \mid \lambda v$$
$$ne \ \ni \ u \ ::= \ x_i \mid u \ v \mid recc(v_z, v_s, u)$$

Then, normal ($\Leftarrow$) terms for system T are given by the following rules:

$$\frac{\Gamma \vdash u \Rightarrow N}{\Gamma \vdash u \Leftarrow N} \qquad \frac{}{\Gamma \vdash zero \Leftarrow N} \qquad \frac{\Gamma \vdash v \Leftarrow N}{\Gamma \vdash succ(v) \Leftarrow N} \qquad \frac{\Gamma, x : S \vdash v \Leftarrow T}{\Gamma \vdash \lambda x v \Leftarrow S \to T}$$

And for neutral ($\Rightarrow$) terms:

$$\frac{(x : T) \in \Gamma}{\Gamma \vdash x \Rightarrow T} \qquad \frac{\Gamma \vdash u \Leftarrow S \leftarrow T \quad \Gamma \vdash v \Leftarrow S}{\Gamma \vdash u \ v \Rightarrow T} \qquad \frac{\Gamma \vdash v_s \Leftarrow T \quad \Gamma \vdash v_s \Leftarrow N \to T \to T \quad \Gamma \vdash u \Rightarrow N}{\Gamma \vdash recc(v_s, v_s, u) \Rightarrow T}$$

Now, to define the domain model D, an applicative structure is required to correctly represent neutrals. Hence, D is encoded as follows:

$$D \ni a, b ::= zero \mid succ(a) \mid (\underline{\lambda} t)\rho \mid updt^T(e)$$

$$DNf \ni d ::= down^T(a)$$

$$DNe \ni e ::= x_k \mid e\ d \mid recc(d_z, d_s, e)$$

DNf represents the functional closure of reifying a domain value to a normal value, and DNe consists of neutrals. As for the evaluation environment, ρ is defined as $\rho \in Env = N \to D$, and can be extended $(\rho, d)(n)$ using pattern matching on the natural number n as $(\rho, d)(0) = d$ and $(\rho, d)(S\ n') = \rho(n')$. On a side note, in mechanization the environment may cause confusion in algorithm encoding since Env is the syntactic sugar for $N \to D$ instead of being an independent object, then a returning object can be a Env value as well as a function value of $N \to D$ and both cases are valid as they do type-check.

3.2 Evaluation Function

The evaluation function has the goal of interpreting logical expressions to obtain their semantic meaning and transform them into a domain value. It also performs all beta reductions. As for the implementation, the evaluation function for terms is mutually inductive with function application and extends to recursion and substitution. Function application consists of the beta-reduction case and a general neutral case. Recursion performs the base case for the domain value zero and the step case for the successor. Notice that the evaluation function is non-compositional since in an environment ρ if r is evaluated to a domain value f, s to a, and f applied to a gives b, then the evaluation of r applied to s yields b. The evaluation of term t in the environment ρ yields domain value d is denoted by $[\![t]\!](\rho) \searrow d$, and the rules are:

$$\frac{}{[\![x_i]\!](\rho) \searrow \rho(i)} \qquad \frac{}{[\![zero]\!](\rho) \searrow zero} \qquad \frac{[\![t]\!](\rho) \searrow d}{[\![succ(t)]\!](\rho) \searrow succ(d)}$$

$$\frac{}{[\![\lambda t]\!](\rho) \searrow (\underline{\lambda} t)\rho} \qquad \frac{[\![r]\!](\rho) \searrow f \quad [\![s]\!](\rho) \searrow a \quad f \cdot a \searrow b}{[\![r\ s]\!](\rho) \searrow b}$$

$$\frac{[\![t_z]\!](\rho) \searrow d_z \quad [\![t_s]\!](\rho) \searrow d_s \quad [\![t_n]\!](\rho) \searrow d_n \quad recc(d_z, d_s, d_n) \searrow d}{[\![recc(t_z, t_s, t_n)]\!](\rho) \searrow d}$$

As for mutual inductive evaluation of function application, and recursion:

$$\frac{[\![t]\!](\rho, a) \searrow b}{(\underline{\lambda} t)\rho \cdot a \searrow b} \qquad \frac{}{e \cdot d \searrow e\ d}$$

$$\frac{}{recc(d_z, d_s, zero) \searrow d_z} \qquad \frac{recc(d_z, d_s, d_n) \searrow a \quad d_s \cdot d_n \searrow f \quad f \cdot a \searrow b}{recc(d_z, d_s, succ(d_n)) \searrow b}$$

Lastly, the rules of substitution are:

$$\frac{[\![\sigma]\!](\rho) \searrow \rho' \quad [\![t]\!](\rho') \searrow a}{[\![t\ \sigma]\!](\rho) \searrow a}$$

$$\frac{}{[\![id]\!](\rho) \searrow \rho} \qquad \frac{}{[\![\uparrow]\!](\rho, a) \searrow \rho} \qquad \frac{[\![\tau]\!](\rho) \searrow \rho' \quad [\![\sigma]\!](\rho') \searrow \rho''}{[\![\sigma\tau]\!](\rho) \searrow \rho''} \qquad \frac{[\![\sigma]\!](\rho) \searrow \rho' \quad [\![s]\!](\rho) \searrow a}{[\![(\sigma, s)]\!](\rho) \searrow (\rho', a)}$$

An important detail in the mechanization of the evaluation function is to apply the correct environment extension and type transformation while dealing with function application and substitution. One needs to understand the meaning behind the evaluation result of these cases, instead of aiming for the type-check. This argument is equally applicable to the mechanization of the readback function, which needs correct domain transformation.

3.3 Readback Function

The readback function converts the resulting domain values from the evaluation function into a normal form and restores all beta-reductions by performing eta-expansions, thus it will be mutually inductive on the normal and neutral forms, denoted by R_n^{nf} and R_n^{ne}. Since reading back a closure suggests the evaluation of the function body where not necessarily all cases are defined, the readback function is partial. The rules for R_n^{nf} are given as follows:

$$\frac{}{R_n^{nf}\ zero \searrow zero} \qquad \frac{R_n^{nf}\ d \searrow v}{R_n^{nf}\ succ(d) \searrow succ(v)}$$

$$\frac{[\![t]\!](\rho, x_n) \searrow b \qquad R_n^{nf}\ b \searrow v}{R_n^{nf}\ (\underline{\lambda} t)\rho \searrow \lambda v} \qquad \frac{R_n^{ne}\ e \searrow u}{R_n^{nf}\ e \searrow u}$$

Similarly for R_n^{ne}:

$$\frac{}{R_n^{ne}\ x_k \searrow v_{(n-k-1)}} \qquad \frac{R_n^{ne}\ e \searrow u \qquad R_n^{nf}\ d \searrow v}{R_n^{ne}\ e\ d \searrow u\ v} \qquad \frac{R_n^{nf}\ d_z \searrow v_z \qquad R_n^{nf}\ d_s \searrow v_s \qquad R_n^{ne}\ e \searrow u}{R_n^{ne}\ recc(d_z, d_s, e) \searrow recc(v_z, v_s, u)}$$

3.4 Lemmas

To ensure the correctness of the evaluation function, a lemma is required to demonstrate that the evaluation function outputs the same domain value for the same term evaluated in the same environment. The lemma will be mutually inductive with function application, recursion, and substitution.

$$\forall (a, a' : D).\ ([\![t]\!](\rho) \searrow a) \wedge ([\![t]\!](\rho) \searrow a') \longleftrightarrow a = a'$$
$$\forall (b, b' : D).\ (f \cdot a \searrow b) \wedge (f \cdot a \searrow b') \longleftrightarrow b = b'$$
$$\forall (d, d' : D).\ (recc(d_z, d_s, d_n) \searrow d) \wedge (recc(d_z, d_s, d_n) \searrow d') \longleftrightarrow d = d'$$
$$\forall (\rho, \rho' : Env).\ ([\![\sigma]\!](\rho) \searrow \rho') \wedge ([\![\sigma]\!](\rho) \searrow \rho'') \longleftrightarrow \rho' = \rho''$$

Similarly, for the readback function, the normalized value must be the same for the same inputs to satisfy the correctness. The lemma will be mutually inductive with normal and neutral values.

$$\forall (v, v' : nf).\ (R_n^{nf}\ d \searrow v) \wedge (R_n^{nf}\ d \searrow v') \longleftrightarrow v = v'$$
$$\forall (u, u' : ne).\ (R_n^{ne}\ d \searrow u) \wedge (R_n^{ne}\ d \searrow u') \longleftrightarrow u = u'$$

4 Completeness

Completeness is the property that NbE covers all terms, which is important as it provides confidence in the correctness of the normalization algorithm and in its ability to handle all possible terms in the language. To establish completeness, the demonstration that all terms can be normalized is required. As an outline, a partial equivalence relation, which is a symmetric and transitive relation, is needed to model extensional function equality. Then, proving realizability shows the convertibility of an abstract construction to a concrete computation. Fundamental theorems are essential as they ensure properties such as termination and uniqueness.

4.1 Partial Equivalence Relation

Extensional function equality means that two functions are considered equal if, for all possible inputs, they produce the same output. In the context of this paper, it can be modeled using a partial equivalence relation (PER). A PER is an equivalence relation but not defined for all pairs of elements,

which is desired to apply on closure. Consequently, PERs are often modeled using groupoids [Abe09] since by definition, the latter is an algebraic structure that consists of a non-empty set with a binary partial function. Moreover, groupoids are isomorphic with PER in operations, where the existence of inverse corresponds to symmetry, and the associativity of morphism composition corresponds to transitivity. Ultimately, PER corresponds to the subgroupoids of D $\times$ D. An immediate drawback of formalizing PER in Coq is that the process can not be done directly as explained in section 5.

4.2 Realizability

Realizability proves that the read-back set of values is well-contained between the spaces of the read-back normal form and read-back neutral form, which are defined as Top ($\top$) and Bot ($\bot$) respectively. Consider the PER model, Top and Bot are defined as $\top \subseteq \mathrm{D}^{nf} \times \mathrm{D}^{nf}$ and $\bot \subseteq \mathrm{D}^{ne} \times \mathrm{D}^{ne}$. Together with read back function, the rules are as follows:

$$d = d' \in \top \longleftrightarrow \forall n. R_n^{nf} d = R_n^{nf} d' \in nf$$
$$e = e' \in \bot \longleftrightarrow \forall n. R_n^{ne} e = R_n^{ne} e' \in ne$$

Then, the definition of a syntactic type T that realizes a PER model A is installed, denoted by $\underline{T} \subseteq A \subseteq \overline{T}$. As a consequence, the equivalence relation is given by the conjunction of the following:

$$\forall (e, e' : DNe).\quad e = e' \in \bot \rightarrow \uparrow^T e = \uparrow^T e' \in \underline{T}$$
$$\forall (a, a' : D).\quad a = a' \in \overline{T} \rightarrow \downarrow^T a = \downarrow^T a' \in \top$$

Finally, using the equivalence relation, one can directly prove the realizability for type T of the algorithm.

$$\forall T.\quad \bot \subseteq [\![T]\!] \subseteq \top$$

The main difficulty in proving realizability is in the modeling of equivalence relations in the function type. The process needs to establish deep-level equivalence starting with the high-level interpretation of the semantics, then into the domain values equivalence, and finally equivalence in the application evaluation. There are no obvious constructors to model such a deep-level equivalence, hence one way of encoding is by using nested inductive definitions.

5 Mechanization

The biggest challenges in creating a mechanization of NbE for the simple types in Coq lay in formalizing the concept, encoding the algorithm, and proving properties such as correctness, soundness, and completeness. To formalize definitions and rules, one needs to fully understand the theory on paper and to adequately apply Coq constructors. The mechanization of the PER can pose problems since Coq is primarily designed to work with total functions, which is a limitation in the expressiveness of the language and thus requires some workaround and compromise. Such a problem can be relatively easy to deal with in simple types but will become significant in dependent types. One solution is to use the option type to handle partiality, however, for the sake of readability, this approach is shelved and PER is modeled using inductive constructors and lemmas.

Moreover, during theorem proving, Coq shows to not be smart in its scope. The proof assistant has a strict termination checking in proofs, thus tactics must be applied on structurally decreasing arguments. On the good side, it guarantees consistency, but it also limits expressiveness. In many cases of this mechanization, Coq does not allow a theoretically correct tactic because of the scope problem when the tactic is called, hence one must rearrange the order of tactics.

6 Conclusion and Future Work

Normalization by evaluation for the simple types is a powerful normalization technique that provides a consistent normal form for all typable terms in the system of simple types. However, this mechanization is incomplete, thus for the sake of the project's completeness, proof of completeness and soundness properties should be mechanized as future work. On the other hand, considering its usefulness, the NbE for the simple types lays the foundations for countless possible extensions to more complex systems or to systems that are more suitable for real-life applications as explained in section 1.1. As a reasonable example of a future project, since real-life machines usually assign a type to a term instead of having a fixed type, by adding the feature where the type of an expression can depend on other terms, simple types will be extended into dependent types, which can be further mechanized. Such a typing system is particularly useful in, for instance, software development as it can provide concurrency, parallelism, and overall more expressive data structures.

References

[AA10] Brigitte Pientka Andreas Abel. Explicit substitutions for contextual type theory. *EPTCS 34*, pages 5–20, 2010.

[Abe09] Andreas Abel. Typed applicative structures and normalization by evaluation for system f. *Computer Science Logic, 23rd international Workshop, CSL 2009, 18th Annual Conference of the EACSL, Coimbra, Portugal, September 7-11, 2009. Proceedings,*, pages 40–54, 2009.

[Abe13] Andreas Abel. Normalization by evaluation dependent types and impredicativity. *Fakultät für Mathematik, Informatik, und Statistik, Ludwig-Maximilians-Universität München*, 2013.

[ea91a] M Abadi et al. Explicit substitutions. *Journal of Functional Programming*, 1:375–416, 1991.

[ea91b] Ulrich Berger et al. An inverse of the evaluation functional for typed lambda -calculus. *Proceedings Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 203–211, 1991.

[ea95] Thorsten Altenkirch et al. Categorical reconstruction of a reduction free normalization proof*. *International Conference on Category Theory and Computer Science CTCS 1995*, pages 182–199, 1995.

[ea98] Djordje Cubric et al. Normalization and the yoneda embedding. *Mathematical Structures in Computer Science, Volume 8, Issue 2*, pages 153–192, 1998.

[ea07] Andreas Abel et al. Normalization by evaluation for martin-löf type theory with typed equality judgements. *22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007)*, pages 3–12, 2007.

[ea16] Thorsten Altenkirch et al. Normalisation by evaluation for dependent types*. *1st International Conference on Formal Structures for Computation and Deduction (FSCD 2016)*, pages 6:1–6:16, 2016.

[ea18] Pawel Wieczorek et al. A coq formalization of normalization by evaluation for martin-löf type theory. *Proceedings of the 7th ACM*, pages 266–279, 2018.

[God58] K. Godel. Uber eine bisher noch nicht benutzte erweiterung des niten standpunktes. *Dialectica*, pages 280–287, 1958.

[ML75] Per Martin-Lof. An intuitionistic theory of types: Predicative part. *Studies in Logic and the Foundations of Mathematics, Volume 80*, pages 73–118, 1975.