Big Step Normalisation for Type Theory

Thorsten Altenkirch
School for Computer Science, University of Nottingham, UK
txa@cs.nott.ac.uk

Colin Geniet
Computer Science Department, ENS Paris-Saclay, France
colin.geniet@ens-paris-saclay.fr

Abstract

Big step normalisation is a normalisation method for typed lambda-calculi which relies on a purely syntactic recursive evaluator. Termination of that evaluator is proven using a predicate called strong computability, similar to the techniques used to prove strong normalisation of β-reduction for typed lambda-calculi. We generalise big step normalisation to a minimalist dependent type theory. Compared to previous presentations of big step normalisation for e.g. the simply-typed lambda-calculus, we use a quotiented syntax of type theory, which crucially reduces the syntactic complexity introduced by dependent types. Most of the proof has been formalised using Agda.

2012 ACM Subject Classification Theory of computation → Type theory

Keywords and phrases Normalisation, big step normalisation, type theory, dependent types, Agda

Digital Object Identifier 10.4230/LIPIcs.TYPES.2019.4

Supplementary Material `https://github.com/colingeniet/big-step-normalisation`

Funding *Thorsten Altenkirch*: supported by COST Action EUTypes CA15123 and USAF, Airforce office for scientific research, award FA9550-16-1-0029.

1 Introduction

1.1 Normalisation

In the context of typed lambda-calculi, normalisation refers to the process of computing a canonical representative, called normal form, in each $\beta\eta$-equivalence class of terms. A very general definition of normalisation, previously used in e.g. [6, 3, 7], is the following. Normalisation is given by a set of normal forms and two (computable) maps: norm from terms to normal forms, and an embedding $\ulcorner \_ \urcorner$ of normal forms into terms, satisfying

soundness If $u \simeq_{\beta\eta} v$, then $\mathsf{norm}\ u = \mathsf{norm}\ v$
completeness[1] For every term u, $\ulcorner \mathsf{norm}\ u \urcorner \simeq_{\beta\eta} u$
stability For every normal form n, $\mathsf{norm}\ \ulcorner n \urcorner = n$

The traditional way to define a normalisation function is through rewriting theory. One proves that $\beta\eta$-reduction is confluent, and terminates[2] on typed terms. Normal forms are defined as terms which can not be $\beta\eta$-reduced, and normalisation is done by reducing a term until reaching a normal form. Termination and confluence ensure the correctness of the definition. Soundness also follows from confluence, while completeness and stability are immediate. See for instance [16] for a detailed proof of this result for the simply-typed lambda-calculus and some variants (System F, System T). Unfortunately, problems arise for

[1] The choice of the words *soundness* and *completeness* comes from viewing normal forms as a model.
[2] We do not use the words strong or weak normalisation to refer to termination of the rewriting process, so as to avoid ambiguity with the generalised notion of normalisation introduced.

some other variants of the lambda-calculus. For instance, the lambda-calculus with explicit substitutions does not terminate in the strong sense [22], and the lambda-calculus with coproduct types (i.e. disjoint union) is not confluent [14]. While some of these issues can be worked around, for instance by using weak termination and more restrictive reductions, these problems have lead to the development of other methods.

One of them is *normalisation by evaluation* (NBE), introduced by Berger and Schwichtenberg [9] for the simply-typed lambda-calculus. The idea is to evaluate terms into a semantic model, meaning for instance that λ-abstractions (syntactic functions) are interpreted by actual (semantic) functions. A map from the model into normal forms is then defined, giving rise to the normalisation function by composition with evaluation. This method was for instance used to prove decidability of equivalence for the lambda-calculus with coproducts [4].

1.2 Big Step Normalisation

Big step normalisation (BSN) is a purely syntactic normalisation method, proposed in [3] by Chapman and the first author for the simply-typed lambda-calculus. The normalisation algorithm is in two parts. First, terms are evaluated by an environment machine, yielding syntactic values. Then, values are mapped to normal forms by a function named quote. Normalisation norm is done by evaluating in the identity environment, then applying quote on the resulting value. The embedding $\ulcorner \_ \urcorner$ is the inclusion of normal forms into terms.

Evaluation and quote both have fairly simple definitions, but are not structurally recursive, hence their termination is not obvious. To prove termination, a Tait-style predicate [25] called *strong computability* (SC) is defined on values:

- A value v of the base type is strongly computable if normalisation terminates on v.
- A value f of a function type is SC if it preserves SC when applied to an argument.

The following results can then be proved.

- quote terminates on any SC value, and conversely any neutral value (i.e. a value which is not a λ-abstraction) on which quote terminates is SC.
- In a SC environment, evaluation terminates and yields SC results.

Termination of norm follows from these results. Completeness and stability are straightforward. The proof of soundness is more involved, and shares some similarities with the proof of termination, but replaces strong computability with a binary relation on values.

1.3 BSN for Type Theory and Quotiented Syntax

Chapman also considered BSN for dependent type theory in [10], but did not provide a full proof of correctness, due to the syntactic complexity added by dependent types.

In this work, we propose some methods to simplify the proof of BSN in the case of dependent types, allowing us to complete it. Notably, we use the quotiented syntax of type theory proposed in [8]. By only considering terms quotiented by $\beta\eta$-equivalence, the syntax becomes significantly lighter. For instance, the coercion constructors which form a large part of the syntactic boilerplate encountered in [10] become unnecessary.

With a quotiented syntax, the notion of normalisation changes slightly. If $\simeq_{\beta\eta}$ is replaced with equality of quotiented terms in the first definition of a normalisation function, then soundness simply states that norm is correctly defined on the quotiented syntax, while completeness and stability state that norm and $\ulcorner \_ \urcorner$ are inverse of each other. This leads to the following definition proposed in [7]: a normalisation function is simply an isomorphism between quotiented terms and normal forms. Obviously, this definition requires a sensible

notion of normal forms – one can not consider quotiented terms to be normal forms, and identity to be normalisation. Thus, we require normal forms to have a simple inductive definition, which ensures decidability of equality.[3]

1.4 Structure of the Paper

Section 2 presents the metatheory, notation, and conventions used in this paper. Section 3 presents the quotiented syntax of type theory. Section 4 introduces a notion of weakening of contexts. Section 5 defines big step normalisation itself. Because it is not a priori clear that BSN defines a correct function (termination for instance is problematic) we formally define normalisation by its big step semantics, i.e. as a relation between inputs and output. Section 6 focuses on the two major correctness proofs: termination and soundness. The proof of termination remains similar to the case of simple types. The main difference is that we develop a simplified and generalised induction principle for types, which allows us to manipulate dependent types in almost the same way as simple types during the proof. The proof of soundness for an unquotiented syntax seems much harder to adapt, we instead provide a simple proof using soundness of NBE. Finally, Section 7 explains how the proof of BSN can be adapted to a cubical metatheory, using higher inductive types to encode quotient inductive types.

1.5 Related Work

Big step semantics have previously been used for the purpose of normalisation. For instance T. Coquand uses a big step relation to decide conversion in type theory [12], but relies on considerations on untyped terms, and focuses on deciding conversion, rather than fully normalising terms. P.B. Levy uses Tait's method to prove termination of a big step semantics in the case of a simple programming language [21]. Big step normalisation was developed by Chapman and the first author for a combinatory calculus [2], and for the simply-typed lambda-calculus [3]. A generalisation to type theory was proposed [10], but without a full proof of correctness. The present paper can be seen as a continuation of these works.

An important difference compared to previous works on big step normalisation is that we use a quotiented syntax of type theory. This builds upon the work by Kaposi and the first author which provides a concise, quotiented syntax of type theory within (a larger) type theory [8], and formalises normalisation by evaluation in this syntax [7]. This quotiented syntax is closely related to categories with families [15, 18], in that the syntax is essentially an initial category with families. The syntax is formalised using quotient inductive-inductive types (QIIT), which were previously used in [24] – although not under that name – to e.g. define Cauchy reals in type theory. More recently, the precise notion and semantics of QIIT has been the subject of work such as [1, 13, 20].

2 Metatheory and Notations

The present work has been formalised using a cubical metatheory [11] implemented by Agda [23]. This cubical theory provides a simple way to define quotient inductive inductive types (QIIT, cf. [8]) as a special case of higher inductive types. However, for simplicy, we

[3] In the unquotiented case, the embedding of normal forms into terms (which can be proved to be injective) ensures that equality of normal forms is decidable, hence why no such restriction was required.

prefer to present this paper in a strict, intentional Martin-Löf Type Theory, extended with QIIT. Functional extensionality is assumed, and can in fact be proved using the interval quotient type. See Section 7 for a discussion of the implementation in a cubical metatheory.

Our metatheoretic notations are loosely based on the syntax of Agda. Function types are written as $(x : A) \to B$, or simply $A \to B$ for non-dependent functions. We use infix arguments denoted by underscores, e.g. $\_,\_$ applied to x and y is written as x, y. Functions with implicit arguments are defined as $f : \{x : A\} \to B$, and the argument can be either omitted, or given in subscript as f_x. Sum types (dependent pairs) are denoted by $\Sigma(x : A),\ B$. We denote by Set the universe of types, and by Prop the universe of mere propositions, i.e. the types in which all elements are equal. The equality type is denoted by $x \equiv y$, while $=$ is only used in definitions. The transport of $x : P\ a$ along an equality $p : a \equiv b$ is denoted by ${}_{p*}x : P\ b$. If $p : a \equiv b$, the type of dependent equalities between $x : P\ a$ and $y : P\ b$ lying over p is denoted by $x \equiv^p y$. For simplicity and readability, transports and dependent equality types will be omitted starting from Section 4.

Inductive types are introduced by **data**, the sort of the defined type, and the signatures of the constructors. Inductive functions are defined by pattern-matching. For instance:

$$\begin{array}{ll}\textbf{data}\ \mathbb{N} : \mathsf{Set}\ \textbf{where} & \_+\_ : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\ \quad 0 : \mathbb{N} & n + 0 \quad\ = n \\ \quad S : \mathbb{N} \to \mathbb{N} & n + (S\ m) = S\ (n + m)\end{array}$$

We allow a very general form of mutual induction, called *inductive-inductive* definitions. A good example is the following fragment of the syntax of dependent types from next section.

$$\begin{array}{ll}\textbf{data}\ \mathsf{Con} : \mathsf{Set}\ \textbf{where} & \textbf{data}\ \mathsf{Ty} : \mathsf{Con} \to \mathsf{Set}\ \textbf{where} \\ \quad \bullet \quad : \mathsf{Con} & \quad \mathsf{U} : \mathsf{Ty}\ \Gamma \\ \quad \_,\_ : (\Gamma : \mathsf{Con}) \to \mathsf{Ty}\ \Gamma \to \mathsf{Con} & \quad \Pi : (A : \mathsf{Ty}\ \Gamma) \to \mathsf{Ty}\ (\Gamma, A) \to \mathsf{Ty}\ \Gamma\end{array}$$

In addition to Con and Ty being defined simultaneously, note that Ty is a family indexed by Con, and the signature of the constructor Π form Ty uses the constructor $\_,\_$ from Con.

QIIT furthermore allow *equality* or *quotient* constructors, which build of equalities in the defined type. For example the interval type is defined by two endpoints and an equality:

$$\begin{array}{l}\textbf{data}\ \mathbb{I} : \mathsf{Set}\ \textbf{where} \\ \quad a : \mathbb{I} \\ \quad b : \mathbb{I} \\ \quad p : a \equiv b\end{array}$$

A function defined by induction on a QIIT must be defined inductively on regular constructors, and must respect all quotient constructors, meaning that it must map the elements equated by a quotient constructors to images which are provably equal. For instance, to define a function by induction on $\mathbb{I}$, one must specify the images $f(a)$ and $f(b)$, then prove that $f(a) \equiv f(b)$. The reader may refer to [8] for more details on QIIT.

Finally, all free variables in definitions and lemmas are implicitly universally quantified. Omitted types can be inferred from the context and the naming conventions.

3 Quotiented Syntax of Type Theory

This section introduces the syntax of type theory based on QIIT proposed by Kaposi and the first author in [8, 7]. The reader should refer to the former for further details.

This syntax is intrinsically typed, with De Bruijn indices, and explicit substitutions. Contexts, types, substitutions and terms are mutually defined. We denote contexts by $\Gamma, \Delta, \Theta, \Phi$, types by A, B, C, substitutions by σ, ν, δ, and terms by s, t, u.

data $\mathsf{Con} : \mathsf{Set}$	Con is the set of contexts
data $\mathsf{Ty} : \mathsf{Con} \to \mathsf{Set}$	$\mathsf{Ty}\ \Gamma$ are the types in context Γ
data $\mathsf{Sub} : \mathsf{Con} \to \mathsf{Con} \to \mathsf{Set}$	$\mathsf{Sub}\ \Gamma\ \Delta$ are the substitutions from Δ to Γ
data $\mathsf{Tm} : (\Gamma : \mathsf{Con}) \to \mathsf{Ty}\ \Gamma \to \mathsf{Set}$	$\mathsf{Tm}\ \Gamma\ A$ are the terms of type A in Γ

Syntax constructors follow closely the definition of a category with families [15, 18] with product types. Contexts and substitutions form a category, types are a presheaf, and terms are a family of presheaves over types. The constructor for dependent function types is denoted by Π. There is a base type U, and a base dependent family El indexed by U. One may see U as a universe, i.e. a type whose elements are types, when interpreted through El – this is reflected by the names of the constructors. Because we consider a minimalist type theory, it is only an abstract universe, meaning that no element of U can be built in a closed context. However, one may use contexts to postulate the existence of types in U.

The syntax constructors are listed below, with regular constructors on the left, and equality constructors on the right.

data Con **where**

$\bullet \quad : \mathsf{Con}$

$\_,\_ : (\Gamma : \mathsf{Con}) \to \mathsf{Ty}\ \Gamma \to \mathsf{Con}$

data Ty **where**

$\_[\_] : \mathsf{Ty}\ \Delta \to \mathsf{Sub}\ \Gamma\ \Delta \to \mathsf{Ty}\ \Gamma$

$\mathsf{U} \quad : \mathsf{Ty}\ \Gamma$

$\mathsf{El} \quad : \mathsf{Tm}\ \Gamma\ \mathsf{U} \to \mathsf{Ty}\ \Gamma$

$\Pi \quad : (A : \mathsf{Ty}\ \Gamma) \to \mathsf{Ty}\ (\Gamma, A) \to \mathsf{Ty}\ \Gamma$

data Ty **where**

$[\mathsf{id}] : A[\mathsf{id}] \equiv A$

$[\circ] \ : A[\sigma \circ \nu] \equiv A[\sigma][\nu]$

$\mathsf{U}[] : \mathsf{U}[\sigma] \equiv \mathsf{U}$

$\mathsf{El}[] : (\mathsf{El}\ u)[\sigma] \equiv \mathsf{El}({}_{\mathsf{U}[]*}u[\sigma])$

$\Pi[] : (\Pi\ A\ B)[\sigma] \equiv \Pi(A[\sigma])(B[\sigma{\uparrow}A])$

data Sub **where**

$\mathsf{id} \quad : \mathsf{Sub}\ \Gamma\ \Gamma$

$\_ \circ \_ : \mathsf{Sub}\ \Delta\ \Theta \to \mathsf{Sub}\ \Gamma\ \Delta \ \to \mathsf{Sub}\ \Gamma\ \Theta$

$\epsilon \quad : \mathsf{Sub}\ \Gamma\ \bullet$

$\_,\_ \quad : (\sigma : \mathsf{Sub}\ \Gamma\ \Delta) \to \mathsf{Tm}\ \Gamma\ A[\sigma]$
$\qquad \to \mathsf{Sub}\ \Gamma\ (\Delta, A)$

$\pi_1 \quad : \mathsf{Sub}\ \Gamma\ (\Delta, A) \to \mathsf{Sub}\ \Gamma\ \Delta$

data Sub **where**

$\mathsf{id}\circ \ : \mathsf{id} \circ \sigma \equiv \sigma$

$\circ\,\mathsf{id} : \sigma \circ \mathsf{id} \equiv \sigma$

$\circ\circ \ : (\sigma \circ \nu) \circ \delta \equiv \sigma \circ (\nu \circ \delta)$

$\epsilon\eta \quad : \{\sigma : \mathsf{Sub}\ \Gamma\ \bullet\} \to \sigma \equiv \epsilon$

$\pi_1\beta : \pi_1(\sigma, u) \equiv \sigma$

$\pi\eta \quad : \pi_1\,\sigma, \pi_2\,\sigma \equiv \sigma$

$,\circ \quad : (\sigma, u) \circ \nu \equiv (\sigma \circ \nu), ({}_{[\circ]^{-1}*}u[\nu])$

data Tm **where**

$\pi_2 \quad : (\sigma : \mathsf{Sub}\ \Gamma\ (\Delta, A)) \to \mathsf{Tm}\ \Gamma\ (A[\pi_1\sigma])$

$\_[\_] : \mathsf{Tm}\ \Delta\ A \to (\sigma : \mathsf{Sub}\ \Gamma\ \Delta)$
$\qquad \to \mathsf{Tm}\ \Gamma\ A[\sigma]$

$\lambda \quad : \mathsf{Tm}\ (\Gamma, A)\ B \to \mathsf{Tm}\ \Gamma\ (\Pi\ A\ B)$

$\mathsf{app} \ : \mathsf{Tm}\ \Gamma\ (\Pi\ A\ B) \to \mathsf{Tm}\ (\Gamma, A)\ B$

data Tm **where**

$\pi_2\beta : \pi_2(\sigma, u) \equiv^{\pi_1\beta} u$

$\beta \quad : \mathsf{app}\ (\lambda u) \equiv u$

$\eta \quad : \lambda(\mathsf{app}\,u) \equiv u$

$\lambda[] \ : (\lambda u)[\sigma] \equiv^{\Pi[]} \lambda(u[\sigma{\uparrow}A])$

Equations $\Pi[]$ and $\lambda[]$ use the lifting of a substitution by a type, defined as follows.

$$\begin{aligned}
&_\uparrow_ : (\sigma : \mathsf{Sub}\ \Gamma\ \Delta) \to (A : \mathsf{Ty}\ \Delta) \to \mathsf{Sub}\ (\Gamma, A[\sigma])\ (\Delta, A)\\
&\sigma\uparrow A = (\sigma \circ \pi_1\ \mathsf{id}), ({}_{[\circ]^{-1}}{*}\pi_2\ \mathsf{id})
\end{aligned}$$

This syntax uses a categorical application constructor app, which is essentially the inverse of λ. One may understand $\mathsf{app}\ f$ as the application of f to a fresh variable. In order to obtain the usual application, denoted $\_\$\_$, this fresh variable must be substituted by the argument. We denote this substitution of the last variable in the context by $< \_ >$.

$$\begin{aligned}
&< _ > : \mathsf{Tm}\ \Gamma\ A \to \mathsf{Sub}\ \Gamma\ (\Gamma, A)\\
&< u > \ = \mathsf{id}\ ,\ {}_{[\mathsf{id}]^{-1}}{*}u\\
&_\$_ \quad : \mathsf{Tm}\ \Gamma\ (\Pi\ A\ B) \to (u : \mathsf{Tm}\ \Gamma\ A) \to \mathsf{Tm}\ \Gamma\ (B[< u >])\\
&f\ \$\ u \quad = (\mathsf{app}\ f)[< u >]
\end{aligned}$$

As a simple example, let us translate the lambda term $\lambda x^A.\lambda y^B.x$ to this syntax. We assume that A is type in context Γ, and B a type in context (Γ, A) (or in short $\Gamma : \mathsf{Con}$, $A : \mathsf{Ty}\ \Gamma$, $B : \mathsf{Ty}\ (\Gamma, A)$), so that (Γ, A, B) is a context. We start with id, intuitively the substitution containing all variables in context.

$$\mathsf{id} : \mathsf{Sub}\ (\Gamma, A, B)\ (\Gamma, A, B)$$

The second to last variable in the context, corresponding to x, is retrieved through projections.

$$\pi_2(\pi_1\ \mathsf{id}) : \mathsf{Tm}\ (\Gamma, A, B)\ A[\pi_1\ \mathsf{id}]$$

Finally, the lambda-abstractions are added.

$$\lambda(\lambda(\pi_2(\pi_1\ \mathsf{id}))) : \mathsf{Tm}\ \Gamma\ (\Pi\ A\ (\Pi\ B\ A[\pi_1\ \mathsf{id}]))$$

4 Weakenings

In this section, we introduce variables and weakenings of contexts. The presentation is the same as in [7], except that the latter uses the name "renamings" instead.

Variables, denoted by x, y, z, are defined as typed De Bruijn indices, with constructors vz and vs standing for "0" and successor. Variables can be embedded into terms by applying projections to id – intuitively, id is the substitution formed by all the variables in context.

$$\begin{aligned}
&\textbf{data}\ \mathsf{Var}\ : (\Gamma : \mathsf{Con}) \to \mathsf{Ty}\ \Gamma \to \mathsf{Set}\ \textbf{where} && \ulcorner_\urcorner : \mathsf{Var}\ \Gamma\ A \to \mathsf{Tm}\ \Gamma\ A\\
&\quad \mathsf{vz} : \mathsf{Var}\ (\Gamma, A)\ (A[\pi_1\ \mathsf{id}]) && \ulcorner\mathsf{vz}\urcorner \ = \pi_2\ \mathsf{id}\\
&\quad \mathsf{vs} : \mathsf{Var}\ \Gamma\ A \to \mathsf{Var}\ (\Gamma, B)\ (A[\pi_1\ \mathsf{id}]) && \ulcorner\mathsf{vs}\,x\urcorner = \ulcorner x\urcorner[\pi_1\ \mathsf{id}]
\end{aligned}$$

Weakening substitutions (or simply weakenings), denoted by α, β, γ, are substitutions composed only of variables. This regroups the usual notions of weakening (i.e. forgetting a variable), contraction, and reordering of independent variables. Note that constructors ϵ and $\_,\_$ are overloaded due to the similarity with substitutions.

$$\begin{aligned}
&\textbf{data}\ \mathsf{Wk}\ : \mathsf{Con} \to \mathsf{Con} \to \mathsf{Set}\ \textbf{where} && \ulcorner_\urcorner : \mathsf{Wk}\ \Gamma\ \Delta \to \mathsf{Sub}\ \Gamma\ \Delta\\
&\quad \epsilon \quad : \mathsf{Wk}\ \Gamma\ \bullet && \ulcorner\epsilon\urcorner \quad = \epsilon\\
&\quad _,_ : (\alpha : \mathsf{Wk}\ \Gamma\ \Delta) \to \mathsf{Var}\ \Gamma\ A[\ulcorner\alpha\urcorner] \to \mathsf{Wk}\ \Gamma\ (\Delta, A) && \ulcorner\alpha, x\urcorner = \ulcorner\alpha\urcorner, \ulcorner x\urcorner
\end{aligned}$$

Unlike regular substitutions, identity and composition of weakenings are not constructors, but inductive definitions. Some auxiliary functions are required: wk weakens the context of a weakening substitution by a type A, and $\_[\_]$ applies a weakening substitution to a variable. These functions all commute with embeddings of variables and weakenings. We omit the inductive definitions and proofs, which are simple.

$$
\begin{array}{ll}
\mathsf{wk} & : (A : \mathsf{Ty}\ \Gamma) \to \mathsf{Wk}\ \Gamma\ \Delta \to \mathsf{Wk}\ (\Gamma, A)\ \Delta \\
\mathsf{id} & : \{\Gamma : \mathsf{Con}\} \to \mathsf{Wk}\ \Gamma\ \Gamma \\
\_[\_] & : \mathsf{Var}\ \Delta\ A \to (\alpha : \mathsf{Wk}\ \Gamma\ \Delta) \to \mathsf{Var}\ \Gamma\ (A[\ulcorner\alpha\urcorner]) \\
\_ \circ \_ & : \mathsf{Wk}\ \Delta\ \Theta \to \mathsf{Wk}\ \Gamma\ \Delta \to \mathsf{Wk}\ \Gamma\ \Theta
\end{array}
\qquad
\begin{array}{ll}
\ulcorner\mathsf{wk}\urcorner & : \ulcorner\mathsf{wk}\ A\ \alpha\urcorner \equiv \ulcorner\alpha\urcorner \circ (\pi_1\ \mathsf{id}) \\
\ulcorner\mathsf{id}\urcorner & : \ulcorner\mathsf{id}\urcorner \equiv \mathsf{id} \\
\ulcorner[]\urcorner & : \ulcorner x[\alpha]\urcorner \equiv \ulcorner x\urcorner[\ulcorner\alpha\urcorner] \\
\ulcorner\circ\urcorner & : \ulcorner\alpha \circ \beta\urcorner \equiv \ulcorner\alpha\urcorner \circ \ulcorner\beta\urcorner
\end{array}
$$

Contexts and weakenings form a category with these operations. Types, terms, and substitutions can be weakened by applying a weakening substitution, seen as a regular substitution through embedding. These operations respects identity and composition, that is types and substitutions are presheaves on the category of weakenings, while terms are a family of presheaves over types. Definitions are below, with the lemmas on the right (proofs omitted).

$$
\begin{array}{ll}
\_^{+\_} & : \mathsf{Ty}\ \Delta \to \mathsf{Wk}\ \Gamma\ \Delta \to \mathsf{Ty}\ \Gamma \\
A^{+\alpha} & = A[\ulcorner\alpha\urcorner] \\
\_^{+\_} & : \mathsf{Tm}\ \Delta\ A \to (\alpha : \mathsf{Wk}\ \Gamma\ \Delta) \to \mathsf{Tm}\ \Gamma\ A^{+\alpha} \\
u^{+\alpha} & = u[\ulcorner\alpha\urcorner] \\
\_^{+\_} & : \mathsf{Sub}\ \Delta\ \Theta \to \mathsf{Wk}\ \Gamma\ \Delta \to \mathsf{Sub}\ \Gamma\ \Theta \\
\sigma^{+\alpha} & = \sigma \circ \ulcorner\alpha\urcorner
\end{array}
\qquad
\begin{array}{ll}
+\mathsf{id} & : A^{+\mathsf{id}} \equiv A \\
+\circ & : A^{+(\alpha\circ\beta)} \equiv (A^{+\alpha})^{+\beta} \\
+\mathsf{id} & : u^{+\mathsf{id}} \equiv u \\
+\circ & : u^{+(\alpha\circ\beta)} \equiv (u^{+\alpha})^{+\beta} \\
+\mathsf{id} & : \sigma^{+\mathsf{id}} \equiv \sigma \\
+\circ & : \sigma^{+(\alpha\circ\beta)} \equiv (\sigma^{+\alpha})^{+\beta}
\end{array}
$$

This will be a general pattern in later constructions and proofs: families of sets (e.g. values, normal forms, ...) have a presheaf-like structure, which simply means that the elements can be weakened coherently. Similarly, functions are natural transformations, i.e. commute with weakening, and predicates are sub-presheaves, i.e. are stable under weakening. The corresponding definitions and proofs are typically straightforward, and we will often not mention them. We abusively denote all applications of weakenings by $\_^{+\_}$.

Finally, given a type A, one may consider $\mathsf{wk}\ A\ \mathsf{id} : \mathsf{Wk}\ (\Gamma, A)\ \Gamma$, the weakening of the context Γ by A. We abuse notations and write u^{+A} for $u^{+(\mathsf{wk}\ A\ \mathsf{id})}$.

5 Normalisation Relation

This section defines the big step normalisation algorithm using the previous syntax of type theory. As further explained in Section 5.2, this algorithm can not yet be formally defined as a function. Thus, it is defined as a relation in order to carry out the correctness proof.

We first define values and the evaluation from terms to values, then normal forms and the function quote mapping values to normal forms. Normalisation is done by applying evaluation followed by quote.

5.1 Values

A value is either a closure, corresponding to the delayed evaluation of a lambda-abstraction, or a neutral value, that is the stuck application of a variable to values. We define mutually values (denoted by v, w), neutral values (denoted by n), and environments (substitutions composed of values, denoted by ρ, ω), together with the associated embeddings.

$$
\begin{array}{ll}
\textbf{data}\ \mathsf{Val} : (\Gamma : \mathsf{Con}) \to \mathsf{Ty}\ \Gamma \to \mathsf{Set}\ \textbf{where} & \ulcorner \_ \urcorner : \mathsf{Val}\ \Gamma\ A \to \mathsf{Tm}\ \Gamma\ A \\
\quad \mathsf{neu} : \mathsf{NV}\ \Gamma\ A \to \mathsf{Val}\ \Gamma\ A & \ulcorner \mathsf{neu}\ n \urcorner \quad = \ulcorner n \urcorner \\
\quad \mathsf{clos} : \mathsf{Tm}\ (\Delta, A)\ B \to (\rho : \mathsf{Env}\ \Gamma\ \Delta) & \ulcorner \mathsf{clos}\ u\ \rho \urcorner = (\lambda u)[\ulcorner \rho \urcorner] \\
\qquad \to \mathsf{Val}\ \Gamma\ ((\Pi\ A\ B)[\ulcorner \rho \urcorner]) & \\
\textbf{data}\ \mathsf{NV} : (\Gamma : \mathsf{Con}) \to \mathsf{Ty}\ \Gamma \to \mathsf{Set}\ \textbf{where} & \ulcorner \_ \urcorner : \mathsf{NV}\ \Gamma\ A \to \mathsf{Tm}\ \Gamma\ A \\
\quad \mathsf{var}\ : \mathsf{Var}\ \Gamma\ A \to \mathsf{NV}\ \Gamma\ A & \ulcorner \mathsf{var}\ x \urcorner \quad = \ulcorner x \urcorner \\
\quad \mathsf{app} : \mathsf{NV}\ \Gamma\ (\Pi\ A\ B) \to (v : \mathsf{Val}\ \Gamma\ A) & \ulcorner \mathsf{app}\ n\ v \urcorner = \ulcorner n \urcorner\, \$\, \ulcorner v \urcorner \\
\qquad \to \mathsf{NV}\ \Gamma\ (B[< \ulcorner v \urcorner >]) & \\
\textbf{data}\ \mathsf{Env} : \mathsf{Con} \to \mathsf{Con} \to \mathsf{Set}\ \textbf{where} & \ulcorner \_ \urcorner : \mathsf{Env}\ \Gamma\ \Delta \to \mathsf{Sub}\ \Gamma\ \Delta \\
\quad \epsilon \quad : \mathsf{Env}\ \Gamma\ \bullet & \ulcorner \epsilon \urcorner \quad = \epsilon \\
\quad \_,\_ : (\rho : \mathsf{Env}\ \Gamma\ \Delta) \to \mathsf{Val}\ \Gamma\ (A[\ulcorner \rho \urcorner]) \to \mathsf{Env}\ \Gamma\ (\Delta, A) & \ulcorner \rho, v \urcorner = \ulcorner \rho \urcorner, \ulcorner v \urcorner
\end{array}
$$

This definition has an issue when used with a quotiented syntax: values can be equivalent as terms (formally, have equal embeddings), but not equal. For instance, in a closure $\mathsf{clos}\ u\ \rho$, if the body u never refers to the environment ρ, then modifying ρ yields a distinct but equivalent value. Then, evaluation would map equivalent terms to distinct values, hence could not be defined on the quotiented syntax. This is fixed by forcing equivalent values to be equal with the following quotient constructor.

$$
\begin{array}{l}
\textbf{data}\ \mathsf{Val}\ \textbf{where} \\
\quad \mathsf{qVal} : (v\ w : \mathsf{Val}\ \Gamma\ A) \to \ulcorner v \urcorner \equiv \ulcorner w \urcorner \to v \equiv w
\end{array}
$$

The corresponding result for environments can be proved by induction on contexts.

$$\mathsf{qEnv} : (\rho\ \omega : \mathsf{Env}\ \Gamma\ \Delta) \to \ulcorner \rho \urcorner \equiv \ulcorner \omega \urcorner \to \rho \equiv \omega$$

Weakening is defined by induction on values, neutral values, and environments, we omit the definitions and the associated lemmas. Finally, the identity environment is defined by induction on the context, and uses weakening of environments.

$$
\begin{array}{l}
\mathsf{idenv} : \{\Gamma : \mathsf{Con}\} \to \mathsf{Env}\ \Gamma\ \Gamma \\
\mathsf{idenv}_{\bullet} \quad = \epsilon \\
\mathsf{idenv}_{\Gamma, A} = \mathsf{idenv}_{\Gamma}{}^{+A}\,,\, \mathsf{neu}\,(\mathsf{var}\ \mathsf{vz})
\end{array}
$$

5.2 Evaluation

The first stage of normalisation is an environment machine, which evaluate terms in an environment, and returns values. It consists of three mutually defined functions: eval and evals evaluate terms and substitutions respectively in an environment, while _@_ computes the application of a value to another.

$$
\begin{array}{ll}
\mathsf{eval} : \mathsf{Tm}\ \Delta\ A \to (\rho : \mathsf{Env}\ \Gamma\ \Delta) \to \mathsf{Val}\ \Gamma\ A[\ulcorner \rho \urcorner] & \\
\mathsf{eval}\ (\pi_2\, \sigma)\ \rho & = \textbf{let}\ (\omega, v) = (\mathsf{evals}\ \sigma\ \rho)\ \textbf{in}\ v \\
\mathsf{eval}\ (u[\sigma])\ \rho & = \mathsf{eval}\ u\ (\mathsf{evals}\ \sigma\ \rho) \\
\mathsf{eval}\ (\lambda u)\ \rho & = \mathsf{clos}\ u\ \rho \\
\mathsf{eval}\ (\mathsf{app}\ u)\ (\rho, v) & = (\mathsf{eval}\ u\ \rho)\ @\ v
\end{array}
$$

$$
\begin{aligned}
&\mathsf{evals} : \mathsf{Sub}\ \Delta\ \Theta \to \mathsf{Env}\ \Gamma\ \Delta \to \mathsf{Env}\ \Gamma\ \Theta\\
&\mathsf{evals}\ \mathsf{id}\ \rho &&= \rho\\
&\mathsf{evals}\ (\sigma \circ \nu)\ \rho &&= \mathsf{evals}\ \sigma\ (\mathsf{evals}\ \nu\ \rho)\\
&\mathsf{evals}\ \epsilon\ \rho &&= \epsilon\\
&\mathsf{evals}\ (\sigma, u)\ \rho &&= (\mathsf{evals}\ \sigma\ \rho), (\mathsf{eval}\ u\ \rho)\\
&\mathsf{evals}\ (\pi_1\ \sigma)\ \rho &&= \mathbf{let}\ (\omega, v) = (\mathsf{evals}\ \sigma\ \rho)\ \mathbf{in}\ \omega\\
&\_@\_ : \mathsf{Val}\ \Gamma\ (\Pi\ A\ B) \to (v : \mathsf{Val}\ \Gamma\ A) \to \mathsf{Val}\ \Gamma\ B[< \ulcorner v \urcorner >]\\
&(\mathsf{clos}\ u\ \rho)\ @\ v &&= \mathsf{eval}\ u\ (\rho, v)\\
&(\mathsf{neu}\ n)\ @\ v &&= \mathsf{neu}\ (\mathsf{app}\ n\ v)
\end{aligned}
$$

Most cases are straightforward. Note how evaluation of a lambda simply returns a closure, delaying the evaluation of the body. The latter occurs in the first case of _@_, as the application of a closure to a value is computed by evaluating the body of the closure in the extended environment. Evaluation of the projections π_1, π_2 performs a projection on an environment, expressed through the **let** ... **in** construct with an obvious meaning.

However, there are several problems with this presentation of the evaluator. Firstly, the functions are defined by recursion on terms and substitutions, which are QIIT, but we did not bother to verify that equality constructors are respected. Perhaps more worryingly, the function is not structurally recursive: the last case of eval applies _@_ to eval $u\ p$, which a priori is an arbitrary value. Thus it is not clear that the evaluator terminates.

The proof of correctness of this algorithm is not trivial, and is the subject of Section 6. For now, we will only define the algorithm, i.e. we consider the previous definition as a *programming* function, rather than an (incorrect) mathematical function. In order to formally define this algorithm, we represent it by its big step semantics, that is the relation between inputs and outputs of the evaluator. For instance, we denote by $\mathsf{eval}\ t\ \rho \Downarrow v$ the proposition "t evaluates to v in environment ρ".

$$
\begin{aligned}
&\mathbf{data}\ \mathsf{eval}\_\_ \Downarrow \_ : \mathsf{Tm}\ \Delta\ A \to \mathsf{Env}\ \Gamma\ \Delta \to \mathsf{Val}\ \Gamma\ B \to \mathsf{Prop}\ \mathbf{where}\\
&\quad \mathsf{eval}\pi_2 \ : \mathsf{evals}\ \sigma\ \rho \Downarrow (\omega, v) \to \mathsf{eval}\ (\pi_2\ \sigma)\ \rho \Downarrow v\\
&\quad \mathsf{eval}[] \ : \mathsf{evals}\ \sigma\ \rho \Downarrow \omega \to \mathsf{eval}\ u\ \omega \Downarrow v \to \mathsf{eval}\ (u[\sigma])\ \rho \Downarrow v\\
&\quad \mathsf{eval}\lambda \ : \mathsf{eval}\ (\lambda u)\ \rho \Downarrow (\mathsf{clos}\ u\ \rho)\\
&\quad \mathsf{evalapp} : \mathsf{eval}\ f\ \rho \Downarrow g \to g\ @\ v \Downarrow w \to \mathsf{eval}\ (\mathsf{app}\ f)\ (\rho, v) \Downarrow w\\
&\mathbf{data}\ \mathsf{evals}\_\_ \Downarrow \_ : \mathsf{Sub}\ \Delta\ \Theta \to \mathsf{Env}\ \Gamma\ \Delta \to \mathsf{Env}\ \Gamma\ \Theta \to \mathsf{Prop}\ \mathbf{where}\\
&\quad \mathsf{evalsid} \ : \mathsf{evals}\ \mathsf{id}\ \rho \Downarrow \rho\\
&\quad \mathsf{evals}\circ \ : \mathsf{evals}\ \nu\ \rho \Downarrow \omega \to \mathsf{evals}\ \sigma\ \omega \Downarrow \xi \to \mathsf{evals}\ (\sigma \circ \nu)\ \rho \Downarrow \xi\\
&\quad \mathsf{evals}\epsilon \ : \mathsf{evals}\ \epsilon\ \rho \Downarrow \epsilon\\
&\quad \mathsf{evals}, \ : \mathsf{evals}\ \sigma\ \rho \Downarrow \omega \to \mathsf{eval}\ u\ \rho \Downarrow v \to \mathsf{evals}\ (\sigma, u)\ \rho \Downarrow (\omega, v)\\
&\quad \mathsf{evals}\pi_1 : \mathsf{evals}\ \sigma\ \rho \Downarrow (\omega, v) \to \mathsf{eval}\ (\pi_1\ \sigma)\ \rho \Downarrow \omega\\
&\mathbf{data}\ \_@\_ \Downarrow \_ : \mathsf{Val}\ \Gamma\ A \to \mathsf{Val}\ \Gamma\ B \to \mathsf{Val}\ \Gamma\ C \to \mathsf{Prop}\ \mathbf{where}\\
&\quad @\mathsf{clos} : \mathsf{eval}\ u\ (\rho, v) \Downarrow w \to (\mathsf{clos}\ u\ \rho)\ @\ v \Downarrow w\\
&\quad @\mathsf{neu} \ : (\mathsf{neu}\ n)\ @\ v \Downarrow (\mathsf{neu}\ (\mathsf{app}\ n\ v))
\end{aligned}
$$

The types of the above relations may seem surprisingly imprecise. For instance, the type of eval does not give any information on the type of the return value – it is a value of some unknown type B – whereas we know that it should have type $A[< \ulcorner \rho \urcorner >]$ when evaluating

in environment ρ. Similarly, we do not even require the first argument of @ to be a function. It would be possible to define the evaluation relation with more restrictive types, but this would only complicate later proofs by requiring many additional transports. This choice may be compared to heterogeneous equality, which can similarly simplify proofs merely by being less restrictive than dependent equality types.

Of course, the expected type restrictions on evaluation can still be proved as lemmas.

▶ **Lemma 1.**

$$\frac{\mathsf{eval}\ u\ \rho \Downarrow v}{\ulcorner v \urcorner \equiv u[\ulcorner \rho \urcorner]} \qquad \frac{\mathsf{evals}\ \sigma\ \rho \Downarrow \omega}{\ulcorner \omega \urcorner \equiv \sigma \circ \ulcorner \rho \urcorner} \qquad \frac{f\ @\ v \Downarrow w}{\ulcorner f \urcorner\ \$\ \ulcorner v \urcorner \equiv \ulcorner w \urcorner}$$

Proof. By simultaneous induction on the definitions of the relations eval, evals, and @. ◀

A soundness property follows.

▶ **Lemma 2.**

$$\frac{\mathsf{eval}\ u\ \rho \Downarrow v \qquad \mathsf{eval}\ u\ \rho \Downarrow w}{v \equiv w} \qquad \frac{\mathsf{evals}\ \sigma\ \rho \Downarrow \omega \qquad \mathsf{evals}\ \sigma\ \rho \Downarrow \delta}{\omega \equiv \delta}$$

$$\frac{f\ @\ u \Downarrow v \qquad f\ @\ u \Downarrow w}{v \equiv w}$$

Proof. Using Lemma 1, and that embeddings of values and environments are injective by qVal and qEnv. ◀

5.3 Normal Forms

Having defined the evaluator, we continue with the function quote which maps values to normal forms. The classic notion of *η-long β-normal forms* (see for instance [19]) is used, which interestingly is shared with normalisation by evaluation (cf. [7]).

Like values, normal forms are defined mutually with neutral normal forms, i.e. the application of a variable to normal forms. An important difference is that not all neutral normal forms are normal forms: it is only true for neutral normal forms of the base types (i.e. U and El). This restriction ensures that normal forms are sufficiently η-expanded.

$$\begin{array}{ll}
\textbf{data}\ \mathsf{Nf} : (\Gamma : \mathsf{Con}) \to \mathsf{Ty}\ \Gamma \to \mathsf{Set}\ \textbf{where} & \ulcorner _ \urcorner : \mathsf{Nf}\ \Gamma\ A \to \mathsf{Tm}\ \Gamma\ A \\
\quad \lambda \quad : \mathsf{Nf}\ (\Gamma, A)\ B \to \mathsf{Nf}\ \Gamma(\Pi\ A\ B) & \ulcorner \lambda n \urcorner \quad = \lambda \ulcorner n \urcorner \\
\quad \mathsf{neuU}\ : \mathsf{NN}\ \Gamma\ \mathsf{U} \to \mathsf{Nf}\ \Gamma\ \mathsf{U} & \ulcorner \mathsf{neuU}\ n \urcorner = \ulcorner n \urcorner \\
\quad \mathsf{neuEl} : \mathsf{NN}\ \Gamma\ (\mathsf{El}\ u) \to \mathsf{Nf}\ \Gamma\ (\mathsf{El}\ u) & \ulcorner \mathsf{neuEl}\ n \urcorner = \ulcorner n \urcorner \\
\textbf{data}\ \mathsf{NN} : (\Gamma : \mathsf{Con}) \to \mathsf{Ty}\ \Gamma \to \mathsf{Set}\ \textbf{where} & \ulcorner _ \urcorner : \mathsf{NN}\ \Gamma\ A \to \mathsf{Tm}\ \Gamma\ A \\
\quad \mathsf{var}\ : \mathsf{Var}\ \Gamma\ A \to \mathsf{NN}\ \Gamma\ A & \ulcorner \mathsf{var}\ x \urcorner \quad = \ulcorner x \urcorner \\
\quad \mathsf{app} : \mathsf{NN}\ \Gamma\ (\Pi\ A\ B) \to (n : \mathsf{NN}\ \Gamma\ A) & \ulcorner \mathsf{app}\ m\ n \urcorner = \ulcorner m \urcorner \$ \ulcorner n \urcorner \\
\qquad \to \mathsf{NN}\ \Gamma\ (B[< \ulcorner n \urcorner >]) &
\end{array}$$

Note that normal forms are indexed by regular types, we do not use a notion of normal types. Indeed, normalising types and terms simultaneously only seems to complicate matters, and it is easier to first normalise a term without worrying about its type, then recursively normalise the type. A disadvantage of this choice is that equality of normal forms is not a priori decidable, because it would require to test equality of types, and in turn equality of terms. This issue can be solved once the normalisation function is defined by proving decidability of equality for terms, normal forms, and types simultaneously, as shown in [7].

5.4 Quote

The function quote is defined by induction on the type of the value, together with quoten which maps neutral values to neutral normal forms by recursively applying quote. Like the evaluator, we begin with an informal definition as a function, which is then translated to a relation.

$$\begin{aligned}
&\mathsf{quote} : \{A : \mathsf{Ty}\} \to \mathsf{Val}\ \Gamma\ A \to \mathsf{Nf}\ \Gamma\ A\\
&\mathsf{quote}_{\mathsf{U}}\ (\mathsf{neu}\ v) \quad\ = \mathsf{neuU}\ (\mathsf{quoten}\ v)\\
&\mathsf{quote}_{(\mathsf{El}\ u)}\ (\mathsf{neu}\ v) = \mathsf{neuEl}\ (\mathsf{quoten}\ v)\\
&\mathsf{quote}_{(\Pi\ A\ B)}\ f \quad\ = \lambda(\mathsf{quote}\ (f^{+A}\ @\ \mathsf{neu}\ (\mathsf{var}\ \mathsf{vz})))\\
&\mathsf{quoten} : \mathsf{NV}\ \Gamma\ A \to \mathsf{NN}\ \Gamma\ A\\
&\mathsf{quoten}\ (\mathsf{var}\ x) \quad = \mathsf{var}\ x\\
&\mathsf{quoten}\ (\mathsf{app}\ f\ v) = \mathsf{app}\ (\mathsf{quoten}\ f)\ (\mathsf{quote}\ v)
\end{aligned}$$

A value of a base type is necessarily neutral, hence it suffice to use quoten in that case. For function types, the definition of normal forms requires the result to be an abstraction. This is done by η-expending the value, and applying quote to the body of the resulting abstraction. The η-expansion is somewhat technical to define. First, the function is weakened as f^{+A} to allow the introduction of a new variable of type A represented by the De Bruijn index vz. This variable is turned into a value by the var and neu constructors, and the weakened function is applied using @, giving the body of the η-expansion.

Beside the problems of termination and correctness with regards to quotient constructors which already appeared in the evaluator, one may note that quote is not defined on the _[_] type constructor. We will later show that the definition for _[_] can in fact be inferred from the other cases and the equality constraints. For now we again ignore all issues by considering the big step semantics of quote.

$$\begin{aligned}
&\textbf{data}\ \mathsf{quote} : \mathsf{Val}\ \Gamma\ A \to \mathsf{Nf}\ \Gamma\ A \to \mathsf{Prop}\ \textbf{where}\\
&\quad \mathsf{quoteU}\ : \{v : \mathsf{NV}\ \mathsf{U}\} \to \mathsf{quoten}\ v \Downarrow n \to \mathsf{quote}\ (\mathsf{neu}\ v) \Downarrow (\mathsf{neu}\ n)\\
&\quad \mathsf{quoteEl} : \{v : \mathsf{NV}\ (\mathsf{El}\ t)\} \to \mathsf{quoten}\ v \Downarrow n \to \mathsf{quote}\ (\mathsf{neu}\ v) \Downarrow (\mathsf{neu}\ n)\\
&\quad \mathsf{quote}\Pi : f^{+A}\ @\ (\mathsf{neu}\ (\mathsf{var}\ \mathsf{vz})) \Downarrow v \to \mathsf{quote}\ v \Downarrow n \to \mathsf{quote}\ f \Downarrow (\lambda\ n)\\
&\textbf{data}\ \mathsf{quoten} : \mathsf{NV}\ \Gamma\ A \to \mathsf{NN}\ \Gamma\ A \to \mathsf{Prop}\ \textbf{where}\\
&\quad \mathsf{quotenvar}\ : \mathsf{quoten}\ (\mathsf{var}\ x) \Downarrow (\mathsf{var}\ x)\\
&\quad \mathsf{quotenapp} : \mathsf{quoten}\ f \Downarrow m \to \mathsf{quote}\ v \Downarrow n \to \mathsf{quoten}\ (\mathsf{app}\ f\ v) \Downarrow (\mathsf{app}\ m\ n)
\end{aligned}$$

A coherence result in the style of Lemma 1 is proved by induction on the relation.

▶ **Lemma 3.**

$$\frac{\mathsf{quote}\ v \Downarrow n}{\ulcorner n \urcorner \equiv \ulcorner v \urcorner} \qquad \frac{\mathsf{quoten}\ m \Downarrow n}{\ulcorner n \urcorner \equiv \ulcorner m \urcorner}$$

5.5 Normalisation

Finally, terms are normalised by evaluating in the identity environment and applying quote.

$$\mathsf{norm}\ u \Downarrow n = \Sigma(v : \mathsf{Val}\ \Gamma\ A)\ \mathsf{eval}\ u\ \mathsf{idenv} \Downarrow v\ \wedge\ \mathsf{quote}\ v \Downarrow n$$

With this definition, stability and completeness of BSN can already be proved.

▶ **Theorem 4** (Completeness).

$$\frac{\mathsf{norm}\ u \Downarrow n}{\ulcorner n \urcorner \equiv u}$$

Proof. Immediate by Lemmas 1 and 3. ◀

▶ **Theorem 5** (Stability).

$$\frac{n : \mathsf{Nf}\ \Gamma\ A}{\mathsf{norm}\ \ulcorner n \urcorner \Downarrow n} \qquad \frac{n : \mathsf{NN}\ \Gamma\ A}{\Sigma(v : \mathsf{NV}\ \Gamma\ A)\ \mathsf{eval}\ \ulcorner n \urcorner \Downarrow (\mathsf{neu}\ v)\ \wedge\ \mathsf{quoten}\ v \Downarrow n}$$

Proof. By simultaneous induction on normal forms and neutral normal forms. ◀

6 Correctness of BSN

Two main results must be proved in order to establish the correctness of BSN. Termination states that the normalisation relation is defined on every term.

$$\forall(u : \mathsf{Tm}\ \Gamma\ A),\ \exists(n : \mathsf{Nf}\ \Gamma\ A),\ \mathsf{norm}\ u \Downarrow n$$

Soundness states that normalisation can only give one result for each term.

$$\frac{\mathsf{norm}\ u \Downarrow n \qquad \mathsf{norm}\ u \Downarrow m}{n \equiv m}$$

Termination and soundness together imply that the normalisation relation defines a function from terms to normal forms, and the remaining coherence properties (completeness and stability) have already been proved in the previous section.

In this section, we first provide a short proof of soundness using known results on NBE.

Next, we define a partial normalisation of types, and the notion of skeleton of a type. Together, they give a very simple induction principle for the syntax of dependent types. Using this simplified induction principle, it is fairly straightforward to adapt the proof of termination for simple types [3], based on the strong computability predicate.

6.1 Soundness, by NBE

The original presentation of BSN for the simply-typed lambda-calculus proves soundness using a logical binary relation, similar to the use of strong computability for termination presented later in this section. Unfortunately, this proof seems hard to adapt to the quotiented syntax.

However there is an alternative proof, much shorter if not as interesting. The key observation is that BSN uses the same notion of normal forms as normalisation by evaluation (cf. [7] for a formal proof of NBE for type theory – we use the very same syntax and definition of normal forms). A direct consequence of the existence of a normalisation function such as NBE is that there is exactly one normal form in each equivalence class of terms, which in the quotiented syntax means that the embedding of normal forms is injective.

▶ **Theorem 6.**

$$\frac{n, m : \mathsf{Nf}\ \Gamma\ A \qquad \ulcorner n \urcorner \equiv \ulcorner m \urcorner}{n \equiv m}$$

Proof. By soundness and stability of normalisation by evaluation. ◀

▶ **Theorem 7** (Soundness).

$$\frac{\mathsf{norm}\ u \Downarrow n \qquad \mathsf{norm}\ u \Downarrow m}{n \equiv m}$$

Proof. Immediate by Theorems 4 and 6. ◀

It can of course be argued that defining a normalisation function using another normalisation function defeats the object. However we think that it is interesting to consider BSN not so much as alternative normalisation function than as an alternative definition for the function which can also be obtained through NBE. This proof of soundness becomes more sensible from this point of view: as soon as we prove that the functions defined by NBE and BSN coincide (for which completeness of BSN is a key result), all correctness properties which are known to hold for NBE – in particular soundness – transfer to BSN.

6.2 Substitution-Free Types

An interesting issue was mentioned while defining quote: the natural definition is by induction on types, but only considers the constructors U, El, and Π, forgetting both _[_] and the quotient constructors. In this subsection, we show that this type of definition is in fact always correct, by defining substitution-free types, and proving that they are isomorphic to regular types.

Substitution free types are defined together with their embedding into regular types.

$$\begin{aligned}
&\textbf{data } \mathsf{Ty^{sf}} : \mathsf{Con} \to \mathsf{Set} \textbf{ where} && \ulcorner _ \urcorner : \mathsf{Ty^{sf}}\ \Gamma \to \mathsf{Ty}\ \Gamma \\
&\quad \mathsf{U} : \mathsf{Ty^{sf}}\ \Gamma && \ulcorner \mathsf{U} \urcorner \quad = \mathsf{U} \\
&\quad \mathsf{El} : \mathsf{Tm}\ \Gamma\ \mathsf{U} \to \mathsf{Ty^{sf}}\ \Gamma && \ulcorner \mathsf{El}\ u \urcorner \quad = \mathsf{El}\ u \\
&\quad \Pi : (A : \mathsf{Ty^{sf}}\ \Gamma) \to \mathsf{Ty^{sf}}\ (\Gamma, \ulcorner A \urcorner) \to \mathsf{Ty^{sf}}\ \Gamma && \ulcorner \Pi\ A\ B \urcorner = \Pi\ \ulcorner A \urcorner\ \ulcorner B \urcorner
\end{aligned}$$

We will now define an evaluation function from types to substitution-free types, which will be the inverse of the embedding $\ulcorner \_ \urcorner$. This requires to interpret every remaining type constructors in substitution-free types.

First, the application of a substitution to a substitution-free type is defined inductively.

$$\begin{aligned}
&_[_] : \mathsf{Ty^{sf}}\ \Delta \to \mathsf{Sub}\ \Gamma\ \Delta \to \mathsf{Ty^{sf}}\ \Gamma \\
&\mathsf{U}[\sigma] \quad = \mathsf{U} \\
&(\mathsf{El}\ u)[\sigma] \quad = \mathsf{El}(u[\sigma]) \\
&(\Pi\ A\ B)[\sigma] = \Pi\ (A[\sigma])\ (B[\sigma \uparrow \ulcorner A \urcorner])
\end{aligned}$$

The definition directly follows the equations U[], El[], and Π[] from the syntax of regular types. The remaining equations can be proved by induction.

$$\frac{A : \mathsf{Ty^{sf}}\ \Gamma}{A[\mathsf{id}] \equiv A} \qquad \frac{A : \mathsf{Ty^{sf}}\ \Theta \qquad \sigma : \mathsf{Sub}\ \Delta\ \Theta \qquad \nu : \mathsf{Sub}\ \Gamma\ \Delta}{A[\sigma \circ \nu] \equiv A[\sigma][\nu]}$$

Put together, this defines the evaluation function: U, El, and Π are interpreted by the respective constructors, substitutions are applied using the previous recursive definition, the equations U[], El[], and Π[] hold trivially, and we just verified that [id] and [][] are respected. It is easy to verify that this evaluation function is indeed the inverse of the embedding, therefore regular and substitution-free types are isomorphic.

This gives an alternative, much simpler induction principle for types.

▶ **Lemma 8.** *To define a function on types, it suffice to define it inductively for the constructors* U*,* El*, and* Π*.*

Proof. The hypothesis of the lemma corresponds exactly to a definition of the function on substitution-free types. This function is then extended to regular types through the isomorphism previously defined. ◀

6.3 Type Skeletons

If we were to immediately define strong computability, we would face a second issue regarding the induction principle for types: it will often be the case that when proving a result by induction on types and considering a type $\Pi\ A\ B$, we need to apply the induction hypothesis not on B, but instead on $B[\sigma]$ for some substitution σ, which is not allowed by the induction principle of types. However, if we were to forget substitutions altogether, then B or $B[\sigma]$ would be the same. This is exactly the idea behind the skeleton of a type: by deleting all substitutions, we obtain a well-founded notion of size of types, for which B and $B[\sigma]$ are equivalent.

Formally, a type skeleton correspond to the non-dependent structure of types: either a base type or a function type.

```
data Sk : Set where
  base : Sk
  Π    : Sk → Sk → Sk
```

Defining the skeleton of a type is straightforward, and all quotient constructors are clearly respected.

```
skeleton : Ty Γ → Sk
skeleton U         = base
skeleton (El u)    = base
skeleton (Π A B) = Π (skeleton A) (skeleton B)
skeleton (A[σ])    = skeleton A
```

Using the skeleton of types as size indicators for induction, the example of problematic induction given at the beginning of this subsection becomes valid.

▶ **Lemma 9.** *To define a function f on types, it suffice to*

- *Define f on the base types* U *and* El*.*
- *Define f on any type $\Pi\ A\ B$, while assuming that f is defined on C for any type C with the same skeleton as either A or B.*

Proof. The proof is the same as for Lemma 8, but additionally uses the skeletons as size indicators to ensure that the inductive definition is well-founded. Formally, this means that the function is defined by induction on type skeletons, then by pattern matching on the types of a given skeleton. ◀

6.4 Strong Computability

The proof of termination is based on a Tait-style [25] predicate on values, called strong computability. This subsection introduces strong computability, together with some important lemmas.

Strong computability is defined by induction on types, using Lemma 9

- A value v of a base type is SC if quote terminates on v.
- A value f of type $\Pi\ A\ B$ is SC if the application of f to a SC value v of type A gives a SC result of type B.

$$
\begin{aligned}
&\mathsf{scv} : \{A : \mathsf{Ty}\} \to \mathsf{Val}\ \Gamma\ A \to \mathsf{Set}\\
&\mathsf{scv}_{\mathsf{U}}\ v \qquad = \Sigma(n : \mathsf{Nf}\ \Gamma\ \mathsf{U})\ \mathsf{quote}\ v \Downarrow n\\
&\mathsf{scv}_{(\mathsf{El}\ u)}\ v \quad = \Sigma(n : \mathsf{Nf}\ \Gamma\ (\mathsf{El}\ u))\ \mathsf{quote}\ v \Downarrow n\\
&\mathsf{scv}_{(\Pi\ A\ B)}\ f = \forall(\alpha : \mathsf{Wk}\ \Delta\ \Gamma)(v : \mathsf{Val}\ \Delta\ A^{+\alpha}) \to \mathsf{scv}\ v \to\\
&\qquad\qquad \Sigma(C : \mathsf{Ty}\ \Delta)\ \Sigma(w : \mathsf{Val}\ \Delta\ C)\\
&\qquad\qquad (f^{+\alpha}\ @\ v \Downarrow w)\ \wedge\ (\mathsf{scv}\ w)\ \wedge\ (\mathsf{skeleton}\ C \equiv \mathsf{skeleton}\ B)
\end{aligned}
$$

Some remarks can be made regarding the case of function types. Firstly, stability under application is understood up to weakening, i.e. the argument v need not be in the same context Γ as the function f, but may instead come from a weaker context Δ, where the weakening $\alpha : \mathsf{Wk}\ \Delta\ \Gamma$ expresses that Δ is weaker than Γ.

Secondly, as in the definition of the evaluation relation, we prefer not to restrict the result type to simplify the upcoming proofs, hence we merely require that there exist a value w of some type C. However, the definition would not be well-founded without any restriction on C, since we inductively refer to strong computability at type C. Thus, we ask for C to have the same skeleton as B. In this way, strong computability for $\Pi\ A\ B$ is defined based on strong computability for types with the same skeleton as either A or B.

Strong computability is extended to environments pointwise.

$$
\begin{aligned}
&\mathsf{sce} : \mathsf{Env}\ \Gamma\ \Delta \to \mathsf{Set}\\
&\mathsf{sce}\ \epsilon \qquad = \top\\
&\mathsf{sce}\ (\rho, v) = \mathsf{sce}\ \rho\ \wedge\ \mathsf{scv}\ v
\end{aligned}
$$

Let us now prove some lemma on strong computability. Throughout this subsection, we implicitly use Lemma 9 when proceeding by induction on types.

▶ **Lemma 10.** *Strong computability is stable under weakening:*

$$\frac{v : \mathsf{Val}\ \Gamma\ A \qquad \mathsf{scv}\ v \qquad \alpha : \mathsf{Wk}\ \Delta\ \Gamma}{\mathsf{scv}\ v^{+\alpha}} \qquad \frac{\rho : \mathsf{Env}\ \Gamma\ \Theta \qquad \mathsf{sce}\ \rho \qquad \alpha : \mathsf{Wk}\ \Delta\ \Gamma}{\mathsf{sce}\ \rho^{+\alpha}}$$

Proof. For values, the proof is by induction on the type. For base types, stability of quote under weakening is used. For function types, the proof is immediate, since the definition of strong computability already accounts for weakening.

For environments, the proof is trivial by induction. ◀

▶ **Lemma 11.** *Strong computability is a mere proposition, i.e. any two proofs of strong computability are equal.*

$$\frac{p, q : \mathsf{scv}\ v}{p \equiv q} \qquad \frac{p, q : \mathsf{sce}\ \rho}{p \equiv q}$$

Proof. For values, the proof is by induction on the type. For base types, we use soundness of quote, that is

$$\frac{\mathsf{quote}\ v \Downarrow n \qquad \mathsf{quote}\ v \Downarrow m}{n \equiv m}$$

which follows easily from Lemma 3 and Theorem 6. For function types, Lemma 2 is used.

For environments, the proof is trivial by induction. ◀

The most important lemma regarding strong computability is that it implies termination of quote. A form of the converse for neutral values is proved simultaneously.

▶ **Lemma 12.**

$$\frac{v : \mathsf{Val}\ \Gamma\ A \qquad \mathsf{scv}\ v}{\Sigma(n : \mathsf{Nf}\ \Gamma\ A),\ \mathsf{quote}\ v \Downarrow n}\ \textit{(quote)}$$

$$\frac{v : \mathsf{NV}\ \Gamma\ A \qquad \Sigma(n : \mathsf{NN}\ \Gamma\ A),\ \mathsf{quoten}\ v \Downarrow n}{\mathsf{scv}\ (\mathsf{neu}\ v)}\ \textit{(unquote)}$$

Proof. By mutual induction on the type A. The base cases are trivial by definition of strong computability. Consider a function type $\Pi\ A\ B$.

For the case (*quote*), let f be a strongly computable value of type $\Pi\ A\ B$. Following the definition of quote for function types, we need to prove that there exist some $v : \mathsf{Val}\ (\Gamma, A)\ B$ and $n : \mathsf{Nf}\ (\Gamma, A)\ B$ such that

$$f^{+A}\ @\ \mathsf{neu}\ (\mathsf{var}\ \mathsf{vz}) \Downarrow v \quad \wedge \quad \mathsf{quote}\ v \Downarrow n$$

In this expression, the variable vz has type $A[\pi_1 \mathsf{id}]$. Furthermore quoten trivially terminates on variables, hence (*unquote*) implies that neu (var vz) is strongly computable by induction hypothesis. Then by definition of strong computability $f^{+A}\ @\ \mathsf{neu}\ (\mathsf{var}\ \mathsf{vz}) \Downarrow v$ holds for some strongly computable v, and we may verify using Lemma 1 that v has type B. Since v is strongly computable of type B, there exist by induction hypothesis $n : \mathsf{Nf}\ (\Gamma, A)\ B$ such that quote $v \Downarrow n$. Therefore, quote $f \Downarrow (\lambda\ n)$.

Inversely, for the case (*unquote*), assume quoten $f \Downarrow n$ with $f : \mathsf{NV}\ \Gamma\ (\Pi\ A\ B)$, and let us prove that neu f is strongly computable. Let $\alpha : \mathsf{Wk}\ \Delta\ \Gamma$ and $v : \mathsf{Val}\ \Delta\ A^{+\alpha}$ strongly computable. Let us prove that neu (app $f^{+\alpha}\ v$) satisfies the conditions of the definition of strong computability for function types. Firstly,

$$(\mathsf{neu}\ f^{+\alpha})\ @\ v \Downarrow (\mathsf{neu}\ (\mathsf{app}\ f^{+\alpha}\ v))$$

is immediate since f is neutral. Furthermore, by induction hypothesis (*unquote*) and definition of quoten, to prove that neu (app $f^{+\alpha}\ v$) is strongly computable, it suffice to check that quoten terminates on $f^{+\alpha}$ and quote terminates on v. The former holds by hypothesis using that quoten is stable by weakening, while the latter holds by induction hypothesis (*quote*). Finally, one may verify that the type of neu (app $f^{+\alpha}\ v$) can be expressed as B with some substitutions and weakenings applied, hence its skeleton is the same as B. It follows that f is strongly computable. ◀

▶ **Lemma 13.** *The identity environment is strongly computable.*

$$\frac{\Gamma : \mathsf{Con}}{\mathsf{sce}\ \mathsf{idenv}_\Gamma}$$

Proof. Lemma 12 implies that all variables are strongly computable because they are neutral values for which quoten trivially terminates. The result follows by induction on Γ, using Lemma 10. ◀

6.5 Termination

All the tools are now available to prove the main termination result.

▶ **Theorem 14.** *Evaluation in a strongly computable environment terminates, and yields a strongly computable result.*

$$\frac{u : \mathsf{Tm}\ \Gamma\ A \qquad \rho : \mathsf{Env}\ \Delta\ \Gamma \qquad \mathsf{sce}\ \rho}{\Sigma(B : \mathsf{Ty}\ \Delta)\Sigma(v : \mathsf{Val}\ \Delta\ B)\ \mathsf{eval}\ u\ \rho \Downarrow v\ \wedge\ \mathsf{scv}\ v}$$

$$\frac{\sigma : \mathsf{Sub}\ \Gamma\ \Theta \qquad \rho : \mathsf{Env}\ \Delta\ \Gamma \qquad \mathsf{sce}\ \rho}{\Sigma(\nu : \mathsf{Env}\ \Gamma\ \Theta)\ \mathsf{evals}\ \sigma\ \rho \Downarrow \nu\ \wedge\ \mathsf{sce}\ \nu}$$

The theorem is proved by induction on terms and substitutions. Regular constructors are unproblematic, in the sense that the proofs does not change significantly compared to the case of an unquotiented syntax. However, we also need to verify that quotient constructors are respected, i.e. that for every equality constructor $u \equiv v$, the proof (seen as a function) of Theorem 14 gives equal results on u and v.

A simple way to ensure this is to prove that the types corresponding to Theorem 14 are mere propositions. In that case, when considering an equality constructor $u \equiv v$, the result of a proof of Theorem 14 on u and v will necessarily be equal since both are elements of the same mere proposition.

▶ **Lemma 15.** *For any* $u : \mathsf{Tm}\ \Gamma\ A$, $\sigma : \mathsf{Sub}\ \Gamma\ \Theta$ *and* $\rho : \mathsf{Env}\ \Delta\ \Gamma$, *the following types are mere propositions.*

$$\Sigma(B : \mathsf{Ty}\ \Delta)\Sigma(v : \mathsf{Val}\ \Delta\ B)\ \mathsf{eval}\ u\ \rho \Downarrow v\ \wedge\ \mathsf{scv}\ v$$
$$\Sigma(\nu : \mathsf{Env}\ \Gamma\ \Theta)\ \mathsf{evals}\ \sigma\ \rho \Downarrow \nu\ \wedge\ \mathsf{sce}\ \nu$$

Proof. By Lemma 2, a term can only evaluate to a single value v. Furthermore, the types $\mathsf{eval}\ u\ \rho \Downarrow v$ and $\mathsf{scv}\ v$ are mere propositions, by definition and by Lemma 11 respectively. The result follows. The proof is similar in the case of substitutions. ◀

Proof of Theorem 14. By induction on terms and substitutions. We split the constructors into three groups:

- All quotient constructors are respected by Lemma 15.
- Almost all regular constructors are very straightforward: the result of evaluation is obtained by following the definition of the evaluator and applying the induction hypotheses, and strong computability of the result comes directly from the hypotheses. The exceptions to this pattern are λ and app, for which we give more detailed proofs below.
- For an abstraction λu of type $\Pi\ A\ B$ evaluated in a strongly computable environment $\rho : \mathsf{Env}\ \Delta\ \Gamma$, evaluation is trivial since it simply yields the closure $\mathsf{clos}\ u\ \rho$. Let us show that this closure is strongly computable.
  Let $\alpha : \mathsf{Wk}\ \Theta\ \Delta$, and $v : \mathsf{Val}\ \Theta\ (A[\ulcorner\rho\urcorner]^{+\alpha})$ strongly computable. Then by Lemma 10, $(\rho^{+\alpha}, v)$ is a strongly computable environment, hence by induction hypothesis there exists w strongly computable such that $\mathsf{eval}\ u\ (\rho^{+\alpha}, v) \Downarrow w$. It follows that $(\mathsf{clos}\ u\ \rho)^{+\alpha}\ @\ v \Downarrow w$. Finally, we may verify using Lemma 1 that the type of w must have the same skeleton as B. It follows that $\mathsf{clos}\ u\ \rho$ is strongly computable.
- Consider an application $\mathsf{app}\ u$ with $u : \mathsf{Tm}\ \Gamma\ (\Pi\ A\ B)$ evaluated in a strongly computable environment $(\rho, v) : \mathsf{Env}\ \Delta\ (\Gamma, A)$. By induction hypothesis, there exists f strongly computable such that $\mathsf{eval}\ u\ \rho \Downarrow f$. It can be verified using Lemma 1 that f has type $\Pi\ (A[\ulcorner\rho\urcorner])\ (B[\ulcorner\rho\urcorner \uparrow A])$. Hence, because f and v are strongly computable, there exist w strongly computable such that $f\ @\ v \Downarrow w$. Then we obtain by the definition of the evaluation relation that $\mathsf{eval}\ (\mathsf{app}\ u)\ (\rho, v) \Downarrow w$, proving the result. ◀

▶ **Theorem 16** (Termination)**.** *Normalisation terminates.*

$$\frac{u : \mathsf{Tm}\ \Gamma\ A}{\Sigma(n : \mathsf{Nf}\ \Gamma\ A),\ \mathsf{norm}\ u \Downarrow n}$$

Proof. Let $u : \mathsf{Tm}\ \Gamma\ A$. By Lemma 13 and Theorem 14, there exist v strongly computable such that $\mathsf{eval}\ u\ \mathsf{idenv} \Downarrow v$. By Lemma 1, one may verify that v has type A. Finally, by Lemma 12, there exist $n : \mathsf{Nf}\ \Gamma\ A$ such that $\mathsf{quote}\ v \Downarrow n$. It follows that $\mathsf{norm}\ u \Downarrow n$. ◀

By Theorems 7 and 16, norm defines a function from quotiented terms to normal forms, and by Theorems 4 and 5, it is the inverse of the embedding of normal forms. Therefore, we have proved that big step normalisation defines a normalisation function.

7 Formalisation of BSN in a Cubical Type Theory

Most of the present work has been formalised [5] using Agda [23]. Precisely, Sections 3 to 5 have been fully formalised, and Section 6 partially so – what remains to do in the latter is equality reasoning. This formalisation is expressed in a cubical type theory (CTT, cf. [11]) using the cubical mode of Agda. This differs from the present paper, which uses a strict type theory for simplicity. The choice of CTT allows to easily express QIIT as a special case of higher inductive types (HIT, cf. [24]), which from the technical point of view is a notable improvement over previous implementations of QIIT in non-cubical Agda, which had to introduce all quotient constructors as additional axioms (e.g. [7, 8]).

As explained in [8], simply considering a QIIT as a special case of HIT leads to unexpected results. For instance, in the case of the quotiented syntax, U[] and [id] give two proofs of U[id] ≡ U, and these proofs are distinct in a non-strict type theory. Therefore, this naive implementation of QIIT leads to a syntax which is not a set in the type theoretic sense, i.e. uniqueness of identity proofs (UIP) does not holds. It follows by Hedberg's theorem [17] that equality is undecidable in this syntax, which is definitively not what was expected.

The solution is to truncate the syntax to a set, by the addition of the following constructors:

$$\begin{aligned}
&\mathsf{setTy} &&: \{A\ B : \mathsf{Ty}\ \Gamma\}\ (p\ q : A \equiv B) \to p \equiv q\\
&\mathsf{setSub} &&: \{\sigma\ \nu : \mathsf{Sub}\ \Gamma\ \Delta\}\ (p\ q : \sigma \equiv \nu) \to p \equiv q\\
&\mathsf{setTm} &&: \{s\ t : \mathsf{Tm}\ \Gamma\ A\}\ (p\ q : s \equiv t) \to p \equiv q
\end{aligned}$$

Note that the corresponding constructor for contexts is unnecessary, because it can be proved that contexts form a set from the fact that types are a family of sets.

In order to adapt the proof of big step normalisation to CTT with this implementation of QIIT, there are two problems to solve:

- The proof of BSN uses the UIP axiom of the strict type theory.[4] Since we lose this axiom in CTT, its uses must be replaced.
- The additional truncation constructors of QIIT must be taken into account whenever we use induction on a QIIT.

Both problems can be solved together by proving that all the types considered in the proof of BSN are in fact sets. Indeed, any use of the UIP axiom can then be replaced by the proof of UIP for the appropriate type, and when defining a function by induction on a QIIT, the set-truncation constructor can be mapped to the proof that the codomain is a set.

We will not detail all the proofs of UIP because they are fairly repetitive, but we will explain the general techniques used. There are some trivial cases: all QIIT (terms, types, values) are explicitly truncated to sets. Mere propositions (the big step relations, strong computability) are always sets. What remains are regular inductive types, such as variables, normal forms, substitution-free types... For such types, Hedberg's theorem [17] is an

[4] While we never explicitly refer to the UIP axiom in this presentation of the proof, it is used whenever we prove a lemma of the form $L : (x : A) \to f(x) \equiv g(x)$ by induction on a QIIT A. Indeed, for a quotient constructor of type $a \equiv b$ in A, we need to provide an equality between equalities $L(a) \equiv L(b)$. This is trivial with UIP – hence why such cases are neglected in the proof – but is in general problematic in a non-strict metatheory. An example is the coherence lemmas for weakening of values.

extremely useful tool. For instance, it is easy to verify that equality of variables is decidable, which implies that they are a set. Even for types which do not a priori have decidable equality (e.g. normal forms), it is still possible to adapt the techniques and lemmas used for Hedberg's theorem to prove UIP.

8 Conclusion and Further Work

We have formalised big step normalisation for a simple dependent type theory, and proved its correctness. Crucially, a quotiented syntax of type theory based on QIIT is used to reduce the complexity of this proof. While the proof of BSN for type theory shares many similarities with the case of the simply-typed lambda-calculus, it requires some additional steps, for instance a simplified induction principle for the syntax of types.

This work is also an interesting application of the QIIT syntax of type theory, since it provides an example in which using this syntax has an important impact on the proof. The implementation of the QIIT syntax using HIT in cubical Agda, and its use in the formalisation of BSN is also a practical validation of ideas which were developed in [8].

Since we have only considered a minimalist type theory in this work, it is natural to try to extend it. A first step in this direction could be the addition of some inductive types. This was already done in the non-dependently typed case in [3], by adding integers (System T). In order to handle inductive types in BSN, the general idea is to add the inductive constructors to values and normal forms, and add the elimination principles to neutral values and neutral normal forms, adapting eval and quote accordingly. A next step could then be to add W-types, so as to allow the use of arbitrary inductive types. Another equally interesting extension would be to replace the abstract universe U – which contains no closed terms – with a more useful universe equipped with type constructors.

References

1 Thorsten Altenkirch, Paolo Capriotti, Gabe Dijkstra, Nicolai Kraus, and Fredrik Nordvall Forsberg. Quotient inductive-inductive types. In *International Conference on Foundations of Software Science and Computation Structures*, pages 293–310. Springer, Cham, 2018.

2 Thorsten Altenkirch and James Chapman. Tait in one big step. In *MSFP@ MPC*, 2006.

3 Thorsten Altenkirch and James Chapman. Big-step normalisation. *Journal of Functional Programming*, 19(3-4):311–333, 2009.

4 Thorsten Altenkirch, Peter Dybjer, Martin Hofmann, and Phil Scott. Normalization by evaluation for typed lambda calculus with coproducts. In *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*, pages 303–310. IEEE, 2001.

5 Thorsten Altenkirch and Colin Geniet. Agda formalisation for the paper big step normalisation for type theory. Available at `https://github.com/colingeniet/big-step-normalisation`, 2019.

6 Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Categorical reconstruction of a reduction free normalization proof. In *International Conference on Category Theory and Computer Science*, pages 182–199. Springer, 1995.

7 Thorsten Altenkirch and Ambrus Kaposi. Normalisation by evaluation for dependent types. In *1st conference on Foundational Structures in Computation and Deduction (FSCD)*, 2016.

8 Thorsten Altenkirch and Ambrus Kaposi. Type theory in type theory using quotient inductive types. *ACM SIGPLAN Notices*, 51(1):18–29, 2016.

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

10 James Chapman. Type theory should eat itself. *Electronic Notes in Theoretical Computer Science*, 228:21–36, 2009.

11 Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: a constructive interpretation of the univalence axiom. *arXiv preprint*, 2016. arXiv:1611.02108.

12 Thierry Coquand. An algorithm for testing conversion in type theory. *Logical frameworks*, 1:255–279, 1991.

13 Gabe Dijkstra. *Quotient inductive-inductive definitions*. PhD thesis, University of Nottingham, 2017.

14 Daniel J Dougherty and Ramesh Subrahmanyam. Equality between functionals in the presence of coproducts. In *Proceedings of Tenth Annual IEEE Symposium on Logic in Computer Science*, pages 282–291. IEEE, 1995.

15 Peter Dybjer. Internal type theory. In *International Workshop on Types for Proofs and Programs*, pages 120–134. Springer, 1995.

16 Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and types*, volume 7. Cambridge university press Cambridge, 1989.

17 Michael Hedberg. A coherence theorem for martin-löf's type theory. *Journal of Functional Programming*, 8(4):413–436, 1998.

18 Martin Hofmann. Syntax and semantics of dependent types. In *Extensional Constructs in Intensional Type Theory*, pages 13–54. Springer, 1997.

19 Jean-Pierre Jouannaud and Albert Rubio. Rewrite orderings for higher-order terms in η-long β-normal form and the recursive path ordering. *Theoretical Computer Science*, 208(1-2):33–58, 1998.

20 Ambrus Kaposi, András Kovács, and Thorsten Altenkirch. Constructing quotient inductive-inductive types. *Proceedings of the ACM on Programming Languages*, 3(POPL):2, 2019.

21 Paul Blain Levy. *Call-by-push-value*. PhD thesis, Queen Mary and Westfield College, University of London, 2001.

22 Paul-André Mellies. Typed λ-calculi with explicit substitutions may not terminate. In *International Conference on Typed Lambda Calculi and Applications*, page 32. Springer, 1995.

23 Ulf Norell. *Towards a practical programming language based on dependent type theory*, volume 32. Citeseer, 2007.

24 Univalent Foundations Program. *Homotopy type theory: Univalent foundations of mathematics*. Univalent Foundations, 2013.

25 W. W. Tait. Intensional interpretations of functionals of finite type i. *Journal of Symbolic Logic*, 32(2):198–212, 1967. doi:10.2307/2271658.