A Formal Proof of the Strong Normalization Theorem for System T in Agda[*]

Sebastián Urciuoli
Universidad ORT Uruguay
Montevideo, Uruguay
urciuoli@ort.edu.uy

We present a framework for the formal meta-theory of lambda calculi in first-order syntax, with two sorts of names, one to represent both free and bound variables, and the other for constants, and by using Stoughton's multiple substitutions. On top of the framework we formalize Girard's proof of the Strong Normalization Theorem for both the simply-typed lambda calculus and System T. As to the latter, we also present a simplification of the original proof. The whole development has been machine-checked using the Agda system.

1 Introduction

In [21] a framework was presented for the formal meta-theory of the pure untyped lambda calculus in first-order abstract syntax (FOAS) and using only one sort of names for both free and bound variables[1]. Based upon Stoughton's work on multiple substitutions [19], the authors were able to give a primitive recursive definition of the operation of substitution which does not identify alpha-convertible terms[2], avoids variable capture, and has a homogeneous treatment in the case of abstractions. Such a definition of substitution is obtained by renaming every bound name to a sufficiently fresh one. The whole development has been formalized in constructive type theory using the Agda system [16].

The framework has been used since then to verify many fundamental meta-theoretic properties of the lambda calculus including: Subject Reduction for the simply-typed lambda calculus (STLC) in [8]; the Church-Rosser Theorem for the untyped lambda calculus also in [8]; the Standardization Theorem in [9], and; the Strong Normalization Theorem for STLC in [24], and by using F. Joachimski and R. Matthes' syntactical method [14]. Now in this paper, we continue the same line of work and formalize the Strong Normalization Theorem for System T, and we also present a new and different mechanization for STLC.

System T extends STLC by adding primitive recursive functions on natural numbers. It has its roots in K. Gödel's work presented in [12], and it was originally developed to study the consistency of Peano Arithmetic. The Strong Normalization Theorem states that every program (term) in some calculus under consideration is strongly normalizing. A term is *strongly normalizing* if and only if its computation always halts regardless of the reduction path been taken. This result for System T is already well known. In this development we mechanize J.-Y. Girard's proof presented in [11], which in turn is based on W. W. Tait's method of *computability* or *reducible functions* [20] (henceforth we shall refer to Girard and Tait's method or proof interchangeably). This method defines a (logical) relation between terms and types that is fitter than the Strong Normalization Theorem, and hence it enables a more powerful

[*]This work is partially supported by Agencia Nacional de Investigación e Innovación (ANII), Uruguay.

[1]Both the previous framework and the one presented here use named variables, it bears repeating. In a contrary sense, there are nameless approaches, e.g., de-Bruijn indices [5] or locally nameless syntax [6], which use numbers to identify the variables.

[2]Or without using Barendregt's variable convention.

© S. Urciuoli
This work is licensed under the
Creative Commons Attribution License.

induction hypothesis. Any term related to some type under such a relation is said to be *reducible*. Then the method consists of two steps: first, to prove that all reducible term are strongly normalizing, and secondly to prove that all typed terms are reducible.

Initially, the sole objective of this work was to formalize a proof of the Strong Normalization Theorem but only for System T, and by using the framework presented in [21]. Of course, the syntax of the pure lambda terms had to be extended to include the term-formers for the natural numbers and the recursion operator[3]. For this, we based ourselves upon a standard definition of the lambda terms in which two disjoint sort of names are used, one to represent the variables, and the other for the constants, e.g., see [13]. Now, instead of restricting ourselves to a specific set of constants, we shall allow any (countable) set. Once the syntax of the framework had been parameterised it felt natural to parameterise the reduction schema as well, as these relations are often defined by the syntax. The work went a bit further, and the first part of the proof was also abstracted for a class of calculi to be defined; this step consists mainly in analysing reduction paths. To round up, hitherto the work evolved from formalizing the proof of the Strong Normalization Theorem in System T, into also providing a general-purpose framework with theories for substitution, alpha-conversion, reduction and reducible terms of simple types.

Now, having such a framework it was a good time to revisit the previous formalization of the Strong Normalization Theorem for STLC presented in [24]. There, the definition of the logical relation was based on the one in the POPLMark Challenge 2 [2], and it included the context of variables. In addition to that, a syntactical characterization based on [14] was used to define the type of the strongly normalizing terms. In this development, we shall use a standard definition of the logical relation which does not contain the context, and an accessibility characterization of the strongly normalizing terms based on [3]. Furthermore, the proof for STLC is contained in the one for System T, so it serves both as a milestone in this exposition, as well as to show the incremental nature of the whole method presented here.

The last result presented in this development is about a simplification in Girard's proof of the Strong Normalization Theorem for System T. More specifically, in the second part of the proof there is a lemma whose principle of induction requires to count the occurrences of the successor operator in the *normal form* of a given strongly normalizing term. This is not strictly necessary, and one can just count such symbols *directly* in the term, and so *avoid evaluating* it.

In summary, the novel contributions in this paper are: (1) a framework for the meta-theory of lambda calculi in FOAS with named variables and constants; (2) a complete mechanization of Girard's proof of The Strong Normalization Theorem for System T in Agda; (3) a new and different mechanization of Girard's proof for STLC in Agda as well, and; (4) a simplification of the principle of induction in Girard's original proof of The Strong Normalization Theorem for System T. To the best of our knowledge, there is not yet a mechanization of the Strong Normalization Theorem for System T. The development has been entirely written in Agda and it is available at: `https://github.com/surciuoli/lambda-c`.

The structure of this paper is the following. In the next section we introduce the new framework: its syntax, substitution, conversion theories and logical relations (reducible terms). Some results presented are completely new, and some others are an extension of [21, 24] to consider the additional syntax. From Section 2.5 on, and unless the opposite is explicitly stated, all results represent new developments. In Section 3, we formalize both STLC and Girard's proof of the Strong Normalization Theorem. In Section 4, we extend both the calculus and Girard's proof to System T, and we also explain the aforementioned simplification. In the last sections we give some overall conclusions and compare our work with related developments.

[3]In contrast to [11], during this development we shall not consider booleans nor tuples as part of the syntax. Nevertheless, they can be easily defined by the machinery presented here.

Throughout this exposition we shall use Agda code for definitions and lemmata, and a mix of code and English for the proofs in the hope of making reading more enjoyable. A certain degree of familiarity with Agda or at least with functional programming languages like Haskell is assumed.

2 The Framework

Let $\mathtt{V} = v_0, v_1 \ldots$ be any infinitely countable collection of names, the variables, ranged over by letters x, $y \ldots$ and equipped with a deciding procedure for definitional equality; for concreteness, we shall define $\mathtt{V} = \mathbb{N}$, i.e., the set of natural numbers in Agda, but it can be any other suitable type, e.g., strings. Let `C` be any possibly infinite countable collection of names, the constants, and ranged over by c. The abstract syntax of the lambda terms with constants is defined:

Definition 2.1 (Syntax)**.**

```
1 module CFramework.CTerm (C : Set) where
2 ...
3 data Λ : Set where
4   k : C → Λ
5   v : V → Λ
6   ƛ : V → Λ → Λ
7   _·_ : Λ → Λ → Λ
```

In line 1 we indicate that the definition is contained in the module `CFramework.CTerm`, which according to Agda's specification must be located in the file CFramework/CTerm.agda. We also specify that the module is parameterised by the set of constants `C`, which can be of any inductive type (`Set`). Lines 4 and 5 define the constructors for the constants and the variables respectively. In line 6 we use ƛ to not interfere with Agda's primitive λ. We shall follow the next convention unless the opposite is explicitly stated: use λ to represent *object-level* abstractions in informal discussions and proofs, and use ƛ in code listings. Line 7 defines the infix binary operator of function application. As usual, we shall use letters M, $N \ldots$ to range over terms.

The module can be then instantiated with any type of constants. For example, the next declaration derives the syntax of the pure lambda terms into the current scope:

Definition 2.2. `open import CFramework.CTerm ⊥`

$\bot$ is the inductive type without any constructor. The `import` statement tells Agda to load the content of the file named after the module into the current scope, while the `open` statement lets one access the definitions in it without having to qualify them. Both statements can be combined into a single one as shown.

Whenever a name x syntactically occurs in a term M and is not bound by any abstraction, we shall say x is free in M, and write it $x * M$. On the other hand, if every occurrence of x is bound by some abstraction (or even if x does not occur at all), we shall say x is fresh in M, and write it $x \# M$ as in nominal techniques, e.g., see [23]. Both relations are inductively defined in a standard manner, and in [21] it was proven that both relations are opposite to each other.

It will come in handy to define both the type of predicates and binary relations on terms respectively by: `Pred = Λ → Set`, and `Rel = Λ → Λ → Set`.

2.1 Substitution

Substitution is the fundamental entity on which alpha- and beta-conversion sit. We shall base ourselves upon the work done in [19], and first define multiple substitutions as functions from variables to terms:

```
Subst = V → Λ
```

We shall use letter σ to range over them. Later, by applying these functions to the free variables in a given term we shall obtain the desired operation of the *action of substitution* (Definition 2.6), i.e., the operation of replacing every free name x in M by its corresponding image σx.

Most substitutions appearing in properties and definitions are identity-almost-everywhere. We can generate them by starting from the identity substitution ι, which maps every variable to itself, and applying the update operation on substitutions `_≺+_` such that for any σ, x and M, `σ ≺+ (x , M)` is the substitution that maps x to M, and y to σy for every y other than x:

Definition 2.3 (Update operation)**.**

```
1  _≺+_ : Subst → V × Λ → Subst
2  (σ ≺+ (x , M)) y with x ≟ y
3  ... | yes _ = M
4  ... | no  _ = σ y
```

In line 1, × is the non-dependent product type, and in line 2, $\stackrel{?}{=}$ is the procedure that decides if two names are equal, and mentioned at the start of this section.

In some places we shall need to restrict the domain of a substitution so to have a finite image or range, therefore we introduce the type of restrictions, written `R`, and defined: `R = Subst × Λ`. Below we extend freshness to restrictions:

Definition 2.4 (Freshness on restrictions)**.**

```
_#⇂_ : V → R → Set
x #⇂ (σ , M) = (y : V) → y * M → x # σ y
```

In English, a name is fresh in the restriction `(σ , M)` if and only if it is fresh in every image σy, for every $y * M$.

Now we shall briefly discuss the mechanism in the framework used to rename the bound names in a given term, and so avoid capturing any free variable during the action of substitution. The complete description can be found in [21]. Let `χ'` be the function that returns the first name not in a given list:

```
χ' : List V → V
```

The algorithm is obtained by a direct consequence of the pigeonhole principle: the list of names given is finite, therefore we can always choose a fresh name from the infinite collection `V`. Then we can define the choice function χ that returns the first name not in a given restriction `(σ , M)`, by first concatenating into a single list every free name that appears in the image σx for every $x * M$, and then selecting the first name not in such a list by using the previous χ' function:

```
χ : R → V
χ (σ , M) = χ' (concat (mapL (fv ∘ σ) (fv M)))
```

`mapL` applies a function to every element in a list, ∘ stands for the usual composition of functions, and `fv` computes the list of free names in a given term. In [21] it was proven that χ computes a sufficiently fresh name, according to our expectations to be addressed shortly:

Lemma 2.5. `χ-lemma2 : (σ : Subst) (M : Λ) → χ (σ , M) #⇂ (σ , M)`

The action of a substitution σ on a term M is the operation that replaces every free name in M by its corresponding image under σ. It is written $M \bullet \sigma$ and defined:

Definition 2.6 (Action of substitution)**.**

```
_•_ : Λ → Subst → Λ
k c • σ = k c
v x • σ = σ x
M · N • σ = (M • σ) · (N • σ)
ƛ x M • σ = ƛ y (M • σ ≺+ (x , v y)) where y = χ (σ , ƛ x M)
```

Notice that in the last equation we always rename the bound variable x to y by using the χ function. We can show that this method avoids variable capture: for any $w * M$ other than x it must follow $y \# (\sigma \prec+ (x , y))w$, otherwise it would mean that we have captured an *undesired* free occurrence of y. Notice that if $w = x$ then its image is y which represents an occurrence of x in the original term λxM and therefore must be "re-bound". So, $x * M$ and $x \neq w$, therefore $w * \lambda xM$. Next, by Lemma 2.5 we have $y \# \downharpoonleft (\sigma , \lambda xM)$. Then, by Definition 2.4 it follows $y \# \sigma w$, and since $(\sigma \prec+ (x , y))w = \sigma w$ by Definition 2.3, so $y \# (\sigma \prec+ (x , y))w$.

Unary substitution is defined:

```
_[_/_] : Λ → Λ → V → Λ
M [ N / x ] = M • ι ≺+ (x , N)
```

Our definition of $\bullet$ has a direct consequence on the terms: when submitted to substitutions, the bound variables become "ordered", for the lack of a better name[4]. Consider the next example. Let $M = \lambda v_1 v_1$. By definition, $M \bullet \sigma = \lambda x(v_1 \bullet \sigma \prec+ (v_1 , x)) = \lambda xx$, where $x = \chi(\sigma , \lambda v_1 v_1)$, and for every σ. We can see that M does not contain any free variable, therefore by definition of χ we have that $x = v_0$, i.e., the first name in `V`, and so we have that the closed term $\lambda v_1 v_1$ turned into $\lambda v_0 v_0$ even though no substitution actually happened. Another example a bit more sophisticated is the next one: $(\lambda v_3 \lambda v_2 \lambda v_0 (v_0 v_1 v_2 v_3))[\, v_0 / v_1 \,] = \lambda v_1 \lambda v_2 \lambda v_3 (v_3 v_0 v_2 v_1)$. This collateral effect will have some implications on our definition of beta-reduction.

2.2 Alpha-conversion

Alpha-conversion is inductively defined by the syntax:

```
1 module CFramework.CAlpha (C : Set) where
2 open import CFramework.CTerm C
3 ...
4 data _∼α_ : Rel where
5   ∼k : {c : C} → k c ∼α k c
6   ∼v : {x : V} → v x ∼α v x
7   ∼· : {M M' N N' : Λ} → M ∼α M' → N ∼α N' → M · N ∼α M' · N'
8   ∼ƛ : {M M' : Λ} {x x' y : V} → y # ƛ x M → y # ƛ x' M'
9      → M [ v y / x ] ∼α M' [ v y / x' ] → ƛ x M ∼α ƛ x' M'
```

Since the syntax of the lambda terms is parameterised by a set `C`, every module that depends on the syntax (all of them) will have to be parameterised by `C` as well. Lines 1 and 2 illustrate this point.

[4]There is a similitude between this designation and A. Church's definition of *principal normal form* terms in [7, p. 348].

Arguments written between braces { and } are called *implicit* and they are not required to be supplied; the type-checker will infer their values, whenever possible. Implicit arguments can be made explicit by enclosing them between braces, e.g., `∼k {c₁}` has type `k c₁ ∼α k c₁`.

The only case in the definition worth mentioning is ∼ƛ. There, we rename both x and x' to a common fresh name y. If such results are alpha-convertible, then the choice of the bound name is irrelevant, and it should be expected to assert that both abstractions are alpha-convertible. This definition can also be seen in nominal techniques, e.g., see [23], though there it happens to be more usual to rename only one side of $\sim\alpha$. Our symmetrical definition has some advantages over those that are not (see [21]). Also, in [21], $\sim\alpha$ was proven to be an equivalence relation.

The next results are quickly extended from [21]:

Lemma 2.7. `lemma•ι : ∀ {M} → M ∼α M • ι`

Lemma 2.8. `corollary1SubstLemma : ∀ {x y σ M N} → y #⇂ (σ , ƛ x M)`
`→ (M • σ ≺+ (x , v y)) • ι ≺+ (y , N) ∼α M • σ ≺+ (x , N)`

Arguments preceded by ∀ are not required to be annotated with their respective types.

2.3 Reduction

Let ▷ be any binary relation on terms and called a contraction relation. The syntactic closure of ▷ is written ⇝ and it is inductively defined:

Definition 2.9.

```
1 import CFramework.CTerm as CTerm
2 module CFramework.CReduction (C : Set) (_▷_ : CTerm.Rel C) where
3 open CTerm C
4 ...
5 data _⇝_ : Rel where
6   abs : ∀ {x M N} → M ⇝ N → ƛ x M ⇝ ƛ x N
7   appL : ∀ {M N P} → M ⇝ N → M · P ⇝ N · P
8   appR : ∀ {M N P} → M ⇝ N → P · M ⇝ P · N
9   redex : ∀ {M N} → M ▷ N → M ⇝ N
```

Line 1 imports the module `CFramework.CTerm`, and at the same time renames it to `CTerm` just for convenience. Line 2 specifies that the module is parameterised by the contraction relation ▷; notice that since we have not opened the module `CTerm` nor specified the set of constants to be used, we wrote `CTerm.Rel C` (compare with line 5). From now until the end of this section, it is assumed that both `C` and ▷ are in the scope of every definition, unless explicitly stated the opposite.

Any term on the left-hand side of ▷ shall be called a redex, as usual, and any term on right-hand side a contractum. Besides, any term on the right-hand side of ⇝ shall be called a reductum.

We can define beta-reduction by means of ⇝ as next. Let beta-contraction be inductively defined:

Definition 2.10 (Beta-contraction)**.**

```
module CFramework.CBetaContraction (C : Set) where
...
data _▷β_ : Rel where
  beta : ∀ {x M N} → ƛ x M · N ▷β M [ N / x ]
```

Then beta-reduction for the pure lambda calculus is derived by importing the modules:

Definition 2.11 (Beta-reduction)**.**

```
open import CFramework.CBetaContraction ⊥
open import CFramework.CReduction ⊥ _▷β_ renaming (_⇝_ to _→β_)
```

Recall that in Definition 2.2 we had explained that by defining `C` $= \bot$ we obtain the syntax of the pure lambda terms.

The renaming done by • is sensitive to the free variables in the subject term. As a consequence, →β is not compatible with substitution, i.e., the next lemma *does not* hold:

```
∀ {M N σ} → M →β N → M • σ →β N • σ
```

Consider the following example. Let $M = \lambda v_1((\lambda v_0 \lambda v_0 v_0) v_0)$ and $N = \lambda v_1 \lambda v_0 v_0$. It can be seen that $M \to\beta\ N$ is derivable. Now, let us apply ι on each side. As to N, v_1 is renamed to the first name fresh in the restriction $(\iota\ ,\ \lambda v_1 \lambda v_0 v_0)$, i.e., to v_0; we obtain $N \bullet \iota = \lambda v_0 \lambda v_0 v_0$. As to M, the variable v_1 is renamed to itself, since it is the first fresh name in the corresponding restriction (renaming it to v_0 would cause a capture). So, $M \bullet \iota = M$, and the only reductum $\lambda v_1((\lambda v_0 v_0)\,[\,v_0\,/\,v_0\,])$ of M equals to $\lambda v_1 \lambda v_0 v_0$, which is not $N \bullet \iota$.

Since we are going to need some form of the lemma of compatibility above as we shall see, we will use the next approximation which is always possible: continuing with the earlier example, after the reduction takes place we shall perform an alpha-conversion step from the reductum to meet $N \bullet \iota$, i.e., $M \bullet \iota \to\beta\ \lambda v_1 \lambda v_0 v_0$ followed by $\lambda v_1 \lambda v_0 v_0 \sim\alpha\ N \bullet \iota$.

So, let r be any binary relation on terms, either a contraction relation or a reduction. We shall say r is alpha-compatible with substitution, and write it `Compat•` r, if and only if, for every directed pair of terms M and N related under r, there must exist some P such that $M \bullet \sigma$ and P are also related, and that $P \sim\alpha\ N \bullet \sigma$. Formally:

Definition 2.12 (Alpha-compatibility with substitution)**.**

```
Compat• r = ∀ {M N σ} → r M N → Σ[ P ∈ Λ ](r (M • σ) P × P ∼α N • σ)
```

In Agda, the dependent product type can be written `Σ[ a ∈ A ] B`, where `a` is some (meta-)variable of type `A`, and `B` is some type which might depend upon `a`.

Similarly, we shall say r is alpha-commutative and define it:

Definition 2.13 (Alpha-commutativity)**.**

```
Comm∼α r = ∀ {M N P} → M ∼α N → r N P → Σ[ Q ∈ Λ ](r M Q × Q ∼α P)
```

We shall restrict this development to contraction relations that preserve freshness, i.e., relations that do not introduce any free name in any contractum:

Definition 2.14. `Preserves# r = ∀ {x M N} → r M N → x # M → x # N`

Then we have that, if ▷ preserves freshness, or it is compatible with substitution, or it commutes with alpha-conversion, then its syntactic closure has the corresponding properties as well:

Lemma 2.15.

```
preser⇝# : Preserves# _▷_ → Preserves# (_⇝_ _▷_)
compat⇝• : Preserves# _▷_ → Compat• _▷_ → Compat• (_⇝_ _▷_)
commut⇝α : Preserves# _▷_ → Compat• _▷_ → Comm∼α _▷_ → Comm∼α (_⇝_ _▷_)
```

Their proofs are extended from [21, 24]. Notice the cascade effect on the lemmata: each of them has all the arguments of the one above. This happens naturally since each lemma relies on the previous one.

Finally, we have that beta-contraction is alpha-commutative, along with two other results (their proofs are extended from [24]):

Lemma 2.16. `Preserves# _▷β_ × Compat• _▷β_ × Comm∼α _▷β_`

2.4 Strongly normalizing terms

A term is strongly normalizing if and only if every reduction path starting from it eventually halts. We shall use their accessible characterization (originally presented in [3]). For any given computation relation ⇝ we define `sn`:

Definition 2.17 (Strongly normalizing terms)**.**

```
sn : Λ → Set
sn = Acc (dual _⇝_)
```

`Acc` is the type of the accessible elements by some order $<$, i.e., the set of elements a such that there is no infinite sequence $\ldots < a' < a$. It is defined in Agda's standard library [22]. `dual` is the function that returns the *type* of the inverse of every binary relation on terms. We use the dual of ⇝ instead of the direct because `Acc` expects an order that descends to its left-hand side, so to speak, which is not the case for ⇝. Line 2 can be read as: `sn` is the set of terms M such that $M \leadsto M' \leadsto \ldots$ is always finite. Below is the definition of `Acc` to support this paragraph:

```
data Acc {a b} {A : Set a} (_<_ : Rel A b) (x : A) : Set (a ⊔ b) where
  acc : (∀ y → y < x → Acc _<_ y) → Acc _<_ x
```

Note that `Rel` above is the type of binary relations between any two types, and it is defined in the standard library. `a` and `b` are universe indices or levels, and ⊔ is the function that returns the greatest of them.

The next result is adapted from [24] and follows easily by induction:

Lemma 2.18. `inversionSnApp : ∀ {M N} → sn (M · N) → sn M × sn N`

`sn` is closed under alpha-conversion, as long as the supporting relation ⇝ is alpha-commutative. The corresponding proof presented here is an adaptation of [24]:

Lemma 2.19. `closureSn∼α : Comm∼α _⇝_ → ∀ {M N} → sn M → M ∼α N → sn N`

Proof. By induction on the derivation of `sn`M. To derive `sn`N we need to prove `sn`P for any $N \leadsto P$. By Definition 2.13 there exists some Q such that $M \leadsto Q$ and $Q \sim\alpha\ P$. By Definition 2.17, `sn`Q holds, i.e., Q is accessible, and `sn`Q is a proper component of the derivation of `sn`P[5]. Then, we can use the induction hypothesis on `sn`Q together with $Q \sim\alpha\ P$ and obtain `sn`P. □

Exceptionally, we show the code of the proof above because it is very compact, and to reinforce the understanding of the principle of structural induction of `sn`:

```
closureSn∼α comm {M} {N} (acc i) M∼N =
  acc λ P P←N → let Q , M→Q , Q∼P = comm M∼N P←N
                in closureSn∼α comm (i Q M→Q) Q∼P
```

The λ occurrence denotes Agda's entity for meta-level lambda terms. `i Q M→Q` is of type `sn`Q, and it is a proper component of `acc i` which is of type `sn`M. `P←N` is of type `(dual _⇝_) P N` which in turn equals to `P ⇝ N`. For the same reason `M→Q` is of type `(dual _⇝_) Q M`.

[5]Put in other words, every reduction beginning in Q is at least one step shorter than every other reduction beginning in M.

2.5 Reducibile terms

Girard's proof of the Strong Normalization Theorem defines a relation between terms and types. A term that is related to some type is said to be reducible. The proof is carried out in two steps: first, it is proven that every reducible term is strongly normalizing, and secondly that every typed term is reducible. In this section we shall define the logical relation of reducible terms, and after that we shall prove some of their properties, including the first step in Girard's proof (`CR1` of Lemma 2.27).

Both in STLC and System T (object-level) types are simple, so regarding this development they will be enough for our definition of the logical relation. We define them by:

Definition 2.20 (Object-level types)**.**

```
data Type : Set where
  τ : Type
  _⇒_ : Type → Type → Type
```

The relation of reducible terms or logical relation is then defined by recursion on the types:

Definition 2.21 (Reducible terms)**.**

```
Red : Type → Λ → Set
Red τ M = sn M
Red (α ⇒ β) M = ∀ {N} → Red α N → Red β (M · N)
```

`Red` is closed under alpha-conversion:

Lemma 2.22 (Closure of `Red` under $\sim\alpha$)**.**

```
closureRed∼α : Comm∼α _⇝_ → ∀ {α M N} → Red α M → M ∼α N → Red α N
```

Proof. By induction on the type α, and by using Lemma 2.19. □

Next we have *neutral terms*. We shall use a different characterization than the one given in [11], and define them as the set of terms which when applied to any non-empty sequence of arguments, the result is never a redex, i.e., M is neutral if and only if $MN_1N_2\ldots N_n$ is not a redex for any $n > 0$.

So first, let us define the type of vectors of applications of terms:

Definition 2.23 (Vectors)**.**

```
data Vec : Λ → Λ → Set where
  nil : ∀ {M} → Vec M M
  cons : ∀ {M N} → Vec M N → ∀ {P} → Vec M (N · P)
```

`Vec M V` will then indicate that $V = MN_1N_2\ldots N_n$ for some $n \geq 0$, and we shall say that M is the head. If $n = 0$ then $M = V$, and M is not applied to any argument (we will see right away why this is convenient despite our motivation required $n > 0$).

Now we can give a precise characterization of the type of neutral terms:

Definition 2.24 (Neutral terms)**.**

```
Ne M = ∀ {V} → Vec M V → ∀ {P Q} → ¬ (V · P) ▷ Q
```

Note that we have added P at the end of `V · P` to have at least one argument applied to M.

The next result follows immediately by induction on the definition of `Vector`:

Lemma 2.25. `lemmaNe : ∀ {M} → Ne M → ∀ {N} → Ne (M · N)`

As to the main result in this section we have some properties about reducible terms, among them, the first part of Girard's proof, i.e., reducible terms are strongly normalizing (`CR1`). Let $\triangleright$ be any relation that does not reduce variables, and such that for any vector V with a variable at the head it follows V is neutral under $\triangleright$[6]; using our definition of vectors (possibly with no applications) we can compact both conditions by:

Definition 2.26 (Condition of $\triangleright$)**.**

```
Cond▷ = ∀ {x N} → Vec (v x) N → ∀ {Q} → ¬ N ▷ Q
```

Then, for any such a relation $\triangleright$ we have that:

Lemma 2.27 (Properties of reducible terms)**.**

```
CR1 : ∀ {α M} → Red α M → sn M
CR2 : ∀ {α M N} → Red α M → M ⇝ N → Red α N
CR3 : ∀ {α M} → Ne M → (∀ {N} → M ⇝ N → Red α N) → Red α M
```

Proof. By mutual induction on the type α:

- Case $\alpha = \tau$:

  `CR1` By Definition 2.21, $\texttt{Red}\,\tau M = \texttt{sn}M$, so `CR1` is a tautology.

  `CR2` Immediate by Definition 2.17.

  `CR3` Analogous to `CR2`.

- Case $\alpha = \beta \Rightarrow \gamma$:

  `CR1` By Definition 2.26 we have both that $\texttt{Ne}\,v_0$, and that $v_0 \rightsquigarrow N$ is absurd for any N. As a direct consequence of the latter, the second hypothesis or argument of `CR3` follows by vacuity, and so we can use the main induction hypothesis `CR3` and obtain $\texttt{Red}\,\beta\, v_0$. Now, by Definition 2.21 on $\texttt{Red}\,(\beta \Rightarrow \gamma)M$, we obtain $\texttt{Red}\,\gamma(Mv_0)$, and by the induction hypothesis $\texttt{sn}\,(Mv_0)$. Finally, by Lemma 2.18 we get $\texttt{sn}M$.

  `CR2` According to Definition 2.21, to prove the thesis $\texttt{Red}\,(\beta \Rightarrow \gamma)N$ we have to prove $\texttt{Red}\,\gamma(NP)$ for any $\texttt{Red}\,\beta\, P$. By Definition 2.21, again, the hypothesis $\texttt{Red}\,(\beta \Rightarrow \gamma)M$ tells us $\texttt{Red}\,\gamma(MP)$, and by the `appL` rule of Definition 2.9 on $M \rightsquigarrow N$ we can derive $MP \rightsquigarrow NP$, so we can use the induction hypothesis and obtain $\texttt{Red}\,\gamma(NP)$ as desired.

  `CR3` Let $\texttt{Red}\,\beta\, P$. To derive our desired result $\texttt{Red}\,\gamma(MP)$ and by using the induction hypothesis, we need to feed it with the required hypotheses or arguments: (1) $\texttt{Ne}\,(MP)$, and (2) that for every N', $MP \rightsquigarrow N'$ implies $\texttt{Red}\,\gamma N'$. (1) follows by Lemma 2.25. As to (2), first of all, by the main induction hypothesis `CR1` we get $\texttt{sn}P$. Now we continue by a nested induction on the derivation of $\texttt{sn}P$[7]. Let us analyse every possible derivation of $MP \rightsquigarrow N'$.

  - Case `redex`: $MP \triangleright N'$ is absurd by Definition 2.26.
  - Case `appL`: If $MP \rightsquigarrow M'N''$ follows from $M \rightsquigarrow M'$ with $N' = M'N''$ then by (2) we get $\texttt{Red}\,(\beta \Rightarrow \gamma)M'$, and so by Definition 2.21, $\texttt{Red}\,\gamma(M'N'')$.
  - Case `appR`: If $MP \rightsquigarrow MP'$ follows from $P \rightsquigarrow P'$ with $N' = MP'$, then by Definition 2.17 we obtain $\texttt{sn}P'$, which is a proper component of $\texttt{sn}P$, and so we can continue by induction on $\texttt{sn}P'$.

[6] Actually, we could have asked the second condition just for one specific variable and Lemma 2.27 would hold anyway (see the proof of `CR1` when α is functional).

[7] In the code, it means to define an auxiliary function in the current scope.

□

Next we have some general definitions regarding the assignment of types. First, there are contexts (of variable declarations). They are defined as list of pairs, possibly with duplicates:

Definition 2.28. `Cxt = List (V × Type)`

Then there is the relation of membership between variables and contexts. We shall write $x \in \Gamma$ and say that x is the *first* variable in Γ, searched from left to right. Below is the inductive definition:

```
data _∈_ : V → Cxt → Set where
  here  : ∀ {x α Γ} → x ∈ Γ ⋓ x : α
  there : ∀ {x y α Γ} → x ≢ y → x ∈ Γ → x ∈ Γ ⋓ y : α
```

`Γ ⋓ x : α` is syntax-sugar for `(x , α) :: Γ`. Finally, there is a lookup function on contexts such that it returns the type of the first variable (provided it is declared), searched in the same fashion, and defined:

```
_⟨_⟩ : ∀ {x} → (Γ : Cxt) → x ∈ Γ → Type
[]              ⟨ ()        ⟩
((k , d) :: xs) ⟨ here      ⟩ = d
((k , d) :: xs) ⟨ there _ p ⟩ = xs ⟨ p ⟩
```

In the second line, `()` is an *absurd* pattern, and it tells Agda to check that there is no possible way of having an object of type $x \in$ `[]`, for any x.

To end this section, we present reducible substitutions. We shall say a substitution is reducible under some context Γ if and only if it maps every variable in Γ to a reducible term of the same type:

Definition 2.29. `RedSubst σ Γ = ∀ x → (k : x ∈ Γ) → Red (Γ ⟨ k ⟩) (σ x)`

The next results follow immediately by definition:

Lemma 2.30. `Red-ι : ∀ {Γ} → RedSubst ι Γ`

Lemma 2.31.

`Red-upd : RedSubst σ Γ → ∀ x → Red α N → RedSubst (σ ≺+ (x , N)) (Γ ⋓ x : α)`

3 STLC

The syntax and theories of substitution, alpha- and beta-conversion for STLC are obtained by instantiating the framework with:

```
module STLC where
open import CFramework.CTerm ⊥
...
open import CFramework.CReduction ⊥ _▷β_ as Reduction renaming (_⇝_ to _→β_)
```

Next is the assignment of types in STLC:

```
data _⊢_:_ (Γ : Cxt) : Λ → Type → Set where
  ⊢var : ∀ {x} → (k : x ∈ Γ) → Γ ⊢ v x : Γ ⟨ k ⟩
  ⊢abs : ∀ {x M α β} → Γ ⋓ x : α ⊢ M : β → Γ ⊢ ƛ x M : α ⇒ β
  ⊢app : ∀ {M N α β} → Γ ⊢ M : α ⇒ β → Γ ⊢ N : α → Γ ⊢ M · N : β
```

3.1 The Strong Normalization Theorem in STLC

Following Girard's proof, first we need to prove that every reducible term is `sn`. We shall use `CR1` of Lemma 2.27 for that matter, so we need to prove that $\triangleright\beta$ satisfies conditions in Definition 2.26.

Lemma 3.1. `cond▷β : ∀ {x N} → Vec (v x) N → ∀ {Q} → ¬(N ▷β Q)`

Proof. Immediate by contradiction from `Vec (x N) N` and `N ▷β Q`. □

Then we can open the following modules and inherit Lemma 2.27 for STLC, particularly `CR1`:

```
open import CFramework.CReducibility ⊥ _▷β_ as Reducibility
open Reducibility.RedProperties cond▷β
```

Next we have to prove that every typed terms is reducible; we shall refer to this as the main lemma. To present the proof, we are going to need some preparatory results. First, by Lemma 2.16 together with Lemma 2.15 we have that $\rightarrow\beta$ is both alpha-compatible with substitution, and alpha-commutative:

Lemma 3.2. `Compat• _→β_ × Comm∼α _→β_`

Secondly, it is immediate that $(\lambda x N)N$ is neutral for every x, M and N:

Lemma 3.3. `lemmaβNe : ∀ {x M N} → Ne ((ƛ x M) · N)`

And finally, since the main lemma proceeds by induction on the derivation of the typing judgement, and the case of abstractions is quite complex, it turns out to be convenient to have a separate lemma for this:

Lemma 3.4. `lemmaAbs : ∀ {x M N α β} → sn M → sn N`
`→ (∀ {P} → Red α P → Red β (M [ P / x ])) → Red α N → Red β (ƛ x M · N)`

Proof. By induction on the derivations of $\texttt{sn}M$ and $\texttt{sn}N$. We shall refer to hypotheses $\texttt{sn}M$, $\texttt{sn}N$, $\forall\{P\} \rightarrow \texttt{Red}\,\alpha P \rightarrow \texttt{Red}\,\beta\,(M\,[\,P\,/\,x\,])$ and $\texttt{Red}\,\alpha N$ as (1) through (4) respectively. So, to use `CR3` of Lemma 2.27 to prove that the neutral term $(\lambda x M)N$ is reducible of type β (the thesis of this lemma) we need to show that every reductum is reducible (the second explicit hypothesis of the mentioned lemma). So, let us analyze every possible case:

- Case `redex`: If $(\lambda x M)N \rightarrow\beta\ M\,[\,N\,/\,x\,]$ then we can quickly derive that $M\,[\,N\,/\,x\,]$ is reducible from (3) and (4).
- Case `appL`: If $(\lambda x M)N \rightarrow\beta\ (\lambda x M')N$ follows from $M \rightarrow\beta\ M'$ then, to use the induction hypothesis on $\texttt{sn}M'$, we need to provide the requested hypotheses (1) through (4) correctly instantiated. (1) follows from Definition 2.17. (2) and (4) are direct. As to (3), we need to prove that $\texttt{Red}\,\beta\,(M'\,[\,P\,/\,x\,])$ holds for any $\texttt{Red}\,\alpha P$. By Lemma 3.2 we know that there exists some R such that $M\,[\,P\,/\,x\,] \rightarrow\beta\ R$ and $R \sim\alpha\ M'\,[\,P\,/\,x\,]$. By hypothesis (3) it follows $\texttt{Red}\,\beta\,(M\,[\,P\,/\,x\,])$, so by `CR2` of Lemma 2.27 we obtain $\texttt{Red}\,\beta R$. Finally, we can use Lemma 3.2 together with inherited Lemma 2.22 to derive $\texttt{Red}\,\beta\,(M'\,[\,P\,/\,x\,])$.
- Case `appR`: If $(\lambda x M)N \rightarrow\beta\ (\lambda x M)N'$ follows from $N \rightarrow\beta\ N'$ then, by Definition 2.17 we have $\texttt{sn}N'$, and by `CR2` of Lemma 2.27 we obtain $\texttt{Red}\,\alpha N'$, therefore we can use the induction hypothesis on $\texttt{sn}N'$ and derive $\texttt{Red}\,\beta\,((\lambda x M)N')$.

□

Now, to use the previous result in the main lemma, we are going to need a stronger induction hypothesis in order to derive the third argument, namely $\forall\{P\} \rightarrow \texttt{Red}\,\alpha P \rightarrow \texttt{Red}\,\beta\,(M\,[\,P\,/\,x\,])$. We shall see that by stating the main lemma as next we can easily derive it:

Lemma 3.5. `main : ∀ {α M σ Γ} → Γ ⊢ M : α → RedSubst σ Γ → Red α (M • σ)`

Proof. By induction on the typing derivation:

- Case ⊢`var`: If M is a variable, then the thesis follows directly from Definition 2.29.
- Case ⊢`abs`: If $M = \lambda x M'$ with type $\alpha \Rightarrow \beta$, then we need to show `Red`$\beta\,(((\lambda x M') \bullet \sigma) N)$ for any `Red`αN. First of all, $\lambda x M' \bullet \sigma = \lambda y (M' \bullet (\sigma \prec\!+ (x\,,\,y)))$ for some fresh name y. Now, to use Lemma 3.4 we need to derive its hypothesis: (1) `sn`$\,(M' \bullet \sigma \prec\!+ (x\,,\,y))$; (2) `sn`$N$; (3) for every `Red`$\alpha P$, `Red`$\beta\,((M' \bullet \sigma \prec\!+ (x\,,\,y))\,[\,P\,/\,y\,])$, and; (4) `Red`$\alpha N$. As to (1), by Lemma 2.31 we have `RedSubst`$\,(\Gamma \uplus x : \alpha)\,(\sigma \prec\!+ (x\,,\,y))$, thus by induction hypothesis `Red`$\beta\,(M' \bullet \sigma \prec\!+ (x\,,\,y))$, and so by `CR1` of Lemma 2.27 we obtain the desired result. (2) follows immediately by `CR1`. As to (3), first by Lemma 2.8 we have $(M' \bullet \sigma \prec\!+ (x\,,\,y))\,[\,P\,/\,y\,] \sim\!\alpha\; M' \bullet \sigma \prec\!+ (x\,,\,P)$. Next, by Lemma 2.31, `RedSubst`$\,(\Gamma \uplus x : \alpha)\,(\sigma \prec\!+ (x\,,\,P))$, so by the induction hypothesis we have `Red`$\beta\,(M'(\sigma, P/x))$. And finally, by Lemma 3.2 together with Lemma 2.22 we can derive the desired result. (4) is an assumption already made. At last, having (1) through (4) we can use Lemma 3.4 and derive `Red`$\,(\alpha \Rightarrow \beta)\,((\lambda x M') \bullet \sigma)$, and so obtain `Red`$\beta\,(((\lambda x M') \bullet \sigma) N)$ by Definition 2.21, as desired.
- Case ⊢`app`: Immediate by the induction hypothesis.

□

Without further ado, we have the Strong Normalization Theorem:

Theorem 3.6. `strongNormalization : ∀ {Γ M α} → Γ ⊢ M : α → sn M`

Proof. By Lemmas 2.30 and 3.5 we have `Red`$\,\alpha\,(M \bullet \iota)$, and so by `CR1` of Lemma 2.27, `sn`$\,(M \bullet \iota)$. Then, by Lemma 2.7, $M \bullet \iota \sim\!\alpha\; M$, and thus by Lemma 3.2 together with Lemma 2.19 it follows `sn`M. □

4 System T

Let `C` and `▷T` be inductively defined:

```
data C : Set where
  O : C; S : C; Rec : C
```

```
data _▷T_ : Rel where
  beta : ∀ {M N} → M ▷β N → M ▷T N
  recO : ∀ {G H} → k Rec · G · H · k O ▷T G
  recS : ∀ {G H N} → k Rec · G · H · (k S · N) ▷T H · N · (k Rec · G · H · N)
```

The syntax and theories of substitution, alpha- and beta-conversion for System T are then obtained by instantiating the framework with both `C` and `▷T`, and similarly to STLC as shown in the previous section.

The assignment of types in System T is extended from STLC and defined:

```
data _⊢_:_ (Γ : Cxt) : Λ → Type → Set where
  ⊢zro : Γ ⊢ k O : nat
  ⊢suc : Γ ⊢ k S : nat ⇒ nat
  ⊢rec : ∀ {α} → Γ ⊢ k Rec : α ⇒ (nat ⇒ α ⇒ α) ⇒ nat ⇒ α
  ⊢var : ∀ {x} → (k : x ∈ Γ) → Γ ⊢ v x : Γ ⟨ k ⟩
  ⊢abs : ∀ {x M α β} → Γ ⊎ x : α ⊢ M : β → Γ ⊢ ƛ x M : α ⇒ β
  ⊢app : ∀ {M N α β} → Γ ⊢ M : α ⇒ β → Γ ⊢ N : α → Γ ⊢ M · N : β
```

`nat` is syntax-sugar for τ.

4.1 The Strong Normalization Theorem in System T

The proof of the Strong Normalization Theorem for System T follows the same structure as the one for STLC: first, we have to prove that `▷T` satisfies condition in Definition 2.26 so to derive the first step in Girard's method, i.e., `CR1`. Then, we need to have a main lemma and reason by induction on the syntax (the typing judgment) to derive reducibiliy. Finally, the Strong Normalization Theorem for System T follows *exactly* as Theorem 3.6.

So, to start with, we have that `▷T` satisfies Definition 2.26 similar to STLC:

Lemma 4.1. `cond▷T : ∀ {x N} → Vec (v x) N → ∀ {Q} → ¬(N ▷T Q)`

Thus, we inherit Lemma 2.27 in System T, particularly `CR1`.

As to the second part, i.e., the main lemma, we have to consider only the additional syntax; the remaining cases follow identically. `0` and `S` are reducible by `CR3` (similar to v_0 in the proof of `CR1`). As to `Rec`, we shall follow the same strategy as in STLC and have a separate lemma, namely the recursion lemma. In the next section we cover this last case, while at the same time we present the announced simplification.

4.2 Recursion

In this section, first we explain the induction used in the proof of the recursion lemma as presented in [11] but using the terminology of our framework, then we present the simplification, and finally we formalize the proof.

We must prove that the neutral term $\texttt{Rec}\,G\,H\,N$ is reducible for any reducible terms G, H and N. First, we shall strengthen our induction hypothesis: by `CR1` we know that G, H and N are `sn`, so we can assume that these derivations are given as additional hypotheses. Also, we need some preparatory definitions: let $\nu(M)$, $\ell(M)$ and $\mathsf{nf}(M)$ be respectively the length of the longest reduction starting in M, the count of `S` symbols in M, and the normal form of the (strongly normalizing) term M. Now, to prove our thesis we shall proceed by induction on the *strict component-wise* order (henceforth, just component-wise order) on the 4-tuple[8] $(\texttt{sn}\,G, \texttt{sn}\,H, \nu(N), \ell(\mathsf{nf}(N)))$, where in $\texttt{sn}\,G$ and $\texttt{sn}\,H$ we shall use the structural order of `sn`, in $\nu(N)$ the complete order on natural numbers[9], and in $\ell(\mathsf{nf}(N))$ the structural order on natural numbers. As we did in Lemma 3.4, we are going to use `CR3` of Lemma 2.27 for the matter, and so we have to prove that every reductum of $\texttt{Rec}\,G\,H\,N$ is reducible. There are five cases: (1) $\texttt{Rec}\,G'\,H\,N$ with $G \rightarrow\beta\; G'$, (2) $\texttt{Rec}\,G\,H'\,N$ with $H \rightarrow\beta\; H'$, (3) $\texttt{Rec}\,G\,H\,N'$ with $N \rightarrow\beta\; N'$, (4) G with $N = \texttt{0}$, and (5) $H N' (\texttt{Rec}\,G\,H\,N')$ with $N = \texttt{S}\,N'$. As to (1) and (2), we can directly use the induction hypothesis on $\texttt{sn}\,G'$ and $\texttt{sn}\,H'$. As to (3), we can suspect that $\nu(N') < \nu(N)$, and so we can proceed likewise. (4) is a hypothesis. As to (5), it is immediate that $\ell(\mathsf{nf}(N')) < \ell(\mathsf{nf}(\texttt{S}N'))$.

We can simplify the induction schema used above by dispensing with `nf`, and instead proceed by induction on the component-wise order of the 3-tuple $(\texttt{sn}\,G, \texttt{sn}\,H, (\nu(N), \ell(N)))$, where in $\texttt{sn}\,G$ and $\texttt{sn}\,H$ we use the same order as above, but in $(\nu(N), \ell(N))$ we use the *lexicographic order* on tuples[10]. As to cases (1), (2) and (4), the induction is the same. As to (3), we have already assumed that $\nu(N') < \nu(N)$, so we can use the (lexicographic-based) induction hypothesis on $(\nu(N'), \ell(N'))$, and disregard if $\ell(N')$ goes off. Finally, as to (5), on the one hand, it is immediate that $\ell(N') < \ell(\texttt{S}N')$. On the other hand, we can also guess that $\nu(N') = \nu(\texttt{S}N')$, therefore we can proceed by induction on $(\nu(\texttt{S}N'), \ell(N'))$.

[8]The component-wise order on a n-tuple is given by: $a_i <_i b \Rightarrow (a_0 \ldots, a_i \ldots, a_n) <_i (a_0 \ldots, b \ldots, a_n)$ for any i, n and b.

[9]The complete order on natural numbers is the same as transitive closure of the structural order on them.

[10]The lexicographic order on a tuple is given by: $a < b \Rightarrow (a,c) < (b,d)$ and $b < c \Rightarrow (a,b) < (a,c)$ for any a, b, c, d.

Now, to formalize the recursion lemma based on the last induction schema, first we need to give some definitions, as usual. Next is the function that computes the list of reductio for any given term M, while at the same time proves it is *sound*, i.e., every element of the list is actually a reductum of M. We present it in two separate parts, first `redAux`, which as the name suggest, is an auxiliary function, and then `reductio` which is the complete and desired operation (we omit some code):

```
1  redAux : (M : Term) → List (Σ[ N ∈ Term ](M →β N))
2  redAux (ƛ x M · N)                = [ (M [ N / x ] , ...) ]
3  redAux (k Rec · G · H · k O)      = [ (G , ...) ]
4  redAux (k Rec · G · H · (k S · N)) = [ (H · N · (k Rec · G · H · N) , ...) ]
5  redAux _                          = []
6
7  reductio : (M : Term) → List (Σ[ N ∈ Term ](M →β N))
8  reductio (k _)   = []
9  reductio (v _)   = []
10 reductio (ƛ x M) = mapL (mapΣ (ƛ x) abs) (reductio M)
11 reductio (M · N) = redAux (M · N) ++ ... (reductio M) ++ ... (reductio N)
```

`mapΣ` is the function that given two other functions and a tuple, it applies each function to one of the components of the tuple. The purpose of the auxiliary function is to put together the cases of redexes, and apart from the `reductio` definition, so to have a cleaner treatment in the case of applications in the latter (see line 11).

The algorithm is also *complete*, i.e., it outputs all reductio of M, and its proof follows by induction on the derivation of any given reduction:

Lemma 4.2. `lemmaReductio : ∀ {M N} (r : M →β N) → (N , r) ∈' (reductio M)`

`∈'` is the standard relation of membership in lists.

We can use the list returned by `reductio` to develop an algorithm that computes our first ordinal ν, i.e., the length of the longest reduction beginning in some strongly normalizing term M given, by recursively computing such a result for every reductum of M, then selecting the longest one, and finally adding one for the first step. Notice that the length of longest path and the height of the derivation tree of sn$M$ are synonyms, so we shall use them interchangeably:

```
ν : ∀ {M} → sn M → ℕ
ν {M} (acc i) = 1 + max (mapL (λ{(N , M→N) → ν (i N M→N)}) (reductio M))
```

`max` is the function that returns the maximum element in a given list. The above definition is standard for computing the height of any inductive type, except for that `sn` has an *infinitary premise* [1, p. 13]. This means that we need to enumerate all possible applications to obtain every possible sub-tree. Since every term M has a finite number of redexes, so there can only be finitely many applications of the premise, i.e., reductions $M \to\beta\ N$ for some N, all of them being enumerated by the `reductio` algorithm, as proven in Lemma 4.2.

The height of sn$N$ equals to the height of sn(SN), as guessed at the start of this section. This is immediate since the prefix S does not add any redex to any reduction path:

Lemma 4.3. `lemmaSν : ∀ {M} (p : sn M) (q : sn (k S · M)) → ν p ≡ ν q`

Proof. By induction on either the derivation of p or q. □

Next we have that the height of $\mathtt{sn}M$ decreases after a computation step is consumed, or in other words, every (immediate) sub-tree of $\mathtt{sn}M$ is strictly smaller. The name of the lemma is `lemmaStep`ν, and its proof follows by properties of lists, and by using Lemma 4.2:

Lemma 4.4. `∀ {M N i} (p : sn M) → p ≡ acc i → (r : M →β N) → ν (i N r) < ν p`

Notice the *apparently* clumsy way it was stated. `i N r` is a proof of $\mathtt{sn}N$. To require such a proof as an argument would be inefficient since we already know $\mathtt{sn}M$ and $M \to\beta\ N$. Instead, by asking for the argument `p ≡ acc i` we can obtain the premise `i` of $\mathtt{sn}M$ (this can be easily supplied afterwards with the constructor of ≡, `refl`), and apply it to `N` and `r`, and so obtain a proof of $\mathtt{sn}N$.

Next is our second ordinal:

Definition 4.5. `ℓ : Term → ℕ` *is the function that counts the number of occurrences of the* `S` *symbol in any given term, and it is defined by recursion on the term.*

Finally, we have the recursion lemma. Let `<-lex` be the lexicographic order on tuples of ℕ. Then `Acc _<-lex_` is the type of pairs that are accessible by such an order; it is easy to prove that for any proof p of $\mathtt{sn}N$ for some N, it follows that $(\nu(p), \ell(N))$ is in the accessible part of the lexicographic order, hence such an argument can always be derived. Also, note that $\mathtt{Rec}\,G\,H\,N$ is neutral for any G, H and N. Then:

Lemma 4.6 (Recursion)**.**

```
lemmaRec : ∀ {α G H N} → sn G → sn H → (p : sn N) → Acc _<-lex_ (ν p , ℓ N)
→ Red α G → Red (nat ⇒ α ⇒ α) H → Red α (k Rec · G · H · N)
```

Proof. By induction on the derivations of $\mathtt{sn}G$ and $\mathtt{sn}H$, and on the lexicographic order of the tuple $(\nu(p), \ell(N))$[11]. As has already been said several times by now, we shall resort to CR3 of Lemma 2.27 for the matter. So let us fast-forward til the reductum analysis:

- Case `rec0`: If $\mathtt{Rec}\,G\,H\,\mathtt{0} \to\beta\ G$ then the result is a hypothesis.
- Case `recS`: If $\mathtt{Rec}\,G\,H\,(\mathtt{S}N) \to\beta\ HN(\mathtt{Rec}\,G\,H\,N)$ then, since we know both that $\nu(N) = \nu(\mathtt{S}N)$ by Lemma 4.3, and that $\ell(N) < \ell(\mathtt{S}N)$ by definition of ℓ, we can apply the induction hypothesis and so obtain $\mathtt{Red}\,\alpha\,(\mathtt{Rec}\,G\,H\,N)$. Finally, by Definition 2.21 on $\mathtt{Red}\,(\mathtt{nat} \Rightarrow \alpha \Rightarrow \alpha)\,H$ we obtain $\mathtt{Red}\,\alpha\,(HN(\mathtt{Rec}\,G\,H\,N))$.
- Case `appR`: If $\mathtt{Rec}\,G\,H\,N \to\beta\ \mathtt{Rec}\,G\,H\,N'$ follows from $N \to\beta\ N'$ then, by Lemma 4.4 we know that $\nu(N') < \nu(N)$, and so we can use the induction hypothesis to derive the desired result.
- Case `appL`: If the reduction follows from one either in G or H, then we can proceed directly by the induction hypothesis.

□

5 Related work

In this development we have encoded the lambda terms using first-order abstract syntax (FOAS). In contrast, other approaches use *higher-order abstract syntax* (HOAS) [17], i.e., binders and variables are

[11] In Agda, every function is structural recursive, and each one of them will successfully pass the type-checking phase if, put it simply, there exists a subset of the arguments such that for every recursive call in any of its definiens, at least one of the arguments is structurally smaller whilst the others remains the same. This is the same as saying that the induction is based on the component-wise order of any arrangement of such a subset, i.e., on a tuple made up of such arguments.

encoded using the same ones in the host language. These systems have the advantage that substitution is already defined. The first such mechanization of the theorem for STLC was presented in [10], and by using the ATS/LF logical framework [25]. However, the theory of (terminating) recursive functions using FOAS is more established, and there are plenty of programming languages that support them. This makes fairly easy to translate this mechanization to other system supporting standard principles of induction.

A second difference with existing work is that in this paper we have used named variables instead of *de-Bruijn indices* [5], e.g., in our framework the identity function can be written λxx for any x, while in the latter $\lambda 0$. Clearly, the former is visually more appealing, making it better suited for textbooks, needless to say it is the actual way programs are written. The main disadvantage is that we do not identify alpha-convertible terms, e.g., $\lambda v_0 v_0$ and $\lambda v_1 v_1$ are different objects, whereas by using indices there is only one possible representative for each class of alpha-convertible terms, and so it is not necessary to deal with alpha-conversion at all. To mention some renowned mechanizations of the theorem for STLC using this encoding: in [3] the author uses the LEGO system [18], and; in [2] two different mechanizations are presented, one in Agda and one in Coq [15].

As to System T, to the best of our knowledge there is not yet a mechanization of the Strong Normalization Theorem.

6 Conclusions

We have presented a framework for the formal meta-theory of lambda calculi in FOAS with constants, that does not identify alpha-convertible terms, and it is parameterised by a reduction schema. On top of it, we have built a complete mechanization of Girard's proof of the Strong Normalization Theorem for System T. In addition, we were able to include a simplification on the principle of induction of the original proof. Finally, we gave a new and different mechanization of the same method but for STLC.

In terms of size, the framework is $\sim$1800LOC long, counting import statements and the like, and of which $\sim$90LOC belong to the first part of Girard's proof, namely the reducibility properties. As to the mechanizations of the proofs for STLC and System T, they are about 70 and 260LOC long repectively.

The proof for STLC presented here is significantly shorter than that of previous works using the same framework. In [24], a proof of the Strong Normalization Theorem for STLC using Joachimski and Matthes' method was presented, and soon after, it was refactored to take alpha-conversion out of the syntactic characterization of the strongly normalizing terms. The final proof was $\sim$400LOC long. The mechanization presented here adds up to $\sim$160LOC, i.e., less than half the size. One of the main differences is that the closure of the accessibility definition of the strongly normalizing terms under alpha-conversion required just 3LOC, while its syntactical counterpart required about 100LOC.

Overall, during this work alpha-conversion was not much of a burden outside the framework. Once the machinery has been set up, just a handful of lemmas were used at specific locations. Beta-reduction was proven to be both alpha-commutative and alpha-compatible with substitution in Lemma 3.2, and after that, both results were used in Lemmas 3.4 and 3.5 and Theorem 3.6, along with Lemmas 2.7, 2.8, 2.19 and 2.22, all of them having been previously defined in the framework. Alpha-conversion was not used at all in the recursion lemma.

We hope that this paper can also serve as a tool to extend the proof method to related calculi and different host languages. The method we have presented uses simple techniques and it is rich in details, so hopefully it can be adjusted to different scenarios.

References

[1] Andreas Abel (2008): *Normalization for the Simply-Typed Lambda-Calculus in Twelf*. In Carsten Schürmann, editor: *Proc. LFM '04*, *ENTCS* 199, Elsevier, pp. 3–16, doi:10.1016/j.entcs.2007.11.009.

[2] Andreas Abel, Guillaume Allais, Aliya Hameer, Brigitte Pientka, Alberto Momigliano, Steven Schäfer & Kathrin Stark (2019): *POPLMark reloaded: Mechanizing proofs by logical relations*. *Journal of Functional Programming* 29, doi:10.1017/S0956796819000170.

[3] Thorsten Altenkirch (1993): *Constructions, Inductive Types and Strong Normalization*. Ph.D. thesis, University of Edinburgh. Available at `https://www.cs.nott.ac.uk/~psztxa/publ/phd93.pdf`.

[4] Hendrik P. Barendregt (1985): *The lambda calculus - its syntax and semantics*. *Studies in logic and the foundations of mathematics* 103, North-Holland.

[5] Nicolaas Govert de Bruijn (1972): *Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem*. *Indagationes Mathematicae* 75(5), pp. 381–392, doi:10.1016/1385-7258(72)90034-0.

[6] Arthur Charguéraud (2012): *The Locally Nameless Representation*. *Journal of Automated Reasoning* 49(3), pp. 363–408, doi:10.1007/s10817-011-9225-2.

[7] Alonzo Church (1936): *An Unsolvable Problem of Elementary Number Theory*. *American Journal of Mathematics* 58(2), pp. 345–363, doi:10.2307/2371045.

[8] Ernesto Copello, Nora Szasz & Álvaro Tasistro (2017): *Formal metatheory of the Lambda calculus using Stoughton's substitution*. *Theoretical Computer Science* 685, pp. 65–82, doi:10.1016/j.tcs.2016.08.025.

[9] Martín Copes, Nora Szasz & Álvaro Tasistro (2018): *Formalization in Constructive Type Theory of the Standardization Theorem for the Lambda Calculus using Multiple Substitution*. In Frédéric Blanqui & Giselle Reis, editors: *Proc. LFMTP '18*, *EPTCS* 274, Open Publishing Association, p. 27–41, doi:10.4204/eptcs.274.3.

[10] Kevin Donnelly & Hongwei Xi (2007): *A Formalization of Strong Normalization for Simply-Typed Lambda-Calculus and System F*. In Alberto Momigliano & Brigitte Pientka, editors: *Proc. LFMTP '06*, *ENTCS* 174, Elsevier, pp. 109–125, doi:10.1016/j.entcs.2007.01.021.

[11] Jean-Yves Girard, Paul Taylor & Yves Lafont (1989): *Proofs and Types*. Cambridge University Press.

[12] V. Kurt Gödel (1958): *Über Eine Bisher Noch Nicht Benützte Erweiterung des Finiten Standpunktes*. *Dialectica* 12(4), pp. 280–287, doi:10.1111/j.1746-8361.1958.tb01464.x.

[13] J. Roger Hindley & Jonathan P. Seldin (2008): *Lambda-Calculus and Combinators: An Introduction*, second edition. Cambridge University Press, doi:10.1017/CBO9780511809835.

[14] Felix Joachimski & Ralph Matthes (2003): *Short proofs of normalization for the simply-typed λ-calculus, permutative conversions and Gödel's T*. *Archive for Mathematical Logic* 42, pp. 59–87, doi:10.1007/s00153-002-0156-9.

[15] The Coq development team (2004): *The Coq proof assistant reference manual*. Available at `http://coq.inria.fr`.

[16] Ulf Norell (2007): *Towards a practical programming language based on dependent type theory*. Ph.D. thesis, Chalmers University of Technology. Available at `https://www.cse.chalmers.se/~ulfn/papers/thesis.pdf`.

[17] Frank Pfenning & Conal Elliott (1988): *Higher-Order Abstract Syntax*. In Richard L. Wexelblat, editor: *Proc. PLDI '88*, Association for Computing Machinery, p. 199–208, doi:10.1145/53990.54010.

[18] Robert Pollack (1994): *The Theory of LEGO*. Ph.D. thesis, University of Edinburgh. Available at `https://era.ed.ac.uk/handle/1842/504`.

[19] Allen Stoughton (1988): *Substitution revisited*. *Theoretical Computer Science* 59(3), pp. 317–325, doi:10.1016/0304-3975(88)90149-1.

[20] William W. Tait (1967): *Intensional Interpretations of Functionals of Finite Type I*. The Journal of Symbolic Logic 32(2), pp. 198–212, doi:10.2307/2271658.

[21] Álvaro Tasistro, Ernesto Copello & Nora Szasz (2015): *Formalisation in Constructive Type Theory of Stoughton's Substitution for the Lambda Calculus*. In Mauricio Ayala-Rincón & Ian Mackie, editors: *Proc. LSFA '14*, *ENTCS* 312, Elsevier, pp. 215–230, doi:10.1016/j.entcs.2015.04.013.

[22] The Agda development team (2011): *The Agda standard library*. Available at `https://github.com/agda/agda-stdlib`.

[23] Christian Urban, Andrew M. Pitts & Murdoch J. Gabbay (2004): *Nominal unification*. Theoretical Computer Science 323(1), pp. 473–497, doi:10.1016/j.tcs.2004.06.016.

[24] Sebastián Urciuoli, Álvaro Tasistro & Nora Szasz (2020): *Strong Normalization for the Simply-Typed Lambda Calculus in Constructive Type Theory Using Agda*. In Cláudia Nalon & Giselle Reis, editors: *Proc. LSFA '20*, *ENTCS* 351, Elsevier, pp. 187–203, doi:10.1016/j.entcs.2020.08.010.

[25] Hongwei Xi (2003): *Applied type system*. In Stefano Berardi, Mario Coppo & Ferruccio Damiani, editors: *Proc. TYPES '03*, *LNCS* 3085, Springer, pp. 394–408, doi:10.1007/978-3-540-24849-1_25.