A constructive formalisation of the Modular Strong Normalisation Theorem

Flávio L. C. de Moura[1], Daniel L. Ventura[2], Raphael S. Ramos[1], and Fabrício S. Paranhos[2]

[1] Departamento de Ciência da Computação, Universidade de Brasília, Brazil
flaviomoura@unb.br,raphael.soares.1996@gmail.com
[2] Instituto de Informática, Universidade Federal de Goiás, Brazil
daniel@inf.ufg.br,paranhos.s.f@gmail.com

Abstract Modularity is a desirable property of rewrite systems because it allows a combined system to inherit the properties of its components. Termination is not modular, nevertheless under certain restrictions modularity can be recovered. In this work we present a formalisation of the Modular Strong Normalisation Theorem in the Coq proof assistant. The formalised proof is constructive in the sense that it does not rely on classical logic, which is interesting from the computational point of view due to the corresponding algorithmic content of proofs.

1 Introduction

It is well-known that termination is not a modular property for rewrite systems [1]. A property P of reduction relation systems is modular if given two systems A and B, the property P holds for the combined system built from A and B whenever P holds for both A and B. In other words, the union of terminating rewrite systems is not necessarily terminating. Nevertheless, under certain restrictions, modularity of termination can be recovered [2].

On the other hand, the preservation of strong normalisation property (PSN) is known to be not satisfied for some calculi with explicit substitutions [3, 4]. A calculus with explicit substitutions [5–7] presents some formalisation for the substitution operation, defined as a meta-operation in the λ-calculus. In other words, for rule $(\lambda x.M)N \to_\beta M[x := N]$ in such a calculus, $M[x := N]$ represents a term where the substitution is defined through a small-step semantics. PSN property in this context guarantees that any strongly normalising, i.e. terminating, term in the λ-calculus is also strongly normalising in the calculus with explicit substitutions.

The Modular Strong Normalisation Theorem states the conditions for the union of two reduction relations over a set A to be PSN through its relation of simulation to a reduction relation over some set B (cf. [7, 8]). In particular, when the reduction relation over B is terminating and the simulation relation is complete, the theorem guarantees that both reductions over A are terminating and so its union, i.e. that termination is modular.

We present in this work a constructive proof of the Modular Strong Normalisation Theorem in the Coq Proof Assistant [9]. The proof is entirely constructive in the sense that no classical reasoning is used, i.e. the law of excluded middle, proofs by contradiction or any equivalent inference rule. This is interesting from the computational point of view since the algorithmic content of proofs can automatically be extracted as certified code [10]. Eventhough no code extraction is executed in the present work, it takes part in a project to certify the proofs of [7], aiming to extract certified code. The choice of Coq as the formalisation tool is natural since the underlying logic behind the calculus of inductive constructions, the theory over which Coq is developed, is also constructive [11, 12].

A constructive proof of the Modular Strong Normalisation Theorem is presented by S. Lengrand in [8] and some of the basic notions used in this proof, such as strong normalisation, is already formalised in Coq [13]. In a certain sense, this work can be seen as a non-trivial expansion of the normalisation theory formalised by Lengrand. In fact, the strong normalisation property defined in [8] uses a specialized inductive principle that should hold for all predicate, i.e. through a second order formula. On the other hand, in this work we use only the standard inductive definition of the strong normalisation property (cf. [7, 8, 14]), and we also prove the equivalence between these definitions. In this way, we understand that we achieved a simpler and straightforward formalisation. The proof of the Modular Strong Normalisation Theorem follows the ideas in Lengrand's PhD thesis, but to the best of our knowledge, this is the first formalisation of this theorem.

The contributions of this work are summarised below.

– We provide a complete formalisation of the constructive normalisation theory based on the simulation technique developed in [8]. In particular:
  - We built a constructive proof of the Modular Strong Normalisation Theorem, and
  - We proved the equivalence between Lengrand's definition of strong normalisation and the standard inductive definition of strong normalisation.

This paper is built up from a Coq script where some code is hidden for the sake of clarity of this document. The formalisation is compatible with Coq 8.8.0. All the files concerning this work are freely available in the repository `https://github.com/flaviodemoura/MSNorm`.

2 The Modular Strong Normalisation Theorem

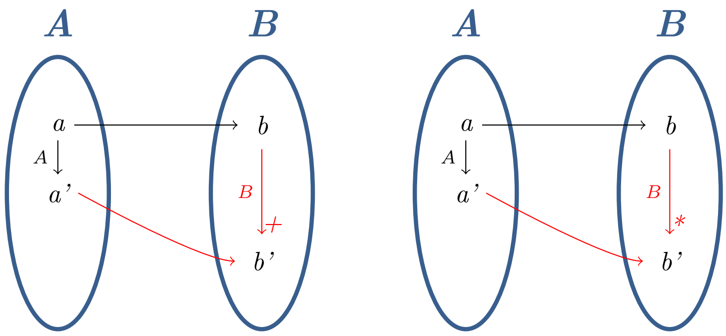
In this section, we present the Modular Strong Normalisation Theorem whose formalisation is detailed in the next section. This is an abstract theorem about termination of reduction relations through the well-known simulation technique [15]. In order to fix notation, let A and B be sets. A relation from A to B is a subset of $A \times B$. If R is a relation from A to B then we write $R\ a\ b$ instead of $(a, b) \in R$ and, in this case, we say that a *reduces* to b or that b is a R-*reduct* of a. Using arrows to denote relations and $\to$ being a relation from A to B then $\leftarrow$ denotes the inverse relation from B to A. If $\to_1$ is a relation from A to B and $\to_2$ is a relation from B to C, then the composition of $\to_1$ with $\to_2$, written $\to_1 \# \to_2$, is a relation from A to C. A relation from a set to itself is a *reduction relation over a set*, i.e. a reduction relation over A is a subset of $A \times A$. If $\to_A$ is a reduction relation over A, then a *reduction sequence* is a sequence of the form $a_0 \to_A a_1 \to_A a_2 \to_A \ldots$ A reduction sequence $a_0 \to_A a_1 \to_A a_2 \to_A \ldots \to_A a_n$ $(n \geq 0)$ is a n-step reduction from a_0. A reduction sequence is finite if it is a n-step reduction for some $n \in \mathbb{N}$, and infinite otherwise. We write $\to_A^+$ (resp. $\to_A^*$) for the transitive (resp. reflexive transitive) closure of $\to_A$.

An element $a \in A$ is *strongly normalising* w.r.t. $\to_A$ if every reduction sequence starting from a is finite, and in this case we write $a \in SN^{\to_A}$. Usually, this idea is expressed inductively as follows:

$$a \in SN^{\to_A} \text{ iff } \forall b, (a \to_A b \text{ implies } b \in SN^{\to_A}) \tag{1}$$

In order to present the theorem, we need to define the notions of strong and weak simulation. In the following definitions A and B are arbitrary sets:

Definition 1. *Let $\to$ be a relation from A to B, $\to_A$ be a reduction relation over A and $\to_B$ be a reduction relation over B. The reduction relation $\to_B$ strongly* *(resp. weakly) simulates* $\to_A$ *through* $\to$ *if* $(\leftarrow \# \to_A) \subseteq (\to_B^+ \# \leftarrow)$ *(resp.* $(\leftarrow \# \to_A) \subseteq (\to_B^* \# \leftarrow)$*).*



In what follows, we present the Modular Strong Normalisation Theorem and a draft of its proof. In addition, we highligth in blue the names of the corresponding results established in the formalisation, detailed in the next section.

Theorem 2 (Modular Strong Normalisation Theorem).
Let $\to$ *be a relation from* A *to* B*,* $\to_1$ *and* $\to_2$ *be two reduction relations over* A*, and* $\to_B$ *be a reduction relation over* B*. Suppose that:*

1. $\to_B$ *strongly simulates* $\to_1$ *through* $\to$*;*
2. $\to_B$ *weakly simulates* $\to_2$ *through* $\to$*;*
3. $A \subseteq SN^{\to_2}$*.*

Then $\leftarrow (SN^{\to_B}) \subseteq SN^{\to_1 \cup \to_2}$*. In other words,*

$$\forall a : A, a \in \leftarrow (SN^{\to_B}) \text{ implies } a \in SN^{\to_1 \cup \to_2}.$$

Proof. This proof follows the lines of [8], but using the standard SN definition in (1). First of all, hypothesis 1 and 2 allow us to conclude that the composition $(\to_2^* \# \to_1)$ is strongly simulated by $\to_B$: in fact, from hypothesis 2 we have that $\to_2^*$ is weakly simulated by $\to_B$ (*SimulWeakReflTrans*). In addition, the composition of two reduction relations that are, respectively, strongly and weakly simulated by the same reduction relation is strongly simulated by this reduction relation (*WeakStrongSimul*). Therefore, $(\to_2^* \# \to_1)$ is strongly simulated by $\to_B$ through $\to$ (*RCSimul*), that together with the fact that $a \in \leftarrow (SN^{\to_B})$ allow us to conclude that $a \in SN^{\to_2^* \# \to_1}$ (*SNbySimul*). Now, from hypothesis 3, we have $a \in SN^{\to_2}$, and we conclude from the fact that $SN^{\to_2^* \# \to_1} \cap SN^{\to_2} = SN^{\to_1 \cup \to_2}$ (*SNunion*). □

3 The Formalisation

In this section we present the formalisation details of the Modular Strong Normalisation Theorem in the Coq Proof Assistant. The first important point is that our proof is constructive, i.e. it does not use classical reasoning such as the law of excluded middle, double negation elimination or proof by contradiction.

In terms of notation, sets are coded as arbitrary types in such a way that the membership relation $a \in A$ (a is an element of the set A) is represented as $a : A$ (a has type A). Also, n-ary predicates and functions are defined in a curryfied version (cf. [16]).

We start with some basic definitions in order to make Coq notation clear[3]. A relation from A to B is defined as a binary predicate:

`Definition` *Rel* (A B : `Type`) := $A \to B \to$ `Prop`.

In this definition, *Rel* receives two types as arguments, and return the signature of a relation from A to B, i.e. the type $A \to B \to$ `Prop`. As mentioned before, if $A = B$ then we have a *reduction relation*:

`Definition` *Red* (A : `Type`) := *Rel* A A.

Given two relations *R1* and *R2* from A to B, if every pair of elements in *R1* is also in *R2* then we say that *R1* is a subrelation of *R2*:

`Definition` *Sub* {A B} (*R1* *R2*: *Rel* A B) : `Prop` := $\forall$ a b, *R1* a b $\to$ *R2* a b.

In the above definition, A and B first appear between curly brackets, meaning that these arguments are *implicit*. Implicit arguments are the types of polymorphic functions which can be inferred from the context. Therefore, *Sub* requires two relations as explicit arguments while Coq automatically infers its type. More convenient notations can be easily defined for objects we are constructing. For instance, in the *Sub* predicate case we define an infix notation as follows:

`Notation` "R1 <# R2" := (*Sub* *R1* *R2*) (`at level` 50).

Now one can write *R1* <# *R2* instead of *Sub* *R1* *R2*. In addition, in order to avoid parsing ambiguity, a precedence level ranging from 0 to 100 can be provided.

Given two relations, say *red1* from A to B and *red2* from B to C, one can build a new relation from A to C by composing its steps:

`Inductive` *comp* {A B C} (*red1*: *Rel* A B)(*red2*: *Rel* B C) : *Rel* A C :=
  compose: $\forall$ b a c, *red1* a b $\to$ *red2* b c $\to$ *comp* *red1* *red2* a c.

`Notation` "R1 # R2" := (*comp* *R1* *R2*) (`at level` 40).

Note that *comp* is inductively defined with just one constructor, named *compose*, that explicitly builds the composite relation from A to C from the given relations *red1* and *red2*. In addition, it is important to know that Coq automatically generates an inductive principle for every inductive definition. For instance, the natural numbers `nat` are inductively defined as:

```
Inductive nat : Set :=
  O : nat | S : nat → nat
```

[3] This paper is written directly from a Coq script file, therefore, the Coq code presented is the real code of the formalisation.

The corresponding induction principle, named `nat_ind`[4], is given by

```
forall P : nat → Prop, P 0 →
   (forall n : nat, P n → P (S n)) → forall n : nat, P n
```

Therefore, in order to prove that a certain property `P` holds for all `n: nat`, one needs to prove that `(P 0)` holds, and that if `(P n)` holds then `(P (S n))` also holds. In general, if `def` is an inductive definition with constructors `c1`, `c2`, ..., `ck` then, in order to prove that a property `P` holds for every element defined by `def`, we need to show, in a certain sense, that `P` is closed for each of its constructors. For a more precise and detailed explanation about Coq induction principles see [9, 17–19].

The inverse of a relation from A to B is defined by induction as the corresponding relation from B to A:

`Inductive` *inverse* $\{A\ B\}$ (R: *Rel* $A\ B$) : *Rel* $B\ A$:=
  inverseof: $\forall\ a\ b$, $R\ a\ b \rightarrow$ *inverse* $R\ b\ a$.

The transitive closure of a reduction relation `red` over A is constructed, as usual, by adding to `red` all possible reductions with at least one step starting from each $a \in A$:

`Inductive` *trans* $\{A\}$ (`red`: *Red* A) : *Red* A :=
| *singl*: $\forall\ a\ b$, `red` $a\ b \rightarrow$ *trans* `red` $a\ b$
| *transit*: $\forall\ b\ a\ c$, `red` $a\ b \rightarrow$ *trans* `red` $b\ c \rightarrow$ *trans* `red` $a\ c$.

Therefore, it is straightforward from this definition that a reduction relation is included in its transitive closure:

`Lemma` *transSub* $\{A$:`Type`$\}$ (`red`: *Red* A) : `red` <# (*trans* `red`).

The reflexive transitive closure of a reduction relation is obtained from its transitive closure just adding reflexivity, i.e. by adding the fact that each element of the set reduces to itself (in 0 steps):

`Inductive` *refltrans* $\{A\}$ (`red`: *Red* A) : *Red* A :=
| *reflex*: $\forall\ a$, *refltrans* `red` $a\ a$
| *atleast1*: $\forall\ a\ b$, *trans* `red` $a\ b \rightarrow$ *refltrans* `red` $a\ b$.

The image of a predicate via a relation is inductively defined as follows:

`Inductive` *Image* $\{A\ B\}$ (R:*Rel* $A\ B$)(P: $A \rightarrow$ `Prop`): $B \rightarrow$ `Prop` :=
  image: $\forall\ a\ b$, $P\ a \rightarrow R\ a\ b \rightarrow$ *Image* $R\ P\ b$.

The notions of weak and strong simulation for reduction relations are a straightforward translation to the Coq syntax (cf. Definition 1):

`Definition` *WeakSimul* $\{A\ B\}$ (*redA*: *Red* A) (*redB*: *Red* B) (R: *Rel* $A\ B$) :=
  ((*inverse* R) # *redA*) <# ((*refltrans redB*) # (*inverse* R)).

`Definition` *StrongSimul* $\{A\ B\}$ (*redA*: *Red* A) (*redB*: *Red* B) (R: *Rel* $A\ B$) :=
  ((*inverse* R) # *redA*) <# ((*trans redB*) # (*inverse* R)).

[4] The name of the automatic induction principle generated follows the pattern `name_ind`, i.e. the name of the inductive definition followed by the string `_ind`.

3.1 Equivalence between strongly normalising definitions

In this section, we prove the equivalence between Lengrand's definition of strong normalisation, denoted by *SN*, and the inductive definition presented in (1), here denoted by *SN'*. In his PhD thesis, Lengrand develops a constructive theory of normalisation in the sense that it does not rely on classical logic. In this theory, the notion of strong normalisation for reduction relations is defined by a second-order formula based on a stability predicate called *patriarchal* (cf. [8, 20]).

```
Definition patriarchal {A} (red:Red A) (P:A → Prop): Prop :=
  ∀ x, (∀ y, red x y → P y) → P x.
```

In this way, one says that a predicate P over A is patriarchal w.r.t. a reduction relation `red` over A, if $(P\ a)$ holds whenever $(P\ b)$ holds for every `red`-reduct b of a. Now, an element a is strongly normalising w.r.t. to the reduction relation `red`, when $(P\ a)$ holds for every patriarchal predicate P w.r.t. reduction relation `red`:

```
Definition SN {A:Type} (red:Red A) (a:A): Prop :=
  ∀ P, patriarchal red P → P a.
```

Most of the Coq code presented so far can be found at [13]. Nevertheless, our proof code is different since library *ssreflect* is not used in the present development.

The definition bellow corresponds to the usual inductive definition of strong normalisation for reduction relations, given in (1):

```
Inductive SN' {A:Type} (red: Red A) (a:A): Prop :=
  sn_acc: (∀ b, red a b → SN' red b) → SN' red a.
```

So, given an element a:A and a reduction relation `red` over A, a is strongly normalising w.r.t. `red` if every one-step `red`-reduct b of a is strongly normalising w.r.t. `red`. This means that in order to conclude *SN'* `red` a, one has to prove first that (∀ b, `red` a b → *SN'* `red` b). In addition, note that predicate *SN'* `red` is patriarchal hence it is straightforward that *SN* implies *SN'*, i.e. that *SN'* `red` a holds whenever (*SN* `red` a) holds. This inductive definition gives only one direction of the biconditional in (1), but the other direction is straightforward:

```
Lemma SNstable {A} {red: Red A}: ∀ a, SN' red a →
                                   ∀ b, red a b → SN' red b.
Proof.
  intros a HSN b Hred.
  inversion HSN; clear HSN.
  apply H; assumption.
Qed.
```

This proof analyses definition *SN'* in order to match the hypothesis *SN'* `red` a, labelled *HSN*, through the `inversion` tactic, that (informally) replaces hypothesis (*SN'* `red` a) by the information it contains. In this case, the known information comes from *SN'* definition and exactly what we need to prove.

The induction principle automatically generated for *SN'*, called *SN'_ind*, is as follows:

```
forall (A : Type) (red : Red A) (P : A -> Prop),
   (forall c : A, (forall b : A, red c b -> SN' red b) ->
   (forall b : A, red c b -> P b) -> P c) ->
   forall a : A, SN' red a -> P a
```

Then, to conclude that some property P holds for any strongly normalising element a, we need to prove that P holds for any strongly normalising c, given it holds for every `red`-reduct b of c. In other words, we need to prove that P is patriarchal and holds for every strongly normalising `red`-reduct of c.

Equivalence between definitions *SN* and *SN'* is an important contribution of this work, we thus comment the proof steps in order to explain it in more detail. Comments are given in blue just after proof commands they refer to. Note that type A and a reduction relation R over A are given as implicit arguments, i.e. they are inferred from the context.

`Theorem` *SN'EquivSN* $\{A$:`Type`$\}$ $\{R : \mathit{Red}\ A\}$: $\forall$ t, *SN'* R $t \leftrightarrow$ *SN* R t.
`Proof`.

`intro` t; `split`. These proof commands introduces a new skolem constant t to the proof context and splits the bi-implication in two steps. This means we are considering t to be an arbitrary element of set A, or more precisely, let t be an element of type A.

\- `intro` *HSN'*. The first implication to be proved is *SN'* R $t \rightarrow$ *SN* R t, so we assume *SN'* R t and we label this assumption as *HSN'*.

`apply` *SN'_ind* `with` R. We proceed by induction on the hypothesis *HSN'*. This corresponds to the application of the induction principle *SN'_ind* as explained above, in which reduction relation `red` is instantiated with R, and predicate P, with (*SN* R).

\+ `intros` a *HredSN'* *HredSN*. We call *HredSN'* (resp. *HredSN*) the hypothesis $\forall$ $b : A$, R a $b \rightarrow$ *SN'* R b (resp. $\forall$ $b : A$, R a $b \rightarrow$ *SN* R b).

`clear` *HredSN'* *HSN'*. Essentially, we need to prove the predicate (*SN* R) is patriarchal, which can be proved from the hypothesis *HredSN*. We then remove the unnecessary hypothesis depending on *SN'*.

`unfold` *SN* `in` *. Unfolding the definition *SN*, we need to prove that a holds for all R-patriarchal predicates.

`intros` P *Hpat*. Let P be a patriarchal predicated.

`apply` *Hpat*. Since P is patriarchal, we have that it holds for any R-reduct of a.

`intros` b *Hred*. Let b be an R-reduct of a.

`apply` *HredSN*; `assumption`. Therefore, we have that a R-reduces to b and P is R-patriarchal, which is exactly the content of the hypothesis *HredSN*.

\+ `assumption`. We conclude stating (*SN'* R t), which is an initial hypothesis.

- `intro` *HSN*. On the other direction, suppose that (*SN r t*), and call this hypothesis *HSN*. We need to prove that (*SN' R*) is *R*-patriarchal.

`apply` *HSN*. Now we can instantiate the universally quantified predicate of the definition of *SN* with (*SN' R*). Proving that (*SN' R*) is patriarchal corresponds to prove that $\forall\ x : A,\ (\forall\ y : A,\ R\ x\ y \rightarrow SN'\ R\ y) \rightarrow SN'\ R\ x$.

`intros` *x HSN'*. So, let *x* be an arbitrary element such that (*SN' R*) holds for every *R*-reduct of *x*. Call this fact *HSN'*.

`apply` *sn_acc*; `assumption`. Now, a proof of (*SN' R x*) corresponds, by definition, to a proof that every *R*-reduct *y* of *x* is such that (*SN' R y*) , which is exactly the content of the hypothesis *HSN'*. `Qed`.

3.2 The Main Theorem

In this section, we present the formal proof main steps of the Modular Strong Normalisation Theorem, including some results the proof depends. The first result concerns the composition of weakly and strongly simulated reductions. More precisely, if a reduction relation *redB* weakly simulates a reduction relation *redA1* through *R* and strongly simulates the reduction relation *redA2* through *R*, then *redB* strongly simulates the composition (*redA1* # *redA2*) through *R*. Although intuitively clear, the proof requires a large amount of details we decided to explain.

`Lemma` *WeakStrongSimul* {*A B*} (*redA1 redA2*:*Red A*)(*redB*:*Red B*)(*R*:*Rel A B*):
  WeakSimul redA1 redB R →
  StrongSimul redA2 redB R →
  StrongSimul (*redA1* # *redA2*) *redB R*.
`Proof`.

`intros` *Hweak Hstrong*. Let *Hweak* (resp. *Hstrong*) be the statement that *redB* weakly (resp. strongly) simulates the reduction relation *redA1* (resp. *redA2*) through *R*.

`unfold` *StrongSimul* `in` *. By definition of strong simulation, the composition (*inverse R*) # *redA2* is a subrelation of the transitive closure of *redB* composed with (*inverse R*). In addition, we have to prove that the composition (*inverse R*) # (*redA1* # *redA2*) is a subrelation of the transitive closure of *redB* composed with (*inverse R*).

`unfold` *WeakSimul* `in` *. By definition of weak simulation the composition (*inverse R*) # *redA1* is a subrelation of the transitive reflexive closure of *redB* composed with (inverse R).

`unfold` *Sub* `in` *. Therefore, every pair of elements *a* and *b* that are related by (*inverse R*) # (*redA1* # *redA2*) is also related by *trans redB* # (*inverse R*).

`intros` *a b Hcomp*. Let *Hcomp* be the hypothesis (*inverse R* # (*redA1* # *redA2*)) *a b*, i.e. *a* and *b* are related by the relation (*inverse R*) # (*redA1* # *redA2*).

inversion *Hcomp*; subst. clear *Hcomp*. From the hypothesis *Hcomp* there exists an element *b0* such that (*inverse R*) *a b0* , call this fact *H*, and (*redA1* # *redA2*) *b0 b*, which we call *H0*.

inversion *H0*; subst. clear *H0*. Similarly, the hypothesis *H0* means there exists an element *b1* such that *RedA1 b0 b1* and *redA2 b1 b*.

assert (*H'*: (*inverse R* # *redA1*) *a b1*).
{ apply *compose* with *b0*; assumption. } Therefore, from *H* and *H1* we get ((*inverse R*) # *redA1*) *a b1*, call this fact *H'*.
apply *Hweak* in *H'*. Since the reduction relation *redA1* is weakly simulated by *redB* through *R*, we get ((*inverse R*) # *redA1*) *a b1*

inversion *H'*; subst. clear *H'*. Which, in turn means that there exists an element *b2* such that *refltrans redB a b2* and (*inverse R*) *b2 b1*

induction *H0*. We proceed by induction on the reflexive transitive closure of *redB*, i.e. on the hypothesis *refltrans redB a b2*. The proof is split in two cases corresponding to the two constructors of the definition of the reflexive transitive closure of a reduction relation. The first case corresponds to the reflexive constructor.

- apply *Hstrong*. The strong simulation hypothesis allows us to prove (*trans redB* # (*inverse R*)) *a b* by showing that ((*inverse R*) # *redA2*) *a b*.

apply *compose* with *b1*; assumption. This composition can be constructed with the element *b1* given above.

- assert (*Hcomp*: (*trans redB* # *inverse R*) *b2 b*).
{ apply *Hstrong*. apply *compose* with *b1*; assumption. }
The second case of the induction, is proved by first observing that (*trans redB* # (*inverse R*)) *b2 b* , which can be proved from the strong simulation hypothesis *Hstrong* and noting that *b2* is related to *b* through the relation (*inverse R*) # *redA2* via the element *b1* above.
inversion *Hcomp*; subst. clear *Hcomp*. Therefore, there exits an element *b3* such that *trans redB b2 b3* and (*inverse R*) *b3 b*.

apply *compose* with *b3*. Hence, the element *a* is related to *b* through *b3* because

+ apply *tailtransit* with *b2*; assumption. *a* is related to *b3* through the relation *trans redB* via the element *b2*.

+ assumption. And *b3* is related to *b* through (*inverse R*). Qed.

The next result is a consequence of lemma *WeakStrongSimul*. In fact, it is easy to prove that if *redA* is weakly simulated by *redB* through *R* then so is its reflexive transitive closure(cf. lemma *SimulWeakReflTrans* in the formalisation source code). Then, by lemma *WeakStrongSimul* the composition of (*refltrans redA*) with *red'A*, a strongly simulated reduction relation, is also strongly simulated by *redB* through *R*.

Corollary *RCSimul* {*A B*} {*redA red'A*: *Red A*} {*redB*: *Red B*} {*R*: *Rel A B*}:

(*StrongSimul red'A redB R*) →
(*WeakSimul redA redB R*) →
(*StrongSimul* ((*refltrans redA*) # *red'A*) *redB R*).

The second result is known as strong normalisation by simulation, proved in [13]. The theorem, here called *SNbySimul*, states that if a reduction relation over A, say *redA*, is strongly simulated by a reduction relation over B, say *redB*, through R then the pre-image of any element that satisfies the predicate (*SN' redB*) also satisfies (*SN' redA*). A more detailed explanation of this result can be found in [8].

`Theorem` *SNbySimul* {*A B*} {*redA*: *Red A*} {*redB*: *Red B*} {*R*: *Rel A B*}:
StrongSimul redA redB R →
∀ *a*, *Image* (*inverse R*) (*SN' redB*) *a* → *SN' redA a*.

The union of two reduction relations is inductively defined as follows:

`Inductive` *union* {*A*} (*red1 red2*: *Red A*) : *Red A* :=
| *union_left*: ∀ *a b*, *red1 a b* → *union red1 red2 a b*
| *union_right*: ∀ *a b*, *red2 a b* → *union red1 red2 a b*.

`Notation` "R1 !_! R2" := (*union R1 R2*) (`at level` 40).

The next lemma shows that predicate *SN'* (*redA* !_! *red'A*) is patriarchal w.r.t. the reduction relation ((*refltrans redA*) # *red'A*).

`Lemma` *inclUnion* {*A*} {*redA red'A*: *Red A*}:
∀ *a*, (*SN' redA a*) →
(∀ *b*, (((*refltrans redA*) # *red'A*) *a b*) →
SN' (*redA* !_! *red'A*) *b*) →
(*SN'* (*redA* !_! *red'A*) *a*).

`Proof`.

`intros` *a HSN*. Given an arbitrary element *a*, let *HSN* be the hypothesis *SN' redA a*.

`induction` *HSN*. `clear` *H*. We proceed by induction on *HSN*. This means we can assume that our goal holds for all one-step *redA*-reduct of *a*. Call this assumption *H0*.

`intro` *H*. Let *H* be the hypothesis ∀ *b* : *A*, (*refltrans redA* # *red'A*) *a b* → *SN'* (*redA* !_! *red'A*) *b*

`apply` *sn_acc*. Applying the definition *SN'* to our goal *SN'* (*redA* !_! *red'A*) *a*, means this property must also be proved for all one-step (*redA* !_! *red'A*)-reduct of *a* , i.e. we need to prove ∀ *b* : *A*, (*redA* !_! *red'A*) *a b* → *SN'* (*redA* !_! *red'A*) *b*.

`intros` *b Hunion*. Let *b* be a one-step (*redA* !_! *red'A*)-reduct of *a*, and *Hunion* the hypothesis (*redA* !_! *red'A*) *a b*.

`inversion` *Hunion*; `subst`. Since (*redA* !_! *red'A*) *a b* we have that either *redA a b* or [red'A a b].

- `apply` *H0*. In the first case the hypothesis *H0* reduces our proof to two subgoals.

+ `assumption`. The first one *redA a b* is closed by assumption

+ `intros` *b' Hrefl*. The second subgoal is *SN'* (*redA* !_! *red'A*) *b'* where *b'* is a (*redA* !_! *red'A*)reduct of *b*.

`apply` *H*. Applying the hypothesis *H*, we need to prove (*refltrans redA* # *red'A*) *a b'*.

`inversion` *Hrefl*; `subst`. From the composition expressed in hypothesis *Hrefl*, we have that there is an element *b0* such that *refltrans redA b b0* and *red'A b0 b'*.

`apply` *compose* `with` *b0*. Similarly, our goal can be decomposed with the above element *b0*, leading to two subcases:

× `apply` *refltailtransit* `with` *b*. The first subcase *refltrans redA a b0* is proved from hypothesis *redA a b* and *refltrans redA b b0*.

```
** apply atleast1.
   apply singl; assumption.
** assumption.
× assumption.
- apply H.
  apply compose with a.
  + apply reflex.
  + assumption.
```

The second subcase is closed by assumption. `Qed`.

The following lemma states the conditions for a union of two reduction relations *redA* and *red'A* to be strongly normalising. It corresponds to one direction of the bi-implication of lemma *SNunion* below.

```
Lemma SNinclUnion {A} {redA red'A: Red A}: (∀ b, SN' redA b →
                              ∀ c, red'A b c → SN' redA c) →
                   (∀ a, (SN' ((refltrans redA) # red'A) a) →
                 (SN' redA a) → (SN' (redA !_! red'A) a)).
Proof.
```

`intros` *Hstable a HSNcomp*. Assume the stability of *SN' redA* w.r.t. the reduction relation *red'A*, and call this hypothesis *Hstable*. In addition, call *HSNcomp* the assumption (*SN'* ((*refltrans redA*) # *red'A*) *a*), for an arbitrary element *a*.

`induction` *HSNcomp*. We proceed by induction on the hypothesis *HSNcomp*. Therefore, we need to prove our goal, assuming that it holds for all (*refltrans redA* # *red'A*)-reduct of *a*, and we call *H0* this fact.

`intros` *HSN*. Let *HSN* be the hypothesis *SN' redA a*.

`apply` *inclUnion*. By lemma *SNinclUnion* we can prove *SN'* (*redA* !_! *red'A*) *a* if *SN' redA a* and (∀ *b* : *A*, (*refltrans redA* # *red'A*) *a b* → *SN'* (*redA* !_! *red'A*) *b*).

- `assumption`. The first fact is the hypothesis *HSN*

```
- intros b Hcomp.
  apply H0.
```

+ `assumption`. The second fact comes partially from the hypothesis *H0*, but we also have to prove *SN' redA b*.

```
+ inversion Hcomp; subst. clear Hcomp.
  assert(H': SN' redA b0).
  {
    apply stabComp with a; assumption.
  }
  apply Hstable with b0; assumption.
```
We conclude using the fact that *SN' redA* is stable w.r.t *red'A*.

`Qed.`

The next lemma gives a characterisation of the predicate *SN'* (*redA* !_! *red'A*). Another important property used is called *stability*. We say a predicate *P* is stable w.r.t. the reduction relation *R* when, for all *a* and *b* such that *R a b*, *P a* implies *P b*. Under hypothesis of stability of (*SN' redA*) w.r.t. the reduction relation *red'A*, predicate *SN'* (*redA* !_! *red'A*) can then be decomposed as the conjunction (*SN'* ((*refltrans redA*) # *red'A*)) ∧ (*SN' redA*):

```
Lemma SNunion {A} {redA red'A: Red A}:
  (∀ b, SN' redA b → ∀ c, red'A b c → SN' redA c) →
  ∀ a, (SN' (redA !_! red'A) a) ↔
       (SN' ((refltrans redA) # red'A) a) ∧ ((SN' redA) a).
Proof.
  intros Hstable a; split.
```
The proof is as follows: Suppose that (*SN' redA*) is stable w.r.t. the reduction relation *red'A*. Call this hypothesis *Hstable*. Split the bi-implication in two cases:

- `intro HSN. split.` In the first case, call *HSN* the hypothesis *SN'* (*redA* !_! *red'A*) *a*.

```
+ apply HId in HSN.
  generalize dependent HSN.
```
In order use lemma *SNbySimul*, we rewrite *SN'* (*redA* !_! *red'A*) *a* as *Image* (*inverse Id*) (*SN'* (*redA* !_! *red'A*)) *a*.

```
  apply SNbySimul.
```
By lemma *SNbySimul*, we then need to prove that (*refltrans redA* # *red'A*) is strongly simulated by (*redA* !_! *red'A*) through some relation.

```
  apply UnionReflStrongSimul.
```
This fact is proved in lemma *UnionReflStrongSimul* using the identity relation over *A*, that is (*refltrans redA* # *red'A*) is strongly simulated by (*redA* !_! *red'A*) through *Id*.

```
+ apply HId in HSN.
  generalize dependent HSN.
```
The second component of the conjunction requires a similar strategy to use lemma *SNbySimul*.

```
  apply SNbySimul.
  apply UnionStrongSimul.
```
We conclude using the fact that a reduction relation is strongly simulated by the union of itself with any other reduction relation through the identity relation, formalised in lemma *UnionStrongSimul*.

- `intro` *Hand*.

  `destruct` *Hand* `as` [*Hcomp HredA*]. On the other direction, call *Hcomp* (resp. *HredA*) the hypothesis *SN' (refltrans redA # red'A) a* (resp. *SN' redA a*).

  `generalize dependent` *HredA*.
  `generalize dependent` *a*.
  `apply` *SNinclUnion*; `assumption`. We conclude that *SN' (redA !_! red'A) a* by lemma *SNinclUnion*.

`Qed.`

The Modular Strong Normalisation Theorem, here called *ModStrNorm*, is specified in Coq's syntax as follows:

`Theorem` *ModStrNorm* {*A B*: `Type`} {*redA red'A*: *Red A*}
        {*redB*: *Red B*} {*R*: *Rel A B*}:
  (*StrongSimul red'A redB R*) →
  (*WeakSimul redA redB R*) →
  (∀ *b*: *A*, *SN' redA b*) → ∀ *a*:*A*, *Image* (*inverse R*) (*SN' redB*) *a* →
                                        SN' (*redA* !_! *red'A*) *a*.

`Proof.`

Let *A* and *B* be types, *redA* and *red'A* be two reduction relations over *A*, *redB* a reduction relation over *B*, and *R* a relation from *A* to *B*.

`intros` *Hstrong Hweak HSN a HImage*. Assume that *red'A* is strongly simulated by *redB* through *R* (hypothesis *Hstrong*), that *redA* is weakly simulated by *redB* through *R* (hypothesis *Hweak*), that *SN' redA b* holds for every *b*:*A* (hypothesis *HSN*), and let *a*:*A* be an arbitrary element in the inverse image of *SN' redB* (hypothesis *HImage*). We need to prove that *SN'* (*redA* !_! *red'A*) *a*. By lemma *SNunion* this is equivalent to prove that *SN'* (*refltrans redA* # *red'A*) *a* ∧ *SN' redA a*, under the hypothesis of stability of *SN' redA* w.r.t. the reduction relation *red'A*, which is trivially obtained from hypothesis *HSN* since every element of *A* satisfies the predicate *SN' redA*.

`assert`(*Hsplit*: *SN'* (*redA* !_! *red'A*) *a* ↔
                   SN' (*refltrans redA* # *red'A*) *a* ∧ *SN' redA a*).
{
  `apply` *SNunion*.
  `intros` *b HSN' c Hred*.
  `apply` *HSN*.
}
`destruct` *Hsplit* `as` [*H Hunion*]; `clear` *H*. Note that just one direction of this equivalence is needed.

`apply` *Hunion*; `split`. The proof of this conjunction is split in two parts. We prove that *SN'* (*refltrans redA* # *red'A*) *a*, which can be proved by lemma *SNbySimul*, as long as (*refltrans redA* # *red'A*) is strongly simulated by *redB* through *R*.

- `generalize dependent` *HImage*.

`apply` *SNbySimul*.

`apply` *RCSimul*; `assumption`. Now we need to prove that (*refltrans redA* # *red'A*) is strongly simulated by *redB* through *R*, which is achieved by lemma *RCSimul*.

- `apply` *HSN*. The second part of the conjunction corresponds to the hypothesis *HSN*, and we conclude.

`Qed`.

4 Conclusion

In this work we presented a constructive formalisation of the Modular Strong Normalisation Theorem in the Coq Proof Assistant. The proof is constructive in the sense that it does not use the principle of excluded middle or any other classical rule, such as proof by contradiction. The constructive approach is not the standard way to prove termination of a reduction relation. In fact, the usual technique to prove termination of a reduction relation is showing that it does not have infinite reduction sequences through a proof by contradiction (cf. [15, 21]). For instance, a classical proof of the Modular Strong Normalisation Theorem is presented at [7]. Constructive proofs are usually more difficult and elaborate than classical ones, but the former are preferred in the context of Computer Science.

The Modular Strong Normalisation Theorem is an abstract result that states the conditions for the union of two reduction relations to preserve strong normalisation (PSN). It is a non-trivial result in abstract reduction systems that uses the well-known technique of termination by simulation, i.e. the termination of a reduction system is obtained by simulating its steps via another reduction relation known to be terminating. The theorem is, for instance, applied in [7] to establish the PSN property of a calculus with explicit substitutions.

The proofs developed in this formalisation follow the ideas presented in [8], where a theory of constructive normalisation is developed. This theory is based on a different definition of strong normalisation. Instead of using Lengrand's definition, we used a more standard inductive definition of strong normalisation (cf. [7, 8, 14]). A formal proof of the equivalence between these definitions of strong normalisation is also provided. In this way, we have a simpler and straithforward formalisation of the constructive normalisation theory.

References

[1] Y. Toyama. "Counterexamples to Termination for the Direct Sum of Term Rewriting Systems". In: *Information Processing Letters* 25.3 (1987), pp. 141–143. DOI: 10.1016/0020-0190(87)90122-0. URL: https://doi.org/10.1016/0020-0190(87)90122-0.

[2] B. Gramlich. "Modularity in Term Rewriting Revisited". In: *Theoretical Computer Science* 464.nil (2012), pp. 3–19. DOI: 10.1016/j.tcs.2012.09.008. URL: https://doi.org/10.1016/j.tcs.2012.09.008.

[3] P.-A. Melliès. "Typed lambda-calculi with explicit substitutions may not terminate". In: *TLCA '95: Proceedings of the Second International Conference on Typed Lambda Calculi and Applications*. Vol. 902. LNCS. London, UK: Springer-Verlag, 1995, pp. 328–334.

[4] B. Guillaume. "The λs_e-calculus Does Not Preserve Strong Normalization". In: *J. of Func. Programming* 10.4 (2000), pp. 321–325.

[5] R. D. Lins. "A New Formula for the Execution of Categorial Combinators". In: *8th International Conference on Automated Deduction, Oxford, England, July 27 - August 1, 1986, Proceedings*. Ed. by J. H. Siekmann. Vol. 230. Lecture Notes in Computer Science. Springer, 1986, pp. 89–98. ISBN: 3-540-16780-3. DOI: 10.1007/3-540-16780-3_82. URL: https://doi.org/10.1007/3-540-16780-3_82.

[6] M. Abadi et al. "Explicit substitutions". In: *Journal of Functional Programming* 1(4) (1991), pp. 375–416.

[7] D. Kesner. "A Theory of Explicit Substitutions with Safe and Full Composition". In: *Logical Methods in Computer Science* 5.3:1 (2009), pp. 1–29.

[8] S. Lengrand. "Normalisation & Equivalence in Proof Theory & Type Theory". PhD Thesis. Université Paris 7 & University of St Andrews, 2006.

[9] T. C. D. Team. *The Coq Proof Assistant, version 8.7.2*. Feb. 2018. DOI: 10.5281/zenodo.1174360. URL: https://doi.org/10.5281/zenodo.1174360.

[10] P. Letouzey. "Coq Extraction, an Overview". In: *Logic and Theory of Algorithms, Fourth Conference on Computability in Europe, CiE 2008*. Ed. by A. Beckmann, C. Dimitracopoulos and B. Löwe. Vol. 5028. Lecture Notes in Computer Science. Springer-Verlag, 2008.

[11] C. Paulin-Mohring. "Inductive Definitions in the system Coq - Rules and Properties". In: *Typed Lambda Calculi and Applications, International Conference on Typed Lambda Calculi and Applications, TLCA '93, Utrecht, The Netherlands, March 16-18, 1993, Proceedings*. Ed. by M. Bezem and J. F. Groote. Vol. 664. Lecture Notes in Computer Science. Springer, 1993, pp. 328–345. ISBN: 3-540-56517-5. DOI: 10.1007/BFb0037116. URL: https://doi.org/10.1007/BFb0037116.

[12] B. Werner. "Une Théorie des Constructions Inductives". PhD Thesis. Université Paris 7, 1994.

[13] S. Lengrand. *Normalisation Theory*. Accessed on May, 2018. 2018. URL: http://www.lix.polytechnique.fr/~lengrand/Work/HDR/.

[14] F. van Raamsdonk. "Confluence and Normalization for Higher-Order Rewriting". Netherlands: Amsterdam University, 1996.

[15] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[16] H. Geuvers. "Introduction to Type Theory". In: *Language Engineering and Rigorous Software Development*. Language Engineering and Rigorous Software Development. Springer Science + Business Media, 2009, pp. 1–56. DOI: 10.1007/978-3-642-03153-3_1. URL: http://dx.doi.org/10.1007/978-3-642-03153-3_1.

[17] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. EATCS - Texts in Theoretical Computer Science. Springer, 2004.

[18] A. Chlipala. *Certified Programming with Dependent Types*. MIT Press, 2017. URL: {http://adam.chlipala.net/cpdt/}.

[19] B. C. Pierce et al. *Software Foundations*. http://www.cis.upenn.edu/~bcpierce/sf. Electronic textbook, 2014.

[20] S. Lengrand. *Induction principles as the foundation of the theory of normalisation: Concepts and Techniques*. Technical Report. PPS laboratory, Université Paris 7, Mar. 2005. URL: http://hal.ccsd.cnrs.fr/ccsd-00004358.

[21] R. d. V. Marc Bezem Jan Willem Klop, ed. *Term Rewriting Seminar – Terese*. Cambridge University Press, 2003.