Universidad ORT Uruguay

Facultad de Ingeniería

Normalization Proofs for the Simply-Typed Lambda Calculus in Agda

Entregado como requisito para la obtención del título de Master en Ingeniería

Sebastián Urciuoli - 161023

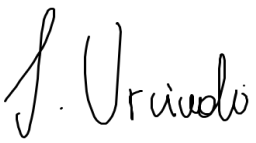
Tutores: Nora Szasz, Álvaro Tasistro

2020

Declaración de Autoría

Yo, Sebastián Urciuoli, declaro que el trabajo que se presenta en esta obra es de mi propia mano. Puedo asegurar que:

- La obra fue producida en su totalidad mientras realizaba el Proyecto;
- Cuando he consultado el trabajo publicado por otros, lo he atribuido con claridad;
- Cuando he citado obras de otros, he indicado las fuentes. Con excepción de estas citas, la obra es enteramente mía;
- En la obra, he acusado recibo de las ayudas recibidas;
- Cuando la obra se basa en trabajo realizado conjuntamente con otros, he explicado claramente qué fue contribuido por otros, y qué fue contribuido por mi;
- Ninguna parte de este trabajo ha sido publicada previamente a su entrega, excepto donde se han realizado las aclaraciones correspondientes.



Sebastián Urciuoli
28-10-2020

A mis padres.

Agradecimientos

Agradezco a mis tutores, el Dr. Álvaro Tasistro y la Dra. Nora Szasz por la dedicación, paciencia e invaluable formación —tanto profesional y académica como humana— que me brindaron en este maravilloso proceso.

También me gustaría agradecer al Dr. Ernesto Copello por su buena disposición y por la elaboración de la grandiosa librería en la que éste trabajo está basado ya que, al fin y al cabo, sin ella esto no sería posible.

Quisiera además darles las gracias al Dr. Miguel Pagano por su pericia en algunas de las técnicas de inducción usadas en este desarrollo.

Resumen

En el trabajo previo realizado en [1], los autores presentan una formalización del Cálculo Lambda en la Teoría Constructiva de Tipos [2] —más especificamente, en Agda [3]— en forma de biblioteca o *framework*. Los términos lambda son representados mediante una sintaxis de primer orden con una sola clase de nombres tanto para las variables libres como para las ligadas y usando sustituciones múltiples [4] como operación fundamental. En el mismo artículo, se presentan varios resultados metateóricos clásicos, como el teorema de Church-Rosser y la preservación de tipos (simples) bajo la relación de computación beta (*Subject Reduction*). Más tarde en [5], los autores logran formalizar completamente el teorema de estandarización usando dicha librería.

En [1, 5] el principal objetivo fue determinar hasta qué punto es posible construir una base de conocimiento de resultados metateóricos para el Cálculo Lambda usando la representación mencionada al comienzo y con principios de inducción simples. Hasta el momento los resultados han sido satisfactorios. Continuando esta misma línea es que ahora nos proponemos formalizar el teorema de normalización para el Cálculo Lambda con Tipos Simples en sus dos versiones: débil y fuerte. Dicho teorema tiene una prueba que creemos es más compleja que las vistas en los trabajos previos ya que utiliza algunas técnicas no desarrolladas en ellos, como inducción completa y definiciones mutuas, entre otras.

El teorema de normalización débil es ya un resultado bien conocido en la literatura y establece que todo término bien tipado bajo el sistema de asignación de sipos simples tiene forma normal, alcanzable por alguno de los caminos de evaluación posibles. Más aún se tiene el teorema de normalización —adecuadamente llamado— fuerte que afirma que sin importar el camino elegido, el programa siempre alcanzará una forma normal.

Si bien actualmente existe una vasta cantidad de formalizaciones de pruebas para ambos teoremas —algunas de las cuales serán discutidas en el Capitulo 1— según nuestro

conocimiento no existe una formalización completa según nuestra presentación del Cálculo Lambda —i.e. usando sintaxis de primer orden y con una sola clase de nombres. Esto es aún de mayor motivación para nuestro presente trabajo. En cuanto a la prueba a formalizar elegimos la de Joachimki y Matthes [6] ya que es compacta y fácil de entender. Dicha prueba depende de un lema que procede mediante una inducción completa en un orden lexicográfico que será explicado en detalle más adelante en este texto. En esta ocasión no haremos uso de la técnica de Relaciones Lógicas, aunque es un buen candidato para futuros trabajos.

Parte de este trabajo fue presentado en la conferencia LSFA20 [7] y posteriormente publicado en su acta (o *proceedings*, en inglés) [8].

Todas las definiciones y pruebas que aparecen en esta tesis han sido verificadas con el sistema Agda y dicho código puede ser encontrado en: https://github.com/surciuoli/lambda-metatheory.

Palabras clave: Cálculo Lambda, Metateoría Formal, Normalización Débil, Normalización Fuerte, Teoría Constructiva de Tipos, Agda.

Abstract

In previous work done in [1], the authors presented a formalization of the Lambda Calculus in the Constructive Type Theory [2] —more specifically, in Agda [3]— as a reusable library or framework. Lambda terms are represented using a first-order syntax with only one sort of names for both bound and free variables, and using multiples substitutions [4] as fundamental operation. In the same article, various meta-theoretical classic results are shown, e.g. the Church-Rosser Theorem and Subject Reduction with Simple Types. Later in [5], the authors are able to fully formalize the Standarization Theorem using the referred library.

In [1, 5], the primary objective was to test the extent on which it is possible to build a corpus of fully formalized meta-theoretical results of the Lambda Calculus using the presentation early mentioned and resorting only to basic induction principles. So far, the outcome is rather satisfactory. Continuing this line of development is that we tackle the normalization problem for the Lambda Calculus with Simple Types. The Weak Normalization Theorem is a well-known result for the Lambda Calculus and it states that every simply typable term has a normal form. Furthermore, the Strong Normalization Theorem states that, not only has a normal form but every reduction strategy will eventually end in one.

There exists a vast variety of formalizations of the proof for both the weak and strong versions —some of which will be discussed in Chapter 1—, but to the best to our knowledge there is not yet a full formalization using our presentation of the calculus mentioned at the very beginning, i.e. using first-order syntax with one sort of names. This is even a more challenging aspect to our present work. As to the proof to formalize, we have chosen the one by Joachimski and Matthes given in [6] as it is very slick and short. Said proof proceeds by complete induction on a lexicographic order which will be later explained in detail. In this occasion we are not going to use the method of Logical Relations, though

is a good candidate for future work.

Part of this work was submitted and presented at the international conference LSFA20 [7] and later published in its proceedings [8].

All definitions and proofs shown in this text have been completely machine-checked using the Agda system and its source code can be located at: https://github.com/surciuoli/lambda-metatheory.

Keywords: Lambda Calculus, Formal Metatheory, Weak Normalization, Strong Normalization, Constructive Type Theory, Agda.

Index

Chapter 1

Introduction

In [1], E. Copello et al. presented a framework which formalizes Lambda Calculus in Constructive Type Theory using first-order syntax with only one sort of names for both bound and free variables and relying on A. Stoughton's [4] multiple substitution as fundamental operation. In addition, a couple of well-known meta-theoretical results were verified: the Church-Rosser Theorem and the Subject Reduction Theorem for the Simply-Typed Lambda Calculus. Later, in [5] the library was used to check a proof of the Standarisation Theorem.

It was then the aim of the authors to investigate whether this specific approach was in any way amenable to full formalization. The approach goes back to the studies of the calculus by Curry and Feys [9] where (unary) substitution was given a well-known definition as a non-structural recursive function whose justification that it is a total function remains implicit and relies on the fact that renaming does not affect the size of a term. The formalization of the operation defined in this way is quite complex and it is not very practical, since every time the function is used the size of the term should be known in advance. A possible alternative would be to index (the definition of) the terms with a size ordinal or to use a function which computes it.

On the other hand, the use of simultaneous multiple substitution as explained in [4, 1], makes it possible to avoid recursion on the size of terms since, when substitution is applied to an abstraction, the bound name is changed to an appropriate one and the corresponding renaming is recorded into an enlarged, multiple substitution that goes on to operate on the body of the abstraction. Hence the effect of substitution can be given a simple definition by structural recursion on the terms, and this fact makes it plausible that the meta-theory of the calculus can be pursued using also simple forms of induction. The fact that bound

names are always changed permits to avoid case analyses too.

In the present work we continue this line of development with the hope of reaching a comprehensive corpus of fully verified meta-theoretical results of the Lambda Calculus. Is then necessary to test the extent to which [1] enables the task —by just using basic induction principles. Hence, we now decided to tackle the Weak and Strong Normalization theorems, which are considered a little more complex than previous theorems —as them implies using mutually defined sets and lemmas, course of values induction, etc.

The Strong Normalization Theorem for the Simply-Typed Lambda Calculus (STLC) states that for any (simply) typable term every reduction path starting from it must end. There are several proofs using different techniques, arguably the best known being the one presented by W. Tait in [10] and later adapted by J. Girard in [11]. In the latter, an abstract notion called *reducibility candidate* is introduced which is later refined into what now we call *Logical Predicates* or *Logical Relations* (LR) [12, p. 2]. This general proof method is mostly used to prove properties about programming languages.

As to the proof of choice, we use the one given by F. Joachimski and R. Matthes in [6] which we believe is very smart and short. It is done by induction on a syntax-directed characterization of the set of strongly normalizing terms —originally presented by F. van Rammsdonk and P. Severi in [13]— which makes it an optimal candidate for being formalized in Constructive Type Theory. This characterization must be accompanied by a proof of its soundness with respect to a more natural definition —e.g. the traditional one given by T. Altenkirch in [14]— since it might not be very easy to see it really defines the desired set. Such soundness proof can be found in the same article; however we shall use the one given in [15] as it is described in more detail to how to implement it. Another candidate of similar complexity and interest would be the one by Renè David presented in [16], which is a derivation of the one we have chosen.

The whole development has been formalized in Constructive Type Theory and machine-checked in the system Agda [3]. The corresponding code is available at https://github.com/surciuoli/lambda-metatheory.

State of the Art

As previously said, there is currently a wide variety of formalization of proofs of the Strong Normalization Theorem for the STLC written in various programming languages and using

different proof techniques. We now wish to comment some of the contributions found by the time this text was written:

- In [17], K. Donnelly and H. Xi give a formalization based on Tait's method. It is written entirely in ATS/LF [18] which is a logical framework with support for Higher-Order Abstract Syntax (HOAS) [19] so no names are used —needless to say, neither substitution nor alpha-conversion are considered;

- In [20], A. Abel also presents a formalization founded on Tait's method but for the Weak Normalization Theorem. Moreover, it is also written in a logical framework with support for HOAS: Twelf [21]. In the article, the author clarifies that, due to the limitations of the language, the proof cannot be stated for the Strong Normalization Theorem. However, he gives a glance on how it can be achieved in a language that would allow it.

- In [14] a formalization is given by T. Altenkirch for the System F —which is an extension of the Lambda Calculus with polymorphic types— written in the LEGO [22] language still based on Tait/Girard's reducibility candidates. For the terms, de Bruijn [23] encoding is used, hence alpha-conversion is not an issue.

- In [24], U. Berger et al. give a formalization written in three different languages: Minlog [25], Coq [26] and Isabelle/HOL [27]. Again, all of three follow Tait's method. As to the calculus presentation, they use de Bruijn indexes in all three implementations, except for Minlog, where they also give an alternative proof where terms are named. Nevertheless, they somehow are able to left undefined the substitution for the critical case, i.e. abstractions, so the variable capture issue is eluded —therefore, yet not formalized. Besides, the proof "starts from a large number of unproven axioms" [20, p. 122]

- In [28], A. Koprowski gives a formalization of the calculus using the *locally nameless* representation of terms with a proof of the Strong Normalization Theorem written entirely in Coq.

- Last but not least, we would like to mention the *POPLMark Challenge Reloaded* [15] (A. Abel et al). In this paper, a benchmark is carried out to test and compare different implementations of the theorem using LR. The proof is written in three languages: Beluga [29], Coq and Agda. However, none of them deals with the variable-capture issue nor alpha-conversion; in the first one —Beluga— since it is a HOAS system, no

(variables) names is taker under consideration, and in the last two implementations —in Coq and Agda— terms are presented using de Bruijn's encoding.

So to the best to our knowledge, there is not yet formal verification of the Strong Normalization Theorem for our approach of the Lambda Calculus as introduced at the very beginning of this section —using first-order syntax with (only) one sort of names and using multiple substitution.

Structure of this Thesis

- In Chapter 2, we shall introduce the framework of the Lambda Calculus presented in [1]. For this purpose, we first show some fundamental notions: the syntax of terms, substitutions, alpha-conversion, beta-reduction, etc. Then, in Sections 2.8 and 2.9 we propose new definitions we will need later, namely unary substitutions and neutral terms.

- In Chapter 3 we formalize the Weak Normalization Theorem as a previous step towards the Strong Normalization Theorem. As said earlier, we follow the proof presented in [6]; first we give a syntactic characterization of the set of terms that are weakly normalizing and then prove, by induction on the typing judgement, that all typable terms are in the former set, for which we will need a lemma for the case of applications.

- In Chapter 4 we formalize the Strong Normalization Theorem also following the proof in [6]. Hence, the general structure is the same as the previous section: first we introduce a syntactic characterization of the set of strongly normalizing terms —originally given in [13]— and then we proceed to the proof, also carried out by induction on the typing judgments. Both the definition and the proof itself can be extended straightforwardly from its counterpart —the weak version— and no line of code is needed to be modified, except for some of the names of the predicates. In a system with datatype extension this would not be even necessary. However, as commented in the beginning of this section, this version of the characterization is not very obviously correct, so we must also prove it is sound with respect to a more intuitive or traditional one; for this purpose we follow the proof given in [15] —with some minor changes in order to adapt it to our presentation of the Lambda Calculus, e.g. considering alpha-conversion, among some other thoughts.

- In Chapter 5 we present some overall conclusions; we assess drawbacks and benefits of our approach; report costs in terms of duration over time and lines of code; propose a new formalization using a different techniques and also some improvements of the actual code; etc.

- Finally, the complete Agda code can be located at the Appendix A.

Chapter 2

Preliminaries

In this chapter we first introduce some of the definitions and lemmas in the framework of the lambda calculus presented in [1]: syntax, substitutions, alpha-conversion, beta-reduction, types, etc. Finally, from Section 2.6 up to the end of the chapter, we present some new lemmas and definitions.

2.1 Syntax

We start with a denumerable type V of names, also to be called variables; i.e. for concreteness we can put $\mathsf{V} = \mathbb{N}$ or $\mathsf{V} = \mathsf{String}$. Letters x, y, z with primes or sub-indices shall stand for variables. Next, we define terms in Λ:

Definition 2.1 (Terms in Λ)**.** Terms are defined inductively using the following grammar:

$$M, N ::= x \mid MN \mid \lambda x.M.$$

From now on, capital letters M, N, P, $Q\ldots$ will range over lambda terms. In the concrete syntax we assume the usual convention according to which application binds tighter than abstraction.

Next, we introduce freshness and freedom relations:

Definition 2.2. That a variable x is free in M is written $x * M$ and its opposite is written $x\#M$ and to be pronounced x fresh in M. Both relations are defined as follows:

$$\frac{}{x * x} \qquad \frac{x * M}{x * MN} \qquad \frac{x * N}{x * MN} \qquad \frac{x * M}{x * \lambda y.M}\,(x \neq y)$$

$$\frac{}{x\#y}\,(x \neq y) \qquad \frac{x\#M \qquad x\#N}{x\#MN} \qquad \frac{x\#M}{x\#\lambda y.M} \qquad \frac{}{x\#\lambda x.M}$$

2.2 Substitutions

Substitution is the fundamental operation of the calculus, as it arises from the process of computation. Besides it is used to define alpha-conversion.

We work with multiple (simultaneous) substitutions, i.e. mappings associating a term to every variable. We use Σ to denote the corresponding type:

Definition 2.3 (Substitutions)**.** $\Sigma = \mathsf{V} \to \Lambda$.

Actually, the substitutions we must handle are identity almost everywhere, and so we can generate them by starting up from the empty substitution ι, which maps each variable to itself as a term, and applying the following operation:

Definition 2.4. The update operation on substitutions is defined by:

$$(\sigma, M/x)\ x = M$$
$$(\sigma, M/x)\ y = \sigma y, \text{ if } x \neq y.$$

That is, given a substitution σ, a name x and a term M, $(\sigma, M/x)$ is the substitution which maps x to M and (simultaneously) every other variable y to σy.

Whenever needed, we will write $[N/x]$, $[N/x, M/y \dots]$ and $(\sigma, N/x \dots)$ as syntax sugar for $(\iota, N/x)$, $(((\iota, N/x), M/y) \dots)$ and $((\sigma, N/x) \dots)$, respectively.

Restrictions

Often, we would like to *restrict* the effect of a substitution to only the free variables of a given term. Therefore, we shall introduce the following definition:

Definition 2.5. A restriction is a pair $\Sigma \times \Lambda$ to be written $\sigma \restriction M$.

Then we can extend freshness and freedom to restrictions:

Definition 2.6. $x\#\sigma \restriction M = (\forall y * M)\ x\#\sigma y$.

Definition 2.7. $x * \sigma \restriction M = (\exists y * M)\ x * \sigma y$.

That is, a name x is fresh in a restriction $\sigma \restriction M$ if it is fresh in every image through σ of a free variable of M. Freedom in a restriction is just the opposite condition.

Effect of Substitutions

Having defined what a substitution is —namely a mapping from variables to terms— we now wish to establish how this mapping is actually effected on terms:

Definition 2.8. Given a term M and a substitution σ, the effect of σ on M, i.e. applying σ on the (free) variables of M, is just written $M\sigma$ and defined by simple recursion on M:

$$\begin{aligned} x\sigma &= \sigma x \\ (MN)\sigma &= M\sigma N\sigma \\ (\lambda x.M)\sigma &= \lambda z.(M(\sigma, z/x)) \text{ with } z = \chi(\sigma, \lambda x.M). \end{aligned} \tag{2.1}$$

Notice in equation (2.1) how the bound variable x is always replaced with a new one. The new variable z is obtained by means of a choice function χ that computes the first variable fresh in $\sigma \restriction \lambda x.M$, so that it does not capture any of the names introduced into its scope by effect of the substitution. For a more detailed explanation, please refer to [1].

Besides implementing capture-avoidance, the use of multiple substitutions and uniform renaming of bound variables allows us to employ simple structural induction, avoiding case analyses when reasoning with substitutions. The main drawback is one attributable to the general approach chosen, i.e. we have to explicitly consider alpha-conversion in many of the proofs —mostly, where substitutions are mentioned.

The following properties about substitutions are proven in [1]:

Lemma 2.9. 1. $\chi(\sigma, M)\#(\sigma, M)$; 2. $\chi(\iota, M)\#M$, *and* 3. $M(\sigma, N\sigma/x) = M[N/x]\sigma$.

We shall name Lemma 2.9.1 the *Chi Lemma*.

Next, we can consider an equality on restrictions as follows:

Definition 2.10. $\sigma \lfloor M = \sigma' \lfloor M'$ if and only if M and M' share the same free variables and $(\forall x * M)\ \sigma x = \sigma' x$.

Then, $\sigma = \sigma' \lfloor M$ will be sugar for $\sigma \lfloor M = \sigma' \lfloor M$ and the following results are immediate:

Lemma 2.11. 1. $\sigma = \sigma' \lfloor M \ \Rightarrow\ M\sigma = M\sigma'$, *and* 2. $x\#M \ \Rightarrow\ \sigma = (\sigma, N/x) \lfloor M$.

2.3 Alpha-conversion

We shall now define the binary relation of alpha-conversion. We say two terms M and N are alpha-convertible and write it $M \sim_\alpha N$ if and only if they differ in nothing else but the choice of the bound names. We define it on top of substitution as follows:

Definition 2.12. Alpha-conversion is the binary relation $\sim_\alpha$ defined by:

$$\sim\mathtt{v}:\ \frac{}{x \sim_\alpha x} \qquad \sim\mathtt{app}:\ \frac{M \sim_\alpha M' \qquad N \sim_\alpha N'}{MN \sim_\alpha M'N'} \qquad \sim\lambda:\ \frac{M[z/x] \sim_\alpha N[z/y]}{\lambda x.M \sim_\alpha \lambda y.N}$$

where in the last rule $z\#\lambda x.M, \lambda y.N$.

The only rule worth to comment on is the last one: two functional terms are alpha-convertible if replacing the arguments in the bodies with the same fresh name yields alpha-convertible terms. Clearly, this reflects the idea that the bound names should not matter —speaking in a "functional" sense, of course.

We can then extend $\sim_\alpha$ to restrictions —of substitutions— as well:

Definition 2.13. $\sigma \sim_\alpha \sigma' \lfloor M = (\forall x * M)\ \sigma x \sim_\alpha \sigma' x$.

Now, the following lemmas —proven in [1]— hold:

Lemma 2.14.

1. $\iota \sim_\alpha \iota \lfloor M$
2. $\sigma \sim_\alpha \sigma' \lfloor M \ \wedge\ N \sim_\alpha P \ \Rightarrow\ (\sigma, N/x) \sim_\alpha (\sigma', P/x) \lfloor M$
3. $\sim_\alpha$ *is an equivalence relation*
4. $y\#\sigma \lfloor \lambda x.M \ \Rightarrow\ M(\sigma, y/x)[N/y] \sim_\alpha M(\sigma, N/x)$

5. $y\#\lambda x.M \;\Rightarrow\; \lambda x.M \sim_\alpha \lambda y.(M[y/x])$
6. $M \sim_\alpha N \;\Rightarrow\; \lambda x.M \sim_\alpha \lambda x.N$
7. $y\#\sigma \lfloor \lambda x.M \;\Rightarrow\; (\lambda x.M)\sigma \sim_\alpha \lambda y.(M(\sigma, y/x))$
8. $x\#M \;\wedge\; M \sim_\alpha N \;\Rightarrow\; x\#N$
9. $M \sim_\alpha M' \;\wedge\; \sigma \sim_\alpha \sigma' \lfloor M \;\Rightarrow\; M\sigma \sim_\alpha M'\sigma'$
10. $M \sim_\alpha N \;\Rightarrow\; M\sigma = N\sigma$
11. $M \sim_\alpha M\iota$.

We shall call Lemma 2.14.6 *Closure of Alpha-conversion under Lambda*; Lemma 2.14.8 *Closure of Freshness under Alpha-conversion*; Lemma 2.14.4 *Composition of Substitutions*, and Lemma 2.14.5 *Renaming of Abstraction*.

Lemma 2.14.10 is worthy of being mentioned. It is a consequence of the definition of substitution; if we have two alpha-convertible abstractions with different bound names $\lambda x.M$ and $\lambda y.N$, then if we apply the same substitution on both sides the result is that both x and y are renamed the same, i.e, $\chi(\sigma, \lambda x.M) = \chi(\sigma, \lambda y.N)$ —since the abstractions must share the same free variables. We shall call this property: *Equalization*.

From these, some useful corollaries —which are not included in [1]— are immediate:

Corollary 2.15.

1. $z\#\sigma \lfloor \lambda y.M \;\Rightarrow\; M(\sigma, z/y)[N\sigma/z] \sim_\alpha M[N/y]\sigma$
2. $(\lambda x.M)N \sim_\alpha (\lambda y.M')N' \;\Rightarrow\; M[N/x] \sim_\alpha M'[N'/y]$
3. $\lambda x.M \sim_\alpha \lambda y.N \;\Rightarrow\; M[y/x] \sim_\alpha N$.

Lemma 2.15.3 shall be named *Alpha-inversion*.

2.4 Beta-reduction

We say a redex is a term of the form $(\lambda x.M)N$ and define the beta-contraction relation on terms as:

$$\beta\colon \frac{}{(\lambda x.M)N \triangleright M[N/x]}$$

Then we shall say $M[N/x]$ is the contraction of the redex $(\lambda x.M)N$.

Besides, we say a term N is obtained by performing a step of computation of M and written $M \to_\beta N$ if and only if contracting a redex of M yields N. We can define this binary relation as follows:

Definition 2.16. Let $\to_\beta$ be defined by augmenting the beta-contraction relation with the following rules:

$$\texttt{appL}: \frac{M \to_\beta M'}{MN \to_\beta M'N} \quad \texttt{appR}: \frac{N \to_\beta N'}{MN \to_\beta MN'} \quad \lambda: \frac{M \to_\beta M'}{\lambda x.M \to_\beta \lambda x.M'}$$

Now, if a term M computes in a term N in multiple steps —zero or more— we shall write $M \twoheadrightarrow N$. Then, we can define this relation as the transitive closure of beta-reduction augmented with alpha-conversion:

Definition 2.17. Let $\twoheadrightarrow$ be defined as:

$$\frac{M \sim_\alpha N}{M \twoheadrightarrow N} \quad \frac{M \to_\beta N}{M \twoheadrightarrow N} \quad \frac{M \twoheadrightarrow N \qquad N \twoheadrightarrow P}{M \twoheadrightarrow P}$$

In addition, we can extend $\twoheadrightarrow$ to substitutions as follows:

Definition 2.18. $\sigma \twoheadrightarrow \sigma' \;=\; (\forall x)\ \sigma x \twoheadrightarrow \sigma' x.$

Then, the following hold:

Lemma 2.19.

1. $x\#M \;\wedge\; M \to_\beta N \;\Rightarrow\; x\#N$
2. $x\#\sigma \restriction M \;\wedge\; M \to_\beta N \;\Rightarrow\; x\#\sigma \restriction N$
3. $\sigma \twoheadrightarrow \sigma' \;\Rightarrow\; M\sigma \twoheadrightarrow M\sigma'$
4. $M \twoheadrightarrow M' \;\Rightarrow\; MN \twoheadrightarrow M'N \;\wedge\; NM \twoheadrightarrow NM' \;\wedge\; \lambda x.M \twoheadrightarrow \lambda x.M'.$

We shall name Lemmas 2.19.1 and 2.19.2 *Closure of Freshness under Beta-reduction* and Lemma 2.19.4 *Closure of Multiple-step Reduction under* Λ*-constructors.*

2.5 Types

Now we define the category of simple types:

Definition 2.20 (Types)**.** Let τ be a category of ground types. Then, the category of simple types is defined by the following grammar:

$$\alpha, \beta ::= \tau \mid \alpha \to \beta.$$

Next, we introduce an ordering between types in order to later perform complete induction on them:

Definition 2.21. We write $\alpha \preceq \beta$ when the type α is a (structural) component of type β, and $\alpha \prec \beta$ when it is a proper component (or sub-expression).

In order to assign types to terms, first we have to introduce the notion of a *context*. Contexts are mappings from variables to types and they are usually represented with the Greek symbols Γ and Δ. Now, we can say a term M types α under a given context Γ and write it $\Gamma \vdash M : \alpha$ if and only if it can be derived using the following rules:

Definition 2.22 (System of Assignment of Simple Types)**.**

$$\vdash\mathtt{v}: \frac{x \in \Gamma}{\Gamma \vdash x : \Gamma x} \quad \vdash\mathtt{a}: \frac{\Gamma \vdash M : \alpha \to \beta \qquad \Gamma \vdash N : \alpha}{\Gamma \vdash MN : \beta} \quad \vdash\lambda: \frac{\Gamma, x : \alpha \vdash M : \beta}{\Gamma \vdash \lambda x M : \alpha \to \beta}$$

where $x \in \Gamma$ means that x is declared in Γ, Γx is the (type) image of x and $\Gamma, x : \alpha$ is the context Γ updated with x having type α.

If some derivation exists for a term M we shall say M is *typable*.

Following are proven in [1]:

Lemma 2.23 (Weakening)**.** $x \# M \ \wedge \ \Gamma \vdash M : \alpha \ \Rightarrow \ \Gamma, x : \beta \vdash M : \alpha.$

Lemma 2.24 (Subject Reduction)**.** $\Gamma \vdash M : \alpha \ \wedge \ M \twoheadrightarrow N \ \Rightarrow \ \Gamma \vdash N : \alpha.$

Typed Substitutions

Sometimes we would like to refer only to (restrictions of) substitutions whose effect on a given typable term M is also typable. Therefore, we introduce the notion of *typed substitutions*:

Definition 2.25. A restriction $\sigma \lfloor M$ assigns to the variables in Γ terms of appropriate types under Δ —all of which is to be written $\sigma : \Gamma \rightharpoonup \Delta \lfloor M$— if and only if for all $x * M$ s.t. $x \in \Gamma$, we have $\Delta \vdash \sigma x : \Gamma x$.

Then, the following hold:

Lemma 2.26.

1. $\Gamma \vdash M : \alpha \ \wedge \ \sigma : \Gamma \rightharpoonup \Delta \lfloor M \ \Rightarrow \ \Delta \vdash M\sigma : \alpha$
2. $x \# \sigma \lfloor \lambda y.M \ \wedge \ \sigma : \Gamma \rightharpoonup \Delta \lfloor \lambda y M \ \Rightarrow \ (\sigma, x/y) : \Gamma, y{:}\alpha \rightharpoonup \Delta, x{:}\alpha \lfloor M$
3. $x \# M \ \wedge \ \sigma : \Gamma \rightharpoonup \Delta \lfloor M \ \Rightarrow \ (\sigma, y/x) : \Gamma, y{:}\alpha \rightharpoonup \Delta, x{:}\alpha \lfloor M$
4. $\Gamma \vdash M : \alpha \ \Rightarrow \ [M/x] : \Gamma, x{:}\alpha \rightharpoonup \Gamma \lfloor M$

From now on —and unless the opposite is stated explicitly— all results presented constitute new developments.

2.6 Substitution Compatibility Lemma

The lemma of compatibility of substitution with beta-reduction states that, if there exists a sequence $M \twoheadrightarrow N$, then $M\sigma \twoheadrightarrow N\sigma$ must follow. That is, we can replace the occurrences of the free variables in M and N in the same manner and obtain a valid reduction sequence. This is a fundamental lemma as we shall use it extensively throughout the text.

In [1], a similar lemma for *parallel* reductions is used to prove the Church-Rosser Theorem. A parallel reduction is a computation relation on terms which allows to contract multiple redexes at the same time —hence the name "parallel". In the same article, a proof is shown of the reflexive and transitive closure of parallel relation to be equivalent with our definition of multiple step reduction, therefore having the compatibility mentioned in the previous paragraph as a corollary. Also, in [5] a compatibility property is shown for head reductions[1].

However, as we shall see later, we also need a simpler lemma for one-step beta-reduction —which is not in [1]. Nevertheless, since as beta-reduction performs renaming of bound variables and alpha-conversion is not included in beta-reduction, $M \rightarrow_\beta N$ does not necessarily imply $M\sigma \rightarrow_\beta N\sigma$. To illustrate this, let us have e.g. $M = \lambda x.M'$ and $N = \lambda x.N'$. First, notice that in the process of reducing M to N in one step some

[1] A reduction sequence is called *head* if it is of the form $(\lambda x M.N_0)N_1N_2 \cdots \rightarrow M[N_0/x]N_1N_2 \ldots$

free variables might have been discarded. Then, if we apply a substitution σ on both sides M and N we get $\lambda y.(M(\sigma, y/x))$ and $\lambda y'.(N(\sigma, y'/x))$ with $y = \chi(\sigma, \lambda x.M)$ and $y' = \chi(\sigma, \lambda x.N)$. Now, since χ depends on the free variables of the restriction it follows $y \neq y'$, hence $\lambda y.(M(\sigma, y/x)) \to_\beta \lambda y'.(N'(\sigma, y'/x)$ can never be derived. Nonetheless, we can show that there always exists some P such that $M\sigma \to_\beta P$ and $P \sim_\alpha N\sigma$ as the following lemma establishes there:

Lemma 2.27. $M \to_\beta N \;\Rightarrow\; (\exists P)\; M\sigma \to_\beta P \;\wedge\; P \sim_\alpha N\sigma.$

Proof. By induction on the derivation of $M \to_\beta N$.

Case β: We have $M = (\lambda x.Q)R$ and $N = Q[R/x]$. Let $P = Q(\sigma, z/x)[R\sigma/z]$ with $z = \chi(\sigma, \lambda x.Q)$. First, by Chi Lemma (2.9.1) we have $z\#\sigma \restriction \lambda x.Q$. Then, on the one hand by definition of the effect of substitution (Def. 2.8), we get $((\lambda x.Q)R)\sigma = (\lambda z.Q(\sigma, z/x))R\sigma$, thus by β rule we have $(\lambda z.Q(\sigma, z/x))R\sigma \to_\beta P$. On the other hand, by Corollary 2.15.1 we have $P \sim_\alpha Q[R/x]\sigma$, as desired.

Case `appL`: Then $M = QR$, $N = Q'R$ and $QR \to_\beta Q'R$ follows from $Q \to_\beta Q'$. By ind. hyp. we know there exists some P' s.t. $Q\sigma \to_\beta P'$ and $P' \sim_\alpha Q'\sigma$. Let $P = P'R\sigma$. On the one hand, using `appL` rule we then have $(QR)\sigma = Q\sigma R\sigma \to_\beta P$ by definition of the effect of substitution. On the other hand, we can use $\sim$`app` rule with $P' \sim_\alpha Q'\sigma$ together with $R\sigma \sim_\alpha R\sigma$ —which holds because $\sim_\alpha$ is reflexive— and obtain $P \sim_\alpha (Q'R)\sigma$, as desired.

Case `appR`: Analogous to `appL`.

Case λ: Then $M = \lambda x.Q$, $N = \lambda x.R$ and $\lambda x.Q \to_\beta \lambda x.R$ follows from $Q \to_\beta R$. First of all, let us fix $z = \chi(\sigma, \lambda x.Q)$. Then, by ind. hyp. we have that there exists some P' s.t. $Q(\sigma, z/x) \to_\beta P'$ and $P' \sim_\alpha R(\sigma, z/x)$. Let $P = \lambda z.P'$. On the one hand, by definition of effect of substitution we have $\lambda z.Q(\sigma, z/x) = (\lambda x.Q)\sigma$, thus by λ rule of beta-reduction we have $\lambda x.Q' \to_\beta P\sigma$. On the other hand, first notice that by Chi Lemma we have $z\#\lambda x.Q \restriction \sigma$, thus by closure of freshness under beta-reduction (Lemma 2.19.2) we also have $z\#\lambda x.R \restriction \sigma$. Besides, by closure of alpha-conversion under λ (Lemma 2.14.6) using the second induction result we have $P \sim_\alpha \lambda z.(R(\sigma, z/x))$. Finally, by Lemma 2.14.7 using the last freshness assertion we have $P \sim_\alpha (\lambda x.R)\sigma$, as desired. □

Now, we can easily extend the previous lemma to multiple steps —and since as alpha-conversion is included into $\twoheadrightarrow$ we do not need an intermediate term P as in the previous case:

Lemma 2.28. $M \twoheadrightarrow N \;\Rightarrow\; M\sigma \twoheadrightarrow N\sigma$.

Proof. Follows easily by induction on the derivation of $M \twoheadrightarrow N$ and using Lemmas 2.27 and 2.14.10. □

2.7 Commutativity of Alpha and Beta

Beta-reduction and alpha-conversion relations commute, that is, we can exchange them in the following manner:

Lemma 2.29. $M \sim_\alpha N \;\wedge\; N \to_\beta P \;\Rightarrow\; (\exists Q)\; M \to_\beta Q \;\wedge\; Q \sim_\alpha P$.

In other words, if we have a beta-reduction $N \to_\beta P$ and we know $M \sim_\alpha N$, then we should be able to first perform the contraction of the redex on M and therefore obtain some Q such that $Q \sim_\alpha P$.

Proof. By case analysis on $N \to_\beta P$ and induction on M.

Case β: Then $M = (\lambda x.M')M''$, $N = (\lambda y.N')N''$ and $P = N'[N''/y]$. Let $Q = M'[M''/x]$. On the one hand, by β rule we have $(\lambda x.M')M'' \to_\beta Q$. On the other hand, by Corollary 2.15.2 using $M \sim_\alpha N$ we get $Q \sim_\alpha N'[N''/y]$.

Case `appL`: Then $M = M'M''$, $N = N'N''$ and $P = P'N''$, and $N'N'' \to_\beta P'N''$ follows from $N' \to_\beta P'$ and $M'M'' \sim_\alpha N'N''$ from $M' \sim_\alpha N'$ and $M'' \sim_\alpha N''$. By ind. hyp. with $M' \sim_\alpha N'$ and $N' \to_\beta P'$ we get there exists some Q' s.t. $M' \to_\beta Q'$ and $Q' \sim_\alpha P'$. Let $Q = Q'M''$. On the one hand, by appL rule with $M' \to_\beta Q'$ we get $M'M'' \to_\beta Q$. On the other hand, by $\sim$`a` applied to $Q' \sim_\alpha P'$ and $M'' \sim_\alpha N''$ we obtain $Q'M'' \sim_\alpha P'N''$.

Case `appR`: Analogous.

Case λ: Then $M = \lambda x.M'$, $N = \lambda y.N'$ and $P = \lambda y.P'$. Also, $\lambda x.M' \sim_\alpha \lambda y.N'$ and $\lambda y.N' \to_\beta \lambda y.P'$ follows from $N' \to_\beta P'$. On the one hand, by compatibility of substitution using $N' \to_\beta P'$ we have that there exists some R s.t. $N'[x/y] \to_\beta R$ and $R \sim_\alpha P[x/y]$. Besides, by alpha inversion (Corollary 2.15.3) with hypothesis $\lambda x.M' \sim_\alpha \lambda y.N'$ we obtain $M' \sim_\alpha N'[x/y]$. Hence, by ind. hyp. on $M' \sim_\alpha N'[x/y]$ and $N'[x/y] \to_\beta R$ we have that there exists a Q' s.t. $M' \to_\beta Q'$ and $Q' \sim_\alpha R$. Let $Q = \lambda x.Q'$. Then, by λ rule with $M' \to_\beta Q'$ we obtain $\lambda x.M' \to_\beta Q$, as the first desired thesis. Now, on the other hand, first

notice that by transitivity of alpha using $Q' \sim_\alpha R$ and $R \sim_\alpha P[x/y]$ we get $Q' \sim_\alpha P[x/y]$, thus by closure of alpha under λ with x we obtain $Q \sim_\alpha \lambda x.(P[x/y])$. Secondly, by definition of freshness we know $x\#\lambda x.M'$, thus by preservation of freshness under alpha using $\lambda x.M' \sim_\alpha \lambda y.N'$ we have $x\#\lambda y.N'$, hence by preservation under beta with $\lambda y.N' \to_\beta \lambda y.P'$ we get $x\#\lambda y.P'$. Then, by renaming of bound variable (Lemma 2.14.5) we have $\lambda y.P' \sim_\alpha \lambda x.(P[x/y])$, so by symmetry we obtain $\lambda x.(P[x/y]) \sim_\alpha \lambda y.P'$. Finally, again by transitivity using $Q \sim_\alpha \lambda x.(P[x/y])$ and $\lambda x.(P[x/y]) \sim_\alpha \lambda y.P'$ we derive $Q \sim_\alpha \lambda y.P'$, as the second desired thesis. □

We can easily extend this result to reductions of multiple steps as follows:

Lemma 2.30. $M \sim_\alpha N \ \wedge \ N \twoheadrightarrow P \ \Rightarrow \ (\exists Q)\ M \twoheadrightarrow Q \ \wedge \ Q \sim_\alpha P.$

Proof. Immediate by induction on the derivation of $N \twoheadrightarrow P$ using previous lemma. □

2.8 Unary Substitutions

It turns out convenient to introduce substitutions that arise in the process of contracting redexes $(\lambda x.M)N \to_\beta M[N/x]$. The result is the (unary) substitution $[N/x]$. However, when computing the effect of the substitution, every time an abstraction is crossed the substitution is updated with a renaming, thus leading to something of the form:

$$(\lambda x_1.\lambda x_2 \ldots \lambda x_n.M)[N/y] = \lambda z_1.\lambda z_2 \ldots \lambda z_n.M[N/y, z_1/x_1, z_2/x_2 \ldots z_n/x_n]$$

with $z_i = \chi(\lambda x_i.\lambda x_{i+1} \ldots, [N/y, z_1/x_1 \ldots z_{i-1}/x_{i-1}])$.

As it can be seen, the final substitution is still unary in the sense that, the "real" —or original if we want— effect of the substitution is the replacement of y to N, whereas the renamings are merely a consequence of how substitution is defined. Thus, we can capture this concept of unary substitutions as follows:

Definition 2.31. A substitution σ is unary of variable x and term M —written as $\mathsf{Unary}\,\sigma\,x\,M$— if and only if $\sigma x = M$ and, for all $y \neq x$, then σy is a variable.

Then, the following lemmas hold immediately by definition:

Lemma 2.32.

1. $\mathsf{Unary}\,[M/x]\,x\,M$
2. $\mathsf{Unary}\,\sigma\,x\,M \;\Rightarrow\; \mathsf{Unary}\,(\sigma, N/x)\,x\,N$
3. $y \neq x \;\wedge\; \mathsf{Unary}\,\sigma\,x\,M \;\Rightarrow\; \mathsf{Unary}\,(\sigma, z/y)\,x\,M$
4. $\mathsf{Unary}\,\sigma\,x\,y \;\Rightarrow\; \mathsf{Unary}\,(\sigma, M/z)\,z\,M$.

2.9 Neutral Terms

Neutral terms are iterated applications having a variable as "head":

Definition 2.33 (Neutral terms)**.**

$$\frac{}{\mathsf{ne}_x\,x} \qquad \frac{\mathsf{ne}_x\,M}{\mathsf{ne}_x\,MN.}$$

In the definition, $\mathsf{ne}_x\,M$ is to be pronounced: *M is neutral of head x.*

The following property holds for neutral terms —which we will exploit further to perform complete induction on types:

Lemma 2.34. $\mathsf{ne}_x\,M \;\wedge\; \Gamma \vdash M : \alpha \;\wedge\; \Gamma \vdash x : \beta \;\Rightarrow\; \alpha \preceq \beta$.

Proof. By induction on the derivation of $\mathsf{ne}_x\,M$.

Case $M = x$: Then $\Gamma \vdash x : \alpha$ and $\Gamma \vdash x : \beta$. It follows immediately $\alpha = \beta$. **Case** $M = M'N$. Then $\mathsf{ne}_x\,M'$ and $\Gamma \vdash M'N : \alpha$ follows from $\Gamma \vdash M' : \gamma \to \alpha$ and $\Gamma \vdash N : \gamma$. By ind. hyp. using $\mathsf{ne}_x\,M'$ and $\Gamma \vdash M' : \gamma \to \alpha$ we have either $\gamma \to \alpha = \beta$ or $\gamma \to \alpha \prec \beta$[1]. As to the first case, the thesis $\gamma \preceq \gamma \to \alpha$ follows immediately since by definition of $\preceq$ we know $\gamma \preceq \gamma \to \alpha$. As to the second case, again by definition of $\prec$ we have $\gamma \prec \gamma \to \alpha$. Then, by transitivity of $\prec$ using $\gamma \prec \gamma \to \alpha$ and $\gamma \to \alpha \prec \beta$ we obtain $\gamma \prec \beta$, which implies $\gamma \preceq \beta$. □

In addition, if we have M is functional of type $\gamma \to \alpha$, then we have $\gamma \to \alpha \preceq \beta$, thus $\gamma \prec \beta$ follows immediately —which is stated in the next corollary:

Corollary 2.35 (Argument Lemma)**.** $\mathsf{ne}_x\,M \;\wedge\; \Gamma \vdash M : \gamma \to \alpha \;\wedge\; \Gamma \vdash x : \beta \;\Rightarrow\; \gamma \prec \beta$.

[1] The following property is used: $\alpha \preceq \beta$ if and only if $\alpha = \beta$ or $\alpha \prec \beta$

Chapter 3

Weak Normalization

In the previous chapter we presented some basic notions of the Lambda Calculus which were formalized in [1]. Also, we have mentioned some additional results.

Now in this chapter we introduce the Weak Normalization Theorem. For the purpose, we shall first define exactly what it is for a term to be weakly normalizing (WN) and after that, we will proceed to fully formalize the theorem.

Later in the next chapter, we will show how to extend the definition of WN to capture the set of strongly normalizing terms and further proceed to prove the Strong Normalization Theorem. Then as we shall see, the proof of this last one is the same as the one in this chapter except for an additional few cases —resulting of the extension mentioned above.

3.1 Inductive Definition

A term M is said to be weakly normalizing —written $\mathsf{WN}\,M$— if and only if it has normal form. Furthermore, if $\mathsf{ne}\,x\,M$ then we shall say it is neutral weakly normalizing of head x and write it $\mathsf{WNe}\,x\,M$. Following [13, 15], we define both predicates mutually using the following rules:

Definition 3.1 (WN and WNe)**.**

$$\mathtt{v}\colon \frac{}{\mathsf{WNe}\,x\,x} \qquad \mathtt{app}\colon \frac{\mathsf{WNe}\,x\,M \quad \mathsf{WN}\,N}{\mathsf{WNe}\,x\,M\,N} \qquad \mathtt{ne}\colon \frac{\mathsf{WNe}\,x\,M}{\mathsf{WN}\,M} \qquad \lambda\colon \frac{\mathsf{WN}\,M}{\mathsf{WN}\,\lambda x.M} \qquad \mathtt{exp}\colon \frac{\mathrm{M}\twoheadrightarrow\mathrm{N} \quad \mathsf{WN}\,N}{\mathsf{WN}\,M}$$

Rule v stats that any variable is weakly normalizing —in fact, it is in neutral normal form. Rule app says that given a term M neutral weakly normalizing and a term N (just) weakly normalizing, then MN is also neutral weakly normalizing. Next, rule ne tell us every term in WNe is also in WN. Then, rule λ says that, provided a term has normal form, then enclosing it in an abstraction has also a normal form; the value of the former with an lambda constructor prefixed. Finally, the last rule known as the "expansion" rule states that given a normalizing term N which has normal form, then every term M that computes N must have normal form too.

3.2 The Weak Normalization Theorem

We are ready to state the Weak Normalization Theorem. The theorem says that any typable term is weakly normalizing. For the proof, we follow the one given in [6] which is simply driven by induction on the derivation of the typing judgment. However, for the case of the application a lemma is needed: given two weak normalizing terms M and N then its application MN must also be weakly normalizing —this is Lemma 3.4 proven below.

Theorem 3.2 (Weak Normalization)**.** $\Gamma \vdash M : \alpha \;\Rightarrow\; \mathsf{WN}\, M.$

Proof. By induction on the derivation of $\Gamma \vdash M : \alpha$. **Case** $\vdash$v: By definition of WN (Def. 3.1). **Case** $\vdash$a: $\Gamma \vdash MN : \alpha$ is obtained from $\Gamma \vdash M : \beta \to \alpha$ and $\Gamma \vdash N : \beta$. By ind. hyp. we get $\mathsf{WN}\, M$ and $\mathsf{WN}\, N$. Then, by Lemma 3.4 2 (ii) we have $\mathsf{WN}\, MN$. **Case** $\vdash\lambda$: By ind. hyp. and definition of WN. □

Again, the lemma used in the case $\vdash$a states that WN is closed under application. However, as we shall see, we need two additional results. The first one is needed for the case of the left term being an abstraction. Say we have $\mathsf{WN}\, \lambda x.M$ and $\mathsf{WN}\, N$: then we must show $\mathsf{WN}\, (\lambda x.M)N$. By beta-reduction we know $(\lambda x.M)N \to_\beta M[N/x]$. Now, if we knew that the right term of the the reduction is WN , then by exp rule we would also know the redex —the left side— is WN, as desired. For establishing this is we need an extra hypothesis, i.e. $\mathsf{WN}\, M[N/x]$ provided M and N are WN. Finally, a second additional result is still needed: $\mathsf{WNe}_x\, M$ and $\mathsf{WN}\, N$ implies $\mathsf{WNe}_{\sigma x}\, M[N/y]$ for any $y \neq x$. That is, replacing a variable different from the head in a weakly neutral normalizing term must also yield a weakly neutral normalizing term[1].

[1] In [20], the same idea is applied.

Also, we need the following lemma, which is proven by simple induction:

Lemma 3.3 (Strong Neutral Weakening)**.** $\mathsf{WNe}_x\, M \;\Rightarrow\; \mathsf{ne}_x\, M$.

Without further ado, let us proceed to the main lemma:

Lemma 3.4. *Let* $\Gamma \vdash M : \alpha$, $\Gamma \vdash N : \beta$, $\mathsf{WN}\, N$, $\sigma : \Gamma \rightharpoonup \Delta \restriction M$, $\mathsf{Unary}\, \sigma\, x\, N$ *and* $\Gamma \vdash x : \beta$. *Then:*

1. $\mathsf{WNe}_y\, M \Rightarrow (i)\ \mathsf{WN}\, M\sigma$, $(ii)\ y{\neq}x \Rightarrow \mathsf{WNe}_{\sigma y}\, M\sigma$ *and* $(iii)\ \alpha = \beta{\to}\gamma \Rightarrow \mathsf{WN}\, MN$.
2. $\mathsf{WN}\, M \Rightarrow (i)\ \mathsf{WN}\, M\sigma$ *and* $(ii)\ \alpha = \beta{\to}\gamma \Rightarrow \mathsf{WN}\, MN$.

Proof. The proof proceeds by complete induction on the structure of the type β, and subordinate induction on the definition of WN (Def. 3.1). So, to begin with, take β and assume the statement for all of its proper components. We proceed to the subordinate induction by case analysis:

Case `v`: Then $M = y$ and we must show 1: (i) $\mathsf{WN}\, y\sigma$; (ii) $y{\neq}x \Rightarrow \mathsf{WNe}_{\sigma y}\, y\sigma$ and (iii) $\alpha = \beta{\to}\gamma \Rightarrow \mathsf{WN}\, yN$. As to (i), either $y = x$ or not. In the first case, by definition of unary substitution (Def. 2.31) we have $\sigma y = N$, which is normalizing by hypothesis. In the second case, by the same definition we get $\sigma y = z$ is a variable, which is also normalizing. As to (ii), if $y{\neq}x$ then by definition of WNe we have $\mathsf{WNe}_z\, z$. As to (iii): by rule `app` of WN with $\mathsf{WNe}_y\, y$ and $\mathsf{WN}\, N$ we have $\mathsf{WNe}_y\, yN$, thus by rule `ne` we get $\mathsf{WN}\, yN$, as desired.

Case `app`: Then $M = PQ$ and $\mathsf{WNe}_y\, PQ$ follows from $\mathsf{WNe}_y\, P$ and $\mathsf{WN}\, Q$, and $\Gamma \vdash PQ : \alpha$ follows from $\Gamma \vdash P : \delta \to \alpha$ and $\Gamma \vdash Q : \delta$. We need to show 1: (i) $\mathsf{WN}\, (PQ)\sigma$, (ii) $y{\neq}x \Rightarrow \mathsf{WNe}_{\sigma y}\, (PQ)\sigma$ and (iii) $\alpha = \beta{\to}\gamma \Rightarrow \mathsf{WN}\, (PQ)N$. First, by ind. hyp. we have $\mathsf{WN}\, P\sigma$ and $\mathsf{WN}\, Q\sigma$. Besides, by Lem. 2.26.1 we get $\Delta \vdash P\sigma : \delta \to \alpha$ and $\Delta \vdash Q\sigma : \delta$. Now, we have either $y = x$ or not. In the first case, by neutral weakening (Lem. 3.3) we get $\mathsf{ne}_x\, P$, thus by Argument Lemma (2.35) it follows $\delta \prec \beta$. So, as to (i), we can apply the main ind. hyp. 2(ii) with $M{:=}P\sigma$, $N{:=}Q\sigma$ and $\beta{:=}\delta$[1] and get $\mathsf{WN}\, P\sigma Q\sigma$ which —by definition of effect of substitution (Def. 2.8)— equals to $\mathsf{WN}\, (PQ)\sigma$, as desired. On the contrary, if $y{\neq}x$, then by ind. hyp. (ii) we get $\mathsf{WNe}_{\sigma y}\, P\sigma$. Furthermore, by rule `app` of definition of WN it follows $\mathsf{WNe}_{\sigma y}\, P\sigma Q\sigma$, thus by `ne` rule we obtain $\mathsf{WN}\, (PQ)\sigma$. (ii) Just proven. (iii) Similar to case `v`.

[1] By $M{:=}P\sigma$ we mean that the induction is made with its parameter M set to $P\sigma$; analogously to N and β.

Case ne: Then $\mathsf{WN}\,M$ follows from $\mathsf{WNe}_y\,M$ and 2 is direct by ind. hyp. 1 (i) and (iii).

Case λ: Then $M = \lambda y.P$ and $\mathsf{WN}\,\lambda y.P$ follows from $\mathsf{WN}\,P$ and $\Gamma \vdash \lambda y.P : \delta \to \epsilon$ from $\Gamma, y : \delta \vdash P : \epsilon$, and we have $\sigma : \Gamma \rightharpoonup \Delta \downharpoonright \lambda y.P$. We need to show 2: (i) $\mathsf{WN}\,(\lambda y.P)\sigma$ and (ii) $\alpha{=}\beta{\to}\gamma \;\Rightarrow\; \mathsf{WN}\,(\lambda y.P)N$. As to (i), by definition of substitution we have $(\lambda y.P)\sigma = \lambda z.P(\sigma, z/y)$ with $z = \chi(\sigma, \lambda y.P)$. We now proceed by cases: either $y = x$ or not. In the first case, before we can apply the ind. hyp. with $M{:=}\mathsf{WN}\,P$ and $\sigma{:=}(\sigma, z/x)$, we need to show $\Gamma, x : \delta \vdash x : \beta$ —since P types under the context $\Gamma, x{:}\delta$. However, this does not necessarily follow since δ is arbitrary. Nevertheless, we can proceed as follows: let us fix some $u = \chi(\iota, P)$. Then, by Lemma 2.9.2 we have $u\#P$. Now, seeing that $z = \chi(\sigma, \lambda x.P)$, by Chi Lemma (2.9.1) we have $z\#\lambda x.P$ and thus, by successive applications of typed substitution Lemmas 2.26.2 and 2.26.3, we obtain $(\sigma, z/x, v/u) : \Gamma, x{:}\delta, u{:}\beta \rightharpoonup \Delta, z{:}\delta, v{:}\beta \downharpoonright P$ —for any v. Besides, by Lemma 2.23 we have $\Gamma, x{:}\delta, u{:}\beta \vdash P : \epsilon$ hence by definition of assignment of types (Def. 2.22) we get $\Gamma, x{:}\delta, u{:}\beta \vdash u : \beta$. Thus, by consecutive application of (unary) Lemmas 2.32.2 and 2.32.4 we have $\mathsf{Unary}\,(\sigma, z/x, v/u)\,u\,v$, and since v is normalizing by definition, we can apply the ind. hyp. and get $\mathsf{WN}\,P(\sigma, z/x, v/u)$. Then, by Lemma 2.11.2 using $u\#P$ followed by Lemma 2.11.1 we have $P(\sigma, z/x, v/u) = P(\sigma, z/x)$, thus $\mathsf{WN}\,P(\sigma, z/x)$. Finally, by rule λ of WN and definition of substitution we obtain $\mathsf{WN}\,(\lambda x.P)\sigma$, as desired. Now if, on the other hand, $y \neq x$, then by (unary) Lemma 2.32.3 we have $\mathsf{Unary}\,(\sigma, z/y)\,x\,N$ and thus we can straightforwardly apply the ind. hyp. to obtain $\mathsf{WN}\,P(\sigma, z/y)$ and therefore derive $\mathsf{WN}\,(\lambda y.P)\sigma$ in the same way. As to (ii), first notice that, since $\alpha = \beta \to \gamma$, it follows $\beta = \delta$, hence $\Gamma, y{:}\beta \vdash y : \beta$. Then, by (unary) Lemma 2.32.1 we have $\mathsf{Unary}\,[N/y]\,y\,N$ and by Lemma 2.26.4 $[N/y] : \Gamma, y{:}\beta \rightharpoonup \Gamma \downharpoonright P$, thus we can apply the ind. hyp. (i) and obtain $\mathsf{WN}\,P[N/y]$. Lastly, since $(\lambda y.P)N \twoheadrightarrow P[N/y]$ by exp rule we can derive $\mathsf{WN}\,(\lambda y.P)N$.

Case exp: Then $\mathsf{WN}\,M$ follows from $M \twoheadrightarrow P$ and $\mathsf{WN}\,P$, and we have $\Gamma \vdash M : \alpha$. We need to show 2: (i) $\mathsf{WN}\,M\sigma$ and (ii) $\alpha{=}\beta{\to}\gamma \supset \mathsf{WN}\,MN$. As to (i), by compatibility of substitution (Lemma 2.28) we have $M\sigma \twoheadrightarrow P\sigma$, and by main ind. hyp. 2(i) we get $\mathsf{WN}\,P\sigma$, therefore by rule exp we obtain $\mathsf{WN}\,M\sigma$. As to (ii), by subject reduction (Lemma 2.24) on the typing of M we have $\Gamma \vdash P : \beta{\to}\gamma$, thus we can apply the ind. hyp. (ii) and have $\mathsf{WN}\,PN$. Then, by closure of multiple-step reduction under the Λ-constructors (Lemma 2.19.4) applied to $M \twoheadrightarrow P$ we get $MN \twoheadrightarrow PN$, hence by exp we can derive $\mathsf{WN}\,MN$, as desired. □

Chapter 4

Strong Normalization

In the previous chapter we proved that every typable term has a normal form. Now, we will show a stronger result: typable terms not only have a normal form, but every reduction strategy used will eventually lead to it. This is known as the Strong Normalization Theorem.

The structure of the proof is almost exactly the same as the one of the weak version. First, we give a syntactic characterization of the set of strongly normalizing terms (SN) and then, we follow Joachimksi and Matthes proof's skeleton. Furthermore, SN is obtained from WN by adding a few rules; the same applies to the proof itself. However, since as is might not easy to see that SN characterizes the desired set, a proof of its soundness respect to a more natural definition is desired.

So, to begin with, we start this chapter by giving two different characterizations of the set of strongly normalizing terms; a syntactic one (SN) and a more natural one (sn). Then, we will show the first one implies the second one[1] and finally prove every typable term is SN.

4.1 Traditional Definition

A first traditional definition of the set of strongly normalizing terms originally given by T. Altenkirch in [14] is obtained by considering the accessible —or well-founded— part of

[1] Actually, the sets are equivalent. However, we only need one side of the equivalence, that is, the soundness of SN with respect to sn. For a proof of the other implication, i.e. its completeness, the interested reader may refer to [6, 13].

the beta-reduction relation; put in other words, a term M is strongly normalizing —and written $\mathsf{sn}\, M$— if and only if for every $M \to_\beta N$ then N is also strongly normalizing:

Definition 4.1 (Traditional Definition of Strongly Normalizing Terms)**.**

$$\frac{(\forall N)\ (M \to_\beta N \ \Rightarrow\ \mathsf{sn}\, N)}{\mathsf{sn}\, M.}$$

Then, the following properties about sn are proven:

Lemma 4.2. 1. sn *is closed under* $\sim_\alpha$*;* 2. $\mathsf{sn}\, x$*;* 3. sn *is closed under* $\twoheadrightarrow$*;* 4. $\mathsf{sn}\, MN \ \Rightarrow\ \mathsf{sn}\, M \ \wedge\ \mathsf{sn}\, N$*, and* 5. $\mathsf{sn}\, M \ \Rightarrow\ \mathsf{sn}\, \lambda x.M$*.*

Proof. (4.2.1) By induction on $\mathsf{sn}\, M$. Take N and P such that $M \sim_\alpha N$ and $N \to_\beta P$. By Lemma 2.29 we have Q s.t. $M \to_\beta Q \sim_\alpha P$. Hence $\mathsf{sn}\, Q$ and by ind. hyp., since $Q \sim_\alpha P$, $\mathsf{sn}\, P$ as desired. (4.2.2) By vacuity, since there is no possible way of having $x \to_\beta M$ for any x. (4.2.3) By induction on $M \twoheadrightarrow N$. (4.2.4) By induction on $\mathsf{sn}\, MN$. (4.2.5) By induction on $\mathsf{sn}\, M$. □

4.2 Syntactic Definition

Definition 4.1 might be simple, but since it is not driven by syntax, performing induction on it becomes quite difficult. Nevertheless, —as we said earlier in the introduction of this chapter— we can extend the previous definition of WN to characterize the set of strongly normalizing terms just by replacing the reduction relation used in the `exp` rule:

Definition 4.3 (SN, SNe and $\to_{\mathsf{SN}}$)**.**

$$\mathtt{v}\colon \frac{}{\mathsf{SNe}_x\, x} \qquad \mathtt{app}\colon \frac{\mathsf{SNe}_x\, M \quad \mathsf{SN}\, N}{\mathsf{SNe}_x\, MN}$$

$$\mathtt{ne}\colon \frac{\mathsf{SNe}_x\, M}{\mathsf{SN}\, M} \qquad \lambda\colon \frac{\mathsf{SN}\, M}{\mathsf{SN}\, \lambda x.M} \qquad \mathtt{exp}\colon \frac{M \to_{\mathsf{SN}} N \quad \mathsf{SN}\, N}{\mathsf{SN}\, M}$$

$$\beta\colon \frac{\mathsf{SN}\, N}{(\lambda x.M)N \to_{\mathsf{SN}} M[N/x]} \qquad \mathtt{appL}\colon \frac{M \to_{\mathsf{SN}} M'}{MN \to_{\mathsf{SN}} M'N} \qquad \alpha\colon \frac{M \to_{\mathsf{SN}} R \qquad R \sim_\alpha N}{M \to_{\mathsf{SN}} N}$$

As it can be seen, instead of using $\twoheadrightarrow$ we have now introduced a new reduction relation $\to_{\mathsf{SN}}$ —to be pronounced (big) strongly normalizing head reduction or strong reduction,

for short— which is a left-most strategy, in addition to requiring the argument of the redex being strongly normalizing. Besides, it does not step inside abstractions —hence is an outer-most strategy. This relation guarantees that both paths, that is, contracting the redex (left-most) and reducing the argument (right-most) are normalizing.

Also, we have introduced a new rule, i.e. α which allow us to perform an alpha-conversion after a strong reduction step. The reason of this is that, similarly as what happens with WN where use the lemma of compatibility of multi-step reduction with substitution in the proof of the main lemma —more specifically, in the `exp` case— we will also need a lemma of compatibility of $\to_{\mathsf{SN}}$ for SN, i.e. $M \to_{\mathsf{SN}} N$ implies $M\sigma \to_{\mathsf{SN}} N\sigma$. This is only possible if we have alpha-conversion as part of the reduction relation.

4.3 Soundness of SN

Now we will show that SN is sound with respect of sn, i.e. every term in SN is also in sn, and then we shall prove every typable term is in SN and —because of what we have just mentioned— so in sn.

But before, we will need to show some properties about neutral terms. In addition, we need to define (small) $\to_{\mathsf{sn}}$ which is an equivalent relation to (big) $\to_{\mathsf{SN}}$ except it uses sn for the argument being normalizing instead of SN (in the β rule).

All lemmas in this section follow almost the same structure of the proof given in [15] —except we have to consider alpha-conversion.

Properties of Neutral Terms

So, to begin with, we now will turn our attention to some interesting properties about neutral terms and sn:

Lemma 4.4. *Let* $\mathsf{ne}\, M$. *Then:* 1. $M \to_\beta N \Rightarrow \mathsf{ne}\, N$, *and* 2. $\mathsf{sn}\, M \wedge \mathsf{sn}\, N \Rightarrow \mathsf{sn}\, MN$.

Proof. (4.4.1) By induction on $\mathsf{ne}\, M$. (4.4.2) By well-founded induction on the lexicographic order of $(\mathsf{sn}\, M, \mathsf{sn}\, N)$. Assume $MN \to_\beta P$ and proceed by cases analysis on P to prove $\mathsf{sn}\, P$. Case $P = M'N$ with $M \to_\beta M'$: using Lemma 4.4.1 we obtain $\mathsf{ne}\, M'$.

Then, by definition of $\mathsf{sn}\, M$ we get $\mathsf{sn}\, M'$, thus by ind. hyp., $\mathsf{sn}\, P$. Case $P = MN'$ with $N \to_\beta N'$: proceed analogously using $\mathsf{sn}\, N'$. □

Strongly Normalizing Head Reductions

Now, before showing (big) SN is sound respect of (small) sn, it is convenient to show $\to_{\mathsf{SN}}$ is sound respect to an analogous strategy that does not mention SN. Thus, we define the following:

Definition 4.5. $\to_{\mathsf{sn}}$ —to be pronounced (small) strongly normalizing head reduction or strong reduction for short— is the reduction relation defined as:

$$\beta\colon \frac{\mathsf{sn}\, N}{(\lambda x.M)N \to_{\mathsf{sn}} M[N/x]} \qquad \mathtt{appL}\colon \frac{M \to_{\mathsf{sn}} M'}{MN \to_{\mathsf{sn}} M'N} \qquad \alpha\colon \frac{M \to_{\mathsf{sn}} R \qquad R \sim_\alpha N}{M \to_{\mathsf{sn}} N}$$

That is, $\to_{\mathsf{sn}}$ and $\to_{\mathsf{SN}}$ are exactly the same except for the β rule.

Also, it will be convenient to show $\to_{\mathsf{sn}}$ and $\to_\beta$ are compatible in the sense that, if given M we can perform one beta-reduction step and arrive to P and one strong reduction step and go to N, then there must exist a common term Q such that both paths will eventually converge to it, from P in one step of $\to_{\mathsf{sn}}$ and from N in multiple steps (of any strategy):

Lemma 4.6. $M \to_{\mathsf{sn}} N \ \wedge\ M \to_\beta P \ \Rightarrow\ N \sim_\alpha P \ \vee\ (\exists Q)\ P \to_{\mathsf{sn}} Q \ \wedge\ N \twoheadrightarrow Q.$

Proof. By induction on the derivation of $M \to_{\mathsf{sn}} N$ and subordinate analysis of $M \to_\beta P$. So take $M \to_{\mathsf{sn}} N$ and proceed by cases:

Case β: Then $M = (\lambda x.R)S$, $N = R[S/x]$ and we have $\mathsf{sn}\, S$. We now proceed by (sub)cases on the derivation of $(\lambda x.R)S \to_\beta P$. **Sub-case** β: Then, we have $Q = P = R[S/x]$. **Sub-case** `appL`: Then $P = (\lambda x.R')S$ and we have $R \to_\beta R'$. Let $Q = R'[S/x]$. By definition of multi-step reduction (Definition 2.17) we have $R \twoheadrightarrow R'$, thus by compatibility of substitution (Lemma 2.28) we get $R[S/x] \twoheadrightarrow Q$. Hence, by β rule of definition of (small) strong reduction (Definition 4.5) we get $(\lambda x.R')S \to_{\mathsf{sn}} Q$, as desired. **Sub-case** `appR`: Then $P = (\lambda x.R)S'$ and we have $S \to_\beta S'$. Let $Q = R[S'/x]$. On the one hand, by extension of multi-step reduction up to substitutions (Definition 2.18) we have $[S/x] \twoheadrightarrow [S'/x]$ thus by Lemma 2.19.3 it follows $R[S/x] \twoheadrightarrow Q$. On the other hand, by definition of sn (Definition 4.1) —since we have $\mathsf{sn}\, S$ and $S \to_\beta S'$— if follows $\mathsf{sn}\, S'$, thus by β rule of sn we can get $(\lambda x.R)S' \to_{\mathsf{sn}} Q$.

Case appL: Then $M = RS$, $N = R'S$ and we have $R \to_{\mathsf{sn}} R'$. We now proceed by (sub)cases on the derivation of $RS \to_\beta P$. **Sub-case** β: Absurd, for in such case R would be an abstraction but we know strong reductions do not step into them. **Sub-case appL**: Then $P = R''S$ and we have $R \to_\beta R''$. By ind. hyp. with $R \to_{\mathsf{sn}} R'$ and $R \to_\beta R''$ we have two possibilities —one for each thesis. In the first case, we have $R' \sim_\alpha R''$, thus it is immediate that $R'S \sim_\alpha R''S$. In the second case, we have that there exists some Q' such that $R'' \to_{\mathsf{sn}} Q$ and $R' \twoheadrightarrow Q$. Let $Q = Q'S$. Then, by appL rule of definitions of $\to_{\mathsf{sn}}$ and $\to_\beta$ we have $R''S \to_{\mathsf{sn}} Q$ and $R'S \to_\beta Q$, respectively. Thus, with the latter and by definition of $\twoheadrightarrow$ we have $R'S \twoheadrightarrow Q$, as desired. **Sub-case appR**: Then $P = RS'$ and we have $S \to_\beta S'$. Let $Q = R'S'$. Then, by rule appL of strong reduction we have $RS' \to_{\mathsf{sn}} R'S'$ and by rule appR of beta-reduction together with definition of $\twoheadrightarrow$ we have $R'S \twoheadrightarrow R'S'$.

Case α: Then $M \to_{\mathsf{sn}} N$ follows form $M \to_{\mathsf{sn}} R$ and $R \sim_\alpha N$. The ind. hyp. gives us either $R \sim_\alpha P$ or there exists some Q s.t. $R \twoheadrightarrow Q$ and $P \to_{\mathsf{sn}} Q$. In the first case, by transitivity and symmetry of alpha-conversion on $R \sim_\alpha N$ and $R \sim_\alpha P$, we obtain $N \sim_\alpha P$. In the second case, by commutativity of alpha and beta using $R \sim_\alpha N$ and $R \twoheadrightarrow Q$ it follows $N \twoheadrightarrow S$ and $Q \sim_\alpha S$, for some S. Finally, by α rule using $P \to_{\mathsf{sn}} Q$ and $Q \sim_\alpha S$ we have $P \to_{\mathsf{sn}} S$, hence S was the term we were looking for. □

The Proof of Soundness

In this section we finally show the proof of the soundness of SN.

However, since $\to_{\mathsf{SN}}$ is part of the definition of SN it will be convenient to show the it is sound with respect to one that does not mention SN; that is, $\to_{\mathsf{sn}}$. For this last part we will need some preparatory result, namely that sn is backwardly closed under beta contraction —or as called in [15], *Weak Head Expansion*:

Lemma 4.7 (Weak Head Expansion)**.** $\mathsf{sn}\, N \ \wedge\ \mathsf{sn}\, Q \ \wedge\ Q \sim_\alpha M[N/x] \ \Rightarrow\ \mathsf{sn}\,(\lambda x.M)N$.

Proof. By well-founded induction on the lexicographic order of $(\mathsf{sn}\, N, \mathsf{sn}\, Q)$. Thus, assume $(\lambda x.M)N \to_\beta P$ for some P and then proceed by cases on the derivation of the reduction to prove $\mathsf{sn}\, P$.

Case α: Then $P = M[N/x]$. By closure of sn under alpha-conversion on the hypothesis $Q \sim_\alpha M[N/x]$ we have $\mathsf{sn}\, M[N/x]$, as desired. **Case appL**: Then $P = (\lambda x M')N$ and

$\lambda x.M \to_\beta \lambda x.M'$ follows from $M \to_\beta M'$. By compatibility (Lemma 2.27) on the latter we have $M[N/x] \to_\beta R$ and $R \sim_\alpha M'[N/x]$, for some R. Then, by commutativity of alpha and beta using hypothesis $Q \sim_\alpha M[N/x]$ and $M[N/x] \to_\beta R$ we have $Q \to_\beta S$ and $S \sim_\alpha R$, for some S. Thus, by definition of sn using $\mathsf{sn}\, Q$ and $Q \to_\beta S$ we have $\mathsf{sn}\, S$ —which precedes $\mathsf{sn}\, Q$ in the derivation, hence it can be used to obtain an ind. hyp. Besides, by transitivity of alpha-conversion using $S \sim_\alpha R$ and $R \sim_\alpha M'[N/x]$ we have $S \sim_\alpha M'[N/x]$, thus we can apply ind. hyp. with the latter, $\mathsf{sn}\, N$ and $\mathsf{sn}\, Q$ and therefore obtain $\mathsf{sn}\, (\lambda x M')N$, as desired. **Case** `appR`: Then $P = (\lambda x.M)N'$ and $N \to_\beta N'$. By Lemma 2.19.3 we have $M[N/x] \twoheadrightarrow M[N'/x]$ and by closure of sn under alpha-conversion using the hypothesis $Q \sim_\alpha M[N/x]$ we have $\mathsf{sn}\, M[N/x]$, thus by closure under multi-step reduction using the latter and $M[N/x] \twoheadrightarrow M[N'/x]$ we get $\mathsf{sn}\, M[N'/x]$. Finally, since $N \to_\beta N'$ we have $\mathsf{sn}\, N'$ is smaller than $\mathsf{sn}\, N$, thus we can apply ind. hyp. with $\mathsf{sn}\, N'$, $\mathsf{sn}\, M[N'/x]$ and $M[N'/x] \sim_\alpha M[N'/x]$ and obtain $\mathsf{sn}\, (\lambda x.M)N'$, as desired. □

We are almost ready to establish a more general result, i.e. sn is backwardly closed with respect to $\to_{\mathsf{sn}}$. Yet, we still need one more preliminary result, which is the following:

Lemma 4.8. *Let $M \to_{\mathsf{sn}} M'$ as well as M, N and $M'N$ be sn. Then $\mathsf{sn}\, MN$.*

Proof. By well-founded induction on the lexicographic order of $(\mathsf{sn}\, M, \mathsf{sn}\, N)$. Thus, assume $MN \to_\beta P$ and proceed by cases analysis to show $\mathsf{sn}\, P$:

Case `appL`: Then $P = M''N$ and $MN \to_\beta M''N$ follows from $M \to_\beta M''$. By Lemma 4.6, either $M' \sim_\alpha M''$ or there exists some Q such that $M' \twoheadrightarrow Q$ and $M'' \to_{\mathsf{sn}} Q$. The first case follows directly from the hypothesis $\mathsf{sn}\, M'N$ together with Lemma 4.2.1. In the second case, first notice that by Lemma 4.2.3 and by Definition 2.17 we get $\mathsf{sn}\, QN$. Then, by definition of $\mathsf{sn}\, M$, we have $\mathsf{sn}\, M''$ hence we can apply the ind. hyp. to obtain $\mathsf{sn}\, P$. **Case** `appR`: Then $P = MN'$ and $MN \to_\beta MN'$ follows from $N \to_\beta N'$. By definition of sn using $MN \to_\beta MN'$ with $\mathsf{sn}\, MN$ and $N \to_\beta N'$ with $\mathsf{sn}\, N$ we have $\mathsf{sn}\, MN'$ and $\mathsf{sn}\, N'$, thus by ind. hyp. we can derive $\mathsf{sn}\, MN'$, as desired. □

Finally, with these last two results we can prove $\to_{\mathsf{sn}}$ is backwardly closed under sn:

Lemma 4.9 (Backward Closure of sn)**.** $\mathsf{sn}\, N \ \wedge\ M \to_{\mathsf{sn}} N \ \Rightarrow\ \mathsf{sn}\, M$

Proof. By induction on the derivation of $M \to_{\mathsf{sn}} N$. **Case** β: Using Lemma 4.7. **Case** `appL`: Assume $M \to_{\mathsf{sn}} M'$ and $\mathsf{sn}\, M'N$. By Lemma 4.2.4 we have $\mathsf{sn}\, M'$ and $\mathsf{sn}\, N$. Then,

by ind. hyp. we have $\mathsf{sn}\, M$. Finally, using Lemma 4.8, we get $\mathsf{sn}\, MN$. **Case** α: Use Lemma 4.2.1 and then the ind. hyp. □

Now, as to the complete proof —the soundness of SN— we only need two more preparatory results:

Lemma 4.10.

1. $\mathsf{SNe}_x\, M \Rightarrow \mathsf{ne}_x\, M$ (Strong Neutral Weakening)
2. $M \to_{\mathsf{SN}} N \Rightarrow M \twoheadrightarrow N$ (Reduction Weakening).

Proof. (4.10.1) By simple induction on $\mathsf{SNe}_x\, M$. (4.10.2) By simple induction on the derivation of $M \to_{\mathsf{SN}} N$. **Case** α: Then $M \to_{\mathsf{SN}} N$ follows from $M \to_{\mathsf{SN}} P$ and $P \sim_\alpha N$. By ind. hyp. we have $M \twoheadrightarrow N$. Also, by Definition 2.17 we get $P \twoheadrightarrow N$. Then, by transitivity of $\sim_\alpha$ we have $M \twoheadrightarrow N$. **Case** β: By Definition 2.17. **Case** `appL`: $MN \to_{\mathsf{SN}} M'N$ is obtained from $M \to_{\mathsf{SN}} M'$. By ind. hyp. we have $M \twoheadrightarrow N$ hence, by Lemma 2.19.4 we have $MN \twoheadrightarrow M'N$. □

Finally, we can show SN is sound respect to sn:

Theorem 4.11 (Soundness of SN)**.**

1. $\mathsf{SN}\, M \;\Rightarrow\; \mathsf{sn}\, M$
2. $\mathsf{SNe}_x\, M \;\Rightarrow\; \mathsf{sn}\, M$
3. $M \to_{\mathsf{SN}} N \;\Rightarrow\; M \to_{\mathsf{sn}} N$.

Proof. By simultaneous induction, following Definition 4.3. **Case** `v`: By Lemma 4.2.2. **Case** `app`: Then $M = PQ$ and $\mathsf{SNe}_x\, (PQ)$ is obtained from $\mathsf{SNe}_x\, P$ and $\mathsf{SN}\, Q$. By ind. hyp. we have $\mathsf{sn}\, P$ and $\mathsf{sn}\, Q$, and by Lemma 4.10.1 we get $\mathsf{ne}\, P$, thus by Lemma 4.4.2 we obtain $\mathsf{sn}\, PQ$. **Case** `ne`: By ind. hyp. **Case** λ: By ind. hyp. followed by Lemma 4.2.5. **Case** `exp`: By ind. hyp. followed by Lemma 4.9. **Case** β: By ind. hyp. and Definition 4.5. **Case** `appL`: By ind. hyp. and Definition 4.5. □

4.4 The Strong Normalization Theorem

We can now extend the Weak Normalization Theorem (3.2) up to SN, and since SN implies sn, then every typable term must be in sn as well:

Theorem 4.12 (Strong Normalization)**.** $\Gamma \vdash M : \alpha \;\Rightarrow\; \mathsf{SN}\, M.$

The proof proceeds exactly the same as in the theorem for WN except for the case of application, where instead of using Lemma 3.4, we will we use the following adaptation to SN:

Lemma 4.13. *Let* $\Gamma \vdash M : \alpha$, $\Gamma \vdash N : \beta$, $\mathsf{SN}\, N$, $\sigma : \Gamma \rightharpoonup \Delta \restriction M$, $\mathsf{Unary}\, \sigma\, x\, N$ *and* $\Gamma \vdash x : \beta$. *Then:*

1. $\mathsf{SNe}_{\,y}\, M \Rightarrow (i)\ \mathsf{SN}\, M\sigma,\ (ii)\ y{\neq}x \Rightarrow \mathsf{SNe}_{\,\sigma y}\, M\sigma$ *and* $(iii)\ \alpha = \beta{\to}\gamma \Rightarrow \mathsf{SN}\, MN.$
2. $\mathsf{SN}\, M \Rightarrow (i)\ \mathsf{SN}\, M\sigma$ *and* $(ii)\ \alpha = \beta{\to}\gamma \Rightarrow \mathsf{SN}\, MN.$
3. $M \to_{\mathsf{SN}} P \Rightarrow M\sigma \to_{\mathsf{SN}} P\sigma.$

Proof. Again, the proof proceeds by complete induction on β and subordinate induction on the definition of SN (Def. 4.3). We will only show the cases corresponding to the additional rules of SN —which are not in WN— plus `exp` which is a bit different; the remaining cases are exactly the same:

Case `exp`: Then $\mathsf{SN}\, M$ follows from $M \to_{\mathsf{SN}} P$ and $\mathsf{SN}\, P$. Also, we have $\Gamma \vdash M : \alpha$. Then, we need to show 2: (i) $\mathsf{SN}\, M\sigma$ and (ii) $\alpha{=}\beta{\to}\gamma \supset \mathsf{SN}\, MN$. As to (i), by the main ind. hyp. 3 we have $M\sigma \to_{\mathsf{SN}} P\sigma$ and by ind. hyp. 2(i) $\mathsf{SN}\, P\sigma$. Thus by rule `exp` we obtain $\mathsf{SN}\, M\sigma$, as desired. As to (ii), by reduction weakening (Lemma 4.10.2) we have $M \twoheadrightarrow P$, therefore by subject reduction (Lemma 2.24) $\Gamma \vdash P : \beta{\to}\gamma$. Thus, we can apply ind. hyp. (ii) on $\mathsf{SN}\, P$ and have $\mathsf{SN}\, PN$, hence by rule `appL` applied to $M \to_{\mathsf{SN}} P$ we get $MN \to_{\mathsf{SN}} PN$. Finally, by `exp` we can derive $\mathsf{SN}\, MN$, as desired.

Case β: Then $(\lambda x.M)N \to_{\mathsf{SN}} M[N/x]$ follows from $\mathsf{SN}\, N$ and we need to show 3: $((\lambda x.M)N)\sigma \to_{\mathsf{SN}} M[N/x]\sigma$. First, notice that by definition of substitution (Def. 2.8) we have $((\lambda x.M)N)\sigma = (\lambda z.M(\sigma, z/x))N\sigma$ with $z = \chi(\sigma, \lambda x.M)$. In addition, by the main ind. hyp. 1(i) we have $\mathsf{SN}\, N\sigma$, thus we can apply β rule of SN (Def. 4.3) and have $(\lambda z.M(\sigma, z/x))N\sigma \to_{\mathsf{SN}} M(\sigma, z/x)[N\sigma/z]$. Besides, by Chi Lemma (Lem. 2.9.1) we have $z\#(\sigma, \lambda y M)$, hence we can use Corollary 2.15.1 and have $M(\sigma, z/x)[N\sigma/z] \sim_\alpha M[N/x]\sigma$. Finally, we can use rule α of SN and obtain $((\lambda x.M)N)\sigma \to_{\mathsf{SN}} M[N/x]\sigma$, as desired.

Case `appL`: Then $MN \to_{\mathsf{SN}} M'N$ follows from $M \to_{\mathsf{SN}} M'$ and we must show 3: $(MN)\sigma \to_{\mathsf{SN}} (M'N)\sigma$. By ind. hyp. we have $M\sigma \to_{\mathsf{SN}} M'\sigma$, thus by `appL` rule we can apply $N\sigma$ on both sides and obtain $M\sigma N\sigma \to_{\mathsf{SN}} M'\sigma N\sigma$ which by definition of substitution equals $(MN)\sigma \to_{\mathsf{SN}} (M'N)\sigma$.

Case α: Then $M \to_{\mathsf{SN}} P$ follows from $M \to_{\mathsf{SN}} N$ and $N \sim_\alpha P$, and we need to show 3: $M\sigma \to_{\mathsf{SN}} P\sigma$. By ind. hyp. we have $M\sigma \to_{\mathsf{SN}} N\sigma$. Besides, by equalization (Lemma 2.14.10) we get $N\sigma = P\sigma$. Then, by reflexivity of alpha-conversion (Lemma. 2.14.3) we can obtain $N\sigma \sim_\alpha P\sigma$, and finally by α rule of SN we can derive $M\sigma \to_{\mathsf{SN}} P\sigma$, as desired. □

Chapter 5

Conclusions

We were able to fully formalize the Weak and Strong Normalization Theorem for the Lambda Calculus in first-order syntax with only one sort of names and explicitly considering alpha-conversion. The result should be assessed in connection with the expectations stated in the Introduction. We assign relevance first of all to the complexity of the formalized version, both in Agda and in mathematical English.

Concerning the latter, we believe that this method makes it possible to come closer to a presentation in textbook style that does not hide details that need formalization. Of course, the main price paid has been the explicit handling of alpha-conversion; this has shown itself mainly in the formulation of the main definitions and results, which often must replace identity of terms by their alpha conversion, due to the renaming implicit in the effect of substitution on abstractions. As to additional, housekeeping lemmas, these seem to have been circumscribed to results of closure under alpha conversion.

Another aspect of the complexity has to do with the kind of proof methods required: when dealing with terms, we have been able to proceed by using only structural induction. In addition to this, we used of course induction on the various predicates and relations introduced, among which there is the somewhat complex mutual Definitions 3.1 and 4.3 of Sections 3.1 and 4.2, as well as the well-founded induction on types which was the main method in the proof we chose to formalize.

On the other hand, the size of the Agda code has by no means exploded: it is of about 800 lines, which approximately 300 belongs to the Weak Normalization Theorem of Chapter 3 and some auxiliary lemmas, and the remaining 500 lines for the Strong Normalization Theorem split almost in halves between the proof of soundness of the inductive

characterization of the strongly normalizing terms of Section 4.3 and the theorem itself of Section 4.4. The complexity of the development can also be assessed by the effort required: about 500 hours of work. This of course reveals a quite high cost for each Agda line of code.

Our pre-existing library providing the basics for the formalization of the calculus by the method chosen was only marginally updated. The required new results and definitions are shown in Sections 2.6, 2.7, 2.8 and 2.9. The library was readily understood and made use of by the author of the formalization, despite his not having much prior experience in Agda (this was of only one tutorial course).

We believe we have made progress towards developing a corpus of formalized meta-theory of the Lambda Calculus somewhat along the lines of [30]. We also think that the present approach allows to aspire at achieving such a goal using the sort of explicit mathematical English we have used above, in turn supported by the Agda version.

This work should be compared to other formalizations of strong normalization, most notably those referred to in [15]. They employ variants of higher-order abstract syntax and of the de Bruijn notation, both of which of course dispense with alpha conversion. Besides, as far to our knowledge, there is not yet formalization of our approach —using first-order syntax with one sort of names. A full discussion is outside the scope of the present thesis, but should be carried out. We should for that matter experiment with the full challenge posed in [15], using Kripke-style logical relations for proving strong normalization. The use of typed reduction will then force us to reformulate the calculus. We also have tried a formalization in Constructive Type Theory using Agda of principles of induction implementing the Barendregt variable convention –see [31, 32], somewhat similarly to [33]. Again, we are still short of a full comparison in this respect.

Finally, we would like to make some observations with regard to the code written and how it can be improved, and further about our very concrete approach of the calculus we have chosen, specifically in account of substitutions. The first consideration is about our notion of unary substitutions (Definition 2.31) introduced in Chapter 2 as a manner of formalizing Joachimksi and Matthes' proof [6] in Lemmas 3.4 and 4.13; to put it roughly, the proof requires a lemma which states that $\mathsf{SN}\,M$ and $\mathsf{SN}\,N$ implies $\mathsf{SN}\,M[N/x]$. As to substitutions —and already established— they arise from the process of beta-contracting redexes, viz. $(\lambda x.M)N \to_\beta M[N/x]$, being its purpose to replace the (free) occurrences of *one* variable —the name of the argument— with a term —the argument itself. However, as the substitution travels across the structure of the term, some abstractions might be

encountered where in such case a renaming is performed, leading to the crafting of a new *multiple* substitution made up of the original plus an update, viz. $\lambda z.M[N/x, z/y]$. Nevertheless, the substitution is in essence still unary. Thus we now would like to propose, as an experiment to see drawbacks and benefits, to consider directly defining the set of substitution to —instead of $\Sigma = \mathsf{V} \to \Lambda$ as currently is in the framework [1]— a less general characterization, e.g. $\Sigma = \mathsf{V} \times \Lambda \times \mathsf{V} \to \mathsf{V}$, i.e. a 3-tuple composed of the name of the argument, the argument itself and a renaming (given as a mapping from V to V) and therefore dispense of Definition 2.31.

Another observation is respect to the (main) Lemmas 3.4 and 4.13. There, we believe at least three improvements might have been made. First of all, actually it is not mandatory to have as part of the lemma the thesis 1(ii) —which states neutrability is preserved under certain unary substitutions— that is to say, it could and it should be extracted, thus reducing the complexity of the lemma. In fact, we have discovered it while coding it and so we also uploaded the suggested version to the repository, as said earlier somewhere in this thesis. We believe we should have presented the latter in this thesis, if we had the time. Secondly, the typing judgments of M and x, and the hypothesis $\sigma : \Gamma \rightharpoonup \Delta \lfloor M$ may be replaced by two different hypothesis, each of one to be placed only in one of the thesis; as to thesis 1 and 2(i), we could have used $\Gamma \vdash M\sigma : \alpha \wedge (x, \beta) \in \Gamma$ instead, and as to 1(iii) and 2(ii), $\Gamma \vdash MN : \alpha$ besides of dropping the hypothesis $\alpha = \beta \to \gamma$. And finally, a third observation is in regard to Lemma 2.11.2. It could be replaced by a proof that Definition 4.3 is closed under alpha conversion, hence the case λ in the proof of Lemma 3.4 will have shorten significantly, as the sub-case $y = x$ would be solved by just using this new closure lemma.

Lastly, in the repository a version tagged "buggy" can be found. In this version of the Strong Normalization Theorem, the induction is done on the type of β and subordinate induction straightly on the derivation of SN —as opposed to the main version where the subordinate inductions is done on the height of the derivation instead. This is because for some reason, if the induction was directly made in the derivation of SN, Agda's termination checking phase would fail. Using the computed height seems to solve the issue. We really do not know if it is a bug or else and neither we reported it. To investigate the cause of this anomaly is a pending task.

Bibliography

[1] E. Copello, N. Szasz, and Álvaro Tasistro, "Formal Metatheory of the Lambda calculus using Stoughton's Substitution," *Theoretical Computer Science*, vol. 685, pp. 65 – 82, 2017.

[2] P. Martin-Löf, *Intuitionistic type theory*, ser. Studies in Proof Theory. Lecture Notes. Bibliopolis, Naples, 1984, vol. 1, notes by Giovanni Sambin.

[3] U. Norell, "Towards a Practical Programming Language Based on Dependent Type Theory," Ph.D. dissertation, Department of Computer Science and Engineering, Chalmers University of Technology, September 2007.

[4] A. Stoughton, "Substitution Revisited," *Theoretical Computer Science*, vol. 59, pp. 317–325, 1988.

[5] M. Copes, N. Szasz, and A. Tasistro, "Formalization in Constructive Type Theory of the Standardization Theorem for the Lambda Calculus using Multiple Substitution," in Proceedings of the 13th International Workshop on *Logical Frameworks and Meta-Languages: Theory and Practice,* Oxford, UK, 7th July 2018, ser. Electronic Proceedings in Theoretical Computer Science, F. Blanqui and G. Reis, Eds., vol. 274. Open Publishing Association, 2018, pp. 27–41.

[6] F. Joachimski and R. Matthes, "Short Proofs of Normalization for the Simply- Typed λ-Calculus, Permutative Conversions and Gödel's T," *Archive for Mathematical Logic*, vol. 42, no. 1, pp. 59–87, 2003.

[7] "The 15th International Workshop on Logical and Semantic Frameworks with Applications," 2020. [Online]. Available: http://lsfa2020.ufba.br

[8] S. Urciuoli, Álvaro Tasistro, and N. Szasz, "Strong normalization for the simply-typed lambda calculus in constructive type theory using agda," *Electronic Notes in*

Theoretical Computer Science, vol. 351, pp. 187 – 203, 2020, proceedings of LSFA 2020, the 15th International Workshop on Logical and Semantic Frameworks, with Applications (LSFA 2020).

[9] H. B. Curry and R. Feys, *Combinatory Logic, Volume I.* North-Holland, 1958, second printing 1968.

[10] W. Tait, "Intensional Interpretations of Functionals of Finite Type I," *Journal of Symbolic Logic*, vol. 32, no. 2, p. 198–212, 1967.

[11] J. Girard, Y. Lafont, and P. Taylor, *Proofs and Types*, ser. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1989.

[12] L. Skorstengaard, "An introduction to logical relations," *CoRR*, vol. abs/1907.11133, 2019. [Online]. Available: http://arxiv.org/abs/1907.11133

[13] F. van Raamsdonk and P. Severi, "On Normalisation," Centrum voor Wiskunde en Informatica (CWI), Amsterdam, Tech. Rep., 1995.

[14] T. Altenkirch, "A Formalization of the Strong Normalization Proof for System F in LEGO," *LNCS*, vol. 664, 07 1996.

[15] A. Abel, G. Allais, A. Hameer, B. Pientka, A. Momigliano, S. Schäfer, and K. Stark, "POPLMark reloaded: Mechanizing Proofs by Logical Relations," *Journal of Functional Programming*, vol. 29, 2019.

[16] R. David and K. Nour, "A short proof of the strong normalization of classical natural deduction with disjunction," *The Journal of Symbolic Logic*, vol. 68, no. 4, pp. 1277–1288, 2003.

[17] K. Donnelly and H. Xi, "A Formalization of Strong Normalization for Simply-Typed Lambda-Calculus and System F," *Electronic Notes in Theoretical Computer Science*, vol. 174, no. 5, pp. 109 – 125, 2007, proceedings of the First International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2006).

[18] C. Chen and H. Xi, "Combining Programming with Theorem Proving," *ACM SIGPLAN Notices*, vol. 40, no. 9, pp. 66–77, 2005.

[19] F. Pfenning and C. Elliott, "Higher-order abstract syntax," *ACM sigplan notices*, vol. 23, no. 7, pp. 199–208, 1988.

[20] A. Abel, "Normalization for the simply-typed lambda-calculus in twelf," *Electronic Notes in Theoretical Computer Science*, vol. 199, pp. 3 – 16, 2008, proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages (LFM 2004).

[21] F. Pfenning and C. Schürmann, "System description: Twelf—a meta-logical framework for deductive systems," in *International Conference on Automated Deduction*. Springer, 1999, pp. 202–206.

[22] The LEGO Team, "The LEGO Proof Assistant Home Page." [Online]. Available: http://www.dcs.ed.ac.uk/home/lego

[23] N. G. de Bruijn, "Lambda-Calculus Notation with Nameless Dummies, A Tool for Automatic Formula Manipulation, with Applications to the Church-Rosser Theorem," *Indagationes Mathematicae (Koninglijke Nederlandse Akademie van Wetenschappen)*, vol. 34, no. 5, pp. 381–392, 1972.

[24] U. Berger, S. Berghofer, P. Letouzey, and H. Schwichtenberg, "Program Extraction from Normalization Proofs," *Studia Logica*, vol. 82, pp. 25–49, 02 2006.

[25] H. Schwichtenberg, "Minimal logic for computable functions," in *Logic and Algebra of Specification*, F. L. Bauer, W. Brauer, and H. Schwichtenberg, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 289–320.

[26] The Coq Development Team, *The Coq proof assistant reference manual*, LogiCal Project, 2012, version 8.0. [Online]. Available: http://coq.inria.fr

[27] T. Nipkow, *Programming and Proving in Isabelle/HOL*, 2016. [Online]. Available: https://isabelle.in.tum.de/doc/prog-prove.pdf

[28] A. Koprowski, "A Formalization of the Simply Typed Lambda Calculus in Coq," 2006. [Online]. Available: http://color.inria.fr/papers/koprowski06draft.pdf

[29] B. Pientka and J. Dunfield, "Beluga: A framework for programming and reasoning with deductive systems (system description)," in *Automated Reasoning*, J. Giesl and R. Hähnle, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 15–21.

[30] T. Nipkow and S. Berghofer, "Fundamental Properties of Lambda-calculus," 2019. [Online]. Available: https://www.researchgate.net/publication/250220907_Fundamental_Properties_of_Lambda-calculus_Contents

[31] E. Copello, Álvaro Tasistro, N. Szasz, A. Bove, and M. Fernández, "Alpha-Structural Induction and Recursion for the Lambda Calculus in Constructive Type Theory," *Electronic Notes in Theoretical Computer Science*, vol. 323, pp. 109 – 124, 2016.

[32] E. Copello, N. Szasz, and Álvaro Tasistro, "Machine-checked Proof of the Church-Rosser Theorem for the Lambda Calculus Using the Barendregt Variable Convention in Constructive Type Theory," *Electronic Notes in Theoretical Computer Science*, vol. 338, pp. 79 – 95, 2018, the 12th Workshop on Logical and Semantic Frameworks, with Applications (LSFA 2017).

[33] C. Urban and C. Tasson, "Nominal Techniques in Isabelle/HOL," in *Automated Deduction – CADE-20*, ser. Lecture Notes in Computer Science, R. Nieuwenhuis, Ed. Springer Berlin Heidelberg, 2005, vol. 3632, pp. 38–53.

Appendix A

Code

```
module Neutral where

open import Term
open import Chi
open import Substitution
open import Alpha
open import Beta
open import Relation Λ

open import Data.Sum
open import Relation.Binary.PropositionalEquality

_↠_ = _→α*_

-- Neutral/normal

data ne : V → Λ → Set
data nf : Λ → Set

data ne where
  var : ∀ {x} → ne x (v x)
  app : ∀ {x M N} → ne x M → nf N → ne x (M · N)

data nf where
```

```
25    nfe : ∀ {M x} → ne x M → nf M
26    abs : ∀ {x M} → nf M → nf (ƛ x M)
27
28  -- Weak-neutral
29
30  data wne : V → Λ → Set where
31    var   : ∀ {x} → wne x (v x)
32    app : ∀ {x M N} → wne x M → wne x (M · N)
33
34  -- Lemmas
35
36  closure-wne→β : ∀ {x M N} → wne x M → M →β N → wne x N
37  closure-wne→β var (ctxinj ())
38  closure-wne→β (app wneM) (ctx·l M→N) = app (closure-wne→β wneM M→N)
39  closure-wne→β (app wneM) (ctx·r _) = app wneM
40  closure-wne→β (app ()) (ctxinj ▷β)
41
42  closure-wne∼α : ∀ {x M N} → wne x M → M ∼α N → wne x N
43  closure-wne∼α var ∼v = var
44  closure-wne∼α (app wneM) (∼· M∼N _) = app (closure-wne∼α wneM M∼N)
45
46  lemma₁ : ∀ {x M N} → wne x M → M ↠ N → wne x N
47  lemma₁ wneM refl = wneM
48  lemma₁ wneM (just (inj₁ M→N)) = closure-wne→β wneM M→N
49  lemma₁ wneM (just (inj₂ M∼N)) = closure-wne∼α wneM M∼N
50  lemma₁ wneM (trans M→N N→P) = lemma₁ (lemma₁ wneM M→N) N→P
51
52  corollary₁ : ∀ {x M} → wne x M → nf M → ne x M
53  corollary₁ var _ = var
54  corollary₁ (app wneM) (nfe (app neM nfN)) = app (corollary₁ wneM (nfe neM)) nfN
55
56  lemma₃ : ∀ {σ x y M} → wne x M → σ x ≡ v y → wne y (M • σ)
57  lemma₃ {σ} {x} var σx≡y with σ x
58  lemma₃ var refl | v y = var
59  lemma₃ (app wneM) σx≡y = app (lemma₃ wneM σx≡y)
```

End of Neutral.agda

```
module TypeLemmas where

open import SoundnessSN
open import Term
open import Data.Sum
open import Neutral

open import Relation.Binary.PropositionalEquality hiding (trans)
open import Induction.WellFounded
open import Data.Unit

data _τ<_ : Type → Type → Set where
  τ<l : ∀ {α β} → α τ< (α ⟶ β)
  τ<r : ∀ {α β} → β τ< (α ⟶ β)

wfτ< : Well-founded _τ<_
wfτ< τ = acc λ y ()
wfτ< (A ⟶ B) = acc wfτ<-aux
  where wfτ<-aux : (𝔶 : Type) → 𝔶 τ< (A ⟶ B) → Acc _τ<_ 𝔶
        wfτ<-aux .A τ<l = wfτ< A
        wfτ<-aux .B τ<r = wfτ< B

open Transitive-closure _τ<_

_τ<⁺_ = _<⁺_

wfτ<⁺ : Well-founded _τ<⁺_
wfτ<⁺ = well-founded wfτ<

lemma-≡Γx : ∀ {α β Γ x} → Γ ⊢ v x : α → Γ ⊢ v x : β → α ≡ β
lemma-≡Γx (⊢v x∈Γ₁) (⊢v x∈Γ₂) = lemma∈! x∈Γ₁ x∈Γ₂

lemma-τ≤ : ∀ {α β Γ M x} → wne x M → Γ ⊢ v x : β → Γ ⊢ M : α → α τ<⁺ β ⊎ α ≡ β
lemma-τ≤  var hdM:β hdM:α = inj₂ (lemma-≡Γx hdM:α hdM:β)
lemma-τ≤  {A} {B} (app M) hdM:β (⊢· M:𝔶→α _) with lemma-τ≤ M hdM:β M:𝔶→α
... | inj₁ 𝔶→α<β = inj₁ (trans [ τ<r ] 𝔶→α<β)
```

```
lemma-τ≤ {A} .{𝔜 ⟶ A} (app _) hdM:β (⊢· {𝔜} M:𝔜→α _) | inj₂ refl = inj₁ [
↪  τ<r ]

lemma-τ< : ∀ {α 𝔜 β Γ M x} → wne x M → Γ ⊢ v x : β → Γ ⊢ M : α ⟶ 𝔜 → α τ<⁺ β
lemma-τ< M⇓ hdM:β M:α→𝔜 with lemma-τ≤ M⇓ hdM:β M:α→𝔜
... | inj₁ α→𝔜<β = trans [ τ<l ] α→𝔜<β
lemma-τ< {A} {𝔜} .{A ⟶ 𝔜} M⇓ hdM:β M:α→𝔜 | inj₂ refl = [ τ<l ]
```

End of TypeLemmas.agda

```
module Unary where

open import Chi
open import Term
open import Substitution
open import ListProperties

open import Data.Nat
open import Data.Sum
open import Data.Product hiding (Σ)
open import Relation.Binary.PropositionalEquality
open import Relation.Nullary
open import Data.Empty

data IsVar : Λ → Set where
  isv : ∀ {x} → IsVar (v x)

Unary : Σ → V → Λ → Set
Unary σ x M = σ x ≡ M × ∀ {y} → y ≢ x → IsVar (σ y)

unary : ∀ {x M} → Unary (ι ≺+ (x , M)) x M
unary {x} {M} with x ≟ x
... | no x≢x = α-elim (x≢x refl)
... | yes refl = refl , aux
  where aux : {y : V} → y ≢ x → IsVar ((ι ≺+ (x , M)) y)
        aux {y} _ with x ≟ y
        ... | no _ = isv
```

```
129         aux {.x} y≢x | yes refl = α-elim (y≢x refl)
130
131 unary≺+≡ : ∀ {x y σ M} → Unary σ x M → Unary (σ ≺+ (x , v y)) x (v y)
132 unary≺+≡ {x} {y} {σ} Unyσ with x ≟ x
133 ... | no x≢x = α-elim (x≢x refl)
134 ... | yes _ = refl , aux
135   where aux : {z : V} → z ≢ x → IsVar ((σ ≺+ (x , v y)) z)
136         aux {z} z≢x with x ≟ z
137         ... | no _ = proj₂ Unyσ z≢x
138         aux {.x} z≢x | yes refl = α-elim (z≢x refl)
139
140 unary≺+≢ : ∀ {x y z σ M} → y ≢ x → Unary σ x M → Unary (σ ≺+ (y , v z)) x M
141 unary≺+≢ {x} {y} {z} {σ} y≢x Unyσ with y ≟ x
142 ... | yes y≡x = α-elim (y≢x y≡x)
143 ... | no _ = proj₁ Unyσ , aux
144   where aux : {w : V} → w ≢ x → IsVar ((σ ≺+ (y , v z)) w)
145         aux {w} w≢x with y ≟ w
146         ... | no _ = proj₂ Unyσ w≢x
147         aux {.y} _ | yes refl = isv
148
149 unaryv : ∀ {x y z σ M} → Unary σ x (v y) → Unary (σ ≺+ (z , M)) z M
150 unaryv {x} {y} {z} Unyσ with z ≟ z
151 ... | no z≢z = α-elim (z≢z refl)
152 unaryv {x} {y} {z} {σ} {M} Unyσ | yes refl = refl , aux
153   where aux : {w : V} → w ≢ z → IsVar ((σ ≺+ (z , M)) w)
154         aux {w} _ with z ≟ w
155         ... | no _ with x ≟ w
156         ... | no x≢w = proj₂ Unyσ (sym≢ x≢w)
157         aux {.x} _ | no _ | yes refl with σ x | proj₁ Unyσ
158         ... | v .y | refl = isv
159         aux {.z} w≢z | yes refl = α-elim (w≢z refl)
```

End of Unary.agda

```
160 module WeakNormalization where
161
162 open import Term
```

```
163 open import Chi
164 open import Substitution
165 open import Data.Sum
166 open import Data.Product hiding (Σ)
167 open import Beta
168 open import Alpha
169 open import SubstitutionLemmas
170 open import ListProperties
171 open import Relation using (just; trans) renaming (refl to reflR)
172 open import Unary
173 open import TypeLemmas
174 open import ParallelReduction
175 open import SubstitutionCompatibilityLemmas hiding (_↠_)
176 open import Neutral
177
178 open import Data.Nat hiding (_*_)
179 open import Relation.Binary.PropositionalEquality renaming (trans to trans≡)
180 open import Induction.WellFounded
181 open import Data.Unit hiding (_≟_)
182 open import Data.Empty
183 open import Relation.Nullary
184 open import Induction.Nat
185 open import Data.Nat.Properties
186 open import Relation.Binary hiding (_⇒_)
187 open import Algebra.Structures
188
189 -- Well-foundness proofs
190
191 open Lexicographic _<⁺_ (λ _ m n → m <' n) renaming (_<_ to _<ℕ,τ_ ;
    ↪  well-founded to wfΣ)
192
193 wf<ℕ,τ : Well-founded _<ℕ,τ_
194 wf<ℕ,τ = wfΣ wf<⁺ <-well-founded
195
196 -- Definitions
197
198 data WN : Λ → Set
```

```
data WNe : V → Λ → Set

data WNe where
  var : ∀ {x} → WNe x (v x)
  app : ∀ {x M N} → WNe x M → WN N → WNe x (M · N)

data WN where
  wke : ∀ {M x} → WNe x M → WN M
  abs : ∀ {x M} → WN M → WN (ƛ x M)
  exp : ∀ {M N} → M ⇛ N → WN N → WN M

heightNe : ∀ {M x} → WNe x M → ℕ
height   : ∀ {M} → WN M → ℕ

heightNe var = 0
heightNe (app M⇓ N⇓) = suc (heightNe M⇓ ⊔ height N⇓)

height (abs M⇓) = suc (height M⇓)
height (wke M⇓) = suc (heightNe M⇓)
height (exp _ M⇓) = suc (height M⇓)

-- Auxiliary lemmas

m<'m⊔n+1 : ∀ m n → m <' suc (m ⊔ n)
m<'m⊔n+1 m n = s≤'s (≤⇒≤' (m≤m⊔n m n))

⊔-comm = IsCommutativeMonoid.comm
  ↪ (IsCommutativeSemiringWithoutOne.+-isCommutativeMonoid
  ↪ ⊔-⊓-0-isCommutativeSemiringWithoutOne)

m<'n⊔m+1 : ∀ m n → m <' suc (n ⊔ m)
m<'n⊔m+1 m n with n ⊔ m | ⊔-comm n m
m<'n⊔m+1 m n | .(m ⊔ n) | refl = m<'m⊔n+1 m n

lemmaσ⌊· : ∀ {σ Γ Δ P Q} → σ : Γ ⇁ Δ ⌊ P · Q → (σ : Γ ⇁ Δ ⌊ P) × (σ : Γ ⇁
  ↪ Δ ⌊ Q)
lemmaσ⌊· σ⌊PQ = (λ x*P → σ⌊PQ (*·l x*P)) , (λ x*Q → σ⌊PQ (*·r x*Q))
```

```
WNe⇒wne : ∀ {x M} → WNe x M → wne x M
WNe⇒wne var = var
WNe⇒wne (app M⇓ _) = app (WNe⇒wne M⇓)

ne⇒WNe : ∀ {x M} → ne x M → WNe x M
nf⇒WN : ∀ {M} → nf M → WN M

ne⇒WNe var = var
ne⇒WNe (app neM nfN) = app (ne⇒WNe neM) (nf⇒WN nfN)

nf⇒WN (nfe neM) = wke (ne⇒WNe neM)
nf⇒WN (abs nfM) = abs (nf⇒WN nfM)

-- Soundness WN

wn : Λ → Set
wn M = ∃ λ N → M ↠ N × nf N

sound-WN : ∀ {M} → WN M → wn M
sound-WNe : ∀ {x M} → WNe x M → wn M

sound-WN (wke M⇓) = sound-WNe M⇓
sound-WN (abs {x} M⇓) =
  let N , M→N , nfN = sound-WN M⇓
  in ƛ x N , abs-star M→N , abs nfN
sound-WN (exp M→N N⇓) =
  let P , N→P , nfP = sound-WN N⇓
  in P , trans M→N N→P , nfP

sound-WNe (var {x}) = v x , reflR , nfe var
sound-WNe (app M⇓ N⇓) =
  let M' , M→M' , nfM' = sound-WNe M⇓
      N' , N→N' , nfN' = sound-WN N⇓
      wneM' = lemma₁ (WNe⇒wne M⇓) M→M'
      neM' = corollary₁ wneM' nfM'
  in M' · N' , trans (app-star-l M→M') (app-star-r N→N') , nfe (app neM' nfN')
```

```
-- Main lemma

WN-lemma : ∀ {M Γ α β N}
         → (M⇓ : WN M)
         → Acc _<_ (β , height M⇓)
         → Γ ⊢ M : α
         → WN N
         → (∀ {σ Δ} → σ : Γ ⇀ Δ ⌊ M → Unary σ N Γ β → WN (M • σ))
           × ((∃ λ ɣ → α ≡ β ⟶ ɣ) → Γ ⊢ N : β → WN (M · N))

WN-lemmaNe : ∀ {M Γ α β x N}
           → (M⇓ : WNe x M)
           → Acc _<_ (β , heightNe M⇓)
           → Γ ⊢ M : α
           → WN N
           → (∀ {σ Δ} → σ : Γ ⇀ Δ ⌊ M → Unary σ N Γ β → WN (M • σ))
             × ((∃ λ ɣ → α ≡ β ⟶ ɣ) → Γ ⊢ N : β → WN (M · N))

WN-lemmaNe .{v x} {Γ} {_} {B} {.x} {N} (var {x}) _ _ N⇓  = thesis₁ , λ _ _ →
↪  wke (app var N⇓)
  where thesis₁ : ∀ {σ} {Δ} → σ : Γ ⇀ Δ ⌊ (v x) → Unary σ N Γ B → WN (v x • σ)
        thesis₁ {σ} _ Unyσ with σ x | Unyσ {x}
        ... | .(v y) | inj₁ (isv {y}) = wke var
        ... | _ | inj₂ (refl , _) = N⇓
WN-lemmaNe {P · Q} {Γ} {_} {B} {.x} {N} (app {x} P⇓ Q⇓) (acc hi) (⊢· {ɣ} {ε}
↪  P:ɣ→ε Q:ɣ) N⇓ =
  thesis₁ , λ _ _ → wke (app (app P⇓ Q⇓) N⇓)
    where
       thesis₁ : ∀ {σ Δ} → σ : Γ ⇀ Δ ⌊ P · Q → Unary σ N Γ B → WN (P · Q • σ)
        thesis₁ {σ} {Δ} σ⌊PQ Unyσ =
          let m , n = heightNe P⇓ , height Q⇓
              σ⌊P : σ : Γ ⇀ Δ ⌊ P
              σ⌊P = λ x*P → σ⌊PQ (*·l x*P)
              Pσ⇓ : WN (P • σ)
              Pσ⇓ = proj₁ (WN-lemmaNe P⇓ (hi (B , m) (right (m<'m⊔n+1 m n)))
               ↪  P:ɣ→ε N⇓) σ⌊P Unyσ
```

```
304               σ⌊Q : σ : Γ ⇀ Δ ⌊ Q
305               σ⌊Q = λ x*Q → σ⌊PQ (*·r x*Q)
306               Qσ⇓ : WN (Q • σ)
307               Qσ⇓ = proj₁ (WN-lemma Q⇓ (hi (B , n) (right (m<'n⊔m+1 n m))) Q:𝔶
                  ↪  N⇓) σ⌊Q Unyσ
308               Pσ:𝔶→ε : Δ ⊢ P • σ : 𝔶 ⟶ ε
309               Pσ:𝔶→ε = lemma⊢σM P:𝔶→ε σ⌊P
310               Qσ:𝔶 : Δ ⊢ Q • σ : 𝔶
311               Qσ:𝔶 = lemma⊢σM Q:𝔶 σ⌊Q
312               PQσ⇓₁ = λ IsVarσx →
313                 let R , Pσ→R , nfR = sound-WN Pσ⇓
314                     y , σx≡y = IsVar⇒∃y σ x IsVarσx
315                     wneP = WNe⇒wne P⇓
316                     wnePσ = lemma₃ wneP σx≡y
317                     wneR = lemma₁ wnePσ Pσ→R
318                     neR = corollary₁ wneR nfR
319                     R⇓ = ne⇒WNe neR
320                  in exp (app-star-1 Pσ→R) (wke (app R⇓ Qσ⇓))
321               PQσ⇓₂ = λ { (_ , Γ⊢x:B) →
322                 let 𝔶<β : 𝔶 ₜ<⁺ B
323                     𝔶<β = lemma-ₜ< (WNe⇒wne P⇓) Γ⊢x:B P:𝔶→ε
324                  in proj₂ (WN-lemma Pσ⇓ (hi (𝔶 , height Pσ⇓) (left 𝔶<β)) Pσ:𝔶→ε
                   ↪  Qσ⇓) (ε , refl) Qσ:𝔶 }
325           in [ PQσ⇓₁ , PQσ⇓₂ ]' (Unyσ {x})
326
327 WN-lemma {β = B} (wke M⇓) (acc hi) = WN-lemmaNe M⇓ (hi (B , heightNe M⇓)
    ↪  (right ≤'-refl))
328 WN-lemma {ƛ x P} {Γ} {δ ⟶ ε} {B} {N} (abs P⇓) (acc hi) (⊢ƛ P:ε) N⇓ = thesis₁
    ↪  , thesis₂ (⊢ƛ P:ε)
329   where thesis₁ : ∀ {σ Δ} → σ : Γ ⇀ Δ ⌊ ƛ x P → Unary σ N Γ B → WN (ƛ x P • σ)
330         thesis₁ {σ} {Δ} σ⌊ƛxP Unyσ =
331           let z : V
332               z = χ (σ , ƛ x P)
333               σ⌊P : (σ ≺+ (x , v z)) : (Γ , x : δ) ⇀ (Δ , z : δ) ⌊ P
334               σ⌊P = lemmaaux⇀ (χ-lemma2 σ (ƛ x P)) σ⌊ƛxP
335               Unyσ,x=z : Unary (σ ≺+ (x , v z)) N (Γ , x : δ) B
336               Unyσ,x=z = lemma-Unary≺+ Unyσ
```

```
337               Pσ,x=z⇓ = proj₁ (WN-lemma P⇓ (hi (B , height P⇓) (right
                  ↪ ≤'-refl)) P:ε N⇓) σ⌊P Unyσ,x=z
338             in abs Pσ,x=z⇓
339           thesis₂ : ∀ {δ ε} → Γ ⊢ ƛ x P : δ ⟶ ε → (∃ λ ɣ → δ ⟶ ε ≡ B ⟶ ɣ)
              ↪ → Γ ⊢ N : B → WN (ƛ x P · N)
340           thesis₂ {.B} {.ɣ} (⊢ƛ P:ɣ) (ɣ , refl) N:B =
341             let x=N⌊P : (ι ≺+ (x , N)) : (Γ , x : B) ⇁ Γ ⌊ P
342                 x=N⌊P = lemma⇁ (lemmaι≺+⇁ N:B)
343                 Unyx=N : Unary (ι ≺+ (x , N)) N (Γ , x : B) B
344                 Unyx=N = lemma-Unaryι N:B
345                 Px=N⇓ = proj₁ (WN-lemma P⇓ (hi (B , height P⇓) (right ≤'-refl))
                    ↪ P:ɣ N⇓) x=N⌊P Unyx=N
346             in exp (just (inj₁ (ctxinj ▷β))) Px=N⇓
347   WN-lemma {M} {Γ} {α} {B} {P} (exp {.M} {N} M↠N N⇓) (acc hi) M:α P⇓ = thesis₁
      ↪ , thesis₂
348     where n = height N⇓
349           N:α : Γ ⊢ N : α
350           N:α = lemma⊢→α* M:α M↠N
351           thesis₁ : ∀ {σ Δ} → σ : Γ ⇁ Δ ⌊ M → Unary σ P Γ B → WN (M • σ)
352           thesis₁ {σ} {Δ} σ⌊M Unyσ =
353             let σ⌊N : σ : Γ ⇁ Δ ⌊ N
354                 σ⌊N = λ x*N → σ⌊M ((dual-#-* lemma→α*#) x*N M↠N)
355                 Nσ⇓ : WN (N • σ)
356                 Nσ⇓ = proj₁ (WN-lemma N⇓ (hi (B , n) (right ≤'-refl)) N:α P⇓)
                    ↪ σ⌊N Unyσ
357             in exp (subst-compat↠ M↠N) Nσ⇓
358           thesis₂ : (∃ λ ɣ → α ≡ B ⟶ ɣ) → Γ ⊢ P : B → WN (M · P)
359           thesis₂ α=β→ɣ P:B =
360             let NP⇓ = proj₂ (WN-lemma  N⇓ (hi (B , n) (right ≤'-refl)) N:α P⇓)
                ↪ α=β→ɣ P:B
361             in exp (app-star-l M↠N) NP⇓
362
363   WN-theo : ∀ {Γ M α} → Γ ⊢ M : α → WN M
364   WN-theo (⊢v _) = wke var
365   WN-theo (⊢· {α} {B} {M} M:α→β N:α) =
366     let M⇓ = WN-theo M:α→β
```

```
367     in proj₂ (WN-lemma M⇓ (wf_τ,ℕ< (α , height M⇓)) M:α→β (WN-theo N:α)) (B ,
        ↪  refl) N:α
368   WN-theo (⊢ƛ M:α) = abs (WN-theo M:α)
369
370   -- Main theorem
371
372   wn-theo : ∀ {Γ M α} → Γ ⊢ M : α → wn M
373   wn-theo M:α = sound-WN (WN-theo M:α)
```

End of WeakNormalization.agda

```
374   module SoundnessSN where
375
376   open import Chi
377   open import Term renaming (_⟶_ to _⇒_) hiding (_:_)
378   open import Beta
379   open import Substitution hiding (_∘_)
380   open import Alpha
381   open import Relation Λ
382   open import ParallelReduction
383   open import SubstitutionLemmas
384   open import SubstitutionCompatibilityLemmas hiding (_↠_)
385   open import Neutral
386
387   open import Relation.Nullary
388   open import Data.Empty
389   open import Data.Sum
390   open import Relation.Binary.Core hiding (Rel)
391   open import Data.Product hiding (Σ)
392   open import Data.Nat hiding (_≟_)
393   open import Function
394   open import Relation.Binary.PropositionalEquality using (sym; subst₂)
395
396   infix 5 _→sn_
397   infix 4 _→SN_
398
399   -- Accessibility definition of strongly normalizing terms
```

```
400
401 data sn : Λ → Set where
402   def : ∀ {M} → (∀ {N} → M →β N → sn N) → sn M
403
404 data _→sn_ : Λ → Λ → Set where
405   β    : ∀ {x M N} → sn N → ƛ x M · N →sn M [ x := N ]
406   appl : ∀ {M M' N} → M →sn M' → M · N →sn M' · N
407   αsn  : ∀ {M N P} → M →sn N → N ∼α P → M →sn P
408 --  αsn  : ∀ {M N P} → M ∼α N → M →sn N
409
410 -- Convenient definition of strongly normalizing terms
411
412 data SN : Λ → Set
413 data SNe : V → Λ → Set
414 data _→SN_ : Λ → Λ → Set
415
416 data _→SN_ where
417   β    : ∀ {M N x} → SN N → ƛ x M · N →SN M [ x := N ]
418   appl : ∀ {M M' N} → M →SN M' → M · N →SN M' · N
419   αsn  : ∀ {M N P} → M →SN N → N ∼α P → M →SN P
420 --  αsn  : ∀ {M N P} → M ∼α N → M →SN N
421
422 data SNe where
423   v   : ∀ {x} → SNe x (v x)
424   app : ∀ {x M N} → SNe x M → SN N → SNe x (M · N)
425
426 data SN where
427   sne   : ∀ {M x} → SNe x M → SN M
428   abs   : ∀ {M x} → SN M → SN (ƛ x M)
429   exp : ∀ {M N} → M →SN N → SN N → SN M
430
431 -- Auxiliary lemmas
432
433 sn-α : ∀ {M N} → M ∼α N → sn M → sn N
434 sn-α {_}{N} M~N (def hi) = def λ N→P → sn-α-aux N→P
435   where sn-α-aux : ∀ {P} → N →β P → sn P
436         sn-α-aux N→P with conflα→β (∼σ M~N) N→P
```

```
437         ... | _ , M→Q , P~Q = sn-α (∼σ P~Q) (hi M→Q)
438
439 var-irred : ∀ {x M} → (v x) →β M → α
440 var-irred (ctxinj ())
441
442 ⟶²⇒⟶* : ∀ {M N P} → M →β N × N ∼α P → M ↠ P
443 ⟶²⇒⟶* (M→P , P~N) = trans (just (inj₁ M→P)) (just (inj₂ P~N))
444
445 -- Lemma 5 (subst-compat₁): See SubstitutionCompatibilityLemmas module
446
447 -- Lemma 6 (subst-compat₂): See SubstitutionCompatibilityLemmas module
448
449 -- Lemma 8
450
451 multistep : ∀ {M M'} → M ↠ M' → sn M → sn M'
452 multistep refl snM = snM
453 multistep (just (inj₁ M→M')) (def snM) = snM M→M'
454 multistep (just (inj₂ M~M')) snM = sn-α M~M' snM
455 multistep (trans M→*N N→*P) snM = multistep N→*P (multistep M→*N snM)
456
457 -- Lemma 9
458
459 lemma-sn-v : ∀ {x} → sn (v x)
460 lemma-sn-v = def λ x→M → α-elim (var-irred x→M)
461
462 lemma-sn-ƛ : ∀ {x M} → sn M → sn (ƛ x M)
463 lemma-sn-ƛ snM = def λ ƛxM→P → lemma-sn-ƛ-aux snM ƛxM→P
464   where lemma-sn-ƛ-aux : ∀ {x M P} → sn M → ƛ x M →β P → sn P
465         lemma-sn-ƛ-aux _ (ctxinj ())
466         lemma-sn-ƛ-aux (def M→Q⇒snQ) (ctxƛ M→M') = lemma-sn-ƛ (M→Q⇒snQ M→M')
467
468 inv-app-lemma : ∀ {M N} → sn (M · N) → sn M × sn N
469 inv-app-lemma snMN = (def λ M→P → lemma-sn-app-aux₁ snMN M→P) , (def λ N→P →
    ↪ lemma-sn-app-aux₂ snMN N→P)
470   where lemma-sn-app-aux₁ : ∀ {M N P} → sn (M · N) → M →β P → sn P
471         lemma-sn-app-aux₁ (def MN→P⇒snP) M→P = proj₁ (inv-app-lemma (MN→P⇒snP
          ↪ (ctx·l M→P)))
```

```
472         lemma-sn-app-aux₂ : ∀ {M N P} → sn (M · N) → N →β P → sn P
473         lemma-sn-app-aux₂ (def MN→P⇒snP) N→P = proj₂ (inv-app-lemma (MN→P⇒snP
            ↪ (ctx·r N→P)))
474
475 -- Lemma 10 (Weak head expansion)
476
477 wkh-exp-α : ∀ {M N x Q} → sn N → sn Q → Q ∼α M [ x := N ] → sn (ƛ x M · N)
478
479 wkh-exp-α-aux : ∀ {M x N P Q} → sn N → sn Q → Q ∼α M [ x := N ] → ƛ x M · N →β
    ↪ P → sn P
480 wkh-exp-α-aux snN snQ Q˜M[N/x] (ctxinj ▷β) = sn-α Q˜M[N/x] snQ
481 wkh-exp-α-aux _ _ _ (ctx·l (ctxinj ()))
482 wkh-exp-α-aux {_}{x} snN (def hi) Q˜M[N/x] (ctx·l (ctxƛ M→M')) =
483   let _ , M[N/x]→R , R˜M'[N/x] = subst-compat→β M→M'
484       _ , Q→S , R˜S  = conflα→β (∼σ Q˜M[N/x]) M[N/x]→R
485   in wkh-exp-α snN (hi Q→S) (∼τ (∼σ R˜S) R˜M'[N/x])
486 wkh-exp-α-aux {M}{x} (def hi) snQ Q˜M[N/x] (ctx·r N→N') = wkh-exp-α (hi N→N')
    ↪ (multistep (subst-compat x M N→N') (sn-α Q˜M[N/x] snQ)) ∼ρ
487
488 wkh-exp-α snN snQ Q˜M[N/x] = def λ ƛxMN→Q → wkh-exp-α-aux snN snQ Q˜M[N/x]
    ↪ ƛxMN→Q
489
490 wkh-exp : ∀ {M N x} → sn N → sn (M [ x := N ]) → sn (ƛ x M · N)
491 wkh-exp snN snM[N/x] = wkh-exp-α snN snM[N/x] ∼ρ
492
493 -- Lemma 11
494
495 closure→Ne : ∀ {R R' x} → wne x R → R →β R' → wne x R'
496 closure→Ne var (ctxinj ())
497 closure→Ne (app ()) (ctxinj ▷β)
498 closure→Ne (app R∈ne) (ctx·l R→P) = app (closure→Ne R∈ne R→P)
499 closure→Ne (app R∈ne) (ctx·r {_}{_}{P} N→P) = app R∈ne
500
501 closure·Ne : ∀ {R N x} → wne x R → sn R → sn N → sn (R · N)
502
503 closure·Ne-aux : ∀ {R N Q x} → wne x R → sn R → sn N → R · N →β Q → sn Q
504 closure·Ne-aux () snR snN (ctxinj ▷β)
```

```
505 closure·Ne-aux neR (def R→P⇒snP) snN (ctx·l R→R') = closure·Ne (closure→Ne
    ↪  neR R→R') (R→P⇒snP R→R') snN
506 closure·Ne-aux neR snR (def N→P⇒snP) (ctx·r N→N') = closure·Ne neR snR
    ↪  (N→P⇒snP N→N')
507
508 -- closure·Ne : ∀ {R N} → ne R → sn R → sn N → sn (R · N)
509 closure·Ne R∈ne R∈sn N∈sn = def λ RN→Q → closure·Ne-aux R∈ne R∈sn N∈sn RN→Q
510
511 -- Lemma 12 (Confluence)
512
513 abs-snred-ƛ : ∀ {x M P} → ƛ x M →sn P → α
514 abs-snred-ƛ (αsn ƛxM→P _) = abs-snred-ƛ ƛxM→P
515
516 confluence : ∀ {M N N'} → M →sn N → M →β N' → N ∼α N' ⊎ ∃ (λ Q → N' →sn Q × N
    ↪  →» Q)
517 confluence (β _) (ctxinj ▷β) = inj₁ ∼ρ
518 confluence {ƛ x M · N} (β N∈sn) (ctx·l (ctxƛ {._}{._}{M'} M→M')) = inj₂ (M' [
    ↪  x := N ] , β N∈sn , ⟶²⇒⟶* (proj₂ (subst-compat→β M→M')))
519 confluence (β _) (ctx·l (ctxinj ()))
520 confluence {ƛ x M · N} (β (def N→N'⇒N∈sn)) (ctx·r .{_}{._}{N'} N→N') = inj₂
    ↪  (M [ x := N' ] , β (N→N'⇒N∈sn N→N') , subst-compat x M N→N')
521 confluence (appl (αsn λxM→P _)) (ctxinj ▷β) = α-elim (abs-snred-ƛ λxM→P)
522 confluence (appl (appl _)) (ctxinj ())
523 confluence (appl (β _)) (ctxinj ())
524 confluence {M · N} (appl M→snM') (ctx·l M→M₂) with confluence M→snM' M→M₂
525 ... | inj₁ M'∼M₂ = inj₁ (∼· M'∼M₂ ∼ρ)
526 ... | inj₂ (P , M₂→snP , M'→*P) = inj₂ (P · N , appl M₂→snP , app-star-l M'→*P)
527 confluence {M · N}{M' · .N}{.M · N'} (appl M→snM') (ctx·r N→N') = inj₂ (M' · N'
    ↪  , appl M→snM' , just (app-step-r (inj₁ N→N')))
528 confluence (αsn M→N N∼P) M→Q with confluence M→N M→Q
529 ... | inj₁ N∼Q = inj₁ (∼τ (∼σ N∼P) N∼Q)
530 ... | inj₂ (S , Q→S , N→*S) =
531   let T , P→*T , S∼T = conflα→» N∼P N→*S
532   in inj₂ (T , (αsn Q→S S∼T) , P→*T)
533
534 -- Lemma 13
535
```

```
backward→sn-aux : ∀ {M N M'} → sn M → sn N → M →sn M' → sn (M' · N) → sn (M · N)

backward→sn-aux' : ∀ {M N M' Q} → M · N →β Q → sn M → sn N → M →sn M' → sn (M'
  ↪  · N) → sn Q
backward→sn-aux' (ctxinj ▷β) _ _ (αsn λxM→P _) _ = α-elim (abs-snred-ƛ λxM→P)
backward→sn-aux' {.M} {.N} (ctx·l {M} {M''} {N} M→M'') (def M→Q⇒Q∈sn) N∈sn
  ↪  M→snM' M'N∈sn with confluence M→snM' M→M''
... | inj₁ M'∼M'' = sn-α (∼· M'∼M'' ∼ρ) M'N∈sn
... | inj₂ (P , M''→snP , M'→*P) = backward→sn-aux (M→Q⇒Q∈sn M→M'') N∈sn
  ↪  M''→snP (multistep (app-star-l M'→*P) M'N∈sn )
backward→sn-aux' {_}{N}{M'} (ctx·r {_}{_}{N'} N→N') M∈sn (def N→Q⇒Q∈sn)
  ↪  M→snM' (def M'N→Q⇒Q∈sn) =
   backward→sn-aux M∈sn (N→Q⇒Q∈sn N→N') M→snM' (M'N→Q⇒Q∈sn (ctx·r N→N'))

-- backward→sn-aux : ∀ {M N M'} → sn M → sn N → M →sn M' → sn (M' · N) → sn (M
  ↪  · N)
backward→sn-aux M∈sn N∈Sn M→snM' M'N∈sn = def ƛ MN→Q → backward→sn-aux' MN→Q
  ↪  M∈sn N∈Sn M→snM' M'N∈sn

backward→sn : ∀ {M M'} → M →sn M' → sn M' → sn M
backward→sn (αsn M→N N∼P) P∈sn = backward→sn M→N (sn-α (∼σ N∼P) P∈sn)
backward→sn (β N∈sn) M[x=N]∈sn = wkh-exp N∈sn M[x=N]∈sn
backward→sn {M · N} {M' · .N} (appl M→M') M'N∈sn = let snM' , snN =
  ↪  inv-app-lemma M'N∈sn
                                                    in backward→sn-aux
                                                    ↪  (backward→sn M→M' snM')
                                                    ↪  snN M→M' M'N∈sn
-- Lemma 14

lemma-ne : ∀ {M x} → SNe x M → wne x M
lemma-ne v = var
lemma-ne (app M∈ne _) = app (lemma-ne M∈ne)

-- Theorem 1

sound-SN  : ∀ {M} → SN M → sn M

```

```
564 sound-SNe : ∀ {M x} → SNe x M → sn M
565
566 sound→SN  : ∀ {M N} → M →SN N → M →sn N
567
568 -- sound-SN : ∀ {M} → SN M → sn M
569 sound-SN (sne x) = sound-SNe x
570 sound-SN (abs x) = lemma-sn-ƛ (sound-SN x)
571 sound-SN (exp M→N N∈Sn) = backward→sn (sound→SN M→N) (sound-SN N∈Sn)
572
573 -- sound-SNe : ∀ {M} → SNe M → sn M
574 sound-SNe v = lemma-sn-v
575 sound-SNe (app M∈SNe N∈Sn) = closure·Ne (lemma-ne M∈SNe) (sound-SNe M∈SNe)
    ↪  (sound-SN N∈Sn)
576
577 -- sound→SN : ∀ {M N} → M →SN N → M →sn N
578 sound→SN (αsn M→N N∼P) = αsn (sound→SN M→N) N∼P
579 sound→SN (β M∈Sn) = β (sound-SN M∈Sn)
580 sound→SN (appl M→M') = appl (sound→SN M→M')
```

End of SoundnessSN.agda

```
581 module StrongNormalizationA where
582
583 open import SoundnessSN
584 open import Term
585 open import Chi
586 open import Substitution
587 open import Data.Sum
588 open import Data.Product hiding (Σ)
589 open import Beta
590 open import Alpha
591 open import SubstitutionLemmas
592 open import ListProperties
593 open import Relation using (just; trans)
594 open import Unary
595 open import TypeLemmas
596 open import SubstitutionCompatibilityLemmas
597
```

```
open import Data.Nat hiding (_*_)
open import Relation.Binary.PropositionalEquality renaming (trans to trans≡)
open import Induction.WellFounded
open import Data.Unit hiding (_≟_)
open import Data.Empty
open import Relation.Nullary
open import Induction.Nat
open import Data.Nat.Properties
open import Relation.Binary hiding (_⇒_)
open import Algebra.Structures

-- Well-foundness proofs

open Lexicographic _τ<⁺_ (λ _ m n → m <′ n) renaming (_<_ to _τ,ℕ<_ ;
  ↪  well-founded to wfΣ)

wfτ,ℕ< : Well-founded _τ,ℕ<_
wfτ,ℕ< = wfΣ wfτ<⁺ <-well-founded

-- Definitions

height→  : ∀ {M N} → M →SN N → ℕ
heightNe : ∀ {M x} → SNe x M → ℕ
height   : ∀ {M} → SN M → ℕ

height→ (β N⇓) = suc (height N⇓)
height→ (appl M→N) = suc (height→ M→N)
height→ (αsn M→N N∼P) = suc (height→ M→N)

heightNe v = 0
heightNe (app M⇓ N⇓) = suc (heightNe M⇓ ⊔ height N⇓)

height (abs M⇓) = suc (height M⇓)
height (sne M⇓) = suc (heightNe M⇓)
height (exp M→N N⇓) = suc (height→ M→N ⊔ height N⇓)

-- Auxiliary lemmas
```

```
m<'m⊔n+1 : ∀ m n → m <' suc (m ⊔ n)
m<'m⊔n+1 m n = s≤'s (≤⇒≤' (m≤m⊔n m n))

⊔-comm = IsCommutativeMonoid.comm
  ↪ (IsCommutativeSemiringWithoutOne.+-isCommutativeMonoid
  ↪ ⊔-⊓-0-isCommutativeSemiringWithoutOne)

m<'n⊔m+1 : ∀ m n → m <' suc (n ⊔ m)
m<'n⊔m+1 m n with n ⊔ m | ⊔-comm n m
m<'n⊔m+1 m n | .(m ⊔ n) | refl = m<'m⊔n+1 m n

→SN⊂→α : ∀ {M N} → M →SN N → M →α* N
→SN⊂→α (β _)  = just (inj₁ (ctxinj ▷β))
→SN⊂→α (appl M→M') = app-star-l (→SN⊂→α M→M')
→SN⊂→α (αsn M→N N∼P) = trans (→SN⊂→α M→N) (just (inj₂ N∼P))

lemmaσ⌊· : ∀ {σ Γ Δ P Q} → σ : Γ ⇁ Δ ⌊ P · Q → (σ : Γ ⇁ Δ ⌊ P) × (σ : Γ ⇁
  ↪ Δ ⌊ Q)
lemmaσ⌊· σ⌊PQ = (λ x*P → σ⌊PQ (*·l x*P)) , (λ x*Q → σ⌊PQ (*·r x*Q))

≡⇒α : ∀ {M N} → M ≡ N → M ∼α N
≡⇒α {M} M≡N = subst₂ _∼α_ refl M≡N (∼ρ {M})

invol≺+ : ∀ {σ x M P} → x # P → (σ ≺+ (x , M)) ≅ σ ⌊ P
invol≺+ {σ} {x} {M} {P} x#P = ∼*ρ , aux
  where aux : (y : V) → y * P → (σ ≺+ (x , M)) y ≡ σ y
        aux y y*P with x ≟ y
        ... | no _ = refl
        aux .x x*P | yes refl = α-elim (lemma-free→¬# x*P x#P)

weaken-dom : ∀ {x M σ Γ Δ} → σ : Γ ⇁ Δ ⌊ M → σ : Γ ⇁ Δ ⌊ ƛ x M
weaken-dom {x} {M} {σ} {Γ} {Δ} σ⌊M = λ y*M → aux y*M
  where aux : ∀ {x y} → y * ƛ x M → (p : y ∈ Γ) → Δ ⊢ σ y : Γ  p
        aux {x} {y} (*ƛ x*M _) y∈Γ with y ≟ x
        ... | no _ = σ⌊M x*M y∈Γ
```

```
667         aux {x} {.x} (*↛ _ x≢x) _ | yes refl = α-elim (x≢x refl)
668
669 -- Main lemma
670
671 SN-lemma : ∀ {M Γ α β N}
672          → (M⇓ : SN M)
673          → Acc _<_ (β , height M⇓)
674          → Γ ⊢ M : α
675          → SN N
676          → (∀ {x σ Δ} → σ : Γ ⇀ Δ ⌊ M → Unary σ x N → Γ ⊢ ν x : β → SN (M •
             ↪ σ))
677            × ((∃ λ ɣ → α ≡ β ⟶ ɣ) → Γ ⊢ N : β → SN (M · N))
678
679 SN-lemma→ : ∀ {M N σ Γ Δ α β P x}
680           → (M→N : M →SN N)
681           → Acc _<_ (β , height→ M→N)
682           → Γ ⊢ M : α
683           → SN P
684           → σ : Γ ⇀ Δ ⌊ M
685           → Unary σ x P
686           → Γ ⊢ ν x : β
687           → (M • σ) →SN (N • σ)
688
689 SN-lemmaNe : ∀ {M Γ α β y N}
690            → (M⇓ : SNe y M)
691            → Acc _<_ (β , heightNe M⇓)
692            → Γ ⊢ M : α
693            → SN N
694            → (∀ {x σ Δ} → σ : Γ ⇀ Δ ⌊ M → Unary σ x N → Γ ⊢ ν x : β → SN (M
               ↪ • σ) × (y ≢ x → ∃ λ z → SNe z (M • σ)))
695              × ((∃ λ ɣ → α ≡ β ⟶ ɣ) → Γ ⊢ N : β → SN (M · N))
696
697 SN-lemma→ {_} {_} {σ} {Γ} {Δ} {α} {B} (αsn M→N N∼P) (acc hi) M:A P⇓ σ⌊M Unyσ
    ↪ x:B =
698   let Mσ→Nσ = SN-lemma→ M→N (hi (B , height→ M→N) (right ≤'-refl)) M:A P⇓ σ⌊M
      ↪ Unyσ x:B
699       Nσ∼Pσ = ≡⇒α (lemmaM∼M'→Mσ≡M'σ N∼P)
```

```
700   in αsn Mσ→Nσ Nσ∼Pσ
701 SN-lemma→ {ƛ y M · N} {_} {σ} {Γ} {Δ} {α} {B} {_} {x} (β N⇓) (acc hi) (⊢· _
    ↪  N:𝔶) P⇓ σ⌊ƛyMN Unyσ x:B =
702   let z : V
703       z = χ (σ , ƛ y M)
704       σ⌊N : σ : Γ ⇁ Δ ⌊ N
705       σ⌊N = proj₂ (lemmaσ⌊· σ⌊ƛyMN)
706       Nσ⇓ : SN (N • σ)
707       Nσ⇓ = proj₁ (SN-lemma N⇓ (hi (B , height N⇓) (right ≤'-refl)) N:𝔶 P⇓)
          ↪  σ⌊N Unyσ x:B
708       Mσ,x=z,z=Nσ˜Mx=Nσ : (M • σ ≺+ (y , v z)) • ι ≺+ (z , N • σ) ∼α (M • ι
          ↪  ≺+ (y , N)) • σ
709       Mσ,x=z,z=Nσ˜Mx=Nσ = lemma∼α• (χ-lemma2 σ (ƛ y M))
710   in αsn (β Nσ⇓) Mσ,x=z,z=Nσ˜Mx=Nσ
711 SN-lemma→ {L · J} {L' · .J} {σ} {Γ} {Δ} {_} {B} (appl L→L') (acc hi) (⊢· L:𝔶
    ↪  _) P⇓ σ⌊LJ Unyσ x:B =
712   let σ⌊L : σ : Γ ⇁ Δ ⌊ L
713       σ⌊L = proj₁ (lemmaσ⌊· σ⌊LJ)
714       Lσ→L'σ = SN-lemma→ L→L' (hi (B , height→ L→L') (right ≤'-refl)) L:𝔶 P⇓
          ↪  σ⌊L Unyσ x:B
715   in appl Lσ→L'σ
716
717 SN-lemmaNe .{v y} {Γ} {_} {B} {.y} {N} (v {y}) _ _ N⇓  = thesis₁ , λ _ _ → sne
    ↪  (app v N⇓)
718   where thesis₁ : ∀ {x σ Δ} → σ : Γ ⇁ Δ ⌊ (v y) → Unary σ x N → Γ ⊢ v x : B
      ↪  → SN (v y • σ) × (y ≢ x → ∃ λ z → SNe z (v y • σ))
719         thesis₁ {x} {σ} _ Unyσ _ with y ≟ x
720         ... | no y≢x with σ y | (proj₂ Unyσ) y≢x
721         ... | .(v z) | isv {z} = sne v , λ _ → z , v
722         thesis₁ {.y} {σ} _ Unyσ _ | yes refl with σ y | proj₁ Unyσ
723         ... | .N | refl = N⇓ , λ y≢x → α-elim (y≢x refl)
724 SN-lemmaNe {P · Q} {Γ} {_} {B} {.y} {N} (app {y} P⇓ Q⇓) (acc hi) (⊢· {𝔶} {ε}
    ↪  P:𝔶→ε Q:𝔶) N⇓ =
725   thesis₁ , λ _ _ → sne (app (app P⇓ Q⇓) N⇓)
726     where
727         thesis₁ : ∀ {x σ Δ} → σ : Γ ⇁ Δ ⌊ P · Q → Unary σ x N → Γ ⊢ v x : B
          ↪  → SN (P · Q • σ) × (y ≢ x → ∃ λ z → SNe z (P · Q • σ))
```

```
728           thesis₁ {x} {σ} {Δ} σ⌊PQ Unyσ x:B with y ≟ x
729           ... | no y≢x =
730             let m , n = heightNe P⇓ , height Q⇓
731                 σ⌊P : σ : Γ ⇁ Δ ⌊ P
732                 σ⌊P = λ y*P → σ⌊PQ (*·l y*P)
733                 Pσ⇓ : ∃ λ z → SNe z (P • σ)
734                 Pσ⇓ = proj₂ (proj₁ (SN-lemmaNe P⇓ (hi (B , m) (right (m<'m⊔n+1 m
                     ↪ n))) P:𝔶→ε N⇓) σ⌊P Unyσ x:B) y≢x
735                 σ⌊Q : σ : Γ ⇁ Δ ⌊ Q
736                 σ⌊Q = λ y*Q → σ⌊PQ (*·r y*Q)
737                 Qσ⇓ : SN (Q • σ)
738                 Qσ⇓ = proj₁ (SN-lemma Q⇓ (hi (B , n) (right (m<'n⊔m+1 n m))) Q:𝔶
                     ↪ N⇓) σ⌊Q Unyσ x:B
739                 PQσ⇓y = app (proj₂ Pσ⇓) Qσ⇓
740             in sne PQσ⇓y , λ _ → (proj₁ Pσ⇓ , PQσ⇓y)
741           thesis₁ {.y} {σ} {Δ} σ⌊PQ Unyσ y:B | yes refl =
742             let m , n = heightNe P⇓ , height Q⇓
743                 σ⌊P : σ : Γ ⇁ Δ ⌊ P
744                 σ⌊P = λ y*P → σ⌊PQ (*·l y*P)
745                 Pσ⇓ : SN (P • σ)
746                 Pσ⇓ = proj₁ (proj₁ (SN-lemmaNe P⇓ (hi (B , m) (right (m<'m⊔n+1 m
                     ↪ n))) P:𝔶→ε N⇓) σ⌊P Unyσ y:B)
747                 σ⌊Q : σ : Γ ⇁ Δ ⌊ Q
748                 σ⌊Q = λ y*Q → σ⌊PQ (*·r y*Q)
749                 Qσ⇓ : SN (Q • σ)
750                 Qσ⇓ = proj₁ (SN-lemma Q⇓ (hi (B , n) (right (m<'n⊔m+1 n m))) Q:𝔶
                     ↪ N⇓) σ⌊Q Unyσ y:B
751                 Pσ:𝔶→ε : Δ ⊢ P • σ : 𝔶 ⟶ ε
752                 Pσ:𝔶→ε = lemma⊢σM P:𝔶→ε σ⌊P
753                 Qσ:𝔶 : Δ ⊢ Q • σ : 𝔶
754                 Qσ:𝔶 = lemma⊢σM Q:𝔶 σ⌊Q
755                 𝔶<β : 𝔶 τ<⁺ B
756                 𝔶<β = lemma-τ< (lemma-ne P⇓) y:B P:𝔶→ε
757             in proj₂ (SN-lemma Pσ⇓ (hi (𝔶 , height Pσ⇓) (left 𝔶<β)) Pσ:𝔶→ε Qσ⇓)
                 ↪ (ε , refl) Qσ:𝔶 , λ y≢x → α-elim (y≢x refl)
758
759 SN-lemma {β = B} (sne M⇓) (acc hi) M:α N⇓ =
```

```
760   let th₁ , th₂ = SN-lemmaNe M⇓ (hi (B , heightNe M⇓) (right ≤'-refl)) M:α N⇓
761   in (λ p1 p2 p3 → proj₁ (th₁ p1 p2 p3)) , th₂
762 SN-lemma {ƛ y P} {Γ} {δ ⟶ ε} {B} {N} (abs P⇓) (acc hi) (⊢ƛ P:ε) N⇓ = thesis₁
    ↪  , thesis₂ P:ε
763   where thesis₁ : ∀ {x σ Δ} → σ : Γ ⇀ Δ ⌊ ƛ y P → Unary σ x N → Γ ⊢ v x : B
    ↪  → SN (ƛ y P • σ)
764         thesis₁ {x} {σ} {Δ} σ⌊ƛyP Unyσ x:B with y ≟ x
765         ... | no y≢x =
766           let z : V
767               z = χ (σ , ƛ y P)
768               σ,y=z⌊P : (σ ≺+ (y , v z)) : (Γ , y : δ) ⇀ (Δ , z : δ) ⌊ P
769               σ,y=z⌊P = lemmaaux⇀ (χ-lemma2 σ (ƛ y P)) σ⌊ƛyP
770               Unyσ,y=z : Unary (σ ≺+ (y , v z)) x N
771               Unyσ,y=z = unary≺+≢ y≢x Unyσ
772               x:B' : Γ , y : δ ⊢ v x : B
773               x:B' = lemmaWeakening⊢# (#v (sym≢ y≢x)) x:B
774               Pσ,y=z⇓ : SN (P • (σ ≺+ (y , v z)))
775               Pσ,y=z⇓ = proj₁ (SN-lemma P⇓ (hi (B , height P⇓) (right
                  ↪  ≤'-refl)) P:ε N⇓) σ,y=z⌊P Unyσ,y=z x:B'
776           in abs Pσ,y=z⇓
777         thesis₁ {.y} {σ} {Δ} σ⌊ƛyP Unyσ y:B | yes refl =
778           let z : V
779               z = χ (σ , ƛ y P)
780               u : V
781               u = χ (ι , P)
782               w : V
783               w = χ (σ ≺+ (y , v z) , ƛ u P)
784               σ,y=z⌊P : (σ ≺+ (y , v z)) : (Γ , y : δ) ⇀ (Δ , z : δ) ⌊ P
785               σ,y=z⌊P = lemmaaux⇀ (χ-lemma2 σ (ƛ y P)) σ⌊ƛyP
786              σ,y=z⌊ƛuP : (σ ≺+ (y , v z)) : (Γ , y : δ) ⇀ (Δ , z : δ) ⌊ ƛ u P
787               σ,y=z⌊ƛuP = weaken-dom σ,y=z⌊P
788              σ⌊P : (σ ≺+ (y , v z) ≺+ (u , v w)) : (Γ , y : δ , u : B) ⇀ (Δ
                  ↪  , z : δ , w : B) ⌊ P
789               σ⌊P = lemmaaux⇀ (χ-lemma2 (σ ≺+ (y , v z)) (ƛ u P)) σ,y=z⌊ƛuP
790               Unyσ,y=z,u=w : Unary (σ ≺+ (y , v z) ≺+ (u , v w)) u (v w)
791               Unyσ,y=z,u=w = unaryv (unary≺+≡ Unyσ)
792               u:B : Γ , y : δ , u : B ⊢ v u : B
```

```
793               u:B = ⊢v (here refl)
794               u#P : u # P
795               u#P = lemmaM∼N# (∼σ lemma•ι) u (lemmafree#→# (χ-lemma2 ι P))
796               P:ε' : Γ , y : δ , u : B ⊢ P : ε
797               P:ε' = lemmaWeakening⊢# u#P P:ε
798               Pσ,y=z,u=w⇓ : SN (P • (σ ≺+ (y , v z) ≺+ (u , v w)))
799               Pσ,y=z,u=w⇓ = proj₁ (SN-lemma P⇓ (hi (B , height P⇓) (right
                  ↪ ≤'-refl)) P:ε' (sne v)) σ⌊P Unyσ,y=z,u=w u:B
800               σ,y=z,u=w≡σ,y=z : (σ ≺+ (y , v z) ≺+ (u , v w)) ≅ σ ≺+ (y , v
                  ↪ z) ⌊ P
801               σ,y=z,u=w≡σ,y=z = invol≺+ u#P
802               Pσ,y=z⇓ : SN (P • (σ ≺+ (y , v z)))
803               Pσ,y=z⇓ = subst SN (lemma-subst-σ≡ σ,y=z,u=w≡σ,y=z) Pσ,y=z,u=w⇓
804             in abs Pσ,y=z⇓
805           thesis₂ : ∀ {δ ε} → Γ , y : δ ⊢ P : ε → (∃ λ ɣ → δ ⟶ ε ≡ B ⟶ ɣ) →
              ↪ Γ ⊢ N : B → SN (ƛ y P · N)
806           thesis₂ {.B} {.ɣ} P:ɣ (ɣ , refl) N:B =
807             let y=N⌊P : (ι ≺+ (y , N)) : (Γ , y : B) ⇀ Γ ⌊ P
808                 y=N⌊P = lemma⇀ (lemmaι≺+⇀ N:B)
809                 Unyy=N : Unary (ι ≺+ (y , N)) y N
810                 Unyy=N = unary
811                 Γ,y:B⊢y:B : Γ , y : B ⊢ v y : B
812                 Γ,y:B⊢y:B = ⊢v (here refl)
813                 Py=N⇓ = proj₁ (SN-lemma P⇓ (hi (B , height P⇓) (right ≤'-refl))
                    ↪ P:ɣ N⇓) y=N⌊P Unyy=N Γ,y:B⊢y:B
814             in exp (β N⇓) Py=N⇓
815   SN-lemma {M} {Γ} {α} {B} {P} (exp {.M} {N} M→N N⇓) (acc hi) M:α P⇓ = thesis₁ ,
      ↪ thesis₂
816     where m = height→ M→N
817           n = height N⇓
818           M→αN : M →α* N
819           M→αN = →SN⊂→α M→N
820           N:α : Γ ⊢ N : α
821           N:α = lemma⊢→α* M:α M→αN
822           thesis₁ : ∀ {x σ Δ} → σ : Γ ⇀ Δ ⌊ M → Unary σ x P → Γ ⊢ v x : B → SN
              ↪ (M • σ)
823           thesis₁ {x} {σ} {Δ} σ⌊M Unyσ x:B =
```

```
824             let Mσ→Nσ = SN-lemma→ M→N (hi (B , m) (right (m<'m⊔n+1 m n))) M:α P⇓
                ↪  σ⌊M Unyσ x:B
825                 σ⌊N : σ : Γ ⇁ Δ ⌊ N
826                 σ⌊N = λ y*N → σ⌊M ((dual-#-* lemma→α*#) y*N M→αN)
827                 Nσ⇓ : SN (N • σ)
828                 Nσ⇓ = proj₁ (SN-lemma N⇓ (hi (B , n) (right (m<'n⊔m+1 n m))) N:α
                    ↪  P⇓) σ⌊N Unyσ x:B
829             in exp Mσ→Nσ Nσ⇓
830           thesis₂ : (∃ λ 𝛾 → α ≡ B ⟶ 𝛾) → Γ ⊢ P : B → SN (M · P)
831           thesis₂ α=β→𝛾 P:B =
832             let NP⇓ = proj₂ (SN-lemma  N⇓ (hi (B , n) (right (m<'n⊔m+1 n m)))
                ↪  N:α P⇓) α=β→𝛾 P:B
833             in exp (appl M→N) NP⇓
834
835 SN-theo : ∀ {Γ M α} → Γ ⊢ M : α → SN M
836 SN-theo (⊢v _) = sne v
837 SN-theo (⊢· {α} {B} {M} M:α→β N:α) =
838   let M⇓ = SN-theo M:α→β
839   in proj₂ (SN-lemma M⇓ (wf_{τ,ℕ}< (α , height M⇓)) M:α→β (SN-theo N:α)) (B ,
      ↪  refl) N:α
840 SN-theo (⊢ƛ M:α) = abs (SN-theo M:α)
841
842 -- Main theorem
843 sn-theo : ∀ {Γ M α} → Γ ⊢ M : α → sn M
844 sn-theo M:α = sound-SN (SN-theo M:α)
```

End of StrongNormalization.agda