Universidad ORT Uruguay

Facultad de Ingeniería

A machine-checked proof of the Standardization Theorem in Lambda Calculus using multiple substitution

Entregado como requisito para la obtención del título de

Master en Ingeniería

Martín Copes - 172160

Tutores: Nora Szasz, Álvaro Tasistro

2018

Declaración de autoría

Yo, Martín Copes, 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, hem 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.

Martín Copes

14-02-2018

A mi madre por el apoyo y el ánimo que me brindó en toda mi carrera estudiantil.

A mi padre que aunque no esté físicamente conmigo, estuvo muy presente dentro de mí en todo momento al desarrollar este trabajo, en las investigaciones, en mis desvelos, en mis momentos más difíciles; siempre fue y será mi inspiración.

A mis amigos que de forma incondicional, entendieron mis ausencias y mis malos momentos.

A mi novia por ser un apoyo fundamental en esta etapa de mi vida, apoyándome a culminar mi carrera profesional y a alcanzar este logro.

A todos ustedes va dedicado este trabajo.

Agradecimientos

Agradezco a mis tutores, Dra. Nora Szasz y Dr. Álvaro Tasistro por haber confiado en mí, por la paciencia, por las largas horas de discusión, por sus conocimientos impartidos y por sus esfuerzos leyendo, opinando, corrigiendo y dando ánimo.

También me gustaría agradecer al Dr. Ernesto Copello, quien ha desarrollado las librerías en las que el presente trabajo está basado, por su constante buena disposición.

Resumen

El teorema de estandarización para el Cálculo Lambda es un resultado conocido que establece que si un término M β-reduce a un término N, entonces existe una secuencia de reducciones β estándar de M a N. Una secuencia de reducciones β es considerada estándar si todos los redexes de la misma se contraen de izquierda a derecha según lo indicado por la posición del λ en el redex $(\lambda x A)\, B$ de acuerdo a la sintaxis lineal de los términos.

En esta tesis presentamos una formalización completa de la demostración del teorema de estandarización para el Cálculo Lambda en la Teoría Constructiva de Tipos [1] utilizando sintaxis concreta de primer orden para los términos y sustitución múltiple definida por Stoughton en [2].

El presente desarrollo se basa en una formalización previa del Cálculo Lambda [3], donde los autores demostraron resultados metateóricos significativos que han sido verificados en máquina con el asistente de demostración Agda [4]. El principal objetivo de esta tesis es evaluar hasta qué punto la biblioteca anteriormente mencionada puede ser reutilizada para nuestro desarrollo.

La versión de la prueba del teorema de estandarización que formalizamos se basa en una presentada por Kashima en [5], donde la noción de β-reducibilidad estándar se define de forma inductiva. Es justamente debido a esta estructura inductiva, que la demostración es ideal para ser formalizada usando Teoría Constructiva de Tipos.

El uso de sustituciones múltiples sobre la sintaxis concreta para los términos lambda nos permite proceder utilizando sólo inducción estructural en los términos y las relaciones de reducción, produciendo pruebas que son fáciles de seguir y se asemejan a las escritas con lápiz y papel. Esto es novedoso en relación a las otras

formalizaciones de esta prueba presentes en la literatura que usan índices de Brujin para implementar la sintaxis. Debido a esto, las mismas requieren un esfuerzo adicional para manipular los términos codificados, y presentan dificultad a la hora de manejar la correspondencia entre versiones informales y formales del mismo resultado.

Por otra parte, como parte de esta tesis se formalizaron otros resultados relevantes relacionados con el teorema de estandarización, como el teorema de Reducción más a la izquierda.

Todas las definiciones y pruebas que aparecen en esta tesis han sido verificadas por máquina con el sistema Agda [4]. El código correspondiente está disponible en https://github.com/mcopes73/standardization-agda/.

Palabras clave: Cálculo Lambda, Metateoría Formal, Estandarización, Teoría Constructiva de Tipos

Abstract

The Standardization theorem for the Lambda Calculus is a well-known result that states that if a term M β-reduces to a term N, then there exists a standard β-reduction sequence from M to N. A β-reduction sequence is considered standard if successive contractions take place from left to right possibly with some jumps.

In this thesis we present a full formalization of the proof of the Standardization Theorem for the Lambda Calculus in Constructive Type Theory [1] using first-order concrete syntax and multiple substitution as defined by Stoughton in [2].

The development is based on a previous formalization of the Lambda Calculus [3], where the authors proved significant metatheoretical results which have been machine-checked with the system Agda [4]. One main objective of this thesis is to test the extent to which the produced library is able to support our extension.

The version of the proof of the Standardization theorem that we formalized is based on one presented by Kashima in [5], where a notion of β-reducibility with a standard sequence is captured by a an inductively defined reduction relation. Due to its inductive structure, the proof can be conveniently formalized in Constructive Type Theory.

The use of multiple substitutions over the concrete syntax for λ-terms enables us to proceed using only simple standard methods of structural induction on terms and reduction relations, producing proofs that are easy to follow, yet fully formal. This is novel in relation to previous efforts to formalize this proof which use de Brujin indexes to implement the syntax, requiring extra overhead to handle term encoding, thus making it difficult to keep track of the correspondence between informal and formal versions of the same result. It is worth noticing that these efforts use

induction on the size of terms while our proof proceeds by structural induction only.

In addition, some other relevant results related to the Standardization theorem such as the Leftmost Reduction theorem were proved.

All the definitions and proofs that appear in this thesis have been machine-checked with the system Agda [4]. The corresponding code is available at https://github.com/mcopes73/standardization-agda/.

Keywords: Lambda Calculus, Formal Metatheory, Standardization, Constructive Type Theory

Index

Chapter 1

Introduction

In [3] a formalization of the Lambda Calculus in Constructive Type Theory is presented, which uses an approach with one sort of names for both free and bound variables that does not identify α-convertible terms. In that development, a multiple substitution operation based on the one defined by Stoughton in [2] is used. Such an approach enables the authors to prove significant results about the metatheory of the Lambda Calculus, namely the Church-Rosser theorem and Subject Reduction for the simply typed Lambda Calculus à la Curry. They developed a library [6] with definitions and lemmas for implementing and manipulating substitutions that were key to establishing the mentioned results in a quite elegant way, in particular using only simple standard methods of structural induction on terms and reduction relations.

In the present work we extend the above mentioned metatheoretical study by proving the Standardization theorem for β-reduction, which we further use to prove that the leftmost-outermost reduction strategy always finds the normal form of a term provided that it exists. The Standardization theorem is a well-known result in the Lambda Calculus that was first proved by Curry and Feys in [7], and subsequently by several authors ([5; 8; 9; 10]). We give an overview of it in the next section.

One main objective of this thesis is to test the extent to which the above mentioned library is able to support this extended development as well as whether this can be carried out still using simple induction principles.

1.1 The Standardization Theorem

The Standardization theorem for the Lambda Calculus states that if a term M β-reduces to a term N, then there exists a standard β-reduction sequence from M to N. A reduction sequence is considered standard if successive redexes are contracted from left to right, possibly with some jumps. This means that, for example, a reduction sequence starting with a term of the form $M_1\ M_2\ M_3...$ must first perform all the desired reductions in M_1 before it can contract any redex in M_2 and must similarly perform all contractions in M_2 before reducing any redex in M_3, and so on. The Standardization theorem sheds a light on the relationships within redexes, showing that any set of reductions over the term $M_1\ M_2\ M_3...$ can be reordered to be made standard and are thus independent of each other. One of the most interesting corollaries of the Standardization theorem in the Lambda Calculus is that the leftmost-outermost reduction strategy always finds the normal form of a term provided there is one. This result is particularly useful for programming languages as it introduces an algorithm for semi-deciding term equality, i.e. the leftmost-outermost reduction strategy.

Arguably, the most well-known proof of the Standardization theorem in the literature is that of Barendregt [8]. Although we acknowledge the existence of other ones such as [7; 9; 10], in this section we shall focus on this classical proof and its main differences with respect to our development.

A key notion to Barendregt's proof is that of the *residuals* of a redex after a β-contraction. Intuitively, we can think of the residuals of a redex Δ after the contraction of redex Δ' as all of the resulting copies of redex Δ. After the contraction of redex Δ', we have a few possible scenarios: redex Δ may (1) remain unaffected if it is disjoint with Δ', (2) copied into zero or more equal redexes if it is a subterm of the operand term of Δ', (3) remain in the term with changes within its components if it contains Δ' as a subterm or if it is a subterm of the operator term of Δ', or (4) just disappear if it coincides with Δ'. In any case, we call the set of the *remaining versions of* Δ its residuals. Barendregt uses this concept to define, among other things, a standard reduction sequence as follows:

A reduction sequence $M_0 \xrightarrow{\Delta_1} M_1 \xrightarrow{\Delta_2} ... \xrightarrow{\Delta_n} M_n$ is considered standard if $(\forall i)(\forall j < i)$ Δ_i is not a residual of a redex to the left of Δ_j

We could imagine that, after having chosen a certain redex to contract, all of the redexes to its left are marked. The residuals of the marked redexes are not allowed to be reduced anymore and remain marked even after the contraction of other redexes. In fact, Klop [10] uses this idea of marking redexes in order to inductively define a standard reduction sequence. Barendregt uses the notation $M \overset{s}{\twoheadrightarrow} N$ to indicate the existence of a standard reduction sequence from M to N.

In the proof, Barendregt distinguishes *head* redexes from the so-called *internal* redexes. A redex $(\lambda x.A)\ M_1$ is said to be in the head position of a term if the latter has the following form:

$\lambda x_1 ... x_n.(\lambda x.A)\ M_1\ M_2\ ...\ M_m$ where $n \geq 0$ and $m \geq 1$.

A redex is said to be internal if it is not in the head position. Note that there can be at most one head redex in a given term and that all redexes can be classified as either head or internal redexes. The author uses the notation $M \overset{h}{\rightarrow} N$ to express the reduction of the redex at head position and $M \overset{i}{\rightarrow} N$ to express that of an internal redex.

Barendregt proves the main lemma (11.4.6) stating the following:

$M \overset{\beta}{\twoheadrightarrow} N \implies M \overset{h}{\twoheadrightarrow} M' \overset{i}{\twoheadrightarrow} N$ for some M'
where $\twoheadrightarrow$ stands for the reflexive-transitive closure of the corresponding $\rightarrow$ relation.

The proof of this lemma is rather involved as it is based on several other lemmas and non-trivial results such as the fact that the complete development[1] of a finite set of redexes always terminates (FD) and that for a given initial term, all complete developments of a fixed set of redexes end with the same term (FD!).

The Standardization theorem, i.e. $M \overset{\beta}{\twoheadrightarrow} N \implies M \overset{s}{\twoheadrightarrow} N$, can be then proven by induction on the *size* of N using the previous lemma. The above lemma tells us that there must exist a term Z such that $M \overset{h}{\twoheadrightarrow} Z \overset{i}{\twoheadrightarrow} N$. We can now identify two cases for N:

1. If N is a variable x, the proof is straightforward as $Z \equiv x$ and a head reduction sequence is a standard one.

[1] Given a term and a set of redexes, a complete development implies contracting all of the given redexes and their residuals in the term.

2. If N is of the form $\lambda x_1 ... x_n.N_0\ N_1\ ...\ N_m$, then Z must also be of the form $\lambda x_1 ... x_n.Z_0\ Z_1\ ...\ Z_m$ with $Z_i \overset{\beta}{\twoheadrightarrow} N_i$ for $0 \leq i \leq m$. By the induction hypothesis there exists $\sigma_i : Z_i \overset{s}{\twoheadrightarrow} N_i$ for $0 \leq i \leq m$. Let σ be the reduction sequence $M \overset{h}{\twoheadrightarrow} Z$. Then, we can construct a standard reduction sequence from M to N by appending these reduction sequences in the following order: $\sigma, \sigma_0, ..., \sigma_m$. □

It can be observed that some notions involved in this proof such as the one of residuals, the definition of a standard reduction sequence or that of a complete development of a set of redexes may be very challenging to define inductively. In addition, some non-trivial results such as FD and FD! are needed.

The proof hereby presented follows the one proposed by Ryo Kashima in [5] where a notion of β-reducibility with a standard sequence is captured by an inductive relation as opposed to using residuals, allowing for an elegant inductive development. This proof neither relies on the concept of residuals nor distinguishes between head and internal reductions.

Due to its inductive structure, Kashima's proof can be conveniently formalized in Constructive Type Theory. In fact, this proof has been checked using Matita [11] by Ferruccio Guidi in [12] where formalizations for the Standardization and Confluence theorems for the Lambda Calculus are presented. Other formalizations include a proof in Coq [13] by Ignas Vyšniauskas and Johannes Emerich [14].

A noticeable difference between these efforts and our development is the syntax used to represent λ-terms, which in their case is based on de Brujin indices. While this is a common strategy across the literature, in some cases it forces the induction to be done on the size of the terms, which results in rather intricate proofs. As we shall see in the next chapters, our definition of multiple substitution allows us to work with concrete syntax and to define stronger lemmas that can be easily proven using structural induction only.

All the definitions and proofs that appear in this thesis have been fully formalized in Constructive Type Theory [1] and machine-checked with the system Agda [4]. In the subsequent text we will mix Agda code and (informal) proofs in English with a considerable level of detail so that they serve for clarifying their formalization. Some familiarity with with the Agda syntax or at least

functional languages will be assumed. The corresponding code is available at `https://github.com/mcopes73/standardization-agda/`.

1.2 Structure of this Thesis

This thesis is structured as follows:

- In Chapter 2 we present the basic concepts of the Lambda Calculus, together with some definitions and results from the library produced in [3] in which our work is based and the extensions thereof.

- In Chapter 3 we present the proof of the Standardization theorem.

- In Chapter 4 we present the proof of the Leftmost Reduction theorem for β-reduction, i.e., the fact that the leftmost-outermost reduction strategy always finds the normal form for normalizing terms, as is a consequence of the Standardization theorem.

- In Chapter 5 we compare our development with other similar efforts in the literature, and present our overall conclusions.

- Finally, the complete Agda development can be found in the Appendix.

Chapter 2

Preliminaries

In the first sections of this chapter we will introduce the main definitions and results in [3; 6] that are previous to this work and are used in our formalization. We present the definitions directly using our Agda code along with informal explanations, while the proofs are written in English, and can be found in their formal version in the Appendix. A certain degree of familiarity with the Agda syntax or at least with that of functional languages is assumed.

2.1 Lambda Terms

We shall start by defining λ-terms using the same set of names for both bound and free variables. We use natural numbers to name variables for sake of concreteness.

```
V = ℕ
data Λ : Set where
  v    : V → Λ
  _·_  : Λ → Λ → Λ
  ƛ    : V → Λ → Λ
```

Agda is pretty liberal with regard to the naming of functions and the positions of their arguments. Notice the notation for declaring the infix application constructor, i.e. `_·_`. This underscore notation is extended to mixfix operators.

The classical notions of *free* and *fresh* (not free) variable in a term, which are denoted by $*$ and $\#$ respectively, are defined as follows:

```
data _*_ : V → Λ → Set where
  *v   :  {x : V}  → x * v x
  *·l  :  {x : V}{M N : Λ} → x * M  → x * M · N
  *·r  :  {x : V}{M N : Λ} → x * N  → x * M · N
  *ƛ   :  {x y : V}{M : Λ} → x * M → y ≢ x  → x * ƛ y M

data _#_ : V → Λ → Set where
  #v  :  {x y : V} →  y ≢ x  →  x # v y
  #·  :  {x : V}{M N : Λ}  →  x # M → x # N  →  x # M · N
  #ƛ≡ :  {x : V}{M : Λ}  →  x # ƛ x M
  #ƛ  :  {x y : V}{M : Λ}  →  x # M  →  x # ƛ y M
```

Arguments to a function declared between braces `{}` are optional in that in subsequent applications of the function in question they can be omitted and are inferred by the type-checker.

2.2 Substitutions

Substitutions are *identity-almost-everywhere* functions associating a term to every variable. Therefore, we can generate every concrete substitution by starting up from the empty substitution ι that maps each variable to itself as a term, and employing the update operation $\prec\!+$, such that if σ is a substitution, then $\sigma \prec\!+ (x, M)$ is the substitution equal to σ everywhere except at x, where it yields M:

```
Σ = V → Λ

ι : Σ
ι = id ∘ v

_≺+_ : Σ → V × Λ → Σ
(σ ≺+ (x , M)) y with x ≟ y
```

```
... | yes _ = M
... | no  _ = σ y
```

Notice that in the definition of $\prec+$ we use the `with` construct, which allows us to perform pattern matching on the result of evaluating the expression $\mathtt{x} \stackrel{?}{=} \mathtt{y}$. This expression represents the decidable equality between the numbers `x` and `y` and has type `Dec` $\equiv$, which has constructors `yes` and `no` applied to the corresponding proof objects.

In general, we shall consider properties concerning the substitutions for the free variables of a term M, i.e. their *restriction* to such variables. The type of restrictions `R` is defined as: `R = Σ × Λ`, and we note in the informal language such a restriction as $\sigma \restriction M$. This means that we are restricting the substitution σ to the free variables of M only.

We will also use the following notion: $x \# (\sigma \restriction M)$, which stands for *$x$ fresh in the σ-images of all the free variables of M*:

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

The application of substitution σ to the term M is noted $M \bullet \sigma$, and it is defined by *structural recursion* on M. The fact that structural recursion is sufficient for stating this very concrete definition is a (very welcome) non-trivial consequence of the employment of multiple substitutions.

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

Notice the last line of the definition: when performing a substitution over an abstraction, the bound variable is always replaced with a new one. This new variable is obtained by means of a choice function χ, such that $\chi(\sigma, M) \# (\sigma \restriction M)$. In this

way, the new variable does not capture any of the names introduced into its scope by effect of the substitution[1]. When reasoning with substitutions, this uniform renaming of bound variables allows us to avoid case analyses; it also has other nice consequences, to be noticed shortly.

For the sake of readability, we define the single substitution of a term N for a variable x in M with the traditional notation $M[x := N]$.

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

2.3 Alpha-conversion

Alpha-conversion ($\sim_\alpha$) is defined as the following inductive binary relation on terms:

```
data _∼α_ : Λ → Λ → Set where
  ∼v : {x : V} → (v x) ∼α (v x)
  ∼· : {M M' N N' : Λ} → M ∼α M' → N ∼α N' → M · N ∼α M' · N'
  ∼ƛ : {M M' : Λ}{x x' y : V} → y # ƛ x M → y # ƛ x' M'
                              → M [x := v y] ∼α M' [x' := v y]
                              → ƛ x M ∼α ƛ x' M'
```

The first two constructors implement the classical rules for variables and application. The last constructor states that two abstractions are α-convertible if and only if their bodies are α-convertible after replacing the bound variables with a common fresh name. From this definition it follows that $\sim_\alpha$ is an equivalence relation, as shown in [3].

Another interesting property of this definition is that α-equivalent terms become identical when submitted to the same substitution. This is due to the fact that abstractions are uniformly renamed, and that the new name chosen by the χ function is determined only by the restriction of the substitution to the free variables of the terms, which is the same one if the terms are α-equivalent. This is proven in [3], and we just mention the corresponding lemma here:

[1] In fact, χ is implemented by just finding the first variable not free in the given restriction.

```
lemmaM∼M'→Mσ≡M'σ : {M M' : Λ}{σ : Σ} → M ∼α M' → M • σ ≡ M' • σ
```

As part of our work we have proven that this definition of alpha equivalence is decidable (the proof can be found in the Appendix):

```
_∼α?_ : ∀ A B -> Dec (A ∼α B)
```

2.4 Alpha Reflexive Transitive Closure

From now on we present definitions and results not included in the library [6].

Given a binary relation $\rightsquigarrow$, we define its α-reflexive-transitive closure as follows:

```
data α-star (⇝ : Rel) : Rel where
    refl : ∀{M} → α-star ⇝ M M
    α-step : ∀{M N N'} → α-star ⇝ M N' → N' ∼α N → α-star ⇝ M N
    append : ∀ {M N K} → α-star ⇝ M K → ⇝ K N → α-star ⇝ M N
```

where `Rel` is the type of binary relations over terms.

This is the kind of closure that will be applied to our contraction relations. It represents sequences of $\rightsquigarrow$ steps allowing α conversions, which have to be made explicit when dealing with concrete terms. We note the α-reflexive-transitive closure of a relation with the classical two-headed arrow. From this definition we can easily prove that, for any relation $\rightarrow$, $M \rightarrow N$ implies $M \twoheadrightarrow N$, and that $\twoheadrightarrow$ is transitive. The first proof is straightforward using the constructors `append` and `refl`. Transitivity is proven by induction on the definition of `α-star`.

```
α-star-singl : ∀{⇝ M N} -> ⇝ M N -> α-star ⇝ M N

α-star-trans : ∀{⇝ M N K} -> α-star ⇝ M K -> α-star ⇝ K N -> α-star ⇝ M N
```

2.5 Beta Reduction

Following Kashima [5], we define β-contraction taking into account the position where the contracted redex appears in the term relative to the other redexes. We start by defining two auxiliary functions: `isAbs` is a predicate that decides whether a term is an abstraction and `countRedexes` a function that counts the number of β-redexes in a term.

```
data isAbs : Λ -> Set where
  abs : forall {x M} -> isAbs (ƛ x M)
```

We need to prove that `isAbs` is decidable before being able to define `countRedexes`, since the number of redexes for the application case depends on whether the left term is an abstraction. The proof is declared as follows and defined straightforwardly:

```
isAbs? : (M : Λ) -> Dec (isAbs M)
```

Using this property we can define `countRedexes` as follows:

```
countRedexes : Λ -> ℕ
countRedexes (v _) = 0
countRedexes (M · N) with isAbs? M
... | yes _ = suc (countRedexes M + countRedexes N)
... | no _ = countRedexes M + countRedexes N
countRedexes (ƛ _ M) = countRedexes M
```

Considering the linear syntax of terms, redexes will be numbered in a left-to-right fashion, starting from zero. We shall start by defining the *contraction of the n-th redex* as a relation between terms depending on the natural number n.

```
data _β_@_ : Λ -> Λ -> ℕ -> Set where
  outer-redex : ∀ {x A B} -> ((ƛ x A) · B) β (A [ x := B ]) @ 0
  appNoAbsL : ∀ {n A B C} -> A β B @ n -> ¬ isAbs A
```

```
    -> (A · C) β (B · C) @ n
  appAbsL : ∀ {n A B C} -> A β B @ n -> isAbs A
    -> (A · C) β (B · C) @ (suc n)
  appNoAbsR : ∀ {n A B C} -> A β B @ n -> ¬ isAbs C
    -> (C · A) β (C · B) @ (n + countRedexes C)
  appAbsR : ∀ {n A B C} -> A β B @ n -> isAbs C
    -> (C · A) β (C · B) @ (suc (n + countRedexes C))
  abs : ∀ {n x A B} -> A β B @ n -> (ƛ x A) β (ƛ x B) @ n
```

The `outer-redex` constructor allows the contraction of the outermost redex, numbered as the one at position zero. The next four constructors are used to perform contractions inside applications. In order to determine the number of the redex contracted we need to identify whether the left hand side term of the application is an abstraction or not. This is necessary to know whether we are stepping over a redex to reduce an inner one. Finally, the `abs` constructor allows contractions inside an abstraction.

For example, we can prove that $((\lambda x\ (\lambda y.y)\ x)\ z)\ \beta\ ((\lambda x\ (y\ [y := x]))\ z)\ @\ 1$. In Agda:

```
β-example1 : {x y z} -> ((ƛ x ((ƛ y (v y)) · (v x))) · (v z))
    β ((ƛ x ((v y) [ y := v x ])) · (v z)) @ 1
β-example1 = appAbsL (abs outer-redex) ƛabs
```

In this example we are performing a contraction on the left hand side of an application and since the term is an abstraction, we are stepping over a redex. Thus, we use the the `appAbsL` rule which increments by one the number of the redex contracted on the left hand side to account for the outermost one.

One-step *β-reduction* ($\longrightarrow_\beta$) from M to N can now be defined as the existence of a natural number n such that N can be obtained by contracting the n-th redex from M. We use Agda's dependent ordered pair constructor Σ to express existential quantification.

```
_⟶β_ : Λ -> Λ -> Set
M ⟶β N = Σ ℕ (\n -> M β N @ n)
```

It is easily proven by structural induction that this definition of β-contraction is equivalent to the following classical definition:

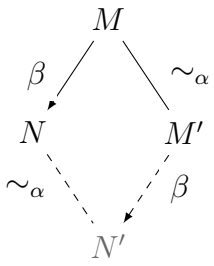
```
data _⟶βcl_ : Λ -> Λ -> Set where
  redex : ∀ {x M N} -> ((ƛ x M) · N) ⟶βcl (M [ x := N ])
  app-l : ∀ {M M' N} -> M ⟶βcl M' -> (M · N) ⟶βcl (M' · N)
  app-r : ∀ {N N' M} -> N ⟶βcl N' -> (M · N) ⟶βcl (M · N')
  abs : ∀ {x M M'} -> M ⟶βcl M' -> (ƛ x M) ⟶βcl (ƛ x M')
```

The proofs for both inclusions are direct and they are shown in the Appendix.

```
β@-implies-βcl : ∀{M N} -> M ⟶β N -> M ⟶βcl N
```

```
βcl-implies-β@ : ∀{M N} -> M ⟶βcl N -> M ⟶β N
```

One interesting result that will be useful in our development is the following α-β diamond property:



which we state in Agda as the following lemma:

```
lem-βα : ∀{M N M'} -> M ⟶β N -> M ∼α M'
                           -> Σ Λ  (λ N' -> (M' ⟶β N') ∧ (N' ∼α N))
```

We finally introduce β-reduction $\twoheadrightarrow_\beta$ as the α-reflexive-transitive closure of the contraction $\longrightarrow_\beta$:

```
_→→β_ : Λ -> Λ -> Set
_→→β_ = α-star (_⟶β_)
```

Chapter 3

The Standardization Theorem

In the present chapter we show the formalization of the Standardization theorem in Constructive Type Theory that follows the proof given by Kashima in [5]. We present all the definitions using Agda code, and write the proofs in English with a considerable level of detail so that they serve for clarifying their formalization (which can be found in the Appendix). For the sake of clarity, some lemmas are presented in a different order than the one proposed by Kashima. Nonetheless, the formalized results and definitions are the same unless otherwise stated.

3.1 Standard Reduction Sequences

A *reduction sequence* is a sequence of terms $M_0, M_1, ..., M_n$ such that M_{i+1} is obtained from M_i by the contraction of some redex, i.e.,

$$(\forall\ i \in 0...n{-}1)\ M_i\ \longrightarrow_\beta M_{i+1}$$

We call a reduction sequence *standard* if and only if subsequent steps are non decreasing in the number of the redex contracted. Kashima defines a standard beta reduction sequence as:

$$\text{If } M_0\ \xrightarrow{n_1}_\beta M_1\ \xrightarrow{n_2}_\beta\ ...\ \xrightarrow{n_k}_\beta\ M_k\ \text{ then }\ n_1 \leq n_2 \leq ... \leq n_k$$

where $M \xrightarrow{n}_\beta N$ is the β-contraction of the n-th redex in M, which we note $M\,\beta\,N\,@\,n$ in our development.

We implement this notion in Agda by defining a relation indexed on a natural number that keeps track of the lower bound allowed for the next redex to be contracted.

```
data seqβ-st (M : Λ) : (N : Λ) -> ℕ -> Set where
  nil : seqβ-st M M 0
  α-step : ∀ {n K N} -> seqβ-st M K n -> K ∼α N -> seqβ-st M N n
  β-step : ∀ {K n n₀ N} -> seqβ-st M K n -> K β N @ n₀ -> n₀ ≥ n
                                                    -> seqβ-st M N n₀
```

The relation is reflexive and allows for α-steps, which do not appear explicitly in Kashima's definition, but are implicitly used whenever β-reduction is involved. The "three dots" of Kashimas's sequence $M_0, M_1, ..., M_k$ are implemented as follows: we can append a term to the reduction sequence provided that it can be obtained by the contraction of a redex at a position greater than or equal to the current lower bound. Clearly this leaves out of admissible further choice every residual of a redex appearing to the left of the one contracted, for such a residual must necessarily remain with the same number it had received in the original term, i.e. one smaller than that of the redex being contracted. Therefore, this definition captures exactly Barendregt's one given in Chapter 1.

Using this relation the Standardization theorem can be precisely stated as the existence of a standard sequence between two terms among which there is a β-reduction:

```
standardization : ∀{M N} -> M →→β N -> Σ ℕ (λ n -> seqβ-st M N n)
```

3.2 Two Useful Reduction Relations

The next step is to capture the existence of a standard sequence in an inductively defined reduction relation between terms. To this end, Kashima introduces two auxiliary contraction relations, namely $\longrightarrow_l$ and $\longrightarrow_{hap}$.

$\longrightarrow_l$ stands for *leftmost* and denotes the contraction of the leftmost redex, i.e. the one at position zero:

```
data _⟶l_ : Λ -> Λ -> Set
    M ⟶l N =  M β N @ 0
```

$\longrightarrow_{hap}$ stands for *head reduction in application* and represents the β-contraction of the redex in the head position of a chain of applications, i.e.:

$$(\lambda x M_0) M_1 M_2 ... M_n \ \longrightarrow_{hap} \ M_0[x := M_1] \ M_2 ... M_n$$

We define this relation in Agda as follows:

```
data _⟶hap_ : Λ -> Λ -> Set where
  hap-head : ∀{x A B} -> (ƛ x A) · B ⟶hap (A [ x := B ])
  hap-chain : ∀{C A B} -> A ⟶hap B -> (A · C) ⟶hap (B · C)
```

$\twoheadrightarrow_l$ and $\twoheadrightarrow_{hap}$ are defined as the α-reflexive-transitive closures of $\longrightarrow_l$ and $\longrightarrow_{hap}$ respectively.

```
_→→hap_ : Λ -> Λ -> Set
_→→hap_ = α-star (_⟶hap_)

_→→l_ : Λ -> Λ -> Set
_→→l_ = α-star (_⟶l_)
```

Properties of $\twoheadrightarrow_{hap}$ and $\longrightarrow_l$

The first two lemmas state that head reduction in application $\twoheadrightarrow_{hap}$ is compatible with application and substitution respectively.

```
hap-app-r : ∀{M N P} -> M →→hap N -> M · P →→hap N · P
```

Proof. By induction on the definition of $M \twoheadrightarrow_{hap} N$.

- Case `refl`: We have to prove $(M\ P) \twoheadrightarrow_{hap} (M\ P)$, which follows by `refl`.
- Case `α-step`: Assume that $M \twoheadrightarrow_{hap} N$ follows from $M \twoheadrightarrow_{hap} N'$ and $N' \sim_\alpha N$. Then, we obtain $M\ P \twoheadrightarrow_{hap} N'\ P$ from the induction hypothesis, and since $N'\ P \sim_\alpha N\ P$, we construct our goal using `st-alpha`.
- Case `append`: Assume $M \twoheadrightarrow_{hap} N$ follows from $M \twoheadrightarrow_{hap} K$ and $K \longrightarrow_{hap} N$. Then we can obtain $M\ P \twoheadrightarrow_{hap} K\ P$ from the induction hypothesis and $K\ P \longrightarrow_{hap} N\ P$ from rule `hap-chain` applied to $K \longrightarrow_{hap} N$. From these, we construct our goal using `append`.

□

In order to prove that substitution preserves the head reduction relation, we need two lemmas from the substitution library [6]. The first one states that substituting y for x and then N for y yields a result α-equivalent to substituting N for x, provided y is fresh enough. The second one is a form of the substitution composition lemma:

```
corollary1SubstLemma : {x y : V} {σ : Σ}{M N : Λ} → y #↓ (σ , ƛ x M)
    → ((M • (σ ≺+ (x , v y))) [y := N]) ∼α  (M • (σ ≺+ (x , N)))

corollary1Prop7 : {M N : Λ}{σ : Σ}{x : V}
    → M • (σ ≺+ (x , N • σ)) ≡ (M [x := N]) • σ
```

Now we prove that substitution preserves $\longrightarrow_{hap}$ up to $\sim_\alpha$:

```
lem-hap-subst : ∀{σ M N} -> M ⟶hap N
    ->  Σ Λ (λ N' -> ((M • σ) ⟶hap N') ∧ (N' ∼α (N • σ)))
```

Proof. By induction on the definition of $M \longrightarrow_{hap} N$

- Case `hap-head`: We want to prove that $((ƛx\ A)\ B) \bullet \sigma \longrightarrow_{hap} N \wedge N \sim_\alpha (A[x := B]) \bullet \sigma$, for some term N. Starting from the left hand side:
  $((ƛx\ A)\ B) \bullet \sigma$
  $\equiv$ (Def. $\bullet$)
  $(ƛy\ A \bullet (\sigma \prec + (x, y)))\ (B \bullet \sigma)$ where $y = \chi(\sigma, ƛx\ A)$
  $\longrightarrow_{hap}$ (`hap-head`)
  $(A \bullet (\sigma \prec + (x, y)))\ [y := B \bullet \sigma]$

$\sim_\alpha$ (`corollary1substLemma`, $y \# (\sigma, \lambda x\ A)$)
$A \bullet (\sigma \prec + (x,\ B \bullet \sigma))$
$\equiv$ (`corollary1Prop7`)
$(A\ [x := B]) \bullet \sigma$

- Case `hap-chain`: We need to prove that there exists a term K such that $(M\ P) \bullet \sigma \longrightarrow_{hap} K \wedge K \sim_\alpha (N\ P) \bullet \sigma$, assuming $M \bullet \sigma \longrightarrow_{hap} N \bullet \sigma$. This follows directly from rule `hap-chain` applied to $M \bullet \sigma \longrightarrow_{hap} N \bullet \sigma$ and the definition of $\bullet$.

□

Kashima originally formulates the previous result just for single substitutions, i.e., of the form $[x := P]$. Our result using multiple substitutions will allow us to rely only on structural induction in our proofs, as we shall see later. We can easily extend the previous result to $\twoheadrightarrow_{hap}$ and generalize it:

```
hap-subst : ∀{M N σ} -> M →→hap N -> (M • σ) →→hap (N • σ)
```

Proof. By induction on $M \twoheadrightarrow_{hap} N$:

- Case `refl`: Direct using `refl`.
- Case `α-step`: Assume $M \twoheadrightarrow_{hap} N'$ and $N' \sim_\alpha N$. Then, we obtain $M \bullet \sigma \twoheadrightarrow_{hap} N' \bullet \sigma$ from the induction hypothesis, and by `lemmaM∼M'→Mσ≡M'σ` mentioned in Section 2.3, we have that $N' \bullet \sigma \equiv N \bullet \sigma$, so we construct our goal using the `α-step` rule, since $\sim_\alpha$ is reflexive.
- Case `append`: Assume $M \twoheadrightarrow_{hap} N$ follows from $M \twoheadrightarrow_{hap} K$ and $K \longrightarrow_{hap} N$. Then we can obtain $M \bullet \sigma \twoheadrightarrow_{hap} K \bullet \sigma$ from the induction hypothesis and $(\exists N')(K \bullet \sigma \longrightarrow_{hap} N' \wedge N' \sim_\alpha N \bullet \sigma)$ from the previous lemma (`lem-hap-subst`) applied to $K \longrightarrow_{hap} N$. From this, using `α-star-single` we construct $K \bullet \sigma \twoheadrightarrow_{hap} N'$ and since $N' \sim_\alpha N$ we obtain $K \bullet \sigma \twoheadrightarrow_{hap} N$ from rule α`-step`. Finally, we prove our goal from the transitivity of $\twoheadrightarrow_{hap}$.

□

Finally, notice that head reduction in application implies leftmost reduction:

```
lem-hap→l : ∀ {M N} -> M ⟶hap N -> M ⟶l N
```

Proof. By induction in the definition of $M \longrightarrow_{hap} N$.

- The `hap-head` case is direct.
- The `hap-chain` case follows from the fact that if $A \longrightarrow_{hap} B$ then A cannot be an abstraction (it can only be an application). The induction hypothesis tells us that $A\,\beta\,B\,@\,0$ and since A is not an abstraction we know that $(A\ C)\,\beta\,(B\ C)\,@\,0$ because of rule `appNoAbsL`.

□

Now the same inclusions can be proven for their α-reflexive-transitive closures:

```
hap→l : ∀{M N} -> M →→hap N -> M →→l N
```

3.3 Standard Reduction

Using $\twoheadrightarrow_{hap}$ Kashima characterizes the existence of a standard sequence as a further reduction relation $\twoheadrightarrow_{st}$, which stands for *standard reduction*, as follows:

```
data _→→st_ (L : Λ) : Λ -> Set where
  st-var : ∀{x} -> L →→hap (v x) -> L →→st (v x)
  st-app : ∀{A B C D} -> L →→hap (A · B) -> A →→st C -> B →→st D ->
    L →→st (C · D)
  st-abs : ∀{x A B} -> L →→hap (ƛ x A) -> A →→st B -> L →→st (ƛ x B)
  st-alpha : ∀{A' A} -> L →→st A' -> A' ∼α A -> L →→st A
```

The intention behind this relation is to characterize standard reduction sequences inductively. This definition allows us to perform as many $\twoheadrightarrow_{hap}$ steps as we want. After that, if we reach a variable, then we are done since we cannot do any more reductions (`st-var`). If the term is an application $A\ B$, then we can continue performing standard reductions on A and then on B (`st-app`). Finally, if the term is an abstraction, we can continue performing standard reductions inside the body of the abstraction (`st-abs`). Note that we are not forced to reduce all of the redexes

that we encounter; given a redex, we can still apply `st-app` while skipping the head reduction. The last constructor (`st-alpha`) allows us to perform α-conversion.

The notion of standard reduction can be extended to substitutions. We say that *substitution σ standard-reduces to σ'* ($\sigma \to_{st} \sigma'$) if and only if for all variables x, $\sigma\ x \twoheadrightarrow_{st} \sigma'\ x$.

```
_→st_ : Σ → Σ → Set
σ →st σ' = (x : V) → σ x →→st σ' x
```

Properties of $\twoheadrightarrow_{st}$

Reflexivity is proven by a direct induction on $M : \Lambda$.

```
st-refl : ∀{M} -> M →→st M
```

Appending head reductions in applications at the beginning of a standard reduction results in a standard reduction.

```
hap-st→st : ∀{L M N} -> L →→hap M -> M →→st N -> L →→st N
```

Proof. By induction on the definition of $M \twoheadrightarrow_{st} N$.

- Case `st-var`: We know that $M \twoheadrightarrow_{hap} x$. From this, $L \twoheadrightarrow_{hap} M$ and the transitivity of $\twoheadrightarrow_{hap}$ we conclude that $L \twoheadrightarrow_{hap} x$, and then $L \twoheadrightarrow_{st} x$ follows from `st-var`.
- For the `st-app` case, assume $M \twoheadrightarrow_{hap} A\ B$, $A \twoheadrightarrow_{st} C$ and $B \twoheadrightarrow_{st} D$. From $L \twoheadrightarrow_{hap} M$ and $M \twoheadrightarrow_{hap} A\ B$ we conclude that $L \twoheadrightarrow_{hap} A\ B$ by transitivity of $\twoheadrightarrow_{hap}$. Finally, from this plus $A \twoheadrightarrow_{st} C$ and $B \twoheadrightarrow_{st} D$, we conclude that $L \twoheadrightarrow_{st} C\ D$ using `st-app`.
- For the `st-abs` case, we assume $M \twoheadrightarrow_{hap} \lambda x\ A$ and $A \twoheadrightarrow_{st} B$. Similarly to the preceding case, we conclude that $L \twoheadrightarrow_{hap} \lambda x\ A$ from $L \twoheadrightarrow_{hap} M$, $M \twoheadrightarrow_{hap} \lambda x\ A$ and the transitivity of $\twoheadrightarrow_{hap}$. From this and $A \twoheadrightarrow_{st} B$, we conclude $L \twoheadrightarrow_{st} \lambda x\ B$ using `st-abs`.

- For the `st-alpha` case, we assume $M \twoheadrightarrow_{st} A'$ and $A' \sim_\alpha A$. We use constructor `st-alpha` applied to the induction hypothesis $L \twoheadrightarrow_{st} A'$ and $A' \sim_\alpha A'$ to complete our goal.

□

We can now use the preceding lemma to prove that substitution is preserved by the $\twoheadrightarrow_{st}$ relation. This lemma is key to the proof and was originally stated by Kashima as: $M \twoheadrightarrow_{st} N$ and $P \twoheadrightarrow_{st} Q \implies M[z := P] \twoheadrightarrow_{st} N[z := Q]$. However, depending on the definition of substitution, stating the theorem in such way may require induction on the size of the lambda term to prove it, as we mentioned in the introduction. Our definition of substitution allows us to prove this result using just structural induction if we take the substitution to be an arbitrary (multiple) σ instead of the particular case where we replace just one variable z. This substitution lemma is stated as follows:

```
st-substσ≅σ' : ∀{M N σ σ'} -> M →→st N -> σ →st σ' -> M • σ →→st N • σ'
```

Proof. By induction on the definition of $M \twoheadrightarrow_{st} N$

- Case `st-var`: We have to prove $M \bullet \sigma \twoheadrightarrow_{st} x \bullet \sigma'$ under the hypotheses $M \twoheadrightarrow_{hap} x$ and $\sigma \rightarrow_{st} \sigma'$.
  From `hap-subst` applied to $M \twoheadrightarrow_{hap} x$ we know that $M \bullet \sigma \twoheadrightarrow_{hap} x \bullet \sigma$ and from the definition of $\sigma \rightarrow_{st} \sigma'$ we get that $x \bullet \sigma \twoheadrightarrow_{st} x \bullet \sigma'$. Therefore, from $M \bullet \sigma \twoheadrightarrow_{hap} x \bullet \sigma \twoheadrightarrow_{st} x \bullet \sigma'$ we conclude that $M \bullet \sigma \twoheadrightarrow_{st} x \bullet \sigma'$ using `hap-st→st`.
- Case `st-app`: Assume $M \twoheadrightarrow_{hap} A\ B$, $\sigma \rightarrow_{st} \sigma'$, $A \bullet \sigma \twoheadrightarrow_{st} C \bullet \sigma'$ and $B \bullet \sigma \twoheadrightarrow_{st} D \bullet \sigma'$. We have to prove $M \bullet \sigma \twoheadrightarrow_{st} (C\ D) \bullet \sigma'$. Now:
  $M \twoheadrightarrow_{hap} A\ B$
  $\implies$ (`hap-subst`)
  $M \bullet \sigma \twoheadrightarrow_{hap} (A\ B) \bullet \sigma$
  $\equiv$ (Def. $\bullet$)
  $M \bullet \sigma \twoheadrightarrow_{hap} (A \bullet \sigma)\ (B \bullet \sigma)$
  $\implies$ (`st-app` and hypothesis)
  $M \bullet \sigma \twoheadrightarrow_{st} (C \bullet \sigma')\ (D \bullet \sigma')$

$\equiv$ (Def. $\bullet$)
$M \bullet \sigma \twoheadrightarrow_{st} (C\ D) \bullet \sigma'$.

- Case `st-abs`: Assume $M \twoheadrightarrow_{hap} \lambda x\ A$, $A \twoheadrightarrow_{st} B$ and $\sigma \rightarrow_{st} \sigma'$. We prove $M \bullet \sigma \twoheadrightarrow_{st} (\lambda x\ B) \bullet \sigma'$:
  $M \twoheadrightarrow_{hap} \lambda x\ A$
  $\Longrightarrow$ (`hap-subst`)
  $M \bullet \sigma \twoheadrightarrow_{hap} (\lambda x\ A) \bullet \sigma$
  $\equiv$ (Def $\bullet$)
  $M \bullet \sigma \twoheadrightarrow_{hap} \lambda y_A\ (A \bullet \sigma \prec + (x,\ y_A))$ where $y_A = \chi(\sigma, \lambda x\ A)$ (1).
  Let $z = \chi(\iota \downarrow ((\lambda x\ A) \bullet \sigma)\ ((\lambda x\ B) \bullet \sigma'))$. Due to the definition of the choice function χ, z is fresh in every term and substitution involved. We can now prove that:
  $\lambda y_A\ (A \bullet \sigma \prec + (x,\ y_A)) \sim_\alpha \lambda z\ (A \bullet \sigma \prec + (x,\ z))$
  $\Longrightarrow$ (`hap-`α and (1))
  $M \bullet \sigma \twoheadrightarrow_{hap} \lambda z\ (A \bullet \sigma \prec + (x,\ z))$ (2).
  Note that by using multiple substitutions, our induction hypothesis is strong enough to allow us to use it with any pair of substitutions σ, σ' as long as $\sigma \rightarrow_{st} \sigma'$. Therefore, we can extract the following induction hypothesis from $A \twoheadrightarrow_{st} B$:
  $A \bullet \sigma \prec + (x,\ z) \twoheadrightarrow_{st} B \bullet \sigma' \prec + (x,\ z)$ (3).
  We can prove that $\sigma \prec + (x,\ z) \rightarrow_{st} \sigma' \prec + (x,\ z)$ because the $\twoheadrightarrow_{st}$ relation is reflexive (lemma `st-refl`), so replacing x for z in both substitutions will preserve the $\rightarrow_{st}$ relation. So, from (2), (3) and constructor `st-abs` we obtain that $M \bullet \sigma \twoheadrightarrow_{st} \lambda z\ (B \bullet \sigma' \prec + (x,\ z))$ and we obtain our thesis using `st-alpha`, since $\lambda z\ (B \bullet \sigma' \prec + (x,\ z)) \sim_\alpha (\lambda x\ B) \bullet \sigma'$, .
- Case `st-alpha`: We assume that $M \twoheadrightarrow_{st} N'$ and $N' \sim_\alpha N$ and want to prove that $M \bullet \sigma \twoheadrightarrow_{st} N \bullet \sigma'$. From the induction hypothesis we get that $M \bullet \sigma \twoheadrightarrow_{st} N' \bullet \sigma'$. In addition, we know that $N' \bullet \sigma' \sim_\alpha N \bullet \sigma'$, since they are equal (`lemmaM`$\sim$`M'`$\rightarrow$`M`$\sigma\equiv$`M'`σ) and $\sim_\alpha$ is reflexive. From these we obtain our goal using the `st-alpha` rule.

□

The following lemma states that if there is a standard reduction to an application that is a redex $(\lambda x.M)\ N$, then it is possible to construct a standard reduction to the contractum $M[x := N]$, somehow "inserting" the contraction in a right place:

```
st-abs-subst : ∀ {L M N x} -> L →→st (ƛ x M) · N -> L →→st (M [ x := N ])
```

Proof. From $L \twoheadrightarrow_{st} (\lambda x\ M)\ N$ and the definition of $\twoheadrightarrow_{st}$ we know that $L \twoheadrightarrow_{hap} P\ N'$ for some P and N' such that $P \twoheadrightarrow_{st} (\lambda x\ M)$ and $N' \twoheadrightarrow_{st} N$. In addition, from $P \twoheadrightarrow_{st} (\lambda x\ M)$ we know that $P \twoheadrightarrow_{hap} (\lambda x\ M')$ for some M' such that $M' \twoheadrightarrow_{st} M$. Then,
$P \twoheadrightarrow_{hap} (\lambda x\ M')$
$\Longrightarrow$ (`hap-app-r`)
$P\ N' \twoheadrightarrow_{hap} (\lambda x\ M')\ N'$ (1).
On the other hand,
$(\lambda x\ M')\ N'$
$\twoheadrightarrow_{hap}$ (`α-star-singl` applied to constructor `hap-head`)
$M'\ [x := N']$
$\twoheadrightarrow_{st}$ (`st-substσ≅σ'` with $M' \twoheadrightarrow_{st} M$ and $N' \twoheadrightarrow_{st} N$)
$M\ [x := N]$ (2).
From $L \twoheadrightarrow_{hap} P\ N'$, (1), (2) and the transitivity of $\twoheadrightarrow_{hap}$ we get that $L \twoheadrightarrow_{hap} M'\ [x := N']$, and since $M'\ [x := N'] \twoheadrightarrow_{st} M\ [x := N]$ we conclude that $L \twoheadrightarrow_{st} M\ [x := N]$ using lemma `hap-st→st`.

□

Using this result, we can prove now that any β-contraction can be "inserted" into a standard reduction:

```
st-β→st : ∀{L M N} -> L →→st M -> M ⟶β N -> L →→st N
```

Proof. By induction on $M \longrightarrow_{\beta} N$.

- The case `outer-redex` follows directly from the previous lemma `st-abs-subst`.
- All the application cases are solved by simply using `st-app` applied to the induction hypotheses. For example, if $M \longrightarrow_{\beta} N$ was constructed using the rule `appAbsL` then we know that $M = A\ C$, $N = B\ C$ and $(A\ C)\,\beta\,(B\ C)\,@\,(suc\ n)$, with $A\,\beta\,B\,@\,n$ for some n. Since M is an application, $L \twoheadrightarrow_{st} M$ must have been constructed using either the `st-app` constructor or the `st-alpha` constructor. We will deal with all the `st-alpha` cases uniformly at the end, so let

us focus on the `st-app` case for now. We know that $L \twoheadrightarrow_{hap} A'\ C'$, $A' \twoheadrightarrow_{st} A$ and $C' \twoheadrightarrow_{st} C$. We want to prove that $L \twoheadrightarrow_{st} B\ C$. From $A' \twoheadrightarrow_{st} A$ and $A \longrightarrow_{\beta} B$ we get that $A' \twoheadrightarrow_{st} B$ by the induction hypothesis. Finally, we prove this case using the `st-app` rule applied to $L \twoheadrightarrow_{hap} A'\ C'$, $A' \twoheadrightarrow_{st} B$ and $C' \twoheadrightarrow_{st} C$. The proofs for the other three application cases follow the same structure.

- The `abs` case also follows a similar pattern. We know that $\lambda x A \longrightarrow_{\beta} \lambda x B$ where $A \longrightarrow_{\beta} B$. Therefore, considering that $L \twoheadrightarrow_{st} \lambda x A$ was constructed using the `st-abs` rule, we have that $L \twoheadrightarrow_{hap} \lambda x A'$ and $A' \twoheadrightarrow_{st} A$ for some A'. The induction hypothesis applied to $A' \twoheadrightarrow_{st} A$ and $A \longrightarrow_{\beta} B$ gives us that $A' \twoheadrightarrow_{st} B$, and we obtain our goal $L \twoheadrightarrow_{st} \lambda x B$ using the `st-abs` rule applied to $L \twoheadrightarrow_{hap} \lambda x A'$ and $A' \longrightarrow_{\beta} B$.
- In all the previous cases we ignored the case where `L` $\twoheadrightarrow_{st}$`M` was constructed using the `st-alpha` constructor since we can prove this uniformly for all cases. We know that $L \twoheadrightarrow_{st} M'$ and $M' \sim_{\alpha} M$. In order to use the induction hypothesis we would need to have that $M' \longrightarrow_{\beta} K$ for some K. Since we know that $M \longrightarrow_{\beta} N$ and $M' \sim_{\alpha} M$ we can use the α-β diamond property of Section 2.5 (`lem-βα`), to obtain a term K such that $M' \longrightarrow_{\beta} K$ and $K \sim_{\alpha} N$, so we prove our goal using the `st-alpha` rule.

□

Finally, using this last result we can prove that if there is a sequence of β-reductions from M to N, then there is also a standard reduction between those two terms. The proof is a direct induction on $M \twoheadrightarrow_{\beta} N$:

```
β→st : ∀{M N} -> M →→β N -> M →→st N
```

3.4 Standard Sequences

The next results show the relation between the reduction relations $\twoheadrightarrow_{l}$, $\twoheadrightarrow_{hap}$ and $\twoheadrightarrow_{st}$ with the existence of a standard reduction sequence.

Firstly notice that, since leftmost reductions always involve the reduction of re-

dexes at position 0, then any sequence of leftmost reductions is a standard reduction sequence with lower bound 0.

```
lem-leftmost→seqβst : ∀{M N} -> M →→l N -> seqβ-st M N 0
```

As a corollary of this lemma and the fact that $M \twoheadrightarrow_{hap} N$ implies $M \twoheadrightarrow_{l} N$ (`lem-hap→l`), we obtain that if $M \twoheadrightarrow_{hap} N$, then there is a standard reduction sequence from M to N with lower bound 0:

```
hap→seqβst : ∀{M N} -> M →→hap N -> seqβ-st M N 0
```

The next result about *seqβ-st* will be useful to prove the subsequent lemma.

```
abs-seq : ∀ {x M N n} -> seqβ-st M N n -> seqβ-st (ƛ x M) (ƛ x N) n
```

Proof. We proceed by induction on the definition of *seqβ-st* M N n.

- Case `nil`: We know that *seqβ-st* M M 0 and therefore we construct our goal, *seqβ-st* $(\lambda x M)$ $(\lambda x M)$ 0, using `nil`.
- Case α`-step`: We know that *seqβ-st* M K n for some K such that $K \sim_\alpha N$. From the induction hypothesis we get that *seqβ-st* $(\lambda x M)$ $(\lambda x K)$ n and since $K \sim_\alpha N$ we can easily prove that $\lambda x K \sim_\alpha \lambda x N$. From this we prove the case using the `st-alpha` constructor.
- Case β`-step`: We know that *seqβ-st* M K n for some K such that $K\,\beta\,N\,@\,m$ with $n \leq m$. Similarly to the last case, the induction hypothesis tells us that *seqβ-st* $(\lambda x M)$ $(\lambda x K)$ n and from $K\,\beta\,N\,@\,m$ we can construct $(\lambda x K)\,\beta\,(\lambda x N)\,@\,m$ using rule `abs`. Finally, we prove our goal using constructor β`-step`.

□

As for the $\twoheadrightarrow_{st}$ relation, if $M \twoheadrightarrow_{st} N$ then there is a standard reduction sequence from M to N, which we code in Agda as the existence of a lower bound for a standard reduction sequence:

```
st→seqβst : ∀{M N} -> M →→st N -> Σ ℕ (\n -> seqβ-st M N n)
```

Proof. By induction on the definition of $M \twoheadrightarrow_{st} N$.

- The case `st-var` can be easily proven using lemma `hap→seqβst`: since $M \twoheadrightarrow_{hap} x$, then there is a standard reduction sequence (with lower bound 0) from M to x.
- Similarly, the case `st-abs` also relies in this lemma, and the induction hypothesis: we know from `hap→seqβst` that there is a standard reduction sequence with lower bound 0 from M to $\lambda x.A$; the induction hypothesis tells us that there exists a natural number n such that there is a reduction sequence from A to B with lower bound n. Therefore, using lemma `abs-seq` we conclude that there must be a standard reduction sequence from M to B with lower bound n since $0 \leq n$.
- The case for `st-app` is slightly trickier since the lower bound that exists depends on certain characteristics of the terms involved: If $M \twoheadrightarrow_{st} N$ was constructed using the constructor `st-app`, that means that for some terms A, B , C and D: (1) $M \twoheadrightarrow_{hap} (A\ B)$, (2) $A \twoheadrightarrow_{st} C$ and (3) $B \twoheadrightarrow_{st} D$. We need to prove that there is a standard reduction sequence from M to $(C\ D)$. Using the induction hypotheses, let m and n be the lower bounds for the standard reduction sequences from A to C and from B to D respectively:

  1. If C is not an abstraction, and $B \sim_\alpha D$[1], then the lower bound for the standard reduction sequence will be m.
  2. If C is not an abstraction, and $B \not\sim_\alpha D$, then the lower bound for the standard reduction sequence will be $n +$ `countRedexes` C.
  3. If C is an abstraction, and $B \not\sim_\alpha D$, then the lower bound for the standard reduction sequence will be $suc\,(n +$ `countRedexes` $C\,)$.
  4. If C is an abstraction, and $B \sim_\alpha D$, then we need to do some further case analysis using the following lemma:

     ```
     lem-seq-appACBC-abs : ∀ {A C B n} -> seqβ-st A C n -> isAbs C
       -> (seqβ-st (A · B) (C · B) n) ∨ (seqβ-st (A · B) (C · B) (suc n))
     ```

     Note that if a reduction sequence ends in an abstraction, by appending the same application (or an α-equivalent one) to all of the terms in the sequence, the lower bound will remain the same if and only the abstraction is generated in the last beta step of the sequence and therefore does not affect the redex count in β-reductions. However, if the abstraction appears

[1] This is a possible scenario, since $\twoheadrightarrow_{st}$ includes $\sim_\alpha$.

before that, then the lower bound of the reduction sequence must be increased by one, since a new redex at position 0 is formed. From this lemma we conclude that the lower bound must be either m or $suc\ m$ for this case.

- Finally, for the `st-alpha` case, we have that $M \twoheadrightarrow_{st} N'$ and $N' \sim_\alpha N$ and we want to prove the existence of a standard reduction sequence from M to N. The induction hypothesis gives us a standard reduction sequence from M to N' and we can directly perform an alpha step to N by using the α`-step` constructor from `seq`β`-st`.

□

The Standardization theorem finally follows from this last result and lemma `β→st` that states that $M \twoheadrightarrow_\beta N$ implies $M \twoheadrightarrow_{st} N$:

```
standardization : ∀{M N} -> M →→β N -> Σ ℕ (λ n -> seqβ-st M N n)
standardization M→→βN = st→seqβst (β→st M→→βN)
```

An interesting observation about this proof, not mentioned by Kashima, is that the relations $\twoheadrightarrow_{st}$ and `seq`β`-st` are actually equivalent. Despite the fact that only one of the directions of the implication is needed, that is `st→seqβst`, we have proven the other direction too by using lemma `st-β→st` as follows:

```
seqβ-st→st : ∀{M N n} -> seqβ-st M N n -> M →→st N
```

Proof. By induction on the definition of *seq*β*-st* $M\ N\ n$.

- Case `nil`: we have that *seq*β*-st* $M\ M\ 0$ and $M \twoheadrightarrow_{st} M$ follows by the reflexivity of $\twoheadrightarrow_{st}$.
- Case α`-step`: we have that *seq*β*-st* $M\ N'\ n$ and $N' \sim_\alpha N$. Form the induction hypothesis we get that $M \twoheadrightarrow_{st} N'$ and we can conclude that $M \twoheadrightarrow_{st} N$ using rule `st-alpha` with $N' \sim_\alpha N$.
- Case β`-step`: we have that *seq*β*-st* $M\ N'\ n$ and $N' \longrightarrow_\beta N$. From the induction hypothesis, we obtain that $M \twoheadrightarrow_{st} N'$. Using this and $N' \longrightarrow_\beta N$ we conclude our thesis using lemma `st-β→st`.

□

Chapter 4

The Leftmost Reduction Theorem

Besides the intrinsic relevance of the Standardization theorem for the metatheory of the Lambda Calculus, several well-known results about reduction strategies in the Lambda Calculus can be proven by using this property. For example, in [15] Plotkin relates a calculus (an equational theory with a rewrite relation) with a programming language using the Standardization theorem by having the programming language implement a standard strategy associated to the calculus. In the Lambda Calculus, it is possible to describe such a reduction strategy that produces standard derivations only, namely the leftmost-outermost strategy[1], and for which the normalization property holds as a corollary of the Standardization theorem: If a term M has a normal form, then the leftmost-outermost reduction strategy will find this normal form. Given a term that has a normal form, this strategy involves a finite number of steps, i.e. an algorithm, that can be used to decide equality between two terms. In general, we can say that this algorithm semi-decides equality since non-normalizing terms would result in infinite reduction sequences. However, this is the best we can achieve since equality is undecidable for the untyped Lambda Calculus[2].

In the present chapter we show how this property can be derived from standardization. It is worth noticing that this proof was developed as part of the present work and is not part of Kashima's contribution. The complete Agda code for this proof can be found in the Appendix.

[1] This reduction strategy contracts in each step the redex at position zero

[2] Note that decidable equality would imply solving the halting problem.

We can characterize a term in normal form as one without redexes using the `countRedexes` function from Chapter 2:

```
nf : Λ -> Set
nf M = countRedexes M ≡ 0
```

And now we can state the aforementioned property as the following lemma:

```
leftmost-nf : ∀{M N} -> M →→β N -> nf N -> M →→l N
```

In order to prove this result, we must first consider some previous lemmas:

The first one states that the number of redexes of two α-equivalent terms is the same:

```
α→sameRedexCount : ∀ {M N} -> M ∼α N -> countRedexes M ≡ countRedexes N
```

The proof of this lemma easily follows by induction on $M \sim_\alpha N$.

The second lemma states that if a term M β-reduces to a term N in normal form, then the contracted redex must be the leftmost redex of M, i.e., the one at position zero:

```
nf→l : ∀{M N n} -> M β N @ n -> nf N -> n ≡ 0
```

Proof. We proceed by induction on $M \,\beta\, N \,@\, n$

- Case `outer-redex`: we have that $(\lambda x A)\, B \,\beta\, B[x := A] \,@\, 0$. Our goal follows directly since rule `outer-redex` contracts the redex at position 0.
- Case `appNoAbsL`: we have that $(A\, C) \,\beta\, (B\, C) \,@\, n$ where $A \,\beta\, B \,@\, n$, A is not an abstraction and $(B\ C)$ is in normal form. From this, we know that `countRedexes` B + `countRedexes` $C \equiv 0$, and therefore `countRedexes` $B \equiv$ `0` which allows us to use the induction hypothesis with $A \,\beta\, B \,@\, n$ and conclude that $n \equiv 0$.

- Case `appAbsL`: we have that $(A\,C)\,\beta\,(B\,C)\,@\,(suc\,n)$ where $A\,\beta\,B\,@\,n$, A is an abstraction and $(B\ C)$ is in normal form. Since $(B\ C)$ is in normal form B cannot be an abstraction, but this is a contradiction since $A\,\beta\,B\,@\,n$ and A is an abstraction because contracting a redex in an abstraction always results in an abstraction (rule `abs`).
- Case `appNoAbsR`: we have that $(C\,A)\,\beta\,(C\,B)\,@\,(\texttt{countRedexes}\,C + n)$ where $A\,\beta\,B\,@\,n$, C is not an abstraction and $(C\ B)$ is in normal form. From this we know that $\texttt{countRedexes}\ C \equiv 0$ and $\texttt{countRedexes}\ B \equiv 0$. From the induction hypothesis we have that $n \equiv 0$, and since $\texttt{countRedexes}\ C \equiv 0$, $n + \texttt{countRedexes}\ C \equiv 0$.
- Case `appAbsR`: we have that $(C\,A)\,\beta\,(C\,B)\,@\,suc\,(\texttt{countRedexes}\,C + n)$ where $A\,\beta\,B\,@\,n$, C is an abstraction and $(C\ B)$ is in normal form. However, this is a contradiction since $(C\ B)$ cannot be in normal form if C is an abstraction.
- Case `abs`: we have that $\lambda x A\ \beta\ \lambda x B\ @\ n$ where $A\ \beta\ B\ @\ n$ and $\lambda x B$ is in normal form. From this we know that B must be in normal form too and we can call the induction hypothesis for $A\ \beta\ B\ @\ n$, concluding that $n \equiv 0$.

□

Finally, notice that a standard sequence with lower bound 0 must be a sequence of leftmost reductions, since all of the β-steps must involve the contraction of the redex at position 0, i.e. a leftmost reduction.

```
seqβ0→l : ∀ {A B} -> seqβ-st A B 0 -> A →→l B
```

The proof follows by a direct induction on *seqβ-st A B 0*.

Lets now turn our attention to the main lemma:

```
seqst→l : ∀{M N n} -> seqβ-st M N n -> nf N -> M →→l N
```

Proof. We proceed by induction on the definition of *seqβ-st M N n*.

- Case `nil`: We have that *seqβ-st A A 0* and we need to prove that $A \twoheadrightarrow_l A$, which follows by constructor `refl`.

- Case α-`step`: We have that *seqβ-st A B′ n* with $B' \sim_\alpha B$ and `nf` B. From this, we have that `countRedexes` $B' \equiv 0$ by lemma `α→sameRedexCount` and therefore, using the induction hypothesis we obtain `A` $\twoheadrightarrow_l$ `B'`. Finally, we construct our goal using rule α-`step`.
- Case β-`step`: We have *seqβ-st A B′ n* and $B' \,\beta\, B \,@\, n'$, where $n \leq n'$ and `countRedexes` $B \equiv 0$. Using lemma `nf→l` we have that $n' \equiv 0$, and so $n \equiv 0$. We then apply lemma `seqβ0→l` to *seqβ-st A B′* 0 and get that $A \twoheadrightarrow_l B'$. Note that since $n' \equiv 0$, we also have that $B' \longrightarrow_l B$. Finally, from $A \twoheadrightarrow_l B'$ and $B' \longrightarrow_l B$ we conclude that $A \twoheadrightarrow_l B$ using rule `append`.

□

Finally, if we have that $M \twoheadrightarrow_\beta N$, the Standardization theorem lets us conclude that there exists a standard reduction sequence from M to N. Therefore, the desired property follows directly combining this result and the previous lemma:

```
leftmost-nf : ∀{M N} -> M →→β N -> nf N -> M →→l N
leftmost-nf M→→βN crN≡0 = seqst→l (proj2 (standardization M→→βN)) crN≡0
```

Chapter 5

Conclusions

In this work we have extended some metatheoretical results from [3] by formalizing a proof of the Standardization theorem in Lambda Calculus using Constructive Type Theory. We use concrete syntax for λ-terms and the notion of multiple substitution. The latter enables us to proceed by structural induction only, producing proofs that are easy to follow, yet fully formal. This is novel in relation to previous efforts in the literature which use de Brujin indexes to implement the syntax, and require the induction to be done on the size of the terms. This work has also served to showcase the usefulness of the library produced in [6] and its suitability for the formalization of other metatheoretical properties of the Lambda Calculus. It is worth highlighting that the definitions and lemmas used to handle syntax and substitutions did not need to be modified or extended in any way. Thus, we conclude that the library represents a useful framework for reasoning with concrete syntax of Lambda Terms that would lend itself well for use in further formalizations.

In addition, we have managed to extend some of the results presented by Kashima in [5] to prove that:

1. The given definition of β-reduction is equivalent to the classical definition.
2. We introduced an inductive definition of a standard reduction sequence, namely `seq`β`-st`, which we proved equivalent to the author's notion of $\twoheadrightarrow_{st}$.
3. The leftmost-outermost reduction strategy is complete for normalizing terms.

Other efforts to formalize Kashima's proof in the literature include one by Guidi in Matita [12] and another one by Vyšniauskas and Emerich in Coq [14]. However, what sets our development apart from these efforts is the use of a concrete syntax and our definition of multiple substitution. While this allows us to prove our lemmas using a clean structural inductive argument, they use a nameless syntax based on de Brujin indexes which results in some inductions being done on the size of the λ-terms. Another effort worth highlighting is that of McKinna and Pollack [16] who formalized a proof of the Standardization theorem due to Takahashi [9] using the LEGO proof assistant [17]. Takahashi's proof basically follows Barendregt's argument, proving that head reductions can be systematically carried out before internal reductions. The main difference lies in the way in which this result is approached. Takahashi uses the inductive definition of parallel β-reduction, a key notion of Tait and Martin-Löf's proof of the Church-Rosser theorem for β-reduction, which consists in performing the contraction of a set of redexes in a λ-term simultaneously. Thus it has an effect similar to a complete development, but can easily be defined by induction on the structure of λ-terms.

In addition to proving the Standardization theorem, Kashima proves a few other interesting results which could be a good follow up to the present work, e.g. the *quasi-leftmost reduction theorem*. An infinite β-reduction sequence is called quasi-leftmost if it contains infinitely many leftmost reduction steps $\longrightarrow_l$. As a corollary of the Standardization theorem it can be proved that if M has a β-normal form, then there is no infinite quasi-leftmost β-reduction sequence from M. Representing infinite reduction sequences in type theory was left outside of the scope of the present work.

Bibliographical References

[1] P. Martin-Löf, "Intuitionistic type theory," ser. Studies in Proof Theory. Lecture Notes, vol. 1. Bibliopolis, Naples, 1984, pp. 91–98.

[2] A. Stoughton, "Substitution revisited," vol. 59, 1988, pp. 317–325.

[3] E. Copello, N. Szasz, and A. Tasistro, "Formal metatheory of the lambda calculus using Stoughton's substitution," *Theoretical Computer Science*, vol. 685, pp. 65 – 82, 2017. [Online]. Available: https://github.com/ernius/formalmetatheory-stoughton

[4] 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.

[5] R. Kashima, "A proof of the standardization theorem in lambda-calculus," Tokyo Institute of Technology. Department of Information Sciences, Tech. Rep., 2000. [Online]. Available: http://www.is.titech.ac.jp/~kashima/pub/C-145.pdf

[6] E. Copello, "Agda library for Formal metatheory of the lambda calculus using Stoughton's substitution." [Online]. Available: https://github.com/ernius/formalmetatheory-stoughton

[7] H. B. Curry and R. Feys, "Combinatory logic, volume i." North-Holland, 1958, pp. xvi+417, second printing 1968.

[8] H. Barendregt, "The lambda calculus its syntax and semantics," ser. Studies in Logic and the Foundations of Mathematics, vol. 103. North Holland, 1984.

[9] M. Takahashi, "Parallel Reductions in λ-Calculus," *Information and Computation*, vol. 118, no. 1, pp. 120 – 127, 1995. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0890540185710577

[10] J. W. Klop, "Combinatory reduction systems." Mathematisch Centrum, 1980, pp. 120–125.

[11] A. Asperti, W. Ricciotti, C. Sacerdoti Coen, and E. Tassi, "The matita interactive theorem prover," in *Automated Deduction*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 64–69.

[12] F. Guidi, "Standardization and confluence in pure lambda-calculus formalized for the Matita theorem prover," *Journal of Formalized Reasoning*, vol. 5, no. 1, pp. 1–25, 2012. [Online]. Available: https://jfr.unibo.it/article/view/3392

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

[14] J. Emerich and I. Vyšniauskas, "Coq formalisation of postponement and standardization theorems in the untyped lambda-calculus," 2014, iLLC, Universiteit van Amsterdam. [Online]. Available: https://github.com/knuton/la-girafe-sportive

[15] G. Plotkin, "Call-by-name, call-by-value and the λ-calculus," *Theoretical Computer Science*, vol. 1, no. 2, pp. 125 – 159, 1975. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0304397575900171

[16] J. McKinna and R. Pollack, "Some lambda calculus and type theory formalized," *Journal of Automated Reasoning*, vol. 23, no. 3, pp. 373–409, Nov 1999. [Online]. Available: https://doi.org/10.1023/A:1006294005493

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

Appendix A

Agda Code

```
module Standardization where

-- All these files can be found at
https://github.com/ernius/formalmetatheory-stoughton
open import Term
open import Chi
open import Substitution
open import Alpha
open import SubstitutionLemmas

open import Data.Nat.Properties
open import Data.Vec
open import Data.List
open import Data.Empty
open import Relation.Nullary
open import Relation.Binary hiding (Rel)
open import Relation.Binary.PropositionalEquality as PropEq renaming
(trans to ≡-trans)
open import Data.Product renaming (Σ to Σx)
open import Data.Nat as Nat hiding (_*_)

infixl 1 _⟶β_
infixl 1 _⟶hap_
```

```
infixl 1 _→→hap_
infix 1 _→→st_

infix 0 _[_:=_]

_[_:=_] : Λ -> V -> Λ -> Λ
M [ x := N ] = M • (ι ≺+ (x , N))

-- ----------------------
-- Natural Properties
-- ----------------------

≤-trans : ∀{m n k} -> m ≤ n -> n ≤ k -> m ≤ k
≤-trans z≤n y = z≤n
≤-trans (s≤s x) (s≤s y) = s≤s (≤-trans x y)

≤-refl : ∀{x} -> x ≤ x
≤-refl = ≤'->≤ ≤'-refl

lem-sum-ab : ∀{a b c d} -> a ≡ b -> c ≡ d -> a + c ≡ b + d
lem-sum-ab refl refl = refl

≤-sum-r : ∀ {x y z} -> x ≤ y -> x ≤ y + z
≤-sum-r z≤n = z≤n
≤-sum-r (s≤s x) = s≤s (≤-sum-r x)

≤-sum-l : ∀ {x y z} -> x ≤ y -> x ≤ z + y
≤-sum-l {x} {y} {z} x≤y = subst2 _≤_ refl (+-comm y z) (≤-sum-r {z = z}
x≤y)

≤-suc : ∀{x y} -> x ≤ y -> x ≤ suc y
≤-suc z≤n = z≤n
≤-suc (s≤s x≤y) = s≤s (≤-suc x≤y)

lem-≤-both : ∀{a b c} -> a ≤ b -> a + c ≤ b + c
lem-≤-both {b = b} z≤n = ≤-sum-l {z = b} ≤-refl
lem-≤-both (s≤s x) = s≤s (lem-≤-both x)
```

```
lem-sum-l : ∀{a b} -> a + b ≡ 0 -> a ≡ 0
lem-sum-l {zero} {b} zero+b≡0 = refl
lem-sum-l {suc _} ()

lem-sum-r : ∀{a b} -> a + b ≡ 0 -> b ≡ 0
lem-sum-r {zero} {b} zero+b≡0 = zero+b≡0
lem-sum-r {suc _} ()

lem-sum-zero : ∀{a b} -> a ≡ 0 -> b ≡ 0 -> a + b ≡ 0
lem-sum-zero refl refl = refl

data _∨_ (P Q : Set) : Set where
   ∨-intro1 : P → P ∨ Q
   ∨-intro2 : Q → P ∨ Q

data _∧_ (P Q : Set) : Set where
   ∧-intro : P → Q → P ∧ Q

∧-elim-l : ∀{P Q} -> P ∧ Q -> P
∧-elim-l (∧-intro x y) = x

∧-elim-r : ∀{P Q} -> P ∧ Q -> Q
∧-elim-r (∧-intro x y) = y

subst3 : ∀ {a b c p} {A : Set a} {B : Set b} {C : Set c} (P : A → B → C →
Set p)
         {x1 x2 z1 y1 y2 z2} → x1 ≡ x2 → y1 ≡ y2 → z1 ≡ z2 -> P x1 y1 z1
→ P x2 y2 z2
subst3 P refl refl refl p = p

≤-sum : ∀{x y z} -> y ≤ z -> y + x ≤ z + x
≤-sum {x = zero} z≤n = z≤n
≤-sum {x = (suc m)} {z = z} z≤n = ≤-sum-l {z = z} ≤-refl
≤-sum (s≤s x1) = s≤s (≤-sum x1)

lem≤0 : ∀{a} -> a ≤ 0 -> a ≡ 0
```

```
lem≤0 {zero} _ = refl
lem≤0 {suc _} ()

lem+≡ : ∀ {x y z w} -> x ≡ y -> z ≡ w -> x + z ≡ y + w
lem+≡ refl refl = refl


-- ---------------------------------------------------------
-- 2 PRELIMINARIES
-- ---------------------------------------------------------

αapp→≡ : ∀{M N M' N'} -> M · N ∼α M' · N' -> (M ∼α M') ∧ (N ∼α N')
αapp→≡ (∼· M∼M' N∼N') = ∧-intro M∼M' N∼N'

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

α-sides : ∀{M N M' N'} -> M ∼α N -> M ∼α M' -> N ∼α N' -> M' ∼α N'
α-sides MαN MαM' NαN' = ∼τ (∼τ (∼σ MαM') (MαN)) NαN'

freshness-subst-rest : ∀{A B x y σ} -> x # A • σ -> x # B -> x #⌊ (σ ≺+ (y
: B) , A)
freshness-subst-rest {A} {B} {x} {y} {σ} x#Aσ x#B w w*M with y ≟ w
... | yes _ = x#B
... | no _ = (lemma#→free# x#Aσ) w w*M

freshness-subst : ∀{A B x y σ} -> x # A • σ -> x # B -> x # A • σ ≺+ (y :
B)
freshness-subst {A} {y = y} {σ = σ} x#Aσ x#B = lemmafree#→# {M = A}
(freshness-subst-rest {y = y} {σ = σ} x#Aσ x#B)

lem#ι : ∀{A x} -> x # A -> x # A • ι
lem#ι {A} {x} x#A = lemmafree#→# lem#ι-rest
      where lem#ι-rest : x #⌊ (ι , A)
            lem#ι-rest m m*A with x ≟ m
            ... | yes x≡m = ⊥-elim ((lemma-free→¬# m*A) (subst2 _#_ x≡m
refl x#A))
```

```
            ... | no x≠m = #v (sym≠ x≠m)

_∼α?_ : ∀ A B -> Dec (A ∼α B)
v x ∼α? v x1 with x ≟ x1
... | yes x≡x1 = yes (subst2 _∼α_ refl (cong v x≡x1) (∼v {x}))
... | no  x≠x1 = no (λ x∼x1 -> x≠x1 (αv→≡ x∼x1))
          where αv→≡ : ∀{x y} -> v x ∼α v y -> x ≡ y
                αv→≡ ∼v = refl
v x ∼α? (y · y1) = no (λ ())
v x ∼α? ƛ x1 y = no (λ ())
(M · N) ∼α? v x = no (λ ())
(M · N) ∼α? (M' · N') with M ∼α? M' | N ∼α? N'
... | yes M∼M' | yes N∼N' = yes (∼· M∼M' N∼N')
... | yes M∼M' | no ¬N∼N' = no (λ MN∼M'N' -> (¬N∼N' (∧-elim-r (αapp→≡
MN∼M'N'))))
... | no ¬M∼M' | yes N∼N' = no (λ MN∼M'N' -> (¬M∼M' (∧-elim-l (αapp→≡
MN∼M'N'))))
... | no ¬M∼M' | no ¬N∼N' = no (λ MN∼M'N' -> (¬N∼N' (∧-elim-r (αapp→≡
MN∼M'N'))))
(M · N) ∼α? ƛ x A = no (λ ())
ƛ x M ∼α? v y = no (λ ())
ƛ x M ∼α? (M' · N') = no (λ ())
ƛ x M ∼α? ƛ y M' with M ∼α? (M' • (ι ≺+ (y , v x))) | x #? ƛ y M'
... | yes M∼M'yx | yes x#λyM'  = yes (∼ƛ #ƛ≡ x#λyM' (∼τ (subst2 _∼α_ (sym
(lemmaMι≺+x,x {x} {M})) refl (∼σ lemma•ι)) M∼M'yx))
... | yes M∼M'yx | no ¬x#λyM'  = no (λ M∼M' -> ⊥-elim (¬x#λyM' (lemmaM∼N#
M∼M' x #ƛ≡)))
... | no ¬M∼M'yx | yes x#λyM' = no (λ { (∼ƛ {y = z} z#λxM z#λyM' Mxz∼M'xz)
 -> ⊥-elim (¬M∼M'yx (α-sides (≡→α (lemmaM∼M'→Mσ≡M'σ {σ = ι ≺+ (z , v x)}
Mxz∼M'xz)) (∼τ (≡→α (lemma≺+ι z#λxM)) (∼σ lemma•ι)) (≡→α (sym (lemma≺+
z#λyM')))))})
... | no ¬M∼M'yx | no ¬x#λyM' = no (λ M∼M' -> ⊥-elim (¬x#λyM' (lemmaM∼N#
M∼M' x #ƛ≡)))

data α-star (⇝ : Λ -> Λ -> Set) : Λ -> Λ -> Set where
    refl : ∀{M} → α-star ⇝ M M
    α-step : ∀{M N N'} → α-star ⇝ M N' → N' ∼α N → α-star ⇝ M N
```

```
    append : ∀ {M N K} → α-star ⇝ M K → ⇝ K N → α-star ⇝ M N

α-star-singl : ∀{⇝ M N} -> ⇝ M N -> α-star ⇝ M N
α-star-singl = append refl

α-star-trans : ∀{⇝ M N K} -> α-star ⇝ M K -> α-star ⇝ K N -> α-star ⇝ M N
α-star-trans M→→K refl = M→→K
α-star-trans M→→K (α-step K→→N' N'~N) = α-step (α-star-trans M→→K K→→N')
N'~N
α-star-trans M→→K (append K→→N' N'→N) = append (α-star-trans M→→K K→→N')
N'→N

data isAbs : Λ -> Set where
  abs : forall {x M} -> isAbs (ƛ x M)

isAbs? : (M : Λ) -> Dec (isAbs M)
isAbs? (v x) = no (λ ())
isAbs? (M · N) = no (λ ())
isAbs? (ƛ x M) = yes abs

absι→ : ∀{A} -> isAbs A -> isAbs (A • ι)
absι→ {v x} isAbsA = isAbsA
absι→ {A · A1} ()
absι→ {ƛ x A} isAbsA = abs

absι<- : ∀{A} -> isAbs (A • ι) -> isAbs A
absι<- {v x} isAbsA = isAbsA
absι<- {A · A1} ()
absι<- {ƛ x A} isAbsA = abs

noAbsα : ∀ {A B} -> ¬ isAbs A -> B ∼α A -> ¬ isAbs B
noAbsα ¬isAbsx ∼v = ¬isAbsx
noAbsα ¬isAbsAB (∼· y y1) = λ ()
noAbsα ¬isAbsA (∼ƛ x2 x3 y1) = λ _ → ¬isAbsA abs

¬isAbsAB : ∀ {M N} -> ¬ isAbs (M · N)
¬isAbsAB = λ ()
```

```
isAbsα : ∀ {A B} -> isAbs A -> B ∼α A -> isAbs B
isAbsα isAbsx ∼v = isAbsx
isAbsα isAbsAB (∼· y y1) = ⊥-elim (¬isAbsAB isAbsAB)
isAbsα x1 (∼ƛ x2 x3 y1) = abs

-- A substitution where each variable is only changed for another variable.
onlyVars : Σ -> Set
onlyVars σ = (x : V) → Σx V (λ y -> σ x ≡ v y)

ιonlyVars : onlyVars ι
ιonlyVars = λ x → x : refl

onlyVars-append : ∀{σ x y} -> onlyVars σ -> onlyVars (σ ≺+ (x , v y))
onlyVars-append {σ} {x}{y} onlyVarsσ x' with x ≟ x'
... | yes x≡x' = y : refl
... | no  x≠x' = onlyVarsσ x'

onlyVars-¬isAbs : ∀{M σ} -> onlyVars σ -> ¬ (isAbs M) -> ¬ (isAbs (M • σ))
onlyVars-¬isAbs {v x} {σ} onlyVarsσ ¬isAbsM with onlyVarsσ x
... | ( y , σx≡v ) = (λ isAbsσx -> (⊥-elim (¬isAbsy (subst isAbs σx≡v
isAbsσx))))
                  where ¬isAbsy : ¬ (isAbs (v y))
                        ¬isAbsy ()
onlyVars-¬isAbs {M · M1} {σ} onlyVarsσ ¬isAbsM = λ ()
onlyVars-¬isAbs {ƛ x M} {σ} onlyVarsσ ¬isAbsM = λ _ → ¬isAbsM abs

onlyVars-isAbs : ∀{M σ} -> onlyVars σ -> isAbs M -> isAbs (M • σ)
onlyVars-isAbs {v x} {σ} onlyVarsσ isAbsM with onlyVarsσ x
... | ( y , σx≡v ) = ⊥-elim (¬isAbsx isAbsM)
                 where ¬isAbsx : ¬ (isAbs (v x))
                       ¬isAbsx ()
onlyVars-isAbs {M · M1} {σ} onlyVarsσ isAbsM = ⊥-elim (¬isAbsM isAbsM)
                 where ¬isAbsM : ¬ (isAbs (M · M1))
                       ¬isAbsM ()
onlyVars-isAbs {ƛ x M} {σ} onlyVarsσ isAbsM = abs
```

```
-- Counts the number of redexes in a Term
countRedexes : Λ -> ℕ
countRedexes (v _) = 0
countRedexes (M · N) with isAbs? M
... | yes _ = suc (countRedexes M + countRedexes N)
... | no _ = countRedexes M + countRedexes N
countRedexes (ƛ _ M) = countRedexes M

-- Definition 2.1 For λ-terms M, N and a natural number n ≥ 0, we define a
relation M →n N inductively as follows.
-- M β N @ n represents that N is obtained by contracting the n-th redex
in M.
data _β_@_ : Λ -> Λ -> ℕ -> Set where
  outer-redex : ∀ {x A B} -> ((ƛ x A) · B) β (A [ x := B ]) @ 0
  appNoAbsL : ∀ {n A B C} -> A β B @ n -> ¬ isAbs A
    -> (A · C) β (B · C) @ n
  appAbsL : ∀ {n A B C} -> A β B @ n -> isAbs A
    -> (A · C) β (B · C) @ (suc n)
  appNoAbsR : ∀ {n A B C} -> A β B @ n -> ¬ isAbs C
    -> (C · A) β (C · B) @ (n + countRedexes C)
  appAbsR : ∀ {n A B C} -> A β B @ n -> isAbs C
    -> (C · A) β (C · B) @ (suc (n + countRedexes C))
  abs : ∀ {n x A B} -> A β B @ n -> (ƛ x A) β (ƛ x B) @ n

β-example1 : ∀{x y z} -> ((ƛ x ((ƛ y (v y)) · (v x))) · (v z)) β ((ƛ x ((v
y) [ y := v x ])) · (v z)) @ 1
β-example1 = appAbsL (abs outer-redex) abs

-- Definition 2.2
-- Beta reduction
open import Relation Λ public
_⟶β_ : Λ -> Λ -> Set
M ⟶β N = Σx ℕ (\n -> M β N @ n)

_↠β_ : Λ -> Λ -> Set
_↠β_ = α-star _⟶β_
```

```
data _⟶βcl_ : Λ -> Λ -> Set where
  redex : ∀ {x M N} -> ((ƛ x M) · N) ⟶βcl (M [ x := N ])
  app-l : ∀ {M M' N} -> M ⟶βcl M' -> (M · N) ⟶βcl (M' · N)
  app-r : ∀ {N N' M} -> N ⟶βcl N' -> (M · N) ⟶βcl (M · N')
  abs : ∀ {x M M'} -> M ⟶βcl M' -> (ƛ x M) ⟶βcl (ƛ x M')

β@→β : ∀{M N n} -> M β N @ n -> M ⟶β N
β@→β {n = n} MβNn = n : MβNn

β@-implies-βcl : ∀{M N} -> M ⟶β N -> M ⟶βcl N
β@-implies-βcl (_ , outer-redex) = redex
β@-implies-βcl (_ , appNoAbsL AβBn _) = app-l (β@-implies-βcl (β@→β AβBn))
β@-implies-βcl (_ , appAbsL AβBn _) = app-l (β@-implies-βcl (β@→β AβBn))
β@-implies-βcl (_ , appNoAbsR AβBn _) = app-r (β@-implies-βcl (β@→β AβBn))
β@-implies-βcl (_ , appAbsR AβBn _) = app-r (β@-implies-βcl (β@→β AβBn))
β@-implies-βcl (_ , abs AβBn) = abs (β@-implies-βcl (β@→β AβBn))

βcl-implies-β@ : ∀{M N} -> M ⟶βcl N -> M ⟶β N
βcl-implies-β@ redex = zero : outer-redex
βcl-implies-β@ (app-l {M} {M'} {N} M→βM') with isAbs? M
... | yes isAbsM = β@→β (appAbsL (proj2 (βcl-implies-β@ M→βM')) isAbsM)
... | no ¬isAbsM = β@→β (appNoAbsL (proj2 (βcl-implies-β@ M→βM')) ¬isAbsM)
βcl-implies-β@ (app-r {M} {M'} {N} M→βM') with isAbs? N
... | yes isAbsN = β@→β (appAbsR (proj2 (βcl-implies-β@ M→βM')) isAbsN)
... | no ¬isAbsN = β@→β (appNoAbsR (proj2 (βcl-implies-β@ M→βM')) ¬isAbsN)
βcl-implies-β@ (abs M→βM') =  β@→β (abs (proj2 (βcl-implies-β@ M→βM')))

lem-βex : ∀{A B n} -> A β B @ n -> A ⟶β B
lem-βex {A} {B} {n} AβBn = (n , AβBn)

lem-βappr : ∀{A B C} -> A ⟶β B -> A · C ⟶β B · C
lem-βappr (.0 : outer-redex) = zero : appNoAbsL outer-redex (λ ())
lem-βappr (n : appNoAbsL AβBn x) = n : appNoAbsL (appNoAbsL AβBn x) (λ ())
lem-βappr (n : appAbsL AβBn x) = lem-βex (appNoAbsL (appAbsL AβBn x) (λ
()))
lem-βappr (n : appNoAbsR AβBn x) = lem-βex (appNoAbsL (appNoAbsR AβBn x)
(λ ()))
```

```
lem-βappr (n : appAbsR AβBn x) = lem-βex (appNoAbsL (appAbsR AβBn x) (λ
()))
lem-βappr (n : abs AβBn) = suc n : appAbsL (abs AβBn) abs

lem-βappl : ∀{A B C} -> A ⟶β B -> C · A ⟶β C · B
lem-βappl {C = C} (m , AβBn) with isAbs? C
... | yes isAbsC = lem-βex (appAbsR AβBn isAbsC)
... | no ¬isAbsC = lem-βex (appNoAbsR AβBn ¬isAbsC)



freshness-β : ∀{A B x} -> A ⟶β B -> x # A -> x # B
freshness-β {(ƛ w A) · N} {B} {x} (.0 : outer-redex) (#· #ƛ≡ w#N) = w#Aw,N
{A}
            where lemw#A : ∀{w x y σ A} -> w # (A • σ) -> w ≠ y -> w # (A
• σ ≺+ (x : (v y)))
                  lemw#A {w} {x} {y} {σ} {A} w#Aσ w≠y = freshness-subst
{A} {v y} {w} {x} w#Aσ (#v (sym≠ w≠y))
                  w#w,N-rest : ∀{A} -> w #⌊ (ι ≺+ (w : N) , A)
                  w#w,N-rest m m*A with w ≟ m
                  ... | yes _ = w#N
                  ... | no w≠m = #v (sym≠ w≠m)
                  w#Aw,N : ∀{A} -> w # A • ι ≺+ (w : N)
                  w#Aw,N {A} = lemmafree#→# {w} {ι ≺+ (w : N)} {A}
w#w,N-rest
freshness-β {(ƛ w A) · N} {B} {y} (.0 : outer-redex) (#· (#ƛ y#A) w#N) =
freshness-subst {A} {y = w} (lem#ι y#A) w#N
freshness-β (_ : appNoAbsL {n} A⟶Bn _) (#· x#C x#A) =  (#· (freshness-β (n
, A⟶Bn) x#C) x#A)
freshness-β (_ : appAbsL {n} A⟶Bn _) (#· x#C x#A) =  (#· (freshness-β (n ,
A⟶Bn) x#C) x#A)
freshness-β (_ : appNoAbsR {n} A⟶Bn _) (#· x#C x#A) =  (#· x#C
(freshness-β (n , A⟶Bn) x#A))
freshness-β (_ : appAbsR {n} A⟶Bn _) (#· x#C x#A) = (#· x#C (freshness-β
(n , A⟶Bn) x#A))
freshness-β {A} {B} {x} (n : abs {x = y} A⟶Bn) (#ƛ x#A) = #ƛ (freshness-β
(n , A⟶Bn) x#A)
freshness-β {A} {B} {x} (n : abs {x = y} A⟶Bn) #ƛ≡ = #ƛ≡
```

```
free-β : ∀{A B x} -> A ⟶β B -> x * B -> x * A
free-β {A} {B} {x} A⟶βB x*B with x #? A
... | yes x#A = ⊥-elim ((lemma-free→¬# x*B) (freshness-β A⟶βB x#A))
... | no ¬x#A = lemma¬#→free ¬x#A

lem-βsubst : ∀{M N σ} -> M ⟶β N -> Σx Λ (λ N' -> (M • σ ⟶β N') ∧ (N' ∼α N
• σ))
lem-βsubst {σ = σ} (.0 : outer-redex {x} {A} {B}) = (((A • (σ ≺+ (x , (v
y)))) [ y := B • σ ]) , ∧-intro (0 , outer-redex) (subst2 _∼α_ refl
(corollary1Prop7 {A} {B} {x = x}) Aσ'Bσ))
                where y = χ (σ , ϰ x A)
                      Aσ'Bσ = corollary1SubstLemma {x}{χ (σ , ϰ x A)}{M =
A}{N = B • σ} (χ-lemma2 σ (ϰ x A))
lem-βsubst {σ = σ} (m : appNoAbsL {C = C} AβBn _) with lem-βsubst {σ = σ}
(m , AβBn)
... | (N' , Aσ→N'∧N'~Bσ) = (N' · (C • σ) , ∧-intro (lem-βappr (∧-elim-l
Aσ→N'∧N'~Bσ)) (∼· (∧-elim-r Aσ→N'∧N'~Bσ) ∼ρ))
lem-βsubst {σ = σ} (m : appAbsL {C = C} AβBn _) with lem-βsubst {σ = σ}
(lem-βex AβBn)
... | (N' , Aσ→N'∧N'~Bσ) = (N' · (C • σ) , ∧-intro (lem-βappr (∧-elim-l
Aσ→N'∧N'~Bσ)) (∼· (∧-elim-r Aσ→N'∧N'~Bσ) ∼ρ))
lem-βsubst {σ = σ} (_ : appNoAbsR {C = C} AβBn _) with lem-βsubst {σ = σ}
(lem-βex AβBn)
... | (N' , Aσ→N'∧N'~Bσ) = ((C • σ) · N' , ∧-intro (lem-βappl (∧-elim-l
Aσ→N'∧N'~Bσ)) (∼· ∼ρ (∧-elim-r Aσ→N'∧N'~Bσ)))
lem-βsubst {σ = σ} (_ : appAbsR {C = C} AβBn _) with lem-βsubst {σ = σ}
(lem-βex AβBn)
... | (N' , Aσ→N'∧N'~Bσ) = ((C • σ) · N' , ∧-intro (lem-βappl (∧-elim-l
Aσ→N'∧N'~Bσ)) (∼· ∼ρ (∧-elim-r Aσ→N'∧N'~Bσ)))
lem-βsubst {σ = σ} (m : abs {x = x} {A = A} {B = B} AβBn) with lem-βsubst
{σ = σ ≺+ (x , (v y))} (lem-βex AβBn)
   where y = χ (σ , ϰ x A)
... | (N' , Aσ→N'∧N'~Bσ) = (ϰ y N' , ∧-intro (lem-βex (abs (proj2
(∧-elim-l Aσ→N'∧N'~Bσ)))) (∼τ (lemma∼λ (∧-elim-r Aσ→N'∧N'~Bσ)) (∼σ
(corollary4-2 {x = x} {M = B} {σ = σ} y#σλxB))))
   where y = χ (σ , ϰ x A)
```

```
          σ' = σ ≺+ (x , (v y))
          y#⌊σ,A : y #⌊ (σ , ϰ x A)
          y#⌊σ,A = χ-lemma2 σ (ϰ x A)
          y#σλxB : y #⌊ (σ , ϰ x B)
          y#σλxB w (*ϰ w*B w!=x) = y#⌊σ,A w (*ϰ (free-β (lem-βex AβBn) w*B)
w!=x)

lem-βα : ∀{M N M'} -> M ⟶β N -> M ∼α M' -> Σx Λ  (λ N' -> (M' ⟶β N') ∧ (N'
∼α N))
lem-βα {M} {N} {M'} (.0 : outer-redex) (∼· {N = B} {N' = B'} (∼ϰ {A} {A'}
{x} {z} {w} w#λxA w#λzA' A∼A') B∼B')
                                                    = ( (A' [ z := B' ]) ,
∧-intro (0 , outer-redex) (∼τ A'z,B'αAxB ∼ρ))
       where Ax,B˜A'z,B : (A [ x := B ]) ∼α (A' [ z := B ])
             Ax,B˜A'z,B = subst2 _∼α_ (sym (lemma≺+ w#λxA)) (sym (lemma≺+
w#λzA')) (≡→α (lemmaM∼M'→Mσ≡M'σ {σ = (ι ≺+ (w , B))} A∼A'))
             z,B'˜z,B : (ι ≺+ (z , B')) ∼ασ (ι ≺+ (z , B))
             z,B'˜z,B y with z ≟ y
             ... | yes  _ = (∼σ B∼B')
             ... | no   _ = ∼ρ
             A'z,B'αAxB : (A' [ z := B' ]) ∼α (A [ x := B ])
             A'z,B'αAxB = ∼τ (lemma-subst {M = A'} ∼ρ (lemma∼ασ
z,B'˜z,B)) (∼σ Ax,B˜A'z,B)
lem-βα (n : appNoAbsL {.n} {M} {N} {C} M⟶Nn ¬isAbsM) (∼· {N' = C'} M∼M'
C∼C') with lem-βα (n , M⟶Nn) M∼M'
... | ( N' , β∧∼) = (N' · C' , ∧-intro (proj1 (∧-elim-l β∧∼) , appNoAbsL
(proj2 (∧-elim-l β∧∼)) (noAbsα ¬isAbsM (∼σ M∼M'))) (∼· (∧-elim-r β∧∼) (∼σ
C∼C')))
lem-βα (n : appAbsL {m} {M} {N} {C} M⟶Nn isAbsM) (∼· {N' = C'} M∼M' C∼C')
with lem-βα (m , M⟶Nn) M∼M'
... | ( N' , β∧∼) = (N' · C' , ∧-intro (suc (proj1 (∧-elim-l β∧∼)) ,
appAbsL (proj2 (∧-elim-l β∧∼)) (isAbsα isAbsM (∼σ M∼M'))) (∼· (∧-elim-r
β∧∼) (∼σ C∼C')))
lem-βα (n : appNoAbsR {m} {M} {N} {C} M⟶Nn ¬isAbsM) (∼· {M' = C'} C∼C'
M∼M') with lem-βα (m , M⟶Nn) M∼M'
... | ( N' , β∧∼) = (C' · N' , ∧-intro (proj1 (∧-elim-l β∧∼) +
countRedexes C' , appNoAbsR (proj2 (∧-elim-l β∧∼)) (noAbsα ¬isAbsM (∼σ
```

```
C∼C’))) (∼· (∼σ C∼C’) (∧-elim-r β∧∼)))
lem-βα (_ : appAbsR {m} {M} {N} {C} M⟶Nn isAbsM) (∼· {M’ = C’} C∼C’ M∼M’)
with lem-βα (m , M⟶Nn) M∼M’
... | ( N’ , β∧∼) = (C’ · N’ , ∧-intro (suc ((proj1 (∧-elim-l β∧∼) +
countRedexes C’)) , appAbsR (proj2 (∧-elim-l β∧∼)) (isAbsα isAbsM (∼σ
C∼C’))) (∼· (∼σ C∼C’) (∧-elim-r β∧∼)))
lem-βα (n : abs {.n} {x} {A} {B} A⟶Bn) (∼ƛ {.A} {A’} {.x} {x’} {y} y#λxA
y#λx’A’ Ax,y~A’x’y) = λx’A’⟶βλxB
       where A’~Ax,x’ : (A [ x := v x’ ]) ∼α A’
             A’~Ax,x’ = subst2 _∼α_ (sym (lemma≺+ y#λxA)) refl (∼τ (∼τ
(≡→α (lemmaM∼M’→Mσ≡M’σ {σ = (ι ≺+ (y , v x’))} Ax,y~A’x’y)) (≡→α (lemma≺+ι
y#λx’A’))) (∼σ lemma•ι))
             Ax,x’⟶βBx,x’ : Σx Λ (\K -> ((A [ x := v x’ ]) ⟶β K) ∧ (K ∼α
(B [ x := (v x’)])))
             Ax,x’⟶βBx,x’ = lem-βsubst (n , A⟶Bn)
             lemα : ∀{A B B’} -> Σx Λ (\K -> (A ⟶β K) ∧ (K ∼α B)) -> B ∼α
B’ -> Σx Λ (\K -> (A ⟶β K) ∧ (K ∼α B’))
             lemα (K , AK∧K~B) B~B’ = (K , ∧-intro (∧-elim-l AK∧K~B) (∼τ
(∧-elim-r AK∧K~B) B~B’))
             A’⟶βBx,x’ : Σx Λ (\K -> ((A’ ⟶β K) ∧ (K ∼α (B [ x := (v
x’)]))))
             A’⟶βBx,x’ = lemα (lem-βα (∧-elim-l (proj2 Ax,x’⟶βBx,x’))
A’~Ax,x’) (∧-elim-r (proj2 Ax,x’⟶βBx,x’))
             AβK→λxAβλxK : ∀ {A B x} -> A ⟶β B -> ƛ x A ⟶β ƛ x B
             AβK→λxAβλxK (n , A⟶Bn )= (n , abs A⟶Bn)
             λx’A’⟶βλx’Bx,x’ : Σx Λ (\K -> ((ƛ x’ A’ ⟶β K) ∧ (K ∼α (ƛ x’
(B [ x := (v x’)])))))
             λx’A’⟶βλx’Bx,x’ = (ƛ x’ (proj1 A’⟶βBx,x’) , ∧-intro
(AβK→λxAβλxK (∧-elim-l (proj2 A’⟶βBx,x’))) (lemma∼λ (∧-elim-r (proj2
A’⟶βBx,x’))))
             λx’Bx,x’~λxB : ƛ x’ (B [ x := (v x’)]) ∼α ƛ x B
             λx’Bx,x’~λxB = ∼σ (corollary4-2’ (freshness-β (n , abs A⟶Bn)
(lemmaM∼N# (∼σ (∼ƛ y#λxA y#λx’A’ Ax,y~A’x’y)) x’ #ƛ≡)))
             λx’A’⟶βλxB : Σx Λ (\K -> ((ƛ x’ A’ ⟶β K) ∧ (K ∼α ƛ x B)))
             λx’A’⟶βλxB = lemα λx’A’⟶βλx’Bx,x’ λx’Bx,x’~λxB

-- A substitution mapping only vars has no effect on the count of redexes.
```

```
lem-λredexCount : ∀{M σ} -> onlyVars σ -> countRedexes (M • σ) ≡
countRedexes M
lem-λredexCount {v x}{σ} onlyVarsσ with onlyVarsσ x
... | ( _ , σx≡v ) = cong countRedexes σx≡v
lem-λredexCount {M · M'}{σ} onlyVarsσ with isAbs? (M • σ) | isAbs? M
... | yes isAbsM | yes isAbsM' = cong suc (lem+≡ (lem-λredexCount {M}
onlyVarsσ) (lem-λredexCount {M'} onlyVarsσ))
... | yes isAbsM | no ¬isAbsM' = ⊥-elim ((onlyVars-¬isAbs onlyVarsσ
¬isAbsM') isAbsM)
... | no ¬isAbsM | yes isAbsM' = ⊥-elim (¬isAbsM (onlyVars-isAbs onlyVarsσ
isAbsM'))
... | no ¬isAbsM | no ¬isAbsM' = lem+≡ (lem-λredexCount {M} onlyVarsσ)
(lem-λredexCount {M'} onlyVarsσ)
lem-λredexCount {ƛ x1 M}{σ} onlyVarsσ = lem-λredexCount {M}
(onlyVars-append {σ} {x1} onlyVarsσ)

β-abs : ∀ {A B n} -> A β B @ n -> ¬ (isAbs B) -> ¬ (isAbs A)
β-abs outer-redex ¬isAbsB = λ ()
β-abs (appNoAbsL x x1) ¬isAbsB = λ ()
β-abs (appAbsL x x1) ¬isAbsB = λ ()
β-abs (appNoAbsR x x1) ¬isAbsB = λ ()
β-abs (appAbsR x x1) ¬isAbsB = λ ()
β-abs (abs x1) ¬isAbsB = λ _ → ¬isAbsB abs

β-countRedexesR : ∀ {M N n} -> M β N @ n -> n ≤ countRedexes N
β-countRedexesR outer-redex = z≤n
β-countRedexesR (appNoAbsL {n} {A} {B} A⟶Bn ¬isAbsA) with isAbs? B
... | yes _ = ≤-suc (≤-sum-r (β-countRedexesR A⟶Bn))
... | no _ = ≤-sum-r (β-countRedexesR A⟶Bn)
β-countRedexesR (appAbsL {n} {A} {B} A⟶Bn isAbsA) with isAbs? B
... | yes _ = s≤s (≤-sum-r (β-countRedexesR A⟶Bn))
... | no ¬isAbsB = ⊥-elim ((β-abs A⟶Bn ¬isAbsB) isAbsA)
β-countRedexesR (appNoAbsR {n} {A} {B} {C} A⟶Bn ¬isAbsA) with isAbs? C
... | yes _ = ≤-suc (subst2 _≤_ refl (+-comm (countRedexes B)
(countRedexes C)) (lem-≤-both {c = countRedexes C} (β-countRedexesR A⟶Bn)))
... | no _ = subst2 _≤_ refl (+-comm (countRedexes B) (countRedexes C))
(lem-≤-both {c = countRedexes C} (β-countRedexesR A⟶Bn))
```

```
β-countRedexesR (appAbsR {n} {A} {B} {C} A⟶Bn isAbsC) with isAbs? C
... | yes _ = s≤s (subst2 _≤_ refl (+-comm (countRedexes B) (countRedexes
C)) (lem-≤-both {c = countRedexes C} (β-countRedexesR A⟶Bn)))
... | no ¬isAbsC = ⊥-elim (¬isAbsC isAbsC)
β-countRedexesR (abs x1) = β-countRedexesR x1

α→sameRedexCount : ∀ {A B} -> A ∼α B -> countRedexes A ≡ countRedexes B
α→sameRedexCount ∼v = refl
α→sameRedexCount (∼· {A} {A'} {B} {B'} A'αA B'αB) with isAbs? A' | isAbs? A
... | yes isAbsA' | yes isAbsA = cong suc (lem+≡ (α→sameRedexCount A'αA)
(α→sameRedexCount B'αB))
... | yes isAbsA' | no ¬isAbsA = ⊥-elim (¬isAbsA (isAbsα isAbsA' A'αA))
... | no ¬isAbsA' | yes isAbsA = ⊥-elim (¬isAbsA' (isAbsα isAbsA (∼σ
(A'αA))))
... | no ¬isAbsA' | no ¬isAbsA = lem+≡ (α→sameRedexCount A'αA)
(α→sameRedexCount B'αB)
α→sameRedexCount (∼ƛ {A} {B} {x} {x'} {y} _ _ A∼αB) = subst2 _≡_
(lem-λredexCount {A} (onlyVars-append {x = x} ιonlyVars)) (lem-λredexCount
{B} (onlyVars-append {x = x'} ιonlyVars)) IH
                where IH = α→sameRedexCount A∼αB

lem-crabs : ∀{x A} -> countRedexes (ƛ x A) ≡ countRedexes A
lem-crabs = refl

isAbsA→isAbsAσ : ∀{A σ} -> isAbs A -> isAbs (A • σ)
isAbsA→isAbsAσ {v x} ()
isAbsA→isAbsAσ {A · A1} ()
isAbsA→isAbsAσ {ƛ x A} isAbsA = abs

isAbsAσ→isAbsA : ∀{A σ} -> onlyVars σ -> isAbs (A • σ) -> isAbs A
isAbsAσ→isAbsA {A} {σ} onlyVarsσ isAbsAσ with isAbs? A
... | yes isAbsA = isAbsA
... | no ¬isAbsA = ⊥-elim ((onlyVars-¬isAbs {σ = σ} onlyVarsσ ¬isAbsA)
isAbsAσ)

crAιx,y≡crA : ∀{σ A} -> onlyVars σ -> countRedexes (A • σ) ≡ countRedexes A
crAιx,y≡crA {σ} {v w} onlyVarsσ with onlyVarsσ w
```

```
... | (x , σw=x) = cong countRedexes σw=x
crAιx,y≡crA {σ} {A · B} onlyVarsσ with isAbs? A | isAbs? (A • σ)
... | yes _ | yes _ = cong suc (lem-sum-ab (crAιx,y≡crA {σ} {A} onlyVarsσ)
(crAιx,y≡crA {σ} {B} onlyVarsσ))
... | no ¬isAbsA | yes isAbsAσ = ⊥-elim (¬isAbsA (isAbsAσ→isAbsA onlyVarsσ
isAbsAσ))
... | yes isAbsA | no ¬isAbsAσ = ⊥-elim (¬isAbsAσ (isAbsA→isAbsAσ isAbsA))
... | no _ | no _ = lem-sum-ab (crAιx,y≡crA {σ} {A} onlyVarsσ)
(crAιx,y≡crA {σ} {B} onlyVarsσ)
crAιx,y≡crA {σ}{ƛ w A} onlyVarsσ = crAιx,y≡crA {A = A} (onlyVars-append
{σ} {w} onlyVarsσ)

lem-crα : ∀{A B} -> A ∼α B -> countRedexes A ≡ countRedexes B
lem-crα ∼v = refl
lem-crα (∼· {A} {A'} {B} {B'} A~B A~B1) with isAbs? A | isAbs? A'
... | yes _ | yes _ = cong suc (lem-sum-ab (lem-crα A~B) (lem-crα A~B1))
... | yes isAbsA | no ¬isAbsA' = ⊥-elim (¬isAbsA' (isAbsα isAbsA (∼σ A~B)))
... | no ¬isAbsA | yes isAbsA' = ⊥-elim (¬isAbsA (isAbsα isAbsA' A~B))
... | no _ | no _ = lem-sum-ab (lem-crα A~B) (lem-crα A~B1)
lem-crα (∼ƛ {A} {B} {x} {x'} y#A y#B Ax,y~Bw,y) = subst2 _≡_ (crAιx,y≡crA
{A = A}(onlyVars-append {x = x} ιonlyVars)) (crAιx,y≡crA {A = B}
(onlyVars-append {x = x'} ιonlyVars)) (lem-crα Ax,y~Bw,y)

-- ---------------------------------------------------
-- 3.1 STANDARD REDUCTION SEQUENCES
-- ---------------------------------------------------

-- Standard sequence from M to N with lower bound n
data seqβ-st (M : Λ) : (N : Λ) -> ℕ -> Set where
  nil : seqβ-st M M 0
  α-step : ∀ {n K N} -> seqβ-st M K n -> K ∼α N -> seqβ-st M N n
  β-step : ∀ {K n n₀ N} -> seqβ-st M K n -> K β N @ n₀ -> n₀ ≥ n ->
seqβ-st M N n₀

seq-redexCount : ∀ {A B n} -> seqβ-st A B n -> n ≤ countRedexes B
seq-redexCount nil = z≤n
seq-redexCount (α-step seqAKn K∼αB) = subst2 _≤_ refl (α→sameRedexCount
```

```
K∼αB) (seq-redexCount seqAKn)
seq-redexCount (β-step seqAKn K→B@m n≤m) = β-countRedexesR K→B@m

-- ------------------------------------------------
-- 3.2 TWO USEFUL REDUCTION RELATIONS
-- ------------------------------------------------

-- Leftmost reduction
_⟶l_ : Λ -> Λ -> Set
M ⟶l N =  M β N @ 0

_→→l_ : Λ -> Λ -> Set
_→→l_ = α-star _⟶l_

-- Head reduction in application
data _⟶hap_ : Λ -> Λ -> Set where
  hap-head : ∀{x A B} -> (ƛ x A) · B ⟶hap (A [ x := B ])
  hap-chain : ∀{C A B} -> A ⟶hap B -> (A · C) ⟶hap (B · C)

_→→hap_ : Λ -> Λ -> Set
_→→hap_ = α-star _⟶hap_

hap-app-r : ∀{M N P} -> M →→hap N -> M · P →→hap N · P
hap-app-r refl = refl
hap-app-r (α-step M→→hapN' N'˜N) = α-step (hap-app-r M→→hapN') (∼· N'˜N ∼ρ)
hap-app-r (append M→→hapK K⟶hapN) = append (hap-app-r M→→hapK) (hap-chain
K⟶hapN)

lem-hap-subst : ∀{σ M N} -> M ⟶hap N ->  Σx Λ (λ N' -> ((M • σ) ⟶hap N') ∧
(N' ∼α (N • σ)))
lem-hap-subst {σ} (hap-head {x} {A} {B}) = ( (A • σ') • (ι ≺+ (y , B • σ))
, ∧-intro λyAσ'·Bσ⟶hapAσ'y,Bσ (subst2 _∼α_ refl (corollary1Prop7 {A} {B}
{x = x}) Aσ'Bσ))
                              where y = χ (σ , ƛ x A)
                                    σ' = σ ≺+ (x : v y)
                                    λyAσ'·Bσ⟶hapAσ'y,Bσ = hap-head {y} {A •
σ'} {B • σ}
```

```
                                    Aσ'Bσ : (A • σ') • (ι ≺+ (y , B • σ))
∼α (A • σ  ≺+ (x , B • σ))
                                    Aσ'Bσ = corollary1SubstLemma {x}{χ (σ ,
ϰ x A)}{M = A}{N = B • σ} (χ-lemma2 σ (ϰ x A))
lem-hap-subst {σ} (hap-chain {C = P} A⟶hapB) with lem-hap-subst A⟶hapB
... | (N' , Aσ−>N'∧N'˜Bσ) = (N' · (P • σ) , ∧-intro (hap-chain (∧-elim-l
Aσ−>N'∧N'˜Bσ)) (∼· (∧-elim-r Aσ−>N'∧N'˜Bσ) ∼ρ))

hap-subst : ∀{M N σ} -> M →→hap N -> (M • σ) →→hap (N • σ)
hap-subst refl = refl
hap-subst (α-step M→→hapN' N'˜N) = α-step (hap-subst M→→hapN') (≡→α
(lemmaM∼M'→Mσ≡M'σ N'˜N))
hap-subst {M} {N} {σ} (append {K = K} M→→hapK K⟶hapN) =
  α-star-trans (hap-subst M→→hapK) (α-step (α-star-singl Kσ⟶N') N'˜Nσ)
                 where N' : Λ
                       N' = proj1 (lem-hap-subst K⟶hapN)
                       Kσ⟶N' : (K • σ) ⟶hap N'
                       Kσ⟶N' = ∧-elim-l (proj2 (lem-hap-subst K⟶hapN))
                       N'˜Nσ : N' ∼α (N • σ)
                       N'˜Nσ = ∧-elim-r (proj2 (lem-hap-subst K⟶hapN))

lem-hap¬Abs : ∀{M N} -> M ⟶hap N -> ¬ isAbs M
lem-hap¬Abs hap-head  ()
lem-hap¬Abs (hap-chain x) ()

lem-hap→l : ∀ {M N} -> M ⟶hap N -> M ⟶l N
lem-hap→l hap-head = outer-redex
lem-hap→l (hap-chain A→hapB) with lem-hap→l A→hapB
... | A⟶B@0 = appNoAbsL A⟶B@0 (lem-hap¬Abs A→hapB)

hap→l : ∀{M N} -> M →→hap N -> M →→l N
hap→l refl = refl
hap→l (α-step M→→hapK K˜N) = α-step (hap→l M→→hapK) K˜N
hap→l (append M→→hapK K⟶hapN) = append (hap→l M→→hapK) (lem-hap→l K⟶hapN)

lem-l-app : ∀ {M N C} -> M ⟶l N -> ¬ isAbs M -> (M · C) ⟶l (N · C)
lem-l-app M⟶N@0 ¬isAbsM = appNoAbsL M⟶N@0 ¬isAbsM
```

```
-- ------------------------------------------------------------------
-- 3.3 STANDARD REDUCTION
-- ------------------------------------------------------------------

data _→→st_ (L : Λ) : Λ -> Set where
  st-var : ∀{x} -> L →→hap (v x) -> L →→st (v x)
  st-app : ∀{A B C D} -> L →→hap (A · B) -> A →→st C -> B →→st D ->
    L →→st (C · D)
  st-abs : ∀{x A B} -> L →→hap (ƛ x A) -> A →→st B -> L →→st (ƛ x B)
  st-alpha : ∀{A' A} -> L →→st A' -> A' ∼α A -> L →→st A

infix 1 _→st_
_→st_ : Σ → Σ → Set
σ →st σ' = (x : V) → σ x →→st σ' x

ι≅ι : ι →st ι
ι≅ι x = st-var refl

≅-append : ∀{σ σ' M N x} -> σ →st σ' -> M →→st N -> σ ≺+ (x , M) →st  σ'
≺+ (x , N)
≅-append {σ} {σ'} {M} {N} {x} σ≅σ' M→→stN var with x ≟ var
... | yes _ = M→→stN
... | no _  = σ≅σ' var

st-refl : ∀{M} -> M →→st M
st-refl {v x} = st-var refl
st-refl {A · B} = st-app refl (st-refl {A}) (st-refl {B})
st-refl {ƛ x A} = st-abs refl (st-refl {A})

st-app-r : ∀{M M' N N'} -> M →→st M' -> N →→st N' -> M · N →→st M' · N'
st-app-r (st-var x1) y = st-app refl (st-var x1) y
st-app-r (st-app x x1 x2) y = st-app refl (st-app x x1 x2) y
st-app-r (st-abs x1 x2) y = st-app refl (st-abs x1 x2) y
st-app-r (st-alpha x x1) y = st-app refl (st-alpha x x1) y

append-seq0 : ∀ {M N K n} -> seqβ-st M N 0 -> seqβ-st N K n -> seqβ-st M K
```

```
n
append-seq0 {n = n} seqMNn nil = seqMNn
append-seq0 seqMNn (α-step seqNHn' H∼αK) = α-step (append-seq0 seqMNn
seqNHn') H∼αK
append-seq0 seqMNn (β-step seqNHm H→K@n n≥m) = β-step (append-seq0 seqMNn
seqNHm) H→K@n n≥m

lem-seq-appACBC : ∀ {A C B n} -> seqβ-st A B n -> ¬ (isAbs B) -> seqβ-st
(A · C) (B · C) n
lem-seq-appACBC nil ¬isAbsB = nil
lem-seq-appACBC (α-step seqAK K∼αB) ¬isAbsB = α-step (lem-seq-appACBC
seqAK (noAbsα ¬isAbsB K∼αB)) (∼· K∼αB ∼ρ)
lem-seq-appACBC (β-step seqAKm K→B@n m≤n) ¬isAbsB = β-step
(lem-seq-appACBC seqAKm (β-abs K→B@n ¬isAbsB)) (appNoAbsL K→B@n (β-abs
K→B@n ¬isAbsB)) m≤n

hap-st→st : ∀{L M N} -> L →→hap M -> M →→st N -> L →→st N
hap-st→st L→→hapM (st-var M→→hapx) = st-var (α-star-trans L→→hapM M→→hapx)
hap-st→st L→→hapM (st-app M→→hapAB A→→stC B→→stD) = st-app (α-star-trans
L→→hapM M→→hapAB) A→→stC B→→stD
hap-st→st L→→hapM (st-abs M→→hapλxA A→→stB) = st-abs (α-star-trans L→→hapM
M→→hapλxA) A→→stB
hap-st→st L→→hapM (st-alpha M→→stA' A'˜αA) = st-alpha (hap-st→st L→→hapM
M→→stA') A'˜αA

st-substσ≅σ' : ∀{M N σ σ'} -> M →→st N -> σ →st σ' -> M • σ →→st N • σ'
st-substσ≅σ' {M} {.(v _)} {σ} {σ'} (st-var {x} M→→hapx) σ→stσ' = hap-st→st
Mσ→→hapσx (σ→stσ' x)
                                          where Mσ→→hapσx = hap-subst {σ
= σ} M→→hapx
st-substσ≅σ' {M} {.(_ · _)} {σ} {σ'} (st-app M→→hapAB A→→stC B→→stD) σ→stσ'
                 = st-app (hap-subst {σ = σ} M→→hapAB) AC BD
                                          where AC = st-substσ≅σ' A→→stC
σ→stσ'
                                                BD = st-substσ≅σ' B→→stD
σ→stσ'
st-substσ≅σ' {M} {A} {σ} {σ'} (st-alpha M→→stA' A'˜αA) σ→stσ' = st-alpha
```

```
(st-substσ≅σ' M→→stA' σ→stσ') (≡→α (lemmaM∼M'→Mσ≡M'σ A'˜αA))
st-substσ≅σ' {M} {.(ϰ _ _)} {σ} {σ'} (st-abs {x} {A} {B} M→→hapλxA A→→stB)
σ→stσ' = st-alpha Mσ→→stλzBσz' z∼αx
                                                  where Mσ→→hapλyaAσ1 = hap-subst
{σ = σ} M→→hapλxA
                                                        ya = χ (σ , ϰ x A)
                                                        yb = χ (σ' , ϰ x B)
                                                        yz = χ (ι , ((ϰ x A) • σ)
· ((ϰ x B) • σ'))
                                                        σ1 = σ ≺+ (x , v ya)
                                                        σz = σ ≺+ (x , v yz)
                                                        σ1' = σ' ≺+ (x , v yb)
                                                        σz' = σ' ≺+ (x , v yz)
                                                        #app : ∀{x A B} -> x # (A
· B) -> (x # A) ∧ (x # B)
                                                        #app (#· x#A x#B) =
∧-intro x#A x#B
                                                        #zA : yz # (ϰ x A) • σ
                                                        #zA =  ∧-elim-l (#app
(lemma-χι (((ϰ x A) • σ) · ((ϰ x B) • σ'))))
                                                        #zB : yz # (ϰ x B) • σ'
                                                        #zB = ∧-elim-r (#app
(lemma-χι (((ϰ x A) • σ) · ((ϰ x B) • σ'))))
                                                        g : (A • σ1) • ι ≺+ (ya :
v yz) ∼α (A • σz) • ι ≺+ (yz : v yz)
                                                        g = ∼τ
(corollary1SubstLemma {x}{ya}{σ}{A} (χ-lemma2 σ (ϰ x A)))
                                                               (∼τ (lemma•ι {A •
σz}) (≡→α (sym (lemmaMι≺+x,x {yz} {A • σz}))))
                                                        λyaAσ1∼αλzAσz = ∼ϰ {A •
σ1} {A • σz} {ya} {yz} {yz} #zA #ϰ≡ g
                                                        Mσ→→hapλzAσz = α-step
Mσ→→hapλyaAσ1 λyaAσ1∼αλzAσz
                                                        σz≅σz' : σz →st σz'
                                                        σz≅σz' var with x ≟ var
                                                        ... | yes var≡x = st-refl
                                                        ... | no  var≠x = σ→stσ'
```

```
var
                                                    Aσz→→stBσz' =
st-substσ≅σ' A→→stB σz≅σz'
                                                    Mσ→→stλzBσz' = st-abs
Mσ→→hapλzAσz Aσz→→stBσz'
                                                    z∼αx :  ϰ yz (B • σz') ∼α
(ϰ x B) • σ'
                                                    z∼αx = ∼ϰ #ϰ≡ #zB (∼σ (∼τ
(corollary1SubstLemma {x}{yb}{σ'}{B} (χ-lemma2 σ' (ϰ x B))) (∼τ (lemma•ι
{B • σz'}) (≡→α (sym (lemmaMι≺+x,x {yz} {B • σz'}))))))

st-alpha-abs : ∀{P x M} -> P →→st ϰ x M -> Σx Λ (\M' -> (Σx V (\x' -> Σx Λ
(\M'' -> ((P →→hap ϰ x' M') ∧ (M' →→st M'')) ∧ (ϰ x' M'' ∼α ϰ x M)))))
st-alpha-abs (st-abs {x} {M'} {M} P→→hapλxM' M'→→stM) = (M' , (x , (M ,
∧-intro (∧-intro P→→hapλxM' M'→→stM) ∼ρ)))
st-alpha-abs (st-alpha P→→stλx'M' (∼ϰ y#λx'M' y#λxM M'x',y~Mx,y)) with
st-alpha-abs P→→stλx'M'
... | (M' , (x' , (M'' , ∧-intro (∧-intro P→→hapλx'M' M'→→stM'')
λx'M''~λxM))) = (M' , (x' , (M'' , ∧-intro (∧-intro P→→hapλx'M' M'→→stM'')
(∼τ λx'M''~λxM (∼ϰ y#λx'M' y#λxM M'x',y~Mx,y)))))

st-abs-subst : ∀ {L M N x} -> L →→st (ϰ x M) · N -> L →→st (M [ x := N ])
st-abs-subst {L} {M} {N} {x} (st-app {P} {N'} L→→hapPN' P→→stλxM N'→→stN)
with (st-substσ≅σ' M'→→stM (≅-append {x = x'} ι≅ι N'→→stN))
  where M'→→stM = ∧-elim-r (∧-elim-l (proj2 (proj2 (proj2 (st-alpha-abs
P→→stλxM)))))
        x' = proj1 (proj2 (st-alpha-abs P→→stλxM))
... | Hst = st-alpha (hap-st→st (α-star-trans (α-star-trans L→→hapPN'
(hap-app-r {P}{ϰ x' M'}{N'} P→→hapλx'M')) (append refl (hap-head {x'} {M'}
{N'}))) Hst) M''x',N~Mx,N
  where M'→→stM = ∧-elim-r (∧-elim-l (proj2 (proj2 (proj2 (st-alpha-abs
P→→stλxM)))))
        x' = proj1 (proj2 (st-alpha-abs P→→stλxM))
        M' = proj1 (st-alpha-abs P→→stλxM)
        P→→hapλx'M' = ∧-elim-l (∧-elim-l (proj2 (proj2 (proj2
(st-alpha-abs P→→stλxM)))))
        M'' = proj1 (proj2 (proj2 (st-alpha-abs P→→stλxM)))
```

```
        M''x',N~Mx,N : (M'' [ x' := N ]) ∼α (M [ x := N ])
        M''x',N~Mx,N with ∧-elim-r (proj2 (proj2 (proj2 (st-alpha-abs
P→→stλxM))))
        ... | (∼ƛ {y = y}  y#λx'M'' y#λxM M''x',y~Mx,y) = ∼τ (∼σ
(corollary1SubstLemma {x'}{y}{ι}{M''}{N} (lemma#→ι#⌊ y#λx'M''))) (∼τ (≡→α
(lemmaM∼M'→Mσ≡M'σ {σ = (ι ≺+ (y , N))} M''x',y~Mx,y))
(corollary1SubstLemma {x}{y}{ι}{M}{N} (lemma#→ι#⌊ y#λxM)))
st-abs-subst {L} {M} {N} {x} (st-alpha {(ƛ y M') · N'} L→→stλyM'N' (∼· (∼ƛ
{y = w} p q rs) N'~N)) = st-alpha (st-abs-subst L→→stλyM'N') (∼τ
M'y,N'~MxN' (lemma-subst {M = M} ∼ρ x,N'~xN))
  where lem-α-both : ∀{M N M' N'} -> M ∼α N -> M ∼α M' -> N ∼α N' -> M' ∼α
N'
        lem-α-both M~N M~M' N~N' = ∼τ (∼τ (∼σ M~M') (M~N)) N~N'
        M'y,N'~MxN' : (M' [ y := N' ]) ∼α (M [ x := N' ])
        M'y,N'~MxN' = lem-α-both (≡→α (lemmaM∼M'→Mσ≡M'σ {σ = ι ≺+ (w ,
N')} rs)) (corollary1SubstLemma {y} {w} {ι} {M = M'} (lemma#→ι#⌊ p))
(corollary1SubstLemma {x} {w} {ι} {M = M} (lemma#→ι#⌊ q))
        x,N'~xN : ι ≺+ (x : N') ∼α ι ≺+ (x : N) ⌊ M
        x,N'~xN w w*M with x ≟ w
        ... | yes _ = N'~N
        ... | no _ = ∼v

st-β→st : ∀{L M N} -> L →→st M -> M ⟶β N -> L →→st N
st-β→st L→→stM M⟶βN with proj2 M⟶βN
... | outer-redex = st-alpha N' ∼ρ
                      where N' = st-abs-subst L→→stM
st-β→st (st-app {D = C} L→→hapA'C' A'→→stA C'→→stC) M⟶βN | (appNoAbsL {n =
n} A⟶B@n _)
                                                = st-app L→→hapA'C' A'B
C'→→stC
                                            where A'B = st-β→st A'→→stA (
n , A⟶B@n)
st-β→st (st-alpha L→→stM' M'αM) M⟶βN | _ = st-alpha (st-β→st L→→stM' N')
N'αN
                                           where N' = ∧-elim-l (proj2
(lem-βα M⟶βN (∼σ M'αM)))
                                                 N'αN = ∧-elim-r (proj2
```

```
(lem-βα M⟶βN (∼σ M'αM)))
st-β→st (st-app {D = C} L→→hapA'C' A'→→stA C'→→stC) M⟶βN | (appAbsL {n =
n} A⟶B@n _)
                                                  = st-app L→→hapA'C' A'B
C'→→stC
                                                where A'B = st-β→st A'→→stA (
n , A⟶B@n)
st-β→st (st-app {C = C} L→→hapC'A' C'→→stC A'→→stA) M⟶βN |  (appNoAbsR {n
= n} A⟶B@n _)
                                                  = st-app L→→hapC'A'
C'→→stC A'B
                                                where A'B = st-β→st A'→→stA (
n , A⟶B@n)
st-β→st (st-app {C = C} L→→hapC'A' C'→→stC A'→→stA) M⟶βN |  (appAbsR {n =
n} A⟶B@n _)
                                                  = st-app L→→hapC'A'
C'→→stC A'B
                                                where A'B = st-β→st A'→→stA (
n , A⟶B@n)
st-β→st (st-abs {x = x} L→→hapA A→→stB) M⟶βN | (abs {n = n} A⟶B@n) =
st-abs L→→hapA A'B
                                                where A'B = st-β→st A→→stB (
n , A⟶B@n)

β→st : ∀{M N} -> M →→β N -> M →→st N
β→st {M} refl = st-refl
β→st (α-step M→→βN' N'˜N) = st-alpha (β→st M→→βN') N'˜N
β→st (append M→→βK K⟶βN) with β→st M→→βK
... | M→→stK = st-β→st M→→stK K⟶βN

-- ------------------------------------------------
-- 3.4 STANDARD SEQUENCES
-- ------------------------------------------------

lem-leftmost→seqβstl : ∀{M N} -> M ⟶l N -> seqβ-st M N 0
lem-leftmost→seqβstl MβN = β-step nil MβN (z≤n)
```

```
lem-leftmost→seqβst : ∀{M N} -> M →→l N -> seqβ-st M N 0
lem-leftmost→seqβst refl = nil
lem-leftmost→seqβst (α-step M→→lK K~N) = α-step (lem-leftmost→seqβst
M→→lK) K~N
lem-leftmost→seqβst (append M→→lK K⟶lN) = append-seq0 (lem-leftmost→seqβst
M→→lK) (lem-leftmost→seqβstl K⟶lN)

hap→seqβst : ∀{M N} -> M →→hap N -> seqβ-st M N 0
hap→seqβst MhapN = lem-leftmost→seqβst (hap→l MhapN)

abs-seq : ∀ {x M N n} -> seqβ-st M N n -> seqβ-st (ƛ x M) (ƛ x N) n
abs-seq nil = nil
abs-seq {x} {M} {N} (α-step {K = K} seqMKn K∼αN) = α-step seqλMλKn (∼ƛ #ƛ≡
#ƛ≡ (≡→α (lemmaM∼M'→Mσ≡M'σ K∼αN)))
                        where seqλMλKn = abs-seq seqMKn
abs-seq {x} (β-step seqMKn K→N@m n≤m) = β-step (abs-seq {x} seqMKn) (abs
K→N@m) n≤m

lem-seq-appACBC-abs : ∀ {A C B n} -> seqβ-st A B n -> isAbs B
  -> (seqβ-st (A · C) (B · C) n) ∨ (seqβ-st (A · C) (B · C) (suc n))
lem-seq-appACBC-abs nil _ = ∨-intro1 nil
lem-seq-appACBC-abs (α-step seqAK K∼αB) isAbsB with (lem-seq-appACBC-abs
seqAK (isAbsα isAbsB K∼αB))
... | ∨-intro1 seqACKCn    = ∨-intro1 (α-step seqACKCn (∼· K∼αB ∼ρ))
... | ∨-intro2 seqACKCsucn = ∨-intro2 (α-step seqACKCsucn (∼· K∼αB ∼ρ))
lem-seq-appACBC-abs {C = C} (β-step {K = K} seqAKm K→B@n m≤n) isAbsB with
isAbs? K
... | yes isAbsK with lem-seq-appACBC-abs {C = C} seqAKm isAbsK
...    | ∨-intro1 seqACKCm = ∨-intro2 (β-step seqACKCm (appAbsL K→B@n
isAbsK) (≤-step m≤n))
...    | ∨-intro2 seqACKCsucm = ∨-intro2 (β-step seqACKCsucm (appAbsL
K→B@n isAbsK) (s≤s m≤n))
lem-seq-appACBC-abs {C = C} (β-step {K = K} seqAKm K→B@n m≤n) isAbsB | no
¬isAbsK
                    = ∨-intro1 (β-step seqACKCm (appNoAbsL K→B@n ¬isAbsK)
m≤n)
                          where seqACKCm = lem-seq-appACBC {C = C}
```

```
seqAKm ¬isAbsK

lem-seq-app-abs-¬α : ∀{A B C D n m} -> seqβ-st A B n -> seqβ-st C D m ->
isAbs B -> ¬ (C ∼α D)
                                                        -> seqβ-st (A ·
C) (B · D) (suc (m + countRedexes B))
lem-seq-app-abs-¬α seqABn nil isAbsB ¬C∼αD = ⊥-elim (¬C∼αD ∼ρ)
lem-seq-app-abs-¬α {C = C} seqABn (α-step {K = K} seqCKm K∼αD) isAbsB
¬C∼αD with C ∼α? K
... | yes C∼αK = ⊥-elim (¬C∼αD (∼τ C∼αK K∼αD))
... | no ¬K∼αB =  α-step (lem-seq-app-abs-¬α seqABn seqCKm isAbsB ¬K∼αB)
(∼· ∼ρ K∼αD)
lem-seq-app-abs-¬α {A} {B} {C} seqABn (β-step {K = K} {n = m}{n₀ = m'}
seqCKm K→Dm' m≤m') isAbsB ¬C∼αD with C ∼α? K | lem-seq-appACBC-abs {C = C}
seqABn isAbsB
... | yes C∼αK | ∨-intro1 seqACBCn = β-step (α-step seqACBCn (∼· ∼ρ C∼αK))
(appAbsR K→Dm' isAbsB) (≤-sum-l {z = suc m'} (seq-redexCount seqABn))
... | yes C∼αK | ∨-intro2 seqACBCsucn = β-step (α-step seqACBCsucn (∼· ∼ρ
C∼αK)) (appAbsR K→Dm' isAbsB) (s≤s (≤-sum-l {z = m'} (seq-redexCount
seqABn)))
... | no ¬C∼αK | ∨-intro1 seqACBCn = β-step seqACBKsucm+crB (appAbsR K→Dm'
isAbsB) (s≤s (≤-sum m≤m'))
               where seqACBKsucm+crB = lem-seq-app-abs-¬α seqABn seqCKm
isAbsB ¬C∼αK
... | no ¬C∼αK | ∨-intro2 seqACBCsucn = β-step seqACBKsucm+crB (appAbsR
K→Dm' isAbsB) (s≤s (≤-sum m≤m'))
               where seqACBKsucm+crB = lem-seq-app-abs-¬α seqABn seqCKm
isAbsB ¬C∼αK




lem-seq-app-¬α : ∀{A B C D n m} -> seqβ-st A B n -> seqβ-st C D m -> ¬
(isAbs B) -> ¬ (C ∼α D)
                                                        -> seqβ-st (A ·
C) (B · D) (m + countRedexes B)
lem-seq-app-¬α seqABn nil ¬isAbsB ¬C∼αD = ⊥-elim (¬C∼αD ∼ρ)
```

```
lem-seq-app-¬α {C = C} seqABn (α-step {K = K} seqCKm K∼αD) ¬isAbsB ¬C∼αD
with C ∼α? K
... | yes C∼αK = ⊥-elim (¬C∼αD (∼τ C∼αK K∼αD))
... | no ¬K∼αB = α-step (lem-seq-app-¬α seqABn seqCKm ¬isAbsB ¬K∼αB) (∼·
∼ρ K∼αD)
lem-seq-app-¬α {C = C} seqABn (β-step {K = K} {n₀ = m'} seqCKm K→Dm' m≤m')
¬isAbsB ¬C∼αD with C ∼α? K
... | yes C∼αK = β-step seqACBKn (appNoAbsR K→Dm' ¬isAbsB) (≤-sum-l {z =
m'} (seq-redexCount seqABn))
          where seqACBCn = lem-seq-appACBC seqABn ¬isAbsB
                seqACBKn = α-step seqACBCn (∼· ∼ρ C∼αK)
... | no ¬C∼αK = β-step seqACBKm+crB (appNoAbsR K→Dm' ¬isAbsB) (≤-sum m≤m')
          where seqACBKm+crB = lem-seq-app-¬α seqABn seqCKm ¬isAbsB ¬C∼αK


st→seqβst : ∀{M N} -> M →→st N -> Σx ℕ (\n -> seqβ-st M N n)
st→seqβst {M} {v x} (st-var M→hapx) = ( 0 , lem-leftmost→seqβst (hap→l
M→hapx))
st→seqβst (st-app {A} {B} {C} {D} L→→hapAB A→→stC B→→stD) with st→seqβst
A→→stC | st→seqβst B→→stD | isAbs? C | B ∼α? D
... | (m , seqACm) | (n , seqBDn) | no ¬isAbsC | yes B∼αD = (m ,
append-seq0 seqLAB0 seqABCDm)
                  where seqLAB0 = lem-leftmost→seqβst (hap→l L→→hapAB)
                        seqABCBm = lem-seq-appACBC {C = B} seqACm ¬isAbsC
                        seqABCDm = α-step seqABCBm (∼· ∼ρ B∼αD)
... | (m , seqACm) | (n , seqBDn) | no ¬isAbsC | no ¬B∼αD = (n +
countRedexes C , append-seq0 seqLAB0 seqABCDn+crC)
                  where seqLAB0 = lem-leftmost→seqβst (hap→l L→→hapAB)
                        seqABCDn+crC = lem-seq-app-¬α seqACm seqBDn
¬isAbsC ¬B∼αD
... | (m , seqACm) | (n , seqBDn) | yes isAbsC | no ¬B∼αD = ( suc (n +
countRedexes C) , append-seq0 seqLAB0 (lem-seq-app-abs-¬α seqACm seqBDn
isAbsC ¬B∼αD))
                  where seqLAB0 = lem-leftmost→seqβst (hap→l L→→hapAB)
... | (m , seqACm) | (n , seqBDn) | yes isAbsC | yes B∼αD with
lem-seq-appACBC-abs {C = B} seqACm isAbsC
... | ∨-intro1 seqABCBn = (m , append-seq0 seqLAB0 (α-step seqABCBn (∼· ∼ρ
```

```
B∼αD)))
                    where seqLAB0 = lem-leftmost→seqβst (hap→l L→→hapAB)
... | ∨-intro2 seqABCBsucn = (suc m , append-seq0 seqLAB0 (α-step
seqABCBsucn (∼· ∼ρ B∼αD)))
                    where seqLAB0 = lem-leftmost→seqβst (hap→l L→→hapAB)
st→seqβst (st-abs {x} L→→hapA A→→stB) with st→seqβst A→→stB
... | (n , seq) = (n , append-seq0 (lem-leftmost→seqβst (hap→l L→→hapA))
(abs-seq {x} seq))
st→seqβst (st-alpha L→→stA' A'~αA) with st→seqβst L→→stA'
... | (n , seq) = (n , α-step seq A'~αA)

lem-seq-appCACB : ∀ {A C B n} -> seqβ-st A B n -> ¬ (isAbs C) -> ¬ (A ∼α
B) -> seqβ-st (C · A) (C · B) (n + countRedexes C)
lem-seq-appCACB nil _ ¬AαA = ⊥-elim (¬AαA ∼ρ)
lem-seq-appCACB {A} (α-step {n = n} {K = K} seqAKn K∼αB) ¬isAbsC ¬AαB with
A ∼α? K
... | yes A∼αK = ⊥-elim (¬AαB (∼τ A∼αK K∼αB))
... | no ¬A∼αK = α-step (lem-seq-appCACB seqAKn ¬isAbsC ¬A∼αK) (∼· ∼ρ K∼αB)
lem-seq-appCACB {A} {C} (β-step {K = K} seqAKn K→B@m n≤m) ¬isAbsC ¬AαB
with A ∼α? K
... | yes A∼αK = β-step seqCACK (appNoAbsR K→B@m ¬isAbsC) z≤n
          where seqCACK = α-step nil (∼· (∼ρ {C}) A∼αK)
... | no ¬A∼αK = β-step (lem-seq-appCACB seqAKn ¬isAbsC ¬A∼αK) (appNoAbsR
K→B@m ¬isAbsC) (≤-sum n≤m)


lem-seq-appCACB-abs : ∀ {A C B n} -> seqβ-st A B n -> isAbs C -> ¬ (A ∼α
B) -> seqβ-st (C · A) (C · B) (suc (n + countRedexes C))
lem-seq-appCACB-abs nil _  ¬AαA = ⊥-elim (¬AαA ∼ρ)
lem-seq-appCACB-abs {A} (α-step {n = n} {K = K} seqAKn K∼αB) isAbsC ¬AαB
with A ∼α? K
... | yes A∼αK = ⊥-elim (¬AαB (∼τ A∼αK K∼αB))
... | no ¬A∼αK = α-step (lem-seq-appCACB-abs seqAKn isAbsC ¬A∼αK) (∼· ∼ρ
K∼αB)
lem-seq-appCACB-abs {A} {C} (β-step {K = K} seqAKn K→B@m n≤m) isAbsC ¬AαB
with A ∼α? K
... | yes A∼αK = β-step seqCACK (appAbsR K→B@m isAbsC) z≤n
```

```
          where seqCACK = α-step nil (∼· (∼ρ {C}) A∼αK)
... | no ¬A∼αK = β-step (lem-seq-appCACB-abs seqAKn isAbsC ¬A∼αK) (appAbsR
K→B@m isAbsC) (s≤s (≤-sum n≤m))

st-exist : ∀{M N} -> M →→st N -> Σx Λ (λ N' -> (M →→st N') ∧ (N' ∼α N))
st-exist {N = N} M→→stN = (N , ∧-intro M→→stN ∼ρ)

-- STANDARDIZATION THEOREM --

standardization : ∀{M N} -> M →→β N -> Σx ℕ (λ n -> seqβ-st M N n)
standardization M→→βN = st→seqβst (β→st M→→βN)

seqβ-st→st : ∀{M N n} -> seqβ-st M N n -> M →→st N
seqβ-st→st nil = st-refl
seqβ-st→st (α-step seqAB'n B'˜B) = st-alpha (seqβ-st→st seqAB'n) B'˜B
seqβ-st→st (β-step seqAB'n B'⟶Bn' n≤n') = st-β→st (seqβ-st→st seqAB'n)
(lem-βex B'⟶Bn')

-------------------------------------------------------------
-- 4 THE LEFTMOST REDUCTION THEOREM
-------------------------------------------------------------

nf : Λ -> Set
nf M = countRedexes M ≡ 0

nf→l : ∀{M N n} -> M β N @ n -> nf N -> n ≡ 0
nf→l outer-redex crN≡0 = refl
nf→l (appNoAbsL {n} {A} {B} AβBn ¬isAbsA) with isAbs? B
... | yes isAbsB = λ ()
... | no ¬isAbsB = λ crB+crC≡0 -> nf→l AβBn (lem-sum-l crB+crC≡0)
nf→l (appAbsL {n} {A} {B} AβBn isAbsA) with isAbs? B
... | yes isAbsB = λ ()
... | no ¬isAbsB = λ crB+crC≡0 -> ⊥-elim ((β-abs AβBn ¬isAbsB) isAbsA)
nf→l (appNoAbsR {n} {A} {B} {C} AβBn ¬isAbsC) with isAbs? C
... | yes isAbsC = ⊥-elim (¬isAbsC isAbsC)
... | no _ = λ crC+crB≡0 -> lem-sum-zero (nf→l AβBn (lem-sum-r
{countRedexes C} crC+crB≡0)) (lem-sum-l crC+crB≡0)
```

```
nf→l (appAbsR {n} {A} {B} {C} AβBn isAbsC) with isAbs? C
... | yes _ = λ ()
... | no ¬isAbsC = ⊥-elim (¬isAbsC isAbsC)
nf→l (abs x1) crN≡0 = nf→l x1 crN≡0

seqβ0→l : ∀ {A B} -> seqβ-st A B 0 -> A →→l B
seqβ0→l nil = refl
seqβ0→l (α-step seqAB0 x) = α-step (seqβ0→l seqAB0) x
seqβ0→l (β-step {n = zero} seqAB0 AβBn n<=0) = append (seqβ0→l seqAB0) AβBn
seqβ0→l (β-step {n = suc _} seqAB0 AβBn ())

seqst→l : ∀{A B n} -> seqβ-st A B n -> nf B -> A →→l B
seqst→l nil crB≡0 = refl
seqst→l (α-step seqβAKn K˜B) crB≡0 = α-step (seqst→l seqβAKn (subst2 _≡_
(sym (lem-crα K˜B)) refl crB≡0)) K˜B
      where seqst→β : ∀{M N n} -> seqβ-st M N n -> M →→β N
            seqst→β nil = refl
            seqst→β (α-step x x1) = α-step (seqst→β x) x1
            seqst→β {n = n} (β-step x x1 x2) = append (seqst→β x) (n : x1)
seqst→l {A} (β-step {K} {n} {n'} {B} seqβAKn KβBn' n<=n') crB≡0 = append
A→→lK K⟶lB
      where n'≡0 : n' ≡ 0
            n'≡0 = nf→l KβBn' crB≡0
            n≡0 : n ≡ 0
            n≡0 = lem≤0 (subst2 _≤_ refl n'≡0 n<=n')
            seqβstAK0 : seqβ-st A K 0
            seqβstAK0 = subst3 seqβ-st refl refl n≡0 seqβAKn
            A→→lK : A →→l K
            A→→lK = seqβ0→l seqβstAK0
            K⟶lB : K ⟶l B
            K⟶lB = subst3 _β_@_ refl refl n'≡0 KβBn'

leftmost-nf : ∀{A B} -> A →→β B -> nf B -> A →→l B
leftmost-nf A→→βB crB≡0 = seqst→l (proj2 (standardization A→→βB)) crB≡0
```