Università degli Studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea Magistrale in Informatica

Normalization by Evaluation for Typed λ-Calculi with Weak Conversions

Tesi di Laurea Magistrale

Relatore

Prof.ssa Maria Emilia Maietti

Laureando

Filippo Sestini

Abstract

This thesis addresses the topic of constructive normalization proofs for typed λ-calculi with weak notions of conversion between λ-terms, and their formalization in a computer proof-checker. The name "weak conversion" encompasses various definitions, but all of them share the common property of not validating the ξ rule of the λ-calculus, which is responsible for allowing arbitrary reductions under λ-abstractions.

A particularly interesting notion of weak conversion is what we call CH-weak conversion. This relation is subtle, since despite the absence of the ξ rule, it still allows to perform a limited class of reductions on terms under λ-abstractions. This notion of reduction is different in some crucial aspects from both weaker and stronger reduction relations, thus existing constructive proofs of normalization do not adapt neatly to it, and new solutions have to be found. This thesis provides an analysis of the term-rewriting properties of CH-weak conversion, and develops a novel method that is used to prove normalization for a version of Gödel's System T with this notion of equality. The proof has been fully formalized and verified in the Agda proof-checker, and provides the first account of a constructive proof of normalization for a typed λ-calculus with CH-weak equality.

The thesis also investigates weak equality in the context of dependent types, with the definition of an original version of Martin-Löf Type Theory (MLTT) with a notion of weak conversion called "weak explicit substitutions", and a full computer formalization of the normalization theorem. This version of MLTT is also compared with the Minimal Type Theory, a dependent type theory with CH-weak conversion that constitutes the intensional level of the Minimalist Foundation.

Contents

Chapter 1

Introduction

Typed λ-calculi have first arised in logic as foundational theories for mathematics, starting from the works of Church and Curry [23, 17]. They then evolved into expressive formalisms for defining and reasoning about programming languages and their properties [39]. A rich and active line of research is the one that studies these formalisms from both points of view, according to what is known as the *Curry-Howard correspondence*, or the *propositions-as-types* paradigm [46].

The λ-calculus is, in its essence, a term-rewriting system [19], usually considered together with a notion of *reduction*, that is a binary relation between λ-terms that gives a set of rewrite rules. A reduction relation defines how to "compute" with terms of the λ-calculus, or "run" programs written as λ-terms, by repeated application of rewrite rules. When a term cannot be reduced any further, we say that it reached a *normal form*. The process of finding a normal form for a term is called *normalization*. Normal forms can be seen as representing *values* for λ-terms, thus they attach *meaning* to them. Two terms are then considered equal whenever they have the same normal form.

Through the lens of the Curry-Howard correspondence, type theories can be seen as systems of formal logic, where propositions are identified with the type of their proofs, so that proving a proposition is nothing more than providing a term of the corresponding type. A characteristic of most typed λ-calculi is that they enjoy the *normalization* property, meaning that every well-typed term has a normal form. This property is important because it implies that certain types are empty, making the theory consistent as a logic. It follows that, when considering typed λ-calculi as foundational systems for mathematics, the particular notion of reduction that is considered, as well as its normalizability, are crucial aspects that must be taken into great consideration.

The standard reduction relation of the λ-calculus is β-reduction [23, 16]. Sometimes a stronger notion is considered, with the addition of the η-reduction scheme that represents a weak principle of extensionality of functions. Stronger reduction relations imply that more equations hold, something that is particularly useful in proof assistant based on dependent type theories, where computational equalities between λ-terms can be checked automatically by the machine.

There are, however, some good reasons to consider notions of reduction that are *weaker* then the standard, "full" β-reduction, in the sense that the allowed reductions are a subset of full β: a weak reduction relation equates fewer terms, so weak equational theories have more models. Moreover, certain properties of term-rewriting systems, like the standardization theorem, are easier to prove under weaker reduction relations than under full β-reduction [22].

There are several different notions of reduction that can be deemed as *weak*, but they all share the common property of not validating the ξ rule, which is responsible for allowing arbitrary reductions under λ binders, to be performed on the body of abstractions. This aspect makes weak reduction particularly interesting for the study of the dynamics of programming languages [41, 31], where evaluation only involves *closed* terms, and thus never goes under binders. Among the most elementary notions of reduction is *weak-head reduction.* Weak-head reduction can be informally described as β-reduction without the ξ rule.

Another kind of weak λ-calculus can be defined in the framework of explicit substitutions [7]. Weak explicit substitutions are usually defined from explicit substitution calculi by disallowing substitutions to be propagated under λs or other binding forms [41, 12, 26]. Explicit substitutions, including their weak variant, are easier to formalize, since they do not involve metalinguistic substitution operations. Moreover, their generally lead to efficient implementations, because they correspond quite closely to abstract machines used in concrete implementations of the λ-calculus [7].

A perhaps more obscure notion of weak reduction is known in the literature as *restricted reduction* [32] or *weak combinatory reduction* [22], that we call CH-weak reduction here to avoid confusion with other weak reduction relations. CH-weak reduction can be informally described as a β-reduction where contraction under λ-abstractions is restricted to a particular class of terms, called *weak redexes.* Formally, this is achieved by defining the reduction relation without the usual congruence rules for term constructors, as in the full β-reduction, but using a substitution rule instead. As a consequence of this definition, the ξ rule is rendered invalid. CH-weak reduction can be shown to correspond in a precise way to reduction in combinatory logic [22].

A notable example of a typed λ-calculus equipped with a CH-weak reduction relation is found within the framework of the Minimalist Foundation, a two-level foundation for constructive mathematics ideated in [43] and completed in [42]. The *intensional* level of the Minimalist Foundation in [42] is represented by a dependent type theory in the style of Martin-Löf Type Theory, called the Minimal Type Theory, or mTT. mTT exhibits a peculiar formulation of judgmental equality that replaces all congruence rules with a primitive substitution rule, thus implementing CH-weak reduction instead of full β-reduction.

The subtlety of CH-weak reduction lies in the fact that, despite the absence of the ξ rule, it is still possible to perform certain reductions on terms under λ-abstractions and binders in general, via the substitution rule. Thus, when defining an algorithm to compute the normal form of terms, it is not possible to just ignore λ-abstractions, because their bodies could contain valid redexes. But at the same time, one can not simply proceed by recursion inside the λ and evaluate the body, because such operation is only sound, in general, in presence of the ξ rule.

1.1 Contributions

This thesis addresses the problem of constructive normalization proofs for typed λ-calculi with weak notions of equality between terms. The work was originally motivated by the open normalization problem for the mTT, a dependent type theory with CH-weak reduction. This notion of reduction is different in some crucial aspects from both weaker and stronger reduction relations, thus existing constructive proofs of normalization do not adapt neatly to it, and new solutions have to be found. For this reason, in this thesis we specifically focus our attention on typed calculi with CH-weak reduction and similar notions.

The first contribution is represented by a method to define a normalization procedure for CH-weak reduction on untyped λ-terms, and use it to prove normalization for simply-typed λ-calculi with CH-weak equality judgments. To do this, we show how it is possible to systematically construct an "explicit" version of the original, "implicit" weak calculus that one intends to prove normalization for. The explicit calculus is specifically designed to facilitate the metatheoretical analysis of CH-weak definitional equality, by making certain aspects of this particular notion of reduction syntactically evident in the terms and judgments of the calculus, so that all the missing congruence rules can be restored. The explicit calculus is proved normalizing by the semantic method of Normalization by Evaluation [8]. Then, we show that one can establish a suitable correspondence between the equality judgments of the two calculi, that allows to transfer all normalization results from the explicit calculus to the implicit one. The proposed proof method is demonstrated in its entirety on a version of System T with CH-weak equality as typed equality judgments. This appears to be first analysis of normalization for a typed λ-calculus with CH-weak equality, encompassing the definition of a normalization procedure and a full formal proof of its correctness. In fact, we observe that CH-weak equality, and especially typed calculi employing it, have received rather limited attention in the literature so far. Current normalization proofs for typed λ-calculi either refer to stronger reduction relations, like full β-reduction in which the ξ rule is valid and computation under binders is treated normally, or target calculi with weaker notions of computation [45, 25], like *weak-head reduction* or *weak explicit substitutions*, where the absence of the ξ rule is not an obstacle since no reductions ever happen under binders.

The second part of the thesis focuses on dependent types. We begin by pointing out certain characteristics of dependently-typed calculi that prevent a straightforward adaptation of the "explicit" construction used to normalize weak System T. We thus propose an alternative approach to weak reduction in a dependently-typed setting, based on *weak explicit substitutions*. We formulate MLTT^{wk}, a version of Martin-Löf Type Theory with one universe and large elimination, and a weak notion of reduction defined by means of explicit substitutions. The system can be seen as a dependently-typed version of existing simply-typed calculi with weak explicit substitutions, like [12]. However, the system appears to be an original formulation, or at least the author is not aware, at the moment of writing, of similar investigations of MLTT with specifically *weak* explicit substitutions. The calculus is proved normalizing by instantiating normalization by evaluation. We observe that weak explicit substitutions exhibit many of the good properties of CH-weak reduction, despite giving rise to a weaker equational theory, and are also easier to formalize and reason about. We thus consider the possibility of integrating weak explicit substitutions in the treatment of the Minimal Type Theory.

All the results presented in this thesis have been fully formalized and checked in the proof assistant Agda [21]. Thus, this work additionally contributes to the community of formalized mathematics with two completely verified proofs of normalization [5, 4], which, given the difficulty of mechanization of λ-calculi in general, and regarding normalization proofs in particular, is an achievement on its own.

1.2 Outline

- Chapter 2 establishes the foundational notions on which the rest of the thesis is based.

- Chapter 3 starts with an analysis of weak reduction in the untyped λ-calculus, and then develops an "explicit" syntax for λ-terms that leads to a simple definition of a recursive evaluator for CH-weak equality. It then focuses on an alternative weak notion of reduction, based on explicit substitutions. It is observed that, although weaker than CH-weak and thus not useful to CH-weakly normalize λ-terms, weak explicit substitutions exhibit many of the good properties of CH-weak equality, making them an alternative worth exploring.

- Chapter 4 introduces System T^{wk}, a version of System T with CH-weak equality judgments, that can be seen as a propositional fragment of mTT. It then points out the difficulties that one faces when attempting to prove normalization for the system. These problems are addressed with the definition of an "explicit" reformulation of T^{wk}'s type system, System T^{ex}, that directly reflects the "explicit" syntax of Chapter 3 in the very definition of judgments. It is shown that the problems that were found in System T^{wk} do not arise in T^{ex}, making it more suitable for a normalization proof. The chapter concludes with the proof of correspondence between T^{wk} and T^{ex}, that will allow in Chapter 5 to transfer all normalization results proved for System T^{ex} to T^{wk}.

- Chapter 5 proves normalization by evaluation for System T^{ex}. It concludes by establishing normalization for System T^{wk} via the correspondence previously shown in Chapter 4.

- Chapter 6 addresses weak equality in the context of dependent types. It begins with a discussion of the challenges that would arise in the attempt of adapting the "explicit" construction to a dependently-typed calculus. It then shifts the focus to an alternative notion of weak reduction, by defining a version of Martin-Löf Type Theory with weak explicit substitutions and large elimination. The calculus is proved normalizing, again via normalization by evaluation.

- Chapter 7 concludes with a summary of the contributions, and a discussion on the advantages of weak explicit substitutions over CH-weak reduction, in particular in the context of the Minimal Type Theory. It finally identifies possible directions for future work, and points out related work in the area of weak λ-calculi and normalization proofs.

Chapter 2

Background

This chapter establishes the conceptual and technical background of the thesis.

2.1 Lambda calculus

The lambda calculus [16] is a formalism in mathematical logic introduced in the 1930s by Alonzo Church [23]. It is a formal language, whose terms are inductively defined as follows:

$$t, u := x \mid \lambda x.t \mid tu$$

The term $\lambda x.t$ is a λ-*abstraction*, tu is an *application*. Variables in a term can either appear *free* or *bound*. A variable x is bound in a term t if it has been abstracted by a λ-abstraction. For example, in the term $\lambda x.xy$, the occurrence of the free variable x in the term xy has been abstracted by λx. We denote by $FV(t)$ the set of free variables of a term t. A term t such that $FV(t) = \emptyset$ is *closed*. Closed terms are also called *combinators*.

The λ-calculus is usually considered together with some notion of *reduction* telling us how to "run" terms. The most basic form of computation for λ-terms is the β-contraction scheme

$$(\lambda x.t)s \rightsquigarrow t[s/x]$$

where $t[s/x]$ stands for capture-avoiding substitution, that replaces every occurrence of x in t with s, making sure that no free variables in s become bound by abstractions in t. We define β-*reduction*, in its standard formulation, as the binary relation corresponding to the congruence closure of β-contraction.

$$\frac{}{(\lambda x.t)s \longrightarrow t[s/x]}\ (\beta) \qquad \frac{t \longrightarrow s}{\lambda x.t \longrightarrow \lambda x.s}\ (\xi) \qquad \frac{t \longrightarrow r}{ts \longrightarrow rs}\ (\nu) \qquad \frac{s \longrightarrow r}{ts \longrightarrow tr}\ (\mu)$$

We write the reflexive transitive closure of β-reduction as $\longrightarrow^*$, and define β-*conversion* as the equivalence closure of β-reduction, and write it like $=_\beta$. Conversion relations usually include α-conversion, telling us that bound variables can be renamed freely:

$$\lambda x.t = \lambda y.t[y/x] \qquad\qquad y \notin \mathrm{FV}(t)$$

An extension of β-reduction is the so-called $\beta\eta$-reduction, that is obtained from β-reduction with the addition of the η-rule, corresponding to the following η contraction schema:

$$\lambda x.(t\ x) \rightsquigarrow t \qquad\qquad x \notin \mathrm{FV}(t)$$

The η rule is a weak extensionality principle. In fact, η and ξ together imply admissibility of the rule schema of function extensionality [16], asserting that $t = s$ follows from $t\ x = s\ x$ for any terms t, s and variables x.

We say that a term M is in *normal form* (or it is a normal form), when there is no term N such that $M \to N$. We say that a term M *has a normal form* when there exists a normal form N such that $M \longrightarrow^* N$. A fundamental property of the λ-calculus is that the order in which β-reductions are performed does not matter.

Theorem 1 (Church-Rosser property)**.** *Let M, P, Q be λ-terms, such that $M \longrightarrow^* P$ and $M \longrightarrow^* Q$. Then, there exists a term R such that $P \longrightarrow^* R$ and $Q \longrightarrow^* R$.*

This property is called *confluence* or *Church-Rosser property*, and it implies the uniqueness of normal forms.

Corollary 1. *Let M, N_1, N_2 be λ-terms, such that $M \longrightarrow^* N_1$ and $M \longrightarrow^* N_2$, and N_1, N_2 are normal forms. Then, $N_1 \equiv N_2$.*

Proof. By confluence, there exists N such that $N_1 \longrightarrow^* N$ and $N_2 \longrightarrow^* N$. But since N_1, N_2 are normal, the steps of reduction must be zero, hence $N_1 \equiv N \equiv N_2$. □

Thus, a confluent notion of reduction like β-reduction gives us a way to assign *meaning* to λ-terms, namely their normal form. As we will see, not all notions of reductions are confluent.

2.1.1 Typed lambda calculus

The untyped lambda calculus can be enriched in various ways. One of these is the addition of a type discipline [15]. Typed λ-calculus is concerned with typing judgments of the form $\Gamma \vdash t : A$, where Γ is a context of typing assumptions assigning types to t's free variables, t is a λ-term, and A is a type. The collection of terms t such that $\Gamma \vdash t : A$ holds for some Γ and A constitutes the *well-typed* terms. An example of typed calculus is the Simply-Typed λ-Calculus (STLC) [15]. Its types are given by the following grammar:

$$A, B ::= \iota \mid A \to B$$

where ι is an primitive type that is left unspecified, and $A \to B$ is the higher-order function space. Its inference rules are given as follows:

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B} \qquad \frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash s : A}{\Gamma \vdash ts : B}$$

Typed λ-calculi are usually formulated in either *Church-style* or *Curry-style.* In the first case, the syntax of terms comes already defined with a built-in notion of type. In particular, variables always specify the type of terms over which they range, and abstractions always specify their domain:

$$t, u ::= x^A \mid \lambda x^A.t \mid t\ u$$

The Curry-style syntax is what we have seen so far: λ-terms are conceived before types, defined as an untyped syntax, and subsequently *assigned* a type. A consequence of this type-assignment approach is that the same Curry-style term may be assigned different types.

In addition to a notion of typed term, we can have a notion of typed conversion between terms of a certain type. We do so by defining *typed equality judgments*, $\Gamma \vdash a = b : A$, asserting that the terms a and b are convertible as terms of type A. Here are two examples, axiomatizing the β and ξ rules of the λ-calculus in a typed setting.

$$\frac{\Gamma, x : A \vdash t : B \qquad \Gamma \vdash s : A}{\Gamma \vdash (\lambda x.t)s = t[s/x] : B}\ (\beta) \qquad \frac{\Gamma, x : A \vdash t = s : B}{\Gamma \vdash \lambda x.t = \lambda x.s : A \to B}\ (\xi)$$

This kinds of judgments gives rise to a so called *definitional equality*, or *judgmental equality*, since it is axiomatized within the rules of the type system itself, rather than from a reduction relation assumed *a priori* on the untyped syntax.

2.1.2 Nameless representation

Syntax with binders is usually considered modulo α-renaming, that is, terms are identified with their α-equivalence classes. This tacit convention works fine within informal reasoning, but in computer formalizations it is important that terms have a unique representation. The usual solution is to employ a so-called *nameless representation*, in which binders do not name the variables they bind; rather, variables themselves uniquely determine the abstraction that binds them, so α-renaming holds by construction.

One way to implement nameless representation is with *De Bruijn indices* [27, 52]. In this representation, variable occurrences are given by indices, i.e. natural numbers indicating how many other binders separate the occurrence and its binder, counting outward from the variable to the outside levels. Hence, for example, the term $\lambda x.((\lambda y.x)(\lambda z.z))$ is represented with De Bruijn indices as $\lambda.(\lambda.1)(\lambda.0)$.

Another nameless scheme is given by *De Bruijn levels* [40]. *Levels* are assigned *to binders* according to the order in which they appear from outermost to innermost, and variables refer to their binders by their assigned level. For example, the term

$\lambda x.(\lambda y.x)(\lambda z.z)x$ is represented with levels as $\lambda.(\lambda.0)(\lambda.1)\ 0$. A pleasant property of this scheme is that every occurrence of a variable associated to some binder has the same exact representation no matter its position inside the term.

2.2 Martin-Löf Type Theory

A particular flavor of typed λ-calculus is Martin-Löf Type Theory (MLTT) [47, 49]. This theory includes four kinds of judgments:

- A is a type: $\Gamma \vdash A$;
- a is a term of type A: $\Gamma \vdash a : A$;
- A and B are equal types: $\Gamma \vdash A = B$;
- a and b are equal terms of type A: $\Gamma \vdash a = b : A$.

MLTT is a theory with dependent types, meaning that types can contain arbitrary terms. In fact, there is very little formal distinction between terms and types, and the two are usually defined in the same syntactic category. This explains the need for type and type equality judgments: since a type can contain arbitrary terms, its well-formedness, unlike STLC, is not obvious a priori, and must be established by a derivation. Moreover, types can contain reducible terms, so we must include definitional equality between types, that follows from the computation rules of the theory. A unique characteristic of dependent type theories is a coercion rule, expressing the fundamental link between typing and computation:

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash A = B}{\Gamma \vdash t : B}$$

A consequence of the coercion rule is that decidability of type checking crucially depends on decidability of definitional equality. The usual way to prove decidability of definitional equality is by proving a normalization theorem, stating that every well-typed term has a normal form. This allows to reduce definitional equality to syntactic identity of normal forms, which is trivially decidable.

Dependent type theories like MLTT include a generalized version of function type, called dependent function type or Π-type. The elements of a Π-type are functions whose codomain depends on the argument of the function. Thus, when a dependent function is applied, the result type is specialized to the particular argument:

$$\frac{\Gamma \vdash A \qquad \Gamma, x : A \vdash B}{\Gamma \vdash \Pi(x : A)B} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : \Pi(x : A)B} \qquad \frac{\Gamma \vdash f : \Pi(x : A)B \qquad \Gamma \vdash a : A}{\Gamma \vdash f\ a : B[a/x]}$$

We can generalize product types $A \times B$ in a similar way, and allow the type of the second component of a pair to depend on the first component. This is called a dependent pair type, or Σ-type. Elements of a Σ-type are *pairs* (a, b), where the type of the second component b may depend on the first component a.

$$\frac{\Gamma \vdash A \qquad \Gamma, x : A \vdash B}{\Gamma \vdash \Sigma(x : A)B} \qquad \frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B[a/x]}{\Gamma \vdash (a, b) : \Sigma(x : A)B}$$

$$\frac{\Gamma \vdash p : \Sigma(x : A)B}{\Gamma \vdash \pi_1\ p : A} \qquad \frac{\Gamma \vdash p : \Sigma(x : A)B}{\Gamma \vdash \pi_2\ p : B[\pi_1\ p/x]}$$

Notions of equality An crucial difference between type theory and set theory is in their treatment of equality. In common mathematics equality is a proposition, i.e. a binary relation between objects that can be proved or disproved using the rules of predicate logic. In type theory, equality is a type: given a type A and $a, b : A$, we can construct the *identity type* $a =_A b$. In type theory, proving an equality corresponds to constructing an inhabitant of the corresponding identity type. When $a =_A b$ is inhabited, we say that a and b are *propositionally equal*.

However, in type theory we have a second notion of equality, what we have called judgmental equality of definitional equality. Definitional equality, as the name suggests, is a metatheoretical concept that is part of the definition of the theory, and it is not *internal* to the theory. It is established by inference rules, which mostly correspond to conversion rules of the λ-calculus, and it is usually decidable. This careful distinction between computational equality, that is decidable, and propositional equality, that is undecidable, contributes to make Type Theory a convenient foundation for computer formalization of mathematics.

Universes Universes are types whose elements are themselves types, or, in certain formulations, *codes* standing for types. As in set theory, it is unsound to have a universe of types $\mathcal{U}$ be member of itself, namely $\mathcal{U} : \mathcal{U}$, as this leads to Girard's paradox and inconsistency. Instead, a hierarchy of universes is introduced:

$$\mathcal{U}_0 : \mathcal{U}_1 : \mathcal{U}_2 : ...$$

In the presence of a universe $\mathcal{U}$, types belonging to $\mathcal{U}$ are sometimes called *small types*. With a hierarchy of universes every type is "small", in the sense that every type belongs to some universe $\mathcal{U}_i$, thus the judgment A type is effectively subsumed by $A : \mathcal{U}_i$.

The power of type universes is really unveiled when we allow to define types by induction on elements of other types, something that is also called *large elimination*. With universes and large elimination we can define, for example, a function f of type $\mathbb{N} \to \mathcal{U}$ by recursion on $\mathbb{N}$, a family of types that assigns a type to every natural number.

2.2.1 Curry-Howard-Lambek correspondence

The *Curry-Howard-Lambek correspondence* is the idea that Type Theory can be seen an expressive formalism unifying logic, computer science, and category theory under the shared, fundamental notion of computation. According to the propositions-as-types paradigm, type theories are systems of formal logic, where propositions are identified with the type of their proofs, so that proving a proposition is nothing more than providing a term of the corresponding type. An essential property exhibited by most typed λ-calculi is normalization, which is important because it implies that certain types are empty, making the theory consistent as a logic. From this point of view, Type Theory can be used as a foundational system for mathematics, as an alternative to set theory. Under the so-called *proofs-as-programs* paradigm, Type Theory is an extremely expressive functional programming language, where programs, specifications of programs, and proofs of program correctness with respect to a specification can be expressed in the same language as first-class concepts. Finally, under the identification of types and terms with objects and morphisms of a category, type theories can be seen as domain-specific languages for reasoning about certain classes of categories. For example, extensional Martin-Löf type theories correspond to the internal languages of locally cartesian closed categories.

2.3 Minimalist Foundation

The Minimalist Foundation is a two-level foundation for constructive mathematics [43, 42]. Its two levels are given by dependent type theories in the style of Martin-Löf Type Theory, with the following characteristics:

- An *intensional* level, the Minimal Type Theory (mTT), defined as a theory in the style of intensional MLTT [49], and intended for program extraction from constructive proofs;
- An *extensional* level, defined as an extensional theory [47] with quotients, intended to support the day-to-day development of mathematics. The extensional level is interpreted in a quotient model built over mTT.

In the view of its authors [43], a necessary condition for the constructivity of a foundational theory is the support of a form of program extraction from constructive proofs, or *proofs-as-programs* paradigm. In practical terms, this property translates to the requirement of consistency with the axiom of choice (AC) and the formal Church thesis (CT).

$$\mathsf{AC} :\equiv \forall(x : A)\ \exists(y : B)\ (R\ x\ y) \to \exists(f : A \to B)\ \forall(x : A)\ (R\ x\ (f\ x))$$

$$\mathsf{CT} :\equiv \forall(f : \mathbb{N} \to \mathbb{N})\ \exists(e : \mathbb{N}).\forall(x : \mathbb{N})\ \exists(y : \mathbb{N})\ (\mathsf{T}\ e\ x\ y\ \wedge\ \mathsf{U}\ y = f\ x)$$

where T is the Kleene predicate and U is a function extracting the result of a terminating computation. MLTT is already known to be consistent with AC, as the result follows from the usual Brouwer–Heyting–Kolmogorov interpretation of intuitionistic connectives. One way to prove consistency with CT is to provide the theory with a Kleene realizability semantics [6]. Unfortunately, it is still an open problem whether such model construction is possible for intensional MLTT. mTT works around this by employing a particular definition of judgmental equality, that does not account for the usual congruence rules of term constructors, but instead replaces them with a primitive substitution rule like the following:

$$\frac{\Gamma, x : A \vdash t : B \qquad \Gamma \vdash a = b : A}{\Gamma \vdash t[a/x] = t[b/x] : B[a/x]}$$

The result is that certain extensionality principles, that would hold in the standard formulation, are invalidated. In particular, the ξ rule (i.e., the congruence rule for λ-abstractions) does not hold, and in general computation of terms under binders is limited. mTT formulated with the substitution rule above is known to support a Kleene realizability semantics [35].

Normalization The Minimal Type Theory can be interpreted into MLTT, by mapping propositions to MLTT types via the Curry-Howard, propositions-as-types interpretation. Since MLTT is normalizing [9], this translation entails normalization for mTT, in the following *negative* sense:

Theorem 2. There are no well-typed terms t in mTT such that every reduction sequence starting from t is infinite.

Proof. Suppose that such term exists. By mTT's interpretation into MLTT, t is well-typed in MLTT, and the infinite reduction starting from t also exists in MLTT, contradicting the fact that the theory is normalizing. □

Under classical reasoning, the theorem above implies normalization in the following *positive* sense:

Theorem 3. Assuming classical logic, Theorem 2 implies that for every well-typed term t, there exists a terminating reduction sequence starting from t.

Proof. From Theorem 2, we get that if t is well-typed, then it is not the case that t has no normal forms. By the classical principle of double negation elimination ($\neg\neg P \to P$), this means that a normal form exists. □

Constructively, only Theorem 2 holds. However, it is quite unsatisfying: what its statement tells us is that "it cannot be the case that mTT is not normalizing", which is not at all the same as telling us *why* it is normalizing, and *how*. That is, such a proof does not provide an algorithmic method to compute the normal form of an arbitrary well-typed term of mTT. Thus, besides philosophical reasons, a constructive proof of normalization has very pragmatic motivations: by the computational interpretation of constructive logic, a proof of normalization corresponds to showing that there exists an algorithmic method to evaluate well-typed terms to normal form. A concrete, computable normalization function is essential for type checking, and for computer implementations. It follows that classical proofs are irrelevant for our goals.

As pointed out in the introduction, constructive normalization for typed calculi with a CH-weak conversion, like mTT, does not seem to have been investigated much. For this reason, normalization for mTT is still an open problem.

2.4 Normalization by Evaluation

In more traditional treatments of typed λ-calculi, decidability of conversion is proved by establishing confluence and strong normalization for the small-step reduction relation (see, for example, [30]). This approach works fairly well for some λ-calculi, but does not scale easily to stronger relations, like η equality, or non-standard ones, like explicit substitutions [7]. In general, establishing normalization by repeated application of one-step reductions can be tedious and inefficient.

Normalization by Evaluation (NbE) is a semantic method to prove normalization for typed λ-calculi [8]. NbE exploits a model construction where the interpretation function is invertible by an operation called *reification*. The composition of interpretation and reification gives rise to a normalization function. In addition to the model construction, NbE requires to establish the following soundness and completeness properties of the normalization function nf:

- Completeness: if $\Gamma \vdash t = s : A$, then $\mathsf{nf}\ t \equiv \mathsf{nf}\ s$;
- Soundness: if $\Gamma \vdash t : A$, then $\Gamma \vdash t = \mathsf{nf}\ t : A$.

Soundness of NbE is usually established by a semantic argument based on logical relations, rather than the traditional syntactic reasoning using properties of the term rewriting system. From these properties, we get that convertibility is equivalent to syntactic identity of normal forms: $\Gamma \vdash t = s : A \iff \mathsf{nf}\ t \equiv \mathsf{nf}\ s$. Since identity of normal forms is decidable, so is convertibility.

2.5 Agda

Agda is a functional programming language and a proof assistant, implementing of a version of intensional Martin-Löf Type Theory [21]. It is equipped with powerful constructs such as a predicative hierarchy of universes, inductive-recursive definitions, and full dependent pattern matching. All mathematical material in this thesis has been formalized and proof-checked in Agda, and in fact, the following chapters use an “informal” version of Agda as the metatheory in which definitions and proofs are developed. Thus, since our metatheory is a constructive type theory, all proofs and functions given in the following chapters are total and computable by construction.

We now give a brief overview of the relevant aspects of Agda, as well as explain the notation that will be used for them throughout the thesis. We refer the reader to [21, 50] and the ufficial documentation [3] for more details.

The first kind of definition that Agda allows is that of an inductive definition. A type is inductively defined by enumerating its constructors, that basically correspond to introduction rules for that type. Here is the definition of natural numbers as an inductive type:

```
data ℕ : Set where
  zero : ℕ
  succ : ℕ → ℕ
```

Agda is equipped with an infinite hierarchy of universes $\mathsf{Set}_0, \mathsf{Set}_1, \mathsf{Set}_2, ...$, where $\mathsf{Set} \equiv \mathsf{Set}_0$. Every type is defined as a new element of its universe, like $\mathbb{N} : \mathsf{Set}$. The definition above can be written in a more familiar way with inference rules (where we only specify context extensions, leaving the other assumptions implicit). To stay more close to the common mathematical convention, we will use the rule-based notation to specify inductive definitions.

$$\frac{}{\mathbb{N} : \mathsf{Set}} \qquad \frac{}{\mathsf{zero} : \mathbb{N}} \qquad \frac{n : \mathbb{N}}{\mathsf{succ}\ n : \mathbb{N}}$$

The other way to define a symbol is to give a function definition. Functions are defined like in any other typed functional programming language, such as Haskell or ML. Functions can be defined by pattern matching, as in the following recursive definition of addition between natural numbers:

```
_+_ : ℕ → ℕ → ℕ
zero + m = m
succ n + m = succ (n + m)
```

Whenever a symbol is defined, underscores denote positions for arguments, as in _ + _ above. Being Agda a dependent type theory, it includes dependent product types, which are a built-in notion. A Π-type like $\Pi(x : A)B$ is expressed in Agda as $(x : A) \to B$. Agda inherits the propositions-as-types paradigm of intensional Martin-Löf Type Theory [49], representing logical propositions as types, and proofs as programs inhabiting the corresponding type. Thus, the type system of Agda provides a highly-expressive, multi-sorted, constructive higher-order logic. Under this interpretation, a Π type like $(x : A) \to B$ stands for the universal quantification $\forall x : A.B$. Implication $A \to B$ can be simply encoded as a Π type where B does not depend on the proof of A. We can also define the type Σ of dependent pairs that we have seen in MLTT, as an inductive record:

```
record Σ (A : Set) (P : A → Set) : Set where
  constructor _,_
  field
    π₁ : A
    π₂ : P π₁
```

Similarly to other programming languages, Agda records provide a convenient syntax for simultaneously defining types and providing named accessors to them.[1] In this case, the definition automatically provides accessors $\pi_1 : \Sigma\ A\ P \to A$ and $\pi_2 : (p : \Sigma\ A\ P) \to P(\pi_1\ p)$, as we would expect from a Σ type. In the rest of the thesis we will often write $\Sigma(x : A)P$ as syntax sugar for the Agda type $\Sigma\ A\ (\lambda x.P)$. Σ types can be used to encode

[1]Although Agda records are capable of much more than that. Every record gives rise to a (ML-style) module, so it can also encapsulate arbitrary definitions and proofs, depending on its fields, that are specialized and brought into scope when the record is instantiated and opened. In addition, Agda records support coinductive definitions, η rules, and can be used to encode typeclasses via implicits.

a strong form of intuitionistic existential quantification. In this sense, $\Sigma(x : A)P$ stands for the formula $\exists x : A.P$, whose proof terms are pairs containing a witness a of type A, and a proof term of type $P\ a$. As indicated by the constructor keyword, we use _ , _ as a short-hand syntax for providing pairs inhabiting a Σ type. As an example of use of Π and Σ in a logical statement, consider the following definition of a proof term for the *intensional* axiom of choice $(\forall x : A, \exists y : B.R(x, y)) \to \exists f : A \to B.\forall x : A, R(x, f(x))$ for a given binary relation R, which is a theorem in Type Theory [44]:

```
AC : { A B : Set } { R : A → B → Set }
    → ((x : A) → Σ B (λ y → R x y)) → Σ (A → B) (λ f → (x : A) → R x (f x))
AC h = (λ x → π₁ (h x)) , (λ x → π₂ (h x))
```

Inductive definitions are not limited to types, i.e. elements of some universe, but also allow to define *inductive families* of types [28]. Consider the definition of the type Vector A n of lists of elements in A of a fixed length n:

```
data Vector (A : Set) : ℕ → Set where
  nil : Vector A zero
  cons : {n : ℕ} → A → Vector A n → Vector A (succ n)
```

For every type A : Set, the piece of code above inductively defines a type family Vector $A : \mathbb{N} \to$ Set. In Vector A n , A is a *parameter* and it is fixed for every constructor, whereas n is an *index*, and it is specified on a per-constructor basis. We thus say that the family Vector A is indexed by elements in $\mathbb{N}$. The $\{n : \mathbb{N}\}$ notation in the cons constructor above stands for an implicit argument. The value of implicit arguments is automatically inferred by unification.

When the focus of an inductive definition is on the use of the type as a proposition, with its elements representing proof terms, we sometimes write down the constructors as labels of inference rules, like in the following inductive definition of $\leq$ between natural numbers:

```
data _≤_ : ℕ → ℕ → Set where
  zero-succ : ∀{n} → zero ≤ n
  succ-succ : ∀{n m} → n ≤ m → succ n ≤ succ m
```

$$\frac{n : \mathbb{N}}{\mathsf{zero} \leq n}\ (\text{zero-succ}) \qquad \frac{n \leq m}{\mathsf{succ}\ n \leq \mathsf{succ}\ m}\ (\text{succ-succ})$$

Agda supports dependent pattern matching. This means that pattern matching on elements of a dependent type may refine the information that is known on the shape of some other argument, making it evident in the match clause. Consider the following example:

```
data Maybe (A : Set) : Set where
  just : A → Maybe A
  nothing : Maybe A

maybeHead : {A : Set} → (n : ℕ) → Vector A n → Maybe A
maybeHead .zero nil = nothing
maybeHead .(succ _) (cons x v) = just x
```

Here, we define maybeHead by pattern matching on the argument of type Vector A n, and by doing so, we reveal additional information on the shape of another argument, n. In fact, when the vector is empty, the index must be zero by definition of Vector itself, thus n is automatically refined to the only possible value, zero (the dot in front of a pattern means that it is forcibly implied by other patterns to have that and only that shape). Similarly, when the vector is built with cons, Agda knows that its length is at least 1, i.e. n must be the successor of some other natural number, so it refines n's shape accordingly.

Agda supports induction-recursion [29], a definition scheme allowing the simultaneous definition of an inductive type and a recursive function on elements of that type. Induction-recursion allows definitions that are much more difficult to give in a theory with only inductive types, if not impossible. Universes in dependent type theories are a typical example of inductive-recursive definitions: we have an inductive type U of *codes*, and a function T : U → Set out of U, giving interpretation of codes in U. The following is an Agda example of a basic universe with codes for natural numbers and Π types.

```
mutual
  data U : Set where
    N : U
    Pi : (A : U) → (El A → U) → U

  El : U → Set
  El N = ℕ
  El (Pi A B) = (x : El A) → El (B x)
```

The identity type can be found in the Agda standard library with the symbol ≡, and is defined as an inductive family given by a single constructor refl.

```
data _≡_ {A : Set} : A → A → Set where
  refl : {a : A} → a ≡ a
```

Agda allows, as with any other type, pattern matching on elements of the identity type, making it easier to write certain proofs involving equations. Here is a proof of the fact that _ ≡ _ is a congruence w.r.t. any function f.

```
cong : {A B : Set} {a b : A} → (f : A → B) → a ≡ b → f a ≡ f b
cong f refl = refl
```

The drawback of allowing unrestricted dependent pattern matching on equality is that it implies Streicher's Axiom K [48], also known as "uniqueness of identity proofs"

(UIP), which states that all proofs of proposional equality are propositionally equal to refl. This is a problem when trying to use Agda for theories in which UIP does not hold (such as Homotopy Type Theory), but it shall not a problem in the context of this thesis.

One aspect of Agda that does not carry over to this document is its somewhat confusing notation about propositional equality. The $=$ sign is a reserved symbol in Agda, and it is used in function definitions. Equations introduced with $=$ are, therefore, definitional. Propositional equality is expressed with the aforementioned identity type, which is defined with a symbol, $\equiv$, that usually denotes definitional equality or even syntactic identity in most other settings. In the rest of the thesis, we will use $=$ for propositional equality, and $\equiv$ or $:\equiv$ for definitional equality, with the exception of the Agda syntax for function definitions, that will retain $=$ in its definitional role.[2]

[2]This is unfortunately due to the inability to change the $=$ symbol in the typesetting of Agda definitions, that are automatically generated from the source code.

Chapter 3

Weak notions of conversion

The standard equational theory of the λ-calculus comprises, in addition to the usual β rule, several congruence rules, like the ξ rule. However, in some cases weaker theories are worth consideration.[1] One reason is that they have more models. Another is that simpler notions of reductions make certain rewriting properties easier to prove and reason about. In this chapter, we look at three weak notions of reduction of λ-terms, starting from the well-known *weak-head reduction*, then quickly moving on to define and study in detail a reduction relation that we call *CH-weak reduction*, and ending the chapter with an analysis of *weak explicit substitutions.* These last two constitute the subject of the rest of this thesis. The main contribution of this chapter is a recursive normalization algorithm for a particular syntactic variant of the untyped λ-calculus with CH-weak reduction, that gives the basis for the type systems introduced Chapter 4.

3.1 Weak-head reduction

Weak-head reduction can be described informally as a variation of β-reduction that never performs reductions under λ-abstractions. Formally, weak-head reduction is just standard β-reduction without the ξ rule. The notion of reduction that results corresponds to the usual process of computation in programming languages, where functions are not evaluated without their arguments fully provided [31].

$$\frac{}{(\lambda x.t)s \longrightarrow t[s/x]}\ (\beta) \qquad\qquad \frac{t \longrightarrow r}{t\ s \longrightarrow r\ s}\ (\nu) \qquad\qquad \frac{s \longrightarrow r}{t\ s \longrightarrow t\ r}\ (\mu)$$

For example, the term $\lambda x.(\lambda y.y)z$ that would be reducible under full β-reduction, is a weak-head normal form, whereas $(\lambda x.(\lambda y.y)z)w$ reduces to $(\lambda y.y)z$ and then to z. Unfortunately, weak-head reduction is not confluent. Indeed, suppose we have $N \longrightarrow N'$. Then

[1] We say ‘weak’ in the sense that less equations hold.

$$\begin{tikzcd}
(\lambda x.\lambda y.M)N \arrow[r] \arrow[d] & (\lambda x.\lambda y.M)N' \arrow[d] \\
\lambda y.M[N/x] & \lambda y.M[N'/x]
\end{tikzcd}$$

The term $\lambda y.M[N/x]$ is a normal form, so there is no way to make the diagram above commute. A known solution to this problem consists of extending weak-head reduction with a substitution rule. Capture-avoiding substitution allows a restricted class of contractions to be performed under λs. The result is a new notion of weak reduction, described in the next section, that can be seen as a well-behaved version of weak-head reduction.

3.2 Çağman-Hindley Weak Reduction

The Church-Rosser property, that fails to hold in weak-head reduction, can be recovered by extending the reduction relation with a substitution rule (called σ in [41]):

$$\frac{N \longrightarrow N'}{M[N/x] \longrightarrow M[N'/x]}\ (\sigma)$$

This rule enables a limited form of computation under binders. For example, the term $\lambda x.(\lambda y.y)z$, which is a normal form under weak-head reduction, reduces to $\lambda x.z$ with the addition of the σ rule, as follows

$$\frac{\overline{(\lambda y.y)z \longrightarrow y[z/y] \equiv z}}{(\lambda x.w)[(\lambda y.y)z/w] \longrightarrow (\lambda x.w)[z/w]}$$

However, the term $\lambda x.(\lambda y.y)x$, which would reduce to $\lambda x.x$ under full β-reduction, is a normal form in this setting. The reason is that reduction under λs is only allowed via capture-avoiding substitution, but there is no way, for any M, to obtain $\lambda x.M$ from $\lambda x.w$ and a substitution $[M/w]$, if M contains x free.

Nevertheless, the σ rule is enough to "fix" the broken example of the previous section, and in general, to allow confluence to be proved. Let us call $\longrightarrow_w$ the reduction relation given by weak-head reduction plus the substitution rule σ.

$$\frac{}{(\lambda x.t)s \longrightarrow_w t[s/x]}\ (\beta) \qquad \frac{t \longrightarrow_w r}{t\ s \longrightarrow_w r\ s}\ (\nu) \qquad \frac{s \longrightarrow_w r}{t\ s \longrightarrow_w t\ r}\ (\mu)$$

$$\frac{N \longrightarrow_w N'}{M[N/x] \longrightarrow_w M[N'/x]}\ (\sigma)$$

Then, we have the following result

Theorem 4. The reduction relation $\longrightarrow_w$ is confluent.

Proof. See [41], Theorem 1. □

Despite being an improvement over plain weak-head reduction, the relation $\longrightarrow_w$ has received limited attention in the literature. One of the first uses of it in a weak λ-calculus can be found in [32], where Howard defines it under the name of "restricted reduction", and uses it in a strong normalization argument. This flavour of weak reduction was later revisited by Çağman and Hindley [22], who call it *combinatory weak reduction.* In their analysis it is shown, as the name suggests, that combinatory weak reduction precisely represents the λ-calculus analogous of reduction in combinatory logic. As argued in [22], there are several reasons why one may wish to modify full β-reduction to make it correspond more closely to combinatory logic. One is to have a simpler rewriting system: full β-reduction is powerful, but also more complex than that of combinatory logic. As an example, properties like Church-Rosser or the standardization theorem are easier to prove in combinatory logic. In [22], it is shown how a combinatory weak relation can be used to benefit from the advantages of a simpler, combinatory-like reduction relation, while retaining the conveniences of the λ-syntax.

Çağman and Hindley's definition of weak reduction is reproduced in Definition 1. From now on, we call this relation *CH-weak reduction* to avoid confusion with other weak notions.

Definition 1. [CH-weak reduction] Let the redex R be a subterm of a term P. Then, R is a *weak redex* iff it does not contain free variables that are bound in P. A one-step CH-weak reduction of P, written $\longrightarrow_{ch}$, is one that contracts a weak redex inside P.

For example, the term $P \equiv \lambda x.(\lambda y.y)z$ admits one step of CH-weak reduction by a contraction of the weak redex $(\lambda y.y)z$, after which it becomes the normal form $\lambda x.z$. Conversely, the term $P \equiv \lambda x.(\lambda y.x)z$ is already in normal form, because the redex $(\lambda y.x)z$ contains a free variable x that is bound in P, hence it is not weak.

In [22], it is shown that the two weak reductions, namely $\longrightarrow_w$ and $\longrightarrow_{ch}$, are indeed equivalent, and thus represent the same relation. We reproduce the proof of this result below, since an important part of the following chapter will be about revisiting it in a typed setting.

Let us first define contexts in the λ-calculus. A λ-context C is a regular term with *holes* $\bullet$ in it:

$$C ::= x \mid \bullet \mid C\ C \mid \lambda x.C$$

We write $C[M_1, M_2, ..., M_n]$ for a context C with n holes, filled by the terms $M_1, ..., M_n$. We also write $C[\ ]$ to denote a context with a single hole. Note that hole-filling does *not* correspond to substitution. In particular, hole-filling can capture free variables.

Lemma 1. If $C[\ ]$ is a context that does not bind any free variables in N, and x does not occur in $C[\ ]$, then $C[x][N/x] \equiv C[N]$.

Proof. See [22], Lemma 4.2. □

Theorem 5. The reduction relations $\longrightarrow_w$ and $\longrightarrow_{cw}$ are equivalent:

$$p \longrightarrow_w Q \iff p \longrightarrow_{ch} q$$

Proof.

($\Longrightarrow$) By induction on the size of the terms involved. All rules of $\longrightarrow_w$ except for the substitution rule are present in $\longrightarrow_{ch}$. We can then restrict our attention to substitutions, and consider a reduction of the form

$$\frac{N \longrightarrow_w N'}{M[N/x] \longrightarrow_w M[N'/x]}\ (\sigma)$$

By inductive hypothesis, $N \longrightarrow_{ch} N'$ by a contraction of a weak redexe. But being substitution capture-avoiding, every redex that is weak in a term N is also weak in the term $M[N/x]$ for any M, hence $M[N/x] \longrightarrow_{ch} M[N'/x]$ by contraction of the same weak redex.

($\Longleftarrow$) We must show that if P reduces to Q by a single contraction of a weak redex, then $P \longrightarrow_w Q$. Let R be such weak redex, and R' the result of its contraction. By inductive hypothesis, $R \longrightarrow_w R'$. Since R is a subterm of P, we have $P \equiv C[R]$ for some context C, that, being R weak, does not capture any of its free variables. But similarly, $Q \equiv C[R']$, since R is the only subterm that changes in the reduction from P to Q. By Lemma 1, we have that $C[R] \equiv C[x][R/x]$ and $C[R'] \equiv C[x][R'/x]$, therefore

$$\frac{R \longrightarrow_w R'}{C[x][R/x] \longrightarrow_w C[x][R'/x]}\ (\sigma)$$

that concludes the proof, since $P \equiv C[x][R/x]$ and $Q \equiv C[x][R'/x]$.

□

A key role in the proof above is played by Lemma 1: it shows us how weak redexes of a term P can be "pulled out" of it, leaving a placeholder variable in their place. This allows us to *factor* a weak contraction into two components, namely the weak redex involved R, and the "rest" of P that does not change in the contraction, namely $C[\ ]$. These two pieces can then be reassembled via substitution.

The equivalence between these two formulations of CH-weak reduction is at the core of the technique employed in Chapter 4 and 5 to prove normalization for a typed calculus with CH-weak conversion. We will see that one formulation gives rise to a more compact set of rules, so it is ideal for defining a type theory but not to study its metatheory, whereas the other is less compact but more amenable to a (formalized) metatheoretical analysis. The correspondence between the two ensures that we can rely on the second formulation to prove our properties of interest, because these can be shown to hold for the first one also.

3.2.1 Weak formulation of the ξ rule

What makes CH-weak reduction particularly subtle is the fact that, despite the ξ rule not being validated, some limited form of computation under binders is still allowed. In fact, CH-weak reduction does validate a restricted form of ξ rule, by which some reductions between λ-abstractions can be expressed by reductions on their bodies. Suppose we had terms t and s such that $t \longrightarrow_{ch} s$. Then, there must be a weak constraction $a \rightsquigarrow b$ such that $C[a] \equiv t$ and $C[b] \equiv s$, for some context $C[\,]$. Suppose x is not free in a and b; then, a is also weak in $\lambda x.C[\,]$, from which we can conclude $\lambda x.C[a] \longrightarrow_{ch} \lambda x.C[b]$, that is just $\lambda x.t \longrightarrow_{ch} \lambda x.s$.

The intuition is that to CH-weakly reduce a λ-abstraction $\lambda x.t$, it is possible to proceed by structural recursion on the term and CH-weakly reduce its body t, as long as the abstraction λx does not bind variables involved in any of the contractions performed to reduce t. Notice that the following restricted ξ rule, that also holds under CH-weak reduction, does not capture the this intuition, as it is too restrictive:

$$\frac{t \longrightarrow s \qquad x \notin (\mathrm{FV}(t) \cup \mathrm{FV}(s))}{\lambda x.t \longrightarrow \lambda x.s}$$

In fact, the rule above does not allow the reduction $\lambda x.(\lambda y.y)zx \longrightarrow \lambda x.zx$, which *is* valid under CH-weak reduction. A possible formulation that does capture our intuition could be the following:

$$\frac{t \longrightarrow_{ch} s \qquad x \notin FV(\mathrm{rdx}(t \longrightarrow s))}{\lambda x.t \longrightarrow_{ch} \lambda x.s}$$

where we can imagine $\mathrm{rdx}(t \longrightarrow s)$ to be the weak redex that gets contracted in the given reduction step. This rule, however, is awkward: keeping track of the side conditions is cumbersome even in an informal setting, and more so in a computer formalization. In addition, the rule also *feels* wrong: we should be able to express CH-weak reduction in a direct way, making contractions of non-weak redexes unrepresentable rather than ruling them our after the fact. We will see in the next section, as well as in the chapters that follow how to devise a syntax for both λ-terms and definitional equality judgments where CH-weak reduction can be expressed directly, including the ξ rule above.

3.3 CH-weak normalization

One thing that can be done with a reduction relation is to build an evaluator for it. By it we mean a function that, given a term, produces its normal form if one exists.[2] [3] The standard β-conversion relation is a congruence relation, so a recipe for reducing terms is already implicitly provided in the definition itself. Consider, for example, the congruence rules for abstraction and application:

[2]A normal form may not exist, so our evaluator is necessarily a partial function.

[3]"Evaluation" is usually considered w.r.t. programs, i.e. closed term. Here we use the word in a broader sense to also include open terms, as synonymous of "normalization".

$$\frac{M =_\beta N}{\lambda x.M =_\beta \lambda x.N}\ \xi \qquad \frac{M =_\beta N \qquad P =_\beta Q}{M\ P =_\beta N\ Q}$$

What the ξ rule suggests is that, in order to reduce a λ-abstraction, one has to proceed recursively on its body. Similarly, to reduce an application, we can recursively evaluate its subterms. In general, with a congruent conversion relation, reducing a term can be done by inspecting its outermost constructor, and then operating recursively on its immediate subterms, performing a contraction when a redex is found. Full β-conversion leads to a fairly simple definition of evaluation by structural recursion on terms.

CH-weak reduction, instead, presents some complications: it is not possible to evaluate a λ-abstraction by recursion on its body, because ξ does not hold in general. Nevertheless, as we have seen, *some* contractions are still possible under λ-abstractions, and they must be taken into account. In order to build a normalization function, we need to know how and in which cases to proceed by recursion under binders, and what terms should be treated as reducible terms (i.e., redexes). In this respect, both definitions of CH-weak reduction that we have seen in Section 3.2 are not very helpful, because they make precise *what* counts as a valid reduction, but not *how* to mechanically perform one.

Definition 1 at least tells us what we should look for during normalization, i.e. *weak redexes*. We could implement evaluation by recursively traversing the term, contracting every weak redex that is encountered. Recognizing a weak redex, however, is a non-trivial task, given their *relative* nature: whether a subredex R of a term P is weak depends on the syntactic structure of P.

We observe that to recognize a subterm as a weak redex it suffices to be able to distinguish, among its free variables, between those that are free everywhere, and those that are bound somewhere in the enclosing term P, *regardless* of what P actually is. We call the first *global* variables, and the latter *local* variables. Under this distinction, a weak redex is easily identifiable as one that is closed w.r.t. local variables, or equivalently, one in which all free variables are global.

If we base the syntax of our λ-terms on a nameless representation of De Bruijn indices, then a way to distinguish between global and local variables during evaluation is to index the evaluation function itself with an additional argument, representing the number of binders that have been crossed during recursion. Suppose n is such a number; then, every index $\geq n$ is a global variable, whereas every other is local. A weak redex is thus one in which all free De Bruijn indices are $\geq n$, i.e. global.

We can implement this indexed CH-weak evaluator for De Bruijn terms in a few lines of Haskell [33]:

```
data Term = Idx Int | App Term Term | Lam Term
type Subst = [Term]

sz :: Int -> Int -> Term -> Bool
sz x y (Idx k) = y <= k || k < x
sz x y (Lam t) = sz (succ x) (succ y) t
sz x y (App t s) = sz x y t && sz x y s

weak :: Int -> Term -> Bool
weak i t = sz 0 i t

idxeval :: Int -> Term -> Subst -> Term
idxeval n (Idx i) s = s !! i
idxeval n (Lam t) s = Lam (idxeval (succ n) t (Idx 0 : shift s))
idxeval n (App t u) s = case idxeval n t s of
  Lam t' | weak n t && weak n u -> eval t' (idxeval n u s : s)
         | otherwise -> App (Lam t') (idxeval n u s)
  t' -> App t' (idxeval n u s)
```

In the code above, terms are either De Bruijn indices `Idx n` for some index `n`, applications `App t s` for terms `t` and `s`, or abstractions `Lam t` for a body `t`. As usual with De Bruijn indices, `Lam t` is assumed to bind the shallowest index in `t`. `sz x y t` returns `True` iff all indices in `t` are either $\geq y$ or $< x$. Thus, a redex `t` is weak under `i` binders iff `sz 0 i t`.[4]

With an evaluation function like `idxeval`, a term P can then be CH-weakly evaluated to normal form by calling the evaluation function on P with an initial index of 0, meaning that no binders have been crossed and all free variables should be considered global. This solution is good enough is we limit ourselves to simple evaluation of untyped terms, but it does not scale well when using it to formalize an entire normalization proof for a full-scale typed calculus. In fact, every aspect of the proof, from definitions to functions to proofs would have to be indexed, making the formalization much more difficult to understand and manage.

A simpler solution comes from the following observation: to CH-weakly evaluate a term P, we only need to be able to classify its variables into the two possible *roles*, that of a global or a local variable. But the role of each variable in P depends uniquely on the syntactic structure of P itself, and has nothing to do with the particular state of evaluation that we are in. Moreover, evaluating a term *does not* alter any of its variables' role. Notice, in fact, that the only operation that affects terms during evaluation is β-contraction of a weak redex, which boils down to substitution. Some variables may disappear from the term as a result of substitution, but the ones that don't never change their role. This means that the role of each variable in P is fully known before evaluation, and can be assigned ahead-of-time. An indexed evaluation function like `idxeval` enables us to recompute the role of variables in a redex during evaluation, but this is not needed, because their role never changes.

We make this idea precise, saying that a variable x *has a global role* in a term t if it appears free in t. Conversely, we say that x *has a local role* in t if it appears in t, but it is bound somewhere in it.

[4]`shift :: Subst -> Subst` is just an auxiliary function that shifts all indices of a substitution to avoid capture of free De Bruijn indices, and whose definition is not relevant at this point.

Proposition 1. Let $t \longrightarrow_{ch} s$. If x has role r in t, then either it has role r in s, or it is not present in s at all.

Proof. Let $(\lambda y.M)N \rightsquigarrow M[N/y]$ be the weak contraction involved in $t \longrightarrow_{ch} s$. It is sufficient to show that if x appears in $(\lambda y.M)N$ with some role, then all of its occurrences in $M[N/y]$ have the same role.

Suppose x is global in $(\lambda y.M)N$. Since $FV((\lambda y.M)N) \supseteq FV(M[N/y])$, and x is not bound in $M[N/y]$, every occurrence in $M[N/y]$ must be global.

If x has a local role in the redex, and $x \not\equiv y$, then it appears bound in both terms, so its role is not affected by the substitution. Otherwise, if $x \equiv y$, then when substituting all occurrences of x in M are replaced by N, which either contains x in a local role, or no occurrences of it. □

In what follows we say that a variable x is "locally-free" in a subterm N of M if it has a local role in M, and it is free in N.

Proposition 2. A redex subterm a of t is weak iff it is closed w.r.t. local variables.

Proof. If a redex is weak, then it does not contain free variables that are bound within t. But then, it cannot contain local variables, that are bound in t by definition. Conversely, if a redex does not contain locally-free variables, then all its free variables are also free in t, hence they cannot be bound within it. This makes the redex a weak redex. □

A term can thus be CH-weakly reduced by "tagging" its variables according to their roles, and then reducing it by recursively traversing its structure, and contracting all and only those redexes that are weak, i.e. closed w.r.t. local variables. Since variables with a local role would be syntactically distinguished from those with a global role, the notion of weak redex in this tagged syntax becomes absolute, i.e. it only depends on the syntactic structure of subterms.

Let us define an "explicit" syntax that distinguishes between variables with a global and a local role.

```
data ETerm = Local Int | Global Int | EApp Term Term | ELam Term
```

We can define a `tag` function that tags variables of "implicit" terms, transforming them into "explicit" terms.

```
tag :: Int -> Term -> ETerm
tag n (Idx x) = if n <= x then Global x else Local x
tag n (Lam t) = ELam (tag (succ n) t)
tag n (App t s) = EApp (tag n t) (tag n s)

untag :: ETerm -> Term
untag (Global x) = Idx x
untag (Local x) = Idx x
untag (ELam t) = Lam (untag t)
untag (EApp t s) = App (untag t) (untag s)
```

We can now just implement a function `sz` such that `sz n t` is true whenever all free indices of t are $< n$. Then, a term closed w.r.t. local variables is one for which `sz 0 t` holds.

```
sz :: Int -> ETerm -> Bool
sz _ (Global _) = True
sz x (Local k) = k < x
sz x (ELam t) = sz (succ x) t
sz x (EApp t s) = sz x t && sz x s

weak :: ETerm -> Bool
weak = sz 0
```

CH-weak evaluation then becomes the following:

```
type Subst = [ETerm]

eval :: ETerm -> Subst -> ETerm
eval (Global x) _ = Global x
eval (Local i) s = s !! i
eval (ELam t) s = ELam (eval t (Local 0 : fmap (shift 1 0) s))
eval (EApp t u) s = case eval t s of
  ELam t’ | weak t && weak u -> eval t’ (eval u s : s)
          | otherwise -> EApp (Lam t’) (eval u s)
  t’ -> EApp t’ (eval u s)

eval’ :: Term -> Term
eval’ = untag . flip eval [] . tag 0
```

The solution above separates the process of role classification and that of evaluation, allowing to express both in a cleaner and simpler way. The issue of dealing with the relative aspect of CH-weak reduction is relegated to where it belongs—in the tagging function—and moved *out* of evaluation, that can be expressed in a standard, structurally recursive fashion.

Another interesting way to look at this is that, whenever we call the tagging function on a term P to be CH-weakly evaluated, we are effectively generating an entire syntax of explicit terms specifically tailored to P itself. On such syntax, evaluation can be defined smoothly since all information on P and its weak redexes is already encoded in the structure of terms. One could say that we made the notion of weak redex absolute by making the whole syntax relative. Indeed, redexes in CH-weak reduction are intrinsically relative, and we did not eliminate that aspect at all. Rather, we made it explicit, and used it to our advantage.

3.4 Weak explicit substitutions

Substitution is a central element of the λ-calculus, as β-contraction is defined in terms of it. However, substitution is usually expressed informally as a meta-level operation, and it is assumed to be fully applied every time it appears in a reduction. This is not the case in most implementations of the λ-calculus, for practical reasons. In fact, if the term M contains many occurrences of the free variable x, then the substitution $M[N/x]$ produces a term with many copies of N, leading to potential size explosion.

In practical implementation, substitutions are usually delayed and recorded in terms. In an attempt to bridge the gap between informal treatments of the λ-calculus and its implementations, Abadi et al. developed the $\lambda\sigma$-calculus, a refinement of the λ-calculus with *explicit substitutions* [7]. In this setting, substitutions s have a syntactic representation: given a term M and a substitution s, the syntax $t[s]$ represents a valid term, i.e. t with the explicit, delayed substitution s, rather than a metalinguistic operation. We can then express β-contraction as follows:

$$(\lambda x.M)N \longrightarrow M[(x, N)]$$

where (x, N) is syntax for the substitution that replaces x with N, and leaves the other variables unchanged.

In [41], Lévy and Maranget develop a weak λ-calculus with explicit substitutions, defined as follows:

$$M, N ::= x \mid MN \mid (\lambda x.P)[s] \qquad P, Q ::= x \mid PQ \mid \lambda x.P$$

$$s ::= (x_1, M_1), (x_2, M_2), \ldots, (x_n, M_n) \qquad x_i \text{ distinct } (n \geq 0)$$

Here, we use the metavariables P, Q for *programs*, i.e. constant terms, and M, N for ordinary *terms*. s denotes explicit substitutions, which are just lists variable-term pairs. Terms are formed out of variables, application, and *closures* $(\lambda x.P)[s]$, i.e. pairs made of a functional program $\lambda x.P$ and a substitution s assigning terms to its free variables. Notice that here, unlike in the $\lambda\sigma$-calculus, we only use explicit substitutions for λ-abstractions.

The dynamics of weak explicit substitutions is given as follows. As we can see, closures are needed because we never push substitutions under λ-abstractions, since otherwise we would validate the ξ rule.

$$\frac{}{(\lambda x.P)[s]\ N \longrightarrow P[\![s[N/x]]\!]}\ (\beta) \qquad \frac{s \longrightarrow s'}{(\lambda x.P)[s] \longrightarrow (\lambda x.P)[s']}\ (\xi)$$

$$\frac{M \longrightarrow M'}{MN \longrightarrow M'N}\ (\nu) \qquad \frac{N \longrightarrow N'}{MN \longrightarrow MN'}\ (\mu)$$

$$\frac{s(x) \longrightarrow M' \qquad s' \equiv s[M'/x]}{s \longrightarrow s'}$$

where $[\![\_]\!]$ is a metatheoretical operation that applies a substitution to a program until an abstraction is reached:

$$\begin{aligned} x\ [\![s]\!] &\equiv s(x) \\ (P\ Q)[\![s]\!] &\equiv P[\![s]\!]\ Q[\![s]\!] \\ (\lambda x.P)[\![s]\!] &\equiv (\lambda x.P)[s] \end{aligned}$$

We can easily translate from explicit substitutions to the standard weak λ-calculus. Let us consider the operation $\{\_\}$, defined as follows:

$$\begin{aligned} &\{x\} = x \\ &\{MN\} = \{M\}\{N\} \\ &\{(\lambda x.P)[s]\} = (\lambda x.P)[\{s\}] \\ &\{(x_1, M_1), \ldots, (x_n, M_n)\} = \{M_1\}/x_1, \ldots, \{M_n\}/x_n \end{aligned}$$

It can then be shown that reductions in the calculus of explicit substitutions translate to CH-weak reductions in the standard calculus:

Proposition 3. If $M \longrightarrow M'$ then $\{M\} \longrightarrow_w^* \{M'\}$.

Proof. See [41]. □

Going from the standard calculus to explicit substitutions is more difficult. In [41], the authors consider a translation via maximal free subterms. A subterm Q of a term P is free whenever $P \equiv C[Q]$ for some context $C[\_]$ that does not bind any free variable in Q. A free subterm is maximal whenever it is not a subterm of another free subterm. We define the translation operation $\mathcal{I}$ as in [41]:

$$\begin{aligned} &\mathcal{I}(x) = x \\ &\mathcal{I}(PQ) = \mathcal{I}(P)\mathcal{I}(Q) \\ &\mathcal{I}(\lambda x.P) = (\lambda x.C[x_1, \ldots, x_n])[(x_1, \mathcal{I}(P_1)), \ldots, (x_n, \mathcal{I}(P_n))] \end{aligned}$$

where $P_1, \ldots, P_n$ are the maximal free subterms of P. Notice that maximal free subterms of a term are all disjoint, so that we can use a generalized multi-hole context notation $C[\_, \ldots, \_]$ as above. We can show that a reduction in the standard calculus translates to a reduction with explicit substitutions

Proposition 4. If $P \longrightarrow_w P'$, then $\mathcal{I}(P) \longrightarrow M$, with $\{M\} = P'$.

Unfortunately, this result does not generalize to the reflexive-transitive closure: it is not the case that $P \longrightarrow_w^* P'$ implies $\mathcal{I}(P) \longrightarrow^* M$ with $\{M\} = P'$. Consider the following reduction of a term P in the standard calculus:

$$P \equiv (\lambda x.\lambda y.x(\lambda z.z))(\lambda w.w) \longrightarrow_w \lambda y.(\lambda w.w)(\lambda z.z) \longrightarrow_w \lambda y.\lambda z.z$$

P's translation $\mathcal{I}(P) \equiv (\lambda x.\lambda y.xk)[(\lambda z.z)[]/k]((\lambda w.w)[])$ with weak explicit substitutions can be reduced as follows:

$$(\lambda x.\lambda y.xk)[(\lambda z.z)[]/k]((\lambda w.w)[]) \longrightarrow (\lambda y.xk)[(\lambda z.z)[]/k,(\lambda w.w)[]/x] \equiv M$$

The term M is a normal form in the calculus of weak explicit substitutions, but translates to the reducible term $\{M\} \equiv \lambda y.(\lambda w.w)(\lambda z.z)$ in the standard calculus. The problem is that the property of Proposition 4 only holds for terms that are images of the translation function $\mathcal{I}$, i.e. those terms M such that their explicit substitutions only contain maximal free subterms in $\{M\}$. This property ensures that every weak redex in $\{M\}$ also appears as a redex in M, so that every possible one-step reduction has a counterpart in the other calculus. Unfortunately, β-contraction destroys this maximality property, as shown in the example above, because it may create new free subterms, and thus new (weak) redexes that did not exist at the moment of translation from the standard syntax to the explicit one. This also seems to suggest that the problem may not be necessarily related to this particular "maximal" translation $\mathcal{I}$, but could be inherent in the structure of the weak explicit substitution calculus.

Why should we care if weak explicit substitutions are not equivalent to CH-weak reduction? The problem is that we cannot use the explicit calculus to reduce standard weak terms to CH-weak normal form. An advantage of the explicit substitution calculus is that it supports an evaluation procedure that is easy to define and formalize, and that can also be implemented more efficiently that the evaluator of the previous section. However, since Proposition 4 does not extend to multi-step reductions, normal forms in the explicit substitution calculus do not necessarily translate to normal forms in the standard weak λ-calculus.

Nevertheless, weak explicit substitutions are still very interesting, since they share many of the positive properties of the standard CH-weak λ-calculus, like confluence [41] and admissible substitution rule, while avoiding some of the drawbacks, like a normalization procedure complicated by the relative notion of weak redex. We can further generalize the calculus above to avoid the distinction between "terms" and "programs", using explicit substitutions everywhere instead of only on λ-abstractions. An example is the *weak $\lambda\sigma$-calculus* of [**?**]:

$$M, N ::= x \mid MN \mid \lambda x.M \mid M[s] \qquad s ::= \mathrm{id} \mid (x, M), s \mid s_1 \cdot s_2$$

Substitutions s are built from identity substitutions and list "cons-ing", as before, but now we also have a composition constructor $s_1 \cdot s_2$ encoding the effect of the substitution s_1 followed by that of s_2. This is needed to define the reduction relation in the case where terms are applied to several explicit substitutions: $t[s_1][s_2] \longrightarrow t[s_1 \cdot s_2]$. In addition, we have reduction rules to "resolve" composite substitutions, like associativity $(s_1 \cdot s_2) \cdot s_3 \longrightarrow s_1 \cdot (s_2 \cdot s_3)$ and distributivity $((x, M), s_1) \cdot s_2 \longrightarrow (x, M[s_2]), (s_1 \cdot s_2)$. We do not delve into the details of this calculus now. We will revisit it instead in Chapter 6, where we will use it in a dependently-typed λ-calculus, and specifically a version of Martin-Löf Type Theory with nameless syntax, full dependent types, and weak explicit substitutions.

Chapter 4

System T with weak equality judgments

This chapter and the one that follows present the first main contribution of the thesis, that is a fully-formalized, constructive proof of normalization for a version of System T, called System T^{wk}, whose equality judgments correspond to a typed CH-weak conversion relation. We start by introducing the term syntax and the type system of System T^{wk}. We then realize that there are aspects of T^{wk}'s definitional equality that complicate reasoning about it, and in particular that prevent using the normalization procedure for CH-weak reduction developed in Section 3.3.

We solve these problems by constructing an alternative formulation of System T^{wk}, called System T^{ex}, that has explicit, syntactic support for expressing CH-weak conversion, and is thus more amenable to metamathematical analysis. We conclude by showing that we can establish a precise correspondence between System T^{wk} and T^{ex}, a result that will be crucial in the proof of normalization for System T^{wk}.

All the contents of this chapter can be found in the formalization [5] under the directory `Syntax/`.

4.1 System T^{wk}

We consider System T^{wk}, a variant of Gödel System T, a typed λ-calculus with natural numbers, primitive recursion, and higher-order functions. The calculus is "weak" in the sense that is comes with typed equality judgments axiomatizing CH-weak conversion. Since this thesis was motivated by the problem of normalization for mTT, System T^{wk}'s formulation of CH-weak equality judgments tries to be as close to mTT's definition as possible. In particular, substitution is a total, computable operation in the metalanguage, rather that a construct of the object syntax (that is, we do not employ explicit substitutions); moreover, CH-weak conversion is formulated with a primitive substitution rule in place of congruence rules, rather than relying on a notion of weak redex. For this reasons, System T^{wk} can also be seen as a propositional fragment of mTT.

4.1.1 Raw syntax and type system

We employ a hybrid nameless syntax, that uses De Bruijn indices for bound variables and De Bruijn levels for free variables. This scheme is an example of *locally-nameless* representation, as used for example in [53, 24], and adopts the Barendregt convention according to which free variables (also called *parameters*) and bound variables should be syntactically distinguished. The untyped raw syntax of System T^{wk}, $\mathsf{Term} : \mathsf{Set}$, is given by the following grammar.

$$t, u, z, s ::= \mathsf{Lev}\ n \mid \mathsf{Idx}\ n \mid \mathsf{Lam}\ t \mid t \cdot u \mid \mathsf{Zero} \mid \mathsf{Succ}\ t \mid \mathsf{Rec}\ z\ s\ t$$

where $n : \mathbb{N}$. We use the more convenient syntax x_n for $\mathsf{Lev}\ n$ and v_n for $\mathsf{Idx}\ n$. We also write $\boldsymbol{\lambda} t$ as a more lightweight notation for $\mathsf{Lam}\ t$.[1] Therefore, a term $\lambda x.(\lambda y.xy)z$ can be expressed in the syntax of System T^{wk} as $\boldsymbol{\lambda}(\boldsymbol{\lambda}(\mathsf{v}_1 \cdot \mathsf{v}_0) \cdot \mathsf{x}_0)$. Notice that z is free; it we were representing it with an index, as in $\boldsymbol{\lambda}(\boldsymbol{\lambda}(\mathsf{v}_1 \cdot \mathsf{v}_0) \cdot \mathsf{v}_2)$ for example, we would not produce a well-scoped term.

We then define T^{wk} types $\mathsf{Ty} : \mathsf{Set}$ and contexts $\mathsf{Ctxt} : \mathsf{Set}$ as follows:[2]

$$A, B ::= \mathsf{N} \mid A \Rightarrow B \qquad\qquad \Gamma ::= \diamond \mid \Gamma, A$$

We write $|\Gamma|$ for the natural number giving the length of the context Γ.[3] Notice that, since we use a nameless representation for free variables, context are nothing more that lists of types, with free levels representing locations in these lists. The type system is shown in Figure 4.1, and includes the inductive definition of the relation $\_\langle\_\rangle\mapsto\_$ mapping natural numbers, seen as De Bruijn levels, to locations in a context. Thus, an element of $\Gamma\langle n\rangle\mapsto A$ is a proof that the T^{wk} type A is contained in the context Γ, at position n, counting from the left.

Bound variables are represented as indices, thus we need a way to turn levels (i.e., free variables) into indices, so that they can become bound by λ-abstractions. We thus consider a function $\mathsf{idx} : \mathbb{N} \to \mathsf{Term} \to \mathsf{Term}$ such that $\mathsf{idx}\ n\ t$ takes a term t and replaces its free variable of level n with v_0. This is used, for example, in the rule of λ-introduction.

We write $\Gamma \vdash t : A$ for a type judgment, asserting that the term t has type A under the context Γ mapping assumptions to the free variables of t. [4] We then write $\Gamma \vdash t \sim s : A$ for definitional equality judgments. As we can see, the relation that is

[1]One should not confuse the lambda symbol in $\boldsymbol{\lambda} t$, that describes a term in the object language System T^{wk}, with $\lambda x.t$, that identifies a generic term in a hypothetical λ-calculus, as well as an abstraction in the metalanguage, which is itself a typed λ-calculus. The distinction should always be clear from the context. Nevertheless, notice that the first abstraction uses a bolder font, and a nameless representation, whereas the second has a lighter stroke, and names the abstracted variable. Nameless terms always belong to the object syntax.

[2]In the Agda formalization, we use # instead of commas for building contexts.

[3]In the Agda code, this is given by the function clen.

[4]Notice that we use the colon symbol for type judgments in both the metalanguage and the object language. It should always be clear what refers to what; in particular, object judgments are always written in full, like $\Gamma \vdash t : A$, whereas meta judgments are always given with an implicit context, like $t : A$.

$\boxed{\_\langle\_\rangle\mapsto\_ : \mathsf{Ctxt} \to \mathbb{N} \to \mathsf{Ty} \to \mathsf{Set}}$.

$$\frac{}{\Gamma, A \langle\ |\Gamma|\ \rangle\mapsto A} \qquad \frac{\Gamma \langle\ n\ \rangle\mapsto A}{\Gamma, B \langle\ n\ \rangle\mapsto A}$$

$\boxed{\_\vdash\_:\_ : \mathsf{Ctxt} \to \mathsf{Term} \to \mathsf{Ty} \to \mathsf{Set}}$.

$$\frac{\Gamma \langle\ n\ \rangle\mapsto A}{\Gamma \vdash \mathsf{x}_n : A} \qquad \frac{\Gamma, A \vdash t : B}{\Gamma \vdash \lambda(\mathsf{idx}\ |\Gamma|\ t) : A \Rightarrow B} \qquad \frac{\Gamma \vdash t : A \Rightarrow B \qquad \Gamma \vdash s : A}{\Gamma \vdash t \cdot s : B}$$

$$\frac{}{\Gamma \vdash \mathsf{Zero} : \mathsf{N}} \qquad \frac{\Gamma \vdash t : \mathsf{N}}{\Gamma \vdash \mathsf{Succ}\ t : \mathsf{N}} \qquad \frac{\Gamma \vdash z : A \qquad \Gamma \vdash s : \mathsf{N} \Rightarrow A \Rightarrow A \qquad \Gamma \vdash t : \mathsf{N}}{\Gamma \vdash \mathsf{Rec}\ z\ s\ t : A}$$

$\boxed{\_\vdash\_\sim\_:\_ : \mathsf{Ctxt} \to \mathsf{Term} \to \mathsf{Term} \to \mathsf{Ty} \to \mathsf{Set}}$.

$$\frac{\Gamma, A \vdash t : B \qquad \Gamma \vdash s : A}{\Gamma \vdash \lambda t \cdot s \sim t[s] : B} \qquad \frac{\Gamma \vdash z : A \qquad \Gamma \vdash s : \mathsf{N} \Rightarrow A \Rightarrow A \qquad \Gamma \vdash t : \mathsf{N}}{\Gamma \vdash \mathsf{Rec}\ z\ s\ \mathsf{Z} \sim z : A}$$

$$\frac{\Gamma \vdash z : A \qquad \Gamma \vdash s : \mathsf{N} \Rightarrow A \Rightarrow A \qquad \Gamma \vdash t : \mathsf{N}}{\Gamma \vdash \mathsf{Rec}\ z\ s\ (\mathsf{S}\ t) \sim s \cdot t \cdot \mathsf{Rec}\ z\ s\ t : A} \qquad \frac{\Delta \vdash t : A \qquad \Gamma \vdash_s \sigma \sim \tau : \Delta}{\Gamma \vdash \mathsf{Isub}\ t\ \sigma \sim \mathsf{Isub}\ t\ \tau : A}$$

$$\frac{\Gamma, A \vdash t : B \qquad \Gamma \vdash a \sim b : A}{\Gamma \vdash t\langle|\Gamma| \mapsto a\rangle \sim t\langle|\Gamma| \mapsto a\rangle : A}$$

+ equivalence rules.

Figure 4.1: System T^{wk}

axiomatized by the rules is CH-weak conversion, in that we have β-contraction, and recursion. Congruence rules are not present, and instead we have a primitive substitution rule, that makes us of a function $\_\langle\_ \mapsto \_\rangle : \mathsf{Term} \to \mathbb{N} \to \mathsf{Term} \to \mathsf{Term}$ replacing a term for a given level.

4.1.2 Weakening over levels

Judgments of System T^{wk} are considered under a certain context of assumptions. In several occasions, given some derivation of a typed entity, we will need to consider the same entity under a weakened version of the original context. One of the main advantages of using De Bruijn levels to represent free variables is that they make weakening very cheap, since they require no shifting to be performed in the term whatsoever.

Lemma 2. For all $\Delta : \mathsf{Ctxt}$, if $\Gamma\langle n\rangle\mapsto A$, then $\Gamma, \Delta\langle n\rangle\mapsto A$.

Proof. Straightforward induction on Δ. □

Theorem 6. For all $\Delta : \mathsf{Ctxt}$, if $\Gamma \vdash t : A$, then $\Gamma, \Delta \vdash t : A$.

Proof. By induction on the typing derivation. The only interesting case is when the conclusion introduces a level. Then, we apply Lemma 2. The other cases are immediate applications of the inductive hypothesis. □

4.1.3 Substitution over levels

Substitution over levels is implemented as a straightforward recursive function over terms.

$\_\langle\_\mapsto\_\rangle : \mathsf{Term} \to \mathbb{N} \to \mathsf{Term} \to \mathsf{Term}$
$\mathsf{Lev}\ x \langle\ n \mapsto a\ \rangle\ \underline{\text{with}}\ x \overset{?}{=} n$
$\mathsf{Lev}\ x \langle\ n \mapsto a\ \rangle \mid \mathsf{yes}\ p = a$
$\mathsf{Lev}\ x \langle\ n \mapsto a\ \rangle \mid \mathsf{no}\ \neg p = \mathsf{Lev}\ x$
$\mathsf{Idx}\ x \langle\ n \mapsto a\ \rangle = \mathsf{Idx}\ x$
$\mathsf{Lam}\ t \langle\ n \mapsto a\ \rangle = \mathsf{Lam}\ (t \langle\ n \mapsto a\ \rangle)$
$(t \cdot s) \langle\ n \mapsto a\ \rangle = t \langle\ n \mapsto a\ \rangle \cdot s \langle\ n \mapsto a\ \rangle$
$\mathsf{Zero} \langle\ n \mapsto a\ \rangle = \mathsf{Zero}$
$\mathsf{Succ}\ t \langle\ n \mapsto a\ \rangle = \mathsf{Succ}\ (t \langle\ n \mapsto a\ \rangle)$
$\mathsf{Rec}\ z\ s\ t \langle\ n \mapsto a\ \rangle = \mathsf{Rec}\ (z \langle\ n \mapsto a\ \rangle)\ (s \langle\ n \mapsto a\ \rangle)\ (t \langle\ n \mapsto a\ \rangle)$

In the sections that follow, we will sometimes use $t\langle a/n\rangle$ as an alternate notation for $t\langle n \mapsto a\rangle$.

4.1.4 Weakening over indices

In order to implement operations on bound variables, such as β-reduction, we need to define substitution on De Bruijn indices also. Since the value of indices depends on their position relative to binders, we need some form of *shifting* operation to ensure that indices always denote the intended variable even when their context or position is changed due to substitution. We implement these shiftings by weakening operations. We consider three kinds of weakening, represented as constructors of an inductive type Wk : Set:

$$w ::= \mathsf{Id} \mid \mathsf{Up}\ w \mid \mathsf{Skip}\ w$$

These constructors have the following meaning:

- Id: the identity weakening. It has no effect;
- Up w: increases all indices by 1, and then applies the weakening w;
- Skip w : skips the index at the shallowest level (index 0), applying the weakening w to the rest.

The informal meaning of weakening constructors is even more clear from the following definition of weakening on De Bruijn indices:

$\mathsf{wk\text{-}var} : \mathbb{N} \to \mathsf{Wk} \to \mathbb{N}$
$\mathsf{wk\text{-}var}\ x\ \mathsf{Id} = x$
$\mathsf{wk\text{-}var}\ x\ (\mathsf{Up}\ w) = \mathsf{suc}\ (\mathsf{wk\text{-}var}\ x\ w)$

wk-var zero (Skip w) = zero
wk-var (suc x) (Skip w) = suc (wk-var x w)

We now define weakening for general terms. Lev levels are ignored, whereas Idx indices are weakened according to wk-var. To weaken a λ-abstraction λt, we proceed recursively on its body t. However, the index 0 in t represents the abstracted variable, which is not free in λt, hence we must skip it by wrapping the weakening term in a Skip. The other constructors are treated with a straightforward recursive call.

wk : Term → Wk → Term
wk (Free x) w = Free x
wk (Bound x) w = Bound (wk-var x w)
wk (Lam t) w = Lam (wk t (Skip w))
wk ($t \cdot s$) w = wk t w · wk s w
wk Zero w = Zero
wk (Succ t) w = Succ (wk t w)
wk (Rec z s t) w = Rec (wk z w) (wk s w) (wk t w)

It what follows, we sometimes write $t \uparrow$ for wk t (Up Id). We can define composition of weakenings, as the following total operation.

_ ·ʷ _ : Wk → Wk → Wk
w ·ʷ Id = w
w ·ʷ Up w' = Up (w ·ʷ w')
Id ·ʷ Skip w' = Skip w'
Up w ·ʷ Skip w' = Up (w ·ʷ w')
Skip w ·ʷ Skip w' = Skip (w ·ʷ w')

As one would expect, we get the following lemma

Lemma 3. For all w, w' : Wk, t : Term, wk (wk t w) w' = wk t $(w \cdot^{w} w')$.

Proof. By straightforward induction on t. See `Syntax.Raw.Renaming.wk-comp`. □

4.1.5 Substitution over indices

We use *parallel* substitutions that simultaneously operate on all indices of a term. Here, shifting of De Bruijn indices is needed in order to properly implement capture-avoiding substitution. In particular, all indices must be shifted by one position whenever the substitution is pushed under a binder, otherwise all indices at value 0 would be captured by the binder. Therefore, our definition considers an additional constructor of *weakened substitutions*, in addition to the usual list-forming constructors. We define the inductive type Subst : Set as follows

$$\sigma ::= \mathsf{Id} \mid \sigma, t \mid \sigma \cdot w$$

These constructors have the following intuitive meaning:

- Id : the identity substitution. It has no effect;
- σ, t : extension of a substitution σ with a term t;
- $\sigma \cdot w$: weakening of a substitution. It applies the substitution sigma to the term, and then weakens the result with w.

To apply a substitution to an index, we just use it to select the term with the corresponding location in the substitution.

$\mathsf{sub\text{-}var} : \mathbb{N} \to \mathsf{Subst} \to \mathsf{Term}$
$\mathsf{sub\text{-}var}\ x\ \mathsf{Id} = \mathsf{Bound}\ x$
$\mathsf{sub\text{-}var}\ x\ (\sigma\ \cdot\ w) = \mathsf{wk}\ (\mathsf{sub\text{-}var}\ x\ \sigma)\ w$
$\mathsf{sub\text{-}var}\ \mathsf{zero}\ (\sigma\ ,\ t) = t$
$\mathsf{sub\text{-}var}\ (\mathsf{suc}\ x)\ (\sigma\ ,\ t) = \mathsf{sub\text{-}var}\ x\ \sigma$

Substitution on terms is mostly a straightforward recursion. In the case of λt, we proceed by recursion on the body t, recalling that the index 0 should not be affected by the substitution. To do this, we consider the extended substitution $(\sigma \cdot \mathsf{Up\ Id}), \mathsf{v}_0$, that has the effect of leaving the index 0 untouched, while lifting all indices of σ by one so that no variables get captured by Lam. Consider the following general function performing this extension an arbitrary number $n : \mathbb{N}$ of times

$\mathsf{shift} : \mathbb{N} \to \mathsf{Subst} \to \mathsf{Subst}$
$\mathsf{shift}\ \mathsf{zero}\ \sigma = \sigma$
$\mathsf{shift}\ (\mathsf{suc}\ n)\ \sigma = \mathsf{shift}\ n\ \sigma\ \cdot\ \mathsf{Up\ Id}\ ,\ \mathsf{Bound}\ 0$

Letting $\mathsf{sh} :\equiv \mathsf{shift}\ 1$, we have

$\mathsf{sub} : \mathsf{Term} \to \mathsf{Subst} \to \mathsf{Term}$
$\mathsf{sub}\ (\mathsf{Free}\ x)\ \sigma = \mathsf{Free}\ x$
$\mathsf{sub}\ (\mathsf{Bound}\ x)\ \sigma = \mathsf{sub\text{-}var}\ x\ \sigma$
$\mathsf{sub}\ (\mathsf{Lam}\ t)\ \sigma = \mathsf{Lam}\ (\mathsf{sub}\ t\ (\mathsf{sh}\ \sigma))$
$\mathsf{sub}\ (t \cdot s)\ \sigma = \mathsf{sub}\ t\ \sigma\ \cdot\ \mathsf{sub}\ s\ \sigma$
$\mathsf{sub}\ \mathsf{Zero}\ \sigma = \mathsf{Zero}$
$\mathsf{sub}\ (\mathsf{Succ}\ t)\ \sigma = \mathsf{Succ}\ (\mathsf{sub}\ t\ \sigma)$
$\mathsf{sub}\ (\mathsf{Rec}\ z\ s\ t)\ \sigma = \mathsf{Rec}\ (\mathsf{sub}\ z\ \sigma)\ (\mathsf{sub}\ s\ \sigma)\ (\mathsf{sub}\ t\ \sigma)$

We will use $t[s]$ as a shorthand for $\mathsf{sub}\ t\ (\mathsf{Id}, s)$, that simply substitutes s for the index 0 in t. We also write $t[w, a]$ for $\mathsf{sub}\ t\ (\mathsf{Id} \cdot w, a)$, whenever it is clear from the context that w is a weakening.

Whereas right-composition between substitutions and weakenings is supported by a constructor of Subst, left-composition can be expressed in terms of right-composition, as shown by the following operation:

$\_{}^{\mathsf{w}}\!\cdot\_ : \mathsf{Wk} \to \mathsf{Subst} \to \mathsf{Subst}$
$w\ {}^{\mathsf{w}}\!\cdot\ \mathsf{Id} = \mathsf{Id}\ \cdot\ w$

$w \overset{\mathsf{w}}{\cdot} (\sigma \cdot w') = (w \overset{\mathsf{w}}{\cdot} \sigma) \cdot w'$
$\mathsf{Id} \overset{\mathsf{w}}{\cdot} (\sigma \,,\, t) = \sigma \,,\, t$
$\mathsf{Up}\ w \overset{\mathsf{w}}{\cdot} (\sigma \,,\, t) = w \overset{\mathsf{w}}{\cdot} \sigma$
$\mathsf{Skip}\ w \overset{\mathsf{w}}{\cdot} (\sigma \,,\, t) = w \overset{\mathsf{w}}{\cdot} \sigma \,,\, t$

With it, we can define composition of substitutions as a total operation, just as we did for weakenings.

$\_ \cdot^s \_ : \mathsf{Subst} \to \mathsf{Subst} \to \mathsf{Subst}$
$\sigma \cdot^s \mathsf{Id} = \sigma$
$\sigma \cdot^s (\tau \cdot w) = (\sigma \cdot^s \tau) \cdot w$
$\mathsf{Id} \cdot^s (\tau \,,\, x) = \tau \,,\, x$
$(\sigma \cdot w) \cdot^s (\tau \,,\, t) = \sigma \cdot^s (w \overset{\mathsf{w}}{\cdot} (\tau \,,\, t))$
$(\sigma \,,\, t) \cdot^s (\tau \,,\, s) = (\sigma \cdot^s (\tau \,,\, s)) \,,\, \mathsf{sub}\ t\ (\tau \,,\, s)$

We thus get the following lemma

Lemma 4. For all $\sigma, \tau : \mathsf{Subst}, t : \mathsf{Term}$, $\mathsf{sub}\ (\mathsf{sub}\ t\ \sigma)\ \tau = \mathsf{sub}\ t\ (\sigma \cdot^s \tau)$.

Proof. By induction on t. See `Syntax.Raw.Substitution.sub-comp`. □

Using the above representation for substitution is convenient for certain parts of the formalization, but in many cases it is too "intensional". For example, it is clear that $\mathsf{sub}\ t\ \sigma = \mathsf{sub}\ t\ (\sigma \cdot \mathsf{Id})$, because σ and $\sigma \cdot \mathsf{Id}$ denote essentially the same substitution, although not so as mere terms of type Subst. We formalize this concept of equality of substitutions with the relation $\_ \approx^s \_ \in \mathsf{Subst} \to \mathsf{Subst} \to \mathsf{Set}$, defined as follows

$$\sigma \approx^s \tau :\equiv (x : \mathbb{N}) \to \mathsf{sub\text{-}var}\ x\ \sigma = \mathsf{sub\text{-}var}\ x\ \tau$$

that is, two substitutions are equal whenever they are point-wise equal, meaning that they assign syntactically equal terms to equal variables. It is straightforward to see that the relation is an equivalence, and also a congruence w.r.t. substitution constructors. But most importantly, we get the following general lemma:

Lemma 5. If $t : \mathsf{Term}$, $\sigma, \tau : \mathsf{Subst}$, and $\sigma \approx^s \tau$, then $\mathsf{sub}\ t\ \sigma = \mathsf{sub}\ t\ \tau$.

Proof. By induction on t. See `Syntax.Raw.Substitution.eq-sub`. □

4.2 System T^{ex} with Explicit Weak Equality Judgments

The next step to prove normalization for System T^{wk} is to define an algorithmic method to evaluate T^{wk} terms to normal form. In Section 3.3, we observed that under CH-weak reduction, variables assume two distinct *roles*, and shown that making this distinction syntactically explicit leads to a simple, structurally recursive definition of evaluation.

Unfortunately, the results of Section 3.3 cannot be applied directly to System T^{wk}. First of all, the formal system does not support in a native way the distinction between variable roles. Retrofitting such support would require the addition of awkward side-conditions to the rules of the calculus, and it is very likely to lead to a confused and overly-complicated formalization. But most importantly, the conversion relation that arises from T^{wk}'s equality judgments is not a congruence, since it does not validate the ξ rule.

The absence of congruence rules complicates the metamathematical analysis of equality judgments, because the corresponding equality does not play well with many definitions and proofs made by induction on (well-typed) terms. To illustrate the difficulty that may arise, consider a sketched attemp at showing soundness of NbE (Section 2.4), where the normalization function nf is defined by structural recursion on terms:

$$\Gamma \vdash t : A \qquad \Longrightarrow \qquad \Gamma \vdash t \sim \mathsf{nf}\ t : A$$

We proceed by induction on the derivation. Consider the case where t is a λ-abstraction, so that

$$\frac{\Gamma, A \vdash t : B}{\Gamma \vdash \lambda t : A \Rightarrow B}$$

Since $\mathsf{nf}\ (\lambda t) \equiv \lambda(\mathsf{nf}\ t)$, we have to show that $\Gamma \vdash \lambda t \sim \lambda(\mathsf{nf}\ t) : A \Rightarrow B$. By inductive hypothesis, we know that $\Gamma, A \vdash t \sim \mathsf{nf}\ t : B$. We would like to deduce $\Gamma \vdash \lambda t \sim \lambda(\mathsf{nf}\ t) : A \to B$ from that, but we cannot, because that would require the ξ rule.

We observe that the definition of judgmental equality of T^{wk} is inadequate to establish certain metamathematical properties defined over the inductive structure of terms, and in particular when reasoning about normalization. We therefore employ a *type assignment* approach, and extract a type system from the tagged syntax of Section 3.3, that we call System T^{ex}. This calculus makes the details of CH-weak reduction that we care about explicit in the syntactic structure of judgments; in particular, it provides built-in support for reasoning about variable roles, and allows to enforce that computation rules only apply to weak redexes.

The raw syntax of untyped terms is exactly the same as System T^{wk}, in the sense that we employ a hybrid nameless scheme with both De Bruijn levels and indices. The crucial difference is that now we do not use them to distinguish between *free* and *bound* variables, but between variables with *global role* and variables with *local* role. In particular, we use the constructor Lev of De Bruijn levels to represent variables with a *global* role, whereas Idx of De Bruijn indices is used for variables with a *local* role. We stress the fact that roles are orthogonal to the status of being free/bound; for example, a free variable may be either global or local, hence be represented by either indices or levels, unlike in System T^{wk}, where free variables are always represented by levels.

Type judgments of System T^{ex} are of the form $\Gamma :: \Delta \vdash t : A$, as shown in Figure 4.2. As before, we use the relation $\_\langle\_\rangle\!\mapsto\!\_$ to locate free levels in a context. But in addition, we have another, analogous relation $\_[\_]\!\mapsto\!\_$, mapping De Bruijn indices to locally-free assumptions. What is immediately evident is the use of a double context $\Gamma :: \Delta$, keeping

$\boxed{\_[\_]{\mapsto}\_ : \mathsf{Ctxt} \to \mathbb{N} \to \mathsf{Ty} \to \mathsf{Set}}$

$$\frac{}{\Gamma, A\ [\ 0\ ]{\mapsto}\ A} \qquad \frac{\Gamma\ [\ n\ ]{\mapsto}\ A}{\Gamma, B\ [\ \mathsf{succ}\ n\ ]{\mapsto}\ A}$$

$\boxed{\_{::}\_{\vdash}\_{:}\_ : \mathsf{Ctxt} \to \mathsf{Ctxt} \to \mathsf{Term} \to \mathsf{Ty} \to \mathsf{Set}}$

$$\frac{\Gamma\langle n\rangle{\mapsto}A}{\Gamma :: \Delta \vdash \mathsf{x}_n : A} \qquad \frac{\Delta[n]{\mapsto}A}{\Gamma :: \Delta \vdash \mathsf{v}_n : A} \qquad \frac{\Gamma :: \Delta, A \vdash t : B}{\Gamma :: \Delta \vdash \lambda t : A \Rightarrow B}$$

$$\frac{\Gamma :: \Delta \vdash t : A \Rightarrow B \qquad \Gamma :: \Delta \vdash s : A}{\Gamma :: \Delta \vdash t \cdot s : B} \qquad \frac{}{\Gamma :: \Delta \vdash \mathsf{Zero} : \mathsf{N}} \qquad \frac{\Gamma :: \Delta \vdash t : \mathsf{N}}{\Gamma :: \Delta \vdash \mathsf{Succ}\ t : \mathsf{N}}$$

$$\frac{\Gamma :: \Delta \vdash z : A \qquad \Gamma :: \Delta \vdash s : \mathsf{N} \Rightarrow A \Rightarrow A \qquad \Gamma :: \Delta \vdash t : \mathsf{N}}{\Gamma :: \Delta \vdash \mathsf{Rec}\ z\ s\ t : A}$$

Figure 4.2: System T^{ex}

track of free variables with a global and local role, respectively. We sometimes call the two contexts globally-free context and locally-free context. Double contexts in typing judgments allow us to enforce a precise discipline over the presence (or, really, absence) of locally-free variables, and by consequence, over the presence of weak redexes. In particular, it is easy to see that in the class of well-typed redexes, weak redexes are exactly those typeable under an empty locally-free context $\Gamma :: \diamond$. In fact, the only way for a term t to be typeable in such context is to be closed w.r.t. local variables.

Another aspect of System T^{ex} is the presence of typed one-step reduction judgments, written as $\Gamma :: \Delta \vdash t \longrightarrow s : A$ and defined in Figure 4.3. Unlike System T^{wk}, the typed conversion relation of T^{ex} is not defined directly, but it is expressed as the equivalence closure of one-step reduction. This formulation makes it easier to establish the correspondence between T^{wk} and T^{ex}, as shown in Section 4.3. Nevertheless, this choice is not restrictive in any way, since all rules expressed in Figure 4.3 for one-step reduction are admissible when conversion is used instead:

Lemma 6.

1. If $\Gamma :: \Delta, A \vdash t \sim s : B$, then $\Gamma :: \Delta \vdash \lambda t \sim \lambda s : A \to B$;
2. If $\Gamma :: \Delta \vdash t \sim s : \mathbb{N}$, then $\Gamma :: \Delta \vdash \mathsf{Succ}\ t \sim \mathsf{Succ}\ s : \mathbb{N}$;
3. If $\Gamma :: \Delta \vdash r \sim r' : A \to B$ and $\Gamma :: \Delta \vdash s \sim s' : A$, then $\Gamma :: \Delta \vdash r \cdot s \sim r' \cdot s' : B$;
4. If $\Gamma :: \Delta \vdash z \sim z' : A$, $\Gamma :: \Delta \vdash s \sim s' : \mathbb{N} \to A \to A$, and $\Gamma :: \Delta \vdash t \sim t' : \mathbb{N}$, then $\Gamma :: \Delta \vdash \mathsf{Rec}\ z\ s\ t \sim \mathsf{Rec}\ z'\ s'\ t' : A$.

Proof. By laborious but straightforward induction on derivations. See the module `Syntax.Typed.Equality.Properties` for details. □

4.2.1 Computation and congruence rules

The syntax of System T^{ex} gives us the means to express CH-weak contractions in a precise manner. We do so by definiting computation rules so that they only apply to redexes that are weak, namely that are provably closed w.r.t. *local* variables. Consider, for example, the β-reduction rule:

$$\frac{\Gamma :: \diamond, A \vdash t : B \qquad \Gamma :: \diamond \vdash s : A}{\Gamma :: \Delta \vdash \mathsf{Lam}\ t \cdot s \longrightarrow t[s] : B}$$

The premises of the rule provide evidence that both λt and s are derivable in an empty context of locally-free variables. By construction of the calculus, this is only possible when the two terms are closed w.r.t. local variables, which implies that $\lambda t \cdot s$ is a weak redex. The conclusion is established under an arbitrary locally-free context Δ, to ensure admissibility of weakening. All computation rules are defined in a similar way. As a consequence, we can characterize a reduction of the form $\Gamma :: \Delta \vdash t \longrightarrow s : A$ intuitively as one where all contracted redexes are *weak*, and similarly for an equality $\Gamma :: \Delta \vdash t \sim s : A$.

The expressive double contexts also enable a definition of reduction and equality judgments as a *congruence relation*, and where CH-weak conversion follows the inductive structure of terms. To see how, consider a reduction between two λ-abstractions.

$$\Gamma :: \Delta \vdash \lambda t \longrightarrow \lambda s : A \Rightarrow B$$

One issue with the standard definition of CH-weak reduction (Section 3.2) is that it not only rules out the standard ξ rule, but it also makes very difficult to express a restricted form of ξ rule that, as discussed in Section 3.2.1, *does* hold under CH-weak conversion.

This issue is not present in System T^{ex}, and the restricted ξ rule can be expressed directly. To do so, it is sufficient to consider the abstracted variable as part of the locally-free context when proving the premise. This has the effect of ruling that variable out of any redex that will be contracted within the derivation, by construction of the computation rules.

$$\frac{\Gamma :: \Delta, A \vdash t \longrightarrow s : B}{\Gamma :: \Delta \vdash \lambda t \longrightarrow \lambda s : A \Rightarrow B}$$

4.2.2 Weakening and substitution

We will consider well-typed weakenings, defined in Figure 4.4. A judgment of the form $\Gamma \vdash_r w : \Delta$ denotes a well-typed weakening w that acts on well-typed terms under a locally-free context Δ, and renames their indices so that they can be typed under a new context Γ.

Lemma 7. Let $\Theta :: \Delta \vdash t : A$ and $\Gamma \vdash_r w : \Delta$. Then, $\Theta :: \Gamma \vdash \mathsf{wk}\ t\ w : A$ is derivable.

Proof. By induction on the derivation of t. See `Syntax.Typed.Typed.⊢wk`. □

$\boxed{\_::\_\vdash\_\longrightarrow\_:\_ : \mathsf{Ctxt} \to \mathsf{Ctxt} \to \mathsf{Term} \to \mathsf{Term} \to \mathsf{Ty} \to \mathsf{Set}}$

$$\frac{\Gamma :: \diamond, A \vdash t : B \qquad \Gamma :: \diamond \vdash s : A}{\Gamma :: \Delta \vdash \lambda t \cdot s \longrightarrow t[s] : B}\ (\beta)$$

$$\frac{\Gamma :: \Delta \vdash z : A \qquad \Gamma :: \Delta \vdash s : \mathsf{N} \Rightarrow A \Rightarrow A \qquad \Gamma :: \Delta \vdash t : \mathsf{N}}{\Gamma :: \Delta \vdash \mathsf{Rec}\ z\ s\ \mathsf{Z} \longrightarrow z : A}$$

$$\frac{\Gamma :: \Delta \vdash z : A \qquad \Gamma :: \Delta \vdash s : \mathsf{N} \Rightarrow A \Rightarrow A \qquad \Gamma :: \Delta \vdash t : \mathsf{N}}{\Gamma :: \Delta \vdash \mathsf{Rec}\ z\ s\ (\mathsf{Succ}\ t) \longrightarrow s \cdot t \cdot (\mathsf{Rec}\ z\ s\ t) : A}$$

$$\frac{\Gamma :: \Delta, A \vdash t \longrightarrow s : B}{\Gamma :: \Delta \vdash \lambda t \longrightarrow \lambda s : A \Rightarrow B}\ (\xi) \qquad \frac{\Gamma :: \Delta \vdash t \longrightarrow s : \mathsf{N}}{\Gamma :: \Delta \vdash \mathsf{Succ}\ t \longrightarrow \mathsf{Succ}\ s : \mathsf{N}}$$

$$\frac{\begin{array}{c}\Gamma :: \Delta \vdash s : A \\ \Gamma :: \Delta \vdash t \longrightarrow t' : A \Rightarrow B\end{array}}{\Gamma :: \Delta \vdash t \cdot s \longrightarrow t' \cdot s : B} \qquad \frac{\begin{array}{c}\Gamma :: \Delta \vdash t : A \Rightarrow B \\ \Gamma :: \Delta \vdash s \longrightarrow s' : A\end{array}}{\Gamma :: \Delta \vdash t \cdot s \longrightarrow t \cdot s' : B}$$

$$\frac{\Gamma :: \Delta \vdash z \longrightarrow z' : A \qquad \Gamma :: \Delta \vdash s : \mathsf{N} \Rightarrow A \Rightarrow A \qquad \Gamma :: \Delta \vdash t : \mathsf{N}}{\Gamma :: \Delta \vdash \mathsf{Rec}\ z\ s\ t \longrightarrow \mathsf{Rec}\ z'\ s\ t : A}$$

$$\frac{\Gamma :: \Delta \vdash z : A \qquad \Gamma :: \Delta \vdash s \longrightarrow s' : \mathsf{N} \Rightarrow A \Rightarrow A \qquad \Gamma :: \Delta \vdash t : \mathsf{N}}{\Gamma :: \Delta \vdash \mathsf{Rec}\ z\ s\ t \longrightarrow \mathsf{Rec}\ z\ s'\ t : A}$$

$$\frac{\Gamma :: \Delta \vdash z : A \qquad \Gamma :: \Delta \vdash s : \mathsf{N} \Rightarrow A \Rightarrow A \qquad \Gamma :: \Delta \vdash t \longrightarrow t' : \mathsf{N}}{\Gamma :: \Delta \vdash \mathsf{Rec}\ z\ s\ t \longrightarrow \mathsf{Rec}\ z\ s\ t' : A}$$

$\boxed{\_::\_\vdash\_\sim\_:\_ : \mathsf{Ctxt} \to \mathsf{Ctxt} \to \mathsf{Term} \to \mathsf{Term} \to \mathsf{Ty} \to \mathsf{Set}}$

$$\frac{\Gamma :: \Delta \vdash t \longrightarrow s : A}{\Gamma :: \Delta \vdash t \sim s : A}\ {\sim}{\longrightarrow} \qquad \frac{\Gamma :: \Delta \vdash t : A}{\Gamma :: \Delta \vdash t \sim t : A}\ {\sim}\mathsf{refl}$$

$$\frac{\Gamma :: \Delta \vdash s \sim t : A}{\Gamma :: \Delta \vdash t \sim s : A}\ \mathsf{symm}{\sim} \qquad \frac{\Gamma :: \Delta \vdash t \sim s : A \qquad \Gamma :: \Delta \vdash s \sim r : A}{\Gamma :: \Delta \vdash t \sim r : A}\ \mathsf{trans}{\sim}$$

Figure 4.3: Definitional reduction and equality for System T^{ex}

$$\frac{}{\Gamma \vdash_r \mathsf{Id} : \Gamma} \qquad \frac{\Delta \vdash_r w : \Gamma}{\Delta, A \vdash_r \mathsf{Up}\ w : \Gamma} \qquad \frac{\Delta \vdash_r w : \Gamma}{\Delta, A \vdash_r \mathsf{Skip}\ w : \Gamma, A}$$

$$\frac{}{\Theta :: \Gamma \vdash_s \mathsf{Id} : \Gamma} \qquad \frac{\Theta :: \Gamma \vdash_s \sigma : \Delta \qquad \Theta :: \Gamma \vdash t : A}{\Theta :: \Gamma \vdash_s \sigma, t : \Delta, A}$$

$$\frac{\nabla \vdash_r w : \Delta \qquad \Theta :: \Delta \vdash_s \sigma : \Gamma}{\Theta :: \nabla \vdash_s \sigma \cdot w : \Gamma}$$

Figure 4.4: Well-typed weakenings and substitutions on De Bruijn indices

As with weakenings, we will also consider well-typed parallel substitutions on indices.

Lemma 8. Let $\Theta :: \Delta \vdash t : A$ and $\Theta :: \Gamma \vdash_s \sigma : \Delta$. Then, $\Theta :: \Gamma \vdash \mathsf{sub}\ t\ \sigma : A$ is derivable.

Proof. By induction on the derivation of t. See `Syntax.Typed.Typed.⊢sub`. □

Finally, we consider substitutions over De Bruijn levels.

Lemma 9. If $\Gamma :: \Delta \vdash t : A$ and $\Gamma :: \diamond \vdash a \sim b : A$, then $\Gamma :: \Delta \vdash t\langle|\Gamma| \mapsto a\rangle \sim t\langle|\Gamma| \mapsto b\rangle : A$.

Proof. By induction on the derivation of t. See `Syntax.Typed.MetaSubstitution.⊢sub`. □

4.3 Correspondence between System T^{wk} and System T^{ex}

Before moving on to normalization, it is important to see how the two weak versions of System T that we just defined relate to each other. A first correspondence relates typing derivations. Inference rules for type judgments are essentially the same in the two calculi, apart from indices in System T^{ex}, that may denote free variables in the locally-free context. When we restrict to free levels only, the two calculi produce the same well-typed terms.

Lemma 10. $\Gamma :: \diamond \vdash t : A \iff \Gamma \vdash t : A$

Proof. Straightforward induction of derivations. □

A much more interesting correspondence pertains definitional equality. Consider a derivation of the form $\Gamma \vdash t \sim s : A$, that witnesses a typed CH-weak conversion between the terms t and s. As we learned in Section 3.2, this is equivalent to saying that the conversion only involves *weak* redexes, that is, redexes that have all their free variablese in Γ, so that no one is bound somewhere in t or s. But the same conversion can be expressed in the explicit calculus as $\Gamma :: \diamond \vdash t \sim s : A$, by the very meaning of double contexts. Therefore, we have the following correspondence:

$$\Gamma \vdash t \sim s : A \iff \Gamma :: \diamond \vdash t \sim s : A$$

Notice that the right-to-left implication can be strengthened, in the sense that $\Gamma, \Delta \vdash t \sim s : A \Longleftarrow \Gamma :: \Delta \vdash t \sim s : A$ holds. The opposite is not true, because the variables in Δ, that are free in the first judgment and therefore allowed in any contracted redex, would appear locally-free in the second judgment, and they would be instead ruled out of any contraction.

We prove the correspondence by addressing one implication at a time.

4.3.1 $\Gamma \vdash t \sim s : A \Longrightarrow \Gamma :: \diamond \vdash t \sim s : A$

The left-to-right direction is relatively straightforward: every derivation of a judgment $\Gamma \vdash t \sim s : A$ uses one among computation, substitution, and equivalence rules, by definition of the calculus. We have shown that the substitution rule on levels is admissible in the explicit calculus (Lemma 9), which means that every rule application used to derive the left judgment can be simulated in the explicit calculus (up to trivial De Bruijn indeces/levels readjustments), thus deriving the same equation. The actual proof is a straightforward induction on derivations, and it is not reproduced here. We refer the reader to the module `Syntax.Typed.Correspondence.ImplicitToExplicit` for details.

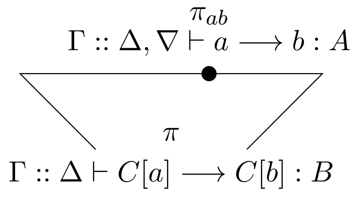
4.3.2 $\Gamma \vdash t \sim s : A \Longleftarrow \Gamma :: \diamond \vdash t \sim s : A$

Proving the right-to-left implication is more involved, since now is not the case that every rule application on the explicit derivation can be simulated in the implicit calculus. The obstacle is, obviously, the ξ rule, that does not have a direct counterpart in the implicit calculus (indeed, this is the reason we developed an alternative calculus in the first place.)

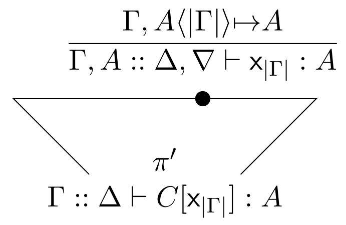
To prove the implication, we have to show that there exists a systematic way to express an explicit derivation in terms of inference rules that *do have* a direct counterpart in the implicit calculus, namely computation, substitution, and equivalence rules. The proof is, in its essence, very similar to what is done in Theorem 5 for the untyped case. We first focus on reductions, showing that any derivation of $\Gamma :: \Delta \vdash t \longrightarrow s : B$ can be transformed into some derivation of $\Gamma, \Delta \vdash t \sim s : B$. The proof then easily extends to the full conversion relation.

Consider a derivation π of the one-step reduction $\Gamma :: \Delta \vdash t \longrightarrow s : B$. Such reduction must involve a weak contraction $a \rightsquigarrow b$ of a weak redex a that gets contracted in t. By construction, a is closed w.r.t. local variables, so it surely does not contain any of those that are bound in t. Thus, just as we did in the untyped case, we can express t as a factor of two components, namely the subterm a and the rest of t, a context $C[\,]$ with a single hole, such that $C[a] \equiv t$ and $C[b] \equiv s$. We can then rewrite the reduction as $\Gamma :: \Delta \vdash C[a] \longrightarrow C[b] : B$, since a is the only subterm of t that changes in the reduction.

The term a is a contracted redex, so π must contain a subderivation $\pi_{ab} : \Gamma :: \Delta, \nabla \vdash a \longrightarrow b : A$ for some ∇ and A, established with a computation rule. The rest of π does not contain any other contractions, but must be composed of congruence rules that simply follow the structure of $C[\,]$, pushing assumptions to the locally-free context.


π_{ab}
$\Gamma :: \Delta, \nabla \vdash a \longrightarrow b : A$
π
$\Gamma :: \Delta \vdash C[a] \longrightarrow C[b] : B$

We can imagine to extract the subtree π_{ab} from π, leaving a placeholder variable x in its place. Clearly, x must be of the same type of a and b. Moreover, since a is a weak redex, both and b must be derivable in an empty locally-free context. We make this official by putting the placeholder variable x, standing for a and b, in the free context, thus extending Γ to Γ, A, and putting $x \equiv \mathsf{x}_{|\Gamma|}$. Intuitively, what remains of π is a derivation π' of $\Gamma, A :: \Delta \vdash C[\mathsf{x}_{|\Gamma|}] : B$.


$\Gamma, A\langle|\Gamma|\rangle \mapsto A$
$\Gamma, A :: \Delta, \nabla \vdash \mathsf{x}_{|\Gamma|} : A$
π'
$\Gamma :: \Delta \vdash C[\mathsf{x}_{|\Gamma|}] : A$

At this point, we are almost done. Since the locally-free context is not relevant in the derivation of $\Gamma :: \Delta, \nabla \vdash a \longrightarrow b : A$, we can safely replace it with another one, hence obtaining a derivation of $\Gamma :: \Delta \vdash a \longrightarrow b : A$. Now, we can apply the substitution rule to get the following derivation

$$\frac{\Gamma, A :: \Delta \vdash C[\mathsf{x}_{|\Gamma|}] : B \qquad \Gamma :: \Delta \vdash a \longrightarrow b : A}{\Gamma :: \Delta \vdash C[\mathsf{x}_{|\Gamma|}]\langle a/|\Gamma|\rangle \longrightarrow C[\mathsf{x}_{|\Gamma|}]\langle b/|\Gamma|\rangle : B}$$

Just as in Lemma 1, it's easy to see that $C[\mathsf{x}_{|\Gamma|}]\langle a/|\Gamma|\rangle \equiv C[a] \equiv t$ and, similarly, that $C[\mathsf{x}_{|\Gamma|}]\langle b/|\Gamma|\rangle \equiv C[b] \equiv s$. The conclusion of the rule is therefore the reduction that we started with, namely $\Gamma :: \Delta \vdash t \longrightarrow s : B$, but expressed with substitution and computation rules alone.

We have informally shown how to factor a derivation of a reduction $\Gamma :: \Delta \vdash t \longrightarrow s : B$ into two objects, namely a derivation of $C[\mathsf{x}_{|\Gamma|}]$ for some $C[\,]$, and a derivation of $a \longrightarrow b$, that can be combined to produce the original reduction, using syntax and rules that have a direct counterpart in the implicit calculus. The following lemma formalizes the argument above, which is then used in Theorem 7 to finally prove the result, with a minor difference: instead of defining contexts $C[\,]$ and then proving that $C[a] \equiv t$ and that $C[\mathsf{x}_{|\Gamma|}]\langle a/|\Gamma|\rangle \equiv C[a]$, we just consider a term C such that $C\langle a/|\Gamma|\rangle \equiv t$, that corresponds to providing $C[\,]$ with its hole already filled by $\mathsf{x}_{|\Gamma|}$.

Lemma 11. For all derivations of a typed reduction $\Gamma :: \Delta \vdash t \longrightarrow s : A$, there exist terms C, a, b and a type ty such that

- $\Gamma, ty :: \Delta \vdash C : A$ is derivable;
- $\Gamma \vdash a \sim b : ty$ is derivable;
- $C\langle a/|\Gamma|\rangle \equiv t$;
- $C\langle b/|\Gamma|\rangle \equiv s$.

Proof. By induction on the derivation. We consider a couple of cases, starting with β-reduction:

$$\frac{\Theta :: \diamond, A \vdash t : B \qquad \Theta :: \diamond \vdash s : A}{\Theta :: \Gamma \vdash \lambda t \cdot s \longrightarrow t[s] : B}$$

Let $a \equiv t, b \equiv s$. Clearly $\Theta :: \diamond \vdash \lambda t : A \to B$ is derivable, so by Lemma 10 we have $\Theta \vdash \lambda t : A \Rightarrow B$ and $\Theta \vdash s : A$, hence $\Theta \vdash \lambda t \cdot s \sim t[s] : B$. Moreover, let $ty \equiv B$ and $C \equiv \mathsf{x}_{|\Gamma|}$. Then, clearly $\Theta, ty :: \Delta \vdash \mathsf{x}_{|\Gamma|} : B$, as well as $(\mathsf{x}_{|\Gamma|})\langle a/|\Gamma|\rangle \equiv a \equiv t$, and similarly for s.

Consider now the ξ rule.

$$\frac{\Theta :: \Gamma, A \vdash t \longrightarrow s : B}{\Theta :: \Gamma \vdash \lambda t \longrightarrow \lambda s : A \Rightarrow B}$$

By inductive hypothesis, there are a, b, ty such that $\Gamma \vdash a \sim b : ty$ is derivable; moreover, $\Gamma, ty :: \Delta, A \vdash C : B$ is derivable for some C, and we have $C\langle a/|\Gamma|\rangle \equiv t$ and $C\langle b/|\Gamma|\rangle \equiv s$. But then, $\Gamma, ty :: \Delta \vdash \lambda C : A \Rightarrow B$ is derivable by λ-introduction, and $(\lambda C)\langle a/|\Gamma|\rangle \equiv \lambda(C\langle a/|\Gamma|\rangle) \equiv \lambda t$, and similarly for λs.

See the full proof in `Syntax/Typed/Correspondence/ExplicitToImplicit.agda`. □

Theorem 7. If $\Gamma :: \diamond \vdash t \sim s : A$ is derivable, then $\Gamma \vdash t \sim s : A$ is derivable.

Proof. By induction on the derivation. We distinguish four possible rule applications used to derive the conclusion.

- Reflexivity. Then, there must be a derivation of $\Gamma :: \diamond \vdash t : A$. By Lemma 10, this implies $\Gamma \vdash t : A$, hence the conclusion by reflexivity in the implicit calculus.
- Symmetry and transitivity. By induction hypothesis and symmetry in the implicit calculus.
- Reduction.

$$\frac{\Gamma :: \diamond \vdash t \longrightarrow s : A}{\Gamma :: \diamond \vdash t \sim s : A}$$

By Lemma 11, there exist terms C, a, b and a type ty such that $\Gamma, ty :: \diamond \vdash C : A$ and $\Gamma \vdash a \sim b : ty$ are derivable, and $C\langle a/|\Gamma|\rangle \equiv t$, $C\langle b/|\Gamma|\rangle \equiv s$. By Lemma 10, we have $\Gamma, ty \vdash C : A$, hence we can apply the substitution rule to get

$$\frac{\Gamma, ty \vdash C : A \qquad \Gamma \vdash a \sim b : ty}{\Gamma \vdash C\langle a/|\Gamma|\rangle \sim C\langle b/|\Gamma|\rangle : A}$$

that is $\Gamma \vdash t \sim s : A$ by hypothesis.

$\square$

4.3.3 Comments

CH-weak conversion can be expressed more faithfully, and in much more detail in the explicit framework of System T^{ex}, albeit with a heavier and less standard syntax. System T^{wk}, on the other hand, achieves a syntactically simpler formulation, at the expense of expressivity, preciseness, and control over weak equality. The previous section shows that the explicit syntax is no more powerful that the implicit one, as one step of reduction in System T^{ex} can be expressed as a single instance of substitution in System T^{wk}. This suggests that we could think of the implicit calculus as an *abstraction* over the explicit one, with the substitution rule abstracting over several low-level details that are instead left explicit in System T^{ex}.

Chapter 5

Normalization of Weak System T

In this chapter, we provide a constructive normalization proof for System T^{wk}. We proceed by first establishing normalization by evaluation for System T^{ex}, from which the normalization theorem follows. We then transfer the result to System T^{wk} by exploiting the correspondence between the two calculi, shown at the end of the previous chapter.

All the contents of this chapter can be found in the formalization [5] under the folder `Semantics/`, in addition to `Syntax/`.

5.1 Normalization by Evaluation for System T^{ex}

To normalize System T^{ex}, we will employ the method of untyped Normalization by Evaluation, as described in [8]. The main ingredient is represented by an evaluation function, interpreting untyped λ-terms into the semantics. We will use a model of normal forms, thus it is crucial to understand and formalize the concept of (weak) redex and normal form in the setting of System T^{ex}.

5.1.1 Weak redexes and normal forms

Weak redexes are just redexes that are closed w.r.t. local variables. To define this, we first give the inductive definition of a relation $\mathsf{Sz} : \mathbb{N} \to \mathsf{Term} \to \mathsf{Set}$, such that $\mathsf{Sz}\ n\ t$ holds whenever all free indices in t are $< n$.

$$\frac{}{\mathsf{Sz}\ n\ (\mathsf{Lev}\ x)} \qquad \frac{x \leq n}{\mathsf{Sz}\ (\mathsf{suc}\ n)\ (\mathsf{Idx}\ x)} \qquad \frac{\mathsf{Sz}\ (\mathsf{suc}\ n)\ t}{\mathsf{Sz}\ n\ (\mathsf{Lam}\ t)} \qquad \frac{\mathsf{Sz}\ n\ t \qquad \mathsf{Sz}\ n\ s}{\mathsf{Sz}\ n\ (t \cdot s)}$$

$$\frac{}{\mathsf{Sz}\ n\ \mathsf{Zero}} \qquad \frac{\mathsf{Sz}\ n\ t}{\mathsf{Sz}\ n\ (\mathsf{Succ}\ t)} \qquad \frac{\mathsf{Sz}\ n\ z \qquad \mathsf{Sz}\ n\ s \qquad \mathsf{Sz}\ n\ t}{\mathsf{Sz}\ n\ (\mathsf{Rec}\ z\ s\ t)}$$

A term t is then closed w.r.t. local variables, i.e. any De Bruijn index, whenever $\mathsf{Sz}\ 0\ t$ holds: the only possible way to have all indices below 0 is to have none. The Sz relation is decidable:

Lemma 12. For all $n : \mathbb{N}$ and $t : \mathsf{Term}$, $\mathsf{Sz}\ n\ t$ is decidable.

Proof. By induction on t. See `Syntax.Raw.Term.decSz`. □

We can observe a connection between well-typed terms under a certain locally-free context Γ and terms of a certain "size" of De Bruijn indices.

Lemma 13. If $\Gamma :: \Delta \vdash t : A$, then $\mathsf{Sz}\ |\Gamma|\ t$.

Proof. By induction on the derivation. See `Syntax.Typed.Typed.tyClosed`. □

We can now formally define what it means to be a (weak) redex, as well as a normal form. We define the inductive family $\mathsf{\beta\text{-}Redex} : \mathsf{Term} \to \mathsf{Term} \to \mathsf{Set}$ of β-redexes, and $\mathsf{N\text{-}Redex} : \mathsf{Term} \to \mathsf{Term} \to \mathsf{Term} \to \mathsf{Set}$ of Rec-redexes as follows:

$$\frac{\mathsf{Sz}\ 1\ t \qquad \mathsf{Sz}\ 0\ s}{\mathsf{\beta\text{-}Redex}\ (\mathsf{Lam}\ t)\ s} \qquad \frac{\mathsf{Sz}\ 0\ z \qquad \mathsf{Sz}\ 0\ s}{\mathsf{N\text{-}Redex}\ z\ s\ \mathsf{Zero}} \qquad \frac{\mathsf{Sz}\ 0\ z \qquad \mathsf{Sz}\ 0\ s \qquad \mathsf{Sz}\ 0\ t}{\mathsf{N\text{-}Redex}\ z\ s\ (\mathsf{Succ}\ t)}$$

Notice that our characterization of well-typed weak redexes as those redexes that are typeable under an empty locally-free context is sound. In fact, Lemma 13 ensures that those redexes are such that the predicate $\mathsf{Sz}\ 0$ holds, as required by the definition above.

We are now ready to define normal forms. We do so by giving a mutual definition of the inductive families $\mathsf{Nf} : \mathsf{Term} \to \mathsf{Set}$ of normal terms and $\mathsf{Ne} : \mathsf{Term} \to \mathsf{Set}$ of *neutral* terms.[1]

$$\frac{\mathsf{Nf}\ t}{\mathsf{Nf}\ (\mathsf{Lam}\ t)} \qquad \frac{}{\mathsf{Nf}\ \mathsf{Zero}} \qquad \frac{\mathsf{Nf}\ t}{\mathsf{Nf}\ (\mathsf{Succ}\ t)} \qquad \frac{\mathsf{Ne}\ t}{\mathsf{Nf}\ t}$$

$$\frac{\mathsf{Ne}\ t \quad \mathsf{Nf}\ s \quad \mathsf{Sz}\ 0\ t \quad \mathsf{Sz}\ 0\ s}{\mathsf{Ne}\ (t \cdot s)} \qquad \frac{\mathsf{Nf}\ t \quad \mathsf{Nf}\ s \quad \neg\mathsf{Sz}\ 0\ t}{\mathsf{Ne}\ (t \cdot s)} \qquad \frac{\mathsf{Nf}\ t \quad \mathsf{Nf}\ s \quad \mathsf{Sz}\ 0\ t \quad \neg\mathsf{Sz}\ 0\ s}{\mathsf{Ne}\ (t \cdot s)}$$

$$\frac{\mathsf{Nf}\ z \quad \mathsf{Nf}\ s \quad \mathsf{Ne}\ t \quad \mathsf{Sz}\ 0\ z \quad \mathsf{Sz}\ 0\ s \quad \mathsf{Sz}\ 0\ t}{\mathsf{Ne}\ (\mathsf{Rec}\ z\ s\ t)} \qquad \frac{\mathsf{Nf}\ z \quad \mathsf{Nf}\ s \quad \mathsf{Nf}\ t \quad \neg\mathsf{Sz}\ 0\ z}{\mathsf{Ne}\ (\mathsf{Rec}\ z\ s\ t)}$$

$$\frac{\mathsf{Nf}\ z \quad \mathsf{Nf}\ s \quad \mathsf{Nf}\ t \quad \mathsf{Sz}\ 0\ z \quad \neg\mathsf{Sz}\ 0\ s}{\mathsf{Ne}\ (\mathsf{Rec}\ z\ s\ t)} \qquad \frac{\mathsf{Nf}\ z \quad \mathsf{Nf}\ s \quad \mathsf{Nf}\ t \quad \mathsf{Sz}\ 0\ z \quad \mathsf{Sz}\ 0\ s \quad \neg\mathsf{Sz}\ 0\ t}{\mathsf{Ne}\ (\mathsf{Rec}\ z\ s\ t)}$$

Neutral terms were introduced in [13] to denote *stuck* terms, and in particular elimination forms whose computation is blocked by a variable (or another neutral term) in recursive position, like in the application $x\ t$. In our setting, we need to extend the notion of neutral terms beyond the standard one, because there are strictly more cases in which CH-weak reduction can be blocked. In particular, in addition to every neutral term of the full β-reduction, we have all non-weak redexs, that is, elimination forms where $\mathsf{Sz}\ 0\ t$ does not hold for at least one component t.[2]

This distinction between weak and non-weak redexes inevitably makes the definition of CH-weak normal forms slightly more complicated than the usual one for full β-reduction. For this reason, it may not be obvious that the predicate Nf we just defined

[1]In the definition of neutral terms, we make use of a type $\neg\mathsf{Sz}$. $\neg\mathsf{Sz}\ n\ t$ is exactly the same as $\neg(\mathsf{Sz}\ n\ t)$, only given as an inductive definition, so that it can be pattern matched.

[2]A trivial example of non-weak redex is $(\lambda \mathsf{v}_1)\mathsf{Zero}$

does really identify CH-weak normal forms. To verify this, we first need to formalize what it means to be a non-reducible term, which means that we must formally define the untyped notion of CH-weak reduction. We do so by inductively defining the binary relation $\_ \longrightarrow_{ch} \_ : \mathsf{Term} \to \mathsf{Term} \to \mathsf{Set}$ as follows

$$\frac{\beta\text{-}\mathsf{Redex}\ (\lambda t)\ s}{\lambda t \cdot s \longrightarrow_{ch} t[s]}\ (\beta) \qquad \frac{t \longrightarrow_{ch} t'}{\lambda t \longrightarrow_{ch} \lambda t'}\ (\xi) \qquad \frac{t \longrightarrow_{ch} t'}{t \cdot s \longrightarrow_{ch} t' \cdot s}\ (\nu) \qquad \frac{s \longrightarrow_{ch} s'}{t \cdot s \longrightarrow_{ch} t \cdot s'}\ (\mu)$$

$$\frac{}{\mathsf{Rec}\ z\ s\ \mathsf{Zero} \longrightarrow_{ch} z} \qquad \frac{}{\mathsf{Rec}\ z\ s\ (\mathsf{Succ}\ t) \longrightarrow_{ch} s \cdot t \cdot (\mathsf{Rec}\ z\ s\ t)}$$

$$\frac{t \longrightarrow_{ch} t'}{\mathsf{Succ}\ t \longrightarrow_{ch} \mathsf{Succ}\ t'} \qquad \frac{z \longrightarrow_{ch} z'}{\mathsf{Rec}\ z\ s\ t \longrightarrow_{ch} \mathsf{Rec}\ z'\ s\ t} \qquad \frac{s \longrightarrow_{ch} s'}{\mathsf{Rec}\ z\ s\ t \longrightarrow_{ch} \mathsf{Rec}\ z\ s'\ t}$$

$$\frac{t \longrightarrow_{ch} t'}{\mathsf{Rec}\ z\ s\ t \longrightarrow_{ch} \mathsf{Rec}\ z\ s\ t'}$$

We can now characterize normal forms, i.e. terms t such that $\mathsf{Nf}\ t$ holds, in the following way

Theorem 8. Let $t : \mathsf{Term}$. If $\mathsf{Nf}\ t$, then there exists no s such that $t \longrightarrow_{ch} s$;

Proof. By induction on the proof of $\mathsf{Nf}\ t$ and $t \longrightarrow_{ch} s$ for some s.
See `Syntax.Evaluation.Conversion.ifNf`. □

5.1.2 Semantic domain and interpretation

Normalization by Evaluation is a semantic method that works by interpreting the syntax into a suitable model, whose semantic values stand for representations of normal forms. One possible choice, that we will use for System T^{ex}, is a syntactic model of normal forms. In this setting, semantic values clearly stand for normal forms (they literally are), and reification is essentially the identity function. Semantic values are given by the type $\Sigma(t : \mathsf{Term})(\mathsf{Nf}\ t)$ of terms in normal form.

The evaluation function of Section 3.3 takes untyped terms as input, and produces normal forms as output, so it is a perfect candidate for the interpretation function. We denote this function by $[\![\_]\!] : \mathsf{Term} \to \mathsf{Env} \to \mathsf{Env} \to \Sigma(t : \mathsf{Term})(\mathsf{Nf}\ t)$, which, according to its signature, should take as input a term t and two *environments*, that is, maps assigning semantic values to globally-free and locally-free variables, respectively. However, notice that variables with global role do not interact in any way with evaluation; substitution is always performed on local variables, and the only context that changes (for example, when normalizing under a binder) is the locally-free one. Variables with global role end up being essentially constant terms. We can simplify things and really treat them as constants, thus removing the need for an environment in the interpretation function.

Locally-free variables, instead, do need an environment. These variables are represented with De Bruijn indices, so we should expect the same shifting operations that are needed for regular substitution to be needed for environments also. In fact, since semantic values are just regular terms (in normal form), we can avoid repeating ourselves and just represent environments as regular substitutions, for which we have already defined and formalized all the shifting and weakening operations that we need.

Finally, notice that having $\Sigma(t : \mathsf{Term})(\mathsf{Nf}\ t)$ as return type of $[\![\_]\!]$ would require annoying packing and unpacking of the Σ-type, as well as force us to provide proofs of $\mathsf{Nf}\ t$ for every t produced, whenever the function is used in recursive position. This additional syntactic noise would end up cluttering the actual algorithm at the core of the interpretation function. What we can do is to simply return terms, that is, define interpretation as a function of type $\mathsf{Term} \to \mathsf{Subst} \to \mathsf{Term}$, and then prove *a posteriori* that every result of interpretation is indeed a normal term.

The following is a sketch of interpretation function, mutually defined with semantic application and recursion. These are in turn defined according to our informal understanding of CH-weak reduction: if the term is a redex, check whether it is a *weak* one. If so, perform a step of reduction, and proceed evaluating the result; otherwise, produce a neutral term. Semantic application and recursion make use of three functions dec-β-Redex, dec-N-Redex-Z, and dec-N-Redex-S, that decide whether the redex under inspection is weak.

```
⟦_⟧ : Term → Subst → Term
⟦ Free x ⟧ ρ = Free x
⟦ Bound x ⟧ ρ = sub-var x ρ
⟦ Lam t ⟧ ρ = Lam (⟦ t ⟧ (sh ρ))
⟦ t · s ⟧ ρ = ⟦ t ⟧ ρ • ⟦ s ⟧ ρ
⟦ Zero ⟧ ρ = Zero
⟦ Succ t ⟧ ρ = Succ (⟦ t ⟧ ρ)
⟦ Rec z s t ⟧ ρ = rec (⟦ z ⟧ ρ) (⟦ s ⟧ ρ) (⟦ t ⟧ ρ)

_•_ : Term → Term → Term
Lam t • s with dec-β-Redex t s
(Lam t • s) | inj₁ x = ⟦ t ⟧ (Id , s)
(Lam t • s) | inj₂ y = Lam t · s
t • s = t · s

rec : Term → Term → Term → Term
rec z s Zero with dec-N-Redex-Z z s
rec z s Zero | inj₁ _ = z
rec z s Zero | inj₂ _ = Rec z s Zero
rec z s (Succ t) with dec-N-Redex-S z s t
rec z s (Succ t) | inj₁ x = z • s • rec z s t
rec z s (Succ t) | inj₂ y = Rec z s (Succ t)
rec z s t = Rec z s t
```

The definition above is conceptually correct, but it is readily rejected by Agda's termination checker, and for very good reasons: the function is defined on untyped terms, which clearly do *not* always have a normal form.[3] The three definitions above give rise to *partial* operations. We are working in a strongly-normalizing metatheory, which means that all functions must be total. However, it is still possible to express partial functions, by encoding their graph as an inductively-defined functional relation. We mutually define the relations

$$\begin{aligned}
[\![\_]\!]\_ \searrow \_ &: \mathsf{Term} \to \mathsf{Subst} \to \mathsf{Term} \to \mathsf{Set} \\
\_ \bullet \_ \searrow \_ &: \mathsf{Term} \to \mathsf{Term} \to \mathsf{Term} \to \mathsf{Set} \\
\mathsf{rec}\_ \cdot \_ \cdot \_ \searrow \_ &: \mathsf{Term} \to \mathsf{Term} \to \mathsf{Term} \to \mathsf{Term} \to \mathsf{Set}
\end{aligned}$$

starting with the inductive definition of application and recursion:

$$\frac{\beta\text{-}\mathsf{Redex}\ (\mathsf{Lam}\ t)\ s \qquad [\![t]\!]\ (\mathsf{Id}, s) \searrow a}{\mathsf{Lam}\ t \bullet s \searrow a}\ (\bullet\beta) \qquad \frac{\mathsf{Ne}\ (t \cdot s)}{t \bullet s \searrow t \cdot s}\ (\bullet\mathsf{Ne})$$

$$\frac{\mathsf{N\text{-}Redex}\ z\ s\ \mathsf{Zero}}{\mathsf{Rec}\ z\ \cdot\ s\ \cdot\ \mathsf{Zero} \searrow z}\ (\mathsf{rZ}) \qquad \frac{\begin{array}{cc} s \bullet t \searrow f & f \bullet a \searrow b \\ \mathsf{N\text{-}Redex}\ z\ s\ (\mathsf{Succ}\ t) & \mathsf{Rec}\ z\ \cdot\ s\ \cdot\ t \searrow a \end{array}}{\mathsf{Rec}\ z\ \cdot\ s\ \cdot\ \mathsf{Succ}\ t \searrow b}\ (\mathsf{rS})$$

We now need to define $[\![\_]\!]\_ \searrow \_$. Notice that, according to the sketched definition above, the result of interpreting a term t under an environment ρ produces exactly the same value[4] that we would get if we first applied ρ as a substitution to t, and then interpreted the result in the identity environment. That is

$$[\![t]\!]\ \rho = [\![\mathsf{sub}\ t\ \rho]\!]\ \mathsf{Id}$$

Interpreting under the identity environment is the same as evaluating open terms to normal form. Evaluation is a partial function that depends only on the term that needs to be evaluated, and can be defined as an inductive relation $\mathsf{Eval}\_ \searrow \_ : \mathsf{Term} \to \mathsf{Term} \to \mathsf{Set}$. We can then avoid introducing environments altogether, and express interpretation as substitution, followed by evaluation

$$[\![t]\!]\ \rho \searrow a :\equiv \mathsf{Eval}\ (\mathsf{sub}\ t\ \rho) \searrow a$$

With this trick, we get to reuse a lot of lemmas on the sub function to prove properties of interpretation. The graph of evaluation is inductively defined as follows:

$$\frac{}{\mathsf{Eval}\ \mathsf{v}_x \searrow \mathsf{v}_x} \qquad \frac{}{\mathsf{Eval}\ \mathsf{x}_x \searrow \mathsf{x}_x} \qquad \frac{}{\mathsf{Eval}\ \mathsf{Zero} \searrow \mathsf{Zero}}$$

$$\frac{\mathsf{Eval}\ t \searrow t'}{\mathsf{Eval}\ \mathsf{Succ}\ t \searrow \mathsf{Succ}\ t'} \qquad \frac{\mathsf{Eval}\ t \searrow t'}{\mathsf{Eval}\ \lambda t \searrow \lambda t'}$$

[3]Consider the archetypal diverging combinator $\Omega \equiv (\lambda x.xx)(\lambda x.xx)$, which diverges under full β-reduction, as well as CH-weak reduction.

[4]although it may not be equivalent from a complexity point of view.

$$\frac{\mathsf{Eval}\ t \searrow t' \qquad \mathsf{Eval}\ s \searrow s' \qquad t' \bullet s' \searrow a}{\mathsf{Eval}\ t \cdot s \searrow a} \qquad \frac{\mathsf{Eval}\ t \searrow t' \qquad \dfrac{\mathsf{Eval}\ z \searrow s' \qquad \mathsf{Eval}\ s \searrow s'}{\mathsf{Rec}\ z' \ \cdot\ s' \ \cdot\ t' \searrow a}}{\mathsf{Eval}\ \mathsf{Rec}\ z\ s\ t \searrow a}$$

We can see that these relations indeed define functional relations:

Lemma 14. For all terms t, s, z, a, b, the following hold

1. If $t \bullet s \searrow a$ and $t \bullet s \searrow b$, then $a = b$;
2. If $\mathsf{Rec}\ z\ \cdot\ s\ \cdot\ t \searrow a$ and $\mathsf{Rec}\ z\ \cdot\ s\ \cdot\ t \searrow b$, then $a = b$;
3. If $\mathsf{Eval}\ t \searrow a$ and $\mathsf{Eval}\ t \searrow b$, then $a = b$;
4. If $[\![t]\!]\ \rho \searrow a$ and $[\![t]\!]\ \rho \searrow b$, then $a = b$.

Proof. Points 1–3 are shown by mutual induction on the proof of the relation. See `•-fun`, `rec-fun`, and `Eval-fun` in module `Syntax.Evaluation.Properties`. Point 4 follows immediately from point 3. □

Moreover, evaluation interacts with normal forms as one would expect:

Lemma 15. For all z, s, t, a : Term,

1. If Nf t, Nf s, and $t \bullet s \searrow a$, then Nf a holds;
2. If Nf z, Nf s, Nf t, and $\mathsf{Rec}\ z\ \cdot\ s\ \cdot\ t \searrow a$, then Nf a holds;
3. If $\mathsf{Eval}\ t \searrow a$, then Nf a holds;
4. If Nf t holds, then $\mathsf{Eval}\ t \searrow t$.

Proof. Points 1–3 are shown by induction on the proof of the relation. See `nfApp`, `nfRec`, and `nfEval` in module `Syntax.Evaluation.Properties`. Point 4 is proved by induction on the proof of Nf t. See `Syntax.Evaluation.Properties.nfSelf`. □

The following lemmas formalize the intuitive fact that evaluation does not create new free indices.

Lemma 16.

1. If Sz n t, Sz n s, and $t \bullet s \searrow a$, then Sz n a;
2. If Sz n z, Sz n s, Sz n t, and $\mathsf{Rec}\ z\ \cdot\ s\ \cdot\ t \searrow a$, then Sz n a;
3. If Sz n t and $\mathsf{Eval}\ t \searrow a$, then Sz n a.

Proof. By mutual induction on the proof of the application, recursion, and evaluation relations. See •-Tm, rec-Tm, and Eval-Tm in module `Syntax.Evaluation.Properties` for the details. □

and it is compatible with weakening:

Lemma 17. Let $w : \mathsf{Wk}$. Then, the following hold

1. If $t \bullet s \searrow a$, then $\mathsf{wk}\ t\ w \bullet \mathsf{wk}\ s\ w \searrow \mathsf{wk}\ a\ w$;
2. If $\mathsf{Rec}\ z\ \cdot\ s\ \cdot\ t \searrow a$, then $\mathsf{Rec}\ \mathsf{wk}\ z\ w\ \cdot\ \mathsf{wk}\ s\ w\ \cdot\ \mathsf{wk}\ t\ w \searrow \mathsf{wk}\ a\ w$;
3. If $\mathsf{Eval}\ t \searrow a$, then $\mathsf{Eval}\ \mathsf{wk}\ t\ w \searrow \mathsf{wk}\ a\ w$.

Proof. All points proved by mutual induction on the proof terms. See the module `Syntax.Evaluation.Properties`. □

Finally, we can establish several ways in which evaluation commutes with substitution.

Lemma 18.

1. For all $t, s, a, b : \mathsf{Term}$ and $n : \mathbb{N}$, if $\mathsf{Eval}\ \mathsf{sub}\ t\ (\mathsf{shift}\ n\ (\mathsf{Id}, a)) \searrow b$ and $\mathsf{Eval}\ s \searrow a$, then $\mathsf{Eval}\ \mathsf{sub}\ t\ (\mathsf{shift}\ n\ (\mathsf{Id}, s)) \searrow b$;
2. For all t, a, b, σ, if $\mathsf{Eval}\ t \searrow a$ and $\mathsf{Eval}\ \mathsf{sub}\ t\ \sigma \searrow b$, then $\mathsf{Eval}\ \mathsf{sub}\ a\ \sigma \searrow b$;
3. For all t, a, b, σ, σ', if $\mathsf{Eval}\ \mathsf{sub}\ t\ \sigma \searrow b$ and $\mathsf{Eval}\ \mathsf{sub}\ t\ (\sigma \cdot^s \sigma') \searrow a$, then $\mathsf{Eval}\ \mathsf{sub}\ b\ \sigma' \searrow a$;
4. For all t, a, b, σ, if $\mathsf{Eval}\ t \searrow a$ and $\mathsf{Eval}\ \mathsf{sub}\ a\ \sigma \searrow b$, then $\mathsf{Eval}\ \mathsf{sub}\ t\ \sigma \searrow b$;
5. For all t, a, b, σ, σ', if $\mathsf{Eval}\ \mathsf{sub}\ t\ \sigma \searrow a$ and $\mathsf{Eval}\ \mathsf{sub}\ a\ \sigma' \searrow b$, then $\mathsf{Eval}\ \mathsf{sub}\ t\ (\sigma \cdot^s \sigma') \searrow b$.

Proof. By tedious but straightforward induction on terms and proofs of the evaluation relation. See `Syntax.Evaluation.Properties.sub-comm2` for the first point, and the module `Syntax.Evaluation.SubSwap` for the rest. □

5.2 Subset model and completeness of NbE

Completeness of NbE amount to showing that if two well-typed terms are definitionally equal, then their interpretation produces equal semantic values, according to the notion of equality of the particular model. We will continue with the syntactic model of normal forms described in the previous section, refining it into a *subset model*, that is, one where syntactic types are interpreted as suitable "*subsets*" of all the normal forms. [5] Semantic equality in this model of normal forms is syntactic identity. In our setting, completeness of NbE will imply that evaluation is terminating on well-typed terms, and that definitionally equal terms have the same normal form.

[5]Subsets are usually not a primitive notion in intentional type theory, including our metatheory (hence the quotation marks). Rather, the "subset" of a type A is represented as a predicate $P : A \to \mathsf{Set}$ such that $P\ x$ is inhabited for all $x : A$ belonging to the subset P.

5.2.1 Semantic types

As discussed in the previous section, we simply consider terms as the result of interpretation. However, we shall be a little more clear, at least typographically, and distinguish between the type of semantic values D as produced by evaluation, and that of syntactic terms Term. Nevertheless, $\mathsf{D} :\equiv \mathsf{Term}$.

Evaluation is a partial operation, i.e., there are terms t for which it is not possible to find a term a and environment ρ such that $[\![t]\!]\ \rho \searrow a$ holds. This is not only for the presence of diverging terms among the untyped λ-terms, but also because of potentially bogus arguments. For example, $\mathsf{Zero} \bullet \mathsf{Zero} \searrow a$ does not hold for any instantiation of a, because $\mathsf{Zero} \cdot \mathsf{Zero}$ is not a valid application, at least from a typed perspective.

We then follow [8] and strengthen our syntactic model, by identifying "subsets" of values on which application and recursion are well-defined and terminating, called *semantic types*.

Definition 2 (Semantic type)**.** A *semantic type* is a predicate $\mathcal{A} : \mathsf{D} \to \mathsf{Set}$ such that

1. For all $d : \mathsf{D}$, if $\mathcal{A}\ d$ then $\mathsf{Nf}\ d$;
2. For all $e : \mathsf{D}$, if $\mathsf{Ne}\ e$ then $\mathcal{A}\ e$.

That is, elements of a semantic type are is normal form. Moreover, all neutral forms inhabit any semantic type. From now on, we sometimes write $a \in_t \mathcal{A}$ to say that a is element of the semantic type $\mathcal{A}$, i.e., that $\mathcal{A}\ a$ holds. Moreover, we sometimes use $\mathcal{P}(\mathsf{D})$ as an informal abbreviation for $\mathsf{D} \to \mathsf{Set}$.[6] We now define a semantic type for each syntactic type, starting from natural numbers. The semantic type $\mathsf{Nat} : \mathsf{D} \to \mathsf{Set}$ is inductively defined as follows[7]:

$$\frac{}{\mathsf{Nat\ Zero}} \qquad \frac{\mathsf{Nat}\ d}{\mathsf{Nat\ (Succ}\ d)} \qquad \frac{\mathsf{Ne}\ e}{\mathsf{Nat}\ e}$$

In fact, the natural numbers for which recursion is well-defined and terminating are precisely neutral terms, and numerals produced by Zero and finite applications of of the successor. It is easy to see that Nat does indeed define a semantic type. The function type is a little bit more involved. We first need to define a relation on semantic applications belonging to a semantic type. Let $f, a : \mathsf{D}$ and $\mathcal{B} : \mathsf{D} \to \mathsf{Set}$. Then

$$f \bullet a\ \in_t\ \mathcal{B} :\equiv \Sigma(b : \mathsf{D})(b \in_t \mathcal{B} \wedge f\ \bullet\ a \searrow b)$$

Thus, given $\mathcal{A}, \mathcal{B} : \mathsf{D} \to \mathsf{Set}$, we define the semantic function space $(\mathcal{A} \to \mathcal{B}) : \mathsf{D} \to \mathsf{Set}$ as follows

$$\mathcal{A} \to \mathcal{B} :\equiv \lambda f.\mathsf{Nf}\ f \wedge (\forall \{a\ w\} \to a \in_t \mathcal{A} \to \mathsf{wk}\ f\ w \bullet a\ \in_t\ \mathcal{B})$$

[6] Recall that a function from D to Set can be viewed as a subset of D. Hence, it makes sense to consider $\mathsf{D} \to \mathsf{Set}$ as the subset space of D.

[7] In the Agda code, this is instead called NatP.

that is, $\mathcal{A} \to \mathcal{B}$ is the subset of all terms that are in normal form, and such that given an argument in $\mathcal{A}$, their semantic application to it is well-defined, terminating, and produces a result in $\mathcal{B}$. The weakening on f makes the subset model very similar to a Kripke model, and is needed to enable weakening of arbitrary semantic values.

Observe that $\mathcal{A} \to \mathcal{B}$ is a semantic type whenever $\mathcal{A}$ and $\mathcal{B}$ are. In fact, if $f \in_t \mathcal{A} \to \mathcal{B}$, then $\mathsf{Nf}\ f$ holds by definition. Moreover, if $\mathsf{Ne}\ f$, then $f \in_t \mathcal{A} \to \mathcal{B}$ because then $f \cdot a$ is a neutral term for any $a \in_f \mathcal{A}$, hence $f \bullet a \searrow f \cdot a$ holds for the neutral term $f \cdot a$, that is an element of $\mathcal{B}$ by definition of semantic type.

Let $\mathsf{rec}\_\cdot\_\cdot\_ \in_t \_ : \mathsf{D} \to \mathsf{D} \to \mathsf{D} \to \mathcal{P}(\mathsf{D}) \to \mathsf{Set}$ be the analogous of $\_\bullet\_ \in_t \_$ for recursion, defined as follows:

$$\mathsf{rec}\ z \cdot s \cdot t\ \in_t\ \mathcal{A} :\equiv \Sigma(d : \mathsf{D})(d \in_t \mathcal{A}\ \wedge\ \mathsf{Rec}\ z\ \cdot\ s\ \cdot\ t \searrow d)$$

then, we prove the following properties of semantic types:

Lemma 19. For all semantic types $\mathcal{A}, \mathcal{B}$,

1. If $f : \mathcal{A} \to \mathcal{B}$, and $a : \mathcal{A}$, then $f \bullet a\ \in_t\ \mathcal{B}$;

2. If $z : \mathcal{A}$, $s : \mathsf{Nat} \to \mathcal{A} \to \mathcal{A}$, and $t : \mathsf{Nat}$, then $\mathsf{rec}\ z \cdot s \cdot t\ \in_t\ \mathcal{A}$.

Proof. Point 1 follows almost immediately from the definition of $\mathcal{A} \to \mathcal{B}$ and its elements. To prove point 2, we discriminate on the decidable predicate $\mathsf{N\text{-}Redex}\ z\ s\ t$.

- Case $\mathsf{Rec}\ z\ s\ t$ is a redex, and $t \equiv \mathsf{Zero}$. Then $\mathsf{Rec}\ z\ \cdot\ s\ \cdot\ \mathsf{Zero} \searrow z$, and $z \in_t \mathcal{A}$ by hypothesis.

- Case $\mathsf{Rec}\ z\ s\ t$ is a redex, and $t \equiv \mathsf{Succ}\ n$ for some n. By inductive hypothesis, we have $p : \mathsf{rec}\ z\ \cdot\ s\ \cdot\ n\ \in_t\ \mathcal{A}$, and by point 1 of this lemma, we have $q : s\ \bullet\ n\ \in_t\ (\mathcal{A} \to \mathcal{A})$. Again, by point 1, we have $r : \pi_1\ q \bullet \pi_1\ p\ \in_t\ \mathcal{A}$. Putting it all together, we get $\mathsf{Rec}\ z\ \cdot\ s\ \cdot\ \mathsf{Succ}\ n \searrow \pi_1\ r$, and we conclude.

- Case $\mathsf{Ne}\ (\mathsf{Rec}\ z\ s\ t)$. Then $\mathsf{Rec}\ z\ s\ t \in_t \mathcal{A}$ by definition of semantic type.

□

5.2.2 Completeness of typing judgments

We interpret syntactic types $T : \mathsf{Ty}$ as semantic types. In particular, we define $[\![\_]\!]_t : \mathsf{Ty} \to \mathcal{P}(\mathsf{D})$ as $[\![\mathsf{N}]\!]_t = \mathsf{Nat}$ and $[\![A \Rightarrow B]\!]_t = [\![A]\!]_t \to [\![B]\!]_t$. Given a syntactic type A, the elements of $[\![A]\!]_t$ can be lifted according to arbitrary weakenings. The Kripke-like definition of the semantic function space is crucial for this proof to go through.

Lemma 20. For all $A : \mathsf{Ty}$, $w : \mathsf{Wk}$, $d : \mathsf{D}$, if $d \in_t [\![A]\!]_t$ then $\mathsf{wk}\ d\ w \in_t [\![A]\!]_t$.

Proof. By induction on A. See `Semantics.Completeness.Type.SemTy.liftD`. □

Just as semantic types are subsets of terms, we also interpret contexts $\Gamma : \mathsf{Ctxt}$ as subsets of substitutions, precisely those that map free indices to values of the corresponding semantic type. We define the interpretation of contexts as the inductive family $[\![\_]\!]_s : \mathsf{Ctxt} \to \mathsf{Subst} \to \mathsf{Set}$. Similarly to semantic types, we sometimes write $\rho \in_s [\![\Gamma]\!]_s$ for $[\![\Gamma]\!]_s\ \rho$.

$$\frac{\rho : \mathsf{Subst}}{[\![\diamond]\!]_s\ \rho}\ \mathsf{cId} \qquad \frac{\rho \in_s [\![\Gamma]\!]_s \qquad d \in_t [\![A]\!]_t}{[\![\Gamma, A]\!]_s\ (\rho, d)}\ \mathsf{cCons} \qquad \frac{\rho \in_s [\![\Gamma]\!]_s}{[\![\Gamma]\!]_s\ (\rho \cdot w)}\ \mathsf{cWk}$$

We now define what it means for two terms t, s of type T to have equal interpretations as elements of a semantic type $\mathcal{A}$.

Definition 3. Terms $t, s : \mathsf{Term}$ have equal interpretations as elements of semantic type $\mathcal{A}$ under an environment $\rho : \mathsf{Subst}$, written $[\![t]\!] \simeq [\![s]\!]\ \rho \in_t \mathcal{A}$, whenever there exists $d : \mathsf{D}$ such that $[\![t]\!]\ \rho \searrow d$, $[\![s]\!]\ \rho \searrow d$, and $d \in_t \mathcal{A}$.

In other words, $[\![t]\!] \simeq [\![s]\!]\ \rho \in_t \mathcal{A}$ holds whenever both t and s have normal forms, and these normal forms are the same. We can now prove the following semantic counterparts of System T^{ex} typing and conversion rules.

Lemma 21.

1. If $(a : [\![A]\!]_t \to [\![t]\!] \simeq [\![t']\!]\ (\rho \cdot w, a) \in_t [\![B]\!]_t)$, then $[\![\lambda t]\!] \simeq [\![\lambda t']\!]\ \rho \in_t [\![A \Rightarrow B]\!]_t$;
2. If $[\![\lambda t]\!] \simeq [\![\lambda t]\!]\ \rho \in_t [\![A \Rightarrow B]\!]_t$, $[\![s]\!] \simeq [\![s]\!]\ \rho \in_t [\![A]\!]_t$, $\mathsf{Sz}\ 1\ t$, and $\mathsf{Sz}\ 0\ s$, then $[\![\lambda t \cdot s]\!] \simeq [\![t[s]]\!]\ \rho \in_t [\![B]\!]_t$;
3. If $\Delta\ [n]\!\mapsto A$ and $\rho \in_s [\![\Delta]\!]_s$, then $[\![\mathsf{v}_n]\!] \simeq [\![\mathsf{v}_n]\!]\ \rho \in_t [\![A]\!]_t$;
4. If $[\![t]\!] \simeq [\![t']\!]\ \rho \in_t [\![\mathsf{N}]\!]_t$, then $[\![\mathsf{Succ}\ t]\!] \simeq [\![\mathsf{Succ}\ t']\!]\ \rho \in_t [\![\mathsf{N}]\!]_t$;
5. If $[\![t]\!] \simeq [\![t']\!]\ \rho \in_t [\![A \Rightarrow B]\!]_t$ and $[\![s]\!] \simeq [\![s']\!]\ \rho \in_t [\![A]\!]_t$, then $[\![t \cdot s]\!] \simeq [\![t' \cdot s']\!]\ \rho \in_t [\![B]\!]_t$;
6. If $[\![z]\!] \simeq [\![z']\!]\ \rho \in_t [\![A]\!]_t$, $[\![s]\!] \simeq [\![s']\!]\ \rho \in_t [\![\mathsf{N} \Rightarrow A \Rightarrow A]\!]_t$, and $[\![t]\!] \simeq [\![t']\!]\ \rho \in_t [\![\mathsf{N}]\!]_t$, then $[\![\mathsf{Rec}\ z\ s\ t]\!] \simeq [\![\mathsf{Rec}\ z'\ s'\ t']\!]\ \rho \in_t [\![A]\!]_t$;
7. If $[\![z]\!] \simeq [\![z]\!]\ \rho \in_t [\![A]\!]_t$, $[\![s]\!] \simeq [\![s]\!]\ \rho \in_t [\![\mathsf{N} \Rightarrow A \Rightarrow A]\!]_t$, then $[\![\mathsf{Rec}\ z\ s\ t]\!] \simeq [\![z]\!]\ \rho \in_t [\![A]\!]_t$;
8. If $[\![z]\!] \simeq [\![z]\!]\ \rho \in_t [\![A]\!]_t$, $[\![s]\!] \simeq [\![s]\!]\ \rho \in_t [\![\mathsf{N} \Rightarrow A \Rightarrow A]\!]_t$, $[\![t]\!] \simeq [\![t]\!]\ \rho \in_t [\![\mathsf{N}]\!]_t$, and $\mathsf{Sz}\ 0\ z$, $\mathsf{Sz}\ 0\ s$, $\mathsf{Sz}\ 0\ t$, then $[\![\mathsf{Rec}\ z\ s\ (\mathsf{Succ}\ t)]\!] \simeq [\![s \cdot t \cdot \mathsf{Rec}\ z\ s\ t]\!]\ \rho \in_t [\![A]\!]_t$.

Proof. We reproduce here the proof of only the first three points. See the module `Semantics.Completeness.Type.Lemmata` for the full details.

1. $[\![A]\!]_t$ is a semantic type, so $\mathsf{v}_0 \in_t [\![A]\!]_t$. Hence, by hypothesis, we have $[\![t]\!] \simeq [\![t']\!]\ \mathsf{sh}\ \rho \in_t [\![B]\!]_t$. This means that there exist $d : \mathsf{D}$ such that $[\![t]\!]\ \mathsf{sh}\ \rho \searrow d$, $[\![t']\!]\ \mathsf{sh}\ \rho \searrow d$, hence we have $[\![\lambda t]\!]\ \rho \searrow \lambda d$, and $[\![\lambda t']\!]\ \rho \searrow \lambda d$.

It is left to show that $\lambda d \in_t [\![A \Rightarrow B]\!]_t$. $\mathsf{Nf}\ (\lambda d)$ follows from $\mathsf{Nf}\ d$, which in turn follows from Lemma 15. We now have to show that for any $a : \mathsf{D}, w : \mathsf{Wk}$, if $a \in_t [\![A]\!]_t$ then $\mathsf{wk}\ (\lambda d)\ w \bullet a \in\!\mathsf{ap}\ [\![B]\!]_t$. Let w and a be such that $a \in_t [\![A]\!]_t$. We proceed by case analysis on the decidable proposition $\beta\text{-}\mathsf{Redex}\ (\mathsf{wk}\ (\lambda d)\ w)\ a$.

- Case $\beta\text{-}\mathsf{Redex}\ (\mathsf{wk}\ (\lambda d)\ w)\ a$ holds.
  By Lemma 17, from $[\![t]\!]\ \mathsf{sh}\ \rho \searrow d \equiv \mathsf{Eval\ sub}\ t\ (\mathsf{sh}\ \rho) \searrow d$ we have $\mathsf{Eval\ sub}\ t\ (\mathsf{sh}\ \rho \cdot \mathsf{Skip}\ w) \searrow \mathsf{wk}\ d\ (\mathsf{Skip}\ w)$.
  By hypothesis, $[\![t]\!] \simeq [\![t']\!]\ (\rho \cdot w, a) \in_t [\![B]\!]_t$, thus we have $[\![t]\!]\ (\rho \cdot w, a) \searrow d' \equiv \mathsf{Eval\ sub}\ t\ (\rho \cdot w, a) \searrow d'$, for some $d' \in_t [\![B]\!]_t$.
  By equality of substitutions and Lemma 18, we get
  $\mathsf{Eval\ sub}\ (\mathsf{wk}\ d\ (\mathsf{Skip}\ w))(\mathsf{Id}, a) \searrow d'$. But $\mathsf{wk}\ (\lambda d)\ w \equiv \lambda(\mathsf{wk}\ d\ (\mathsf{Skip}\ w))$, so $\mathsf{wk}\ (\lambda d)\ w \bullet a \searrow d'$. This yields a proof of $\mathsf{wk}\ (\lambda d)\ w \bullet a \in\!\mathsf{ap}\ [\![B]\!]_t$.
- Case $\beta\text{-}\mathsf{Redex}\ (\mathsf{wk}\ (\lambda d)\ w)\ a$ does not hold. Then $\mathsf{wk}\ (\lambda d)\ w \cdot a$ is a neutral term, so it is an element of $[\![B]\!]_t$ by definition of semantic type.

2. By point 4 of this lemma, we have $[\![t \cdot s]\!] \simeq [\![t \cdot s]\!]\ \rho \in_t [\![B]\!]_t$, which means that $[\![\lambda t \cdot s]\!]\ \rho \searrow d \equiv \mathsf{Eval\ sub}\ (\lambda t)\ \rho \cdot \mathsf{sub}\ s\ \rho \searrow d$ for some $d \in_t [\![B]\!]_t$.

   By induction on the proof of this evaluation, we must have $\mathsf{Eval\ sub}\ (\lambda t)\ \rho \searrow \lambda t'$ from $\mathsf{Eval\ sub}\ t\ (\mathsf{sh}\ \rho) \searrow t'$, as well as $\mathsf{Eval\ sub}\ s\ \rho \searrow s'$, for some $t', s' : \mathsf{D}$, such that $\lambda t' \bullet s' \searrow d$ holds. We distinguish the two possible derivations for this semantic application.

   - Case $\bullet\beta$. Then, it must be the case that $\mathsf{Eval\ sub}\ t'\ (\mathsf{Id}, s') \searrow d$. But then from this evaluation, in addition to $\mathsf{Eval\ sub}\ t\ (\mathsf{sh}\ \rho) \searrow t'$ and $\mathsf{Eval}\ afsub\ s\ \rho \searrow s'$, we get $[\![t[s]]\!]\ \rho \searrow d$ by Lemma 4 and Lemma 18. Recalling that $[\![\lambda d \cdot s]\!]\ \rho \searrow d$, we conclude.
   - Case $\bullet\mathsf{Ne}$. Then, $\lambda t' \cdot s'$ is a neutral redex, which means that either $\mathsf{Sz}\ 1\ t'$ or $\mathsf{Sz}\ 0\ s'$ must be false. But $\mathsf{Sz}\ 1\ t$ and $\mathsf{Sz}\ 0\ s$ hold by assumption, and so do $\mathsf{Sz}\ 1\ t'$ and $\mathsf{Sz}\ 0\ s'$ by Lemma 16., leading to a contradiction. The conclusion follows by ex falso quodlibet.

3. By induction on the proof of $\rho \in_s [\![\Delta]\!]_s$.

   - Case cId is impossible, since otherwise we would have $\Gamma = \diamond$, and $\diamond\ [n]\!\mapsto A$ is not provable;
   - Case cCons. Then $\rho \equiv (\rho', d)$ for $\rho' \in_s [\![\Delta]\!]_s$ and $d \in_t [\![A]\!]_t$.
     Then, $\Gamma, B\ [n]\!\mapsto A$ for some B. We proceed by case analysis on this.
     - Case $\Gamma, A\ [0]\!\mapsto A$. Since $\mathsf{Eval}\ d \searrow d$ by Lemma 15, we have $[\![\mathsf{v}_0]\!]\ \rho', d \searrow d$. Hence, $[\![\mathsf{v}_0]\!] \simeq [\![\mathsf{v}_0]\!]\ \rho', d \in_t [\![A]\!]_t$.
     - Case $\Gamma, B\ [\mathsf{suc}\ n']\!\mapsto A$ for some $n' : \mathbb{N}$, so that $\Gamma\ [n']\!\mapsto A$. By inductive hypothesis, $[\![\mathsf{v}_{n'}]\!] \simeq [\![\mathsf{v}_{n'}]\!]\ \rho' \in_t [\![d']\!]_t$ for some $d' \in_t [\![A]\!]_t$. But then $[\![\mathsf{v}_{(\mathsf{suc}\ n')}]\!] \simeq [\![\mathsf{v}_{(\mathsf{suc}\ n')}]\!]\ (\rho', d) \in_t [\![d']\!]_t$.

- Case cWk. Then $\rho \equiv \rho' \cdot w$. By inductive hypothesis, $[\![\mathsf{v}_n]\!] \simeq [\![\mathsf{v}_n]\!]\ \rho' \in_t [\![A]\!]_t$, meaning that we have $[\![\mathsf{v}_n]\!]\ \rho' \searrow d$ for some $d \in_t [\![A]\!]_t$. By Lemma 17, we have $[\![\mathsf{v}_n]\!]\ \rho' \cdot w \searrow \mathsf{wk}\ d\ w$, and by Lemma 20, we have $\mathsf{wk}\ d\ w \in_t [\![A]\!]_t$.

□

We now define semantic typing judgments in the following way:

$$\Delta \models t : A :\equiv \forall \{\rho : \mathsf{Subst}\} \to \rho \in_s [\![\Delta]\!]_s \to [\![t]\!] \simeq [\![t]\!]\ \rho \in_t [\![T]\!]_t$$

Notice that we only consider environments for the locally-free context, since global variables are treated as constants during evaluation, and therefore do not need an interpretation of their context. We can now prove completeness of the semantics w.r.t. typing judgments.

Theorem 9. For all $t : \mathsf{Term}$, $A : \mathsf{Ty}$, and $\Gamma, \Delta : \mathsf{Ctxt}$, if $\Gamma :: \Delta \vdash t : A$, then $\Delta \models t : A$.

Proof. By induction on the typing derivation, and application of Lemma 21. We consider the interesting case of λ-introduction. See the module `Semantics.Completeness.Rule` for the full details.

$$\frac{\Gamma :: \Delta, A \vdash t : B}{\Gamma :: \Delta \vdash \lambda t : A \Rightarrow B}$$

Let $\rho \in_s [\![\Delta]\!]_s$. We have to show that $[\![\lambda t]\!] \simeq [\![\lambda t]\!]\ \rho \in_t [\![A \Rightarrow B]\!]_t$. By inductive hypothesis, $\Delta, A \models t : B$, that is, for all $\rho' \in_s [\![\Delta, A]\!]_s$ we have $[\![t]\!] \simeq [\![t]\!]\ \rho' \in_t [\![B]\!]_t$.

Let $w : \mathsf{Wk}$, and $a : \mathsf{D}$ such that $a \in_t [\![A]\!]_t$. Then, $(\rho \cdot w, a) \in_s [\![\Delta, A]\!]_s$, hence $[\![t]\!] \simeq [\![t]\!]\ (\rho \cdot w, a) \in_t [\![B]\!]_t$ by inductive hypothesis. By Lemma 21, this implies $[\![\lambda t]\!] \simeq [\![\lambda t]\!]\ \rho \in_t [\![A \Rightarrow B]\!]_t$.

□

5.2.3 Completeness of reduction and equality judgments

We define semantic equality judgments in the following way

$$\Delta \models t \sim s : A :\equiv \forall \{\rho : \mathsf{Subst}\} \to \rho \in_s [\![\Delta]\!]_s \to [\![t]\!] \simeq [\![s]\!]\ \rho \in_t [\![A]\!]_t$$

Similarly to typing judgments, we restrict our attention to the locally-free context. We can now prove completeness of the semantics w.r.t. reduction and equality judgments:

Theorem 10. For all $t, s : \mathsf{Term}$, $A : \mathsf{Ty}$, and $\Gamma, \Delta : \mathsf{Ctxt}$, the following hold

1. if $\Gamma :: \Delta \vdash t \longrightarrow s : A$, then $\Delta \models t \sim s : A$;
2. if $\Gamma :: \Delta \vdash t \sim s : A$, then $\Delta \models t \sim s : A$.

Proof. Point 1 is proved by induction on the derivation, using Lemma 21. Point 2 follows easily by induction on the derivation, Theorem 9, and symmetry and transitivity of the relation $[\![\_]\!] \simeq [\![\_]\!]\ \rho \in_t [\![A]\!]_t$ for all ρ, A. □

Corollary 2. For all Γ, A, t, s,

1. If $\Gamma :: \Delta \vdash t : A$, then t has a normal form;
2. If $\Gamma :: \Delta \vdash t \sim s : A$, then t and s have the same normal form.

Proof. Point 1 follows from Theorem 9. Point 2 follows from Theorem 10. □

In concrete terms, this corollary means that we can define a metatheoretical normalization function nf, that is therefore total and computable, with the following type:

$$\mathsf{nf} : \Gamma :: \Delta \vdash t : A \to \mathsf{Term}$$

Notice how the function takes the whole proof of well-typedness of t as argument, from which is evident that normalization is only totally defined on well-typed System T^{ex} terms. Then, completeness of NbE means that for well-typed terms $p : \Gamma :: \Delta \vdash t : A$ and $q : \Gamma :: \Delta \vdash t : A$, the conversion $\Gamma :: \Delta \vdash t \sim s : A$ implies $\mathsf{nf}\ p = \mathsf{nf}\ q$.

5.3 Kripke logical relations and soundness of NbE

Soundness of NbE is the statement that well-typed terms are provably convertible with their normal forms, that is, it corresponds to the following implication:

$$(p : \Gamma :: \Delta \vdash t : A) \to \Gamma :: \Delta \vdash t \sim \mathsf{nf}\ p : A$$

To prove this statement we rely on the definition of a suitable *Kripke logical relation.* Logical relations are families of relations defined by induction on types. Kripke logical relations [36] are additionally indexed by a set of *worlds* together with an accessibility relation, in the sense of Kripke semantics. In our case, these roles are played by contexts Ctxt and context extension (weakening). In particular, if a relation holds in the context Δ, it also holds in every ∇ such that $\nabla \vdash_r w : \Delta$ for some w.

Our logical relation is $\_ :: \_ \vdash \_ ® \_ : \_ : \mathsf{Ctxt} \to \mathsf{Ctxt} \to \mathsf{Term} \to \mathsf{D} \to \mathsf{Ty} \to \mathsf{Set}$, and relates well-typed terms with semantic values. The idea is to define the relation in such a way that $\Gamma :: \Delta \vdash t ® a : A$ implies $\Gamma :: \Delta \vdash t \sim a : A$. It then remains to prove that every well-typed term is logically related to its interpretation in the semantics (i.e., just its normal form). This result is generally called *fundamental lemma of logical relations*, and in our case immediately implies soundness of NbE.

We now proceed to define the logical relation. Just like we did for semantic types in Section 5.2, we first define relations for each syntactic type $A : \mathsf{Ty}$ separately, telling us what it means for syntactic terms and semantic values to be logically-related *at type A*. In the case of natural numbers, this simply means convertibility. We define $\mathsf{NatRel} : \mathsf{Ctxt} \to \mathsf{Ctxt} \to \mathsf{Term} \to \mathsf{D} \to \mathsf{Set}$ as follows:

$$\frac{\Gamma :: \Delta \vdash n \sim \mathsf{Zero} : \mathsf{N}}{\mathsf{NatRel}\ \Gamma\ \Delta\ n\ \mathsf{Zero}} \qquad \frac{\Gamma :: \Delta \vdash n \sim \mathsf{Succ}\ n' : \mathsf{N} \qquad \mathsf{NatRel}\ \Gamma\ \Delta\ n'\ m}{\mathsf{NatRel}\ \Gamma\ \Delta\ n\ (\mathsf{Succ}\ m)}$$

$$\frac{\mathsf{Ne}\ e \qquad \Gamma :: \Delta \vdash n \sim e : \mathsf{N}}{\mathsf{NatRel}\ \Gamma\ \Delta\ n\ e}$$

Thus, a term n is related to Zero or to a neutral term if it is convertible to it, whereas it is related to Succ m if it converts to some successor Succ n' such that n' is related to m. Note that, by looking at this definition, it seems like we could just have put $\mathsf{NatRel}\ \Gamma\ \Delta\ n\ m :\equiv \Gamma :: \Delta \vdash n \sim m : \mathsf{N}$. However, we need the additional requirement that terms be related, and thus convertible, to semantic values m for which recursion is well-defined (that is, either numerals or neutral forms), otherwise it is not clear how to prove the second point of Lemma 27, showing that logical relations are preserved by recursion.

The logical relation at a function type $A \Rightarrow B$ is such that, given relations R_A and R_B telling how to relate terms and values at type A and B, it tells us how to relate functional terms and values at type $A \Rightarrow B$. Thus, suppose $R_A, R_B : \mathsf{Ctxt} \to \mathsf{Ctxt} \to \mathsf{Term} \to \mathsf{D} \to \mathsf{Ty} \to \mathsf{Set}$. Then, we inductively define the relation FunRel on type $A \Rightarrow B$ as follows:

$$\frac{\begin{array}{c}\Gamma :: \Delta \vdash t \sim \lambda t' : A \Rightarrow B \\ \forall\{\nabla\ w\ s\ a\ b\} \to \nabla \vdash_r w : \Delta \to R_A\ \Gamma\ \nabla\ s\ a \to [\![d]\!]\ (\mathsf{Id} \cdot w), a \searrow b \to R_B\ \Gamma\ \nabla\ (\mathsf{sub}\ t'\ ((\mathsf{Id} \cdot w), s))\ b\end{array}}{\mathsf{FunRel}\ A\ B\ R_A\ R_B\ \Gamma\ \Delta\ t\ (\lambda d)}$$

$$\frac{\mathsf{Ne}\ e \qquad \Gamma :: \Delta \vdash t \sim e : A \Rightarrow B}{\mathsf{FunRel}\ A\ B\ R_A\ R_B\ \Gamma\ \Delta\ t\ e}$$

That is, we say that a term t is related to a semantic value λd if t is convertible to some $\lambda t'$ such that for any term s and value a related at type A, the results of substituting into the bodies t' and d are related at type B. Since these substitutions model β-reductions, what this means is that t and λd are related as functions of type $A \Rightarrow B$ when they map related inputs to related outputs. Notice that, as customary in Kripke semantics, we consider the relation between inputs and outputs of functional terms in an arbitrary extended context ∇. This setup is crucial to be able to prove that the logical relation is Kripke, i.e. that is preserved by context extensions. On the other hand, t is related to a neutral term e if it is convertible to it.

We are now ready to define the logical relation, by induction on syntactic types, and using the specialized definition for each case:[8]

$$\begin{array}{l}
\_::\_\vdash\_\circledR\_:\_ \;:\; \mathsf{Ctxt} \to \mathsf{Ctxt} \to \mathsf{Term} \to \mathsf{D} \to \mathsf{Ty} \to \mathsf{Set} \\
\Theta :: \Gamma \vdash t \circledR a : \mathsf{N} = \mathsf{NatRel}\ \Theta\ \Gamma\ t\ a \\
\Theta :: \Gamma \vdash t \circledR a : (A \Rightarrow B) = \\
\quad \mathsf{FunRel}\ A\ B\ (\_::\_\vdash\_\circledR\_: A)\ (\_::\_\vdash\_\circledR\_: B)\ \Theta\ \Gamma\ t\ a
\end{array}$$

Traditionally, logical relations on function spaces have a simpler definition, that in our setting would translate to the following:

[8]In the definition of the logical relation, $\_ :: \_ \vdash \_ \circledR \_ : A$ is Agda notation for a partially applied function, whose inputs are represented by the underscored parts.

$$\Gamma :: \Delta \vdash r \circledR f : A \Rightarrow B :\equiv \nabla \vdash_r w : \Delta \to \Gamma :: \nabla \vdash s \circledR a : A \to f \bullet a \searrow b \to \Gamma :: \nabla \vdash r \cdot s \circledR b : B \tag{5.1}$$

This is a purely behavioural characterization of functions, as simply those terms that map related arguments to related results. Unfortunately, our calculus is *too intensional*, and we need the additional specification that functional terms in the semantics are related with functional terms in the syntax. This is what we achieve with the definition above, that requires terms t related to functional values λd to be themselves convertible to functions, i.e. $\Gamma :: \Delta \vdash t \sim \lambda t' : A \Rightarrow B$. Without this additional requirement, we would not be able to prove Lemma 23, as we further explain after its proof. A similar trick is used in Pagano's PhD thesis [51], for the same reasons.

A first property of logical relations is that they are compatible with judgmental equality, as shown in Lemma 22. We clearly also have compatibility with semantic equality, which is just identity of normal forms; thus, the property trivially follows by substitutivity of propositional equality.

Lemma 22. If $\Gamma :: \Delta \vdash t \sim t' : A$ and $\Gamma :: \Delta \vdash t' \circledR a : A$, then $\Gamma :: \Delta \vdash t \circledR a : A$.

Proof. By induction on A, and subsequently on the proof of the logical relation. See `Semantics.Soudness.LogicalRelation.`∼preservation. □

The main result that we want to obtain from our definition of logical relations is that related objects are convertible:

Lemma 23. If $\Gamma :: \Delta \vdash t \circledR a : T$ then $\Gamma :: \Delta \vdash t \sim a : T$.

Proof. By induction on T and on the logical relation. We report the most interesting case of $T \equiv A \Rightarrow B$, with the logical relation derived by ⇒-®-Lam. By hypothesis, $\Gamma :: \Delta \vdash t \sim \lambda t' : A \Rightarrow B$ and $\Gamma :: \Delta, A \vdash t' \circledR d : B$ for some t', d. By inductive hypothesis we have $\Gamma :: \Delta, A \vdash t' \sim d : B$, but then

$$\dfrac{\Gamma :: \Delta \vdash t \sim \lambda t' : A \Rightarrow B \qquad \dfrac{\Gamma :: \Delta, A \vdash t' \sim d : B}{\Gamma :: \Delta \vdash \lambda t' \sim \lambda d : A \Rightarrow B}\,(\xi)}{\Gamma :: \Delta \vdash t \sim \lambda d : A \Rightarrow B}\,\sim\mathsf{trans}$$

□

Now, suppose we defined the logical relation on function types as in 5.1, and wanted to prove that $\Gamma :: \Delta \vdash r \circledR f : A \Rightarrow B$ implies $\Gamma :: \Delta \vdash r \sim f : A \Rightarrow B$. We have no choice but to use the inductive hypothesis on $\Gamma :: \nabla \vdash r \cdot s \circledR b : B$, which comes from hypothesis for a suitable instantiation of ∇, w, s, a, b, and somehow try to conclude $\Gamma :: \Delta \vdash r \sim f : A \Rightarrow B$. The only possibility is to use $\Gamma :: \Delta, A \vdash \mathsf{v}_0 \circledR \mathsf{v}_0 : A$ to get $\Gamma :: \Delta, A \vdash \mathsf{wk}\ r\ (\mathsf{Up}\ \mathsf{Id}) \cdot \mathsf{v}_0 \sim \mathsf{wk}\ f\ (\mathsf{Up}\ \mathsf{Id}) \cdot \mathsf{v}_0 : B$, and then $\Gamma :: \Delta \vdash \lambda(\mathsf{wk}\ r\ (\mathsf{Up}\ \mathsf{Id}) \cdot \mathsf{v}_0) \sim \lambda(\mathsf{wk}\ f\ (\mathsf{Up}\ \mathsf{Id}) \cdot \mathsf{v}_0) : A \Rightarrow B$ by ξ. At this point, we are stuck, and the proof cannot proceed. If we had the η-rule, that with De Bruijn indices would be defined as

$$\frac{}{\Gamma :: \Delta \vdash \lambda(\mathsf{wk}\ t\ (\mathsf{Up}\ \mathsf{Id}) \cdot \mathsf{v}_0) \sim t : A \Rightarrow B}\ (\eta)$$

we could use it to successfully conclude $\Gamma :: \Delta \vdash r \sim f : A \Rightarrow B$ from $\Gamma :: \Delta \vdash \lambda(\mathsf{wk}\ r\ (\mathsf{Up}\ \mathsf{Id}) \cdot \mathsf{v}_0) \sim \lambda(\mathsf{wk}\ f\ (\mathsf{Up}\ \mathsf{Id}) \cdot \mathsf{v}_0) : A \Rightarrow B$. However, we do not have the η-rule in System T^{ex} allowing us to turn every functional term into a λ-abstraction, so we have to bake it into the definition of the logical relation, and explicitly require that related functional terms are always convertible to λ-abstractions.

The following lemma establishes that weakening is an accessibility relation for the Kripke logical relation:

Lemma 24. If $\nabla \vdash_r \Delta$ and $\Gamma :: \Delta \vdash t \circledR a : T$, then $\Gamma :: \nabla \vdash \mathsf{wk}\ t\ w \circledR \mathsf{wk}\ a\ w : T$.

Proof. By induction on T and the logical relation.
See `Semantics.Soudness.LogicalRelation.kripke`. □

Moreover, we have that terms are related to neutral values whenever they are convertible to them.

Lemma 25. If $\mathsf{Ne}\ e$ and $\Gamma :: \Delta \vdash t \sim a : T$, then $\Gamma :: \Delta \vdash t \circledR e : T$.

Proof. By straightforward induction on T. See `Semantics.Soudness.LogicalRelation.allNe`. □

The fact that logically-related terms are well-typed follows indirectly from the definition:

Lemma 26. If $\Gamma :: \Delta \vdash t \circledR a : T$, then $\Gamma :: \Delta \vdash t : T$.

Proof. By induction on T and the logical relation. □

Finally, we can show application and recursion for logical relations

Lemma 27.

- If $\Gamma :: \Delta \vdash r \circledR f : A \Rightarrow B$, $\Gamma :: \Delta \vdash s \circledR a : A$, and $f \bullet a \searrow b$, then $\Gamma :: \Delta \vdash r \cdot s \circledR b : B$.
- If $\Gamma :: \Delta \vdash z \circledR sz : A$, $\Gamma :: \Delta \vdash s \circledR ss : \mathsf{N} \Rightarrow A \Rightarrow A$, $\Gamma :: \Delta \vdash n \circledR sn : \mathsf{N}$, and $\mathsf{Rec}\ sz \cdot ss \cdot sn \searrow a$, then $\Gamma :: \Delta \vdash \mathsf{Rec}\ z\ s\ n \circledR a : A$.

Proof. Both proved by induction on the logical relation, respectively that on r for the first point and on pp$n$ for the second point. We reproduce the significant part of the first point below. See `Semantics.Soudness.LogicalRelation.appLemma` and `Semantics.Soudness.LogicalRelation.rekk` for the full details.

Consider the case where $f \bullet a \searrow b$ has been derived by $\bullet\beta$. Then, $f \equiv \lambda d$ for some d, and $\Gamma :: \Delta \vdash r \circledR f : A \Rightarrow B$ must have been derived by $\Rightarrow$-$\circledR$-Lam.

Since $\lambda d \cdot a$ is a weak redex, we know that $\mathsf{Sz}\ 1\ d$ and $\mathsf{Sz}\ 0\ a$. But by Lemma 23, $\Gamma :: \Delta, A \vdash t' \sim d : B$ and $\Gamma :: \Delta \vdash s \sim a : A$. We therefore get $\mathsf{Sz}\ 1\ t'$ and $\mathsf{Sz}\ 0\ s$, that imply $\Gamma :: A \vdash t' : B$ and $\Gamma :: \diamond \vdash s : A$.

Since $\Gamma :: \Delta \vdash r \sim \lambda t' :$ holds by hypothesis, we can conclude the following:

$$\dfrac{\dfrac{\Gamma :: \Delta \vdash r \sim \lambda t' : \qquad \dfrac{\Gamma :: \Delta \vdash s : A}{\Gamma :: \Delta \vdash s \sim s : A}}{\Gamma :: \Delta \vdash r \cdot s \sim \lambda t' \cdot s : B} \qquad \dfrac{\Gamma :: A \vdash t' : B \qquad \Gamma :: \diamond \vdash s : A}{\Gamma :: \Delta \vdash \lambda t' \cdot s \sim t'[s] : B}\,(\beta)}{\Gamma :: \Delta \vdash r \cdot s \sim t'[s] : B}$$

By Lemma 22, it now suffices to show $\Gamma :: \Delta \vdash t'[s] \circledR b : B$. Since $\lambda d \bullet a \searrow b$ holds by β-reduction, we must have $[\![d]\!]\ \mathsf{Id}, a \searrow b$. Moreover, by hypothesis of having used $\Rightarrow$-®-Lam, we know that for any ∇, w, s', a', b', having $\nabla \vdash_r w : \Delta, \Gamma :: \nabla \vdash s' \circledR a' : A, [\![d]\!]\ \mathsf{Id} \cdot w, a' \searrow b'$ implies $\Gamma :: \nabla \vdash t'[\mathsf{Id} \cdot w, s'] \circledR b' : B$. Thus, since $\Delta \vdash_r \mathsf{Id} : \Delta$ and $\Gamma :: \Delta \vdash s \circledR a : A$ is assumed, we get $\Gamma :: \Delta \vdash t'[\mathsf{Id}, s] \circledR b : B$.

□

5.3.1 Logical relation on substitutions

Before proving the fundamental lemma of logical relations, we must address substitutions, which in our case also serve as environments of semantic values. In particular, we define a logical relation $\_ :: \_ \vdash_s \_ \circledR \_ : \_ : \mathsf{Ctxt} \to \mathsf{Ctxt} \to \mathsf{Subst} \to \mathsf{Subst} \to \mathsf{Ctxt} \to \mathsf{Set}$ between substitutions σ and environments ρ such that σ and ρ are related whenever they are point-wise related, that is, they assign related objects to the same De Bruijn indices.

As with terms, we give the relation as an inductive definition, so that we can reveal the syntactic structure of the substitutions involved in the relation by performing case analysis of the proof of the relation.

$$\dfrac{\Gamma :: \Delta \vdash_s \sigma : \diamond \qquad \rho : \mathsf{Subst}}{\Gamma :: \Delta \vdash_s \sigma \circledR \rho : \diamond}\,(\diamond\circledR) \qquad \dfrac{\Gamma :: \Delta \vdash_s \sigma \circledR \rho : \nabla \qquad \Gamma :: \Delta \vdash t \circledR a : A}{\Gamma :: \Delta \vdash_s \sigma, t \circledR \rho, a : \nabla, A}\,(\#\circledR)$$

$$\dfrac{\Theta :: \Gamma \vdash_s \sigma \circledR \rho : \Delta \qquad \nabla \vdash_r w : \Gamma}{\Theta :: \nabla \vdash_s \sigma \cdot w \circledR \rho \cdot w : \Delta}\,(\mathsf{w}\circledR)$$

Lemma 28. If $\Gamma :: \nabla \vdash_s \sigma \circledR \rho : \Delta$, then $\Gamma :: \nabla \vdash_s \sigma : \Delta$.

Proof. By induction on the proof of the relation. □

Let us define the identity substitution $\mathsf{idsub} : \mathsf{Ctxt} \to \mathsf{Subst}$ as the substitution mapping all variables of a context to themselves.

$$\mathsf{idsub}\ \Gamma :\equiv \mathsf{shift}\ |\Gamma|\ \mathsf{Id}$$

We can see that the identity substitution and environment are logically-related.

Lemma 29. For all $\Delta : \mathsf{Ctxt}$, we have $\Gamma :: \Delta \vdash_s \mathsf{idsub}\ \Delta \circledR \mathsf{idsub}\ \Delta : \Delta$.

Proof. By induction on Δ and straightforward application of Lemma 25. □

5.3.2 Soundness of NbE

To support the proof of the fundamental lemma of logical relations below, we first show that the logical relation on substitutions and environments means what it is supposed to mean, i.e. that they logically-related substitutions and environments map the same De Bruijn index to related values. We do so in the following lemma, which generalizes this statement by considering it under an arbitrary well-typed weakening to achieve a sufficiently strong inductive hypothesis.

Lemma 30. If $\Gamma\ [n]\!\mapsto A$, $\mathsf{Eval\ wk\ (sub\text{-}var}\ n\ \rho)\ w \searrow a$, $\Theta :: \Delta \vdash_s \sigma \circledR \rho : \Gamma$, and $\nabla \vdash_r w : \Delta$, then $\Theta :: \nabla \vdash \mathsf{wk\ (sub\text{-}var}\ n\ \sigma)\ w \circledR a : A$.

Proof. By induction on the proof of $\Theta :: \Delta \vdash_s \sigma \circledR \rho : \Gamma$. We consider the case $\#\circledR$, when $n \equiv 0$ and $\Gamma', A\ [0]\!\mapsto A$ for some Γ'. Then, $\sigma \equiv (\sigma', t)$, $\rho \equiv (\rho, b)$ for some σ', ρ', t, b, and $\mathsf{Eval\ wk}\ b\ w \searrow a$. Then, $\Theta :: \Delta \vdash t \circledR b : A$ by hypothesis, and $\Theta :: \nabla \vdash \mathsf{wk}\ t\ w \circledR \mathsf{wk}\ b\ w : A$ by Lemma 24. Since $\mathsf{wk}\ b\ w$ is a normal form, then $\mathsf{Eval\ wk}\ b\ w \searrow \mathsf{wk}\ b\ w$, both by Lemma 15, so we deduce $\mathsf{wk}\ a\ w = a'$ by functionality of evaluation and conclude $\Theta :: \Delta \vdash t \circledR a : A$.

See `Semantics.Soundness.Soundness.varFundamental` for the full details. □

We are now ready to show the fundamental lemma of logical relations, namely that every well-typed term is logically-related to its interpretation in the semantics.

Lemma 31. [Fundamental lemma] If $\Theta :: \Gamma \vdash t : T$, $\Theta :: \Delta \vdash_s \sigma \circledR \rho : \Gamma$, and $[\![t]\!]\ \rho \searrow a$, then $\Theta :: \Delta \vdash \mathsf{sub}\ t\ \sigma \circledR a : T$.

Proof. By induction on the derivation of t. We show the most interesting case, where $T \equiv A \Rightarrow B$. Thus, we have $t \equiv \lambda t'$ and $d : \mathsf{D}$ such that

$$\frac{\Theta :: \Gamma, A \vdash t' : B}{\Theta :: \Gamma \vdash \lambda t' : A \Rightarrow B} \qquad\qquad \frac{[\![t']\!]\ \mathsf{sh}\ \rho \searrow d}{[\![\lambda t']\!]\ \rho \searrow \lambda d}$$

meaning that we have to show $\Theta :: \Delta \vdash \lambda(\mathsf{sub}\ t\ \mathsf{sh}\ \sigma) \circledR \lambda d : A \Rightarrow B$. This can be done by $\Rightarrow$-$\circledR$-Lam. $\mathsf{Nf}\ d$ follows from Lemma 15. It remains to show that for all ∇, w, s, a, b such that $\nabla \vdash_r w : \Delta$, $\Theta :: \nabla \vdash s \circledR a : A$, and $[\![d]\!]\ \mathsf{Id} \cdot w, a \searrow b$, we have $\Theta :: \nabla \vdash \mathsf{sub\ (sub}\ t'\ (\mathsf{sh}\ \sigma))(\mathsf{Id} \cdot w, s) \circledR b : B$.

Let us assume such ∇, w, s, a, b. By definition of the logical relation on substitutions, we get $\Theta :: \nabla \vdash_s \sigma \cdot w, s \circledR \rho \cdot w, a : \Gamma, A$. Moreover, by $[\![t']\!]\ \mathsf{sh}\ \rho \searrow d$, $[\![d]\!]\ \mathsf{Id} \cdot w, a \searrow b$, and Lemma 18, we get $[\![t']\!]\ \rho \cdot w, a \searrow b$. Hence, by inductive hypothesis we get $\Theta :: \nabla \vdash \mathsf{sub}\ t'\ (\sigma \cdot w, s) \circledR b : B$, with which we conclude since $\mathsf{sub\ (sub}\ t'\ (\mathsf{sh}\ \sigma))((\mathsf{Id} \cdot w), s) = \mathsf{sub}\ t'\ ((\sigma \cdot w), s)$ by equality of substitutions. □

Theorem 11. [Soundness of NbE] If $p : \Gamma :: \Delta \vdash t : A$, then $\Gamma :: \Delta \vdash t \sim \mathsf{nf}\ p : A$.

Proof. By completeness, $[\![t]\!]\ \mathsf{idsub}\ \Delta \searrow \mathsf{nf}\, p$. By Lemma 31, $\Gamma :: \Delta \vdash \mathsf{sub}\, t\ (\mathsf{idsub}\ \Delta) \circledR \mathsf{nf}\, p : A$, which is just $\Gamma :: \Delta \vdash t \circledR \mathsf{nf}\ p : A$ since $\mathsf{idsub}\ \Delta$ is an identity substitution. By Lemma 23, we get $\Gamma :: \Delta \vdash t \sim \mathsf{nf}\ p : A$. □

5.4 Normalization for System T^{wk}

We are finally ready to conclude the chapter with the normalization result for System T^{wk}. In particular, we will show in Theorem 13 below that for any well-typed System T^{wk} term t, there exists a definitionally equal term t' such that $\mathsf{Nf}\ t'$. We know from Theorem 8 that the predicate Nf correctly identifies System T^{ex} normal forms. However, this tells us nothing about the weak reduction relation of System T^{wk}. For Theorem 13 to really correspond to normalization for T^{wk}, we have to prove that Nf also correctly identifies normal forms according to T^{wk}'s reduction relation. The informal proof of equivalence of the two untyped formulations (Theorem 5) suggests that this should indeed be the case. We proceed to establish this result formally. The following definitions and lemmas can be found in the formalization at the module `Syntax.Evaluation.Conversion`. We start with the definition of T^{wk}'s reduction relation $\_ \longrightarrow_w \_$ in its untyped, nameless form.

$$\frac{}{\lambda t \cdot s \longrightarrow_w t[s]}\ (\beta) \qquad \frac{a \longrightarrow_w b}{t\langle n \mapsto a\rangle \longrightarrow_w t\langle n \mapsto b\rangle}\ (\sigma)$$

$$\frac{}{\mathsf{Rec}\ z\ s\ \mathsf{Zero} \longrightarrow_{ch} z} \qquad \frac{}{\mathsf{Rec}\ z\ s\ (\mathsf{Succ}\ t) \longrightarrow_{ch} s \cdot t \cdot (\mathsf{Rec}\ z\ s\ t)}$$

where all terms are always considered closed w.r.t. De Bruijn indices, since in T^{wk} they are only used for bound variables. We begin by observing that the substitution rule (σ) is admissible in CH-weak reduction, if we generalize to its reflexive-transitive closure.

Lemma 32. If $a \longrightarrow^*_{ch} b$, then $t\langle n \mapsto a\rangle \longrightarrow^*_{ch} t\langle n \mapsto b\rangle$.

Proof. By induction on t. □

We get the following result as an immediate consequence:

Lemma 33.

1. If $t \longrightarrow_w s$, then $t \longrightarrow^*_{ch} s$.
2. If $t \longrightarrow_w s$ and $t \neq s$, then there exists r such that $t \longrightarrow_{ch} r$.

Proof. The first point is proved by induction on the proof of $t \longrightarrow_w s$. The (β) case is immediate. The (σ) case relies on Lemma 32. The second point relies on the first, and case analysis on its result. □

The second point of the lemma above establishes the fact that "true" one-step reductions in $\longrightarrow_w$ imply one-step reductions in $\longrightarrow_{ch}$. Notice that we need the extra condition $t \neq s$ to ensure that we are actually dealing with one-step reductions. This is because $t \longrightarrow_w t$ is always derivable via (σ), by performing substitution over a variable that is not free in t.

Theorem 12. If $\mathsf{Nf}\ t$, then there exists no s such that $t \neq s$ and $t \longrightarrow_w s$.

Proof. Suppose we had $t \longrightarrow_w s$ for some s such that $s \neq t$. By Lemma 33, we must have $t \longrightarrow_{ch} r$ for some r. But then, we reach absurdity by Theorem 8. $\square$

Notice again the condition $t \neq s$, since $t \longrightarrow_w t$ is always derivable, even if t is a normal form. Having established that Nf does indeed also identify System T^w normal forms, we can proceed with the Normalization theorem.

Theorem 13. [Normalization] If $\Gamma \vdash t : A$, then there exists $t' : \mathsf{Term}$ such that $\mathsf{Nf}\ t'$, and $\Gamma \vdash t \sim t' : A$.

Proof. By Lemma 10, we have a proof $p : \Gamma :: \diamond \vdash t : A$, so $\Gamma :: \diamond \vdash t \sim \mathsf{nf}\ p : A$ follows by Theorem 11. Let $t' \equiv \mathsf{nf}\ p$. Then $\mathsf{Nf}\ t'$ follows by Lemma 15, and $\Gamma \vdash t \sim t' : A$ follows from Theorem 7. $\square$

A direct corollary of normalization is decidability of conversion.

Corollary 3 (Decidability of conversion)**.** If $\Gamma \vdash t : A$ and $\Gamma \vdash s : A$, then $\Gamma \vdash t \sim s : A \vee \neg(\Gamma \vdash t \sim s : A)$.

Proof. By Lemma 10, we have $p : \Gamma :: \diamond \vdash t : A$ and $q : \Gamma :: \diamond \vdash s : A$, as well as $\Gamma :: \diamond \vdash t \sim \mathsf{nf}\ p : A$ and $\Gamma :: \diamond \vdash s \sim \mathsf{nf}\ q : A$ by Theorem 11. We proceed by case analysis on the syntactic equality of the two normal forms, namely $\mathsf{nf}\ p = \mathsf{nf}\ q$, which is decidable.

- Case $\mathsf{nf}\ p = \mathsf{nf}\ q$. Then $\Gamma :: \diamond \vdash t \sim \mathsf{nf}\ p : A$. We conclude $\Gamma :: \diamond \vdash t \sim s : A$ by symmetry and transitivity, and $\Gamma \vdash t \sim s : A$ by Theorem 7.

- Case $\neg(\mathsf{nf}\ p = \mathsf{nf}\ q)$. Then $\neg(\Gamma \vdash t \sim s : A)$. In fact, suppose $\Gamma \vdash t \sim s : A$ were true. Then $\Gamma :: \diamond \vdash t \sim s : A$, and by completeness of NbE we have $[\![t]\!]\ \mathsf{Id} \searrow d$, $[\![s]\!]\ \mathsf{Id} \searrow d$ for some $d : \mathsf{D}$. But $[\![t]\!]\ \mathsf{Id} \searrow \mathsf{nf}\ p$ and $[\![s]\!]\ \mathsf{Id} \searrow \mathsf{nf}\ q$ hold by hypothesis, so $\mathsf{nf}\ p = d = \mathsf{nf}\ q$ by functionality of interpretation, contradicting our hypothesis.

$\square$

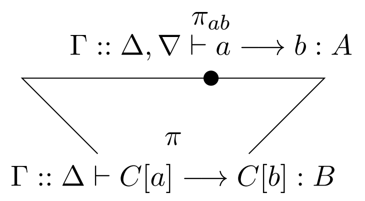
Chapter 6

Dependent types

In this chapter, we address weak reduction in the context of dependent types. We begin by observing that the "explicit" construction that we have employed for System T requires particular care in order to be adapted to a dependently-typed setting. Investigating these issues is left for future work. In the rest of the chapter, instead, we turn our attention to another weak notion of reduction, i.e. weak explicit substitutions. In particular, we describe the second main contribution of this thesis, namely the definition of a version of Martin-Löf Type Theory with large elimination and weak explicit substitutions, and a fully-formalized proof of untyped Normalization by Evaluation. We conclude by arguing that, despite weak explicit substitutions not being equivalent to CH-weak conversion (as shown in Chapter 3), they exhibit most of its desirable properties, while being much easier to formalize and reason about.

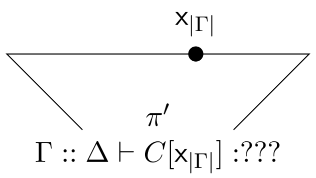
6.1 Implicit-explicit correspondence and dependent types

The proof of normalization for CH-weak reduction shown in the previous chapters relies on the ability to perform an extraction operation that pulls a weak redex out of a term, in a type-preserving way. In the context of simple types, like in System T, this extraction can be shown without too many complications, since types do not depend on terms and therefore are not affected by the extraction. In a dependent type theory, the type T of a term t may depend on the internal syntactic structure of t itself, so extracting a subterm a from t does not necessarily yield a term of the same type. Suppose we had the following derivation


π_{ab}
$\Gamma :: \Delta, \nabla \vdash a \longrightarrow b : A$
π
$\Gamma :: \Delta \vdash C[a] \longrightarrow C[b] : B$

and we wanted to perform the factorization into $C[\mathsf{x}_{|\Gamma|}]$ and $a \longrightarrow b$, as we did in

Chapter 4. Then, we would expect to obtain something like the following



but now we are left to determine what the type of $C[\mathsf{x}_{|\Gamma|}]$ should be. The type B of $C[a]$ and $C[b]$ may depend on a and b, hence when extracting a from $C[a]$ (and similarly for $C[b]$), the type of the resulting term $C[x]$ must be altered accordingly. It is not yet clear how to do this in a systematic way.

6.2 MLTT^{wk}: Martin-Löf Type Theory with weak explicit substitutions

CH-weak normalization is challenging to formalize and prove normalizing even in a simply-typed setting, given some aspects like a relative notion of redex, and the reliance on the metalinguistic substitution operation for expressing any kind reduction step. With dependent types, types may depend on terms, and moreover typing and equality judgments are mutually defined. It is not too difficult to imagine how these characteristics of dependent type theories may be a sounce of additional complication when considering then in the context of CH-weak reduction. It thus seems reasonable to ask ourselves whether there could be alternative notions of reduction that exhibit the same properties that are appreciated in CH-weak equality (like confluence), but that are more amenable to (formalized) metatheoretical analysis. In conclusion of Chapter 3, we argued that a good candidate for this is represented by *weak explicit substitutions*. In the rest of this chapter we study this notion of reduction in more detail, considering it in the context of dependent types. We do so with the definition of MLTT^{wk}, a formulation of Martin-Löf Type Theory with weak explicit substitutions, and a fully-formalized proof of normalization [4]. The calculus appears to be original, or at least the author is not aware, at the time of writing, of other versions of MLTT with specifically *weak* explicit substitutions. Abel et al. [9, 8] use explicit substitutions extensively in their treatment of NbE, but always for the stronger $\beta\eta$ conversion. [25] use explicit substitutions in the simply-typed case. Altenkirch and Chapman [12] also employ weak explicit substitutions in the simply-typed case, and formalize big-step normalization for intrinsic syntax. Our definition of MLTT^{wk} can be seen as an adaptation of their calculus to full dependent types, albeit we use extrinsic syntax instead.[1]

Our goal here is to test weak explicit substitutions in the context of a λ-calculus with full dependent types. We thus consider a minimal theory, in order not be distracted

[1]A presentation of a system is *intrinsic* when terms are defined in an inherently well-typed form, i.e. terms are well-typed by construction. Extrinsic syntax, on the contrary, is given by first defining raw, untyped pre-terms, and then defining inference rules that assign types to them.

with details that are less relevant at the moment. We include dependent products and one universe. The universe is empty, but it still allows non-trivial computation at the type level via *large elimination*, so the theory is strong enough to be relevant for our analysis.

Within dependent type theories, terms and types have the same status, so it is convenient to define them as part of the same syntactic category. Unlike in System T^{wk}, we do not distinguish between global and local variables, thus we only include De Bruijn indices for variables, both free and bound. In addition, we have a constructor for dependent products Π, as well as the usual λ-abstraction and application as its introduction and elimination forms. $\mathsf{Lam}\ t$ is intended to bind the first index in t, whereas $\Pi\ A\ B$ binds the first index in B. We also employ a universe a la Russell, and use the same syntax for both proper types and their codes inhabiting the universe.

We main difference from the previous chapters and from more traditional definitions of MLTT is the use of explicit substitutions, given by the type $\mathsf{Subst} : \mathsf{Set}$. Since these are part of the term language, we define them mutually with terms $\mathsf{Term} : \mathsf{Set}$.

$$\begin{aligned} t, u, A, B &::= \mathsf{Var} \mid \mathsf{Lam}\ t \mid t \cdot u \mid \mathsf{U} \mid \Pi\ A\ B \mid t[\sigma] \\ \sigma, \tau &::= \mathsf{Id} \mid \sigma, t \mid\ \uparrow\ \mid \sigma \cdot \tau \end{aligned}$$

Explicit substitutions are defined like in [8], which in turn follows [7]. The constructors of substitutions play a similar role to those of System T^{wk}. In particular, we still have Id for the identity substitution, and σ, t for the extension of σ with an additional term t. The major difference is that now we have composition of substitutions as a constructor, rather than as a metatheoretical operation. As a consequence, we only have the constructor $\uparrow$, which shifts all indices in a term by 1, for weakening. In fact, $\uparrow$ and composition are enough to derive all the other constructors of System T^{wk}. We also follow [12, 51] and encode De Bruijn indices out of a base constructor Var standing for the 0-th index, and liftings $\uparrow$, so that $\mathsf{Var} \cdot \uparrow^n$ stands for the n-th index, where $\uparrow^n \equiv \uparrow \cdot\ \ldots\ \cdot \uparrow n$ times.

Since substitutions are part of the syntax, their effect on terms is not given by a metalinguistic operation, but it is provided *definitionally* in the judgment of the theory. In particular, the substitution rules for types and terms hold definitionally, rather than as a lemma. Figure 6.1 shows the inductive definition of judgments regarding contexts, types and terms, together with their equality. In particular, type and term equality judgments show how explicit substitutions propagate throught term constructors. The *weak* charater of the theory is achieved by two things: first, we do not include congruence rules for terms with binders, i.e. λ-abstractions and Π types. Second, the substitution-propagation rules are defined so that substitutions are never pushed under binders, that is, we do not allow further propagation of σ in $(\lambda t)[\sigma]$ and $(\Pi A B)[\sigma]$. Since substitution is "blocked" for these terms, we always consider them together with an additional substitution whenever we want to substitute some arguments in place of their bound variable. An example of this is given by the (β) rule, which considers functional terms $(\lambda t)[\sigma]$ for some σ, and implements β-contraction by just extending σ with the argument.

Figure 6.2 shows substitution and substitution equality judgments, which tell us when a substitution is well-typed, and when two well-typed substitutions are equal. These last rules can be seen as an axiomatixation the substitution equality considered for System T^{wk}, and implement basic equations like congruence, composition, and identity elimination. This definition of explicit substitutions is fairly standard, and closely follows other implementations with dependent types and nameless syntax, like [8, 10]. We refer the reader to these references for a more detailed treatment of explicit substitutions in a similar setting.

6.3 Normalization by Evaluation for MLTT^{wk}

In this section, we establish normalization for MLTT^{wk} by instantiating untyped normalization by evaluation. The proof is essentially an adaptation of [8] to weak explicit substitutions, and of the previous Chapter to dependent types. Therefore, in the sections that follow, we mainly focus the details that are unique to MLTT^{wk}, referring the interested reader to the references and the Agda formalization [4] for the complete picture.

6.3.1 Normal forms

We now give an inductive definition of normal forms of MLTT^{wk}. As before, we define a predicate $\mathsf{Nf} : \mathsf{Term} \to \mathsf{Set}$ of normal terms mutually with a predicate $\mathsf{Ne} : \mathsf{Term} \to \mathsf{Set}$ of neutral terms. Terms with an applied explicit substitution can always be reduced by propagating the substitution on the subterms, with the exception of λ and Π. Hence, for example, $(\lambda t)[\sigma]$ is a normal form, whenever σ is in normal form. Consequently, we need to define what it means for a substitution to be in normal form. We do so with a predicate $\mathsf{Nfs} : \mathsf{Subst} \to \mathsf{Set}$.

$$\frac{\mathsf{Ne}\ e}{\mathsf{Nf}\ e} \qquad \frac{\mathsf{Nfs}\ \sigma}{\mathsf{Nf}\ ((\lambda t)[\sigma])} \qquad \frac{\mathsf{Nfs}\ \sigma}{\mathsf{Nf}\ ((\Pi\ A\ B)[\sigma])} \qquad \frac{}{\mathsf{Nf}\ \mathsf{U}}$$

$$\frac{}{\mathsf{Ne}\ (\mathsf{Var}[\uparrow^n])} \qquad \frac{\mathsf{Ne}\ e \qquad \mathsf{Nf}\ d}{\mathsf{Ne}\ (e \cdot d)} \qquad \frac{}{\mathsf{Nfs}\ (\mathsf{Id}\cdot \uparrow^w)} \qquad \frac{\mathsf{Nfs}\ \sigma \qquad \mathsf{Nf}\ a}{\mathsf{Nfs}\ (\sigma, a)}$$

6.3.2 Semantic domain

We now define the type of semantic values, that will represent the result of *evaluation*. The semantics must include *closures*, that is, syntactic terms together with environments assigning a semantic value to every free variable, since these give rise to normal forms. We define values $d : \mathsf{D}$, neutral values $e : \mathsf{Dne}$, and environments of values $\rho : \mathsf{Env}$ by the following mutually-inductive definition.

$$\begin{aligned} d &::= \mathsf{DU} \mid \lambda\mathsf{Clo}\ t\ \rho \mid \Pi\mathsf{Clo}\ A\ B\ \rho \mid \mathsf{DNe}\ e \\ e &::= \mathsf{Lev}\ n \mid \mathsf{NeApp}\ e\ d \\ \rho &::= \epsilon \mid \rho, d \end{aligned}$$

$\boxed{\vdash\_ : \mathsf{Ctxt} \to \mathsf{Set}}$

$$\frac{}{\vdash \diamond} \qquad \frac{\vdash \Gamma \qquad \Gamma \vdash A}{\vdash \Gamma, A}$$

$\boxed{\_\vdash\_ : \mathsf{Ctxt} \to \mathsf{Term} \to \mathsf{Set}}$

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathsf{U}} \qquad \frac{\Gamma \vdash A : \mathsf{U}}{\Gamma \vdash A} \qquad \frac{\Gamma \vdash A \qquad \Gamma, A \vdash B}{\Gamma \vdash \Pi\ A\ B} \qquad \frac{\Gamma \vdash A \qquad \Delta \vdash_s \sigma : \Gamma}{\Delta \vdash A[\sigma]}$$

$\boxed{\_\vdash\_:\_ : \mathsf{Ctxt} \to \mathsf{Term} \to \mathsf{Term} \to \mathsf{Set}}$

$$\frac{\Gamma \vdash A}{\Gamma, A \vdash \mathsf{Var} : A[\uparrow]} \qquad \frac{\Gamma, A \vdash t : B}{\Gamma \vdash \lambda t : \Pi\ A\ B} \qquad \frac{\Gamma \vdash r : (\Pi\ A\ B)[\sigma] \qquad \Gamma \vdash s : A[\sigma] \qquad \Gamma \vdash_s \sigma : \Delta}{\Gamma \vdash r \cdot s : B[\sigma, s]}$$

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash A \sim B}{\Gamma \vdash t : B} \qquad \frac{\Gamma \vdash t : A \qquad \Delta \vdash_s \sigma : \Gamma}{\Delta \vdash t[\sigma] : A[\sigma]}$$

$\boxed{\_\vdash\_\sim\_ : \mathsf{Ctxt} \to \mathsf{Term} \to \mathsf{Term} \to \mathsf{Set}}$

$$\frac{\Gamma \vdash A \sim B : \mathsf{U}}{\Gamma \vdash A \sim B} \qquad \frac{\Delta \vdash_s \sigma : \Gamma}{\Delta \vdash \mathsf{U}[\sigma] \sim \mathsf{U}} \qquad \frac{\Gamma \vdash A \qquad \Delta \vdash_s \sigma : \Gamma \qquad \nabla \vdash_s \tau : \Delta}{\Gamma \vdash A[\sigma][\tau] \sim A[\sigma \cdot \tau]}$$

$$\frac{\Gamma \vdash A \sim B \qquad \Delta \vdash_s \sigma \sim \tau : \Gamma}{\Delta \vdash A[\sigma] \sim B[\tau]} \qquad \frac{\Gamma \vdash A}{\Gamma \vdash A[\mathsf{Id}] \sim A}$$

plus equivalence rules

$\boxed{\_\vdash\_\sim\_:\_ : \mathsf{Ctxt} \to \mathsf{Term} \to \mathsf{Term} \to \mathsf{Term} \to \mathsf{Set}}$

$$\frac{\Gamma \vdash t : B \qquad \Delta \vdash s : A[\sigma] \qquad \Delta \vdash_s \sigma : \Gamma}{\Delta \vdash (\lambda t)[\sigma] \cdot s \sim t[\sigma, s] : B[\sigma, s]}\ (\beta) \qquad \frac{\begin{array}{c}\Gamma \vdash s \sim s' : A[\sigma] \\ \Gamma \vdash t \sim t' : (\Pi\ A\ B)[\sigma]\end{array}}{\Gamma :: \Delta \vdash t \cdot s \sim t' \cdot s' : B[\sigma, s]}$$

$$\frac{\Gamma \vdash t \sim s : A \qquad \Gamma \vdash A \sim B}{\Gamma \vdash t \sim s : B} \qquad \frac{\Gamma \vdash t \sim s : A \qquad \Delta \vdash_s \sigma \sim \tau : \Gamma}{\Delta \vdash t[\sigma] \sim s[\sigma] : A[\sigma]}$$

$$\frac{\begin{array}{cc}\Delta \vdash s : A[\sigma] & \nabla \vdash_s \tau : \Delta \\ \Delta \vdash r : (\Pi\ A\ B)[\sigma] & \Delta \vdash_s \sigma : \Gamma\end{array}}{\nabla \vdash (r \cdot s)[\tau] \sim r[\tau] \cdot s[\tau] : B[\sigma \cdot \tau, s[\tau]]} \qquad \frac{\begin{array}{cc} & \nabla \vdash_s \tau : \Delta \\ \Gamma \vdash t : A & \Delta \vdash_s \sigma : \Gamma\end{array}}{\nabla \vdash t[\sigma][\tau] \sim t[\sigma \cdot \tau] : A[\sigma \cdot \tau]}$$

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash t[\mathsf{Id}] \sim t : A} \qquad \frac{\Gamma \vdash t : A[\sigma] \qquad \Gamma \vdash_s \sigma : \Delta}{\Gamma \vdash \mathsf{Var}[\sigma, t] \sim t : A[\sigma]}$$

plus equivalence rules

Figure 6.1: MLTT^{wk}: term judgments

$$\boxed{\_ \vdash_s \_ : \_ : \mathsf{Ctxt} \to \mathsf{Subst} \to \mathsf{Ctxt} \to \mathsf{Set}}$$

$$\frac{}{\Gamma \vdash_s \mathsf{Id} : \Gamma} \qquad \frac{\Gamma \vdash_s \sigma : \Delta \qquad \Gamma \vdash t : A[\sigma]}{\Gamma \vdash_s \sigma, t : \Delta, A} \qquad \frac{\Gamma \vdash_s \sigma : \Delta \qquad \nabla \vdash_s \tau : \Gamma}{\Delta \vdash_s \sigma \cdot \tau : \Delta} \qquad \frac{\Gamma \vdash A}{\Gamma, A \vdash_s \uparrow : \Gamma}$$

$$\boxed{\_ \vdash_s \_ \sim \_ : \_ : \mathsf{Ctxt} \to \mathsf{Subst} \to \mathsf{Subst} \to \mathsf{Ctxt} \to \mathsf{Set}}$$

$$\frac{\Delta \vdash_s \sigma : \Gamma}{\Delta \vdash_s \sigma \cdot \mathsf{Id} \sim \Gamma :} \qquad \frac{\Delta \vdash_s \sigma : \Gamma}{\Delta \vdash_s \mathsf{Id} \cdot \sigma \sim \Gamma :} \qquad \frac{\Delta \vdash_s \sigma \sim \tau : \Gamma \qquad \Delta \vdash t \sim s : A[\sigma]}{\Delta \vdash_s \sigma, t \sim \tau, s : \Gamma, A}$$

$$\frac{\Delta \vdash_s \sigma : \Gamma \qquad \Delta \vdash t : A[\sigma]}{\Delta \vdash_s \uparrow \cdot (\sigma, t) \sim \sigma : \Gamma} \qquad \frac{\Gamma_4 \vdash_s \sigma_3 : \Gamma_3 \qquad \Gamma_3 \vdash_s \sigma_2 : \Gamma_2 \qquad \Gamma_2 \vdash_s \sigma_1 : \Gamma_1}{\Gamma_4 \vdash_s (\sigma_1 \cdot \sigma_2) \cdot \sigma_3 \sim \sigma_1 \cdot (\sigma_2 \cdot \sigma_3) : \Gamma_1}$$

$$\frac{\begin{array}{l}\Gamma \vdash_s \tau : \Gamma' \\ \Gamma' \vdash_s \sigma : \Delta \qquad \Gamma' \vdash_s t : A[\sigma]\end{array}}{\Gamma \vdash_s (\sigma, t) \cdot \tau \sim (\sigma \cdot \tau), t[\tau] : \Delta, A} \qquad \frac{\Gamma \vdash_s \sigma \sim \sigma' : \Delta \qquad \Gamma \vdash t \sim t' : A[\sigma]}{\Gamma \vdash_s \sigma, t \sim \sigma', t' \sim \Delta, A :}$$

$$\frac{\Gamma_1 \vdash_s \sigma \sim \sigma' : \Gamma_2 \qquad \Gamma_3 \vdash_s \tau \sim \tau' : \Gamma_1}{\Gamma_3 \vdash_s \sigma \cdot \tau \sim \sigma' \cdot \tau' : \Gamma_2} \qquad \frac{\Gamma \vdash A}{\Gamma, A \vdash_s \mathsf{Id} \cdot \uparrow, \mathsf{v}_0 \sim \mathsf{Id} : \Gamma, A}$$

plus equivalence rules

Figure 6.2: MLTT^{wk}: substitution judgments

We do not need to test semantic values for "closeness" anymore. Thus, we follow [8] and use De Bruijn levels for variables in the semantics, avoiding the need to implement liftings and weakenings for semantic values, because levels do not change their value under context extensions.[2] As before, we sometimes write x_n as a shorthand for $\mathsf{Lev}\ n$.

We mutually define the relations

$$\begin{aligned}
[\![\_]\!]\ \_ \searrow \_ &: \mathsf{Term} \to \mathsf{Env} \to \mathsf{D} \to \mathsf{Set}\\
[\![\_]\!]_s\ \_ \searrow \_ &: \mathsf{Subst} \to \mathsf{Env} \to \mathsf{D} \to \mathsf{Set}\\
\_ \bullet \_ \searrow \_ &: \mathsf{D} \to \mathsf{D} \to \mathsf{D} \to \mathsf{Set}
\end{aligned}$$

Representing, respectively, the interpretation of terms and substitutions, and partial application between semantic values.

$$\frac{[\![t]\!]\ (\rho, a) \searrow b}{\lambda\mathsf{Clo}\ t\ \rho \bullet a \searrow b}\ (\bullet\beta) \qquad \frac{e : \mathsf{Dne} \qquad d : \mathsf{D}}{e \bullet d \searrow \mathsf{NeApp}\ e\ d}\ (\bullet\mathsf{Ne})$$

$$\frac{}{[\![\mathsf{Id}]\!]_s\ \rho \searrow \rho}\ (\mathsf{sId}) \qquad \frac{\begin{array}{c}[\![t]\!]\ \rho \searrow a\\ [\![\sigma]\!]_s\ \rho \searrow \rho'\end{array}}{[\![\sigma, t]\!]_s\ \rho \searrow (\rho', a)}\ (\mathsf{sCons}) \qquad \frac{\begin{array}{c}[\![\sigma]\!]_s\ \rho' \searrow \rho''\\ [\![\tau]\!]_s\ \rho \searrow \rho'\end{array}}{[\![\sigma \cdot \tau]\!]_s\ \rho \searrow \rho''}\ (\mathsf{sComp})$$

$$\frac{}{[\![\uparrow]\!]_s\ (\rho, a) \searrow \rho}\ (\mathsf{sUp})$$

$$\frac{}{[\![\mathsf{Var}]\!]\ (\rho, a) \searrow a}\ (\mathsf{eVar}) \qquad \frac{}{[\![\lambda t]\!]\ \rho \searrow \lambda\mathsf{Clo}\ t\ \rho}\ (\mathsf{eLam})$$

$$\frac{\begin{array}{l}[\![t]\!]\ \rho \searrow a\\ [\![s]\!]\ \rho \searrow b \qquad a \bullet b \searrow c\end{array}}{[\![t \cdot s]\!]\ \rho \searrow c}\ (\mathsf{eApp}) \qquad \frac{}{[\![\Pi\ A\ B]\!]\ \rho \searrow \Pi\mathsf{Clo}\ A\ B\ \rho}\ (\mathsf{ePi}) \qquad \frac{}{[\![\mathsf{U}]\!]\ \rho \searrow \mathsf{DU}}\ (\mathsf{eU})$$

$$\frac{[\![\sigma]\!]_s\ \rho \searrow \rho' \qquad [\![t]\!]\ \rho' \searrow a}{[\![t[\sigma]]\!]\ \rho \searrow a}\ (\mathsf{eSub})$$

We define the following *reification* functions by mutual recursion on values, neutral values, and environments. Reification returns the syntactic term corresponding to a given semantic object. Since we are using De Bruijn levels for values, whose meaning is context-dependent, we need to index reification by some natural number indicating the number of assumptions that we are under, so that we can correctly translate each level to the corresponding index.

$\mathsf{wks} : \mathbb{N} \to \mathsf{Subst}$
$\mathsf{wks}\ \mathsf{zero} = \mathsf{Id}$
$\mathsf{wks}\ (\mathsf{suc}\ n) = \mathsf{wks}\ n \cdot \uparrow$

[2]Recall that we already exploited this property of De Bruijn levels in System T^{wk}. In fact, we used levels to represent free variables so that weakening could be implemented straightforwardly, without syntactic alterations to terms.

```
mutual
  reify : ℕ → D → Term
  reify n (DNe x) = reifyNe n x
  reify n (λClo t ρ) = Lam t [ reifyEnv n ρ ]
  reify n (ΠClo A B ρ) = (Π A B) [ reifyEnv n ρ ]
  reify n DU = U

  reifyNe : ℕ → Dne → Term
  reifyNe n (Lev x) = Var [ wks (n - suc x) ]
  reifyNe n (NeApp e x) = reifyNe n e · reify n x

  reifyEnv : ℕ → Env → Subst
  reifyEnv n ε = wks n
  reifyEnv n (e , x) = reifyEnv n e , reify n x
```

The goal of reification is to extract normal forms from semantic values. To this end, we now show that semantic values indeed reify to terms in normal form.

Lemma 34. For any $d : \mathsf{D}, e : \mathsf{Dne}, \rho : \mathsf{Env}$, and $n : \mathbb{N}$, the following hold

1. $\mathsf{Nf}\ (\mathsf{reify}\ n\ d)$;
2. $\mathsf{Ne}\ (\mathsf{reifyNe}\ n\ e)$;
3. $\mathsf{Nfs}\ (\mathsf{reifyEnv}\ n\ \rho)$.

Proof. By mutual induction on d, e, ρ. See `Semantics.Domain.D-is-Nf`, `Semantics.Domain.Dne-is-Ne`, and `Semantics.Domain.Env-is-Nfs`. □

6.3.3 Completeness of NbE

We now revisit the subset model used for System T^{ex}, and adapt it to dependent types. We again consider type-theoretic subsets $\mathcal{A} : \mathsf{D} \to \mathsf{Set}$, and we say that $\mathcal{A}$ is a semantic type whenever $\mathcal{A}\ e$ holds for any $e : \mathsf{Dne}$. We do not need to ensure that elements of a semantic type reify to normal forms (like in [8]), because this is true for any semantic value (Lemma 34). We define the semantic dependent product space in a similar way to the function space for System T^{ex}. Thus, given $\mathcal{A} : \mathcal{P}(\mathsf{D})$ and and $\mathcal{B} : \forall\{a\} \to a \in_t \mathcal{A} \to \mathcal{P}(\mathsf{D})$, we put

$$\Pi\ \mathcal{A}\ \mathcal{B} :\equiv \lambda f.(\forall\{a\} \to (p : a \in_t \mathcal{A}) \to f \bullet a \in_t (\mathcal{B}\ a))$$

where $\_ \bullet \_ \in_t \_$ is defined as in $SystemT^{ex}$. Notice that the definition of semantic Π is actually simpler than T^{ex}'s: since we are using De Bruijn levels in the semantics, we do not have to accomodate for arbitrary weakenings.

We now define the valid semantic interpretation of small and large syntactic types. To this end, we inductively define *semantic universes* $\mathscr{U}$ and $\mathscr{T}$ that are semantic types themselves, and classify values standing for valid *codes* for types. Simultaneously, we define the extension functions $\mathsf{El}_{\mathscr{U}} : \forall\{A\} \to A \in_t \mathscr{U} \to \mathcal{P}(\mathsf{D})$ and $\mathsf{El}_{\mathscr{T}} : \forall\{A\} \to A \in_t \mathscr{T} \to \mathcal{P}(\mathsf{D})$ mapping valid type codes to semantic types.[3]

In what follows, we use the abbreviation below for interpretation into a semantic type $\mathcal{A} : \mathsf{D} \to \mathsf{Set}$:

$$[\![t]\!]\rho \in_t \mathcal{A} :\equiv \Sigma(a : \mathsf{D})([\![t]\!]\ \rho \searrow a \wedge a \in_t \mathcal{A})$$

as well as the following shorthand, defined from projections on the Σ type in the obvious way:

$$\mathsf{inSemTy} : (p : [\![t]\!]\rho \in_t \mathcal{A}) \to \pi_1\ p \in_t \mathcal{A}$$

Then, we define semantic universes as the following inductive families $\mathscr{U}, \mathscr{T} : \mathsf{D} \to \mathsf{Set}$. Notice that we explicitly write down the proof terms of the inference rules below, since they are pattern-matched on by the extension functions.

$$\frac{e : \mathsf{Ne}\ A}{\mathsf{uNe}\ e : \mathscr{U}\ A}$$

$$\begin{aligned}
&\mathsf{El}_{\mathscr{U}} : \forall\{A\} \to A \in_t \mathscr{U} \to \mathcal{P}(\mathsf{D})\\
&\mathsf{El}_{\mathscr{U}}(\mathsf{uNe}\ \_) = \mathsf{isDne}
\end{aligned}$$

where $\mathsf{isDne} : \mathsf{D} \to \mathsf{Set}$ is such that $\mathsf{isDne}\ (\mathsf{DNe}\ \_) = \top$, and $\bot$ otherwise.

$$\frac{\rho : \mathsf{Env} \qquad p_A : [\![A]\!]\rho \in_t \mathscr{T} \qquad p_B : \forall\{a\} \to a \in_t \mathsf{El}_{\mathscr{T}}\ p_A \to [\![B]\!](\rho, a) \in_t \mathscr{T}}{\mathsf{tPi}\ p_A\ p_B : \mathscr{T}\ (\Pi\mathsf{Clo}\ A\ B\ \rho)} \qquad \frac{\mathsf{p} : T \in_t \mathscr{U}}{\mathsf{inj}_{\mathscr{U}}\ \mathsf{p} : \mathscr{T}\ T} \qquad \frac{}{\mathsf{tU} : \mathscr{T}\ \mathsf{DU}}$$

$$\begin{aligned}
&\mathsf{El}_{\mathscr{T}} : \forall\{A\} \to A \in_t \mathscr{T} \to \mathcal{P}(\mathsf{D})\\
&\mathsf{El}_{\mathscr{T}}\ (\mathsf{inj}_{\mathscr{U}}\ x) = \mathsf{El}_{\mathscr{U}}\ x\\
&\mathsf{El}_{\mathscr{T}}\ (\mathsf{tPi}\ p_A\ p_B) = \Pi\ (\mathsf{El}_{\mathscr{T}}(\mathsf{inSemTy}\ p_A))\ (\mathsf{El}_{\mathscr{T}} \circ \mathsf{inSemTy} \circ p_B)\\
&\mathsf{El}_{\mathscr{T}}\ \mathsf{tU} = \mathscr{U}
\end{aligned}$$

We define the interpretation of type values into semantic types, $[\![\_]\!]_{t\_} : \mathsf{Term} \to \mathsf{Env} \to \mathcal{P}(\mathsf{D})$, as follows

$$[\![A]\!]_t\ \rho\ :\equiv \lambda d.\Sigma(T : \mathsf{D})([\![A]\!]\ \rho \searrow T \wedge \Sigma(p : T \in_t \mathscr{T})(d \in_t \mathsf{El}_{\mathscr{T}}\ p))$$

together with convenient shorthands, defined from projections on the Σ type in the obvious way:

$$\mathsf{inTy} : (p : d \in [\![A]\!]_t\ \rho) \to \pi_1\ p \in_t \mathscr{T} \qquad \mathsf{inTm} : (p : d \in_t [\![A]\!]_t\ \rho) \to d \in_t \mathsf{El}_{\mathscr{T}}\ (\mathsf{inTy}\ p)$$

The relations on equal terms of a semantic type are defined like in the simply-typed case, as follows

[3]This mutual definition is an example of induction-recursion. See Section 2.5.

$$[\![t]\!] \simeq [\![s]\!]\ \rho \in_t \mathcal{A} :\equiv \Sigma(r : \mathsf{Term})(r \in_t \mathcal{A} \wedge [\![t]\!]\ \rho \searrow r \wedge [\![s]\!]\ \rho \searrow r)$$
$$[\![t]\!] \simeq [\![s]\!]\ \rho \in_t [\![A]\!]_t :\equiv [\![t]\!] \simeq [\![s]\!]\ \rho \in_t [\![A]\!]_t \rho$$

We also define the following convenience functions, by projection on the Σ type in the obvious way

$$\begin{aligned}
&\mathsf{inEqTy} : (p : [\![t]\!] \simeq [\![s]\!]\ \rho \in_t [\![A]\!]_t) \to \pi_1\ ((\pi_1 \circ \pi_2)\ p) \in_t \mathscr{T} \\
&\mathsf{inEqTm} : (p : [\![t]\!] \simeq [\![s]\!]\ \rho \in_t [\![A]\!]_t) \to \pi_1\ p \in_t \mathsf{El}_{\mathscr{T}}\ (\mathsf{inEqTy}\ p)
\end{aligned}$$

However, now $[\![\_]\!]_{s\_} : \mathsf{Ctxt} \to \mathsf{Env} \to \mathsf{Set}$ identifies subsets of environments, rather than substitutions:

$$\frac{}{[\![\diamond]\!]_s\ \epsilon}\ (\mathsf{cEmpty}) \qquad \frac{\rho \in_s [\![\Gamma]\!]_s \qquad a \in_t [\![A]\!]_t\ \rho}{[\![\Gamma, A]\!]_s\ (\rho, a)}\ (\mathsf{cExt})$$

Substitutions are evaluated to environments of semantic values. Since now we have definitional equality judgments between substitutions, we need to define what is means for two substitutions to be semantically equal elements of a subset $\mathcal{S} : \mathsf{Env} \to \mathsf{Set}$ of environments:

$$[\![\sigma]\!] \simeq [\![\tau]\!]\ \rho \in_s \mathcal{S} :\equiv \Sigma(\rho : \mathsf{Env})(\rho \in_s \mathcal{S} \wedge [\![t]\!]\ \rho \searrow r \wedge [\![s]\!]\ \rho \searrow r)$$

In order to prove completeness of NbE, we shall define semantic counterparts of type and equality judgments:

$$\begin{aligned}
&\models \diamond :\equiv \top \qquad \models (\Gamma, A) :\equiv \models \Gamma \wedge (\Gamma \models A) \\
&\Gamma \models A :\equiv \forall\{\rho\} \to \rho \in_s [\![\Gamma]\!]_s \to [\![A]\!] \simeq [\![A]\!]\ \rho \in_t \mathscr{T} \\
&\Gamma \models A \sim B :\equiv \forall\{\rho\} \to \rho \in_s [\![\Gamma]\!]_s \to [\![A]\!] \simeq [\![B]\!]\ \rho \in_t \mathscr{T} \\
&\Gamma \models t : A :\equiv \Gamma \models A \wedge (\forall\{\rho\} \to \rho \in_s [\![\Gamma]\!]_s \to [\![t]\!] \simeq [\![t]\!]\ \rho \in_t [\![A]\!]_t) \\
&\Gamma \models t \sim s : A :\equiv \Gamma \models A \wedge (\forall\{\rho\} \to \rho \in_s [\![\Gamma]\!]_s \to [\![t]\!] \simeq [\![s]\!]\ \rho \in_t [\![A]\!]_t) \\
&\Delta \models_s \sigma : \Gamma :\equiv \forall\{\rho\} \to \rho \in_s [\![\Delta]\!]_s \to [\![\sigma]\!] \simeq [\![\sigma]\!]\ \rho \in_s [\![\Gamma]\!]_s \\
&\Delta \models_s \sigma \sim \tau : \Gamma :\equiv \forall\{\rho\} \to \rho \in_s [\![\Delta]\!]_s \to [\![\sigma]\!] \simeq [\![\tau]\!]\ \rho \in_s [\![\Gamma]\!]_s
\end{aligned}$$

We can now prove the following completeness results:

Theorem 14.

1. If $\vdash \Gamma$, then $\models \Gamma$;
2. If $\Gamma \vdash A$, then $\Gamma \models A$;
3. If $\Gamma \vdash t : A$, then $\Gamma \models t : A$;
4. If $\Gamma \vdash A \sim B$, then $\Gamma \models A \sim B$;

5. If $\Gamma \vdash t \sim s : A$, then $\Gamma \models t \sim s : A$.

Proof. All points proved by mutual induction on derivations.
See the module `Semantics.Completeness.Rule`. □

Since we are working in a constructive type theory as our metatheory, all implications in Theorem 14 give rise to total, computable functions. By abuse of notation, we indicate these by the overloaded symbol $(\!|\_|\!) : \Gamma \vdash \mathcal{J} \to \Gamma \models \mathcal{J}$, where $\mathcal{J}$ is one among the available typing and substitution judgments.

To show completeness of NbE, we first define identity environments:

$$\begin{aligned}&\mathsf{idenv} : \mathsf{Ctxt} \to \mathsf{Env}\\ &\mathsf{idenv}\ \diamond = \epsilon\\ &\mathsf{idenv}\ (\Gamma, A) = (\mathsf{idenv}\ \Gamma \cdot \uparrow)\ ,\ \mathsf{DNe}\ \mathsf{x}_{|\Gamma|}\end{aligned}$$

Lemma 35. If $\vdash \Gamma$, then $\mathsf{idenv}\ \Gamma \in_s [\![\Gamma]\!]_s$.

Proof. By induction on the proof of $\vdash \Gamma$, and application of Theorem 14. □

We call $\mathsf{idenvp} : \vdash \Gamma \to \mathsf{idenv}\ \Gamma \in_s [\![\Gamma]\!]_s$ the proof term for Lemma 6.3.3. We now define the normalization functions $\mathsf{nf\text{-}ty} : \vdash \Gamma \to \Gamma \vdash A \to \mathsf{Term}$ and $\mathsf{nf\text{-}tm} : \vdash \Gamma \to \Gamma \vdash t : A \to \mathsf{Term}$, by interpreting well-formed types and well-typed terms in the model under the identity environment, extracting the semantic value, and reifying it to a term under the same context Γ.[4]

$$\begin{aligned}&\mathsf{nf\text{-}ty}\ \{\Gamma\}\ p\ q :\equiv \mathsf{reify}\ |\Gamma|\ (\pi_1\ ((\!|q|\!)\ (\mathsf{idenvp}\ c)))\\ &\mathsf{nf\text{-}tm}\ \{\Gamma\}\ p\ q :\equiv \mathsf{reify}\ |\Gamma|\ ((\pi_1 \circ \pi_2)\ ((\!|q|\!)\ (\mathsf{idenvp}\ c)))\end{aligned}$$

Corollary 4 (Completeness of NbE)**.** Let $c : \vdash \Gamma$, $p_A :\ \Gamma \vdash A$, $p_B :\ \Gamma \vdash B$, $q_t :\ \Gamma \vdash t : A$, $q_s :\ \Gamma \vdash s : A$ be well-formed types and well-typed terms. Then, the following hold

1. If $\Gamma \vdash A \sim B$, then $\mathsf{nf\text{-}ty}\ c\ p_A = \mathsf{nf\text{-}ty}\ c\ p_B$;

2. If $\Gamma \vdash t \sim s : A$, then $\mathsf{nf\text{-}tm}\ c\ q_A = \mathsf{nf\text{-}tm}\ c\ q_B$.

Proof. By Theorem 14 and functionality of the evaluation relation. □

In showing completeness, we could also derive the proof terms for $\vdash \Gamma$ etc. using inversion lemmas from equality judgments. Indeed, this is exactly what we do in the formalization. However, for this presentation we choose to be particularly explicit, to achieve better clarity.

[4] In the definition of the normalization functions, $\{\Gamma\}$ means that the context is treated as an implicit argument, whose value is inferred automatically by Agda via unification. See Section 2.5 for details.

6.3.4 Kripke logical relations and soundness of NbE

We now establish soundness of NbE with a Kripke logical relation between well-typed terms and their interpretation in the semantics, just like we did for System T^{ex} in Chapter 5. The formalization of the contents of this section can be found under the folder `Semantics/Soundness/` in [4]. Logical relations are defined by induction on types; however, we are in a dependent type theory where terms and types are in the same syntactic category, so there is no *a priori* notion of "type" on which to do induction on. Instead, we can rely on the notion of type values $A \in_t \mathscr{U}$ and $A \in_t \mathscr{T}$, that is, values corresponding to well-formed type codes in the semantics. We thus define two classes of logical relations, for small and large types respectively, by induction on (the proof of membership of) type values in their respective type universe.

$$\begin{aligned}
\_ \vdash \_ ®_{\mathscr{U}} \_ &: \mathsf{Ctxt} \to \mathsf{Term} \to \{A : \mathsf{D}\} \to A \in_t \mathscr{U} \to \mathsf{Set} \\
\_ \vdash \_ ®_{\mathscr{T}} \_ &: \mathsf{Ctxt} \to \mathsf{Term} \to \{A : \mathsf{D}\} \to A \in_t \mathscr{T} \to \mathsf{Set} \\
\_ \vdash \_ : \_ ®_{\mathscr{U}} \_ \ni \_ &: \mathsf{Ctxt} \to \mathsf{Term} \to \mathsf{Term} \\
&\to \forall \{a\ A\} \to (p : A \in_t \mathscr{U}) \to a \in_t \mathsf{El}_{\mathscr{U}}\ p \to \mathsf{Set} \\
\_ \vdash \_ : \_ ®_{\mathscr{T}} \_ \ni \_ &: \mathsf{Ctxt} \to \mathsf{Term} \to \mathsf{Term} \\
&\to \forall \{a\ A\} \to (p : A \in_t \mathscr{T}) \to a \in_t \mathsf{El}_{\mathscr{T}}\ p \to \mathsf{Set}
\end{aligned}$$

Therefore, $\Gamma \vdash T ®_{\mathscr{U}} p$ for $p : A \in_t \mathscr{U}$ means that the syntactic type T is logically-related to a type value A in the semantic universe $\mathscr{U}$. Similarly for $\mathscr{T}$. Whereas, $\Gamma \vdash t : T ®_{\mathscr{U}} p \ni q$ with $p : A \in_t \mathscr{U}$ and $a \in_t \mathsf{El}_{\mathscr{U}}\ p$ means that the syntactic term t of type T is logically-related to the semantic value a belonging to the semantic type associated to the type value A, belonging to the semantic universe $\mathscr{U}$. Similarly for $\mathscr{T}$.

When defining the logical relation for dependent products, we have to enforce the same condition on related values as we did in Section 5.3 for function types, namely that we are relating λ-abstractions in the syntax with λ-abstractions in the semantics. We also need to impose a similar condition on type values corresponding to Π types, i.e. we have to enforce that types related to semantic Π types are convertible to Π in the syntax.

As with System T^{wk}, the logical relations will be Kripke logical relations, with contexts as the set of possible worlds, and weakening as the accessibility relation. To this aim, we define weakenings $\_ \vdash_r \_ : \_ : \mathsf{Ctxt} \to \mathsf{Subst} \to \mathsf{Ctxt} \to \mathsf{Set}$ as special forms of substitutions, precisely those formed out of repeated $\uparrow$ shifts:

$$\frac{}{\Gamma \vdash_r \mathsf{Id} : \Gamma} \qquad \frac{\Delta \vdash_r w : \Gamma}{\Delta, A \vdash_r w \cdot \uparrow : \Gamma}$$

The actual Agda definition of the dependently-typed logical relations and all its cases is quite complex, and reproducing it here in inference rule form would produce an overly-complicated definition, that would risk obfuscating its underlying meaning. For this reason, we give the relations below in a more informal, list-like way:

Definition 4 (Logical relations)**.**

1. $\Gamma \vdash T \circledR_{\mathscr{U}}$ $\mathsf{uNe}\ e$ iff $\forall\{\Delta\ w\} \to \Delta \vdash_r w : \Gamma \to \Delta \vdash T[w] \sim \mathsf{reifyNe}\ |\Delta|\ e$;
2. $\Gamma \vdash T \circledR_{\mathscr{T}}$ $\mathsf{inj}_{\mathscr{U}}\ x$ iff $\Gamma \vdash T \circledR_{\mathscr{U}}\ x$;
3. $\Gamma \vdash T \circledR_{\mathscr{T}}$ tU iff $\Gamma \vdash T \sim \mathsf{U}$;
4. $\Gamma \vdash R \circledR_{\mathscr{T}}$ $(\mathsf{tPi}\ \{A\}\ \{B\}\ \{\rho\}\ p_A\ p_B)$ iff there are Δ, σ such that
   (a) $\Gamma \vdash R \sim (\Pi\ A\ B)[\sigma]$;
   (b) For all ∇, w such that $\nabla \vdash_r w : \Gamma$, then $\nabla \vdash_s \sigma \cdot w \sim \mathsf{reifyEnv}\ |\nabla|\ \rho : \Delta$;
   (c) For all ∇, w such that $\nabla \vdash_r w : \Gamma$, $p : a \in_t \mathsf{El}_{\mathscr{T}}(\mathsf{inSemTy}\ p_A))$, and $\nabla \vdash s : A[\sigma \cdot w] \circledR_{\mathscr{T}}$ $\mathsf{inSemTy}\ p_A \ni p$, then $\nabla \vdash B[\sigma \cdot w, s] \circledR_{\mathscr{T}}$ $\mathsf{inSemTy}\ (p_B\ p)$;
5. $\Gamma \vdash t : T \circledR_{\mathscr{U}}$ $\mathsf{uNe}\ e \ni p$ iff $p : \mathsf{isDne}\ a$ for some $a : \mathsf{D}$, and for all Δ, w such that $\Delta \vdash_r w : \Gamma$,
   (a) $\Delta \vdash T[w] \sim \mathsf{reifyNe}\ |\Delta|\ e$;
   (b) $\Delta \vdash t[w] \sim \mathsf{reify}\ |\Delta|\ a : T[w]$.
6. $\Gamma \vdash t : T \circledR_{\mathscr{T}}$ $\mathsf{inj}_{\mathscr{U}}\ x \ni p$ iff $\Gamma \vdash t : T \circledR_{\mathscr{U}}\ x \ni p$;
7. $\Gamma \vdash t : T \circledR_{\mathscr{T}}$ $\mathsf{tU} \ni p$ iff $p : a \in_t \mathscr{U}$ for some $a : \mathsf{D}$, and
   (a) $\Gamma \vdash T \sim \mathsf{U}$;
   (b) $\Gamma \vdash t \circledR_{\mathscr{U}}\ p$;
   (c) For all Δ, w such that $\Delta \vdash_r w : \Gamma$, $\Delta \vdash t[w] \sim \mathsf{reify}\ |\Delta|\ a : \mathsf{U}$
8. $\Gamma \vdash r : R \circledR_{\mathscr{T}}$ $(\mathsf{tPi}\ \{A\}\ \{B\}\ \{\rho\}\ p_A\ p_B) \ni p$ iff $p : f \in_t (\mathsf{tPi}\ \{A\}\ \{B\}\ \{\rho\}\ p_A\ p_B)$ for some $f : \mathsf{D}$ and
   (a) $f \equiv \lambda\mathsf{Clo}\ t\ \rho$, and there are Δ, σ such that
      i. $\Gamma \vdash R \sim (\Pi\ A\ B)[\sigma]$;
      ii. $\Gamma \vdash r \sim (\lambda t)[\sigma] : R$;
      iii. For all ∇, w such that $\nabla \vdash_r w : \Gamma$, then $\nabla \vdash_s \sigma \cdot w \sim \mathsf{reifyEnv}\ |\nabla|\ \rho : \Delta$;
      iv. For all ∇, w such that $\nabla \vdash_r w : \Gamma$, $q : a \in_t \mathsf{El}_{\mathscr{T}}(\mathsf{inSemTy}\ p_A))$, and $\nabla \vdash s : A[\sigma \cdot w] \circledR_{\mathscr{T}}$ $\mathsf{inSemTy}\ p_A \ni q$, then $\nabla \vdash t[\sigma \cdot w, s] : B[\sigma \cdot w, s] \circledR_{\mathscr{T}}$ $\mathsf{inSemTy}\ (p_B\ p) \ni (\pi_2 \circ \pi_2)\ (p\ q)$.
   (b) $f \equiv \mathsf{DNe}\ e$, and for all Δ, w such that $\Delta \vdash_r w : \Gamma$, we have $\Delta \vdash r[w] \sim \mathsf{reifyNe}\ |\Delta|\ e : R[w]$.

Just like in the previous chapter, the fundamental lemma will be shown w.r.t. substitutions and environment providing terms and values replacing the free variables of the input term, that must be point-wise related. We thus define the following relation between a substitution σ and an environment ρ on a context Δ of free variables, or rather, between σ, ρ, and a proof of $\rho \in_s [\![\Delta]\!]_s$.

$$\begin{aligned}\_ \vdash_s \_ : \_ \circledR \_ : \mathsf{Ctxt} &\to (\Delta : \mathsf{Ctxt}) \to \mathsf{Subst} \\ &\to \{\rho : \mathsf{Env}\} \to \rho \in_s [\![\Delta]\!]_s \to \mathsf{Set}\end{aligned}$$

$$\frac{}{\diamond \vdash_s \mathsf{Id} : \diamond \circledR \mathsf{cEmpty}} \qquad \frac{\begin{array}{c}\Gamma \vdash t : A[\sigma] \circledR_{\mathscr{T}} \mathsf{inTy}\ q \ni \mathsf{inTm}\ q \\ p : \rho \in_s [\![\Delta]\!]_s \qquad q : a \in_t [\![A]\!]_t\ \rho \qquad \Gamma \vdash_s \sigma : \Delta \circledR p\end{array}}{\Gamma \vdash_s (\sigma, t) : \Delta, A \circledR \mathsf{cExt}\ p\ q}$$

$$\frac{\begin{array}{c}p : \rho \in_s [\![\Delta]\!]_s \\ \Gamma \vdash_s \sigma : \Delta \circledR p \qquad \nabla \vdash_r w : \Gamma\end{array}}{\nabla \vdash_s \sigma \cdot w : \Delta \circledR p} \qquad \frac{\begin{array}{c}p : \rho \in_s [\![\Delta]\!]_s \\ \Gamma \vdash_s \sigma : \Delta \circledR p \qquad \Gamma \vdash_s \tau \sim \sigma : \Delta\end{array}}{\nabla \vdash_s \tau : \Delta \circledR p}$$

Before proving the fundamental theorems, we need to establish some properties of logically-related types, terms, and substitutions.

Lemma 36. Let $p : X \in_t \mathscr{T}$, $e : \mathsf{Dne}$, and $\Gamma \vdash t : T$ a well-typed term be such that for any w, Δ and weakening $\Delta \vdash_r w : \Gamma$, we have

1. $\Delta \vdash T[w] \sim \mathsf{reify}\ |\Delta|\ X$;
2. $\Delta \vdash t[w] \sim \mathsf{reifyNe}\ |\Delta|\ e : T[w]$.

Then $\Gamma \vdash t : T \circledR_{\mathscr{T}} p \ni q$, where $q : a \in_t \mathsf{El}_{\mathscr{T}}\ p$ which exists by definition of semantic type.

Proof. By induction on p. □

We can also show that logical relations are preserved by judgmental equality (they are also trivially preserved by semantic equality, since that is simply syntactic identity).

Lemma 37. Let $p_1 : X \in_t \mathscr{U}$, $p_2 : Y \in_t \mathscr{T}$, $q_1 : a \in_t \mathsf{El}_{\mathscr{U}}\ p_1$, $q_2 : b \in_t \mathsf{El}_{\mathscr{T}}\ p_2$ for some X, Y, a, b. Then

1. If $\Gamma \vdash A \circledR_{\mathscr{U}} p_1$ and $\Gamma \vdash A \sim B$, then $\Gamma \vdash B \circledR_{\mathscr{U}} p_1$;
2. If $\Gamma \vdash A \circledR_{\mathscr{T}} p_2$ and $\Gamma \vdash A \sim B$, then $\Gamma \vdash B \circledR_{\mathscr{T}} p_2$;
3. If $\Gamma \vdash t : A \circledR_{\mathscr{U}} p_1 \ni q_1$, $\Gamma \vdash A \sim B$, and $\Gamma \vdash t \sim s : A$, then $\Gamma \vdash s : B \circledR_{\mathscr{U}} p_1 \ni q_1$;
4. If $\Gamma \vdash t : A \circledR_{\mathscr{T}} p_2 \ni q_2$, $\Gamma \vdash A \sim B$, and $\Gamma \vdash t \sim s : A$, then $\Gamma \vdash s : B \circledR_{\mathscr{T}} p_2 \ni q_2$.

Proof. By induction on p_1, p_2 and case analysis on the proof terms of the logical relations. See `Semantics.Soundness.LogicalRelation.Preservation`. □

We now prove that the logical relations are indeed Kripke, in the sense that they are preserved by context extensions $\Gamma \vdash_r w : \Delta$.

Lemma 38. Let $p_1 : X \in_t \mathscr{U}$, $p_2 : Y \in_t \mathscr{T}$, $q_1 : a \in_t \mathsf{El}_{\mathscr{U}}\ p_1$, $q_2 : b \in_t \mathsf{El}_{\mathscr{T}}\ p_2$ for some X, Y, a, b. Then

1. If $\Gamma \vdash A \circledR_{\mathscr{U}}\ p_1$ and $\Delta \vdash_r w : \Gamma$, then $\Delta \vdash B[w] \circledR_{\mathscr{U}}\ p_1$;
2. If $\Gamma \vdash A \circledR_{\mathscr{T}}\ p_2$ and $\Delta \vdash_r w : \Gamma$, then $\Delta \vdash B[w] \circledR_{\mathscr{T}}\ p_2$;
3. If $\Gamma \vdash t : A \circledR_{\mathscr{U}}\ p_1 \ni q_1$ and $\Delta \vdash_r w : \Gamma$, then $\Delta \vdash s[w] : B[w] \circledR_{\mathscr{U}}\ p_1 \ni q_1$;
4. If $\Gamma \vdash t : A \circledR_{\mathscr{T}}\ p_2 \ni q_2$ and $\Delta \vdash_r w : \Gamma$, then $\Delta \vdash s[w] : B[w] \circledR_{\mathscr{T}}\ p_2 \ni q_2$.

Proof. By induction on the proof of logical relations. See the module `Semantics.Soundness.LogicalRelation`
□

The fundamental meaning that we want to attach to logical relations is that a syntactic term t is related to a semantic value a whenever t is convertible to the reification of a from the semantics back to the syntax. We can show that this is indeed the case:

Lemma 39. Let $p_1 : X \in_t \mathscr{U}$, $p_2 : Y \in_t \mathscr{T}$, $q_1 : a \in_t \mathsf{El}_{\mathscr{U}}\ p_1$, $q_2 : b \in_t \mathsf{El}_{\mathscr{T}}\ p_2$ for some X, Y, a, b. Then, for any Δ, w such that $\Delta \vdash_r w : \Gamma$, we have

1. If $\Gamma \vdash A \circledR_{\mathscr{U}}\ p_1$, then $\Delta \vdash A[w] \sim \mathsf{reify}\ |\Delta|\ X$;
2. If $\Gamma \vdash A \circledR_{\mathscr{T}}\ p_2$, then $\Delta \vdash A[w] \sim \mathsf{reify}\ |\Delta|\ Y$;
3. If $\Gamma \vdash t : A \circledR_{\mathscr{U}}\ p_1 \ni q_1$, then $\Delta \vdash t[w] \sim \mathsf{reify}\ |\Delta|\ a : A[w]$;
4. If $\Gamma \vdash t : A \circledR_{\mathscr{T}}\ p_2 \ni q_2$, then $\Delta \vdash t[w] \sim \mathsf{reify}\ |\Delta|\ b : A[w]$;

Proof. By induction on p_1, p_2 and case analysis on the proof terms of the logical relations. See `Semantics.Soundness.LogicalRelation.Conversion`. □

To show the fundamental lemmas, we define the following validity judgments for well-formed types, well-typed terms, and well-typed substitutions. These have the types

$$
\begin{aligned}
&\_ \models \_\langle\_\rangle : (\Gamma : \mathsf{Ctxt}) \to (T : \mathsf{Term}) \to \Gamma \vdash T \to \mathsf{Set}\\
&\_ \models \_ : \_\langle\_\rangle : (\Gamma : \mathsf{Ctxt}) \to (t : \mathsf{Term}) \to (T : \mathsf{Term}) \to \Gamma \vdash t : T \to \mathsf{Set}\\
&\_ \models_s \_ : \_ : \mathsf{Ctxt} \to \mathsf{Subst} \to \mathsf{Ctxt} \to \mathsf{Set}
\end{aligned}
$$

and are defined as follows:

$$\begin{aligned}
\Gamma \models T\langle x\rangle &:\equiv (p : \rho \in_s [\![\Gamma]\!]_s) \to \Delta \vdash_s \sigma : \Gamma \circledR\ p \to \Delta \vdash T[\sigma] \circledR_{\mathscr{T}} (\pi_1 \circ \pi_2)\ (\!|x|\!) \\
\Gamma \models t : T\langle x\rangle &:\equiv (p : \rho \in_s [\![\Gamma]\!]_s) \to \Delta \vdash_s \sigma : \Gamma \circledR\ p \\
&\qquad \to \Delta \vdash t[\sigma] : T[\sigma] \circledR_{\mathscr{T}} (\mathsf{inEqTy} \circ \pi_2)\ (\!|x|\!) \ni (\mathsf{inEqTm} \circ \pi_2)\ (\!|x|\!) \\
\Gamma \models \gamma : \Theta &:\equiv (p : \rho \in_s [\![\Gamma]\!]_s) \to (p' : \rho' \in_s [\![\Theta]\!]_s) \\
&\qquad \to \Delta \vdash_s \delta : \Gamma \circledR\ p \to [\![\gamma]\!]_s\ \rho \searrow \rho' \to \Delta \vdash_s \gamma \cdot \delta : \Theta \circledR\ p'
\end{aligned}$$

Notice that the first two validity judgments are *proof-relevant*, in the sense that they consider types and terms together with a proof of well-formedness/typedness. This is because the logical relations mention the semantic values that result from evaluation, which is only known to be terminating for well-formed types/well-typed terms. We are finally ready to establish the fundamental lemma of logical relations:

Lemma 40. [Fundamental lemma]

1. If $p : \Gamma \vdash T$, then $\Gamma \models T\langle p\rangle$;
2. If $p : \Gamma \vdash t : T$, then $\Gamma \models t : T\langle p\rangle$;
3. If $\Delta \vdash_s \sigma : \Gamma$, then $\Delta \models_s \sigma : \Gamma$.

Proof. By induction on the derivation. We refer the interested reader to the module `Semantics.Soundness.Soundness` for the details. □

Consider now the identity substitution assigning every free variable of a context of assumptions to itself, defined as follows

$$\begin{aligned}
&\mathsf{idsub} : \mathsf{Ctxt} \to \mathsf{Subst} \\
&\mathsf{idsub}\ \diamond = \mathsf{Id} \\
&\mathsf{idsub}\ (\Gamma, A) = (\mathsf{idsub}\ \Gamma \cdot \uparrow), \mathsf{v}_0
\end{aligned}$$

We can see that idsub does indeed construct an identity substitution, and moreover that it is logically-related to identity environments.

Lemma 41. Let $c : \vdash \Gamma$. Then,

1. $\Gamma \vdash_s \mathsf{idsub}\ \Gamma : \Gamma$, and $\Gamma \vdash_s \mathsf{idsub}\ \Gamma \sim \mathsf{Id} : \Gamma$;
2. $\Gamma \vdash_s \mathsf{idsub}\ \Gamma : \Gamma \circledR\ \mathsf{idenvp}\ c$.

Proof. Both points proved by induction on the proof of $\vdash \Gamma$. The second point then uses the first, as well as Lemmas 40, 39. □

We are now ready to establish soundness of NbE, as a corollary of the fundamental lemmas, and the properties of identity substitutions and environments.

Theorem 15 (Soundness of NbE). Let $c : \vdash \Gamma, p :\ \Gamma \vdash A, q :\ \Gamma \vdash t : A$. Then

1. $\Gamma \vdash A \sim \mathsf{nf\text{-}ty}\ c\ p$;
2. $\Gamma \vdash t \sim \mathsf{nf\text{-}tm}\ c\ q : A$.

Proof. By application of Lemma 40 on the identity substitution and environment, followed by Lemma 39 to establish conversion of the semantic values with the reified normal forms. See `Semantics.Soundness.Soundness.soundness-ty` and `Semantics.Soundness.Soundness.soundness-tm` for the details. □

As a direct corollary of soundness and completeness of NbE we get decidability of judgmental equality, that in the case of a dependent type theory like MLTT^{wk} is a crucial result, since it is required for deciding type checking.

Corollary 5 (Decidability of judgmental equality)**.** Let $c : \vdash \Gamma$, $p_A :\ \Gamma \vdash A$, $p_B :\ \Gamma \vdash B$, $q_t :\ \Gamma \vdash t : A$, $q_s :\ \Gamma \vdash s : A$ be well-formed types and well-typed terms. Then,

1. $\Gamma \vdash A \sim B$ is decidable;
2. $\Gamma \vdash t \sim s : A$ is decidable.

Proof. By soundness and completeness of NbE, we have that $\Gamma \vdash A \sim B \iff \mathsf{nf\text{-}ty}\ c\ p_A = \mathsf{nf\text{-}ty}\ c\ p_B$, and similarly $\Gamma \vdash t \sim s : A \iff \mathsf{nf\text{-}tm}\ c\ q_A = \mathsf{nf\text{-}tm}\ c\ q_B$. Syntactic identity of normal forms, which are just elements of type Term, is decidable, therefore so is convertibility. □

Chapter 7

Conclusion

7.1 Summary of the contributions and future work

This thesis gives an analysis of a selected class of weak notions of reductions for typed λ-calculi, particularly with regards to constructive proofs of normalization. The work was motivated by the open normalization problem of mTT, a theory with CH-weak conversion. Unlike other weak relations, like *weak-head reduction*, CH-weak reduction does allow certain terms to be reduced under binders, what we call *weak redexes* after [22]. Moreover, the definition of weak redex is *relative*, since it depends on what term we are considering the redex a subterm of. These aspects make proving normalization for calculi with CH-weak reduction particularly challenging.

In the first part of the thesis we address and solve the normalization problem for a version of System T with CH-weak conversion as typed equality judgments, called System T^{wk}, by providing a fully-formalized proof of normalization by evaluation. The proof method is novel, and relies on the construction of an "explicit calculus" where certain aspects of CH-weak reduction that are made evident into the syntax of judgments. This allows to express CH-weak computation rules, as well as all congruence rules for term constructors, in a direct way. Moreover, our work appears to be the first constructive analysis of normalization for a typed λ-calculus with CH-weak reduction.

In the second part of the thesis, we address weak reduction in the context of dependent types. We first consider CH-weak reduction, and observe that certain lemmas that are crucial to establish normalization for System T^{wk} are complicated by the presence of dependent types. For this reason, in the rest of Chapter 6 we consider another weak conversion relation, *weak explicit substitutions*, as an alternative to CH-weak conversion in the definition of dependent type theories with weak notions of equality. We do so by providing an original formulation of Martin-Löf Type Theory with weak explicit substitutions, and give a full-formalized proof of normalization by evaluation for the calculus.

With respect to CH-weak reduction, weak explicit substitutions are more convenient from the point of view of both term rewriting and computer formalization. Weak explicit substitutions have an intuitive equational theory, that just amounts to pushing substitutions around. No metalinguistic operations are involved, which means that less lemmas have to be shown. Their *weak* character is given by the fact that substitutions are never pushed under binders, which additionally simplifies their definition. Moreover, explicit substitutions are closer to actual machine implementations of the λ-calculus, that usually rely on nameless syntax and delayed substitutions [7], so they lead to a cleaner, in addition to more concise, formalization.

As we have shown in Chapter 3, weak explicit substitutions are weaker than CH-weak conversion, in the sense that strictly less equations hold. Nevertheless, they still exhibit the good properties that set CH-weak reduction apart from weak-head reduction, namely admissible substitution rules (which hold by definition) and confluence, while showing none of the problems, namely a relative notion of redex and a reduction behaviour that is at odds with the lack of congruence rules.

Given the advantages of weak explicit substitutions over CH-weak conversion, it makes sense to consider the possibility of reformulating mTT or part of in terms of them. Let us call this hypothetical reformulation mTT^{ex}. Then, there are a couple of things to consider in order to convince ourselves that mTT^{ex} is a good candidate for the intensional level of the Minimalist Foundation:

- The extensional level of the Minimalist Foundation is interpreted onto the intensional one via a quotient model $\mathcal{Q}(\mathsf{mTT})$ built from mTT. To be able to use mTT^{ex} as an alternative formulation of the intensional level, we must ensure that the quotient model $\mathcal{Q}(\mathsf{mTT}^{ex})$ built from it is sufficient for the extensional level to be interpreted onto. This hypothesis seems plausible, since although mTT^{ex} and mTT are not equivalent, they should be w.r.t. the quotient model construction, since there equality of functions is extensional, and thus differences in the intensional presentation of functional terms matter less;

- mTT^{ex} must admit a Kleene realizability model in order to be compatible with the *proofs-as-programs* paradigm [42]. Weak explicit substitutions give rise to an equational theory than is *contained* in the CH-weak conversion of mTT, so it should be possible to give an interpretation of mTT^{ex} into mTT, via the obvious translation that just takes terms of mTT^{ex} and performs all suspended explicit substitutions inside of them via the usual meta-theoretic substitution operation of mTT. This would provide a model of mTT^{ex} in mTT, from which we would get Kleene realizability semantics for mTT^{ex}.

We thus see two possible directions for future work in the Minimal Type Theory:

- We could continue the development of the theory of dependently-typed weak explicit substitutions, with the goal of integrating them into mTT. The previous chapter seems to suggest that weak explicit substitutions do play well with full dependent types, albeit we only considered an empty universe. It should be possible to extend the proof with a non-empty universe closed under dependent products and other type formers. Employing explicit substitutions in the Minimal Type Theory requires further analysis, as discussed above.

- Another possible direction is to continue with the meta-theoretical study of CH-weak conversion, in the context of dependent types and eventually mTT. One possibility is to find a way to adapt the "explicit" type system construction used with System T to dependent types. Once normalization is proved for MLTT with CH-weak conversion and one universe, it should be possible to extend the result to mTT, since the underlying notion of reduction is identical.

Another opportunity for future developments regards the efficient implementation of CH-weak normalization. The normalization function that we defined in Chapter 3, although convenient for our purposes, is computationally demanding: every time a redex is encountered, the whole term has to be fully traversed to check for the absence of locally-free variables. Moreover, every β-contraction is performed fully, via an operation of substitution that again involves a full traversal of terms at every step of reduction. The efficient implementation of reduction in the λ-calculus is a non-trivial task, and usually relies on calculi with sharing and explicit substitutions. According to our knowledge, it is still not known how to give a formulation of CH-weak reduction with explicit substitutions that is equivalent to the implicit one, in such a way that the first can be used to compute normal forms of the second.

7.2 Related work

In addition to the already discussed CH-weak reduction, several works in the literature address other weak reduction strategies for the λ-calculus, particularly in connection with types and normalization. In [45], Per Martin-Löf gives a particular formulation of his Intuitionistic Type Theory—that includes a primitive substitution rule like mTT and System T^{wk}—and proves normalization by a model construction. However, functional terms are not given as λ-abstractions, but as combinator constants. Hence, the notion of reduction that results corresponds to weak-head reduction rather than CH-weak reduction, since substitution never operates under binders. In [37, 38], Kesner et al. study weak-head reduction in the context of intersection types and call-by-need reduction strategies. In [34], Hyland and Ong construct a PCA of strongly-normalizing λ-terms as a basis for a general method to prove strong normalization for various type theories. The notion of equality in the PCA is a weak conversion relation similar to CH-weak conversion, that only contracts closed redexes. In [11], Akama introduces a translation from λ-terms to combinators, so that a term is strongly-normalizing under strong β reduction if and only if its translation is strongly-normalizing under the weak conversion of combinatory logic.

The proofs of normalization shown in this work are based on Normalization by Evaluation. NbE was first employed by Martin-Löf in [45] for his combinatory theory, although not under this name. The method was later rediscovered in [18] in the context of the simply-typed λ-calculus with η equality. Coquand and Dybjer [25] later revisited some of the ideas of [45], and formalized NbE for combinators and a λ-calculus with weak-head reduction using a model construction inspired by the categorical notion of *glueing*. A technique that is similar to NbE is big-step normalization, as proposed by

Altenkirch and Chapman in [12]. A major difference of big-step normalization is that it is entirely first-order, unlike many definitions of NbE that instead require a metatheory where higher-order functions are a primitive notion.

In this thesis, we use Normalization by Evaluation in its untyped variant, as described in [8]. Essentially, untyped NbE is a semantic argument for normalization that amounts to defining a normalization function on the raw, untyped syntax, and then proving it correct and terminating via a complete model construction. A drawback is that evaluation is thus inevitably partial, therefore it does not have a primitive definition in Type Theory, and must be given as a functional relation, or via the Bove-Capretta method [20]. An advantage of untyped NbE is that it can be defined ignoring many details of typing that do not affect the reduction behaviour of raw terms. For this reason, untyped NbE scales particularly well to dependent types and impredicative systems [8]. Nevertheless, there have been recent developments towards the successful formalization of a fully-typed NbE for dependent type theories with intrinsic syntax [14].

7.3 Formalization

All the mathematical content of this thesis has been formalized and proof-checked with the Agda programming language and proof assistant. A release of this formalization is available on Github at the repositories [5, 4]. Tarballs of the source code [2, 1] can be downloaded directly from the urls
`https://github.com/fsestini/nbe-weak-systemt/archive/v1.0.tar.gz` and `https://github.com/fsestini/nbe-mltt-wes/archive/v1.0.tar.gz`.

7.4 Acknowledgments

The author wishes to thank his supervisor Maria Emilia Maietti for the support during the development of this work, as well as Claudio Sacerdoti Coen, Thorsten Altenkirch, and Delia Kesner for valuable feedback.

Bibliography

[1] Agda formalization of MLTT with weak explicit substitutions. `https://github.com/fsestini/nbe-mltt-wes/archive/v1.0.tar.gz`.

[2] Agda formalization of Weak System T. `https://github.com/fsestini/nbe-weak-systemt/archive/v1.0.tar.gz`.

[3] Agda's documentation. `https://agda.readthedocs.io/en/v2.5.3/`.

[4] Repository of the Agda formalization of MLTT with weak explicit substitutions. `https://github.com/fsestini/nbe-mltt-wes/releases/tag/v1.0`.

[5] Repository of the Agda formalization of Weak System T. `https://github.com/fsestini/nbe-weak-systemt/releases/tag/v1.0`.

[6] *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis.* Lecture Notes in Mathematics. Springer-Verlag, 1973. ISBN 978-3-540-06491-6.

[7] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, Oct 1991. ISSN 1469-7653, 0956-7968. doi:10.1017/S0956796800000186.

[8] A. Abel. *Normalization by Evaluation, Dependent Types and Impredicativity.* Habilitation thesis, Institut für Informatik, Ludwig-Maximilians-Universität München, 2013.

[9] Andreas Abel, Klaus Aehlig, and Peter Dybjer. Normalization by Evaluation for Martin-Löf Type Theory with One Universe. *Electronic Notes in Theoretical Computer Science*, 173:17–39, Apr 2007. ISSN 1571-0661. doi:10.1016/j.entcs.2007.02.025.

[10] Andreas Abel, Thierry Coquand, and Peter Dybjer. Verifying a Semantic $\beta\eta$-Conversion Test for Martin-Löf Type Theory. In Philippe Audebaud and Christine Paulin-Mohring, editors, *Mathematics of Program Construction*, pages 29–56. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. ISBN 978-3-540-70594-9.

[11] Yohji Akama. A λ-to-CL translation for strong normalization. In Philippe de Groote and J. Roger Hindley, editors, *Typed Lambda Calculi and Applications*, pages 1–10. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997. ISBN 978-3-540-68438-1.

[12] Thorsten Altenkirch and James Chapman. Big-step Normalisation. *J. Funct. Program.*, 19(3–4):311–333, Jul 2009. ISSN 0956-7968. doi:10.1017/S0956796809007278.

[13] Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. *Categorical Reconstruction of a Reduction Free Normalization Proof.* 1995.

[14] Thorsten Altenkirch and Ambrus Kaposi. Type Theory in Type Theory Using Quotient Inductive Types. *SIGPLAN Not.*, 51(1):18–29, January 2016. ISSN 0362-1340. doi:10.1145/2914770.2837638.

[15] H. P. Barendregt. *Handbook of Logic in Computer Science (Vol. 2)*, page 117–309. Oxford University Press, Inc., 1992. ISBN 978-0-19-853761-8.

[16] Hendrik Pieter Barendregt. *The lambda calculus: its syntax and semantics.* North-Holland, 1984. ISBN 978-0-444-86748-3.

[17] Oskar Becker. H. B. Curry and R. Feys, Combinatory Logic. *Philosophische Rundschau*, 6(3/4):294, 1958.

[18] U. Berger and H. Schwichtenberg. An inverse of the evaluation functional for typed lambda-calculus. In *[1991] Proceedings Sixth Annual IEEE Symposium on Logic in Computer Science*, page 203–211. Jul 1991. doi:10.1109/LICS.1991.151645.

[19] M. Bezem, J.W. Klop, Terese, and R. de Vrijer. *Term Rewriting Systems.* Cambridge Tracts in Theoretica. Cambridge University Press, 2003. ISBN 9780521391153.

[20] Ana Bove and Venanzio Capretta. Modelling General Recursion in Type Theory. *Mathematical. Structures in Comp. Sci.*, 15(4):671–708, August 2005. ISSN 0960-1295. doi:10.1017/S0960129505004822.

[21] Ana Bove, Peter Dybjer, and Ulf Norell. A Brief Overview of Agda - A Functional Language with Dependent Types. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, pages 73–78. 2009. doi:10.1007/978-3-642-03359-9_6.

[22] Naim Çağman and J.Roger Hindley. Combinatory weak reduction in lambda calculus. *Theoretical Computer Science*, 198(1):239 – 247, 1998. ISSN 0304-3975. doi:https://doi.org/10.1016/S0304-3975(97)00250-8.

[23] Alonzo Church. *The Calculi of Lambda-conversion.* Princeton University Press, 1985. ISBN 978-0-691-08394-0.

[24] Thierry Coquand. *Logical Frameworks*, page 255–279. Cambridge University Press, 1991. ISBN 978-0-521-41300-8.

[25] Thierry Coquand and Peter Dybjer. Intuitionistic Model Constructions and Normalization Proofs. *Mathematical Structures in Computer Science*, 7:75–94, Feb 1997. doi:10.1017/S0960129596002150.

[26] Pierre-Louis Curien, Thérèse Hardin, and Jean-Jacques Lévy. Confluence Properties of Weak and Strong Calculi of Explicit Substitutions. *J. ACM*, 43(2):362–397, March 1996. ISSN 0004-5411. doi:10.1145/226643.226675.

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

[28] Peter Dybjer. Inductive Families. *Formal Aspects of Computing*, 6:440–465, 1997.

[29] Peter Dybjer. A General Formulation of Simultaneous Inductive-Recursive Definitions in Type Theory. *J. Symbolic Logic*, 65(2):525–549, 06 2000.

[30] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. Cambridge University Press, 1989. ISBN 978-0-521-37181-0.

[31] Robert Harper. *Practical Foundations for Programming Languages*. 2012.

[32] W.A. Howard. Assignment of Ordinals to Terms for Primitive Recursive Functionals of Finite Type. 1970.

[33] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A History of Haskell: Being Lazy with Class. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, pages 12–1–12–55. ACM, New York, NY, USA, 2007. ISBN 978-1-59593-766-7. doi:10.1145/1238844.1238856.

[34] J.M.E. Hyland and C. h. L. Ong. Modified Realizability Toposes and Strong Normalization Proofs (Extended Abstract). In *Typed Lambda Calculi and Applications, LNCS 664*, pages 179–194. Springer-Verlag, 1993.

[35] Hajime Ishihara, Maria Emilia Maietti, Samuele Maschio, and Thomas Streicher. Consistency of the intensional level of the Minimalist Foundation with Church's thesis and axiom of choice. *Archive for Mathematical Logic*, Jan 2018. ISSN 1432-0665. doi:10.1007/s00153-018-0612-9.

[36] Achim Jung and Jerzy Tiuryn. *A New Characterization of Lambda Definability*. 1993.

[37] Delia Kesner. Reasoning About Call-by-need by Means of Types. In Bart Jacobs and Christof Löding, editors, *Foundations of Software Science and Computation Structures*, pages 424–441. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016. ISBN 978-3-662-49630-5.

[38] Delia Kesner, Alejandro Ríos, and Andrés Viso. Call-by-Need, Neededness and All That. In Christel Baier and Ugo Dal Lago, editors, *Foundations of Software Science and Computation Structures*, pages 241–257. Springer International Publishing, Cham, 2018. ISBN 978-3-319-89366-2.

[39] P. J. Landin. The Next 700 Programming Languages. *Commun. ACM*, 9(3):157–166, March 1966. ISSN 0001-0782. doi:10.1145/365230.365257.

[40] Pierre Lescanne and Jocelyne Rouyer-Degli. Explicit substitutions with De Bruijn's levels. In *Rewriting Techniques and Applications*, Lecture Notes in Computer Science, page 294–308. Springer, Berlin, Heidelberg, Apr 1995. ISBN 978-3-540-59200-6. doi:10.1007/3-540-59200-8_65.

[41] Jean-Jacques Lévy and Luc Maranget. Explicit Substitutions and Programming Languages. In *Foundations of Software Technology and Theoretical Computer Science*, Lecture Notes in Computer Science, page 181–200. Springer, Berlin, Heidelberg, Dec 1999. ISBN 978-3-540-66836-7. doi:10.1007/3-540-46691-6_14.

[42] M.E. Maietti. A minimalist two-level foundation for constructive mathematics. *Annals of Pure and Applied Logic*, 160(3):319 – 354, 2009.

[43] M.E. Maietti and G. Sambin. Toward a Minimalist Foundation for Constructive Mathematics. *From Sets and Types to Topology and Analysis: Practicable Foundations for Constructive Mathematics*, 48, 2005.

[44] Per Martin-Löf. 100 Years of Zermelo's Axiom of Choice: What Was the Problem with It? *Comput. J.*, 49(3):345–350, May 2006. ISSN 0010-4620. doi:10.1093/comjnl/bxh162.

[45] Per Martin-Löf. *An Intuitionistic Theory of Types: Predicative Part*, volume 80 of *Logic Colloquium '73*, page 73–118. Elsevier, Jan 1975. doi:10.1016/S0049-237X(08)71945-1.

[46] Per Martin-Löf. Constructive Mathematics and Computer Programming. In L. Jonathan Cohen, Jerzy Łoś, Helmut Pfeiffer, and Klaus-Peter Podewski, editors, *Logic, Methodology and Philosophy of Science VI*, volume 104 of *Studies in Logic and the Foundations of Mathematics*, pages 153 – 175. Elsevier, 1982.

[47] Per Martin-Löf. *Intuitionistic type theory.* Bibliopolis, 1984.

[48] Conor McBride. Dependently Typed Functional Programs and their Proofs. Technical report, 2000.

[49] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory: An Introduction.* Clarendon Press, 1990. ISBN 978-0-19-853814-1.

[50] Ulf Norell. Dependently typed programming in Agda. In *Proceedings of TLDI'09: 2009 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Savannah, GA, USA, January 24, 2009*, pages 1–2. 2009. doi:10.1145/1481861.1481862.

[51] M. Pagano. *Type-Checking and Normalization by Evaluation for Dependent Type Systems.* Phd thesis, Universidad Nacional de Cordoba, 2012.

[52] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002. ISBN 978-0-262-16209-8.

[53] Randy Pollack. Closure under alpha-conversion. In *Types for Proofs and Programs*, Lecture Notes in Computer Science, page 313–332. Springer, Berlin, Heidelberg, May 1993. ISBN 978-3-540-58085-0. doi:10.1007/3-540-58085-9_82.