Type-Checking and Normalisation By Evaluation for Dependent Type Systems

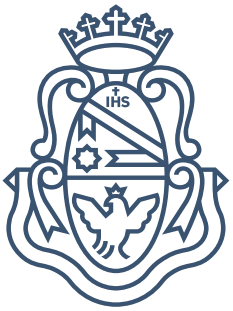
Miguel M. Pagano

Directores: Thierry Coquand y Daniel E. Fridlender

Presentada ante la Facultad de Matemática, Astronomía y Física como parte de los requerimientos para la obtención del grado de Doctor en Ciencias de la Computación de la

Universidad Nacional de Córdoba

Marzo, 2012



Abstract

This thesis presents a new algorithm for Normalisation by Evaluation for different type theories. This algorithm is later used to define a type-checking algorithm and to prove other meta-theoretical results about predicative type systems.

I first present it for simply typed lambda calculus, both with and without η-rule. We use this simple setting to introduce and explain the techniques used for proving the main properties of NbE.

The NbE algorithm is later adapted to Martin-Löf type theory; this algorithm gives rise to a decision procedure for equality. Deciding equality is important because it leads to a type-checking algorithm. We define a type-checking algorithm for a Martin-Löf extended with singleton types and proof-irrelevant types, among other more common types (one universe of small types, dependent function spaces, sigma type, enumeration types, natural numbers). We proved the correctness and completeness of the type-checker. Both algorithms, NbE and type-checking, were implemented in Haskell. At the end of the thesis we sketch an extension of the NbE algorithm to deal with commutativity in MLTT.

We also present a new formulation of Pure Type Systems with explicit substitutions and de Bruijn indices. There are two possible notion for the equality between types: untyped equality between pre-terms and typed equality; it was already know that these two presentations are equivalent. We present a new proof method for that equivalence between predicative PTS. We formalised partially this proof in Agda.

Keywords: Type theory, Normalisation by Evaluation, Type-checking algorithm, Martin-Löf type theory, Pure Type Systems.

Resumen

Esta tesis presenta un nuevo algoritmo de Normalización por Evaluación (NbE) para diferentes teorías de tipos. Utilizamos el algoritmo de normalización para definir un algoritmo de chequeo de tipos y para probar otros meta-teoremas sobre sistemas de tipos predicativos.

Primero consideramos NbE para el cálculo lambda simplemente tipado, tanto con la regla η como sin ella. Utilizando este sistema más sencillo, introducimos y explicamos las técnicas necesarias para probar las principales propiedades del algoritmo.

Luego adaptamos el algoritmo de NbE para una teoría de tipos à la Martin-Löf; este algoritmo da lugar a un procedimiento de decisión para la igualdad. Poder decidir la igualdad formal es importante porque eso permite definir un algoritmo de chequeo de tipos. En la tesis, definimos este algoritmo para una teoría con tipos con un sólo elemento canónico (singleton types) y para proof-irrelevant types (además de otros tipos más comunes y ya considerados en la literatura). Finalmente, probamos la corrección y la completitud del algoritmo de chequeo de tipos. En la última parte de la tesis, bosquejamos cómo se puede extender el algoritmo de decisión de la igualdad para tratar conmutatividad en teoría de tipos.

También se presenta una nueva formulación de Sistemas de Tipos Puros (PTS) con sustituciones explícitas e índices de de Bruijn. Hay dos posibles nociones de igualdad entre tipos: igualdad no tipada sobre pre-términos e igualdad tipada; ya se sabía que estas dos presentaciones son equivalentes. Nosotros presentamos un nuevo método de prueba para dicha equivalencia para PTS predicativos. Hemos formalizado, parcialmente, esta prueba en Agda.

Palabras claves: Teoría de tipos, Normalización por Evaluación, Algoritmo de chequeo de tipos, Teoría de tipos à la Martin-Löf, Sistemas de tipos puros.

1998 ACM Subject Classification: F.4.1 Mathematical Logic.

Contents

1 Introduction 1
- 1.1 Type Theories . . . 1
- 1.2 Using dependent types . . . 4
- 1.3 Type-Checking: Deciding the Typing Relation . . . 11
- 1.4 Related work . . . 13
- 1.5 Our contributions . . . 14

2 Normalisation By Evaluation 17
- 2.1 Syntax of $\lambda^{\to}$ . . . 17
- 2.2 Normalisation by Evaluation for λ^{β} . . . 23
- 2.3 Normalisation by Evaluation for $\lambda^{\to}$ . . . 27
- 2.4 Correctness of NbE . . . 31
- 2.5 A Haskell Implementation of NbE . . . 36

3 NbE for Martin-Löf Type Theory 39
- 3.1 The calculus λ^{Π} . . . 39
- 3.2 Semantics and Normalisation by Evaluation . . . 42
- 3.3 Correctness of NbE via logical relations . . . 46
- 3.4 Implementation of a type-checker for λ^{Π} . . . 51

4 Extended Martin-Löf Type Theory 55
- 4.1 The Calculus . . . 55
- 4.2 Semantics . . . 65
- 4.3 Correctness of NbE . . . 78
- 4.4 Type-checking algorithm . . . 88
- 4.5 Normalisation by evaluation . . . 94
- 4.6 Type-checking algorithm . . . 98

5 Pure Type Systems 101
- 5.1 Formal systems . . . 101
- 5.2 Equivalence between λ^{σ} and $\lambda^{\sigma=}$ . . . 109
- 5.3 Semantics for $\lambda^{\sigma=}$ . . . 109
- 5.4 Correctness of Nbe . . . 114
- 5.5 From λ^{σ} to $\lambda^{\sigma=}$ . . . 118

6 Conclusion 121
- 6.1 Commutativity in Martin-Löf type theory . . . 121
- 6.2 Further work . . . 126

Bibliography 129

Acknowledgements

I'd like to start expressing my gratitude to Daniel Fridlender: he started *all this* seven years ago when he asked me if I was interested in pursuing a Ph.D. He was always patient with my impatience to learn and to express my vague intuitions; he encouraged me when I was not too confident in my abilities. Our meetings were the source of several ideas in the thesis. I also appreciate very much Daniel's humanity and intellectual honesty.

I am thankful to Thierry Coquand for accepting to be my advisor at Chalmers. My year in Sweden was crucial for my progressing, and a part of that should be credited to Thierry: he taught me several of the techniques employed on this thesis and suggested to look at various of the issues covered.

Most of the chapters of this thesis are based on joint works with Andreas Abel, Thierry Coquand, and Daniel Fridlender. I am grateful to them for they kindly granted me to include these works in the thesis. Andreas explained to me various points about dependent type theory; I learned a lot working with him. Danke schön!

I am grateful to Peter Dybjer for accepting to be in the jury, in spite of the long trip from Sweden, and for his valuable suggestions. I thank also to Eduardo Bonelli and Carlos Areces, the Argentinian members of the jury, for reading the thesis and pointing out omissions and errors.

My Ph.D. was supported by scholarships from *Agencia Nacional de Promoción Científica y Tecnológica*, *Consejo Nacional de Investigaciones Científicas y Técnicas*, and the *LerNet* project funded by the EU. Part of my activities and travels were supported by funds provided by *Ministerio de Ciencia de Córdoba* and *Secretaría de Ciencia y Técnica - UNC*.

My roommates and colleagues at FaMAF made of the workplace a nice environment. In particular, I have great times sharing teaching times with Héctor (el Flaco) and Pedro ST. I liked my talks with Ara, Renato, and Javier; with them I discovered a taste for understanding more deeply our daily activities, specially as computer scientist, and questioning the usual state-of-affairs.

I enjoyed the last year at office 332; its atmosphere was so pleasant that someone call it *the smiling office*. I thank my friends Any, Cristian, Chun, Eze, Franco, and Leti for the good time in the office and for the after office hours.

During my stance in Gothenburg I lived with Ruben, who opened his house to me although we did not know before, just because Mónica y Juan asked him to receive me. In great part, I felt at home in Gothenburg, because Ruben adopted me. ¡Muchas gracias! Every meeting with my dearest friends Anna och Anders was joyful: Tack så mycket! I also enjoyed talking with Ana Bove and Andrés Sicard-Ramírez.

My parents and siblings have been supportive in many ways all this time; they never doubted to assist me in every situation when I asked for their help. They respected all my weird decisions, no matter the outcome and always helped me to paid the prices (sometimes literally so). I'd like to thank also to Rafa for his generosity and his hospitality in Barcelona. ¡Muchas gracias!

Vale has been *mi compañera* for most of this journey. She always had tender words when I was anxious or depressed; her backing was most important in the moments of doubt and disappointment. Vale sustained me, even when it meant to live separate for almost a year. Gracias, muchas gracias por todo eso *[...] y [por] lo que paso en silencio*.

Introduction

1

The type systems studied in this thesis can be seen from two perspectives. They can be used as a foundation for constructive mathematics [21]; for example, Martin Löf type theory was originally meant as a formal language for intuitionism [83]. Dependent type systems can also be seen as functional programming languages whose type discipline can be used for enconding the specification of programs [95]. The main contributions of this thesis are the definition of a *normalisation algorithm* that leads to *type-checking algorithms* and *new proof methods for some meta-theoretical results* for predicative type theories.

In the following sections we briefly introduce type theory; then we explain the relationship with constructivism and show a program with a strong specification; finally we shortly discuss some issues related to type-checking dependent type systems and we introduce normalisation by evaluation. In the last section we comment on the organisation of the thesis and the contributions of each chapter.

1.1 Type Theories

The use of types in logic was first proposed by Russell to avoid the paradoxes discovered in formal systems at the dawn of the twentieth century. In this section we briefly expound the type theories studied in the rest of the thesis. [1]

Simple Type Theory: Church and Curry

The simple theory of types was introduced by Church [36] to classify terms of his lambda calculus [35]. In his axiomatisation, types were built up from two basic constants — called the types of individuals and the types of propositions, with symbols ι and o, respectively — and by forming $A \to B$, where A and B are already defined type expressions.

To better explain simple type theory (STT), we introduce the formal terms of Church's lambda calculus: given a set V of variables, the set Λ of terms of the untyped lambda calculus are constructed by the following clauses:

$$\frac{x \in V}{x \in \Lambda}\ (\textsc{var}) \qquad \frac{t \in \Lambda \qquad t' \in \Lambda}{t\,t' \in \Lambda}\ (\textsc{app}) \qquad \frac{x \in V \qquad t \in \Lambda}{\lambda x.t \in \Lambda}\ (\textsc{abs}) \tag{1.1}$$

The rules of the formal system are presented in the following definition.[2]

Definition 1 (Rules of conversion). Let $x, y \in V$ and $r, t, s, s' \in \Lambda$; then $s = s'$ if they are provably equal by using the rules derived from the reflexive, symmetric, transitive, and contextual closure of the following equations:

[1] We refer the reader to [33, 38, 71, 111] for the history of type-theory.

[2] Lambda calculus was intended to be a formal system for predicate logic, thus the original presentation also had constants for logical connectives. We restrict our presentation to the pure lambda calculus and omit the logical constructors and the formal postulates related with them.

(α) If y does not occur freely in t, $\lambda x.t = \lambda y.t[x/y]$

(β) $(\lambda x.t)\, r = t[x/r]$.

(η) If x does not occur freely in t, $\lambda x.t\, x = t$.

Rule (β) makes clear how the lambda notation formalises the notion of function and application of functions to arguments: if t is a term where the variable x occurs, then $\lambda x.t$ is the function such that when applied to r yields the value $t[x/r]$, this being the notation for substituting r for the free occurrences of x in t. In $\lambda x.t$, x is say to be a bound variable. Rule (α) formalises that names of bound variables are absolutely irrelevant and they can be changed (carefully). Finally rule (η) together with congruence under abstraction makes the system extensional; i.e. we can show that two terms are equal iff their application to every term is equal.

Already in [35] there was an idea of computation: the equations are called *reductions* when interpreted as, in modern terminology, rewrite rules transforming the left hand side into the right hand side; in particular one is usually interested in the rewriting relation generated by considering only (β). Given a term t_0 we say that it *evaluates* to t_n if there is a reduction sequence $t_0 \rightsquigarrow t_1 \rightsquigarrow \ldots \rightsquigarrow t_n$, where t_i is transformed into t_{i+1} by applying rule (β) to a sub-term of t_i. We say that t_n is a *normal form* if it cannot be further reduced; then we say that t_n is the *normal form* of t_0 — note that there are terms without normal forms; i.e. terms with all theirs reduction sequences infinite. The lambda calculus is consistent in the sense that a given term can have at most one normal form, up to renaming of bound variables using (α).

After Kleene and Rosser's [73] proof of the inconsistency of logic based on lambda calculus, this untyped version was disregarded as a feasible foundation for logic. The discipline of types of STT [36] avoided the inconsistency; but it also imposed a serious restriction on the expressiveness of the lambda calculus (for example, every typed term is strongly normalising). Instead of adding types on top of untyped terms, Church considered types as part of terms in the form of subscripts: for example, if A and B are distinct types, then x_A and x_B are two different variables. Well-formed terms are given by the following rules:

$$\frac{}{x_A \in \Lambda} \qquad \frac{t_{A\to B} \in \Lambda \qquad t'_A \in \Lambda}{(t\, t')_B \in \Lambda} \qquad \frac{x_A \in V \qquad t_B \in \Lambda}{(\lambda x_A.t)_{A\to B} \in \Lambda} \tag{1.2}$$

Church's STT is still alive as the theoretical foundation of HOL [65] and Isabelle/HOL [93], two of the most prominent proof-assistants based on type theory.

Curry Curry, cf.[48, 49, 50], adopted a different approach to types from that of Church. Instead of building terms out of typed variables, Curry started with untyped terms, those generated by the rules in (1.1); types were later assigned to those terms. Of course, not every untyped lambda term can be assigned a type, and each typeable term can have an infinitude of types. For instance, while in STT $(\lambda x_A.x_A)_{A\to A}$ was a particular term for each type A, in Curry's view $\lambda x.x$ had any type of the form $A \to A$, for any type A. In [50] it is proved the existence of a type-checking algorithm: given a term t and a type A, decide if t can be assigned the type A. Both Hindley [68] and Curry [49]

proved that it is decidable if one can assign any type to a given term; in fact both of them proved that if any type can be given, there exists a most general schema, called the principal schema, which can be inferred from the structure of the term. Hindley's algorithms basically builds a unification problem, over type expressions, by postulating the types each sub-term should have; this procedure is defined by recursion on the structure of the term. The unification problem, then, is to find a substitution for type variables that satisfies all the constraints.

Stronger type-systems: Polymorphism, Martin-Löf, and PTSs

In the last paragraph we saw that in typings à la Curry more than one type can be assigned to a single term. The next step was to understand that phenomenon by adding type variables in the system and analysing the most general type expression assignable by a unification algorithm on type expressions. Girard [61] and Reynolds [98] considered independently the explicit addition of a universal quantifier (Girard's system features also an existential quantifier) to STT à la Curry. This extension is known as the polymorphic lambda calculus; a simpler form of that type system is at the core of functional languages like SML, OCaml, and Haskell. For example we can internalise the fact that the identity function $\lambda x.x$ has type $A \to A$ for each type A by assigning the type $\forall\alpha.\alpha \to \alpha$; in the following we use $t : A$ to say that the term t has the type A. Milner [86] extended Hindley's type-inference algorithm to polymorphic languages where the quantification is only allowed at the top level.

Martin-Löf [83] presented another type theory where types can depend on terms. The most fundamental dependent type is the cartesian product: if A is a type and Ba is also a type for each $a : A$, then $\Pi(x : A)B(x)$ is the type of functions mapping each $a : A$ to an element in Ba. This sort of dependency was previously considered by Howard [69] and by N. G. de Bruijn [30] in his project Automath. The dependently typed programming language Agda [84] is based on Martin-Löf type theory.

Coquand and Huet [41] proposed the Calculus of Constructions (CoC), an impredicative type theory with dependent types. The Edingburgh Logical Framework [63] is weaker than CoC in that dependent types can be only first order: this is enforced by introducing kinds, that are to types (called families in LF) as types are to terms. Kinds are also present in languages like Haskell and SML. In Haskell, for instance, basic types, Bool or Int, have kind $\star$; type constructors, like List, have kind $\star \to \star$; so it will be a type once it is applied to some type of kind $\star$. Berardi [23] and Terlouw [114] presented uniformly several type systems, collectively referred as Pure Type Systems. The basic idea is that all forms of function spaces, from the non-dependent and non-polymorphic of STT, up to the dependent, polymorphic, and impredicative one of CoC, can be captured by the same typing rules by parameterizing the formal system by a set of *sorts*. We refer the reader to Barendregt's [17], the standard reference for PTSs, for more information about them. An extended version of CoC, the Calculus of Inductive Constructions, serves as the theoretical foundation for Coq [85].

1.2 Using dependent types

In this section I explain how dependent type systems can be used both as a basis for writing formal proofs of mathematical theorems and as programming languages whose type systems allow expressing the specification of programs as types. We first explain the use of dependent types as the theoretical foundation of proof assistants; then, we show how to program correctly a type-inference algorithm for STT.

Curry-Howard Isomorphism

In the previous section I sketched the introduction of types and the emergence of more complicated systems; as we moved from the beginning to the present we cited some languages and proof assistants based on the different type systems we mentioned. If we go back to Church's pure lambda calculus we see how logic was treated as a formal system with logical connectives as terms (or term constructors) of the calculus. In this section we review a deeper connection between typing systems and logical formalisms.

Curry was the first to notice that the type schema of his combinators I, K, and S correspond to the axioms of Hilbert's minimal intuitionistic logic. In fact, if we express the combinators as their corresponding lambda terms and then apply Hindley's algorithm we get the following type-schemas ($\to$ associates to the right):

- $I = \lambda x.x \colon A \to A$,
- $K = \lambda x.\lambda y.x \colon A \to B \to A$, and
- $S = \lambda x.\lambda y.\lambda z.(x\,z)\,(y\,z) \colon (A \to B \to C) \to (A \to B) \to A \to C$.

We can think that Hindley's algorithm builds a tree guided by the structure of the term whose type we want to infer. If the algorithm succeeds for some term t, the built tree corresponds to a logical derivation of the propositional formula represented by the inferred type. This means that we can decide if a purported proof of a given theorem is in fact a proof by building a term out of the proof and then applying the type-checking algorithm. Howard [69] later made this correspondence precise and showed how normalisation of terms corresponds to cut-normalisation in proofs. This correspondence was also noticed by Lambek [74], who extended the correspondence of STT with cartesian closed categories.

Identifying proposition with types is one in a series of readings across the history and the different schools of constructivism, which are collectively referred as the Brouwer-Heyting-Kolmogorov interpretation [22, 40, 81, 115]. Kolmogorov viewed propositions as problems; thus to know a proposition true, is to have a method for solving the problem. Brouwer and Heyting's [67] reading is based on the idea that proofs of a composed propositional formula consist in constructions built from the ones corresponding to the sub-formulas involved. For example, a universal statement is intuitionistically true if one has a method to produce, for each element of the domain of discourse, a proof of the instance of the predicate for that particular element; in symbols, say P is a predicate over some type A, the domain of discourse, a proof of $\forall x \in A, Px$

should be a method f which when applied to every $a \in A$ yields a proof of Pa; which fits with the description of dependent products of the previous section.

Let us analyse a concrete example of a proposition to see the formalisation of proofs in type theory. Suppose we want to prove "$1 + 1$ is even". First of all, we need to define a basic type corresponding to the set of natural numbers and define the adding operation; then we need to define a type corresponding to the proposition "n is even".

Martin-Löf proposed [82] that new types can be added to the formal system by giving, and explaining: the conditions for forming the type, the valid forms of constructing (canonical) elements for the new type, the elimination operators for each type, and a method for deciding when two canonical elements of the new type are equal. These conditions are formalised by rules of some *form of judgement*: formation rules, introduction rules, elimination rules, and axioms for equalities.

The *formation rule* for the type of natural numbers is

$$\frac{}{Nat \quad type}$$

there are two *introduction rules* for canonical elements of *Nat*:

$$\frac{}{0 : Nat} \qquad\qquad \frac{n : Nat}{Suc\, n : Nat}$$

Natural numbers can be used, *eliminated*, by the induction operator:

$$\frac{\begin{array}{cccc} [n : Nat] & & [m : Nat \;\; b : A\, m] & \\ \vdots & & \vdots & \\ A\, n \quad type & a : A0 & f\, m\, b : A\, (Suc\, m) & \end{array}}{natrec(n, A, a, f) : A\, n}$$

Notice that A is not a type by itself, but a family of types indexed by natural numbers; we indicate this dependency by putting the hypothesis in brackets. Using the elimination principle for natural numbers we can define addition $n + m = natrec(n, Nat, m, \lambda n'.\lambda p.Suc\, p)$.

Equality rules include those making equality an equivalence relation and also a congruence with respect to constructors. For natural numbers we also have axioms involving the eliminator:

$$natrec(0, A, a, f) = a \qquad\qquad natrec(Suc\, n, A, a, f) = f\, n\, natrec(n, A, a, f)$$

For example, using these axioms, and beta-reduction, we can prove $Suc\, 0 + Suc\, 0 = Suc\, (Suc\, 0)$.

The predicate "n is even" is introduced also with rules; the formation rule for the type corresponding to that propostion should make explicit the condition that n is a natural number:

$$\frac{n : Nat}{Even\, n \quad type}$$

We can say that *Even* is a family of types indexed by *Nat*; the introduction rules correspond to the inductive definition of even numbers:

$$\frac{}{EvZ : Even\, 0} \qquad\qquad \frac{n : Nat \quad evn : Even\, n}{EvInd\, evn : Even\, (Suc\, (Suc\, n))}$$

Notice that *EvZ* has not, a priori, the type *Even* (*Suc* n), for any n: *Nat*. In the proposition-as-type correspondence, a proposition P is true if the type representing P has some inhabitant; i.e., if there is some term with that type. For example, let 1 be *Suc* 0 and 2 be *Suc* 1, we can derive *EvInd EvZ*: *Even* 2. On the other hand we can derive that *Even* 1 + 1 is equal, as a type, to *Even* 2; so, it is to be expected that *EvInd EvZ* had also the type *Even* 1 + 1. In general, different representations of the same proposition should have the same proofs; that means that we need a *conversion* rule:

$$\frac{A \quad \textit{type} \qquad B \quad \textit{type} \qquad A = B \qquad a : A}{a : B}\ (\textsc{conv})$$

Applying conversion we know that a proof of *Even* 2 also counts as a proof of *Even* (1 + 1).

The proposition-as-types principle that we have described justifies the use of type systems as the formalism on which mathematical knowledge is precisely written and proofs can be carried out. In particular, if the type system is decidable, see Sec. 1.3, then it is possible to program a type-checker to verify that some construction, represented as a term, is a proof for some proposition, represented as a type. Geuvers [60] reviews the history of proof assistants, explains the differences between several proof assistants and their theoretical background (some of them are not based on the proposition-as-types principle), and discusses a few issues arising when large pieces of mathematics are formalised. Wiedijk's compilation [118] of formalised proof of the irrationality of $\sqrt{2}$ can give an idea of the outlook of formalisations in different proof assistants. Wiedijk [119] lists the following four theorems as the most impressive pieces of formalised mathematics:

1. Gödel's first incompleteness theorem.
2. Jordan curve theorem.
3. Prime number theorem.
4. Four-colour theorem.

Programming with Correctness

In the previous section we have explained the reasons for constructing proof assistants as implementations on computers of type systems and referred to some papers discussing formalisations of mathematical results. Dependent types allow to program in a correct way by making it possible to write the specification of each function as its type. We illustrate this point by the following implementation in Agda of a type-inference algorithm for a variant of the simple typed lambda calculus.

We only explain the most peculiar aspects of the program; an introduction to Agda can be found in [28].

```
module STT where

open import Data.Product
open import Data.Sum hiding (map)
```

The underscores that appear in definitions are used to indicate arguments of mixfixed operators, be them variables, type constructors, or term constructors. Arguments enclosed between curly braces are implict and can be inferred by Agda. When one needs to know the implicit parameters in an application, they are also enclosed in curly brackets.

```
infixl 10 _∘_
infixr 20 _⇒_
_∘_ : {A B C : Set} → (B → C) → (A → B) → A → C
g ∘ f = λ x → g (f x)
id : {A : Set} → A → A
id x = x
```

Note that the type False lacks any constructor; this fact is recognised by the compiler that permits to define magic. This function can be thought as the elimination principle for falsehood.

```
data False : Set where
magic : {A : Set} → False → A
magic ()
¬_ : Set → Set
¬A = A → False
```

The following type, called *intensional equality*, has only one constructor for equality. We use this type to prove that each term has at most one type. A type, or predicate, is decidable if we know if it has or not any member.

```
data _≡_ {A : Set} : A → A → Set where
  refl : {a : A} → a ≡ a
data Dec (A : Set) : Set where
  yes : (a : A) → Dec A
  no : ¬A → Dec A
```

In the next snippet we introduce the types and prove that their intensional equality is decidable. The **with** construct brings the result of some function call to the left hand side of a definition, allowing to do pattern-matching in the result of the function call.

```
data Ty : Set where
  int  : Ty
  _⇒_ : Ty → Ty → Ty
arrEq : {t t′ s s′ : Ty} → (t⇒t′) ≡ (s⇒s′) → (t ≡ s) × (t′ ≡ s′)
arrEq refl = refl , refl
_≃_ : (t t′ : Ty) → Dec (t ≡ t′)
int ≃ int = yes refl
(t⇒t′) ≃ (s⇒s′) with t ≃ s | t′ ≃ s′
(.s⇒.s′) ≃ (s⇒s′) | yes refl | yes refl = yes refl
(_⇒t′) ≃ (s⇒s′) | _ | no p = no (p ∘ proj₂ ∘ arrEq)
(t⇒_) ≃ (s⇒s′) | no p | _ = no (p ∘ proj₁ ∘ arrEq)
```

```
int ≃ (t⇒t′) = no (λ ())
(t⇒t′) ≃ int = no (λ ())
```

The untyped terms are either constants (natural numbers), addition, variables (we explain after the whole program our encoding of variables), abstractions (we use a bold lambda to distinguish the term constructor from the internal abstraction operator of Agda), and applications.

```
data Nat : Set where
  zero : Nat
  suc : Nat → Nat
data Tm : Set where
  nat : Nat → Tm
  _+_ : Tm → Tm → Tm
  V : Tm
  ↑ : Tm → Tm
  λ_._ : Ty → Tm → Tm
  _·_ : Tm → Tm → Tm
```

The typing relation depends on a context assigning types to free variables; each constructors of the type `_⊢_:_` corresponds to a typing rule. We point out that the typing rules are too strong; for example, we cannot type the weakening of closed terms in the empty context. We accept this artificiality for the sake of simplicity of the algorithm.

```
data Ctx : Set where
  ⟨⟩ : Ctx
  _◁_ : Ctx → Ty → Ctx
data _⊢_:_ : Ctx → Tm → Ty → Set where
  Con : forall {c n} → c⊢nat n:int
  Add : forall {c e e′} → c⊢e:int → c⊢e′:int → c⊢e + e′:int
  Var : forall {c t} → (c◁t)⊢V:t
  Weak : forall {c t t′ e} → c⊢e:t → (c◁t′)⊢↑e:t
  Abs : forall {c t t′ e} → (c◁t)⊢e:t′ → c⊢λ t.e:t⇒t′
  App : forall {c t t′ e e′} → c⊢e:t⇒t′ → c⊢e′:t → c⊢e·e′:t′
_⊢_ : Ctx → Tm → Set
c ⊢ e = ∃ (λ t → c⊢e:t)
_⊬_ : Ctx → Tm → Set
c ⊬ e = ¬(c ⊢ e)
```

In this part we prove several properties for this presentation of STT: uniqueness of typing, some inversion properties, and some properties showing the impossibility of assigning any type to some terms (under particular contexts).

```
uniqTy : (c : Ctx) → (e : Tm) → (t t′ : Ty) → (c⊢e:t) →
  (c⊢e:t′) → t ≡ t′
uniqTy c (nat k) .int .int Con Con = refl
uniqTy c (e + e′) .int .int (Add _ _) (Add _ _) = refl
uniqTy .(c◁t) V .t t (Var {c}) Var = refl
uniqTy .(c◁t″) (↑e) t t′ (Weak {c} {.t} {t″} jdg) (Weak jdg′)
```

```
    = uniqTy c e t t′ jdg jdg′
uniqTy c (λ t.e) .(t⇒t′) .(t⇒t″) (Abs {.c} {.t} {t′} jdg)
  (Abs {.c} {.t} {t″} jdg′)
  with uniqTy (c◁t) e t′ t″ jdg jdg′
uniqTy _ (λ s._) .(s⇒t′) .(s⇒t′) (Abs jdg) (Abs jdg′)
  | refl = refl
uniqTy c (e·e′) t t′ (App {.c} {s} {.t} jf ja)
  (App {.c} {s′} {.t′} jf′ ja′)
  with uniqTy c e (s⇒t) (s′⇒t′) jf jf′
uniqTy c (e·e′) t .t (App {.c} {s} {.t} jf ja)
  (App {.c} {.s} {.t} jf′ ja′)
  | refl = refl
```

The first two results prove that both variables and terms applied to the shifting operator cannot be typed in the empty context. Then we have the inversion lemmas for typing of weakenings and abstractions.

```
varEmptyCtx : ⟨⟩ ⊬ V
varEmptyCtx (t, ())

weakEmptyCtx : (e : Tm) → (⟨⟩ ⊬ ↑e)
weakEmptyCtx e (t, ())

weakTy : (c : Ctx) → (t : Ty) → (e : Tm) → (c◁t) ⊢ (↑e) → c ⊢ e
weakTy c t e (t′, Weak je) = t′, je

absTy : (c : Ctx) → (t : Ty) → (e : Tm) → (c ⊢ (λ t.e)) →
  ((c◁t) ⊢ e)
absTy c t e (.(t⇒t′), Abs {.c} {.t} {t′} je) = t′, je
```

The following lemmata state properties about the typing of sums.

```
invAdd : (c : Ctx) → (e e′ : Tm) → (c ⊢ (e + e′)) →
  (c⊢e:int) (c⊢e′:int)
invAdd c e e′ (int, Add je je′) = je, je′
invAdd c e e′ (t⇒t′, ())

addFunL : (c : Ctx) → (t t′ : Ty) → (e e′ : Tm) → (c⊢e:t⇒t′) →
  c ⊬ (e + e′)
addFunL c t t′ e e′ jdg (.int, (Add je je′))
  with uniqTy c e (t⇒t′) int jdg je
addFunL jdg e′ _ _ _ _ (.int, (Add je je′)) | ()

addFunR : (c : Ctx) → (t t′ : Ty) → (e e′ : Tm) → (c⊢e:t⇒t′) →
  c ⊬ (e′ + e)
addFunR c t t′ e e′ jdg (.int, (Add je je′))
  with uniqTy c e (t⇒t′) int jdg je′
addFunR jdg e′ _ _ _ _ (.int, (Add je je′)) | ()
```

This last group of properties deal with typing judgements of applications.

```
invApp : (c : Ctx) → (e e′ : Tm) → (c ⊢ (e·e′)) →
  ∃₂ (λ t t′ → c⊢e:t⇒t′ c⊢e′:t)
invApp .c .e .e′ (.t′, App {c} {t} {t′} {e} {e′} je je′)
  = t, t′, je, je′
```

```
appFunTy : (c : Ctx) → (e e′ : Tm) → (c ⊢ (e·e′)) → (c ⊢ e)
appFunTy c e e′ japp with invApp c e e′ japp
... | t, t′, je, _ = t⇒t′, je
appArgTy : (c : Ctx) → (e e′ : Tm) → (c ⊢ (e·e′)) → (c ⊢ e′)
appArgTy c e e′ japp with invApp c e e′ japp
... | t, _, _, je′ = t, je′
appFunInt : (c : Ctx) → (e : Tm) → (c⊢e:int) → (e′ : Tm) →
  c ⊬ (e·e′)
appFunInt c e jint e′ (t, (App {.c}{s}{.t} y y′))
  with uniqTy c e int (s⇒t) jint y
appFunInt t _ t′ _ (_, (App y y′)) | ()
appArgOther : (c : Ctx) → (e e′ : Tm) → (t t′ s : Ty) →
  (c⊢e:t⇒t′) → (c⊢e′:s) → ¬(t ≡ s) → c ⊬ (e·e′)
appArgOther c e e′ t t′ s jdge jdge′ neq (s″, (App {.c}{r}{.s″} y y′))
  with uniqTy c e (t⇒t′) (r⇒s″) jdge y | uniqTy c e′ s r jdge′ y′
appArgOther c e e′ .s .s″ s jdge jdge′ neq (s″, (App y y′))
  | refl | refl = neq refl
appEqTy : (c : Ctx) → (t s t′ : Ty) → (e e′ : Tm) → (c⊢e:t⇒t′) →
  (c⊢e′:s) → t ≡ s → c⊢e·e′:t′
appEqTy c .s s t′ e e′ ce ce′ refl = App ce ce′
```

The previous results are useful to program the function to infer types; note the strong type of this function: given a context and a term we either infer a type, in that case we also build a derivation showing that the term can be typed in that context, or we prove that there is no type assignable to the term.

```
tyInf : (c : Ctx) → (e : Tm) → (c ⊬ e) ⊎ ∃ (λ t → c⊢e:t)
tyInf c (nat n) = inj₂ (int, Con)
tyInf c (e + e′) with tyInf c e | tyInf c e′
... | inj₁ er | _ = inj₁ (λ ih → (magic ∘ er) (int, proj₁
  (invAdd c e e′ ih)))
... | _ | inj₁ er = inj₁ (λ ih → (magic ∘ er) (int, proj₂
  (invAdd c e e′ ih)))
tyInf c (e + e′) | inj₂ (t⇒t′, je) | _ = inj₁ (addFunL c t t′ e e′ je)
tyInf c (e + e′) | _ | inj₂ (t⇒t′, je′) = inj₁ (addFunR c t t′ e′ e je′)
... | inj₂ (int, p) | inj₂ (int, p′) = inj₂ (int, Add p p′)
tyInf ⟨⟩ V = inj₁ varEmptyCtx
tyInf (c◁t) V = inj₂ (t, Var)
tyInf ⟨⟩ (↑e) = inj₁ (weakEmptyCtx e)
tyInf (c◁t) (↑e) with tyInf c e
... | inj₁ er = inj₁ (λ ih → (magic ∘ er) (weakTy c t e ih))
... | inj₂ (t′, prf) = inj₂ (t′, Weak prf)
tyInf c (λ t.e) with (tyInf (c◁t) e)
... | inj₁ er = inj₁ (λ ih → (magic ∘ er) (absTy c t e ih))
... | inj₂ (t′, prf) = inj₂ (t⇒t′, Abs prf)
tyInf c (e·e′) with tyInf c e | tyInf c e′
... | inj₁ er | _ = inj₁ (λ ih → (magic ∘ er) (appFunTy c e e′ ih))
... | inj₂ _ | inj₁ er = inj₁ (λ ih → (magic ∘ er) (appArgTy c e e′ ih))
... | inj₂ (s, ie) | inj₂ (s′, ie′) with s
... | int = inj₁ (appFunInt c e ie e′)
```

```
... | t⇒t′ with t ≃ s′
... | no neq = inj₁ (appArgOther c e e′ t t′ s′ ie ie′ neq)
... | yes eq = inj₂ (t′, appEqTy c t s′ t′ e e′ ie ie′ eq)
```

Let us explain briefly the variation of the calculus with respect to that of Sec. 1.1. Rule (α) in Def. 1 says that names of bound variables are irrelevant. Explicit use of (α) becomes tedious and it is customary to avoid it by using some convention over the terms – for example, identify α-equivalent terms or assume all bound variables to be different. De Bruijn [31] was the first to notice that what is relevant is the "distance", measured by how many binders we should cross over by going upwards in the abstract syntax tree, between an occurence of a bound variable and its binder. He replaced the set Var of variables by natural numbers; if a natural number is greater than the distance to the outer-most binder, then it represents a free variable. In our case, free variables should be thought as assumptions referring to some hypotheses in the context. Let us consider some examples showing how to translate to de Bruijn notation. The combinator $K = \lambda x.\lambda y.x$ becomes $\lambda\lambda 1$, and combinator $S = \lambda x.\lambda y.\lambda z.(x\,z)\,(y\,z)$, $\lambda(\lambda(\lambda(2\,0)\,(1\,0)))$. If we consider a term with free variables as in $\lambda x.f\,x$ we can use any index greater than 0 for f: $\lambda 1\,0$.

Some of the subtleties of substitution become more apparent with indices; for instance to avoid the capture of free variables in r when substituting r inside an abstraction one needs to *shift* the free variables in r; that is, if n is an occurrence of a free variable in r, it should become $n+1$ under the abstraction. On the other hand, this shifting should not act on bound-variables. This informal considerations can be internalised in the formal system by adding a new grammatical category for *explicit substitutions*.[3] In that case, the shifting operator belongs to the category of substitutions; in the Agda formalisation, shifting is $\uparrow$. Note that we use an unary representation of natural numbers, V denotes 0 and $\uparrow$ denotes Suc. In the rest of the thesis, 0 is denoted by q and Suc by p.

1.3 Type-Checking: Deciding the Typing Relation

In the previous section we showed an algorithm for type-checking simple typed lambda calculus. The algorithm proceeds by recursion on the structure of the term, for example to type-check an application, it infers the type of the term in the function place and if the inferred type was of the form $s \to t$, then it checks that the argument has type s. When we try to apply the same idea for dependent types, we rapidly discover a problem: the argument can have a syntactically different type s', but convertible to the expected type s; thus the rule for application can still be used, but only after using (conv) on the argument. We came across a similar situation when proving *EvInd EvZ*: *Even* $(1+1)$. So to decide type-checking we need to decide equality between types; and since types can depend on terms, we also need to decide equality between terms.

To check if two terms, or types, are equal, we may put them in normal form and compare if they are syntactically equal. This method yields a decision

[3] We refer to [1, 72, 96, 112] for further information on explicit substitutions.

procedure for equality, provided that equal terms have the same normal form and each term is provably equal to its normal form. Sometimes it is possible to obtain a normalisation algorithm by orienting the equality rules; if the resulting rewrite system has some properties (confluence and termination), then it can be used to decide equality. This approach of orienting the axioms cannot be applied for the type of an abelian monoid:

$$\frac{}{\mathit{Mon}\quad \mathit{type}} \qquad \frac{}{0 : \mathit{Mon}} \qquad \frac{a : \mathit{Mon} \qquad b : \mathit{Mon}}{a * b : \mathit{Mon}}$$

with axioms

$$0 * a = a \qquad a * b = b * a$$

It is clear that in whatever direction we orient the second axiom, the rewriting rule could be used indefinitely; thus making the rewriting system nonterminating. In the next section we introduce another normalisation algorithm that can be used to handle equality rules as these.

Normalisation By Evaluation: Using Semantics to Normalise

In order to explain the normalisation algorithms defined in this thesis, we need to introduce the notion of formal semantics, in particular of denotational semantics. Remember that one of the points of formal systems is to devoid of any meaning its constructions. Denotational semantics provides a mathematical intrepretation, called a *model*, to the syntactical elements of a formal system. In the case of STT, for example, there are two kinds of syntactical elements: types and terms. A *standard model* for STT, as defined by Henkin [66], is one where o is interpreted as the set $D_o = \{T,F\}$ of truth values, ι is interpreted by some arbitrary set D_ι, and $A \to B$, as the set of all the (total) functions from D_A to D_B. Terms are interpreted as elements of those sets. Given an interpretation of the syntax — we use $[\![a]\!]$ to denote the interpretation of a — the definition of a model should also say when a judgement is valid under the interpretation. In the standard model, a judgement with conclusion $a : A$ is valid if $[\![a]\!] \in [\![A]\!]$. An equality judgement $a = b : A$ is valid if $[\![a]\!] = [\![b]\!]$ and $[\![a]\!] \in [\![A]\!]$.

Berger and Schwichtenberg [26] noticed that one can set up a certain model from where normal forms can be extracted. This technique, called Normalisation by Evaluation (NbE), uses semantical rather than syntactical concepts, as in more traditional methods where one speaks of reduction sequences and the like. The idea of NbE is to build a model such that one can go back from the semantics to the syntax; i.e. one not only has the evaluation function $[\![\_]\!]$ but also a *reification* function $R(\_) \in \bigcup_A D_A \to \Lambda$, see Fig. 1.1. If this reification function maps elements in the image of $[\![\_]\!]$ to terms in normal form, then we can compose the two functions and get a normal form for each term. The reification function will be useful if we can prove that the composition of the two functions maps terms to their normal form; that is, we need a proof that $t = R([\![t]\!])$ is provable in the formal system. In order to use NbE for deciding the equality in a theory we need another property: if $t = t'$ is provable, then we need to know that $R([\![t]\!]) \equiv R([\![t']\!])$, where $\equiv$ denotes syntactic equivalence.

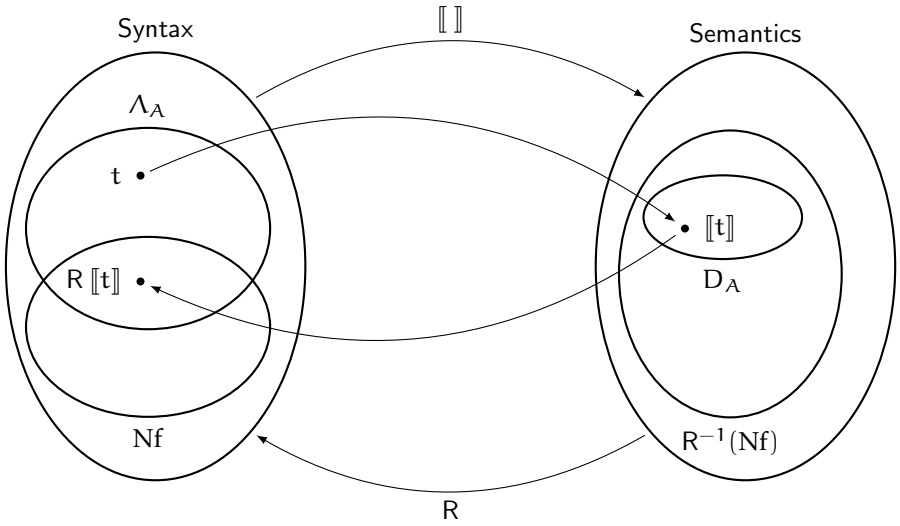


Figure 1.1: Normalisation by Evaluation

1.4 Related work

In this section I revise the state-of-the-art in the main themes of this thesis; in the next section I highlight the contribution of each chapter with respect to the literature mentioned here.

The technique of NbE for STT was introduced by Berger and Schwichtenberg [26]; but a model of normal forms has been already considered for a dependent type systems (but without conversion under abstractions) in [80], where some combinatorial results are proved by reasoning over the model. As far as I know, most of the literature of NbE for STT [11, 12, 13, 25, 39, 54, 56], in various presentations and under different points of view, considered $\beta\eta$-long-normal forms; one exception is [10] where the normalisation algorithm does not η-expand terms.

The first paper on NbE for Martin-Löf type theory (MLTT) is [2], the calculus considered here has an untyped notion of conversion. NbE for MLTT with equality under types is considered in [3]. These papers are followed by [5], which presents an incremental type-checking algorithm based on [37] and uses ideas coming from the previous two works to prove the correctness and completeness of the decision procedure for equality. Another related work is [4] which uses hereditary substitution to decide equality and presents a type-checking algorithm for MLTT without (η). A proof of the correctness and completeness of type-checking for a weaker logical framework is in [64]. In [108] the type-checking algorithm is extended to a logical framework with singleton types and subtyping. Pollack [97] developed a formal proof of the decidability of type-checking for ECC [77]. Barras [18] implemented in Coq a correct and complete type-checking algorithm for the Calculus of Constructions; later Barras [19] considered stonger type systems closer to the actual implementation of Coq.

In the previous paragraph we mentioned two presentations of MLTT: with typed-equality (two terms are equal *under a type*) and with equality generated by a reduction relation on untyped-terms. Geuvers [59] conjectured that both presentations are equivalent for PTSs. For PTSs without (η), Adams [9] proved the equivalence for functional PTSs. Later, Herbelin and Siles [105] extended and formalised Adams' result to semi-full systems; Siles finally settled in his PhD thesis [104] the equivalence for every PTS.

1.5 Our contributions

In this section I briefly comment on the contributions of the following chapters. Most of them are based on work that I have done in collaboration with other people; for each chapter I refer to the relevant papers and also make explicit my personal contributions to them. Let us invoke the *preface paradox*, any mistake or oversight of this thesis is my own responsibility.

A common trait of all the chapters is a NbE algorithm, suggested by Thierry Coquand in the context of dependent type systems. The NbE algorithm is first presented for STT in Chap. 2; we use STT to explain the technical issues to be solved in NbE and how the algorithm handles them. One novelty of this NbE algorithm is that it separates the phases of reification from that of η-expansion; thus the same architecture can be used for calculi with and without (η). All the chapters are organised in a similar way as that of Chap. 2: first we introduce the formal system. Then we introduce a model suitable for the definition of NbE. Correctness of the NbE algorithm is based on logical relations and deserves a separate section. The normalisation algorithm is used in the last section of each chapter.

In Chapter 3 we define NbE for Martin-Löf type theory with a universe and dependent function spaces without rule (η). From the proof of the correctness of the normalisation function we prove the injectivity of the constructor of dependent functions spaces; as far as we know, that result has not been proved by elementary syntactical arguments. This chapter is based on a paper in preparation, co-authored with Thierry Coquand and Daniel Fridlender; this work is based on the papers mentioned in the next paragraph. My contribution to the paper is the adaptation of the NbE algorithm for a calculus without (η).

Chapter 4 is an adapted version of joint-work with Andreas Abel and Thierry Coquand. In this work we considered Martin-Löf type theory extended, among others, with singleton types and proof-irrelevant types. This was first presented in TLCA [6] and later published in a journal version [7]; these articles benefited and can be considered as follow-ups of [2, 3, 5]. My contribution to that work consisted in carrying on the proofs; in particular, I came up with the method of proving completeness for the type-checker and the conservativity of MLTT with a canonical element for proof-irrelevant types. The emphasis of this chapter is on the use of NbE to define a type-checking algorithm; as far as I know, there is no proof of correctness and completeness for MLTT.

In Chapter 5 we define NbE for a class of Pure Type Systems; this chapter is based on joint-work with Daniel Fridlender. As in Chap. 3, we prove the injectivity of the constructor of dependent function spaces for *predicative* PTSs; the model construction also leads to a proof of equivalence between systems

with typed equality and untyped conversion for predicative PTSs. As I said, this result has been already proved for every PTS by Siles [104]; the contribution is the proof method; which can be useful to study the equivalence for some PTSs with (η). The particular presentation of PTSs with explicit substitutions of this chapter is a novelty in itself and we formalised some of their meta-theory. A preliminary report of the results on this chapter was presented at Types-2011 [57]. My contribution to the results on this chapter are the formalisation of the equivalence proof in Agda and the adaptation of the proof method of correctness of NbE.

In Chapter 6 I show that the NbE algorithm can be adapted for theories involving axioms like commutativity. Moreover I suggest some more applications of NbE for dependent type systems.

Normalisation By Evaluation

2

This chapter is a technical introduction to NbE and the methods used in the proofs of completeness and correctness of NbE. We study them in the more elementary setting of STT. We first present the formal system of STT, denoted by $\lambda^{\to}$. Then we present and explain NbE for λ^{β}, a variant of $\lambda^{\to}$ without (η); we skip the proofs of completeness and correctness for λ^{β}. Later we adapt that algorithm for $\lambda^{\to}$, for which we give full proofs of completeness and correctness of NbE.

2.1 Syntax of $\lambda^{\to}$

The formal system of $\lambda^{\to}$ is presented as a *generalised algebraic theory*. This notion, introduced by Cartmell [34], extends many-sorted logic in that it includes dependent sorts; semantically, this kind of sorts may denote, for instance, family of sets (contrast this with the situation of many-sorted logic where each sort denotes a fixed set). GATs are an abstraction of Martin-Löf type theory, and can be understood also as the logical framework used to define our formal systems.

A GAT is defined by prescribing *sorts* and *operators*, rules for the introduction of sorts, introductory rules for operators, and axioms postulating the equality of sort, or operator, expressions. We explain some technicalities of GATs as we encounter them.

Our particular presentation of $\lambda^{\to}$ is close to that of categorical combinators of Curien et al. [1, 45, 47]. The theory of $\lambda^{\to}$ has four sorts: contexts, types, substitutions and terms; their introductory rules are shown in Fig. 2.1. From the rules we can observe that Ctx, the sort for well-formed contexts, and Type are constant sorts, whereas $\Gamma \to \Delta$, the sort for substitutions, depends on contexts, and $\mathsf{Term}(\Gamma, A)$, on contexts and types.

We use the following conventions: capital Greek letters denote metavariables ranging over contexts, small Greek letters are for substitutions, uppercase Latin letters stand for types, and minuscules for terms. Constants are printed in sans. While we are discussing notation, it is in place to contrast the

$$\frac{}{\mathsf{Ctx}\ \text{is a sort}}\ (\textsc{ctx-sort}) \qquad \frac{\Gamma, \Delta \in \mathsf{Ctx}}{\Gamma \to \Delta\ \text{is a sort}}\ (\textsc{subs-sort})$$

$$\frac{}{\mathsf{Type}\ \text{is a sort}}\ (\textsc{type-sort}) \qquad \frac{\Gamma \in \mathsf{Ctx} \qquad A \in \mathsf{Type}}{\mathsf{Term}(\Gamma, A)\ \text{is a sort}}\ (\textsc{term-sort})$$

Figure 2.1: Sort introductory rules

format of the typing rules given below with the more traditional ones. It is customary in type theory to differentiate the various forms of judgement by the way they are written; for example, the judgement for Γ being a well-formed context is written $\Gamma \vdash$ and to have a proof of "t being of type A under context Γ" is printed as $\Gamma \vdash t : A$. In GAT both judgements are represented uniformly as $\Gamma \in \mathsf{Ctx}$ and $t \in \mathsf{Term}(\Gamma, A)$. Setting aside equality, the other form of judgements that we will encounter are $\delta \in \Gamma \to \Delta$ for substitutions, and $A \in \mathsf{Type}$ for types; in type theory they are written as $\Gamma \vdash \delta : \Delta$ and $\Gamma \vdash A$, respectively. From this parallel it is easy to go from one presentation style to the other. We use the GAT style for the introductory rules of $\lambda^{\to}$; in the rest of the chapter we present examples, axioms, and judgements in the more familiar style of type theory.

Introductory rules

In this section we introduce the rules for $\lambda^{\to}$ in small steps; rules are grouped by the form of judgement of their conclusion. We explain the rôle of each operator after its introductory rule.

Contexts A context corresponds to a list of assumptions. The empty context is denoted by $\diamond$; sometimes we write $\vdash t : A$, instead of $\diamond \vdash t : A$. If we have made some assumptions, say Γ, then we can make some more; since types act as assumptions, we can extend Γ with a type A: this context is written $\Gamma.A$.

$$\frac{}{\diamond \in \mathsf{Ctx}}\ (\textsc{empty-ctx}) \qquad \frac{\Gamma \in \mathsf{Ctx} \qquad A \in \mathsf{Type}}{\Gamma.A \in \mathsf{Ctx}}\ (\textsc{ext-ctx})$$

Usually when one adds assumption A to the context Γ (read "Γ is extended with type A"), this new assumption is named. Of course this name should be *fresh* with respect to names of other assumptions in Γ. In our calculus there is no need to name assumptions because they are referred by de Bruijn indices.

Substitutions The introductory rules of operators for substitutions are shown in Fig. 2.2. A substitution $\sigma \in \Gamma \to \Delta$ can be understood as assigning well-typed terms under Γ to assumptions in Δ. Another possible reading, coming from the categorical origin of explicit substitutions [44], is that a substitution $\sigma \in \Gamma \to \Delta$ is a morphism in the category of contexts (which also explains that σ can be thought as a mapping $\mathsf{Term}(\Delta, A) \to \mathsf{Term}(\Gamma, A)$).

The operators can be easily understood using the first reading: the identity substitution id_Γ maps every variable to itself. The empty substitution $\langle\rangle$ should map no variable to anything. Composition is writen as juxta-position; the extension operator, noted by pairing, makes patent that substitutions assigns well-typed terms under one context to variables in another. Finally, p is the shifting operation, also called weakening substitution, needed when extending a context with a new assumption.

In the introductory rules for substitutions we note an asymmetry between the rule for the identity substitution and the rule for the shifting and empty substitutions: whereas the identity operator is applied to a context, but the

$$\frac{\Gamma \in \mathsf{Ctx}}{\mathsf{id}_\Gamma \in \Gamma \to \Gamma}\ (\textsc{id-subs}) \qquad \frac{\Gamma \in \mathsf{Ctx}}{\langle\rangle \in \Gamma \to \diamond}\ (\textsc{empty-subs}) \qquad \frac{\Gamma \in \mathsf{Ctx} \qquad A \in \mathsf{Type}}{\mathsf{p} \in \Gamma.A \to \Gamma}\ (\textsc{fst-subs})$$

$$\frac{\Gamma, \Delta, \Sigma \in \mathsf{Ctx} \qquad \delta \in \Delta \to \Sigma \qquad \sigma \in \Sigma \to \Gamma}{\sigma\,\delta \in \Delta \to \Gamma}\ (\textsc{comp-subs})$$

$$\frac{\Gamma, \Delta \in \mathsf{Ctx} \qquad \sigma \in \Delta \to \Gamma \qquad A \in \mathsf{Type} \qquad t \in \mathsf{Term}(\Delta, A)}{(\sigma, t) \in \Delta \to \Gamma.A}\ (\textsc{ext-subs})$$

Figure 2.2: Introductory rules for substitutions

operators $\langle\rangle$ and p are not. In the formal syntax of GATs every "polymorphic" operator should be fully applied; in our case, the conclusions should read $\langle\rangle_\Gamma \in \Gamma \to \diamond$ and $\mathsf{p}(\Gamma, A) \in \Gamma.A \to \Gamma$, respectively. In the following, we will present polymorphic operators without applying all the arguments to make them monomorphic. A further informality that we allow ourselves in the presentation of introductory rules is the omission of some premises that can be inferred from premisses depending on them or from the conclusion.

Types We consider only a basic type N and the formation of non-dependent function spaces, $A \to B$.

$$\frac{}{\mathsf{N} \in \mathsf{Type}}\ (\textsc{iota-I}) \qquad \frac{A \in \mathsf{Type} \quad B \in \mathsf{Type}}{A \to B \in \mathsf{Type}}\ (\textsc{Fun-I})$$

Terms The introductory rules for terms are presented in Fig. 2.3. Note the rule (SUBS-TERM) for applying substitutions to terms which was not present in Def. 1.2; in this rule t is a term with variables under Δ and σ assigns terms, typed under Γ, to those variables; thus after applying σ to t we have a well-typed term under Γ.

As we explained at the end of Sec. 1.2, we use a variant of unary de Bruijn indices: q corresponds to 0 and the succesor of n is obtained by applying the substitution p to n; e.g. the penultimate assumption is referred by $\mathsf{q}\,\mathsf{p}$, the previous one by $(\mathsf{q}\,\mathsf{p})\,\mathsf{p}$, and so on. The following example illustrates our encoding of variables.

Example 1.

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\mathsf{N} \to \mathsf{N}.\mathsf{N} \vdash \mathsf{q} : \mathsf{N}} \qquad \dfrac{\vdots}{\mathsf{N} \to \mathsf{N}.\mathsf{N} \vdash \mathsf{q}\,\mathsf{p} : \mathsf{N} \to \mathsf{N}}}{\mathsf{N} \to \mathsf{N}.\mathsf{N} \vdash \mathsf{App}\,(\mathsf{q}\,\mathsf{p})\,\mathsf{q} : \mathsf{N}}}{\mathsf{N} \to \mathsf{N} \vdash \lambda(\mathsf{App}\,(\mathsf{q}\,\mathsf{p})\,\mathsf{q}) : \mathsf{N} \to \mathsf{N}}}{\vdash \lambda\lambda(\mathsf{App}\,(\mathsf{q}\,\mathsf{p})\,\mathsf{q}) : (\mathsf{N} \to \mathsf{N}) \to \mathsf{N} \to \mathsf{N}}$$

$$\frac{\Gamma \in \mathsf{Ctx} \qquad A \in \mathsf{Type}}{q \in \mathsf{Term}(\Gamma.A, A)}\ (\textsc{hyp}) \qquad \frac{\Gamma \in \mathsf{Ctx} \qquad A \in \mathsf{Type} \qquad t \in \mathsf{Term}(\Gamma.A, B)}{\lambda t \in \mathsf{Term}(\Gamma, A \to B)}\ (\textsc{Abs-I})$$

$$\frac{\Gamma \in \mathsf{Ctx} \qquad A, B \in \mathsf{Type} \qquad t \in \mathsf{Term}(\Gamma, A \to B) \qquad r \in \mathsf{Term}(\Gamma, A)}{\mathsf{App}\ t\ r \in \mathsf{Term}(\Gamma, B)}\ (\textsc{App-I})$$

$$\frac{\Gamma, \Delta \in \mathsf{Ctx} \qquad A \in \mathsf{Type} \qquad \sigma \in \Gamma \to \Delta \qquad t \in \mathsf{Term}(\Delta, A)}{t\sigma \in \mathsf{Term}(\Gamma, A)}\ (\textsc{subs-term})$$

Figure 2.3: Introductory rules for terms

Axioms The second data of a GAT are its equality rules between sort expressions and between expressions having the same sort. In the case of STT there is no axiom for sort expressions and there are only rules for substitutions and terms. Although we show the rules making the equality an equivalence relation and a congruence with respect to operators, these *derived rules* do not need to be added as axioms, because they are part of the logical framework.

Equalities on terms The first set of axioms correspond to rules (β) and (η) of Def. 1. Note that each axiom has a type and one can prove that both sides of the equation have that type. For example, in (β) the body t of the abstraction λt has type B under the context Γ.A; by extending id_Γ with r and applying the resulting substitution to t, we map the first free variable of t to r. In contrast with implicit substitutions, $t\,(\mathsf{id}_\Gamma, r)$ is not the result of substituting all the occurences of the first free variable of t by r; that result can still be obtained by using axioms to resolve substitutions applied to terms and composition of substitutions.

$$\frac{\Gamma.A \vdash t : B \qquad \Gamma \vdash r : A}{\Gamma \vdash \mathsf{App}\ (\lambda t)\ r = t\,(\mathsf{id}_\Gamma, r) : B}\ (\textsc{beta}) \qquad \frac{\Gamma \vdash t : A \to B}{\Gamma \vdash \lambda(\mathsf{App}\ (t\,p)\ q) = t : A \to B}\ (\textsc{eta})$$

In rule (η) the weakening substitution is explictly applied to term t, thus none of its free variables are bound by the binder.

Substitutions on terms The following rules axiomatise substitution; if we read these axioms as rewrite rules, we can observe that they reduce redices involving substitutions applied to terms. Keeping in mind our explanation of substitutions, we can see that (snd-sub) corresponds to the replacement of the first free variable for the second component. Consider, for example, an application $\Gamma \vdash \mathsf{App}\ (\lambda q)\ t : A$, we can first reduce the β-redex to get $\mathsf{App}\ (\lambda q)\ t = q\,(\mathsf{id}_\Gamma, t)$ and then apply (snd-sub) to get $\mathsf{App}\ (\lambda q)\ t = t$. In (abs-sub) we realise the need of extending the substitution, while σ assigns terms under Γ, inside the binder we should have a substitution assigning terms under Γ.A; note also that by composing σ with p we lift all free variables in σ

to avoid them being captured.

$$\textsc{(sub-ass)}\ \frac{\Sigma \vdash t : A \qquad \Sigma \vdash \sigma : \Delta \qquad \Gamma \vdash \delta : \Delta}{\Gamma \vdash t\,(\sigma\,\delta) = (t\,\sigma)\,\delta : A} \qquad \textsc{(sub-id)}\ \frac{\Gamma \vdash t : A}{\Gamma \vdash t\,\mathsf{id}_\Gamma = t : A}$$

$$\textsc{(snd-sub)}\ \frac{\Gamma \vdash t : A \qquad \Gamma \vdash \sigma : \Delta}{\Gamma \vdash \mathsf{q}\,(\sigma, t) = t : A} \qquad \textsc{(abs-sub)}\ \frac{\Delta \vdash \lambda t : A \to B \qquad \Gamma \vdash \sigma : \Delta}{\Gamma \vdash (\lambda t)\,\sigma = \lambda(t\,(\sigma\,\mathsf{p}, \mathsf{q})) : A \to B}$$

$$\textsc{(app-sub)}\ \frac{\Delta \vdash t : A \to B \qquad \Delta \vdash r : A \qquad \Gamma \vdash \sigma : \Delta}{\Gamma \vdash (\mathsf{App}\ t\ r)\,\sigma = \mathsf{App}\ (t\,\sigma)\ (r\,\sigma) : B}$$

Substitutions The following rules can be understood as the equational theory of a category with finite products: associativity of composition, identity is the neutral element of composition, the universal property for the terminal object, and properties for binary products: post-composition with the first projection, identity for products, and post-composition with a mediating morphism to a product object.

$$\textsc{(sub-ass)}\ \frac{\Theta \vdash \sigma : \Sigma \qquad \Sigma \vdash \delta : \Delta \qquad \Gamma \vdash \gamma : \Delta}{\Gamma \vdash (\sigma\,\delta)\,\gamma = \sigma\,(\delta\,\gamma) : \Theta} \qquad \textsc{(sub-empty)}\ \frac{\Gamma \vdash \sigma : \diamond}{\Gamma \vdash \langle\rangle\,\sigma = \langle\rangle : \diamond}$$

$$\textsc{(sub-idl)}\ \frac{\Gamma \vdash \sigma : \Delta}{\Gamma \vdash \mathsf{id}_\Gamma\,\sigma = \sigma : \Delta} \qquad \textsc{(sub-idr)}\ \frac{\Gamma \vdash \sigma : \Delta}{\Gamma \vdash \sigma\,\mathsf{id}_\Gamma = \sigma : \Delta}$$

$$\textsc{(sub-id-empty)}\ \frac{}{\diamond \vdash \mathsf{id}_\diamond = \langle\rangle : \diamond} \qquad \textsc{(sub-id-ext)}\ \frac{}{\Gamma.A \vdash \mathsf{id}_{\Gamma.A} = (\mathsf{p}, \mathsf{q}) : \Gamma.A}$$

$$\textsc{(sub-fst)}\ \frac{\Gamma \vdash t : A \qquad \Gamma \vdash \sigma : \Delta}{\Gamma \vdash \mathsf{p}\,(\sigma, t) = \sigma : \Delta} \qquad \textsc{(sub-map)}\ \frac{\Gamma \vdash t : A \qquad \Gamma \vdash \sigma : \Delta \qquad \Sigma \vdash \delta : \Delta}{\Gamma \vdash (\sigma, t)\,\delta = (\sigma\,\delta, t\,\delta) : \Sigma.A}$$

Congruence The last two groups of rules correspond to reflexivity, symmetry, transitivity, and contextual clousure of equality.

$$\textsc{(refl)}\ \frac{\Gamma \vdash t : A}{\Gamma \vdash t = t : A} \qquad \textsc{(refl-subs)}\ \frac{\Gamma \vdash \sigma : \Delta}{\Gamma \vdash \sigma = \sigma : \Delta}$$

$$\textsc{(sym)}\ \frac{\Gamma \vdash t = r : A}{\Gamma \vdash r = t : A} \qquad \textsc{(sym-subs)}\ \frac{\Gamma \vdash \sigma = \sigma' : \Delta}{\Gamma \vdash \sigma' = \sigma : \Delta}$$

$$\textsc{(trans)}\ \frac{\Gamma \vdash t = r : A \qquad \Gamma \vdash r = s : A}{\Gamma \vdash t = s : A} \qquad \textsc{(trans-sub)}\ \frac{\Gamma \vdash \sigma = \delta : \Delta \qquad \Gamma \vdash \delta = \gamma : \Delta}{\Gamma \vdash \sigma = \gamma : \Delta}$$

$$\frac{\Gamma \vdash t = t' : A \qquad \Gamma \vdash r = r' : A}{\Gamma \vdash \mathsf{App}\, t\, r = \mathsf{App}\, t'\, r' : A}\ (\textsc{cong-app}) \qquad \frac{\Gamma.A \vdash t = t' : B}{\Gamma \vdash \lambda t = \lambda t' : A \to B}\ (\textsc{cong-abs})$$

$$\frac{\Delta \vdash t = t' : A \qquad \Gamma \vdash \sigma = \sigma' : \Delta}{\Gamma \vdash t\,\sigma = t'\,\sigma' : A}\ (\textsc{cong-subs})$$

$$\frac{\Gamma \vdash t = t' : A \qquad \Gamma \vdash \sigma = \sigma' : \Delta}{\Gamma \vdash (\sigma, t) = (\sigma', t') : \Delta.A}\ (\textsc{cong-map}) \qquad \frac{\Gamma \vdash \sigma = \sigma' : \Delta \qquad \Delta \vdash \delta = \delta' : \Sigma}{\Gamma \vdash \sigma\,\delta = \sigma'\,\delta' : \Sigma}\ (\textsc{cong-comp})$$

Properties of the formal system

In our encoding of de Bruijn variables can be identified with $q, q\,p^1, q\,p^2, \ldots$ where the notation p^i denotes the i-times composition of p with itself:

$$p^i = \begin{cases} \mathsf{id} & \text{if } i = 0 \\ p & \text{if } i = 1 \\ p\,p^{i-1} & \text{if } i > 1 \end{cases} .$$

Note that this substitution can be typed in a context Γ by extending it by successive applications of rules (comp-subs) and (fst-subs). In general we will say that $\Delta \leqslant^i \Gamma$ if $\Delta \vdash p^i : \Gamma$. So, it is clear that if $\Gamma \vdash t : A$ then for any $\Delta \leqslant^i \Gamma$ we can derive $\Delta \vdash t\,p^i : A$.

Remark 1 (Pre-order between contexts). The relation $\Delta \leqslant^i \Gamma$ defines a preorder on contexts; in fact

- $\Gamma \leqslant^0 \Gamma$, by (id-subs); and
- if $\Delta' \leqslant^j \Delta$ and $\Delta \leqslant^i \Gamma$, then $\Delta' \leqslant^{i+j} \Gamma$, by (comp-subs) and associativity of composition.

Remark 2 (Inversion of substitution). It is clear that any substitution $\Delta \vdash \sigma : \Gamma.A$ is judgmentally equal to some substitution $\Delta \vdash (\sigma', t) : \Gamma.A$:

$$\sigma = \mathsf{id}_{\Gamma.A}\,\sigma = (p, q)\,\sigma = (p\,\sigma, q\,\sigma) .$$

Finally we characterise the set of terms in normal form. The shape of normal forms, in the setting of untyped lambda calculus with named variables, is $\lambda x_1.\lambda x_2.\ldots.\lambda x_n.(\ldots((y\,t_1)\,t_2)\ldots)\,t_m$, where $m \geqslant 0$, $n \geqslant 0$, and every t_i has also that shape; it is easy to see that the following grammar captures those terms.

$$\begin{aligned} \mathrm{Ne} \ni k &::= x \mid \mathsf{App}\, k\, v \\ \mathrm{Nf} \ni v &::= \lambda x.v \mid k . \end{aligned}$$

After replacing indexed variables for named variables in that grammar we arrive at the definition of normal forms and neutral terms.

Definition 2 (Neutral terms and normal forms).

$$\begin{aligned} \mathrm{Ne} \ni k &::= q \mid q\,p^{i+1} \mid \mathsf{App}\, k\, v \\ \mathrm{Nf} \ni v &::= \lambda v \mid k \end{aligned}$$

2.2 Normalisation by Evaluation for λ^β

As we explained in Sec. 1.3, NbE is based on the idea of constructing a model from where it is possible to go back to the terms. In this section we first consider a model for λ^β and define the reification function going back to the syntax. Then we recall that models for STT are cartesian closed categories. Finally we analyse the subtleties of NbE and comment briefly on its historical development.

A suitable model for NbE

Our model for normalisation is based on a domain [8, 102, 106] coming from the solution D of the following domain equation

$$D \approx \mathbb{O} \oplus D \times D \oplus [D \to D] \oplus \mathit{Var}_\perp \oplus D \times D \ ; \tag{2.1}$$

where *Var* is a denumerable set (we write x_i and assume $x_i \neq x_j$ if $i \neq j$, for $i, j \in \mathbb{N}$), $\mathbb{O} = \{\perp, \top\}$ (called the Sierpinski's space), $[D \to D]$ is the set of continuous functions from D to D, and $D \times D$ is the cartesian product of D with itself. The set *Var* is viewed as a flat pre-domain. Every element of D which is not $\perp$, is an element of some component of the smashed sum in the right hand side of Eq. 2.1; in this case we write $\top \in D$ for $\top \in \mathbb{O}$ and

$$\begin{array}{ll} \mathsf{pair} : D \times D \to D & \mathsf{lam} : [D \to D] \to D \\ \mathsf{Var} : \mathbb{N} \to D & \mathsf{App} : D \times D \to D \ . \end{array}$$

Reification We define a partial readback function R which given an element in D returns a term of the calculus. This function is similar as the readback function introduced by Gregoire and Leroy [62] to define a normalisation procedure by means of an evaluator. When the evaluator returns an abstraction $\lambda x.t$, the readback function creates a new abstracion $\lambda y.v$, where v is the normal form that results from the evaluation and reading back the term $(\lambda x.t)\,\tilde{y}$; the constant $\tilde{y}$ is later substituted for y by the readback function. In our case, in an element of D of the form $\mathsf{lam}\,f$, the function f can be thought of as being the normaliser of the body of some abstraction, so we need only to apply f to get a value which can be reified.

Definition 3 (Reification function).

$$\begin{aligned} R_j\ (\mathsf{App}\ d\ d') &= \mathsf{App}\ (R_j\ d)\ (R_j\ d') \\ R_j\ (\mathsf{lam}\ f) &= \lambda(R_{j+1}\ (f(\mathsf{Var}\ j))) \\ R_j\ (\mathsf{Var}\ i) &= \begin{cases} q & \text{if } j \leqslant i+1 \\ q\ p^{j-(i+1)} & \text{if } j > i+1 \end{cases} \end{aligned}$$

Recall that in Fig. 1.1 we identified a subset of D by taking the inverse image of R over the set of normal forms; we also need to consider semantical neutral values. Since our R is a family of functions indexed by a natural number, we should consider all the indices:

$$\mathsf{Ne} = \bigcap_{i \in \mathbb{N}} \{d \in D \mid R_i\ d \in \mathsf{Ne}\} \quad \text{and} \quad \mathsf{Nf} = \bigcap_{i \in \mathbb{N}} \{d \in D \mid R_i\ d \in \mathsf{Nf}\} \ .$$

Notice that if the case $j < i + 1$ were undefined in the clause for variables in Def. 3, then for any $m \in \mathbb{N}$ the application $R_0\,(\mathsf{Var}\,m)$ would be undefined; hence $\mathsf{Var}\,m \notin \mathsf{Ne}$ and, consequently, Ne would be empty. Since we depend on having a semantic representation of variables and neutrals we add the case $j < i + 1$. This case will not arise in our use of the readback function.

Note that we can extend the projection functions $\pi_1, \pi_2 : D \times D \to D$ to functions over D:

$$\mathsf{p}\;d = \begin{cases} d_1 & \text{if } d = (d_1, d_2) \\ \bot & \text{otherwise} \end{cases} \qquad \mathsf{q}\;d = \begin{cases} d_2 & \text{if } d = (d_1, d_2) \\ \bot & \text{otherwise} \end{cases} \quad .$$

We also define an *application* operation on D:

$$d \cdot e = \begin{cases} \mathsf{f}\;e & \text{if } d = \mathsf{lam}\,\mathsf{f} \\ \mathsf{App}\,d\,e & \text{if } d \in \mathsf{Ne} \\ \bot & \text{otherwise} \end{cases} \quad .$$

With these functions we can construct an *environment model* for λ^β over the applicative structure $\langle D, \cdot \rangle$. The interpretation of terms and substitutions is given by mappings $[\![\_]\!]^t\_ : \mathit{Terms} \times D \to D$ and $[\![\_]\!]^s\_ : \mathit{Substs} \times D \to D$, respectively. The second argument for both functions is an environment assigning values to free variables.

$$\begin{aligned} [\![\mathsf{App}\;t\;r]\!]^t d &= [\![t]\!]^t d \cdot [\![r]\!]^t d & [\![\langle\rangle]\!]^s d &= \top \\ [\![\lambda t]\!]^t d &= \mathsf{lam}\,(e \mapsto [\![t]\!]^t(d, e)) & [\![\mathsf{id}_\Gamma]\!]^s d &= d \\ [\![t\;\sigma]\!]^t d &= [\![t]\!]^t([\![\sigma]\!]^s d) & [\![(\sigma, t)]\!]^s d &= ([\![\sigma]\!]^s d, [\![t]\!]^t d) \\ [\![\mathsf{q}]\!]^t d &= \mathsf{q}\;d & [\![\mathsf{p}]\!]^s d &= \mathsf{p}\;d \\ & & [\![\sigma\,\delta]\!]^s d &= [\![\sigma]\!]^s([\![\delta]\!]^s d) \end{aligned}$$

Our model is completed by fixing the interpretation of types $[\![\mathsf{N}]\!] = \mathsf{Ne}$ and $[\![A \to B]\!] = \{d \in D \mid d \cdot e \in [\![B]\!], \text{ for all } e \in [\![A]\!]\}$. This model can be viewed as a cartesian closed category, cf. next section; so we have soundness for free. As a corollary we obtain a normalisation function. In fact, let $\Gamma = A_1 \ldots . A_n$ and $\Gamma \vdash t : A$ we only need to specify an appropiate environment d for interpreting t. Since $\mathit{Var} \subseteq [\![A]\!]$ for every type A, we can let d be the sequence $((\ldots(\top, \mathsf{Var}\,0)\ldots), \mathsf{Var}\,n - 1)$ and define our normalisation function

$$\mathbf{nbe}_\Gamma(t) = R_n\,([\![t]\!]^t d) \quad .$$

We claim that $\mathsf{nbe}(\_)$ is a normalisation function that can be used to decide equality; but to support our claim we should prove:

1. **normalisation**: $\Gamma \vdash t : A$, then $\mathbf{nbe}_\Gamma(t) \in \mathrm{Nf}$;
2. **correctness**: if $\Gamma \vdash t : A$, then $\Gamma \vdash t = \mathbf{nbe}_\Gamma(t) : A$; and
3. **completeness**: if $\Gamma \vdash t = t' : A$, then $\mathbf{nbe}_\Gamma(t) \equiv \mathbf{nbe}_\Gamma(t')$.

These results will be proved for $\lambda^\to$ in Sec. 2.3 and 2.4. In Chap. 5 we define NbE and prove these three properties for a class of PTS, one of which is λ^β.

Cartesian closed categories

The categorical models of STT are cartesian closed categories (CCC) [70, 74]. A cartesian closed category is a category with:

1. a terminal object,
2. binary products, and
3. exponentials.

Now we give the interpretation of λ^β in this more abstract setting. The semantic function is given parametrically on the underlying category and on an object D_N, the interpretation of N.

Types and Contexts

$$\begin{aligned} [\![N]\!] &= D_N & [\![\diamond]\!] &= 1 \\ [\![A \to B]\!] &= [\![B]\!]^{[\![A]\!]} & [\![\Gamma.A]\!] &= [\![\Gamma]\!] \times [\![A]\!] \end{aligned}$$

Terms and Substitutions

$$\begin{aligned} [\![\mathsf{App}\ t\ r]\!] &= \epsilon \circ \langle [\![t]\!], [\![r]\!] \rangle & [\![\langle\rangle]\!] &= !_{[\![\Gamma]\!]} \\ [\![\lambda t]\!] &= \widetilde{[\![t]\!]} & [\![\mathsf{id}_\Gamma]\!] &= 1_{[\![\Gamma]\!]} \\ [\![t\gamma]\!] &= [\![t]\!] \circ [\![\gamma]\!] & [\![(\gamma, t)]\!] &= \langle [\![\gamma]\!], [\![t]\!] \rangle \\ [\![q]\!] &= \pi_2 & [\![p]\!] &= \pi_1 \\ & & [\![\gamma\, \delta]\!] &= [\![\gamma]\!] \circ [\![\delta]\!] \end{aligned}$$

The notion of validity in this setting consists of showing that the semantics of two equal terms are equal as morphisms in the underlying category of the model and the semantic of equal substitutions are also equal as morphisms.

Definition 4 (Validity).

- Terms: $\Gamma \vDash t = t' : A$ iff $[\![t]\!] = [\![t']\!] : [\![\Gamma]\!] \to [\![A]\!]$
- Substitutions: $\Gamma \vDash \sigma = \sigma' : \Delta$ iff $[\![\sigma]\!] = [\![\sigma']\!] : [\![\Gamma]\!] \to [\![\Delta]\!]$.

Since the formal system is the initial object in the category of cartesian closed categories, soundness follows from initiality. This result is valid both for λ^β and $\lambda^\to$; because (η) is modeled in every CCC.

Theorem 1 (Soundness [74, Part I, 11]). if $\Gamma \vdash J$, then $\Gamma \vDash J$ in any model.

Analysing NbE

It is clear that one cannot define an inverse of $[\![\_]\!]$ satisfying properties 1,2, and 3 in p. 24, for any model; the immediate counter-example is a proof-irrelevant model, where all the terms of the same type are interpreted as the same element. So, let us consider in more detail what we need in the model to be able to define the reification function: first we need to interpret distinct variables as distinct values, for variables are already in normal form and otherwise we would not be able to prove $x = \mathsf{nbe}(x)$, for some variable x.

The same reason applies also to neutral values, so we also need to be able to represent neutrals in the model. The term model clearly has this property of separating distinct normal forms, but it lacks the capacity of normalising terms. A possible remedy for this can be to take the set Nf of normal forms as the interpretation of each basic type and interpret higher order types, $A \to B$, as $[\![A]\!] \to [\![B]\!]$, the set of functions from $[\![A]\!]$ to $[\![B]\!]$.

Let us try to see how far we can go with the interpretation in that model. It looks like we can interpret variables as themselves; then abstractions are interpreted as usual

$$[\![\lambda x.t]\!]d = d' \mapsto [\![t]\!][d \mid x : d'] \ .$$

The problem with this naïve interpretation is that if application is interpreted as regular evaluation of functions, we cannot interpret open terms: e.g., $[\![\mathsf{App}\ x\ t]\!]d$ is not well-defined, because $x\ ([\![t]\!]d)$ does not make sense — remember that variables are interpreted as themselves. What we do instead is to redefine the semantics of application. This is achieved by *tagging* elements in $[\![A \to B]\!]$ as neutral terms or functions, an idea coming from NbE for the untyped lambda calculus [55]; now we can use the tag of the value in the function place to redefine application:

$$\begin{aligned} \iota_{\mathit{fun}}\, f \cdot d &= f\ d \\ \iota_{\mathit{neut}}\, t \cdot d &= \mathsf{App}\ t\ d \ . \end{aligned}$$

When we use this operator to interpret application and consider $\iota_{\mathit{neut}}\, t \cdot d$, where t is a neutral with type $N \to B$ and d is the semantics of some term with type N, we know that d is a normal form and the right hand side makes sense. On the other hand, if t has type $[\![(A \to B) \to C]\!]$ and $d \in [\![A \to B]\!]$, then the argument d can be a function and the right hand side does not make sense. To solve this, we can switch to a domain-theoretic model based on the solution of the domain equation

$$D \approx \mathit{Var} \oplus D \times D \oplus [D \to D] \ ,$$

analogous to the domain of Eq. 2.1. An element $d \in D$ will be called a *neutral value* if $d = \mathsf{Var}\, x$, with $x \in \mathit{Var}$, or if $d = \mathsf{App}\, e\, e'$, and e is a neutral value — this terminology will become clearer later. Now we define the application $\_ \cdot \_$ as a binary operation on D:

$$d \cdot e = \begin{cases} f\ e & \text{if } d = \mathsf{lam}\, f \\ \mathsf{App}\, d\, e & \text{if } d \text{ is neutral} \\ \bot & \text{otherwise} \ . \end{cases}$$

Coming back to the interpretation of types, we can set $[\![N]\!]$ to be the set of neutral values, and $[\![A \to B]\!] = \{d \mid d \cdot e \in [\![B]\!], \text{ for all } e \in [\![A]\!]\}$. The next step is to define the reification function: $R : D[\![A]\!] \to \mathit{Terms}$.

$$\begin{aligned} R(\mathsf{Var}\, x) &= x \\ R(\mathsf{App}\, d\, e) &= \mathsf{App}\ R(d)\ R(e) \\ R(\mathsf{lam}\, f) &= \lambda x.R(f\ (\mathsf{Var}\, x)) \ . \end{aligned}$$

The reification of functions looks suspicious, because we are putting a binder without checking that x, the binding variable, is fresh. One possibility, cf. [58],

is to use a *dummy* variable $\tilde{x}$, calculate the set of free-variables in the resulting term, $V = FV(R(f\ (Var\ \tilde{x})))$ and pick any variable $z \in Var \setminus V$ to finally compute $\lambda z.R(f\ (Var\ z))$. This method is clearly inefficient, but it illustrates the problem of freshness: a fresh binder is needed to avoid capturing other variables, and the chosen variable should not be captured by any binder in the reification of the body of the new abstraction.

In Berger and Schwichtenberg's [26] seminal paper, they solved the problem of fresh name generation by using the *gensym* facility of Scheme — in a pure functional language one would resort to a state monad, as in Filinski's [54]. However, Berger and Schwichtenberg came up with another solution based on a model where ground types are interpreted as term families indexed by natural numbers, by fixing the index one gets a term with de Bruijn *levels*;[1] higher-order types $A \to B$ are still interpreted as the set of functions $[\![A]\!] \to [\![B]\!]$. The interpretation of open terms depends on a family of *reflection* functions $\uparrow^A : \mathit{Terms} \to [\![A]\!]$, indexed by types. Reification is also different; it is defined by induction on types and turns semantical elements into term families: $R^A : [\![A]\!] \to (\mathbb{N} \to \Lambda_A)$. Note that when reifying a function $f \in [\![A \to B]\!]$ we have at our disposition an index k telling that we can freely construct a term with binding variable x_k and, if we increment that index to $k+1$, no bound variable will be used in the reification of the body: $R^{A\to B}(f, k) = \lambda x_k.R^B(f\ (\uparrow x_k), k+1)$. The normalisation of a closed term $t : A$ is obtained by $R^A([\![t]\!], 0)$. A similar approach is used by Aehlig and Joachimski [11], but they use de Bruijn indices instead of levels, so more accounting is needed when injecting variables into the model and in the reification.

In this thesis the problem of generating fresh variable while reifying are based on the same strategy presented in Sec. 2.2: free variables are interpreted as (semantical) variable corresponding to the position of the variable in the context but in reversed order — note that this correspond to pass from indices to levels. When we reify values into terms, we keep a counter (as an extra parameter of the reification function) of how many variables we have already used.

2.3 Normalisation by Evaluation for $\lambda^{\to}$

In this section we define a concrete CCC for interpreting $\lambda^{\to}$; this model can be seen as what Reynolds [99] calls an *extrinsic semantics*.

PER semantics

Our model for normalisation is based on the domain D defined in Eq. (2.1) in Sec. 2.2. Since we want to model $\lambda^{\to}$ we change the definition of the application operation; thus we obtain a different applicative structure. We use D to refer to the new applicative structure.

$$d \cdot d' = \begin{cases} f\ d' & \text{if } d = \mathsf{lam}\ f \\ \bot & \text{otherwise} \end{cases} \tag{2.2}$$

[1] Like with de Bruijn indices, variables are taken to be natural numbers, but now the binders are counted downwards, if we see terms as abstract trees, $\lambda x.\lambda y.x$ becomes $\lambda.\lambda.0$.

Instead of interpreting typing derivations — this is problematic because one needs to prove a coherence result: if there are two derivations of the same judgement, the interpretation of both derivations are equal — we interpret every term and substitution expression in the *environment model* arising from the new applicative structure.

Interpretation

$$\begin{aligned} [\![\mathsf{App}\ t\ r]\!]d &= [\![t]\!]d \cdot [\![r]\!]d & [\![\langle\rangle]\!]d &= \top \\ [\![\lambda t]\!]d &= \mathsf{lam}\,(e \mapsto [\![t]\!](d,e)) & [\![\mathsf{id}_\sigma]\!]d &= d \\ [\![t\sigma]\!]d &= [\![t]\!]([\![\sigma]\!]d) & [\![(\sigma,t)]\!]d &= ([\![\sigma]\!]d, [\![t]\!]d) \\ [\![q]\!]d &= q\ d & [\![p]\!]d &= p\ d \\ & & [\![\sigma\,\delta]\!]d &= [\![\sigma]\!]([\![\delta]\!]d) \end{aligned}$$

PER semantics Now we define a PER model[2] for the calculus; this amounts to defining a relation for each type which models the equality of the formal system. Let us recall the definition of partial equivalence relations.

Definition 5. A partial equivalence relation (PER) over a set A is a binary relation over A which is symmetric and transitive.

The class of all PERs over a set A is written $\mathsf{PER}(A)$. If $R \in \mathsf{PER}(A)$ then its domain is $\mathrm{dom}(R) = \{a \in A \mid (a,a) \in R\}$. Clearly, R is an equivalence relation over its domain. If $(d,e) \in R$, sometimes we will write $d =_R e$ or $d = e \in R$, and if $d \in \mathrm{dom}(R)$, we tend to write $d \in R$. It seems natural to try to build a category of PERs over a set A; the objects of $\mathsf{PER}(A)$ are PERs over A and morphisms are functions preserving the relation; i.e. functions $f : \mathrm{dom}(R) \to \mathrm{dom}(S)$, such that $fd =_S fe$, for all $d =_R e$.[3]

If A is an applicative structure $\langle A, \_\cdot\_\rangle$, then we can define a notion of exponential in $\mathsf{PER}(A)$ as follows. For $R, S \in \mathsf{PER}(A)$ we say $f = g \in S^R$ iff $f\cdot d =_S g\cdot e$ for all $d =_R e$. It is clear that $\langle D, \_\cdot\_\rangle$, with the application defined in Eq. 2.2, is an applicative structure. As the following lemma shows, $\mathsf{PER}(D)$ is also a cartesian closed category.

Lemma 2. The category $\mathsf{PER}(D)$ is cartesian closed.

Proof. We define the terminal object, the product and the exponential together with the required morphisms.

- The terminal object is given by $1 = \{(\top,\top)\}$. The proof of its universal property is trivial. Note that for any non-empty $D' \subseteq D$, $D' \times D'$ is isomorphic to 1; in particular $D \times D$ as in [14, 42].

- The product of A and B is given by $A \times B = \{(d,e) \mid p\ d =_A p\ e \text{ and } q\ d =_B q\ e\}$. The projection morphisms are p and q. The morphism $\langle f, g\rangle : C \to A \times B$ is given by $\langle f, g\rangle\ d = (f\ d, g\ d)$. We leave the proof of the universal property for products; for it follows straightforward from the definition.

[2] We refer the reader to [88] for a short report on the historical developments of PER models.

[3] To be precise, two functions $f, g : \mathrm{dom}(R) \to \mathrm{dom}(S)$ are considered to be the same morphism if $f\ d =_S g\ e$, for all $d =_R e$.

- The exponential of A and B is $B^A = \{(f, g) \mid f \cdot d =_B g \cdot e, \text{ for all } d =_A e\}$. The adjunction between $\mathrm{Hom}(\_, C^B)$ and $\mathrm{Hom}(\_ \times B, C)$ is given by:

$$\begin{aligned} \phi_A &: \mathrm{Hom}(A \times B, C) \to \mathrm{Hom}(A, C^B) \\ \phi_A f &= d \mapsto \mathsf{lam}\,(e \mapsto f(d, e)) \\ \psi_A &: \mathrm{Hom}(A, C^B) \to \mathrm{Hom}(A \times B, C) \\ \psi_A g &= d \mapsto g(pd) \cdot (qd) \end{aligned}$$ □

Now that we have a cartesian closed category of PERs over D, we can construct the model and get soundness of this model by Thm. 1. As we said previously, for STT it is enough to choose an object of the category, $\mathrm{PER}(D)$ in our case, for base types; the rest of the structure is obtained by the cartesian structure of the category. In our case, N is interpreted by $\mathsf{Ne} \in \mathrm{PER}(D)$, which is analogous to Ne of Sec. 2.2. The PER Ne equates elements of D that are invariably reified as the same neutral term.

Definition 6 (Semantical neutrals and normal forms).

- $d = e \in \mathsf{Ne}$ if, for all $i \in \mathbb{N}$, $\mathsf{R}_i\, d$ and $\mathsf{R}_i\, e$ are both defined, $\mathsf{R}_i\, d \equiv \mathsf{R}_i\, e$, and $\mathsf{R}_i\, d \in \mathsf{Ne}$.
- $d = e \in \mathsf{Nf}$ if, for all $i \in \mathbb{N}$, $\mathsf{R}_i\, d$ and $\mathsf{R}_i\, e$ are both defined, $\mathsf{R}_i\, d \equiv \mathsf{R}_i\, e$, and $\mathsf{R}_i\, d \in \mathsf{Nf}$.

Remark 3. These are clearly PERs over D: symmetry is trivial and transitivity follows from transitivity of the syntactical equality.

Remark 4 (Closure properties over Nf and Ne).

1. $\mathit{Var} \subseteq \mathrm{dom}(\mathsf{Ne})$.
2. Since $\mathsf{Ne} \subseteq \mathsf{Nf}$, $\mathsf{Ne} \subseteq \mathsf{Nf}$.
3. If $k = k' \in \mathsf{Ne}$ and $d = d' \in \mathsf{Nf}$, then $\mathsf{App}\, k\, d = \mathsf{App}\, k'\, d' \in \mathsf{Ne}$.
4. Let $f, g \in [D \to D]$ such that $f\ d = g\ d' \in \mathsf{Nf}$, for all $d = d' \in \mathsf{Nf}$, then $\mathsf{lam}\, f = \mathsf{lam}\, g \in \mathsf{Nf}$:

$$\mathsf{R}_i\,(\mathsf{lam}\, f) = \lambda(\mathsf{R}_{i+1}\,(f\,(\mathsf{Var}\, i))) = \lambda(\mathsf{R}_{i+1}\,(g\,(\mathsf{Var}\, i))) = \mathsf{R}_i\,(\mathsf{lam}\, g)\ .$$

If we choose $[\![\mathsf{N}]\!] = \mathsf{Ne}$, then we have a model of STT given by the cartesian structure of $\mathrm{PER}(D)$:

$$[\![\diamond]\!] = 1 \qquad [\![\Gamma.A]\!] = [\![\Gamma]\!] \times [\![A]\!] \qquad [\![A \to B]\!] = [\![B]\!]^{[\![A]\!]}\ .$$

We recast soundness (Thm. 1) in our concrete model. Of course, soundness can be proved straightforwardly by induction on derivations.

Theorem 3. Let $d = d' \in [\![\Gamma]\!]$;

1. If $\Gamma \vdash t : A$, then $[\![t]\!]d \in [\![A]\!]$.
2. If $\Gamma \vdash \sigma : \Delta$, then $[\![\sigma]\!]d \in [\![\Delta]\!]$.
3. If $\Gamma \vdash t = t' : A$, then $[\![t]\!]d = [\![t']\!]d' \in [\![A]\!]$.

4. If $\Gamma \vdash \delta = \delta' \colon \Delta$, then $[\![\delta]\!]\mathsf{d} = [\![\delta']\!]\mathsf{d}' \in [\![\Delta]\!]$.

Remark 5. Clearly $\bot \notin \mathrm{dom}([\![\mathsf{N}]\!])$; since $\bot \cdot \mathsf{d} = \bot$ we conclude $\bot \notin [\![\mathsf{A}]\!]$ for any type A. Moreover if $\mathsf{d} \in \mathrm{dom}([\![\mathsf{A} \to \mathsf{B}]\!])$, then $\mathsf{d} = \mathsf{lam}\,\mathsf{f}$, for some $\mathsf{f} \in [\mathsf{D} \to \mathsf{D}]$.

We have explained in Sec. 2.2 why the model should contain some syntactical material as to reflect neutral terms. Our model can indeed represent variables and neutrals; but as an immediate consequence of Rem. 5 we cannot interpret syntactic variables with a higher-order type as mere semantic variables. In fact, from that remark we know that injection of variables should transform variables into functions $\mathsf{lam}\,\mathsf{f}$; this has the pleasing side effect of η-expanding variables of type $\mathsf{A} \to \mathsf{B}$ — while $\mathsf{Var}\,\mathsf{i}$ is reified as some variable q or $\mathsf{q}\,\mathsf{p}^n$ for some n, $\mathsf{lam}\,\mathsf{f}$ is read back as $\lambda\mathsf{t}$ for some term t. Remember that injection was done by a family of functions $\uparrow_{\mathsf{A}} \_$ indexed by types; in Berger and Schwichtenberg's work that function turned syntactical variables into term families. Our approach consists in defining reflection $\uparrow_{\mathsf{A}} \_$ completely in the semantic realm; as we explain after its definition, reflection needs a mate $\downarrow_{\mathsf{A}} \_$. These functions are defined simultaneously by induction on types.

Definition 7 (Up and down).

$$\uparrow_{\mathsf{N}} \mathsf{k} = \mathsf{k} \qquad \uparrow_{\mathsf{A}\to\mathsf{B}} \mathsf{k} = \mathsf{lam}\,(\mathsf{d} \mapsto \uparrow_{\mathsf{B}} (\mathsf{App}\,\mathsf{k}\,(\downarrow_{\mathsf{A}} \mathsf{d})))$$
$$\downarrow_{\mathsf{N}} \mathsf{v} = \mathsf{v} \qquad \downarrow_{\mathsf{A}\to\mathsf{B}} \mathsf{d} = \mathsf{lam}\,(\mathsf{e} \mapsto \downarrow_{\mathsf{B}} (\mathsf{d} \cdot \uparrow_{\mathsf{A}} \mathsf{e}))$$

Notice that we do not restrict the domain of the reflection function to elements in *Var*; in fact, since we want to simulate η-expansion by means of $\uparrow_{\mathsf{A}\to\mathsf{B}}$, it is necessary to take into account representation of any neutral term and not only variables. The need of $\downarrow_{\mathsf{A}}$ arises as we want to avoid to reify values as in Berger and Schwichtenberg's approach: since the body of the newly created function is in turn expanded, we need to know that it is morally a neutral value. That would be the case if the second argument of $\mathsf{App}\,\mathsf{k}\,\_$ is a normal value, cf. Rem. 4; the mate of reflection, $\downarrow_{\mathsf{A}}$ maps elements of $\mathrm{dom}([\![\mathsf{A}]\!])$ to $\mathrm{dom}(\mathsf{Nf})$.

The following lemma states precisely that the pair of functions do what we claimed. We can also read the lemma as showing that every PER denoting a type can be seen as a saturated object [78, Ch. 4].

Lemma 4. For all $\mathsf{A} \in \mathsf{Type}$.

1. If $\mathsf{k} = \mathsf{k}' \in \mathsf{Ne}$ then $\uparrow_{\mathsf{A}} \mathsf{k} = \uparrow_{\mathsf{A}} \mathsf{k}' \in [\![\mathsf{A}]\!]$, and
2. if $\mathsf{d} = \mathsf{d}' \in [\![\mathsf{A}]\!]$, then $\downarrow_{\mathsf{A}} \mathsf{d} = \downarrow_{\mathsf{A}} \mathsf{d}' \in \mathsf{Nf}$.

Proof. (By induction on types) We show only the case for functional types.

1. Let $\mathsf{k} = \mathsf{k}' \in \mathsf{Ne}$ and $\mathsf{d} = \mathsf{d}' \in [\![\mathsf{A}']\!]$. To prove $\uparrow_{\mathsf{A}'\to\mathsf{B}} \mathsf{k} = \uparrow_{\mathsf{A}'\to\mathsf{B}} \mathsf{k}' \in [\![\mathsf{A}']\!] \to [\![\mathsf{B}]\!]$, we prove $(\uparrow_{\mathsf{A}'\to\mathsf{B}} \mathsf{k}) \cdot \mathsf{d} = (\uparrow_{\mathsf{A}'\to\mathsf{B}} \mathsf{k}') \cdot \mathsf{d}' \in [\![\mathsf{B}]\!]$. By i.h. on the second part for A' we know $\downarrow_{\mathsf{A}'} \mathsf{d} = \downarrow_{\mathsf{A}'} \mathsf{d}' \in \mathsf{Ne}$. Hence by Rem. 4, $\mathsf{App}\,\mathsf{k}\,(\downarrow_{\mathsf{A}'} \mathsf{d}) = \mathsf{App}\,\mathsf{k}'\,(\downarrow_{\mathsf{A}'} \mathsf{d}') \in \mathsf{Ne}$ and by i.h. on the first part for B we conclude $\uparrow_{\mathsf{B}} (\mathsf{App}\,\mathsf{k}\,(\downarrow_{\mathsf{A}'} \mathsf{d})) = \uparrow_{\mathsf{B}} (\mathsf{App}\,\mathsf{k}'\,(\downarrow_{\mathsf{A}'} \mathsf{d}')) \in [\![\mathsf{B}]\!]$.
2. Let $\mathsf{d} = \mathsf{d}' \in [\![\mathsf{A}']\!] \to [\![\mathsf{B}]\!]$. From Rem. 4 and by i.h. on the first part A' we have $\uparrow_{\mathsf{A}'} \mathsf{Var}\,\mathsf{i} = \uparrow_{\mathsf{A}'} \mathsf{Var}\,\mathsf{i} \in [\![\mathsf{A}']\!]$, so by definition $\mathsf{d} \cdot \uparrow_{\mathsf{A}'} (\mathsf{Var}\,\mathsf{i}) = \mathsf{d}' \cdot \uparrow_{\mathsf{A}'} (\mathsf{Var}\,\mathsf{i}) \in [\![\mathsf{B}]\!]$, therefore we conclude, by i.h. on the second part for B, $\downarrow_{\mathsf{B}} (\mathsf{d} \cdot (\uparrow_{\mathsf{A}'} \mathsf{Var}\,\mathsf{i})) = \downarrow_{\mathsf{B}} (\mathsf{d}' \cdot (\uparrow_{\mathsf{A}'} \mathsf{Var}\,\mathsf{i})) \in \mathsf{Nf}$.

□

Theorem 5 (Completeness). If $\Gamma \vdash t = t' : A$ and $d = d' \in [\![\Gamma]\!]$, then $\downarrow_A ([\![t]\!]d) = \downarrow_A ([\![t']\!]d') \in \mathsf{Nf}$.

Proof. By Thm. 3 and Lem. 4. □

We still do not have a normalisation algorithm because we do not know how to construct a suitable environment. Let us illustrate this point by postulating a base type with two values, call it Bool with constructors True and False. Since these are normal forms, we need to reflect them on the domain; so we extend the domain D with $\{\mathsf{T}, \mathsf{F}\}_\perp$. Now in the context $\Gamma = \mathsf{Bool}.\mathsf{Bool}$ we can derive $\Gamma \vdash q : \mathsf{Bool}$ and $\Gamma \vdash q\,p : \mathsf{Bool}$. Note that $d = (\mathsf{F}, \mathsf{F}) \in \mathrm{dom}(\Gamma)$, so $\mathsf{R}_j\,(\downarrow_{\mathsf{Bool}} ([\![q]\!]d)) \equiv \mathsf{F} \equiv \mathsf{R}_j\,(\downarrow_{\mathsf{Bool}} ([\![q\,p]\!]d))$, but q and $q\,p$ were distinct normal forms and none was F. In the next section we construct an appropiate environment for NbE and obtain correctness of NbE by using logical relations.

2.4 Correctness of NbE

As with many other results for STT, it is not possible to prove correctness by induction on terms — as usual the induction hypotheses for application are not strong enough. To have stronger induction hypotheses we use *logical relations*[4]; these are families of relations, indexed by types, relating two models. In our case, the models involved are the *term model* and the PER model: that means that we should relate equivalence classes of terms and equivalence classes over the domain of PERs. To understand our use of logical relations let us state soundness of NbE:

$$\text{If } \Gamma \vdash t : A, \text{ then } \Gamma \vdash t = \mathsf{R}_n\,(\downarrow_A ([\![t]\!]d)) : A, \text{for some } n \in \mathbb{N}, d \in [\![\Gamma]\!] \ .$$

To prove that property for every well-typed term we first define the relations between well-typed terms and elements in (the domain of) the interpretation of types and prove the more general statement

$$\text{If } \Gamma \vdash t : A \sim d \in [\![A]\!], \text{ then } \Gamma \vdash t = \mathsf{R}_n\,(\downarrow_A d) : A \text{ for some } n \in \mathbb{N} \ . \qquad (2.3)$$

The logical relations are defined by induction on types. For the basic type N, we say that a term t is logically related with $d \in [\![\mathsf{N}]\!]$ if $\Gamma \vdash t = \mathsf{R}_n\,(\downarrow_{\mathsf{N}} d) : \mathsf{N}$ holds — note that we are not relating equivalence classes of terms with equivalent classes over $\mathrm{dom}([\![\mathsf{N}]\!])$; we then prove that the relations preserve both equivalences. For higher-order types $A \to B$, the relation is defined by using the relations for A and B:

$$\Gamma \vdash t : A \to B \sim d \in [\![A]\!] \to [\![B]\!] \text{ if } \Gamma \vdash \mathsf{App}\ t\ t' : B \sim d \cdot d' \in [\![B]\!], \text{ for all } \Gamma \vdash t' : A \sim d' \in [\![A]\!] \ .$$

As we try to prove (2.3) for abstractions we discover that we need to generalise its statement to be able to change the context. The reason is that the term model is a Kripke model, with contexts taking the rôle of possible worlds; this leads us to Kripke logical relations [91]. At this point we would have

[4] A thorough explanation of logical relations can be found in [90, Chap. 8].

proved that if a well-typed term $\Gamma \vdash t : A$ is related with some element d, then $\Gamma \vdash t = \mathsf{R}_{|\Gamma|}\,(\downarrow_A d) : A$. The missing step is to prove that each term is logically related with its denotation. This property is known as the *fundamental theorem* of logical relations: the denotations of every term in both models are logically related. To prove the fundamental theorem we extend logical relations to environments; in the term model, environments are substitutions. In $\lambda^{\to}$, substitutions are internalised in the system; so we introduce logical relations for substitutions and elements in the interpretation of contexts. Finally, by constructing an environment logically related with the identity substitution we can conclude that every term is logically related with its denotation.

Definition 8 (Logical relations).

- For the basic type N: $\Gamma \vdash t : \mathsf{N} \sim d \in [\![\mathsf{N}]\!]$ if $\Delta \vdash t\,\mathsf{p}^i = \mathsf{R}_{|\Delta|}\,(\downarrow_{\mathsf{N}} d) : \mathsf{N}$, for all $\Delta \leqslant^i \Gamma$.
- For function spaces, $A \to B$: $\Gamma \vdash t : A \to B \sim d \in [\![A \to B]\!]$ if $\Delta \vdash \mathsf{App}\,(t\,\mathsf{p}^i)\ s : B \sim f \cdot d \in [\![B]\!]$, for all $\Delta \leqslant^i \Gamma$ and $\Delta \vdash s : A \sim d \in [\![A]\!]$.

The following two lemmas show that the relations introduced over sets of well-typed terms and domains of the PERs are indeed relations between equivalence classes of terms and equivalence classes over the domain of PERs.

Lemma 6 (Preservation of the logical relation by PERs). If $\Gamma \vdash t : A \sim d \in [\![A]\!]$ and $d = d' \in [\![A]\!]$, then $\Gamma \vdash t : A \sim d' \in [\![A]\!]$.

Proof. (By induction on types)

- For N is obtained directly from the definition of the logical relation and from the definition of Ne.
- For $A' \to B$: let $\Delta \leqslant^i \Gamma$ and $\Delta \vdash s : A' \sim e \in [\![A']\!]$. From $e = e \in [\![A]\!]$ we have $d \cdot e = d' \cdot e \in [\![B]\!]$ and by definition of the logical relation we have $\Delta \vdash \mathsf{App}\,(t\,\mathsf{p}^i)\ s : B \sim d \cdot e \in B$. By ind. hypothesis we have $\Delta \vdash \mathsf{App}\,(t\,\mathsf{p}^i)\ s : B \sim d' \cdot e \in B$. □

Lemma 7 (Preservation of the logical relation by judgemental equality). If $\Gamma \vdash t : A \sim d \in [\![A]\!]$ and $\Gamma \vdash t = t' : A$, then $\Gamma \vdash t' : A \sim d \in [\![A]\!]$.

Proof. (By induction on types).

- For N : we have $\Gamma \vdash t = \mathsf{R}_{|\Gamma|}\,(\downarrow_{\mathsf{N}} d) : \mathsf{N}$ and $\Gamma \vdash t = t' : \mathsf{N}$, by transitivity we conclude $\Gamma \vdash t' = \mathsf{R}_{|\Gamma|}\,(\downarrow_{\mathsf{N}} d) : \mathsf{N}$
- For $A' \to B$: Let $\Delta \leqslant^i \Gamma$ and $\Delta \vdash s : A' \sim e \in [\![A]\!]$. From the hypothesis of equality we derive $\Delta \vdash t\,\mathsf{p}^i = t'\,\mathsf{p}^i : A \to B$ and $\Delta \vdash \mathsf{App}\,(t\,\mathsf{p}^i)\ s = \mathsf{App}\,(t'\,\mathsf{p}^i)\ s : B$. From the hypothesis about the logical relation we know $\Delta \vdash \mathsf{App}\,(t\,\mathsf{p}^i)\ s : B \sim d \cdot e \in [\![B]\!]$. By i.h. on B we have $\Delta \vdash \mathsf{App}\,(t'\,\mathsf{p}^i)\ s : B \sim d \cdot e \in [\![B]\!]$. □

Lemma 8 (Monotonicity of the logical relation). If $\Gamma \vdash t : A \sim d \in [\![A]\!]$ and $\Delta \leqslant^i \Gamma$, then $\Delta \vdash t\,\mathsf{p}^i : A \sim d \in [\![A]\!]$.

Proof. (By cases on A) Let $\Delta \leqslant^i \Gamma$. In both cases we need to prove a property involving a new context Δ' such that $\Delta' \leqslant^j \Delta$. By Rem. 1 we know $\Delta' \leqslant^{i+j} \Gamma$.

- To prove $\Delta \vdash t\,p^i \colon \mathsf{N} \sim d \in [\![\mathsf{N}]\!]$, we need to show $\Delta' \vdash (t\,p^i)\,p^j = R_{|\Delta'|}\,d \colon \mathsf{N}$, for any $\Delta' \leqslant^j \Delta$. So by definition of the logical relation we have $\Delta' \vdash t\,p^{i+j} = R_{|\Delta'|}\,d \colon \mathsf{N}$; we conclude by using associativity of substitutions and Lem. 7.
- For $A' \to B$: Let $\Delta' \leqslant^j \Delta$ and $\Delta' \vdash s \colon A' \sim e \in [\![A']\!]$ and show $\Delta' \vdash \mathsf{App}\,((t\,p^i)\,p^j)\,s \colon B \sim d \cdot e \in [\![B]\!]$. Again, by definition of logical relation for function spaces $\Delta' \vdash \mathsf{App}\,(t\,p^{i+j})\,s \colon B \sim d \cdot e \in [\![B]\!]$; we conclude by using associativity of substitutions and Lem. 7. □

The following lemma is one of the key results to prove correctness. Note that we are interested in the second part, which is the general statement of (2.3); the first part is needed to prove the second part for higher-order types.

Lemma 9 (In-out Lemma).

1 Let $\Gamma \vdash t \colon A$, $k \in \mathrm{dom}(\mathsf{Ne})$ and $\Delta \vdash t = R_{|\Delta|}\,k \colon A$ for all $\Delta \leqslant^i \Gamma$, then $\Gamma \vdash t \colon A \sim \uparrow_A k \in [\![A]\!]$.

2 If $\Gamma \vdash t \colon A \sim d \in [\![A]\!]$, then $\Delta \vdash t\,p^i = R_{|\Delta|}\,(\downarrow_A d) \colon A$, for all $\Delta \leqslant^i \Gamma$.

Proof. (We prove simultaneously both points by induction on types) We show only the case for functional types.

1. To prove $\Gamma \vdash t \colon A' \to B \sim \uparrow_{A'\to B} k \in [\![A']\!] \to [\![B]\!]$, let $\Delta \leqslant^i \Gamma$ and $\Delta \vdash s \colon A' \sim d \in [\![A']\!]$ to prove

$$\Delta \vdash \mathsf{App}\,(t\,p^i)\,s \colon B \sim (\uparrow_{A'\to B} k) \cdot d \in [\![B]\!] \ .$$

   Note that by definition of $\uparrow_{A'\to B} k$, that amounts to prove

$$\Delta \vdash \mathsf{App}\,(t\,p^i)\,s \colon B \sim \uparrow_B (\mathsf{App}\,k\,(\downarrow_{A'} d)) \in [\![B]\!] \ .$$

   In turn, this can be proved by i.h. of 9.1 on B, because $k \in \mathsf{Ne}$ and $\downarrow_{A'} d \in \mathsf{Nf}$, by Rem. 4, $\mathsf{App}\,k\,(\downarrow_{A'} d) \in \mathsf{Ne}$; so we take $\Delta' \leqslant^j \Delta$ and prove

$$\Delta' \vdash \mathsf{App}\,(t\,p^{i+j})\,(s\,p^j) = R_{|\Delta'|}\,(\mathsf{App}\,k\,(\downarrow_{A'} d)) \colon B \ .$$

   Note that by definition of readback, our new goal is

$$\Delta' \vdash \mathsf{App}\,(t\,p^{i+j})\,(s\,p^j) = \mathsf{App}\,(R_{|\Delta'|}\,k)\,(R_{|\Delta'|}\,(\downarrow_{A'} d)) \colon B \ .$$

   We know by hypothesis $\Delta' \vdash t\,p^{i+j} = R_{|\Delta'|}\,k \colon A' \to B$ and, by i.h. of 9.2 on A', $\Delta' \vdash s\,p^j = R_{|\Delta'|}\,(\downarrow_{A'} d) \colon A'$ We use congruence to conclude.

2. Let $\Delta \leqslant^i \Gamma$ and let $n = |\Delta|$. First we notice that $\Delta.A' \vdash q \colon A' \sim \mathsf{Var}\,n \in [\![A']\!]$ by i.h. of 9.1 on A'. By definition of the logical relation on the main hypothesis, we know $\Delta.A' \vdash \mathsf{App}\,(t\,p^{i+1})\,q \colon B \sim d \cdot \uparrow_{A'} (\mathsf{Var}\,n) \in [\![B]\!]$. By i.h. of 9.2 on B we have

$$\Delta.A' \vdash \mathsf{App}\,(t\,p^{i+1})\,q = R_{n+1}\,(\downarrow_B (d \cdot (\uparrow_{A'} (\mathsf{Var}\,n)))) \colon B \ .$$

   By (cong-abs),

$$\Delta \vdash \lambda(\mathsf{App}\,(t\,p^{i+1})\,q) = \lambda(R_{n+1}\,(\downarrow_B (d \cdot (\uparrow_{A'} (\mathsf{Var}\,n))))) \colon A' \to B \ .$$

Now, by (η), we have $\Delta \vdash \lambda(\mathsf{App}\ (t\,p^{i+1})\ q) = t\,p^i \colon A' \to B$; on the right hand side, in turn, we have by definition of readback

$$\lambda(R_{n+1}\ (\downarrow_B d \cdot (\uparrow_{A'} \mathsf{Var}\, n))) = R_n\ (\downarrow_{A' \to B} d)\ .$$

By transitivity, we conclude $\Delta \vdash t\,p^i = R_n\ (\downarrow_{A' \to B} d) \colon A' \to B$. □

We extend the logical relations to substitutions and environments. This family of relations is defined by induction on the codomain of substitutions.

Definition 9 (Logical relations for substitutions).

- $\Gamma \vdash \sigma \colon \diamond \sim d \in [\![\diamond]\!]$ if $d \in [\![\diamond]\!]$.
- $\Gamma \vdash (\sigma, t) \colon \Delta.A \sim (d, d') \in [\![\Delta.A]\!]$ if $\Gamma \vdash \sigma \colon \Delta \sim d \in [\![\Delta]\!]$ and $\Gamma \vdash t \colon A \sim d' \in [\![A]\!]$.

Of course we need to prove analogous results for substitutions to that of Lem. 6, 7, and 8. Note that Lem. 9 is not needed because we do not reify substitutions.

Lemma 10. Let $\Gamma \vdash \sigma \colon \Delta \sim d \in [\![\Delta]\!]$.

1. If $\Gamma \vdash \sigma = \sigma' \colon \Delta$, then $\Gamma \vdash \sigma' \colon \Delta \sim d \in [\![\Delta]\!]$.
2. If $d = d' \in [\![\Delta]\!]$, then $\Gamma \vdash \sigma \colon \Delta \sim d' \in [\![\Delta]\!]$.
3. If $\Gamma' \leqslant^i \Gamma$, then $\Gamma' \vdash \sigma\,p^i \colon \Delta \sim d \in [\![\Delta]\!]$.

Let us recapitulate how far we went towards correctness of NbE, the goal of this section: we have defined a relation between the term model and the PER model; we have proved that if a term t is related with some semantic element d, then the reification of d will be provable equal to the t. The next theorem states that every term (and substitution) is related to its denotation.

Theorem 11 (Fundamental theorem of logical relations).

1. If $\Gamma \vdash t \colon A$ and $\Delta \vdash \sigma \colon \Gamma \sim d \in [\![\Gamma]\!]$, then $\Delta \vdash t\,\sigma \colon A \sim [\![t]\!]d \in [\![A]\!]$.
2. If $\Gamma \vdash \delta \colon \Sigma$ and $\Delta \vdash \sigma \colon \Gamma \sim d \in [\![\Gamma]\!]$, then $\Delta \vdash \delta\,\sigma \colon \Sigma \sim [\![\delta]\!]d \in [\![\Sigma]\!]$.

Proof. (By simultaneous induction on $\Gamma \vdash t \colon A$ and $\Gamma \vdash \delta \colon \Sigma$). In several cases, Lem. 6 and 7 are used to get the right premises that permit us to apply induction hypotheses.

1. Terms:
   - (hyp) $\Gamma.A \vdash q \colon A$. Let $\Delta \vdash \sigma \colon \Gamma.A \sim d \in [\![\Gamma.A]\!]$. By definition of logical relation for substitutions we have $\Gamma \vdash \sigma = (\sigma', t) \colon \Delta$ and $\Delta \vdash t \colon A \sim q\ d \in [\![A]\!]$. We finish by Lem. 7 on $\Gamma.A \vdash q\,(\sigma', t) = t \colon A$.
   - (app-i): $\Gamma \vdash \mathsf{App}\ t\ t' \colon B$. By inversion we know $\Gamma \vdash t \colon A \to B$ and $\Gamma \vdash t' \colon A$; by i.h. on each judgement we have $\Delta \vdash t\,\sigma \colon A \to B \sim [\![t]\!]d \in [\![A]\!] \to [\![B]\!]$ and $\Delta \vdash t'\,\sigma \colon A \sim [\![t']\!]d \in [\![A]\!]$. By definition of logical relations we conclude $\Delta \vdash \mathsf{App}\ (t\,\sigma)\ (t'\,\sigma) \colon B \sim [\![t]\!]d \cdot [\![t']\!]d \in [\![B]\!]$. On one side we have $[\![t]\!]d \cdot [\![t']\!]d = [\![\mathsf{App}\ t\ t']\!]d$ and on the other, $\Delta \vdash \mathsf{App}\ (t\,\sigma)\ (t'\,\sigma) = (\mathsf{App}\ t\ t')\,\sigma \colon B$. We conclude by Lem. 7.

- (ABS-I) $\Gamma \vdash \lambda t \colon A \to B$. To prove $\Delta \vdash (\lambda t)\,\sigma \colon A \to B \sim [\![\lambda t]\!]d \in [\![A]\!] \to [\![B]\!]$, we need to take $\Delta' \leqslant^i \Delta$ and $\Delta' \vdash s \colon A \sim e \in [\![A]\!]$ and show $\Delta' \vdash \mathsf{App}\,(((\lambda t)\,\sigma)\,p^i)\,s \colon B \sim [\![\lambda t]\!]d \cdot e \in B$. Notice that we can derive $\Gamma \vdash \mathsf{App}\,((\lambda t)\,\sigma)\,p^i\,s = t\,((\sigma\,p^i), s) \colon B$ by the following equational reasoning:

$$\begin{aligned}\mathsf{App}\,((\lambda t)\,\sigma)\,p^i\,s &= \mathsf{App}\,((\lambda t)\,(\sigma\,p^i))\,s\\ &= \mathsf{App}\,\lambda(t\,(\sigma\,p^{i+1}, q))\,s = (t\,(\sigma\,p^{i+1}, q))\,(\mathsf{id}, s)\\ &= t\,(\sigma\,p^{i+1}, q)\,(\mathsf{id}, s) = t\,((\sigma\,p^{i+1})\,(\mathsf{id}, s), q\,(\mathsf{id}, s))\\ &= t\,((\sigma\,p^i)\,\mathsf{id}, s) = t\,(\sigma\,p^i, s)\end{aligned}$$

  On the other hand, we have by Lem. 10 and definition of logical relation for substitutions $\Delta' \vdash (\sigma\,p^i, s) \colon \Gamma.A \sim (d, e) \in [\![\Gamma.A]\!]$. So by i.h. on $\Gamma.A \vdash t \colon B$, with this extended substitution, we get $\Delta' \vdash t\,(\sigma\,p^i, s) \colon B \sim [\![t]\!](d, e) \in [\![B]\!]$. Note that $[\![\lambda t]\!]d \cdot e = [\![t]\!](d, e)$, thus by Lem. 7, we get $\Delta' \vdash \mathsf{App}\,(((\lambda t)\,\sigma)\,p^i)\,s \colon B \sim [\![\lambda t]\!]d \cdot e \in [\![B]\!]$.
- (SUBS-TERM) $\Gamma \vdash t\,\delta \colon A$. By i.h. on $\Gamma \vdash \delta \colon \Sigma$ we have $\Delta \vdash \delta\,\sigma \colon \Sigma \sim [\![\delta]\!]d \in [\![\Sigma]\!]$. By i.h. using that substitution and environment on $\Sigma \vdash t \colon A$ we conclude $\Delta \vdash t\,(\delta\,\sigma) \colon A \sim [\![t]\!]([\![\delta]\!]d) \in [\![A]\!]$. We conclude by Lem. 7 on $\Delta \vdash t\,(\delta\,\sigma) = (t\,\delta)\,\sigma \colon A$.

2. Substitutions: we omit some trivial cases.

   - (COMP-SUBS). Let $\Gamma \vdash \delta\,\gamma \colon \Sigma$ and $\Delta \vdash \sigma \colon \Gamma \sim d \in [\![\Gamma]\!]$. By inversion, we have $\Gamma \vdash \gamma \colon \Theta$ and $\Theta \vdash \delta \colon \Sigma$. By i.h. on the first one, we have $\Sigma \vdash \gamma\,\sigma \colon \Theta \sim [\![\gamma]\!]d \in [\![\Theta]\!]$ and using that and the second premise by i.h. , we get $\Delta \vdash \delta\,(\gamma\,\sigma) \colon \Sigma \sim [\![\delta]\!]([\![\gamma]\!]d) \in [\![\Sigma]\!]$. By using Lem. 10 we conclude $\Delta \vdash (\delta\,\gamma)\,\sigma \colon \Sigma \sim [\![\delta\,\gamma]\!]d \in [\![\Sigma]\!]$.
   - (EXT-SUBS). Let $\Gamma \vdash (\delta, t) \colon \Sigma.A$. By inversion, we know $\Gamma \vdash \delta \colon \Sigma$ and $\Gamma \vdash t \colon A$. By i.h. we have $\Delta \vdash \delta\,\sigma \colon \Sigma \sim [\![\delta]\!]d \in [\![\Sigma]\!]$; and by i.h. for $\Gamma \vdash t \colon A$ we conclude $\Delta \vdash t\,\sigma \colon A \sim [\![t]\!]d \in [\![A]\!]$: these are the premises for $\Delta \vdash (\delta, t)\,\sigma \colon \Sigma.A \sim [\![(\delta, t)]\!]d \in [\![\Sigma.A]\!]$.
   - (FST-SUBS). Let $\Gamma.A \vdash p \colon \Gamma$. By inversion on $\Delta \vdash \sigma \colon \Gamma.A \sim d \in [\![\Gamma.A]\!]$ we know $\delta = (\delta', t)$, $d = (e, e')$, and $\Delta \vdash \delta' \colon \Gamma \sim e \in [\![\Gamma]\!]$. By Lem. 10, using (SUB-FST) on $\Delta \vdash p\,(\delta', t) = \delta' \colon \Gamma$, we conclude $\Delta \vdash p\,(\delta', t) \colon \Gamma \sim [\![p]\!](e, e') \in [\![\Sigma]\!]$. □

Remark 6. It is immediate to see that $R_n\,(\mathsf{Var}\,(n-1)) = q$ and for all $m > n$ $R_m\,(\mathsf{Var}\,(n-1)) = q\,p^{m-n}$. Let $n = |\Gamma|$, then we can use Lem. 9 to conclude $\Gamma.A \vdash q \colon A \sim \uparrow_A (\mathsf{Var}\,n) \in [\![A]\!]$.

Definition 10 (Canonical environment). By induction on Γ we define an environment $\rho_\Gamma \in [\![\Gamma]\!]$.

1. $\rho_\diamond = \top$.
2. $\rho_{\Gamma.A} = (\rho_\Gamma, \uparrow_A (\mathsf{Var}\,n))$, where $n = |\Gamma|$.

Remark 7. Clearly, by induction on Γ and using Rem. 6, we have $\Gamma \vdash \mathsf{id}_\Gamma \colon \Gamma \sim \rho_\Gamma \in [\![\Gamma]\!]$.

Our normalisation function is the composition of evaluation with the canonical environment, eta-expansion on the model, and reification.

Definition 11 (Normalisation function).

$$\mathfrak{nbe}^{A}_{\Gamma}(t) = R_{|\Gamma|}\,(\downarrow_{A}\,([\![t]\!]\rho_{\Gamma})) \tag{2.4}$$

After the long trip we finally arrive at the main results of this section. First we show the proof of correctness of the normalisation function and then we explain how to decide if two well-typed terms are judgmentally equal.

Corollary 12. If $\Gamma \vdash t : A$, then $\Gamma \vdash t = \mathfrak{nbe}^{A}_{\Gamma}(t) : A$.

Proof. We use Thm. 11 and Lem. 7 with the identity substitution on the context Γ to get $\Gamma \vdash t : A \sim [\![t]\!]\rho_{\Gamma} \in [\![A]\!]$ and by Lem. 9.2 we conclude $\Gamma \vdash t = R_{|\Gamma|}\,(\downarrow_{A}\,([\![t]\!]\rho_{\Gamma})) : A$. □

Corollary 13. If $\Gamma \vdash t : A$ and $\Gamma \vdash t' : A$ we can decide if $\Gamma \vdash t = t' : A$ is derivable.

Proof. Let $\bar{t} = \mathfrak{nbe}^{A}_{\Gamma}(t)$ and $\bar{t'} = \mathfrak{nbe}^{A}_{\Gamma}(t')$. By Cor. 12 we have

$$\Gamma \vdash t = \bar{t} : A \qquad \text{and} \tag{D1}$$

$$\Gamma \vdash t' = \bar{t'} : A \tag{D2}$$

since $\bar{t}$ and $\bar{t'}$ are normal forms we can decide $\Gamma \vdash \bar{t} = \bar{t'} : A$ by checking syntactically that they are the same term. If $\Gamma \vdash t = t' : A$, then by Thm. 5 $\bar{t} \equiv \bar{t'}$. On the other hand, syntactical equivalence of $\bar{t}$ and $\bar{t'}$ implies that they are provably equal by using transitivity (and symmetry) on derivations (D1) and (D2). □

2.5 A Haskell Implementation of NbE

It is relative immediate to program the NbE function in Haskell.

Syntax.

```
data Type = Iota
          | Fun Type Type

data Term = Term :@ Term     -- application
          | Lam Term         -- abstraction
          | Q                -- variable
          | Sub Term Subst   -- substitution
            deriving (Eq)

data Subst = E                 -- empty substitution
           | Is                -- identity substitution
           | Ext Subst Term    -- extension
           | P                 -- weakening
           | Comp Subst Subst  -- composition
             deriving (Eq)

type Ctx = [Type]
```

Semantic domain.

```
data D = T                    -- terminal object (empty context)
       | Ld (D -> D)          -- functions
       | PairD D D            -- context comprehension
       | Vd Int               -- variables
       | AppD D D             -- neutrals

pi1,pi2 :: D -> D
pi1 (PairD d d') = d
pi2 (PairD d d') = d'

ap :: D -> D -> D
ap (Ld f) d = f d
```

Eta expansion in the model.

```
up        :: Type -> D -> D
up Iota k       =  k
up (Fun t t') k = Ld (\d -> up t' (AppD k (down t d)))

down        :: Type -> D -> D
down Iota d        = d
down (Fun t t')  d = Ld (\e -> down t' (d `ap` (up t e)))

readback :: Int -> D -> Term
readback i (Ld f)         = Lam $ readback (i+1) (f (Vd i))
readback i (Vd n)         = mkvar (i-n-1)
readback i (AppD k d)     = (readback i k) :@ (readback i d)
```

Semantic equations.

```
eval :: Term -> D -> D
eval (Lam t)     d = Ld (\d' -> eval t (PairD d d'))
eval (t :@ r)    d = (eval t d) `ap` (eval r d)
eval Q           d = pi2 d
eval (Sub t s)   d = eval t (evalS s d)

evalS :: Subst -> D -> D
evalS E           d = T
evalS Is          d = d
evalS (Ext s t)   d = PairD (evalS s d) (eval t d)
evalS P           d = pi1 d
evalS (Comp s s') d = (evalS s . evalS s') d
```

Normalisation by evaluation.

```
nbe :: Ctx -> Type -> Term -> Term
nbe ctx ty t = readback n . down ty $ eval t env
    where n   = length ctx
          env = mkenv n ctx T
```

Auxiliary functions.

```
mkenv :: Int -> Ctx -> D -> D
mkenv 0 []     d = d
mkenv n (t:ts) d = PairD d' $ up t (Vd (n-1))
    where d' = mkenv (n-1) ts d

mkvar :: Int -> Term
```

```
mkvar n | n <= 0    = Q
        | otherwise = Sub Q $ subs (n-1)

subs n | n <= 0    = P
subs n | otherwise = Comp P $ subs (n-1)

idsub :: Term -> Term -> Term
idsub t t' = Sub t (Ext Is t')
```

NbE for Martin-Löf Type Theory

3

In this chapter we adapt the NbE algorithm of the previous chapter to Martin-Löf type theory. To smooth the transition to dependent types we consider a calculus with dependent products and a universe of small types. As we said in Sec. 1.1 the dependent product $\mathsf{Fun}\ A\ B$ generalises the functional spaces $A \to B$ in that the type of an application can vary with the argument. Universes were first considered by Martin-Löf [83] for formalising category theory; basically a universe has term constructors for some of the type constructors of the calculus. The presentation of Martin-Löf type theory using GATs was first considered by Dybjer [51].

As was explained in Sec. 1.3, the type-checking algorithm for dependent types requires a decision procedure for equality. In this chapter we obtain such an algorithm by means of NbE. The other main result of this chapter is that Fun is injective in both arguments.

3.1 The calculus λ^{Π}

The presentation of λ^{Π} using GAT has the same sort symbols than $\lambda^{\to}$, but now types can depend on terms. This difference can be appreciated in the introductory rules for sorts. Notice that $\mathsf{Type}(\_)$ is not any more a fixed sort; now we have a set of well-formed types for each well-formed context.

$$\frac{}{\mathsf{Ctx}\ \text{is a sort}}\ (\textsc{ctx-sort}) \qquad \frac{\Gamma, \Delta \in \mathsf{Ctx}}{\Gamma \to \Delta\ \text{is a sort}}\ (\textsc{subs-sort})$$

$$\frac{\Gamma \in \mathsf{Ctx}}{\mathsf{Type}(\Gamma)\ \text{is a sort}}\ (\textsc{type-sort}) \qquad \frac{\Gamma \in \mathsf{Ctx} \qquad A \in \mathsf{Type}(\Gamma)}{\mathsf{Term}(\Gamma, A)\ \text{is a sort}}\ (\textsc{term-sort})$$

Introductory Rules

Contexts Several of the introductory rules for λ^{Π} also reflect the dependency of types on terms. For example, in (EXT-CTX) we requires A to be a well-formed type under Γ.

$$\frac{}{\diamond \vdash}\ (\textsc{empty-ctx}) \qquad \frac{\Gamma \vdash \qquad \Gamma \vdash A}{\Gamma.A \vdash}\ (\textsc{ext-ctx})$$

Substitutions The dependency of the sort $\mathsf{Type}(\_)$ on contexts means that substitutions should also act on types. For instance, in (EXT-SUBS) we apply σ to A in order to get a well-formed type under Γ; this is also reflected in the

categorical account of MLTT [51], if σ is a morphism in the category of contexts, then it acts both on types and on terms.

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathsf{id}_\Gamma : \Gamma}\ (\textsc{id-subs}) \qquad \frac{\Gamma \vdash}{\Gamma \vdash \langle\rangle : \diamond}\ (\textsc{emp-subs}) \qquad \frac{\Gamma \vdash A}{\Gamma.A \vdash \mathsf{p} : \Gamma}\ (\textsc{fst-subs})$$

$$\frac{\Gamma \vdash \delta : \Theta \qquad \Theta \vdash \sigma : \Delta}{\Gamma \vdash \sigma\,\delta : \Delta}\ (\textsc{comp-subs}) \qquad \frac{\Gamma \vdash \sigma : \Delta \qquad \Delta \vdash A \qquad \Gamma \vdash t : A\,\sigma}{\Gamma \vdash (\sigma, t) : \Delta.A}\ (\textsc{ext-subs})$$

Types Note that (U-EL) reflects terms with type U as types. The type $\mathsf{Fun}\,A\,B$ is the dependent function space: note that B is a type under $\Gamma.A$, so $B\,(\mathsf{id}, t)$ is a, possible distinct, type under Γ for each $\Gamma \vdash t : A$.

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathsf{U}}\ (\textsc{u-i}) \qquad \frac{\Gamma \vdash \qquad \Gamma \vdash A : \mathsf{U}}{\Gamma \vdash A}\ (\textsc{u-el})$$

$$\frac{\Gamma \vdash A \qquad \Gamma.A \vdash B}{\Gamma \vdash \mathsf{Fun}\,A\,B}\ (\textsc{fun-i}) \qquad \frac{\Delta \vdash A \qquad \Gamma \vdash \sigma : \Delta}{\Gamma \vdash A\,\sigma}\ (\textsc{subs-type})$$

Terms We see in the introductory rules for terms that the universe is closed under formation of dependent products. Rule (FUN-EL) makes explicit that $\mathsf{Fun}\,A\,B$ is a dependent product space. Rule (CONV) is another distinctive feature of dependent type systems; it manifests that types can be computed. It is because of this rule that we need a decision procedure for equality to define a type-checking algorithm.[1]

$$\frac{\Gamma \vdash A : \mathsf{U} \qquad \Gamma.A \vdash B : \mathsf{U}}{\Gamma \vdash \mathsf{Fun}\,A\,B : \mathsf{U}}\ (\textsc{fun-u-i}) \qquad \frac{\Gamma \vdash A}{\Gamma.A \vdash \mathsf{q} : A\,\mathsf{p}}\ (\textsc{hyp})$$

$$\frac{\Gamma \vdash A \qquad \Gamma.A \vdash B \qquad \Gamma.A \vdash t : B}{\Gamma \vdash \lambda t : \mathsf{Fun}\,A\,B}\ (\textsc{fun-i})$$

$$\frac{\Gamma \vdash A \qquad \Gamma.A \vdash B \qquad \Gamma \vdash t : \mathsf{Fun}\,A\,B \qquad \Gamma \vdash r : A}{\Gamma \vdash \mathsf{App}\ t\ r : B\,(\mathsf{id}_\Gamma, r)}\ (\textsc{fun-el})$$

$$\frac{\Delta \vdash A \qquad \Delta \vdash t : A \qquad \Gamma \vdash \sigma : \Delta}{\Gamma \vdash t\,\sigma : A\,\sigma}\ (\textsc{subs-term})$$

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash A = B}{\Gamma \vdash t : B}\ (\textsc{conv})$$

[1] In GATs the conversion rule (CONV) is an instance of a derived rule.

Axioms

We omit the premises for axioms. The premises for substitutions are similar to those for $\lambda^{\to}$. Omitting premises for the action of substitution on terms permits us to read the equalities both as axioms for types and as axioms for terms.

Substitutions

$$\begin{aligned} (\sigma\,\delta)\,\gamma &= \sigma\,(\delta\,\gamma) & \sigma\,\mathsf{id} &= \sigma \\ \mathsf{id}\,\sigma &= \sigma & \langle\rangle\,\sigma &= \langle\rangle \\ \mathsf{id}_{\diamond} &= \langle\rangle & \mathsf{p}\,(\sigma, t) &= \sigma \\ \mathsf{id}_{\Gamma.A} &= (\mathsf{p}, \mathsf{q}) & (\sigma, t)\,\delta &= (\sigma\,\delta, t\,\delta) \end{aligned}$$

Terms and types

$$\begin{aligned} \mathsf{U}\,\gamma &= \mathsf{U} & (\mathsf{Fun}\,A\,B)\,\sigma &= \mathsf{Fun}\,(A\,\sigma)\,(B\,(\sigma\,\mathsf{p}, \mathsf{q})) \\ t\,(\sigma\,\delta) &= (t\,\sigma)\,\delta & t\,\mathsf{id} &= t \\ \mathsf{q}\,(\sigma, t) &= t & (\lambda t)\,\sigma &= \lambda(t\,(\sigma\,\mathsf{p}, \mathsf{q})) \\ (\mathsf{App}\,r\,s)\,\sigma &= \mathsf{App}\,(r\,\sigma)\,(s\,\sigma) & \mathsf{App}\,(\lambda t)\,r &= t\,(\mathsf{id}, r) \end{aligned}$$

Congruence The only new congruences are those involving Fun (we present the one corresponding to Fun as a type constructor and skip the rule for its rôle as term constructor) and substitutions on types. Remember that these rules are obtained freely from the logical framework we are using.

$$\frac{\Gamma \vdash A = A' \qquad \Gamma.A \vdash B = B'}{\Gamma \vdash \mathsf{Fun}\,A\,B = \mathsf{Fun}\,A'\,B'}\ (\textsc{cong-fun}) \qquad \frac{\Delta \vdash A = A' \qquad \Gamma \vdash \sigma = \sigma' : \Delta}{\Gamma \vdash A\,\sigma = A'\,\sigma'}\ (\textsc{cong-subs})$$

Properties of λ^{Π}

Recall that we write $\Delta \leqslant^{i} \Gamma$ to indicate $\Delta \vdash \mathsf{p}^{i} : \Gamma$, cf. Rem. 1. Note that we can adapt Rem. 2, given a substitution δ such that $\Gamma \vdash \delta : \Delta.A$ one can prove $\Gamma \vdash \delta = (\mathsf{p}\,\delta, \mathsf{q}\,\delta) : \Delta.A$.

Remark 8. The following rule is derivable by (sym) and (trans). The same reasoning is valid for types.

$$\frac{\Gamma \vdash t = t' : A \qquad \Gamma \vdash t'' = t' : A}{\Gamma \vdash t = t'' : A}\ (\textsc{sym-trans})$$

The following inversion lemma is easily proved by induction on derivations; we show only one case of the proof, because all the other are analogous.

Lemma 14 (Inversion of typing judgements).

1. If $\Gamma \vdash \mathsf{Fun}\,A'\,B' : A$, then $\Gamma \vdash A = \mathsf{U}$, $\Gamma \vdash A' : \mathsf{U}$ and $\Gamma.A' \vdash B' : \mathsf{U}$;
2. If $\Gamma \vdash \mathsf{Fun}\,A\,B$, then $\Gamma \vdash A$ and $\Gamma.A \vdash B$.

3. If $\Gamma \vdash \lambda t \colon A$, then $\Gamma \vdash A = \mathsf{Fun}\, A'\, B'$ and $\Gamma.A' \vdash t \colon B'$;

4. If $\Gamma \vdash \mathsf{App}\, t\, r \colon A$, then $\Gamma \vdash A = B'\,(\mathsf{id}, r)$, $\Gamma \vdash t \colon \mathsf{Fun}\, A'\, B'$, and $\Gamma \vdash r \colon A'$.

Proof. Let us consider the last point. If $\Gamma \vdash \mathsf{App}\, t\, r \colon A$, it is clear that the last rule used can only be either (fun-el) or (conv). In the first case, the premises give $\Gamma \vdash t \colon \mathsf{Fun}\, A'\, B'$, $\Gamma \vdash r \colon A'$, and $\Gamma \vdash A = B'\,(\mathsf{id}_\Gamma, r)$ is obtained by (refl).

If the last rule used was (conv) from $\Gamma \vdash \mathsf{App}\, t\, r \colon C$ and $\Gamma \vdash C = A$, we use the i.h. on the first premise to get $\Gamma \vdash t \colon \mathsf{Fun}\, A'\, B'$, $\Gamma \vdash r \colon A'$, and $\Gamma \vdash C = B'\,(\mathsf{id}_\Gamma, r)$. By (sym-trans) we get $\Gamma \vdash A = B'\,(\mathsf{id}_\Gamma, r)$. □

The next lemma is a property of the meta-theory of GATs; to prove it directly we should add a new form of judgement for equality between contexts.

Lemma 15 (Inversion of equality).

1. If $\Gamma \vdash A = B$, then $\Gamma \vdash A$ and $\Gamma \vdash B$.

2. If $\Gamma \vdash t = t' \colon A$, then $\Gamma \vdash t \colon A$ and $\Gamma \vdash t' \colon A$.

Remark 9. The main result proved in this chapter is injectivity of Fun: if $\Gamma \vdash \mathsf{Fun}\, A\, B = \mathsf{Fun}\, A'\, B'$, then $\Gamma \vdash A = A'$ and $\Gamma.A \vdash B = B'$. This result cannot be obtained by induction on derivations: if the last rule used was (trans) on $\Gamma \vdash \mathsf{Fun}\, A\, B = C$ and $\Gamma \vdash C = \mathsf{Fun}\, A'\, B'$, then we cannot apply the inductive hypothesis.

As we said before, in λ^{Π} we have computation on types that can occur on typing derivations. This is the reason why we need to decide equality to define a type-checking algorithm. As we have seen for $\lambda^{\to}$, equality can be decided by normalising terms and checking for syntactical equality. As we want to decide equality for types, we extend the set of normal forms with types.

Definition 12 (Normal forms).

$$\begin{aligned} \mathrm{Ne} \ni k &::= q \mid q\, p^{i+1} \mid \mathsf{App}\, k\, v \\ \mathrm{Nf} \ni v, V, W &::= U \mid \mathsf{Fun}\, V\, W \mid \lambda v \mid k \ . \end{aligned}$$

3.2 Semantics and Normalisation by Evaluation

Following the pattern of Chap. 2 we first introduce an appropriate domain D for interpreting terms and then build an applicative structure on top of the domain. The lack of (η) allows us to use a set-theoretical model as in Sec. 2.2; since we do not η-expand terms, the families of functions $\downarrow_A$ and $\uparrow_A$ can be dispensed.

Domain semantics

Definition 13. We define D as the least solution for the following domain equation

$$D \approx \mathbb{O} \oplus D \times D \oplus \mathit{Var}_\perp \oplus [D \to D] \oplus D \times [D \to D] \oplus D \times D \oplus \{U\}_\perp \ .$$

Some of the components are the analogous as those of Def. (2.1): *Var* is a denumerable set of variables, neutrals values accounts for one copy of $D \times D$; environments will be modeled using the same sequences built from $\top$ and elements in $D \times D$. Elements of the lifted singleton $\{U\}_\perp$ and of $D \times [D \to D]$ will be used to represent types in the semantics.

Notation. We use for injections into D the same symbols for constructors in the syntax, so an element distinct from $\perp$ is written as:

$\top$	(d, d')	for $d, d' \in D$
$\mathsf{Var}\, i$	U	for $i \in \mathit{Var}$
$\mathsf{lam}\, f$	$\mathsf{Fun}\, d\, f$	for $d \in D$, and $f \in [D \to D]$
	$\mathsf{App}\, d\, d'$	for $d, d' \in D$

The readback function of the previous chapter should be extended to cover the new elements in the domain.

Definition 14 (Readback function).

$$\begin{aligned} \mathsf{R}_j\, U &= U \\ \mathsf{R}_j\, (\mathsf{Fun}\, X\, F) &= \mathsf{Fun}\, (\mathsf{R}_j\, X)\, (\mathsf{R}_{j+1}\, (F\, \mathsf{Var}\, j)) \\ \mathsf{R}_j\, (\mathsf{App}\, d\, d') &= \mathsf{App}\, (\mathsf{R}_j\, d)\, (\mathsf{R}_j\, d') \\ \mathsf{R}_j\, (\mathsf{lam}\, f) &= \lambda(\mathsf{R}_{j+1}\, f(\mathsf{Var}\, j)) \\ \mathsf{R}_j\, (\mathsf{Var}\, i) &= \begin{cases} q & \text{if } j \leqslant i+1 \\ q\, p^{j-(i+1)} & \text{if } j > i+1 \end{cases} \end{aligned}$$

Our model will be based on the reflection of neutral terms in D, as in Sec. 2.2. Intuitively, the sets of semantical neutral values Ne and semantical normal forms Nf have more elements than those of Sec. 2.2, because we have more normal forms.

Definition 15 (Semantical neutral values and normal forms).

$$\mathsf{Ne} = \bigcap_{i \in \mathbb{N}} \{d \in D \mid \mathsf{R}_i\, d \in \mathsf{Ne}\} \quad \text{and} \quad \mathsf{Nf} = \bigcap_{i \in \mathbb{N}} \{d \in D \mid \mathsf{R}_i\, d \in \mathsf{Nf}\}\ .$$

The application operator for λ^Π is the same as that of Sec. 2.2:

$$d \cdot e = \begin{cases} f\, e & \text{if } d = \mathsf{lam}\, f \\ \mathsf{App}\, d\, e & \text{if } d \in \mathsf{Ne} \\ \perp & \text{otherwise} \end{cases}\ .$$

The interpretation of expressions in this applicative structure is given by a pair of functions $[\![\_]\!]^t\_ \in \mathit{Terms} \to D \to D$ and $[\![\_]\!]^s\_ \in \mathit{Subs} \to D \to D$; they are shown in Fig. 3.1. In the following we omit the superscript from the semantic brackets. Remember the over-loading rôle of p and q as projection functions in D:

$$\mathsf{p}\, d = \begin{cases} d' & \text{if } d = (d', e) \\ \perp & \text{otherwise} \end{cases}\ .$$

$$
\begin{aligned}
[\![\mathsf{U}]\!]^t\,\mathsf{d} &= \mathsf{U} & [\![\langle\rangle]\!]^s\,\mathsf{d} &= \top \\
[\![\mathsf{Fun}\ \mathsf{A}\ \mathsf{B}]\!]^t\,\mathsf{d} &= \mathsf{Fun}\ ([\![\mathsf{A}]\!]^t\,\mathsf{d})\ (e \mapsto [\![\mathsf{B}]\!]^t(\mathsf{d}, e)) & [\![\mathsf{id}]\!]^s\,\mathsf{d} &= \mathsf{d} \\
[\![\mathsf{App}\ \mathsf{t}\ \mathsf{r}]\!]^t\,\mathsf{d} &= [\![\mathsf{t}]\!]^t\,\mathsf{d} \cdot [\![\mathsf{r}]\!]^t\,\mathsf{d} & [\![(\sigma, \mathsf{t})]\!]^s\,\mathsf{d} &= ([\![\sigma]\!]^s\,\mathsf{d}, [\![\mathsf{t}]\!]^t\,\mathsf{d}) \\
[\![\lambda\mathsf{t}]\!]^t\,\mathsf{d} &= \mathsf{lam}\,(\mathsf{d}' \mapsto [\![\mathsf{t}]\!]^t(\mathsf{d}, \mathsf{d}')) & [\![\mathsf{p}]\!]^s\,\mathsf{d} &= \mathsf{p}\ \mathsf{d} \\
[\![\mathsf{t}\ \sigma]\!]^t\,\mathsf{d} &= [\![\mathsf{t}]\!]^t([\![\sigma]\!]^s\,\mathsf{d}) & [\![\sigma\ \delta]\!]^s\,\mathsf{d} &= [\![\sigma]\!]^s([\![\delta]\!]^s\,\mathsf{d}) \\
[\![\mathsf{q}]\!]^t\,\mathsf{d} &= \mathsf{q}\ \mathsf{d}
\end{aligned}
$$

Figure 3.1: Semantics

Domain model

In $\lambda^{\to}$ we used the phrase "the denotation of a type A" to refer to the set of values that could be taken by terms with type A. In λ^{Π} the phrase may have an additional meaning: it can also refer to the element of D assigned by $[\![\mathsf{A}]\!]^t$. Our model is based on a *semantical universe* $T \subseteq \mathsf{D}$, which captures the latter meaning: if A is a type, then $[\![\mathsf{A}]\!]^t \in T$. For each element $\mathsf{d} \in T$ we have also a subset $[\mathsf{d}] \subseteq \mathsf{D}$ interpreting terms of the type denoted by d. In particular, $\mathsf{U} \in T$ and elements of $U = [\mathsf{U}]$ interpret small types. These subsets are introduced using the schema of inductive-recursive definition of [52].

Definition 16 (Universes for small and large types).

1. Subset of small types, U:
   - $\mathsf{Ne} \subseteq U$,
   - if $\mathsf{X} \in U$ and for all $\mathsf{d} \in [\mathsf{X}]$, $\mathsf{F}\ \mathsf{d} \in U$ then $\mathsf{Fun}\ \mathsf{X}\ \mathsf{F} \in U$.
2. Subset of types, T:
   - $U \subseteq T$
   - $\mathsf{U} \in T$,
   - if $\mathsf{X} \in T$, and for all $\mathsf{d} \in [\mathsf{X}]$, $\mathsf{F}\ \mathsf{d} \in T$, then $\mathsf{Fun}\ \mathsf{X}\ \mathsf{F} \in T$.
3. Family of subsets for types, $[\_] \in T \to \mathcal{P}(\mathsf{D})$:
   - $[\mathsf{U}] = U$
   - $[\mathsf{Fun}\ \mathsf{X}\ \mathsf{F}] = \{\mathsf{d} \mid \mathsf{d} \cdot e \in [\mathsf{F}\ e],\ \text{for all } e \in [\mathsf{X}]\}$
   - for all $\mathsf{d} \in \mathsf{Ne}$, $[\mathsf{d}] = \mathsf{Ne}$

Remark 10. There is a well-founded order on T: the minimal elements are U, and elements in Ne; $\mathsf{X} \sqsubset \mathsf{Fun}\ \mathsf{X}\ \mathsf{F}$, and for all $\mathsf{d} \in [\mathsf{X}]$, $\mathsf{F}\ \mathsf{d} \sqsubset \mathsf{Fun}\ \mathsf{X}\ \mathsf{F}$. This order is used to justify most of the following proofs by induction on T.

The next lemma shows that every set denoting types is saturated [78, Ch. 4]. If we compare this lemma with Lem. 4 we recognise a similar situation, we embed elements from [X] into Nf and project neutral values from Nf into [X]; the only difference is that now the embedding-projection pairs are identity functions, instead of $\downarrow\_\ \_$ and $\uparrow\_\ \_$.

Lemma 16 (Closure of semantic sub-sets).

1. $\mathsf{Ne} \subseteq T$ and for all $X \in T$, $\mathsf{Ne} \subseteq [X]$.
2. $T \subseteq \mathsf{Nf}$ and for all $X \in T$, $[X] \subseteq \mathsf{Nf}$.

Proof. We first prove simultaneously $X \in \mathsf{Nf}$ and $\mathsf{Ne} \subseteq [X] \subseteq \mathsf{Nf}$, for all $X \in U$. Since $\mathsf{Ne} \subseteq \mathsf{Nf}$ and $\mathsf{Ne} \in U$ are obvious, we only show $\mathsf{Fun}\, X'\, F \in \mathsf{Nf}$ if $\mathsf{Fun}\, X'\, F \in U$: By i.h. $X' \in \mathsf{Nf}$ and $\mathsf{Var}\, i \in [X']$, hence $F\, \mathsf{Var}\, i \in U$, and by i.h. $F\, \mathsf{Var}\, i \in \mathsf{Nf}$. In particular $R_i\, X' \in \mathsf{Nf}$ and $R_{i+1}\, (F\, \mathsf{Var}\, i) \in \mathsf{Nf}$, hence $R_i\, (\mathsf{Fun}\, X'\, F) \in \mathsf{Nf}$.

Finally we consider $X = \mathsf{Fun}\, X'\, F$ and show $\mathsf{Ne} \subseteq [\mathsf{Fun}\, X'\, F] \subseteq \mathsf{Nf}$. Given $d \in \mathsf{Ne}$ and $d' \in [X']$ we show $d \cdot d' \in [F\, d']$: by i.h. $d' \in \mathsf{Nf}$ and $d \cdot d' = \mathsf{App}\, d\, d'$ which is a neutral value, hence by i.h. on $\mathsf{Ne} \subseteq F\, d' \in U$ we conclude $d \in [\mathsf{Fun}\, X'\, F]$. The last step is to show $[\mathsf{Fun}\, X'\, F] \subseteq \mathsf{Nf}$. Note that an element of $[\mathsf{Fun}\, X'\, F]$ can be either a neutral or a function $\mathsf{lam}\, f$, otherwise we would have $\bot \in [F\, d']$ for each $d' \in [X']$; but that is not the case because by i.h. $[F\, d'] \subseteq \mathsf{Nf}$ and $\bot \notin \mathsf{Nf}$. Let us suppose $d = \mathsf{lam}\, f$: since $\mathsf{Var}\, i \in \mathsf{Ne}$, by i.h. $\mathsf{Var}\, i \in [X']$, so $f\, (\mathsf{Var}\, i) \in [F\, d']$, hence $f\, (\mathsf{Var}\, i) \in \mathsf{Nf}$; in particular $R_{i+1}\, (f\, (\mathsf{Var}\, i)) \in \mathsf{Nf}$, thus $R_i\, (\mathsf{lam}\, f) \in \mathsf{Nf}$.

To prove the lemma, we proceed by induction on $X \in T$: the case of $X \in \mathsf{Ne}$ is trivial; and the case of $X = \mathsf{Fun}\, X'\, F$ is dealt as above, in the proof of the lemma for $X = U$. □

An immediate consequence of this lemma is that normalisation consists, as in Sec. 2.2, just of the composition of evaluation (with a proper environment) and reification (with a proper index).

In the following definition we define simultaneously the notion of validity of judgements and the interpretation of well-formed contexts.

Definition 17 (Validity).

1. Contexts:
   - Validity: $\diamond \vDash$ always, and $\Gamma.A \vDash$ if and only if $\Gamma \vDash A$.
   - Interpretation: $[\![\diamond]\!] = \{\top\}$; and when $\Gamma.A \vDash$, then $[\![\Gamma.A]\!] = \{(d, d') \mid d \in [\![\Gamma]\!] \text{ and } d' \in [[\![A]\!]d]\}$.
2. Types: $\Gamma \vDash A$ if and only if $\Gamma \vDash$ and for all $d \in [\![\Gamma]\!]$, $[\![A]\!]d \in T$.
3. Terms: $\Gamma \vDash t : A$ if and only if $\Gamma \vDash A$ and for all $d \in [\![\Gamma]\!]$, $[\![t]\!]d \in [[\![A]\!]d]$.
4. Substitutions: $\Gamma \vDash \sigma : \Delta$ if and only if $\Gamma \vDash$, $\Delta \vDash$, and for all $d \in [\![\Gamma]\!]$, $[\![\sigma]\!]d \in [\![\Delta]\!]$.

We extend the definition of validity to equality judgements, for example $\Gamma \vDash t = t' : A$ iff $\Gamma \vDash t : A$, $\Gamma \vDash t' : A$, and $[\![t]\!]d = [\![t']\!]d$, for all $d \in [\![\Gamma]\!]$.

The following theorem states that this construction over the applicative structure models the calculus. We omit the proof of soundness which can be done by a tedious, but straightforward, induction on derivations.

Theorem 17 (Soundness). If $\Gamma \vdash J$, then $\Gamma \vDash J$. □

As before, soundness and Lem. 16 are the crucial results for having completeness of the normalisation algorithm, defined as the composition of reification and interpretation.

Corollary 18 (Completeness of normalisation). Let $\Gamma \vdash$ and $d \in [\![\Gamma]\!]$.

1. If $\Gamma \vdash A$, then $R_{|\Gamma|}\,([\![A]\!]d) \in \mathsf{Nf}$.
2. If $\Gamma \vdash A = A'$, then $R_{|\Gamma|}\,([\![A]\!]d) \equiv R_{|\Gamma|}\,([\![A']\!]d)$.
3. If $\Gamma \vdash t : A$, then $R_{|\Gamma|}\,([\![t]\!]d) \in \mathsf{Nf}$.
4. If $\Gamma \vdash t = t' : A$, then $R_{|\Gamma|}\,([\![t]\!]d) \equiv R_{|\Gamma|}\,([\![t']\!]d)$.

Proof. By Thm. 17, $[\![A]\!]d \in T$ and $[\![t]\!]d \in [[\![A]\!]d]$; by Lem. 16, $[\![t]\!]d \in \mathsf{Nf}$. □

3.3 Correctness of NbE via logical relations

As in the previous chapter, correctness of normalisation is proved using logical relations. For λ^{Π} we have, as before, a family of logical relations indexed by (semantical) types; we also have a new family of logical relations involving well-formed types and elements of T. This new logical relation is used to prove correctness of normalisation for types.

Definition 18 (Logical relations). There are two families of logical relations;

1. Let $\Gamma \vdash$, then $\Gamma \vdash \_ \sim \_ \in T \subseteq \{A \mid \Gamma \vdash A\} \times T$.
2. Let $\Gamma \vdash A$ and $X \in T$, then $\Gamma \vdash \_ : A \sim \_ \in X \subseteq \{t \mid \Gamma \vdash t : A\} \times [X]$.

- Neutral types: $X \in \mathsf{Ne}$.
  - $\Gamma \vdash A \sim X \in T$ if and only if for all $\Delta \leqslant^i \Gamma$, $\Delta \vdash A\,p^i = R_{|\Delta|}\,X$.
  - $\Gamma \vdash t : A \sim d \in X$ if and only if $\Gamma \vdash A \sim X \in T$, and for all $\Delta \leqslant^i \Gamma$, $\Delta \vdash t\,p^i = R_{|\Delta|}\,d : A\,p^i$.
- Universe.
  - $\Gamma \vdash A \sim \mathsf{U} \in T$ if and only if $\Gamma \vdash A = \mathsf{U}$.
  - $\Gamma \vdash t : A \sim d \in \mathsf{U}$ if and only if $\Gamma \vdash A = \mathsf{U}$, $\Gamma \vdash t \sim d \in T$, and for all $\Delta \leqslant^i \Gamma$, $\Delta \vdash t\,p^i = R_{|\Delta|}\,d : \mathsf{U}$.
- Function spaces.
  - $\Gamma \vdash A \sim \mathsf{Fun}\,X\,F \in T$ if and only if $\Gamma \vdash A = \mathsf{Fun}\,A'\,B$, and $\Gamma \vdash A' \sim X \in T$, and $\Delta \vdash B\,(p^i, s) \sim F\,d \in T$ for all $\Delta \leqslant^i \Gamma$ and all $\Delta \vdash s : A'\,p^i \sim d \in X$.
  - $\Gamma \vdash t : A \sim d \in \mathsf{Fun}\,X\,F$ if and only if $\Gamma \vdash A = \mathsf{Fun}\,A'\,B$, $\Gamma \vdash A' \sim X \in T$, and for all $\Delta \leqslant^i \Gamma$ and $\Delta \vdash s : A'\,p^i \sim d' \in X$, $\Delta \vdash \mathsf{App}\,(t\,p^i)\,s : B\,(p^i, s) \sim d \cdot d' \in F\,d'$;
    - if $d = \mathsf{lam}\,f$, then $\Gamma \vdash t = \lambda\,t' : A$,
    - if $d \in \mathsf{Ne}$, then $\Delta \vdash (t\,p^i) = R_{|\Delta|}\,d : A\,p^i$.

Note that the definition of the logical relation for dependent products is a bit contrived; in fact, logical relations of functional types are usually defined using the logical relations of the lower types involved, as in Sec. 2.4. For this calculus, however, we need to impose additional conditions on terms — basically, that functions in the semantics are related only with abstractions, otherwise it is not clear how to prove Lem. 21.

The following lemma shows that judgemental equality preserves the logical relation; since we have defined the logical relations using definitional equality, this lemma is easily proved.

Lemma 19 (Preservation by judgemental equality). Let $\Gamma \vdash A = A'$ and $\Gamma \vdash A \sim X \in T$.

1. $\Gamma \vdash A' \sim X \in T$; and
2. if $\Gamma \vdash t = t' \colon A$ and $\Gamma \vdash t \colon A \sim d \in X$, then $\Gamma \vdash t' \colon A' \sim d \in X$.

Proof. By induction on $X \in T$. Every case boils down to get the relevant part of the hypothesis and using (sym-trans) and congruence for getting the key equality judgement in the definition of the relation. We show only some cases. For types, we consider $X = \mathsf{Fun}\, X'\, F$: by hypothesis we know $\Gamma \vdash A = \mathsf{Fun}\, B\, C$ and $\Gamma \vdash A = A'$, thus by (sym-trans) on types $\Gamma \vdash A' = \mathsf{Fun}\, B\, C$. All the other conditions are stated in terms of B and C, so $\Gamma \vdash A' \sim \mathsf{Fun}\, X'\, F \in T$ is immediate.

For terms, we also consider $X = \mathsf{Fun}\, X'\, F$: as before we can extract $\Gamma \vdash A' = \mathsf{Fun}\, B\, C$ by (sym-trans) from the hypothesis $\Gamma \vdash t \colon A \sim d \in \mathsf{Fun}\, X'\, F$; then we make case analysis on $d \in [\mathsf{Fun}\, X'\, F]$. If $d = \mathsf{lam}\, f$, by hypothesis $\Gamma \vdash t = \lambda\, t'' \colon A$; then by (sym-trans) and conversion $\Gamma \vdash t' = \lambda\, t'' \colon A'$. The other possibility for $d \in [\mathsf{Fun}\, X'\, F]$ is that $d \in \mathsf{Ne}$: let $\Delta \leqslant^i \Gamma$, $\Delta \vdash s \colon B\, p^i \sim d' \in X'$; then by (sym-trans) $\Delta \vdash t'\, p^i = R_{|\Delta|}\, d \colon A\, p^i$. By congruence $\Delta \vdash \mathsf{App}\, (t\, p^i)\, s = \mathsf{App}\, (t'\, p^i)\, s \colon C\, (p^i, s)$, so by i.h. on $\Delta \vdash \mathsf{App}\, (t\, p^i)\, s \colon C\, (p^i, s) \sim \mathsf{App}\, d\, d' \in F\, d'$, we conclude $\Delta \vdash \mathsf{App}\, (t'\, p^i)\, s \colon C\, (p^i, s) \sim \mathsf{App}\, d\, d' \in F\, d'$. □

Lemma 20 (Monotonicity). Let $\Delta \leqslant^i \Gamma$ then

1. If $\Gamma \vdash A \sim X \in T$, then $\Delta \vdash A\, p^i \sim X \in T$; and
2. If $\Gamma \vdash t \colon A \sim d \in X$, then $\Delta \vdash t\, p^i \colon A\, p^i \sim d \in X$.

Proof. By induction on $X \in T$. We will show first the proof of the first point, and then the proof of the second point.

Types. We consider only the case of neutrals and function spaces. For $X \in \mathsf{Ne}$: Let $\Gamma \vdash A \sim X \in T$ and $\Delta \leqslant^i \Gamma$. For $\Theta \leqslant^j \Delta$, we have $\Theta \leqslant^{i+j} \Gamma$, by definition of the log. rel. for types, $\Theta \vdash A\, p^{i+j} = R_{|\Theta|}\, X$. Since $\Theta \vdash A\, p^{i+j} = (A\, p^i)\, p^j$, we have $\Theta \vdash (A\, p^i)\, p^j = R_{|\Theta|}\, X$; thus $\Delta \vdash A\, p^i \sim X \in T$.

Let us consider now the case $X = \mathsf{Fun}\, X'\, F$: let $\Gamma \vdash A \sim \mathsf{Fun}\, X'\, F \in T$ and $\Delta \leqslant^i \Gamma$. We first prove that $A\, p^i = \mathsf{Fun}\, B_1\, C_1$ for some B_1 and C_1: note that we can get $\Delta \vdash A\, p^i = \mathsf{Fun}\, (B\, p^i)\, (C\, (p^i\, p, q))$ from the hypothesis after applying congruence under weakening. The second point, $\Delta \vdash B\, p^i \sim X' \in T$, follows from the induction hypothesis for $\Gamma \vdash B \sim X' \in T$. Finally, let $\Theta \leqslant^j \Delta$ and $\Theta \vdash s \colon B\, p^i\, p^j \sim d' \in X'$, by Lem. 19 $\Theta \vdash s \colon B\, p^{i+j} \sim d' \in X'$; thus by def. we have $\Theta \vdash C\, (p^{i+j}, s) \sim F\, d' \in T$; and using Lem. 19 again we conclude $\Theta \vdash C\, (p^i\, p, q)\, (p^j, s) \sim F\, d' \in T$.

Terms. We only consider the case for $X = \mathsf{Fun}\ X'\ F$, because the other cases are similar to the cases shown for types. Let $\Gamma \vdash t : A \sim d \in \mathsf{Fun}\ X'\ F$ and $\Delta \leqslant^i \Gamma$. As before, from $\Gamma \vdash A = \mathsf{Fun}\ B\ C$ we get by congruence $\Delta \vdash A\ p^i = \mathsf{Fun}\ (B\ p^i)\ (C\ (p^{i+1}, q))$, and by i.h. $\Delta \vdash B\ p^i \sim X' \in T$. Let $\Theta \leqslant^j \Delta$ and $\Theta \vdash s : B\ p^i\ p^j \sim d' \in X'$; by Lem. 19 we have $\Theta \vdash s : B\ p^{i+j} \sim d' \in X'$; thus by def. $\Theta \vdash \mathsf{App}\ (t\ p^{i+j})\ s : C\ (p^{i+j}, s) \sim d \cdot d' \in F\ d'$; and using Lem. 19 again we conclude $\Theta \vdash \mathsf{App}\ ((t\ p^j)\ p^j)\ s : C\ (p^i\ p, q)\ (p^j, s) \sim d \cdot d' \in F\ d'$. Finally we do case analysis on $d \in [\mathsf{Fun}\ X'\ F]$. If $d = \mathsf{lam}\ f$, then $\Gamma \vdash t = \lambda t' : A$, hence by congruence $\Theta \vdash t = \lambda(t'\ (p^i\ p, q)) : A$. If $d \in \mathsf{Ne}$, then we have $(t\ p^i)\ p^j = t\ p^{i+j}$, thus $\Theta \vdash (t\ p^i)\ p^j = R_{|\Theta|}\ d : (A\ p^i)\ p^j$, from def. of logical relations. □

The next lemma (sometimes called "in-out" lemma) is almost what we need to finish the proof of soundness; in fact, the only remaining step is to show that each term is logically related to its denotation.

Lemma 21 (Derivability of equality of reified elements). Let $\Gamma \vdash A \sim X \in T$, and $\Gamma \vdash t : A \sim d \in X$, then

1. $\Gamma \vdash A = R_{|\Gamma|}\ X$,
2. $\Gamma \vdash t = R_{|\Gamma|}\ d : A$; and
3. if $k \in \mathsf{Ne}$ and for all $\Delta \leqslant^i \Gamma$, $\Delta \vdash t\ p^i = R_{|\Delta|}\ k : A\ p^i$, then $\Gamma \vdash t : A \sim k \in X$.

Proof. By induction on $X \in T$. The base cases are obtained immediately from the definition. So we show only the proofs for $X = \mathsf{Fun}\ X'\ F$.

1. From $\Gamma \vdash A \sim \mathsf{Fun}\ X'\ F \in T$ we know $\Gamma \vdash A = \mathsf{Fun}\ B\ C$, $\Gamma \vdash B \sim X' \in T$, and $\Gamma.B \vdash C \sim F\,\mathsf{Var}\,|\Gamma| \in T$; thus by i.h. we get $\Gamma \vdash B = R_{|\Gamma|}\ X'$ and $\Gamma.B \vdash C = R_{|\Gamma.B|}\ (F\,\mathsf{Var}\,|\Gamma|)$. Hence by congruence, $\Gamma \vdash \mathsf{Fun}\ B\ C = \mathsf{Fun}\ (R_{|\Gamma|}\ X)\ (R_{|\Gamma|+1}\ (F\,\mathsf{Var}\,|\Gamma|))$; and by def. of readback $\Gamma \vdash \mathsf{Fun}\ B\ C = R_{|\Gamma|}\ (\mathsf{Fun}\ X\ F)$.

2. We show only the case for $X = \mathsf{Fun}\ X'\ F$ and $d = \mathsf{lam}\ f$ (the case when $d \in \mathsf{Ne}$ is immediate from the definition). Let $\Gamma \vdash t : A \sim \mathsf{lam}\ f \in \mathsf{Fun}\ X'\ F$. By i.h. on the third point $\Gamma.B \vdash q : B\ p \sim \mathsf{Var}\,|\Gamma| \in X'$. From the def. of the logical relations $\Gamma.B \vdash \mathsf{App}\ ((\lambda\, t')\ p)\ q : C\ (p, q) \sim f\,\mathsf{Var}\,|\Gamma| \in F\ (\mathsf{Var}\,|\Gamma|)$ and by i.h.

$$\Gamma.B \vdash \mathsf{App}\ ((\lambda\, t')\ p)\ q = R_{|\Gamma|+1}\ (f\ (\mathsf{Var}\,|\Gamma|)) : C\ (p, q)\ ;$$

   by trans. and conversion $\Gamma.B \vdash t' = R_{|\Gamma|+1}\ f\,\mathsf{Var}\,|\Gamma| : C$. By def. of readback $R_{|\Gamma|+1}\ (f\ (\mathsf{Var}\,|\Gamma|)) = R_{|\Gamma|}\ (\mathsf{lam}\ f)$, hence by congruence $\Gamma \vdash \lambda\, t' = R_{|\Gamma|}\ (\mathsf{lam}\ f) : \mathsf{Fun}\ B\ C$.

3. let $k \in \mathsf{Ne}$ and $\Delta \vdash t\ p^i = R_{|\Delta|}\ k : A\ p^i$, for all $\Delta \leqslant^i \Gamma$. The only point to show is $\Delta \vdash \mathsf{App}\ (t\ p^{i+j})\ s : C\ (p^{i+j}, s) \sim \mathsf{App}\ k\ d \in F\ d$, for all $\Delta \leqslant^j \Gamma$ and $\Delta \vdash s : B\ p^{i+j} \sim d \in X'$. We will use the i.h. after showing $\Theta \vdash (\mathsf{App}\ (t\ p^i)\ s)\ p^j = R_{|\Theta|}\ (\mathsf{App}\ k\ d) : C\ (p^i, s)\ p^j$, for all $\Theta \leqslant^j \Delta$.

   Note that we have $\Theta \vdash t\ p^{i+j} = R_{|\Theta|}\ k : A\ p^{i+j}$ and, by Lem. 20 and i.h. $\Theta \vdash s\ p^j = R_{|\Theta|}\ d : B\ p^{i+j}$. Hence by congruence $\Theta \vdash (\mathsf{App}\ (t\ p^i)\ s)\ p^j = \mathsf{App}\ (R_{|\Theta|}\ k)\ (R_{|\Theta|}\ d) : C\ (p^i, s)\ p^j$. We can apply the i.h. on $\mathsf{App}\ k\ d \in \mathsf{Ne}$, because by definition $\mathsf{App}\ (R_{|\Theta|}\ k)\ (R_{|\Theta|}\ d) = R_{|\Theta|}\ (\mathsf{App}\ k\ d)$. □

To prove that each term is related to its denotation, we need the more general result that logical relations are preserved under substitutions.

Definition 19 (Logical relations for substitutions). Given two well-formed contexts $\Gamma \vdash$ and $\Delta \vdash$, then $\Gamma \vdash \_ : \Delta \sim \_ \in [\![\Delta]\!] \subseteq \{\sigma \mid \Gamma \vdash \sigma : \Delta\} \times [\![\Delta]\!]$. By induction on Δ we define:

- $\Gamma \vdash \sigma : \diamond \sim d \in [\![\diamond]\!]$.
- $\Gamma \vdash (\sigma, t) : \Delta.A \sim (d, d') \in [\![\Delta.A]\!]$ if and only if $\Gamma \vdash \sigma : \Delta \sim d \in [\![\Delta]\!]$, $\Delta \vdash A \sim [\![A]\!]d \in T$, and $\Gamma \vdash t : A\,\sigma \sim d' \in [\![A]\!]d$.

Lemma 22 (Preservation by judgemental equality). If $\Gamma \vdash \gamma = \delta : \Delta$, and $\Gamma \vdash \gamma : \Delta \sim d \in [\![\Delta]\!]$, then $\Gamma \vdash \delta : \Delta \sim d \in [\![\Delta]\!]$.

Lemma 23 (Monotonicity). If $\Gamma \vdash \delta : \Delta \sim d \in [\![\Delta]\!]$, then for any $\Theta \leqslant^i \Gamma$, $\Theta \vdash \delta\,p^i : \Delta \sim d \in [\![\Delta]\!]$.

Theorem 24 (Fundamental theorem of logical relations).

1. if $\Delta \vdash A$ and $\Gamma \vdash \delta : \Delta \sim d \in [\![\Delta]\!]$, then $\Gamma \vdash A\,\delta \sim [\![A]\!]d \in T$;
2. if $\Delta \vdash t : A$ and $\Gamma \vdash \delta : \Delta \sim d \in [\![\Delta]\!]$, then $\Gamma \vdash t\,\delta : A\,\delta \sim [\![t]\!]d \in [\![A]\!]d$; and
3. if $\Theta \vdash \gamma : \Gamma$ and $\Gamma \vdash \delta : \Delta \sim d \in [\![\Delta]\!]$, then $\Theta \vdash \gamma\,\delta : \Delta \sim [\![\gamma]\!]d \in [\![\Delta]\!]$.

Proof. By induction on the derivations. We note that for terms we show only some cases when the last rule used was the introductory rule; the case when the last rule used was the conversion rule, we can conclude by i.h., and Lem. 19.

1. For types we show the case for $\Delta \vdash \mathsf{Fun}\,A\,B$. It is clear that we can apply the congruence for Fun to deduce $\Gamma \vdash (\mathsf{Fun}\,A\,B)\,\delta = \mathsf{Fun}\,(A\,\delta)\,(B\,(\delta\,p, q))$, and by inversion and i.h. $\Gamma \vdash A\,\delta \sim [\![A]\!]d \in T$. Let $\Theta \leqslant^i \Delta$ and $\Theta \vdash s : (A\,\delta)\,p^i \sim e \in [\![A]\!]d$, then $\Theta \vdash (\delta\,p^i, s) : \Gamma.A \sim (d, e) \in [\![\Gamma.A]\!]$, hence by i.h. $\Theta \vdash B\,(\delta\,p^i, s) \sim [\![B]\!](d, e) \in T$; thus by Lem. 19, $\Theta \vdash (B\,(\delta\,p, q))\,(p^i, s) \sim [\![B]\!](d, e) \in T$.
2. For terms we only show the proof for rules (FUN-I), (FUN-EL), (HYP), and (SUBS-TERM).
   a) Let $\Delta \vdash \lambda t : \mathsf{Fun}\,A\,B$. As before, $\Gamma \vdash A\,\delta \sim [\![A]\!]d \in T$, and for $\Theta \leqslant^i \Delta$ and $\Theta \vdash s : (A\,\delta)\,p^i \sim e \in [\![A]\!]d$, by i.h. on $\Delta.A \vdash t : B$, we have $\Theta \vdash t\,(\delta\,p^i, s) : B\,(\delta\,p^i, s) \sim [\![t]\!](d, e) \in [\![B]\!](d, e)$. Note that $[\![t]\!](d, e) = [\![\lambda t]\!]d \cdot e$ and $t\,(\delta\,p^i, s) = \mathsf{App}\,\lambda(t\,(\delta\,p^i, q))\,s$, hence by Lem. 19, we have $\Theta \vdash \mathsf{App}\,((\lambda t)\,\delta\,p^i)\,s : B\,(\delta\,p^i, s) \sim [\![\lambda t]\!]d \cdot e \in ([\![B]\!]d)\,e$.
   b) Let $\Delta \vdash \mathsf{App}\,t\,r : B\,(\mathsf{id}, r)$. This case is easy. By inversion and i.h. $\Gamma \vdash r\,\delta : A\,\delta \sim [\![r]\!]d \in [\![A]\!]d$ and $\Gamma \vdash t\,\delta : (\mathsf{Fun}\,A\,B)\,\delta \sim [\![t]\!]d \in [\![\mathsf{Fun}\,A\,B]\!]d$. By def. of log. rel. $\Delta \vdash \mathsf{App}\,(t\,\delta)\,(r\,\delta) : B\,(\delta, r\,\delta) \sim ([\![t]\!]d \cdot [\![r]\!]d) \in [\![B]\!](d, [\![r]\!]d)$; and by Lem. 19, we conclude $\Delta \vdash (\mathsf{App}\,t\,r)\,\delta : (B\,(\mathsf{id}, r))\,\delta \sim [\![\mathsf{App}\,t\,r]\!]d \in [\![B\,(\mathsf{id}, r)]\!]d$.
   c) Let $\Delta.A \vdash q : A\,p$. From Rem. 13 we know $\delta = (\gamma, t)$, and from def. of log. rel. for substitutions $d = (e, e')$, $\Gamma \vdash \gamma : \Delta \sim e \in [\![\Delta]\!]$, and $\Gamma \vdash t : A\,\delta \sim e' \in [\![A]\!]e$. By Lem. 19, $\Gamma \vdash q\,\delta : A\,p \sim [\![q]\!]d \in [\![A\,p]\!]d$.

d) Let $\Delta \vdash t\,\sigma : A\,\sigma$, with $\Delta \vdash \sigma : \Theta$. By i.h. $\Delta \vdash \sigma\,\delta : \Theta \sim [\![\sigma]\!]d \in [\![\Theta]\!]$ and $\Delta \vdash t\,\sigma\,\delta : A\,\sigma\,\delta \sim [\![t]\!]([\![\sigma]\!]d) \in [\![A]\!]([\![\sigma]\!]d)$.

3. All the cases for substitutions are simple and similar to cases already considered. We show the case of (ext-subs), just for illustration. Let $\Delta \vdash (\gamma, t) : \Theta.A$. By i.h. $\Gamma \vdash \gamma\,\delta : \Theta \sim [\![\gamma]\!]d \in [\![\Theta]\!]$ and $\Gamma \vdash t\,\delta : (A\,\gamma)\,\delta \sim [\![t]\!]d \in [\![A\,\gamma]\!]d$; so $\Gamma \vdash (\gamma\,\delta, t\,\delta) : \Theta.A \sim [\![(\gamma, t)]\!]d \in [\![\Theta.A]\!]$ and, by Rem. 22, we conclude $\Gamma \vdash (\gamma, t)\,\delta : \Theta.A \sim [\![(\gamma, t)]\!]d \in [\![\Theta.A]\!]$. □

Now we can define a function mapping each context to an element of the semantic domain; it is, clearly, logically related with the identity substitution. Hence, to get soundness we can instantiate Thm. 24 and then apply Lem. 21.

Definition 20 (Canonical element of a context and Normalisation function). Let $\Gamma \vdash$ and $n = |\Gamma|$.

$$\mathbf{nbe}_\Gamma(\_) = \mathsf{R}_{|\Gamma|}\,([\![\_]\!]\rho_\Gamma)\ \text{, where}$$

$$\rho_\Gamma = ((\ldots(\top, \mathsf{Var}\,0), \ldots), \mathsf{Var}\,(n-1)))\ .$$

Soundness of normalisation, and the uniqueness of normal forms, was already shown by Martin-Löf in [80], where he credits Peter Hancock for the proof method.

Theorem 25 (Soundness of NbE). Let $\Gamma \vdash A$, and $\Gamma \vdash t : A$; then $\Gamma \vdash A = \mathbf{nbe}_\Gamma(A)$ and $\Gamma \vdash t = \mathbf{nbe}_\Gamma(t) : A$.

Proof. By Thm. 24 and Lem. 21. □

From soundness and completeness of NbE we obtain a decision procedure for judgemental equality for well-formed types and well-typed terms.

Corollary 26 (Decision procedure for judgemental equality). Given $\Gamma \vdash A$ and $\Gamma \vdash A'$ we can decide if $\Gamma \vdash A = A'$. Also, if $\Gamma \vdash t : A$ and $\Gamma \vdash t' : A$ we can decide if $\Gamma \vdash t = t' : A$.

Another consequence of soundness is the injectivity of the constructor Fun. As we said in Rem. 9, this cannot be proved by induction on derivations.

Corollary 27 (Injectivity of $\mathsf{Fun}\,\_\,\_$). If $\Gamma \vdash \mathsf{Fun}\,A\,B = \mathsf{Fun}\,A'\,B'$, then $\Gamma \vdash A = A'$ and $\Gamma.A \vdash B = B'$.

Proof. Assume $\Gamma \vdash \mathsf{Fun}\,A\,B = \mathsf{Fun}\,A'\,B'$; then, by Lem. 15 we know $\Gamma \vdash \mathsf{Fun}\,A\,B$; by Thm. 24, and Lem. 19,

$$\Gamma \vdash \mathsf{Fun}\,A\,B \sim [\![\mathsf{Fun}\,A\,B]\!]\rho_\Gamma \in T\ . \tag{3.1}$$

From the definition of logical relations we deduce $\Gamma \vdash A \sim [\![A]\!]\rho_\Gamma \in T$ and $\Gamma.A \vdash B \sim [\![B]\!]\rho_{\Gamma.A} \in T$; by Lem. 21 $\Gamma \vdash A = \mathsf{R}_{|\Gamma|}\,([\![A]\!]\rho_\Gamma)$ and $\Gamma.A \vdash B = \mathsf{R}_{|\Gamma.A|}\,([\![B]\!]\rho_{\Gamma.A})$. By Lem. 19 on (3.1) and $\Gamma \vdash \mathsf{Fun}\,A\,B = \mathsf{Fun}\,A'\,B'$ we get $\Gamma \vdash \mathsf{Fun}\,A'\,B' \sim [\![\mathsf{Fun}\,A\,B]\!]\rho_\Gamma \in T$; and by the same reasoning as before $\Gamma \vdash A = \mathsf{R}_{|\Gamma|}\,([\![A']\!]\rho_\Gamma)$ $\Gamma.A \vdash B = \mathsf{R}_{|\Gamma.A|}\,([\![B']\!]\rho_{\Gamma.A})$. Finally we can apply (sym) and (trans) to conclude $\Gamma \vdash A = A'$ and $\Gamma.A \vdash B = B'$. □

3.4 Implementation of a type-checker for λ^{Π}

In this appendix we present an implementation of the algorithms for deciding equality and type-checking. The decision procedure consists of checking the syntactical equality of the normal forms got by applying the nbe function to both terms. Even though the normalisation function is defined independently of type-checking, it can be applied safely only on well-typed terms (or well-formed types); so checking equality requires to type-check terms.

The syntax of the calculus

```
data Term  =  U
           |  Fun Term Term
           |  Subs Term Subst
           |  Abs Term
           |  App Term Term
           |  Var
            deriving Eq

data Subst  =  Empty
            |  IdSub
            |  Weak
            |  ExtSub Subst Term
            |  Comp Subst Subst
             deriving Eq



type Ctx = [Term]



var    :: Int -> Term
var 0  = Var
var n  = Subs Var (weak (n-1))



weak   :: Int -> Subst
weak 0 = Weak
weak n = Comp (weak (n-1)) Weak



sub :: Term -> Subst
sub = ExtSub IdSub
```

The Semantic Domain and Reification

```
data D  = Top
        | Pair D D
        | VarD Int
        | AppD D D
        | UD
        | Lam (D -> D)
        | FunD D (D -> D)
```

```
readback :: Int -> D -> Term
readback j (VarD i)    = var (j - (i+1))
readback j (AppD d e)  = App (readback j d) (readback j e)
readback _ UD          = U
readback j (Lam f)     = Abs $ readback (j+1) (f (VarD j))
readback j (FunD d f)  = Fun (readback j d) (readback (j+1) (f (VarD j)))
```

The applicative structure

```
app :: D -> D -> D
app (Lam f)       e = f e
app d@(VarD _)    e = AppD d e
app d@(AppD _ _)  e = AppD d e
app _             _ = undefined



proj1,proj2 :: D -> D
proj1 (Pair d _) = d
proj1 _          = undefined
proj2 (Pair _ e) = e
proj2 _          = undefined
```

The interpretation

```
type Env = D

evalTm :: Term -> Env -> D
evalTm U          d = UD
evalTm (Fun a b)  d = FunD (evalTm a d) (\e -> evalTm b (Pair d e))
evalTm (Subs t s) d = evalTm t (evalSubst s d)
evalTm (Abs t)    d = Lam (\e -> evalTm t (Pair d e))
evalTm (App t t') d = evalTm t d `app` evalTm t' d
evalTm Var        d = proj2 d



evalSubst :: Subst -> Env -> D
evalSubst Empty        d = Top
evalSubst IdSub        d = d
evalSubst Weak         d = proj1 d
evalSubst (ExtSub s t) d = Pair (evalSubst s d) (evalTm t d)
evalSubst (Comp s s')  d = evalSubst s (evalSubst s' d)
```

Normalisation By Evaluation

```
env :: Ctx -> (Env, Int)
env []      = (Top,0)
env (_:ctx) = (Pair d (VarD n),n+1)
    where       (d , n ) = env ctx



nbe :: Ctx -> Term -> Term
nbe ctx tm = readback n (evalTm tm d)
    where (d,n) = env ctx



eqCheck :: Ctx -> Term -> Term -> Bool
eqCheck ctx a b = nbe ctx a == nbe ctx b
```

Bi-directional type-checking algorithm

The type-checking algorithm consists in three functions: tyCheck decides the well-formedness of a type, in normal form, under a (well-formed) context; the checking that a term, in normal form, has a given type under a context is done by tmCheck; finally, infTy infers a type for a neutral term under a given context. Note that infTy can fail, but when it succeeds the inferred type is well-formed, so it is safe to use nbe to decide its equality with another well-formed type.

```
tyCheck :: Ctx -> Term -> Bool
tyCheck _    U          =  True
tyCheck ctx (Fun a b)   =  tyCheck ctx a && tyCheck (a:ctx) b
tyCheck ctx k           =  neutral k && tmCheck ctx U k



tmCheck :: Ctx -> Term -> Term -> Bool
tmCheck ctx U         (Fun a b)   =  tmCheck ctx U a && tmCheck (a:ctx) U b
tmCheck ctx (Fun a b) (Abs t)     =  tmCheck (a:ctx) b t
tmCheck ctx b         k           =  neutral k &&
                                     maybe False (eqCheck ctx b) (infTy ctx k)



infTy :: Ctx -> Term -> Maybe Term
infTy (a:ctx) Var       = Just a
infTy (_:ctx) (Subs Var sigma) = tvar sigma >>= \a -> Just $ Subs a Weak
         where    tvar :: Subst -> Maybe Term
                  tvar Weak               =  infTy ctx Var
                  tvar(Comp sigma' Weak)  =  infTy ctx (Subs Var sigma')
                  tvar _                  =  Nothing
infTy ctx     (App k t) = infTy ctx k >>= checkArg
    where  checkArg :: Term -> Maybe Term
           checkArg (Fun a b) =  if  tmCheck ctx a t
                                 then Just $ Subs b (sub t)
                                 else Nothing
           checkArg _         = Nothing
infTy _ _ =  Nothing



neutral :: Term -> Bool
neutral Var           =  True
neutral (Subs Var _)  =  True
neutral (App _ _)     =  True
neutral _             =  False
```

Martin-Löf Type Theory with Singleton Types and Proof-Irrelevance

4

In this chapter we extend the type-checking algorithm for Martin-Löf type theory by considering, besides dependent products and a universe, singleton types, proof-irrelevant propositions, sigma types, natural numbers, and enumeration types. The method of the previous chapter can be adapted in a modular way to these extensions.

Singleton types were introduced by Aspinall [14] in the context of specification languages. An important use of singletons is as definitions by abbreviations (see [14, 42]); they were also used to model translucent sums in the formalisation of SML [75]. It is interesting to consider singleton types because beta-eta phase separation fails: one cannot do eta-expansion before beta-normalisation of types because the shape of the types at which to eta-expand is still unknown at this point; and one cannot postpone eta-expansion after beta-normalisation, because eta-expansion at singleton type can trigger new beta-reductions. Stone and Harper [109] decide type checking in a logical framework (LF) with singleton types and subtyping. Yet it is not clear whether their method extends to computation on the type level. As far as we know, our work is the first where singleton types are considered together with a universe.

De Bruijn proposed the concept of *irrelevance of proofs* [32], for reducing the burden in the formalisation of mathematics. As shown by Werner [116], the use of proof-irrelevance types together with sigma types is one way to get subset types à la PVS [103] in type-theories having the eta rule. This style of subset types was also explored by Sozeau [107, Sec. 3.3]; for another presentation of subset types in Martin-Löf type-theory see [101]. Berardi conjectured that (impredicative) type-theory with proof-irrelevance is equivalent to constructive mathematics [24].

4.1 The Calculus

In this section, we introduce the type theory. In order to show the modularity of our approach, we present it as two calculi λ^{Sing} and λ^{Irr}: the first one has dependent function spaces, singleton types, and a universe closed under function spaces and singletons. In the second calculus we leave out singleton types and we add natural numbers, sigma types, and proof-irrelevant propositions. It is not clear if singleton types can be combined with proof-irrelevant propositions without turning the system inconsistent.

Calculus λ^{Sing} with singleton types

We omit introductory rules for sorts because they are the same as those of Chap. 3. We repeat introductory rules for contexts and substitutions although they also remain unchanged.

In the following, whenever a rule has a hypothesis $A \in \mathsf{Type}(\Gamma)$, then $\Gamma \in \mathsf{Ctx}$ shall be a further, implicit hypothesis. Similarly, $\sigma \in \Gamma \to \Delta$ presupposes $\Gamma \in \mathsf{Ctx}$ and $\Delta \in \mathsf{Ctx}$, and $t \in \mathsf{Term}(\Gamma, A)$ presupposes $A \in \mathsf{Type}(\Gamma)$, which in turn presupposes $\Gamma \in \mathsf{Ctx}$.

Contexts and Substitutions The operators for contexts and substitutions are the same as those of λ^{Π}.

$$\frac{}{\diamond \in \mathsf{Ctx}}\ (\textsc{empty-ctx}) \qquad \frac{\Gamma \in \mathsf{Ctx} \qquad A \in \mathsf{Type}(\Gamma)}{\Gamma.A \in \mathsf{Ctx}}\ (\textsc{ext-ctx})$$

$$\frac{\Gamma \in \mathsf{Ctx}}{\langle\rangle \in \Gamma \to \diamond}\ (\textsc{empty-subs}) \qquad \frac{\Gamma \in \mathsf{Ctx}}{\mathsf{id}_\Gamma \in \Gamma \to \Gamma}\ (\textsc{id-subs}) \qquad \frac{A \in \mathsf{Type}(\Gamma)}{\mathsf{p} \in \Gamma.A \to \Gamma}\ (\textsc{fst-subs})$$

$$\frac{\sigma \in \Gamma \to \Delta \qquad t \in \mathsf{Term}(\Gamma, A\,\sigma)}{(\sigma, t) \in \Gamma \to \Delta.A}\ (\textsc{ext-subs}) \qquad \frac{\delta \in \Gamma \to \Theta \qquad \sigma \in \Theta \to \Delta}{\sigma\,\delta \in \Gamma \to \Delta}\ (\textsc{comp-subs})$$

Types The only new type constructor is $\{t\}_A$, for singletons: a subtype of A containing t as single inhabitant.

$$\frac{\Gamma \in \mathsf{Ctx}}{\mathsf{U} \in \mathsf{Type}(\Gamma)}\ (\textsc{u-f}) \qquad \frac{A \in \mathsf{Term}(\Gamma, \mathsf{U})}{A \in \mathsf{Type}(\Gamma)}\ (\textsc{u-el}) \qquad \frac{A \in \mathsf{Type}(\Gamma) \qquad B \in \mathsf{Type}(\Gamma.A)}{\mathsf{Fun}\,A\,B \in \mathsf{Type}(\Gamma)}\ (\textsc{fun-f})$$

$$\frac{A \in \mathsf{Type}(\Gamma) \qquad t \in \mathsf{Term}(\Gamma, A)}{\{t\}_A \in \mathsf{Type}(\Gamma)}\ (\textsc{sing-f}) \qquad \frac{A \in \mathsf{Type}(\Delta) \qquad \sigma \in \Gamma \to \Delta}{A\,\sigma \in \mathsf{Type}(\Gamma)}\ (\textsc{subs-type})$$

Terms Singleton types do not have a special constructor for their inhabitants; on the other hand, we close the universe under singleton types.

$$\frac{A \in \mathsf{Term}(\Gamma, \mathsf{U}) \qquad B \in \mathsf{Term}(\Gamma.A, \mathsf{U})}{\mathsf{Fun}\,A\,B \in \mathsf{Term}(\Gamma, \mathsf{U})}\ (\textsc{fun-u-i}) \qquad \frac{t \in \mathsf{Term}(\Gamma.A, B)}{\lambda t \in \mathsf{Term}(\Gamma, \mathsf{Fun}\,A\,B)}\ (\textsc{fun-i})$$

$$\frac{B \in \mathsf{Type}(\Gamma.A) \qquad t \in \mathsf{Term}(\Gamma, \mathsf{Fun}\,A\,B) \qquad u \in \mathsf{Term}(\Gamma, A)}{\mathsf{App}\,t\,u \in \mathsf{Term}(\Gamma, B\,(\mathsf{id}_\Gamma, u))}\ (\textsc{fun-el})$$

$$\frac{A \in \mathsf{Type}(\Gamma)}{\mathsf{q} \in \mathsf{Term}(\Gamma.A, A\,\mathsf{p})}\ (\textsc{hyp}) \qquad \frac{\sigma \in \Gamma \to \Delta \qquad t \in \mathsf{Term}(\Delta, A)}{t\,\sigma \in \mathsf{Term}(\Gamma, A\,\sigma)}\ (\textsc{subs-term})$$

$$\frac{A \in \mathsf{Term}(\Gamma, \mathsf{U}) \qquad t \in \mathsf{Term}(\Gamma, A)}{\{t\}_A \in \mathsf{Term}(\Gamma, \mathsf{U})}\ (\textsc{sing-u-i}) \qquad \frac{t \in \mathsf{Term}(\Gamma, A)}{t \in \mathsf{Term}(\Gamma, \{t\}_A)}\ (\textsc{sing-i})$$

$$\frac{a \in \mathsf{Term}(\Gamma, A) \qquad t \in \mathsf{Term}(\Gamma, \{a\}_A)}{t \in \mathsf{Term}(\Gamma, A)}\ (\textsc{sing-el})$$

Axioms for the Equational Theory In the following, we present the axioms of the equational theory of λ^{Sing}. Equality is considered as the congruence closure of these axioms. Congruence rules, also called derived rules, are generated mechanically for each symbol from its typing. For instance, rule (SUBS-TYPE) induces the derived rule

$$\frac{A = B \in \mathsf{Type}(\Gamma) \qquad \gamma = \delta \in \Delta \to \Gamma}{A\,\gamma = B\,\delta \in \mathsf{Type}(\Delta)}\ (\textsc{subs-cong-ty}) \quad .$$

Another instance of a derived rule is conversion, it holds because of equality between sorts, such as $\mathsf{Term}(\Gamma, A) = \mathsf{Term}(\Gamma, A')$:

$$\frac{t \in \mathsf{Term}(\Gamma, A) \qquad A = A' \in \mathsf{Type}(\Gamma)}{t \in \mathsf{Term}(\Gamma, A')}\ (\textsc{conv})$$

In the following, we present equality axioms without the premises concerning typing, except in the cases where they cannot be inferred.

Substitutions The first two equations witness extensionality for the identity substitution, the next three the composition laws for the category of substitutions. Then there is a law for the first projection p, and the last two laws show how to propagate a substitution δ into a tuple.

$$\begin{aligned} \mathsf{id}_\diamond &= \langle\rangle & \mathsf{id}_{\Gamma.A} &= (\mathsf{p}, \mathsf{q}) \\ \mathsf{id}\,\sigma &= \sigma & \sigma\,\mathsf{id} &= \sigma \\ (\sigma\,\delta)\,\gamma &= \sigma\,(\delta\,\gamma) & \mathsf{p}\,(\sigma, t) &= \sigma \\ \langle\rangle\,\delta &= \langle\rangle & (\sigma, t)\,\delta &= (\sigma\,\delta, t\,\delta) \end{aligned}$$

Axioms for β and η, propagation and resolution of substitutions An explicit substitution (id_Γ, r) is created by contracting a β-redex (first law). It is then propagated into the various term constructions until it can be resolved (last two laws).

$$\begin{aligned} \mathsf{App}\,(\lambda t)\,r &= t\,(\mathsf{id}_\Gamma, r) & \lambda(\mathsf{App}\,(t\,\mathsf{p})\,\mathsf{q}) &= t \\ \mathsf{U}\,\sigma &= \mathsf{U} & (\{t\}_A)\,\sigma &= \{t\,\sigma\}_{A\,\sigma} \\ (\mathsf{Fun}\,A\,B)\,\sigma &= \mathsf{Fun}\,(A\,\sigma)\,(B\,(\sigma\,\mathsf{p}, \mathsf{q})) & (\lambda t)\,\sigma &= \lambda(t\,(\sigma\,\mathsf{p}, \mathsf{q})) \\ (\mathsf{App}\,r\,s)\,\sigma &= \mathsf{App}\,(r\,\sigma)\,(s\,\sigma) & (t\,\delta)\,\sigma &= t\,(\delta\,\sigma) \\ \mathsf{q}\,(\sigma, t) &= t & t\,\mathsf{id} &= t \end{aligned}$$

Singleton types All inhabitants of a singleton type are equal (sing-eq-i). We mention the important derived rule (sing-eq-el) here explicitly.

$$\frac{t, t' \in \mathsf{Term}(\Gamma, \{a\}_A)}{t = t' \in \mathsf{Term}(\Gamma, \{a\}_A)}\ (\text{sing-eq-i}) \qquad \frac{t = t' \in \mathsf{Term}(\Gamma, \{a\}_A)}{t = t' \in \mathsf{Term}(\Gamma, A)}\ (\text{sing-eq-el})$$

There is a choice how to express the last two rules; they could be replaced with

$$\frac{t \in \mathsf{Term}(\Gamma, \{a\}_A)}{t = a \in \mathsf{Term}(\Gamma, \{a\}_A)}\ (\text{sing-eq-i}') \qquad \frac{t \in \mathsf{Term}(\Gamma, \{a\}_A)}{t = a \in \mathsf{Term}(\Gamma, A)}\ (\text{sing-eq-el}')$$

The rule (sing-eq-el) is essential; in fact, since we have eta-expansion for singletons, we would like to derive

$$\Gamma.\{\lambda t\}_{\mathsf{Fun}\ A\ B} \vdash \mathsf{App}\ q\ a = t\ (\mathsf{id}, a) : B\ (\mathsf{id}, a)$$

from $\Gamma.\{\lambda t\}_{\mathsf{Fun}\ A\ B} \vdash q = \lambda t : \{\lambda t\}_{\mathsf{Fun}\ A\ B}$, and $\Gamma \vdash a : A$. Which would be impossible if (sing-eq-el) were not a rule.

Properties of λ^{Sing}

An advantage of introducing the calculus as a GAT is that we can derive several syntactical results from the meta-theory of GATs; for instance, some of the following inversion results, which are needed in the proof of completeness of the type-checking algorithm.

Remark 11 (Weakening of judgements). Let $\Delta \leqslant^i \Gamma$, $\Gamma \vdash A = A'$, and $\Gamma \vdash t = t' : A$; then $\Delta \vdash A\ p^i = A'\ p^i$, and $\Delta \vdash t\ p^i = t'\ p^i : A\ p^i$.

Remark 12 (Syntactic validity).

1. If $\Gamma \vdash t : A$, then $\Gamma \vdash A$.
2. If $\Gamma \vdash t = t' : A$, then both $\Gamma \vdash t : A$, and $\Gamma \vdash t' : A$.
3. If $\Gamma \vdash A = A'$, then both $\Gamma \vdash A$, and $\Gamma \vdash A'$.

Lemma 28 (Inversion of types).

1. If $\Gamma \vdash \mathsf{Fun}\ A\ B$, then $\Gamma \vdash A$, and $\Gamma.A \vdash B$.
2. If $\Gamma \vdash \{a\}_A$, then $\Gamma \vdash A$, and $\Gamma \vdash a : A$.
3. If $\Gamma \vdash k$, then $\Gamma \vdash k : U$.

The following lemma can be proved directly by induction on derivations by checking the possibles rules used in the last step.

Lemma 29 (Inversion of typing).

1. If $\Gamma \vdash \mathsf{Fun}\ A'\ B' : A$, then $\Gamma \vdash A = U$, $\Gamma \vdash A' : U$, and also $\Gamma.A' \vdash B' : U$;
2. if $\Gamma \vdash \{b\}_B : A$, then $\Gamma \vdash A = U$, $\Gamma \vdash B : U$, and also $\Gamma \vdash b : B$;
3. if $\Gamma \vdash \lambda t : A$, then either

a) $\Gamma \vdash A = \mathsf{Fun}\, A'\, B$ with $\Gamma.A' \vdash t : B$; or

b) $\Gamma \vdash A = \{a\}_{A'}$ with $\Gamma \vdash \lambda t = a : A'$.

4. if $\Gamma \vdash \mathsf{App}\, t\, r : A$, then $\Gamma \vdash t : \mathsf{Fun}\, A'\, B'$, $\Gamma \vdash r : A'$, and $\Gamma \vdash A = B'\,(\mathsf{id}, r')$.

Proof. (1) The last rule used is one of (FUN-U-I), (CONV), (SING-I), or (SING-E). In the first case the premises of the rule are what is to be proved; in all other cases we have a premise with the form $\Gamma \vdash \mathsf{Fun}\, A'\, B' : B$, hence we can apply the i.h. (2-4) Analogously. □

Remark 13 (Inversion of substitution). Any substitution $\Delta \vdash \sigma : \Gamma.A$ is equal to some substitution $\Delta \vdash (\sigma', t) : \Gamma.A$. It is enough to note $\mathsf{id}_{\Gamma.A} = (p, q)$, hence we have the equalities $\sigma = \mathsf{id}\,\sigma = (p, q)\,\sigma = (p\,\sigma, q\,\sigma)$.

The extension of the calculus with singleton types introduces a new shape for normal forms to those of Def. 12.

Definition 21 (Neutral terms, and normal forms).

$$\begin{aligned} \mathrm{Ne} \ni k &::= q \mid q p^{i+1} \mid \mathsf{App}\, k\, v \\ \mathrm{Nf} \ni v, V, W &::= U \mid \mathsf{Fun}\, V\, W \mid \{v\}_V \mid \lambda v \mid k \end{aligned}$$

Calculus λ^{Irr}*, a type theory with proof-irrelevance*

In this section we keep the basic rules of the previous calculus (those that do not refer to singleton types), and introduce types for natural numbers, enumeration sets, sigma types, and proof-irrelevant types. The main difference with other presentations [81, 94], on the syntactic level, is that the eliminator operator (for each type) has as an argument the type of the result. The presence of the resulting type in the eliminator is needed in order to define the normalisation function; it is also necessary for the type-inference algorithm.

Some of the introductory rules of this section do not meet the requirements of GATs rules; nevertheless, in order to keep the presentation uniform, we use the same format of the previous section.

Sigma types Both U and Type are closed under (strong) sigma-type formation; (a, b) introduces a dependent pair and fst t and snd t eliminate it.

$$\frac{A \in \mathsf{Term}(\Gamma, U) \qquad B \in \mathsf{Term}(\Gamma.A, U)}{\Sigma\, A\, B \in \mathsf{Term}(\Gamma, U)}\ (\text{SUM-U-I}) \qquad \frac{A \in \mathsf{Type}(\Gamma) \qquad B \in \mathsf{Type}(\Gamma.A)}{\Sigma\, A\, B \in \mathsf{Type}(\Gamma)}\ (\text{SUM-F})$$

$$\frac{B \in \mathsf{Type}(\Gamma.A) \qquad a \in \mathsf{Term}(\Gamma, A) \qquad b \in \mathsf{Term}(\Gamma, B\,(\mathsf{id}, a))}{(a, b) \in \mathsf{Term}(\Gamma, \Sigma\, A\, B)}\ (\text{SUM-IN})$$

$$\frac{t \in \mathsf{Term}(\Gamma, \Sigma\, A\, B)}{\mathsf{fst}\ t \in \mathsf{Term}(\Gamma, A)}\ (\text{SUM-EL1}) \qquad \frac{t \in \mathsf{Term}(\Gamma, \Sigma\, A\, B)}{\mathsf{snd}\ t \in \mathsf{Term}(\Gamma, B\,(\mathsf{id}, \mathsf{fst}\ t))}\ (\text{SUM-EL2})$$

The β- and η-laws for pairs are given by the first three equations to follow. The remaining equations propagate substitutions into the new term constructors.

$$\begin{array}{lll} \mathsf{fst}\ (a, b) = a & \mathsf{snd}\ (a, b) = b & (\mathsf{fst}\ t, \mathsf{snd}\ t) = t \\ (\mathsf{fst}\ t)\ \sigma = \mathsf{fst}\ (t\ \sigma) & (\mathsf{snd}\ t)\ \sigma = \mathsf{snd}\ (t\ \sigma) & (a, b)\ \sigma = (a\ \sigma, b\ \sigma) \\ & (\Sigma\ A\ B)\ \sigma = \Sigma\ (A\ \sigma)\ (B\ (\sigma\ p, q)) & \end{array}$$

Propagation laws can be obtained mechanically: to propagate σ into $c\ \vec{t}$, just compose it with each t_i that is not a binder (e.g., A in $\Sigma\ A\ B$), and compose its lifted version $(\sigma\ p, q)$ with each t_j that is a binder (e.g., B in $\Sigma\ A\ B$). Binders are those formed in an extended context (here, $B \in \mathsf{Type}(\Gamma.A)$). In the following, we will skip the propagation laws.

Natural numbers We add an inductive type Nat with constructors zero and $\mathsf{suc}()$ and primitive recursion natrec.

$$\frac{\Gamma \in \mathsf{Ctx}}{\mathsf{Nat} \in \mathsf{Term}(\Gamma, \mathsf{U})}\ (\textsc{nat-u-i}) \qquad \frac{\Gamma \in \mathsf{Ctx}}{\mathsf{zero} \in \mathsf{Term}(\Gamma, \mathsf{Nat})}\ (\textsc{nat-z-i}) \qquad \frac{t \in \mathsf{Term}(\Gamma, \mathsf{Nat})}{\mathsf{suc}(t) \in \mathsf{Term}(\Gamma, \mathsf{Nat})}\ (\textsc{nat-s-i})$$

$$\frac{\begin{array}{c} B \in \mathsf{Type}(\Gamma.\mathsf{Nat}) \\ t \in \mathsf{Term}(\Gamma, \mathsf{Nat}) \qquad z \in \mathsf{Term}(\Gamma, B\ (\mathsf{id}, \mathsf{zero})) \qquad s \in \mathsf{Term}(\Gamma, \mathsf{Rec}(B)) \end{array}}{\mathsf{natrec}(B, z, s, t) \in \mathsf{Term}(\Gamma, B\ (\mathsf{id}, t))}\ (\textsc{nat-el})$$

Here, we used $\mathsf{Rec}(B)$ as an abbreviation for $\mathsf{Fun}\ \mathsf{Nat}\ (\mathsf{Fun}\ B\ (B\ (p, \mathsf{suc}(q))\ p))$ which in conventional notation reads $\Pi x : \mathsf{Nat}.\ B \to B[\mathsf{suc}(x)/x]$. Since B is a big type, it can mention the universe U, thus, we can define small types by recursion via natrec. This so called *large elimination* excludes normalization proofs which use induction on type expressions [43, 46]. We add the usual computation laws for primitive recursion.

$$\begin{aligned} \mathsf{natrec}(B, z, s, \mathsf{zero}) &= z \\ \mathsf{natrec}(B, z, s, (\mathsf{suc}(t))) &= \mathsf{App}\ (\mathsf{App}\ s\ t)\ (\mathsf{natrec}(B, z, s, t)) \end{aligned}$$

Enumeration sets The type N_n has the n canonical inhabitants $c_0^n, \ldots, c_{n-1}^n$, which can be eliminated by the dependent case distinction $\mathsf{case}^n\ B\ t_0 \cdots t_{n-1}\ t$ with n branches.

$$\frac{\Gamma \in \mathsf{Ctx}}{\mathsf{N}_n \in \mathsf{Term}(\Gamma, \mathsf{U})}\ (\textsc{N}_n\textsc{-u-i}) \qquad \frac{\Gamma \in \mathsf{Ctx} \qquad i < n}{c_i^n \in \mathsf{Term}(\Gamma, \mathsf{N}_n)}\ (\textsc{N}_n\textsc{-i})$$

$$\frac{\begin{array}{c} B \in \mathsf{Type}(\Gamma.\mathsf{N}_n) \qquad t \in \mathsf{Term}(\Gamma, \mathsf{N}_n) \\ t_0 \in \mathsf{Term}(\Gamma, B\ (\mathsf{id}, c_0^n)) \ \cdots\ t_{n-1} \in \mathsf{Term}(\Gamma, B\ (\mathsf{id}, c_{n-1}^n)) \end{array}}{\mathsf{case}^n\ B\ t_0 \cdots t_{n-1}\ t \in \mathsf{Term}(\Gamma, B\ (\mathsf{id}, t))}\ (\textsc{N}_n\textsc{-e})$$

We add the usual computational law for case distinction, and weak extensionality, which for booleans (N_2) reads "$\mathsf{if}\ t\ \mathsf{then\ true\ else\ false} = t$" in sugared

syntax.

$$\mathsf{case}^n\ B\ t_0 \cdots t_{n-1}\ c_i^n = t_i$$
$$\mathsf{case}^n\ \mathsf{N}_n\ c_0^n \cdots c_{n-1}^n\ t = t$$

In the sequel we use $\vec{t}$ for denoting, in $\mathsf{case}^n\ B\ t_0 \cdots t_{n-1}\ r$, the n terms $t_0 \cdots t_{n-1}$. We will omit superscripts n in c_i, and in case $B\ \vec{t}\ r$.

For N_0 and N_1 we can formulate strong η-laws: all their inhabitants are considered equal, since there is at most one. To realize this, we introduce a new term $\star$ in N_0 *if it already has an inhabitant* t; we consider $\star$ as normal form of t. Note that this seemingly paradoxical canonical form $\star \in \mathsf{N}_0$ does not threaten consistency, since it cannot exist in the empty context $\Gamma = \diamond$; otherwise there would have already been a term $t \in \mathsf{Term}(\diamond, \mathsf{N}_0)$.

$$\frac{t \in \mathsf{Term}(\Gamma, \mathsf{N}_0)}{\star \in \mathsf{Term}(\Gamma, \mathsf{N}_0)}\ (\text{N}_0\text{-TM}) \qquad \frac{t, t' \in \mathsf{Term}(\Gamma, \mathsf{N}_0)}{t = t' \in \mathsf{Term}(\Gamma, \mathsf{N}_0)}\ (\text{N}_0\text{-EQ}) \qquad \frac{t, t' \in \mathsf{Term}(\Gamma, \mathsf{N}_1)}{t = t' \in \mathsf{Term}(\Gamma, \mathsf{N}_1)}\ (\text{N}_1\text{-EQ})$$

On the one hand, rule (N_0-TM) destroys decidability of type checking: to check whether $\star \in \mathsf{Term}(\Gamma, \mathsf{N}_0)$ we would have to decide the consistency of Γ which is certainly impossible in a theory with natural numbers. On the other hand, it allows us to decide equality by computing canonical forms. We solve this dilemma by forbidding $\star$ in the user syntax which is input for the type-checker; $\star$ is only used internally in the NbE algorithm and in the canonical forms it produces. Formally this is reflected by having two calculi: one with the rule (N_0-TM) and one without it. To distinguish the calculi, we decorate the turnstile ($\vdash^\star$) in judgements of the former and leave ($\vdash$) for the calculus without (N_0-TM). We also use the different turnstiles for referring to each calculus. In Sec. 4.1 we prove that ($\vdash^\star$) is a conservative extension of ($\vdash$).

Strong extensionality for booleans and larger enumeration sets is hard to implement [12, 16] and beyond the scope of this work.

Proof irrelevance Our treatment of proof-irrelevance is based on Awodey and Bauer [15] and Maillard [79]. The constructor $[\_]$ turns a type A into the *proposition* $[A]$ in the sense that only the fact matters *whether* A is inhabited, not by *what*. An inhabited proposition is regarded as *true*, an uninhabited as *false*. The proposition $[A]$ still has all inhabitants of A, but now they are considered equal. If A is not empty, we introduce a trivial proof $\star$ in $[A]$ which we regard as the normal form of any $t \in \mathsf{Term}(\Gamma, [A])$.

$$\frac{A \in \mathsf{Term}(\Gamma, \mathsf{U})}{[A] \in \mathsf{Term}(\Gamma, \mathsf{U})}\ (\text{PRF-F}) \qquad \frac{A \in \mathsf{Type}(\Gamma)}{[A] \in \mathsf{Type}(\Gamma)}\ (\text{PRF-F})$$

$$\frac{a \in \mathsf{Term}(\Gamma, A)}{[a] \in \mathsf{Term}(\Gamma, [A])}\ (\text{PRF-I}) \qquad \frac{a \in \mathsf{Term}(\Gamma, A)}{\star \in \mathsf{Term}(\Gamma, [A])}\ (\text{PRF-TM})$$

$$\frac{A \in \mathsf{Type}(\Gamma) \qquad t, t' \in \mathsf{Term}(\Gamma, [A])}{t = t' \in \mathsf{Term}(\Gamma, [A])}\ (\text{PRF-EQ})$$

Note that (prf-tm) is analogous to (N_0-tm) and the same remarks apply; in particular, (prf-tm) is also a rule in ($\vdash^\star$) but not in ($\vdash$).

We use Awodey and Bauer's [15] elimination rule for proofs.

$$\frac{\Gamma \vdash t : [A] \qquad \Gamma \vdash B \qquad \Gamma, x{:}A \vdash b : B \qquad \Gamma, x{:}A, y{:}A \vdash b = b[y/x] : B}{\Gamma \vdash b \text{ where } [x] \leftarrow t : B}$$

The content $x : A$ of a proof $t : [A]$ can be used in b via the elimination b where $[x] = t$ if b does not actually depend on it, which is expressed via the hypothesis that b should be equal to $b[y/x]$ for an arbitrary y. This elimination principle is stronger than "proofs can only be used inside of proofs" which is witnessed by the rule:

$$\frac{\Gamma \vdash t : [A] \qquad \Gamma \vdash B \qquad \Gamma, x{:}A \vdash b : [B]}{\Gamma \vdash b \text{ where } [x] \leftarrow t : [B]}$$

Note that this weaker elimination rule in the style of a *bind* operation for monads is an instance of the Awodey-Bauer rule, since the equation $\Gamma, x{:}A, y{:}A \vdash b = b[y/x] : [B]$ holds trivially due to proof irrelevance.

The Awodey-Bauer where fulfills β-, η-, and associativity laws analogous to the ones of a monad.

$$\begin{aligned} b \text{ where } [x] \leftarrow [a] &= b[a/x] \\ b[[x]/y] \text{ where } [x] \leftarrow t &= b[t/y] \\ a \text{ where } [x] \leftarrow (b \text{ where } [y] \leftarrow c) &= \\ (a \text{ where } [x] \leftarrow b) \text{ where } [y] \leftarrow c &\quad \text{if } y \notin \mathsf{FV}(a) \end{aligned}$$

After this more readable presentation in named syntax, we add the eliminator and its equations to our framework in de Bruijn style:

(prf-el)

$$\frac{\begin{array}{c} t \in \mathsf{Term}(\Gamma, [A]) \\ B \in \mathsf{Type}(\Gamma) \qquad b \in \mathsf{Term}(\Gamma.A, B\,p) \qquad b\,p = b\,(p\,p, q) \in \mathsf{Term}(\Gamma.A.A\,p, B\,p\,p) \end{array}}{b \text{ where}^B\ t \in \mathsf{Term}(\Gamma, B)}$$

$$\begin{aligned} b \text{ where}^B [a] &= b\,(\mathsf{id}, a) && \text{(prf-}\beta\text{)} \\ b\,(p, [q]) \text{ where}^B t &= b\,(\mathsf{id}, t) && \text{(prf-}\eta\text{)} \\ a \text{ where}^A (b \text{ where}^B c) &= (a\,(p\,p, q) \text{ where}^{A\,p} b) \text{ where}^B c && \text{(prf-assoc)} \end{aligned}$$

After exposition of the formation, introduction, elimination, and equality rules for the types of λ^{Irr}, we continue with basic properties of derivations.

Definition 22 (Neutral terms and normal forms).

$$\begin{aligned} \mathrm{Ne} \ni k ::= &\ldots \mid \mathsf{fst}\ k \mid \mathsf{snd}\ k \mid \mathsf{natrec}(V, v, v', k) \mid \\ &\mathsf{case}^n\ V\ v_0 \cdots v_{n-1}\ k \mid v \text{ where}^V k \mid \star \\ \mathrm{Nf} \ni v, V ::= &\ldots \mid \Sigma\, V\, W \mid \mathsf{Nat} \mid \mathsf{N}_n \mid [V] \mid (v, v') \mid \mathsf{zero} \mid \mathsf{suc}(v) \mid \mathsf{c}_i^n \mid [v] \end{aligned}$$

The next lemma can be strengthened by stating that the judgements in the conclusion of each property corresponds to a sub-derivation of the derivation in the premise. The importance of the stronger version is that, in proof by induction on derivations, one can use induction hypotheses in the inverted judgements.

Lemma 30 (Inversion of types).

1. If $\Gamma \vdash \Sigma\, A\, B$, then $\Gamma \vdash A$, and $\Gamma.A \vdash B$.
2. If $\Gamma \vdash [A]$, then $\Gamma \vdash A$.

Lemma 31 (Inversion of typing).

1. If $\Gamma \vdash \Sigma\, A'\, B : A$, then $\Gamma \vdash A = U$, and $\Gamma \vdash A' : U$, and $\Gamma.A' \vdash B : U$.
2. If $\Gamma \vdash \mathsf{Nat} : A$, then $\Gamma \vdash A = U$.
3. If $\Gamma \vdash \mathsf{N}_n : A$, then $\Gamma \vdash A = U$.
4. If $\Gamma \vdash (t, b) : A$, then $\Gamma \vdash A = \Sigma\, A'\, B$, and $\Gamma \vdash t : A'$, and $\Gamma \vdash b : B\,(\mathsf{id}, t)$.
5. If $\Gamma \vdash \mathsf{fst}\ t : A$, then $\Gamma \vdash A = A'$, and $\Gamma \vdash t : \Sigma\, A'\, B$, for some A', and B.
6. If $\Gamma \vdash \mathsf{snd}\ t : B$, then $\Gamma \vdash B = B'\,(\mathsf{id}, \mathsf{fst}\ t)$, and $\Gamma \vdash t : \Sigma\, A\, B'$, for some A, and B'.
7. If $\Gamma \vdash \mathsf{zero} : A$, then $\Gamma \vdash A = \mathsf{Nat}$.
8. If $\Gamma \vdash \mathsf{suc}(t) : A$, then $\Gamma \vdash t : \mathsf{Nat}$, and $\Gamma \vdash A = \mathsf{Nat}$.
9. If $\Gamma \vdash \mathsf{natrec}(B, z, s, t) : A$, then $\Gamma.\mathsf{Nat} \vdash B$, $\Gamma \vdash z : B\,(\mathsf{id}, \mathsf{zero})$, $\Gamma \vdash s : \mathsf{Rec}(B)$, $\Gamma \vdash t : \mathsf{Nat}$, and $\Gamma \vdash A = B\,(\mathsf{id}, t)$.
10. if $\Gamma \vdash c_i^n : A$, then $\Gamma \vdash A = \mathsf{N}_n$;
11. If $\Gamma \vdash \mathsf{case}\ B\ \vec{t}\ t' : A$, then $\Gamma.\mathsf{N}_n \vdash B$, $\Gamma \vdash t_i : B\,(\mathsf{id}, c_i)$, $\Gamma \vdash t' : \mathsf{N}_n$, and $\Gamma \vdash A = B\,(\mathsf{id}, t)$.
12. If $\Gamma \vdash [t] : A$, then $\Gamma \vdash A = [A']$ and $\Gamma \vdash t' : A'$.
13. If $\Gamma \vdash b\ \mathsf{where}^B\ t : A$, then $\Gamma \vdash A = B$, $\Gamma \vdash t : [A']$ for some A', $\Gamma.A' \vdash b : B\,p$, and $\Gamma.A'.A'\,p \vdash b\,p = b\,(pp, q) : B\,p$.

Conservativity of $\star$

In this section we prove that $(\vdash^\star)$ is a *conservative* extension of $(\vdash)$; i.e., any derivation in $(\vdash^\star)$ has a counterpart derivation in $(\vdash)$ and the components of the conclusions of those derivations are judgmentally equal in $(\vdash^\star)$.

Definition 23. A term is called *pure* if it does not contain any occurrence of $\star$. Let ν be a syntactical entity, if μ is obtained from ν by replacing all occurrences of $\star$ by pure terms, then μ is called a *lifting* of ν.

We will distinguish those liftings that are judgmentally equal to the lifted entity, these liftings are called *good liftings*.

Definition 24 (Good lifting).

1. A context $\Gamma' \vdash$ is a good lifting of $\Gamma \vdash^\star$ if Γ' is a lifting of Γ, such that $\vdash^\star \Gamma = \Gamma'$.
2. A substitution $\Gamma' \vdash \sigma' \colon \Delta'$ is a good lifting of $\Gamma \vdash^\star \sigma : \Delta$ if $\Gamma' \vdash$ and $\Delta' \vdash$ are good liftings of $\Gamma \vdash^\star$ and $\Delta \vdash^\star$, resp., and σ' is a lifting of σ, such that $\Gamma \vdash^\star \sigma = \sigma' : \Delta$.
3. A type $\Gamma \vdash A'$ is a good lifting of $\Gamma \vdash^\star A$ if $\Gamma' \vdash$ is a good lifting of $\Gamma \vdash^\star$ and A' is a lifting of A, such that $\Gamma \vdash^\star A = A'$.
4. A term $\Gamma \vdash t' \colon A'$ is a good lifting of $\Gamma \vdash^\star t : A$ if $\Gamma' \vdash A'$ is a good lifting of $\Gamma \vdash^\star A$ and t' is a lifting of t, such that $\Gamma \vdash^\star t = t' : A$.

Now we can prove that there is a good lifting for each syntactic entity; for proving this, we need the stronger condition that any pair of good liftings for some entity are judgmentally equal.

Theorem 32.

1. Let $\Gamma \vdash^\star$; then there is a good lifting $\Gamma' \vdash$ of $\Gamma \vdash^\star$; moreover if $\Gamma'' \vdash$ is also a good lifting of $\Gamma \vdash^\star$ then $\vdash \Gamma' = \Gamma''$.
2. Let $\Gamma \vdash^\star \sigma : \Delta$; then there is a good lifting $\Gamma' \vdash \sigma' \colon \Delta'$ of $\Gamma \vdash^\star \sigma : \Delta$; moreover if $\Gamma'' \vdash \sigma'' \colon \Delta''$ is also a good lifting of $\Gamma \vdash^\star \sigma : \Delta$ then $\vdash \Gamma' = \Gamma''$, $\vdash \Delta' = \Delta''$, and $\Gamma' \vdash \sigma' = \sigma'' \colon \Delta'$.
3. Let $\Gamma \vdash^\star A$; then there is a good lifting $\Gamma' \vdash A'$ of $\Gamma \vdash^\star A$; moreover if $\Gamma'' \vdash A''$ is also a good lifting of $\Gamma \vdash^\star A$ then $\vdash \Gamma' = \Gamma''$ and $\Gamma' \vdash A' = A''$.
4. Let $\Gamma \vdash^\star t : A$; then there is a good lifting $\Gamma' \vdash t' \colon A'$ of $\Gamma \vdash^\star t : A$; moreover if $\Gamma'' \vdash t'' \colon A''$ is also a good lifting of $\Gamma \vdash^\star t : A$ then $\vdash \Gamma' = \Gamma''$, $\Gamma' \vdash A' = A''$, and $\Gamma' \vdash t' = t'' \colon A'$.

Proof. By induction on derivations, in each rule we use i.h., and build up the corresponding entity to the good lifting for each part of the judgement; then, given any other good lifting of the whole judgement, we do inversion on the definition of good lifting, and get the equalities for each part; we finish using congruence for showing that both good lifting are judgmental equal.

We show the case for (prf-tm). First we prove the existence of a good lifting.

$\Gamma \vdash^\star \star : [A]$	hypothesis	(*)
$\Gamma \vdash^\star t : A$	by inversion on (*)	(†)
$\Gamma' \vdash t' \colon A'$	by ind. hyp. is a good lifting of (†)	
$\Gamma' \vdash [t'] \colon [A']$	by (prf-i), is a good lifting of(*)	

Now we prove the second half of the theorem.

$\Gamma'' \vdash s \colon B$	hypothesis, be other good lifting of (†)	(**)
$\Gamma'' \vdash B$	by inversion, good lifting of $\Gamma \vdash^\star [A]$	
$\Gamma' \vdash B'' = [A']$	by ind. hyp.	
$\Gamma' \vdash [t'] = s \colon [A']$	by (prf-eq) and (conv).	

□

Corollary 33. The calculus $(\vdash^\star)$ is a conservative extension of $(\vdash)$. □

Combining singleton types and proof-irrelevant propositions For illustrating the difficulties one can find when extending λ^{Irr} with singleton types, consider a slightly different calculus where we drop the type annotation of the eliminator for proof-irrelevance terms; i.e. we would have b where t instead of b where$^{\mathsf{B}}$ t. In the resulting system one can derive:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \mathsf{c}_0^2 : \{\mathsf{c}_0^2\}_{\mathsf{N}_2}}}{\vdash \star : [\{\mathsf{c}_0^2\}_{\mathsf{N}_2}]}}{\vdash \mathsf{x} \text{ where } [\mathsf{x}] \leftarrow \star : \{\mathsf{c}_0^2\}_{\mathsf{N}_2}}}{\vdash \mathsf{x} \text{ where } [\mathsf{x}] \leftarrow \star = \mathsf{c}_0^2 : \{\mathsf{c}_0^2\}_{\mathsf{N}_2}}}{\vdash \mathsf{x} \text{ where } [\mathsf{x}] \leftarrow \star = \mathsf{c}_0^2 : \mathsf{N}_2} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \mathsf{c}_1^2 : \{\mathsf{c}_1^2\}_{\mathsf{N}_2}}}{\vdash \star : [\{\mathsf{c}_1^2\}_{\mathsf{N}_2}]}}{\vdash \mathsf{x} \text{ where } [\mathsf{x}] \leftarrow \star : \{\mathsf{c}_1^2\}_{\mathsf{N}_2}}}{\vdash \mathsf{x} \text{ where } [\mathsf{x}] \leftarrow \star = \mathsf{c}_1^2 : \{\mathsf{c}_1^2\}_{\mathsf{N}_2}}}{\vdash \mathsf{x} \text{ where } [\mathsf{x}] \leftarrow \star = \mathsf{c}_1^2 : \mathsf{N}_2}}{\vdash \mathsf{c}_0^2 = \mathsf{c}_1^2 : \mathsf{N}_2}$$

This derivation shows that mixing the rule (SING-EQ-EL) with erasure of proof-terms leads to inconsistencies. It is yet unclear how to combine singleton types and erasure of proof-terms; we leave this topic for a future work. On the other hand, there are no problems in extending $(\vdash)$ with singletons types; in fact, we can construct (see Rem. 17) a model where $[\![\mathsf{c}_0^2]\!] \neq [\![\mathsf{c}_1^2]\!]$, which assures $\nvdash \mathsf{c}_0^2 = \mathsf{c}_1^2 : \mathsf{N}_2$.

4.2 Semantics

In this section we define a class of models for the calculus λ^{Sing}. Since it features (η) for function spaces and singletons, we need to use a PER model and to η-expand in the model as in Sec. 2.3. As is to be expected, the model also shares some structure with that of Sec. 3.2: there are PERs for large types and for the universe of small types, and a family of PERs indexed over semantical types. In the last part of this section we extend the model and the normalisation function for λ^{Irr}.

PER semantics

In this subsection we introduce the abstract notion of PER models for our theory. We have already introduced PERs in Def. 5 and also settled some naming and notation conventions. The only new concept related with PERs that we use is that of a family of PERs indexed by a PER: if $R \in \mathrm{PER}(A)$ and $F \colon \mathrm{dom}(R) \to \mathrm{PER}(A)$, we say that F is *a family of PERs indexed by* R iff for all $a = a' \in R$, $F\, a = F\, a'$. If F is a family indexed by R, we write $F \colon R \to \mathrm{PER}(A)$.

The following definitions are standard (e.g. [14, 42]) in definitions of PER models for dependent types. The first one is even standard for non-dependent types (cf. [89]) and "F-bounded polymorphism" ([29]); its definition clearly shows that equality is interpreted extensionally for dependent function spaces. The second one is the PER corresponding to the interpretation of singleton types; it has as its domain all the elements related to the distinguished element of the singleton, and it relates everything in its domain.

Definition 25. Let A be an applicative structure, $X \in \mathrm{PER}(A)$, and $F \in X \to \mathrm{PER}(A)$.

1. $\prod XF = \{(f, f') \mid f \cdot a = f' \cdot a' \in F\,a, \text{ for all } a = a' \in X\}$;
2. $\{a\}_X = \{(b, b') \mid a = b \in X \text{ and } a = b' \in X\}$.

Besides interpreting function spaces and singletons we need PERs for the denotation of the universe of small types, and for the set of large types; jointly with these PERs we need functions assigning a PER for each element in the domain of these universe PERs. Note that this forces the applicative structure to have some distinguished elements.

Definition 26 (Universe). Given an applicative structure A with distinguished elements Fun and Sing, a universe $(U, \lfloor\_\rfloor)$ is a PER U over A and a family $\lfloor\_\rfloor : U \to \mathsf{Per}(A)$ with the condition that U is closed under function and singleton types. This means:

1. Whenever $X = X' \in U$ and for all $a = a' \in \lfloor X \rfloor$, $F\,a = F'\,a' \in U$, then $\mathsf{Fun}\,X\,F = \mathsf{Fun}\,X'\,F' \in U$, with $\lfloor \mathsf{Fun}\,X\,F \rfloor = \prod \lfloor X \rfloor\,(a \mapsto \lfloor F\,a \rfloor)$.
2. Whenever $X = X' \in U$ and $a = a' \in \lfloor X \rfloor$, then $\mathsf{Sing}\,a\,X = \mathsf{Sing}\,a'\,X' \in U$ and $\lfloor \mathsf{Sing}\,a\,X \rfloor = \{a\}_{\lfloor X \rfloor}$.

An applicative structure paired with one universe for small types and one universe for large types is the minimal structure needed for having a model of our theory.

Definition 27 (PER model). Let A be an applicative structure with distinguished elements U, Fun, and Sing; a *PER model* is a tuple $(A, U, T, \lfloor\_\rfloor)$ satisfying:

1. $U \subset T \in \mathsf{PER}(A)$, such that $(T, \lfloor\_\rfloor)$ and $(U, \lfloor\_\rfloor \restriction_U)$ are both universes, and
2. $\mathsf{U} \in \mathrm{dom}(T)$, with $\lfloor \mathsf{U} \rfloor = U$.

In the following definition we introduce an abstract concept for environments: since variables are represented as projection functions from lists (think of q as taking the head of a list, and p as taking the tail), it is enough having sequences together with projections.

Definition 28 (Sequences). Given a set A, a set A^* *has sequences* over A if there are distinguished operations $\top : A^*$, $\mathsf{Pair} : A^* \times A \to A^*$, $\mathsf{fst} : A^* \to A^*$, and $\mathsf{snd} : A^* \to A$ such that

$$\mathsf{fst}\,(\mathsf{Pair}\,a\,b) = a \quad \mathsf{snd}\,(\mathsf{Pair}\,a\,b) = b \ .$$

Now we need to extend the notion of PERs over A to PERs over A^* for interpreting substitutions.

Definition 29. Let A be an applicative structure and let A^* have sequences over A; moreover let $X \in \mathsf{PER}(A^*)$ and $F \in X \to \mathsf{PER}(A)$.

1. $\mathbf{1} = \{(\top, \top)\}$;
2. $\coprod XF = \{(a, a') \mid \mathsf{fst}\,a = \mathsf{fst}\,a' \in X \text{ and } \mathsf{snd}\,a = \mathsf{snd}\,a' \in F\,(\mathsf{fst}\,a)\}$;

Until here we have introduced semantic concepts. Now we are going to axiomatise the notion of evaluation, connecting the syntactic realm with the semantic one.

Definition 30 (Environment model). Let $(A, U, T, \lfloor\_\rfloor)$ be a PER model and let A^* have sequences over A. We call $M = (A, U, T, \lfloor\_\rfloor, A^*, [\![\,]\!], [\![\,]\!]^s)$ an *environment model* if the *evaluation functions* $[\![\_]\!]\_ : \mathit{Terms} \times A^* \to A$ and $[\![\_]\!]^s\_ : \mathit{Terms} \times A^* \to A^*$ satisfy:

$$
\begin{aligned}
[\![\mathsf{U}]\!]\mathfrak{a} &= \mathsf{U} & [\![\mathsf{id}]\!]^s\mathfrak{a} &= \mathfrak{a}\\
[\![\mathsf{Fun}\ \mathsf{A}\ \mathsf{B}]\!]\mathfrak{a} &= \mathsf{Fun}\ ([\![\mathsf{A}]\!]\mathfrak{a})\ \mathsf{F} & [\![\langle\rangle]\!]^s\mathfrak{a} &= \top\\
&\quad \text{where } \mathsf{F}\ \mathfrak{b} = [\![\mathsf{B}]\!](\mathsf{Pair}\ \mathfrak{a}\ \mathfrak{b}) & [\![\sigma\ \delta]\!]^s\mathfrak{a} &= [\![\sigma]\!]^s([\![\delta]\!]^s\mathfrak{a})\\
[\![\{\mathsf{t}\}_{\mathsf{A}}]\!]\mathfrak{a} &= \mathsf{Sing}\ ([\![\mathsf{t}]\!]\mathfrak{a})\ ([\![\mathsf{A}]\!]\mathfrak{a}) & [\![(\sigma, \mathsf{t})]\!]^s\mathfrak{a} &= \mathsf{Pair}\ ([\![\sigma]\!]^s\mathfrak{a})\ ([\![\mathsf{t}]\!]\mathfrak{a})\\
[\![\mathsf{t}\ \sigma]\!]\mathfrak{a} &= [\![\mathsf{t}]\!]([\![\sigma]\!]^s\mathfrak{a}) & [\![\mathsf{p}]\!]^s\mathfrak{a} &= \mathsf{fst}\ \mathfrak{a}\\
[\![\lambda\mathsf{t}]\!]\mathfrak{a} &= \mathsf{f}\text{ , where } \mathsf{f}\cdot\mathfrak{b} = [\![\mathsf{t}]\!](\mathsf{Pair}\ \mathfrak{a}\ \mathfrak{b}) & &\\
[\![\mathsf{App}\ \mathsf{t}\ \mathsf{u}]\!]\mathfrak{a} &= ([\![\mathsf{t}]\!]\mathfrak{a})\cdot([\![\mathsf{u}]\!]\mathfrak{a}) & &\\
[\![\mathsf{q}]\!]\mathfrak{a} &= \mathsf{snd}\ \mathfrak{a} & &
\end{aligned}
$$

Since no ambiguities arise, we shall henceforth write $[\![\sigma]\!]$ instead of $[\![\sigma]\!]^s$.

Once we have an environment model M we can define the denotation for contexts. The second clause in the next definition is not well-defined a priori; its totality is a corollary of Thm. 34 — contrast this approach with the one in the previous chapter, where we defined simultaneously the notion of validity with the semantics of contexts.

Definition 31. Given an environment model M, we define recursively the semantic of contexts $[\![\_]\!] : \mathsf{Ctx} \to \mathrm{PER}(A^*)$:

1. $[\![\diamond]\!] = \mathbb{1}$,
2. $[\![\Gamma.A]\!] = \coprod [\![\Gamma]\!](\mathfrak{a} \mapsto \lfloor [\![A]\!]\mathfrak{a} \rfloor)$.

We use PERs for validating equality judgements and the domain of each PER for validating typing judgements.

Definition 32 (Validity). Let M be an environment model. We define inductively the predicate of satisfability of judgements by the model, denoted with $\Gamma \vDash^M \mathcal{J}$:

1. $\diamond \vDash$ iff true
2. $\Gamma.A \vDash$ iff $\Gamma \vDash A$
3. $\Gamma \vDash A$ iff $\Gamma \vDash A = A$
4. $\Gamma \vDash A = A'$ iff $\Gamma \vDash$ and for all $\mathfrak{d} = \mathfrak{d}' \in [\![\Gamma]\!]$, $[\![A]\!]\mathfrak{d} = [\![A']\!]\mathfrak{d}' \in T$
5. $\Gamma \vDash \mathsf{t} : A$ iff $\Gamma \vDash \mathsf{t} = \mathsf{t} : A$
6. $\Gamma \vDash \mathsf{t} = \mathsf{t}' : A$ iff $\Gamma \vDash A$ and for all $\mathfrak{d} = \mathfrak{d}' \in [\![\Gamma]\!]$, $[\![\mathsf{t}]\!]\mathfrak{d} = [\![\mathsf{t}']\!]\mathfrak{d}' \in \lfloor [\![A]\!]\mathfrak{d} \rfloor$
7. $\Gamma \vDash \sigma : \Delta$ iff $\Gamma \vDash \sigma = \sigma : \Delta$
8. $\Gamma \vDash \sigma = \sigma' : \Delta$ iff $\Gamma \vDash$, $\Delta \vDash$, and for all $\mathfrak{d} = \mathfrak{d}' \in [\![\Gamma]\!]$, $[\![\sigma]\!]\mathfrak{d} = [\![\sigma']\!]\mathfrak{d}' \in [\![\Delta]\!]$.

Theorem 34 (Soundness). Let M be a model. If $\Gamma \vdash \mathcal{J}$, then $\Gamma \vDash^M \mathcal{J}$.

Proof. By easy induction on $\Gamma \vdash \mathcal{J}$. □

A concrete PER model

In this subsection we define a concrete PER model over a Scott domain which extends that of Def. 13.

Definition 33. In this section the domain used comes from the following domain equation:

$$D \approx \mathbb{O} \oplus \mathit{Var}_\perp \oplus [D \to D] \oplus (D \times D) \oplus (D \times D) \oplus \mathbb{O} \oplus (D \times [D \to D]) \oplus (D \times D) \ .$$

When comparing with the domain of the previous chapter, the only new possible shape for an element of D which is not $\perp$ is $\mathsf{Sing}\ d\ d'$ for $d, d' \in D$; so, we have the following forms for elements that are not $\perp$:

$\top$	(d, d')	for $d, d' \in D$
$\mathsf{Var}\ i$	U	for $i \in \mathit{Var}$
$\mathsf{lam}\ f$	$\mathsf{Fun}\ d\ f$	for $d \in D$, and $f \in [D \to D]$
$\mathsf{App}\ d\ d'$	$\mathsf{Sing}\ d\ d'$	for $d, d' \in D$.

In order to define an environment model over D, we endow it with an applicative structure. Note also that D has pairing, letting us to take the set of sequences over D simply as $D^* = D$ with $\mathsf{Pair}\ a\ b = (a, b)$. We define application $\_ \cdot \_ : [D \times D \to D]$ and the projections $p, q : [D \to D]$ by

$$d \cdot e = \begin{cases} f\,e & \text{if } d = \mathsf{lam}\ f \\ \perp & \text{otherwise} \end{cases}$$

$$p\ d = \begin{cases} d_1 & \text{if } d = (d_1, d_2) \\ \perp & \text{otherwise} \end{cases} \qquad q\ d = \begin{cases} d_2 & \text{if } d = (d_1, d_2) \\ \perp & \text{otherwise} \end{cases}$$

The definition of the readback function is defined just as before; the only new case with respect to Def. 14 is $\mathsf{Sing}\ d\ d'$. Note that we identify elements in *Var* with natural numbers; this is justified because *Var* is denumerable.

Definition 34 (Readback function).

$$\begin{aligned} \mathsf{R}_j\ U &= U \\ \mathsf{R}_j\ (\mathsf{Fun}\ X\ F) &= \mathsf{Fun}\ (\mathsf{R}_j\ X)\ (\mathsf{R}_{j+1}\ (F(\mathsf{Var}\ j))) \\ \mathsf{R}_j\ (\mathsf{Sing}\ d\ X) &= \{\mathsf{R}_j\ d\}_{\mathsf{R}_j\ X} \\ \mathsf{R}_j\ (\mathsf{App}\ d\ d') &= \mathsf{App}\ (\mathsf{R}_j\ d)\ (\mathsf{R}_j\ d') \\ \mathsf{R}_j\ (\mathsf{lam}\ f) &= \lambda(\mathsf{R}_{j+1}\ (f(\mathsf{Var}\ j))) \\ \mathsf{R}_j\ (\mathsf{Var}\ i) &= \begin{cases} q & \text{if } j \leqslant i+1 \\ q\,p^{j-(i+1)} & \text{if } j > i+1 \end{cases} \end{aligned}$$

As in Sec. 2.3 our PER model is based on the notions of semantical normal forms and neutral values.

Definition 35 (Semantical neutrals and normal forms).

- $d = e \in \mathsf{Ne}$ if, for all $i \in \mathbb{N}$, $\mathsf{R}_i\ d$ and $\mathsf{R}_i\ e$ are both defined, $\mathsf{R}_i\ d \equiv \mathsf{R}_i\ e$, and $\mathsf{R}_i\ d \in \mathsf{Ne}$.

- $\mathsf{d} = \mathsf{e} \in \mathsf{Nf}$ if, for all $\mathsf{i} \in \mathbb{N}$, $\mathsf{R_i\ d}$ and $\mathsf{R_i\ e}$ are both defined, $\mathsf{R_i\ d} \equiv \mathsf{R_i\ e}$, and $\mathsf{R_i\ d} \in \mathsf{Nf}$.

Lemma 35 (Closure properties of Ne and Nf).

1. $\mathsf{U} = \mathsf{U} \in \mathsf{Nf}$.
2. Let $\mathsf{X} = \mathsf{X}' \in \mathsf{Ne}$. If $\mathsf{F \cdot k} = \mathsf{F}' \cdot \mathsf{k}' \in \mathsf{Nf}$ for all $\mathsf{k} = \mathsf{k}' \in \mathsf{Ne}$, then $\mathsf{Fun\ X\ F} = \mathsf{Fun\ X'\ F}' \in \mathsf{Nf}$.
3. If $\mathsf{d} = \mathsf{d}' \in \mathsf{Nf}$ and $\mathsf{X} = \mathsf{X}' \in \mathsf{Nf}$, then $\mathsf{Sing\ d\ X} = \mathsf{Sing\ d'\ X}' \in \mathsf{Nf}$.
4. If $\mathsf{f \cdot k} = \mathsf{f}' \cdot \mathsf{k}' \in \mathsf{Nf}$ for all $\mathsf{k} = \mathsf{k}' \in \mathsf{Ne}$, then $\mathsf{f} = \mathsf{f}' \in \mathsf{Nf}$.
5. $\mathsf{Var\ i} = \mathsf{Var\ i} \in \mathsf{Ne}$ for all $\mathsf{i} \in \mathbb{N}$.
6. If $\mathsf{k} = \mathsf{k}' \in \mathsf{Ne}$ and $\mathsf{d} = \mathsf{d}' \in \mathsf{Nf}$, then $\mathsf{App\,k\,d} = \mathsf{App\,k'\,d}' \in \mathsf{Ne}$.

We define $U, T \in \mathrm{PER}(\mathsf{D})$ and $[\_] : \mathit{dom}T \to \mathrm{PER}(\mathsf{D})$ using Dybjer's schema of inductive-recursive definition [52]. We show then that $[\_]$ is a family of PERs over D.

Definition 36 (PER model).

1. Inductive definition of $U \in \mathrm{PER}(\mathsf{D})$.
   a) $\mathsf{Ne} \subseteq U$,
   b) if $\mathsf{X} = \mathsf{X}' \in U$ and $\mathsf{d} = \mathsf{d}' \in [\mathsf{X}]$, then $\mathsf{Sing\ d\ X} = \mathsf{Sing\ d'\ X}' \in U$,
   c) if $\mathsf{X} = \mathsf{X}' \in U$ and for all $\mathsf{d} = \mathsf{d}' \in [\mathsf{X}]$, $\mathsf{F\ d} = \mathsf{F}'\ \mathsf{d}' \in U$ then $\mathsf{Fun\ X\ F} = \mathsf{Fun\ X'\ F}' \in U$.
2. Inductive definition of $T \in \mathrm{PER}(\mathsf{D})$.
   a) $U \subseteq T$,
   b) $\mathsf{U} = \mathsf{U} \in T$,
   c) if $\mathsf{X} = \mathsf{X}' \in T$, and $\mathsf{d} = \mathsf{d}' \in [\mathsf{X}]$ then $\mathsf{Sing\ d\ X} = \mathsf{Sing\ d'\ X}' \in T$,
   d) if $\mathsf{X} = \mathsf{X}' \in T$, and for all $\mathsf{d} = \mathsf{d}' \in [\mathsf{X}]$, $\mathsf{F\ d} = \mathsf{F}'\ \mathsf{d}' \in T$, then $\mathsf{Fun\ X\ F} = \mathsf{Fun\ X'\ F}' \in T$.
3. Recursive definition of $[\_] \in \mathit{dom}(T) \to \mathrm{PER}(\mathsf{D})$.
   a) $[\mathsf{U}] = U$,
   b) $[\mathsf{Sing\ d\ X}] = \{\mathsf{d}\}_{[\mathsf{X}]}$,
   c) $[\mathsf{Fun\ X\ F}] = \prod [\mathsf{X}]\,(\mathsf{d} \mapsto [\mathsf{F\ d}])$,
   d) $[\mathsf{d}] = \mathsf{Ne}$, in all other cases.

Remark 14. The generation order $\sqsubset$ on T is well-founded. The minimal elements are U, and elements in Ne; $\mathsf{X} \sqsubset \mathsf{Fun\ X\ F}$, and for all $\mathsf{d} \in [\mathsf{X}]$, $\mathsf{F\ d} \sqsubset \mathsf{Fun\ X\ F}$; and, finally, $\mathsf{X} \sqsubset \mathsf{Sing\ d\ X}$.

Lemma 36. The function $[\_] : \mathrm{dom}(T) \to \mathrm{PER}(\mathsf{D})$ is a family of $\mathrm{PER}(\mathsf{D})$ over T, i.e., $[\_] : T \to \mathrm{PER}(\mathsf{D})$.

Proof. By induction on $X = X' \in T$. We do not show the base cases, for they are trivial.

1. Let $\mathsf{Sing}\ d\ X = \mathsf{Sing}\ d'\ X' \in T$.

$[X] = [X']$	by ind. hyp.
$d = d' \in [X]$	by ind. hyp.
$e = d \in [X]$ and $e' = d \in [X]$	hypothesis
$e = d' \in [X]$ and $e' = d' \in [X]$	by transitivity
$\{d\}_X = \{d'\}_{X'}$	by definition.

2. Let $\mathsf{Fun}\ X\ F = \mathsf{Fun}\ X'\ F' \in T$.

$[X] = [X']$	by ind. hyp.	
for all $d \in \mathrm{dom}([X]), F\ d = F'\ d \in T$	by definition	(*)
for all $d = d' \in [X], f \cdot d = f' \cdot d' \in [F\ d]$	hypothesis	
$f \cdot d = f' \cdot d' \in [F'\ d]$	by ind. hypothesis in (*).	

□

The previous lemma leads us to the definition of a PER model over D. Note also that D has all the distinguished elements needed to call it a syntactical applicative structure.

Corollary 37. The tuple $(D, U, T, [\_])$ is a PER model. □

Normalisation and η*-Expansion in the Model*

In the following, we adapt the NbE algorithm outlined in Chap. 2 for STT to the dependent type theory λ^{Sing}. Since readback has already be defined, we only require reflection, reification and evaluation functions.

Definition 37 (Reflection and reification). The partial functions $\uparrow\_\ \_, \downarrow\_\ \_ : [D \to [D \to D]]$ and $\Downarrow : [D \to D]$ are given as follows:

$$\uparrow_{\mathsf{Fun}\ X\ F} k = \mathsf{lam}\ (d \mapsto \uparrow_{F\ d} (\mathsf{App}\ k \downarrow_X d))$$
$$\downarrow_{\mathsf{Fun}\ X\ F} d = \mathsf{lam}\ (e \mapsto \downarrow_{F\ \uparrow_X e} (d \cdot \uparrow_X e))$$

$$\uparrow_{\mathsf{Sing}\ d\ X} k = d \qquad \downarrow_{\mathsf{Sing}\ d\ X} e = \downarrow_X d$$
$$\uparrow_U k = k \qquad \downarrow_U d = \Downarrow d$$
$$\uparrow_X k = k, \text{ in all other cases.} \qquad \downarrow_X e = e, \text{ in all other cases.}$$

$$\Downarrow(\mathsf{Fun}\ X\ F) = \mathsf{Fun}\ (\Downarrow X)\ (d \mapsto \Downarrow(F \uparrow_X d))$$
$$\Downarrow(\mathsf{Sing}\ d\ X) = \mathsf{Sing}\ (\downarrow_X d)\ (\Downarrow X)$$
$$\Downarrow U = U$$
$$\Downarrow X = X, \text{ in all other cases.}$$

In the following lemma we show that reflection $\uparrow$ corresponds to Berger and Schwichtenberg's "make self evaluating" and both reification functions $\downarrow$ and $\Downarrow$ correspond to"inverse of the evaluation function" [26]. Note that they are indexed by types values instead of syntactic types, since we are dealing with dependent instead of simple types.

Lemma 38 (Characterisation of $\uparrow$, $\downarrow$, and $\Downarrow$). Let $X = X' \in T$, then

1. if $k = k' \in \mathsf{Ne}$ then $\uparrow_X k = \uparrow_{X'} k' \in [X]$;
2. if $d = d' \in [X]$, then $\downarrow_X d = \downarrow_{X'} d' \in \mathsf{Nf}$;
3. and also $\Downarrow X = \Downarrow X' \in \mathsf{Nf}$.

Proof. By induction on $X = X' \in T$.

1. Case $\mathsf{Sing}\ d\ X = \mathsf{Sing}\ d'\ X' \in T$.

   a) The partial function $\uparrow\_\_$ maps neutrals to related elements in the corresponding PER.

$k = k \in \mathsf{Ne}$	hypothesis
$d = d' \in [X]$ and $X = X' \in T$	by inversion
$\uparrow_{\mathsf{Sing}\ d\ X} k = d$ and $\uparrow_{\mathsf{Sing}\ d'\ X'} k' = d'$	by def.
$d = d' \in \{d\}_X$	by def. of this PER.

   b) The partial function $\downarrow\_\_$ maps related elements to related normal forms.

$d_1 = d_2 \in \{d\}_X$	hypothesis
$d_1 = d_2 = d = d' \in [X]$ and $X = X' \in T$	by inversion
$\downarrow_X d = \downarrow_{X'} d' \in \mathsf{Nf}$	by ind. hyp.
$\downarrow_{\mathsf{Sing}\ d\ X} d_1 = \downarrow_{\mathsf{Sing}\ d'\ X'} d_2 \in \mathsf{Nf}$	by def.

   c) The function $\Downarrow\_$ maps related elements in T to normal forms.

$\Downarrow \mathsf{Sing}\ d\ X = \mathsf{Sing}\ (\downarrow_X d)\ (\Downarrow X)$	by def.
$\Downarrow \mathsf{Sing}\ d'\ X' = \mathsf{Sing}\ (\downarrow_{X'} d')\ (\Downarrow X')$	by def.
$\downarrow_X d = \downarrow_{X'} d' \in \mathsf{Nf}$	by ind. hyp.
$\Downarrow X = \Downarrow X' \in \mathsf{Nf}$	by ind. hyp.
$\mathsf{Sing}\ (\downarrow_X d)\ (\Downarrow X) = \mathsf{Sing}\ (\downarrow_X d)\ (\Downarrow X) \in \mathsf{Nf}$	by Lem. 35.

2. Case $\mathsf{Fun}\ X\ F = \mathsf{Fun}\ X'\ F' \in T$.

a) The partial function $\uparrow\_\,\_$ maps neutrals to related elements in the corresponding PER.

$k = k' \in \mathsf{Ne}$	hypothesis	
$d = d' \in [X]$	hypothesis	(*)
$X = X' \in T$	by inversion	(†)
$F\ d = F'\ d' \in T$	by inversion	(**)
$\downarrow_X d = \downarrow_{X'} d' \in \mathsf{Nf}$	by ind. hyp. on (*) and (†)	
$\mathsf{App}\ k\ (\downarrow_X d) = \mathsf{App}\ k'\ (\downarrow_{X'} d') \in \mathsf{Ne}$	by Lem. 35	(‡)
$\uparrow_{F\ d}\ (\mathsf{App}\ k\ (\downarrow_X d)) =$ $\uparrow_{F'\ d'}\ (\mathsf{App}\ k'\ (\downarrow_{X'} d')) \in [F\ d]$	by ind. hyp. on (**) and (‡)	
$\uparrow_{\mathsf{Fun}\ X\ F} k = \uparrow_{\mathsf{Fun}\ X'\ F'} k' \in [\mathsf{Fun}\ X\ F]$	by def.	

b) The partial function $\downarrow\_\,\_$ maps related elements to related normal forms.

$X = X' \in T$	by inversion	(*)
$f = f' \in [\mathsf{Fun}\ X\ F]$	hypothesis	
$k = k' \in \mathsf{Ne}$	hypothesis	
$\uparrow_X k = \uparrow_{X'} k' \in [X]$	by ind. hyp. on (*)	(†)
$d := \uparrow_X k$	abbreviation	
$d' := \uparrow_{X'} k'$	abbreviation	
$F\ d = F'\ d' \in T$	by inversion and (†)	(**)
$f \cdot d = f' \cdot d' \in [F\ d]$	definition of $[\mathsf{Fun}\ X\ F]$	(‡)
$\downarrow_{F\ d} (f \cdot d) = \downarrow_{F'\ d'} (f' \cdot d') \in \mathsf{Nf}$	by ind. hyp. on (‡)	
$(\downarrow_{\mathsf{Fun}\ X\ F} f) \cdot k =$ $(\downarrow_{\mathsf{Fun}\ X'\ F'} f') \cdot k' \in \mathsf{Nf}$	by def.	
$\downarrow_{\mathsf{Fun}\ X\ F} f = \downarrow_{\mathsf{Fun}\ X'\ F'} f' \in \mathsf{Nf}$	by Lem. 35	

c) The function $\Downarrow\_$ maps related elements in T to normal forms.

$X = X' \in T$	by inversion	(*)
$\Downarrow X = \Downarrow X' \in \mathsf{Nf}$	by ind. hyp. on (*)	(**)
$k = k' \in \mathsf{Ne}$	hypothesis.	
$\uparrow_X k = \uparrow_{X'} k' \in [X]$	by ind. hyp. on (*)	(†)
$d := \uparrow_X k$	abbr.	
$d' := \uparrow_{X'} k'$	abbr.	
$F\ d = F'\ d' \in T$	by inversion and (†)	(‡)
$\Downarrow (F\ d) = \Downarrow (F'\ d') \in \mathsf{Nf}$	by ind. hyp. on (‡)	
$\Downarrow (\mathsf{Fun}\ X\ F) = \Downarrow (\mathsf{Fun}\ X'\ F') \in \mathsf{Nf}$	by Lem. 35	

□

Let us recapitulate what we have achieved: we have defined a PER model over the domain D; then we defined a family of functions $\downarrow_X$ indexed over denotation of types with the property that when applied to elements in the corresponding PER we get back elements which will be reified as normal forms. In fact, we have the stronger result that whenever we apply $\downarrow_X$ to two related elements $d = d' \in [X]$ we get elements to be reified as the same term.

Now we define evaluation which clearly satisfies the environment model conditions in Def. 30; hence, we have a model and, using Thm. 34, we conclude completeness for our normalisation algorithm.

Definition 38 (Semantics). Evaluation of substitutions and terms into D is defined inductively by the following equations.

$$\begin{aligned}
[\![\mathsf{U}]\!]d &= \mathsf{U} & [\![\mathsf{id}]\!]d &= d\\
[\![\mathsf{Fun}\ A\ B]\!]a &= \mathsf{Fun}\ ([\![A]\!]d)\ F & [\![\langle\rangle]\!]d &= \top\\
&\quad \text{where } F\ e = [\![B]\!]((d, e)) & [\![\sigma\ \delta]\!]d &= [\![\sigma]\!]([\![\delta]\!]d)\\
[\![\{t\}_A]\!]d &= \mathsf{Sing}\ ([\![t]\!]d)\ ([\![A]\!]d) & [\![(\sigma, t)]\!]d &= ([\![\sigma]\!]d, [\![t]\!]d)\\
[\![t\ \sigma]\!]d &= [\![t]\!]([\![\sigma]\!]d) & [\![p]\!]d &= \mathsf{fst}\ d\\
[\![\lambda t]\!]d &= \mathsf{lam}\ (d' \mapsto [\![t]\!](d, d'))\\
[\![\mathsf{App}\ t\ u]\!]d &= ([\![t]\!]d) \cdot ([\![u]\!]d)\\
[\![q]\!]d &= \mathsf{snd}\ d
\end{aligned}$$

Theorem 39 (Completeness of NbE). Let $\Gamma \vdash t = t' \colon A$ and let also $d \in [\![\Gamma]\!]$, then $\downarrow_{[\![A]\!]d} ([\![t]\!]d) = \downarrow_{[\![A]\!]d} ([\![t']\!]d) \in \mathsf{Nf}$.

Proof. By Thm. 34 and Lem. 38. □

Calculus λ^{Irr} with proof irrelevance

We extend all the definitions concerning the construction of the model.

Definition 39 (Extension of domain D).

$$\begin{aligned}
D = \ldots &\oplus D \times [D \to D] \oplus D \oplus D\\
&\oplus \mathbb{O} \oplus \mathbb{O} \oplus D \oplus [D \to D] \times D \times [D \to [D \to D]] \times D\\
&\oplus D \oplus \mathbb{O} \oplus \mathbb{N} \oplus \mathbb{N} \times \mathbb{N} \oplus \mathbb{N} \times [D \to D] \times D^{\omega} \times D\ .
\end{aligned}$$

In the last summand D^{ω} is the set of finite tuples over D. We use the following notations for the injections into D:

$\mathsf{Sum}(d, F)$	$\mathsf{Fst}\ d,\ \ \mathsf{Snd}\ d$	for $d \in D, F \in [D \to D]$
$\mathsf{zero}\ \ \mathsf{Nat}\ \star$	$\mathsf{suc}\ d\ \ \mathsf{Prf}\ d$	for $d \in D$
N_n	c_i^n	for $i, n \in \mathbb{N}$
$\mathsf{Natrec}(F, d, g, d')$	for $d, d' \in D, F \in [D \to D], g \in [D \to [D \to D]]$	
$\mathsf{Case}^n(F, \vec{d}, d')$	for $d, d' \in D, F \in [D \to D], \vec{d} \in D^{\omega}, n \in \mathbb{N}$.	

In this extension, the injections Fst, Snd, Natrec, and Case construct neutral elements k. Soundness for the calculus ($\vdash^\star$) requires the canonical element for proof-irrelevant types ($\star$) to be in every PER; thus we need to redefine application $\_ \cdot \_$ to have $\star \in [\mathsf{Fun}\ X\ F]$:

$$\star \cdot d = \star \ .$$

We also redefine the projections p^* and q^* to account for neutrals and because they are used in the definition of $\coprod XF$, which will be used as the denotation of sigma types.

$$p^*\ d = \begin{cases} d_1 & \text{if } d = (d_1, d_2) \\ \star & \text{if } d = \star \\ \mathsf{Proj}_1(d) & \text{otherwise} \end{cases} \qquad q^*\ d = \begin{cases} d_2 & \text{if } d = (d_1, d_2) \\ \star & \text{if } d = \star \\ \mathsf{Proj}_2(d) & \text{otherwise} \end{cases}$$

Definition 40 (Readback function).

$$\begin{aligned} \mathsf{R}_j\ \mathsf{Nat} &= \mathsf{Nat} & \mathsf{R}_j\ (d, d') &= (\mathsf{R}_j\ d, \mathsf{R}_j\ d') \\ \mathsf{R}_j\ \mathsf{zero} &= \mathsf{zero} & \mathsf{R}_j\ (\mathsf{Proj}_1(d)) &= \mathsf{fst}\ (\mathsf{R}_j\ d) \\ \mathsf{R}_j\ (\mathsf{suc}\ d) &= \mathsf{suc}((\mathsf{R}_j\ d)) & \mathsf{R}_j\ (\mathsf{Proj}_2(d)) &= \mathsf{snd}\ (\mathsf{R}_j\ d) \\ \mathsf{R}_j\ (\mathsf{Prf}\ d) &= [(\mathsf{R}_j\ d)] & \mathsf{R}_j\ \star &= \star \\ \mathsf{R}_j\ (\mathsf{N}_n) &= \mathsf{N}_n & \mathsf{R}_j\ (c_i^n) &= c_i^n \end{aligned}$$

$$\begin{aligned} \mathsf{R}_j\ (\mathsf{Sum}(X, F)) &= \Sigma\ (\mathsf{R}_j\ X)\ (\mathsf{R}_{j+1}\ (F\ \mathsf{Var}\ j)) \\ \mathsf{R}_j\ (\mathsf{Natrec}(F, d, f, e)) &= \mathsf{natrec}((\mathsf{R}_{j+1}\ (F\ \mathsf{Var}\ j)), (\mathsf{R}_j\ d), (\mathsf{R}_j\ f), (\mathsf{R}_j\ e)) \\ \mathsf{R}_j\ (\mathsf{Case}^n(F, \langle d_0, \ldots, d_{n-1}\rangle, e)) &= \mathsf{case}^n\ (\mathsf{R}_{j+1}\ (F\ \mathsf{Var}\ j))\ (\mathsf{R}_j\ d_0) \cdots \\ &\qquad (\mathsf{R}_j\ d_{n-1})\ (\mathsf{R}_j\ e) \end{aligned}$$

We define inductively new PERs for interpreting naturals and finite types. Note that C_0 and C_1 are irrelevant, in this way we can model η-expansion for N_0 and N_1; $|X|$ is also irrelevant, even when X distinguishes its elements.

Definition 41 (More semantic types).

1. N is the smallest PER over D, such that
   a) $\mathsf{Ne} \subseteq N$
   b) $\mathsf{zero} = \mathsf{zero} \in N$
   c) $\mathsf{suc}\ d = \mathsf{suc}\ d' \in N$, if $d = d' \in N$
2. If $X \in \mathrm{PER}(D)$ then $|X| := \{(d, d') \mid d, d' \in \mathit{dom}X \cup \{\star\}\} \in \mathrm{PER}(D)$.
3. $C_0 = |\emptyset| = \{(\star, \star)\}$,
4. $C_1 = |\{c_0^1\}| = \{(d, d') \mid d, d' \in \{\star, c_0^1\}\}$,
5. $C_n = \{(c_i^n, c_i^n) \mid i < n\} \cup \mathsf{Ne}$, for $n \geqslant 2$.

We add new clauses in the definitions of the partial equivalences for universe and types, these clauses do not affect the well-foundedness of the order $\sqsubset$ defined in Rem. 14, but now we have that N_n and Nat are also minimal elements for that order.

Definition 42 (Extension of U and T).

1. Inductive definition of $U \in \mathrm{PER}(\mathrm{D})$.
   a) If $X = X' \in U$, and for all $d = d' \in [X]$, $F\,d = F'\,d' \in U$, then $\mathsf{Sum}(X, F) = \mathsf{Sum}(X', F') \in U$.
   b) $\mathsf{Nat} = \mathsf{Nat} \in U$,
   c) $\mathsf{N}_n = \mathsf{N}_n \in U$,
   d) if $X = X' \in U$, then $\mathsf{Prf}\,X = \mathsf{Prf}\,X' \in U$.
2. Inductive definition of $T \in \mathrm{PER}(\mathrm{D})$.
   a) If $X = X' \in T$, and for all $d = d' \in [X]$, $F\,d = F'\,d' \in T$, then $\mathsf{Sum}(X, F) = \mathsf{Sum}(X', F') \in T$.
   b) if $X = X' \in T$, then $\mathsf{Prf}\,X = \mathsf{Prf}\,X' \in T$.
3. Recursive definition of $[\_] \in \mathit{dom}(T) \to \mathrm{PER}(\mathrm{D})$.
   a) $[\mathsf{Sum}(X, F)] = \coprod [X]\,(d \mapsto [F\,d])$,
   b) $[\mathsf{N}_n] = C_n$
   c) $[\mathsf{Nat}] = N$,
   d) if $X \in \mathit{dom}T$, then $[\mathsf{Prf}\,X] = \{(\star, \star)\}$.

Note that in the PER model, all propositions $\mathsf{Prf}\,X$ are inhabited. In fact, all types are inhabited, for there is a reflection from variables into any type, be it empty or not. So, the PER model is unsuited for refuting propositions. However, the logical relation we define in the next section will only be inhabited for non-empty types.

Remark 15. It can be proved by induction on $X \in T$ that $\star \in [X]$.

Definition 43 (Reflection and reification, cf. 37).

$$\uparrow_{\mathsf{Sum}(X,F)} k = (\uparrow_X \mathsf{Fst}\,k, \uparrow_{F\,(\uparrow_X \mathsf{Fst}\,k)} \mathsf{Snd}\,k)$$
$$\downarrow_{\mathsf{Sum}(X,F)} d = (\downarrow_X p^*\,d, \downarrow_{F\,(p^*\,d)} q^*\,d)$$

$$\begin{array}{lll} \uparrow_{\mathsf{Nat}} k = k & & \downarrow_{\mathsf{Nat}} d = d \\ \uparrow_{\mathsf{N}_0} k = \star & & \downarrow_{\mathsf{N}_0} d = \star \\ \uparrow_{\mathsf{N}_1} k = c_0^1 & & \downarrow_{\mathsf{N}_1} d = c_0^1 \\ \uparrow_{\mathsf{N}_n} k = k & \text{for } n \geqslant 2 & \downarrow_{\mathsf{N}_n} d = d \\ \uparrow_{\mathsf{Prf}\,X} k = \star & & \downarrow_{\mathsf{Prf}\,X} d = \star \end{array}$$

$$\Downarrow \mathsf{Sum}(X, F) = \mathsf{Sum}((\Downarrow X), (d \mapsto \Downarrow (F \uparrow_X d))) \qquad \Downarrow \mathsf{Nat} = \mathsf{Nat}$$
$$\Downarrow \mathsf{N}_n = \mathsf{N}_n \qquad \Downarrow \mathsf{Prf}\,X = \mathsf{Prf}\,(\Downarrow X)$$

To give semantics to eliminators for data types we define partial functions $\mathsf{natrec} : [\mathrm{D} \to \mathrm{D}] \times \mathrm{D} \times \mathrm{D} \times \mathrm{D} \to \mathrm{D}$, and $\mathsf{case} : [\mathrm{D} \to \mathrm{D}] \times \mathrm{D} \times \mathrm{D} \times \mathrm{D} \to \mathrm{D}$.

Definition 44 (Eliminations on D).

1. Elimination operator for naturals.

$$\begin{aligned}
\mathsf{natrec}(F, d, f, \star) &= \star \\
\mathsf{natrec}(F, d, f, \mathsf{zero}) &= d \\
\mathsf{natrec}(F, d, f, \mathsf{suc}\, e) &= (f \cdot e) \cdot \mathsf{natrec}(F, d, f, e) \\
\mathsf{natrec}(F, d, f, k) &= \uparrow_{F\ k} (\mathsf{Natrec}(d' \mapsto \Downarrow F\ d', \\
&\qquad \downarrow_{F\ \mathsf{zero}} d, \\
&\qquad \mathsf{lam}\, d' \mapsto (\mathsf{lam}\, e' \mapsto \downarrow_{F\ (\mathsf{suc}\, d')} f \cdot d' \cdot e'), \\
&\qquad k))
\end{aligned}$$

2. Elimination operator for finite types.

$$\begin{aligned}
\mathsf{case}^n(F, \langle d_0, \ldots, d_{n-1}\rangle, \star) &= \star \\
\mathsf{case}^n(F, \langle d_0, \ldots, d_{n-1}\rangle, c_i^n) &= d_i \\
\mathsf{case}^n(F, \langle c_0^n, \ldots, c_{n-1}^n\rangle, d) &= d \\
\mathsf{case}^n(F, \langle d_0, \ldots, d_{n-1}\rangle, k) &= \uparrow_{F\ k} (\mathsf{Case}^n(e \mapsto \Downarrow F\ e, \\
&\qquad \langle \downarrow_{F\ c_0^n} d_0, \ldots, \downarrow_{F\ c_{n-1}^n} d_{n-1}\rangle, k))
\end{aligned}$$

Remark 16. If for all $d = d' \in N$, $F\ d = F'd' \in T$, and $z = z' \in [F\ \mathsf{zero}]$, and for all $d = d' \in N$ and $e = e' \in [F\ d]$, $s \cdot d \cdot e = s' \cdot d' \cdot e' \in [F(\mathsf{suc}\, d)]$, and $d = d' \in N$ then $\mathsf{natrec}(F, z, s, d) = \mathsf{natrec}(F, z, s, d') \in [F\ d]$.

With these new definitions we can now give the semantic equations for the new constructs.

Definition 45 (Extension of interpretation).

$$\begin{aligned}
[\![\Sigma\, A\, B]\!]d &= \mathsf{Sum}(([\![A]\!]d), (d' \mapsto [\![B]\!](d, d'))) & [\![N_n]\!]d &= N_n \\
[\![\mathsf{Nat}]\!]d &= \mathsf{Nat} & [\![[A]]\!]d &= \mathsf{Prf}\ [\![A]\!]d \\
[\![\mathsf{fst}\ t]\!]d &= p^*\ [\![t]\!]d & [\![\mathsf{snd}\ t]\!]d &= q^*\ [\![t]\!]d \\
[\![(t, t')]\!]d &= ([\![t]\!]d, [\![t']\!]d) & [\![\mathsf{zero}]\!]d &= \mathsf{zero} \\
[\![\mathsf{suc}(t)]\!]d &= \mathsf{suc}\ [\![t]\!]d & [\![\star]\!]d &= \star \\
[\![[a]]\!]d &= \star & [\![c_i^n]\!]d &= c_i^n \\
[\![b\ \mathsf{where}^B\ t]\!]d &= [\![b]\!](d, \star)
\end{aligned}$$

$$\begin{aligned}
[\![\mathsf{natrec}(B, z, s, t)]\!]d &= \mathsf{natrec}(e \mapsto [\![B]\!](d, e), [\![z]\!]d, [\![s]\!]d, [\![t]\!]d) \\
[\![\mathsf{case}^n\ B\ t_0 \cdots t_{n-1}\ t]\!]d &= \mathsf{case}^n(e \mapsto [\![B]\!](d, e), \langle [\![t_0]\!]d, \ldots, [\![t_{n-1}]\!]d\rangle, [\![t]\!]d)
\end{aligned}$$

Lemma 40 (Laws of proof elimination). β, η, and associativity for where are modeled by the extended applicative structure.

Proof. The proofs of soundness for (PRF-β) and (PRF-η) have the same structure, so we show only the first one.

- (PRF-β) $b\ \mathsf{where}^B\ [a] = b\ (\mathsf{id}, a)$

  $$\begin{aligned}
  &[\![b\ \mathsf{where}^B\ [a]]\!]d \\
  &= [\![b]\!](d, \star) && \text{def. of semantics for } b\ \mathsf{where}^B\ [a] \\
  &= [\![b]\!](d, [\![a]\!]d) && \text{ind. hypothesis on } \Gamma.A.A\ p \vdash b\ p = b\ (p\ p, q) \colon B\ p\ p \\
  &= [\![b]\!]([\![(\mathsf{id}, a)]\!]d) && \text{def. of semantics for substitutions} \\
  &= [\![b\ (\mathsf{id}, a)]\!]d
  \end{aligned}$$

- (PRF-ASSOC) $a\ \mathsf{where}^A\ (b\ \mathsf{where}^B\ c) = (a\ (p\ p, q)\ \mathsf{where}^{A\ p}\ b)\ \mathsf{where}^B\ c$

  $$\begin{aligned}
  [\![a\ \mathsf{where}^A\ (b\ \mathsf{where}^B\ c)]\!]d &= [\![a]\!](d, \star) \\
  &= [\![a]\!](d, [\![b]\!](d, \star)) \\
  &= [\![a\ (p\ p, q)]\!]((d, \star), [\![b]\!](d, \star)) \qquad \square \\
  &= [\![a\ (p\ p, q)\ \mathsf{where}^{A\ p}\ b)]\!](d, \star) \\
  &= [\![(a\ (p\ p, q)\ \mathsf{where}^{A\ p}\ b)\ \mathsf{where}^B\ c]\!]d
  \end{aligned}$$

Theorem 41. All of lemmata 36, 38, and theorems 34, and 39 are valid for the extended calculus.

Note that we have defined a *proof-irrelevant* semantics for $(\vdash^\star)$ that collapses all elements of $[A]$ to $\star$, which leads to a more efficient implementation of the normalisation function. However, this semantics is not sound if λ^{Irr} is extended with singleton types interpreted analogously to C_1, i.e., $[\mathsf{Sing}\ d\ X] = |\{d\}_X|$, because it does not model (SING-EQ-EL). (We have $d = \star \in [\mathsf{Sing}\ d\ X]$ for all $d \in [X]$, but not necessarily $d = \star \in [X]$.) On the other hand, λ^{Irr} without $\star$ can be extended to singleton types as explained in the following remark.

Remark 17 (Extending λ^{Irr} by singleton types). Singleton types can be added straightforwardly if we employ a *proof-relevant* semantics: the domain D is not changed; in particular we have $\star \in D$, and it is readback as before, $R_j \star = \star$; hence $\star \in \mathrm{dom}(\mathsf{Nf})$.

Enumerated types are modelled in a uniform way: $[\mathsf{N}_n] = \{(c_i^n, c_i^n) \mid i < n\} \cup \mathsf{Ne}$; proof-irrelevance types $[A]$ are interpreted as the irrelevant PER with the same domain as the PER for A: $[\mathsf{Prf}\ X] = \{(d, d') \mid d, d' \in \mathrm{dom}([X])\}$. Reflection and reification for $\mathsf{Prf}\ X$ are defined respectively as

$$\uparrow_{\mathsf{Prf}\ X} d = \uparrow_X d \qquad \text{and} \qquad \downarrow_{\mathsf{Prf}\ X} d = \star\ .$$

With these definitions it is clear that the corresponding result for Lem. 38 is still valid.

Since $dom[\mathsf{Prf}\ X] = dom[X]$, introduction and elimination of proofs can be interpreted as follows

$$[\![[a]]\!]d = [\![a]\!]d \quad \text{and} \quad [\![b\ \mathsf{where}^B\ t]\!]d = [\![b]\!](d, [\![t]\!]d)\ ;$$

this model is sound with respect to the calculus ($\vdash$) extended with singleton types; hence Thm. 39 is valid.

Remark 18. As was previously said we cannot use this PER model for proving that there is no closed term in N_0. Instead, one can build up a PER model, in the sense of 4.2, of closed values, where $[\mathsf{N}_0] = \emptyset$. By soundness (Thm. 34) it follows that there is no possible derivation of $\vdash t : \mathsf{N}_0$.

4.3 Correctness of NbE

As in previous chapters we use logical relations to prove correctness of NbE. The section is similar to Sec. 3.3: as a corollary of the fundamental theorem (Thm. 46) we show that a term is related to its denotation with respect to some canonical environment; previously we prove, in Lem. 45, that if a term is logically related with some semantic element, then its reification will be judgmentally equal to the term. Composing these facts we obtain correctness. As a consequence of having correctness and completeness for NbE, one gets decidability for judgmentally equality: normalise both terms and check they are syntactically the same.

Logical relations

Definition 46 (Logical relations). We define simultaneously two families of binary relations:

1. If $\Gamma \vdash$ then $(\Gamma \vdash \_ \sim \_ \in T) \subseteq \{A \mid \Gamma \vdash A\} \times T$ shall be a Γ-indexed family of relations between well-formed syntactic types A and type values X.
2. If $\Gamma \vdash A \sim X \in T$ then $(\Gamma \vdash \_ : A \sim \_ \in [X]) \subseteq \{t \mid \Gamma \vdash t : A\} \times [X]$ shall be a (Γ, A, X)-indexed family of relations between terms t of type A and values d in PER $[X]$.

These relations are defined simultaneously by induction on $X \in T$.

1. Neutral types: $X \in \mathsf{Ne}$.
   a) $\Gamma \vdash A \sim X \in T$ iff for all $\Delta \leqslant^i \Gamma$, $\Delta \vdash A\,\mathsf{p}^i = \mathsf{R}_{|\Delta|} \Downarrow X$.
   b) $\Gamma \vdash t : A \sim d \in [X]$ iff $\Gamma \vdash A \sim X \in T$, and for all $\Delta \leqslant^i \Gamma$, $\Delta \vdash t\,\mathsf{p}^i = \mathsf{R}_{|\Delta|} \downarrow_X d : A\,\mathsf{p}^i$.
2. Universe.
   a) $\Gamma \vdash A \sim \mathsf{U} \in T$ iff $\Gamma \vdash A = \mathsf{U}$.
   b) $\Gamma \vdash t : A \sim X \in [\mathsf{U}]$ iff $\Gamma \vdash A = \mathsf{U}$, and $\Gamma \vdash t \sim X \in T$.
3. Singletons.
   a) $\Gamma \vdash A \sim \mathsf{Sing}\ d\ X \in T$ iff $\Gamma \vdash A = \{a\}_{A'}$ and $\Gamma \vdash a : A' \sim d \in [X]$.
   b) $\Gamma \vdash t : A \sim d' \in [\mathsf{Sing}\ d\ X]$ iff $\Gamma \vdash A = \{a\}_{A'}$ and $\Gamma \vdash t : A' \sim d \in [X]$, and $\Gamma \vdash A' \sim X \in T$.
4. Function spaces.

a) $\Gamma \vdash A \sim \mathsf{Fun}\ X\ F \in T$ iff $\Gamma \vdash A = \mathsf{Fun}\ A'\ B$, and $\Gamma \vdash A' \sim X \in T$, and $\Delta \vdash B\ (p^i, s) \sim F\ d \in T$ for all $\Delta \leqslant^i \Gamma$ and $\Delta \vdash s : A'\ p^i \sim d \in [X]$.

b) $\Gamma \vdash t : A \sim f \in [\mathsf{Fun}\ X\ F]$ iff $\Gamma \vdash A = \mathsf{Fun}\ A'\ B$, $\Gamma \vdash A' \sim X$, and $\Delta \vdash \mathsf{App}\ (t\ p^i)\ s : B\ (p^i, s) \sim f \cdot d \in [F\ d]$ for all $\Delta \leqslant^i \Gamma$ and $\Delta \vdash s : A'\ p^i \sim d \in [X]$.

Calculus λ^{Irr} with proof irrelevance We present next the definition of logical relations for λ^{Irr}; then we prove the technical results about logical relations for both calculi.

Definition 47.

1. Sigma types.

   a) $\Gamma \vdash A \sim \mathsf{Sum}(X, F)$ iff $\Gamma \vdash A = \Sigma\ A'\ B'$ and $\Gamma \vdash A' \sim X$ and for all $\Delta \leqslant^i \Gamma$ and $\Delta \vdash s : A'\ p^i \sim d \in [X]$, $\Delta \vdash B'\ (p^i, s) \sim F\ d$.

   b) $\Gamma \vdash t : A \sim d \in [\mathsf{Sum}(X, F)]$ iff $\Gamma \vdash A = \Sigma\ A'\ B'$ and $\Gamma \vdash \mathsf{fst}\ t : A' \sim p^*\ d \in [X]$ and $\Gamma \vdash \mathsf{snd}\ t : B'\ (\mathsf{id}_\Gamma, \mathsf{fst}\ t) \sim q^*\ d \in [F\ (p^*\ d)]$.

2. Natural numbers.

   a) $\Gamma \vdash A \sim \mathsf{Nat}$ iff $\Gamma \vdash A = \mathsf{Nat}$.

   b) $\Gamma \vdash t : A \sim d \in [\mathsf{Nat}]$ iff $\Gamma \vdash A \sim \mathsf{Nat}$ and for all $\Delta \leqslant^i \Gamma$, $\Delta \vdash t\ p^i = R_{|\Delta|}\ d : \mathsf{Nat}$.

3. Finite types.

   a) $\Gamma \vdash A \sim N_n$ iff $\Gamma \vdash A = N_n$.

   b) $\Gamma \vdash t : A \sim d \in [N_n]$ iff $\Gamma \vdash A \sim N_n$ and for all $\Delta \leqslant^i \Gamma$, $\Delta \vdash t\ p^i = R_{|\Delta|}\ d : N_n$.

4. Proof-irrelevance types.

   a) $\Gamma \vdash A \sim \mathsf{Prf}\ X \in T$ iff $\Gamma \vdash A = [A']$ and $\Gamma \vdash A' \sim X \in T$.

   b) $\Gamma \vdash t : A \sim d \in [\mathsf{Prf}\ X]$ iff $\Gamma \vdash A \sim \mathsf{Prf}\ X$.

Technical results

The following technical lemmata show that the logical relations are preserved by judgmental equality, weakening of the judgement, and the equalities on the corresponding PERs. These lemmata are proved simultaneously for types and terms.

Lemma 42 (Closure under conversion). Let $\Gamma \vdash A \sim X \in T$ and $\Gamma \vdash A = A'$. Then,

1. $\Gamma \vdash A' \sim X \in T$, and
2. if $\Gamma \vdash t : A \sim d \in [X]$ and $\Gamma \vdash t = t' : A$ then $\Gamma \vdash t' : A' \sim d \in [X]$.

Proof. By induction on $X \in T$.

1. Types; in all cases we use symmetry and transitivity to show the conditions. We only show the case for $\mathsf{Fun}\ X\ F$.

a) $X = \mathsf{Fun}\ X'\ F$:

$\Gamma \vdash A = \mathsf{Fun}\ B\ C$	by definition	(*)
$\Gamma \vdash B \sim X'$	by definition	
$\Delta \vdash C\ (p^i, s) \sim F\ d \in T$ for all $\Delta \leqslant^i \Gamma$ and $\Delta \vdash s : B\ p^i \sim d \in [X']$	by definition	
$\Gamma \vdash A' = \mathsf{Fun}\ B\ C$	by sym. and trans. on (*)	

b) $\mathsf{N}_n \in T$.

$\Gamma \vdash t : A \sim d \in [\mathsf{N}_1]$	hypothesis	(*)
$\Gamma \vdash t = t' : A$	hypothesis	(†)
$\Delta \vdash t\ p^i = \mathsf{R}_i\ d : A\ p^i$	by inversion on (*)	(**)
$\Delta \vdash t\ p^i = t'\ p^i : A\ p^i$	by congruence on (†)	(‡)
$\Delta \vdash t'\ p^i = \mathsf{R}_i\ d : A\ p^i$	by sym. and trans. on (**) and (‡)	

2. Terms. As in the case for types, we use symmetry and transitivity. We show only the case for singletons and functions.

a) $X = \mathsf{Sing}\ d\ X'$:

$\Gamma \vdash A = \{b\}_B$	by hypothesis	(*)
$\Gamma \vdash B \sim X' \in T$	by hypothesis	
$\Gamma \vdash t : B \sim d \in [X']$	by hypothesis	(†)
$\Gamma \vdash A' = \{b\}_B$	by sym. and trans. on (*)	
$\Gamma \vdash t' : B \sim d \in [X']$	By i.h. on (†)	

b) $X = \mathsf{Fun}\ X'\ F$:

$\Gamma \vdash A = \mathsf{Fun}\ B\ C$	by hypothesis	(*)
$\Gamma \vdash B \sim X'$	by hypothesis	
$\Delta \vdash \mathsf{App}\ t\ p^i\ s : C\ (p^i, s) \sim f \cdot d \in [F\ d]$ for all $\Delta \leqslant^i \Gamma, \Delta \vdash s : B\ p^i \sim d \in [X']$	by hypothesis	
$\Gamma \vdash A' = \mathsf{Fun}\ B\ C$	by sym. and trans. on (*)	(†)
$\Delta \vdash \mathsf{App}\ t\ p^i\ s = \mathsf{App}\ t'\ p^i\ s : C\ (p^i, s)$	by congruence on (†)	(‡)
$\Delta \vdash \mathsf{App}\ t'\ p^i\ s : C\ (p^i, s) \sim f \cdot d \in [F\ d]$	by i.h. on (‡)	

□

Lemma 43 (Monotonicity). Let $\Delta \leqslant^i \Gamma$, then

1. if $\Gamma \vdash A \sim X \in T$, then $\Delta \vdash A\ p^i \sim X \in T$; and
2. if $\Gamma \vdash t : A \sim d \in [X]$, then $\Delta \vdash t\ p^i : A\ p^i \sim d \in [X]$.

Proof. By induction on $X \in T$. This property is trivial for the base cases; for singletons is obtained by applying the i.h. We show two cases.

1. Let $X = \mathsf{Fun}\, X'\, F$.

$\Gamma \vdash A = \mathsf{Fun}\, B\, C$	by hypothesis	(*)
$\Gamma \vdash B \sim X'$		(†)
$\Theta \vdash C\,(p^i, s) \sim F\, d \in T$	by hypothesis	
for all $\Theta \leqslant^i \Gamma, \Theta \vdash s : B\, p^i \sim d \in [X']$		
$\Delta \vdash A\, p^i = \mathsf{Fun}\,(B\, p^i)\,(C\,(p^i\, p, q))$	by congruence on (*)	
$\Delta \vdash B\, p^i \sim X$	by i.h. on (†)	
$\Theta' \vdash s : (B\, p^i)\, p^j \sim d \in [X]$, with $\Theta' \leqslant^j \Delta$	hypothesis	
$\Theta' \vdash s : B\, p^{i+j} \sim d \in [X]$	by rem. 11 and 42	(‡)
$\Theta' \vdash C\,(p^{i+j}\, q, s) \sim F\, d$	by hyp. using (‡)	
$\Theta' \vdash C\,(p^i\, p, q)\,(p^j, s) \sim F\, d$	By congruence and 42	

2. $\mathsf{Prf}\, X \in T$. As mentioned earlier if $\Gamma \vdash A \sim \mathsf{Prf}\, X \in T$ then $\Gamma \vdash \_ : A \sim \_ \in [\mathsf{Prf}\, X]$ is non-empty if and only if $\Gamma \vdash \_ : A$ is not empty.

$\Gamma \vdash t : A \sim d \in [\mathsf{Prf}\, X]$	hypothesis	(*)
$\Gamma \vdash t : A$	by inversion on (*)	(†)
$\Gamma \vdash A \sim \mathsf{Prf}\, X \in T$	by inversion on (*)	(**)
$\Delta \vdash t\, p^i : A\, p^i$	by weakening on (†)	
$\Delta \vdash A\, p^i \sim \mathsf{Prf}\, X \in T$	by monotonicity for types on (**)	
$\Delta \vdash t\, p^i : A\, p^i \sim d \in [\mathsf{Prf}\, X]$	by definition of log. rel.	

□

We do not show proofs for the second part, since the most involved case is dealt analogously to the case for $\mathsf{Fun}\, X'\, F$.

Lemma 44 (Closure under PERs). Let $\Gamma \vdash A \sim X \in T$, then

1. if $X = X' \in T$, then $\Gamma \vdash A \sim X' \in T$; and
2. if $\Gamma \vdash t : A \sim d \in [X]$ and $d = d' \in [X]$, then $\Gamma \vdash t : A \sim d' \in [X]$.

Proof. By induction on $X = X' \in T$. Note that the first part for the base cases is trivial; the second point is also trivial for $X \in \mathsf{Ne}$. Thus we do not show those parts of the proof.

1. Types.
   a) $\mathsf{Sing}\, d\, X = \mathsf{Sing}\, d'\, X'$.

$\Gamma \vdash A = \{b\}_B$	by hypothesis	(*)
$\Gamma \vdash B \sim X \in T$	by hypothesis	
$\Gamma \vdash t : B \sim d \in [X]$	by hypothesis	(†)
$\Gamma \vdash t : B \sim d' \in [X']$	By i.h. on (*) and (†)	

b) $\mathsf{Fun}\ X\ F = \mathsf{Fun}\ X'\ F'$.

$\Gamma \vdash A = \mathsf{Fun}\ B\ C$	by hypothesis	
$\Gamma \vdash B \sim X'$	by hypothesis	(*)
$\Theta \vdash C\ (p^i, s) \sim F\ d \in T$	by hypothesis	(†)
for all $\Theta \leqslant^i \Gamma, \Theta \vdash s : B\ p^i \sim d \in [X']$		
$\Gamma \vdash B \sim X' \in T$	By i.h. on (*)	
$\Theta \vdash B\ (p^i, s) \sim F'\ d \in T$	by i.h. on (†)	

2. Terms.

a) $e = e' \in [\mathsf{Sing}\ d\ X]$.

$\Gamma \vdash A = \{b\}_B$	by hypothesis	
$\Gamma \vdash B \sim X \in T$	by hypothesis	(*)
$\Gamma \vdash t : B \sim d \in [X]$	by hypothesis	(†)
$e' = d \in [X]$	by def. of $e = e' \in [\mathsf{Sing}\ d\ X]$	(**)
$\Gamma \vdash t : B \sim e' \in [X]$	by i.h. on (*), (†), and (**),	

b) $f = f' \in [\mathsf{Fun}\ X\ F]$.

$\Gamma \vdash A = \mathsf{Fun}\ B\ C$

$\Gamma \vdash B \sim X$ (*)

$\Delta \vdash \mathsf{App}\ t\ p^i\ s : C\ (p^i, s) \sim f \cdot d \in [F\ d]$ (†)

for all $\Delta \leqslant^i \Gamma$ and $\Delta \vdash s : B\ p^i \sim d \in [X]$. (**)

By i.h. on (*) and (**) and monotonicity 43

$$\Delta \vdash s : A'\ p^i \sim d' \in [X'] \ .$$

By i.h. on (†)

$$\Delta \vdash \mathsf{App}\ (t\ p^i)\ s : B\ (p^i, s) \sim f' \cdot d' \in [F\ d'] \ .$$

c) $d = d' \in [\mathsf{Sum}(X, F)]$.

$d = d' \in [\mathsf{Sum}(X, F)]$	hypothesis	(*)
$\Gamma \vdash t : A \sim d \in [\mathsf{Sum}(X, F)]$	hypothesis	(**)
$\Gamma \vdash A = \Sigma\ A'\ B$	by inversion on (**)	
$\Gamma \vdash \Sigma\ A'\ B \sim \mathsf{Sum}(X, F) \in T$	by inversion on (**)	
$\Gamma \vdash \mathsf{fst}\ t : A' \sim p^*\ d \in [X]$	by inversion on (*)	(†)
$\Gamma.A' \vdash \mathsf{snd}\ t : B\ (\mathsf{id}, \mathsf{fst}\ t) \sim$ $q^*\ d \in [F\ p^*\ d]$	by inversion on (**)	(‡)
$p^*\ d = p^*\ d' \in [X]$	by definition of (*)	(ǂ)
$q^*\ d = q^*\ d' \in [F\ p^*\ d]$	by definition of (*)	(♯)
$\Gamma \vdash \mathsf{fst}\ t : A' \sim p^*\ d' \in [X]$	by ind. hyp. on (†) and (ǂ)	
$\Gamma.A' \vdash \mathsf{snd}\ t : B\ (\mathsf{id}, \mathsf{fst}\ t) \sim$ $q^*\ d' \in [F\ p^*\ d']$	by ind. hyp. on (‡) and (♯).	

□

The following lemma plays a key rôle in the proof of soundness. It proves that if a term is related to some element in (some PER), then it is convertible to the reification of the corresponding element in the PER of normal forms.

Lemma 45. Let $\Gamma \vdash A \sim X \in T$. Then,

1. $\Gamma \vdash A = \mathsf{R}_{|\Gamma|} \Downarrow X$,
2. if $\Gamma \vdash t \colon A \sim d \in [X]$ then $\Gamma \vdash t = \mathsf{R}_{|\Gamma|} \downarrow_X d \colon A$; and
3. if $k \in \mathsf{Ne}$ and for all $\Delta \leqslant^i \Gamma$, $\Delta \vdash t\,p^i = \mathsf{R}_{|\Delta|}\, k \colon A\,p^i$, then $\Gamma \vdash t \colon A \sim \uparrow_X k \in [X]$.

Proof. By induction on $X \in T$. By induction on $X \in T$. For a better organisation of the proof we show the proofs for each point separately.

1. $\Gamma \vdash A = \mathsf{R}_{|\Gamma|} \Downarrow X$. We skip the part for the minimal elements in T.

   a) $\mathsf{Sing}\ d\ X$:

$\Gamma \vdash A' = \mathsf{R}_{\lvert\Gamma\rvert} \Downarrow X$	by ind. hyp.
$\Gamma \vdash t = \mathsf{R}_{\lvert\Gamma\rvert} \downarrow_X d \colon A'$	by ind. hyp.
$\Gamma \vdash \{a\}_{A'} = \{\mathsf{R}_{\lvert\Gamma\rvert} \downarrow_X d\}_{\mathsf{R}_{\lvert\Gamma\rvert} \Downarrow X}$	by congruence and transitivity

   b) $\mathsf{Fun}\ X\ F$:

$\Gamma \vdash A' = \mathsf{R}_{\lvert\Gamma\rvert} \Downarrow X$	by ind. hyp.
$\Delta \vdash B\,(p^i, s) = \mathsf{R}_{\lvert\Delta\rvert}\,(\Downarrow F\ d)$	(*)
for any $\Delta \leqslant^i \Gamma, \Delta \vdash s \colon A'\,p^i \sim d \in [X]$	
$\Gamma.A' \vdash q \colon A'\,p \sim \uparrow_X (\mathsf{Var}\,\lvert\Gamma\rvert)$	by ind. hyp. (†)
$\Gamma.A' \vdash B\,(p, q) =$	
$\quad \mathsf{R}_{\lvert\Gamma.A'\rvert}\,(\Downarrow (F\ (\uparrow_X (\mathsf{Var}\,\lvert\Gamma\rvert))))$	instantiating (*) on (†)
$\Gamma.A' \vdash B = \mathsf{R}_{\lvert\Gamma.A'\rvert}\,(\Downarrow F\ (\uparrow_X (\mathsf{Var}\,\lvert\Gamma\rvert)))$	by Lem. 42.

2. $\Gamma \vdash t = \mathsf{R}_{|\Gamma|} \downarrow_X d \colon A$. We skip the part for the minimal elements in T.

   a) $d' \in [\mathsf{Sing}\ d\ X]$:

$\Gamma \vdash A = \{b\}_B$	(*)
$\Gamma \vdash B \sim X \in T$	
$\Gamma \vdash t \colon B \sim d \in [X]$	(†)
$\Gamma \vdash t = \mathsf{R}_{\lvert\Gamma\rvert} \downarrow_X d \colon B$	by ind. hyp. in (†)
$\Gamma \vdash t = \mathsf{R}_{\lvert\Gamma\rvert} \downarrow_X d \colon \{t\}_B$	by conversion and (SING-EQ-1)
$\Gamma \vdash t = \mathsf{R}_{\lvert\Gamma\rvert} \downarrow_X d \colon A$	by conversion

b) $f \in [\mathsf{Fun}\ X\ F]$:

$\Gamma.A' \vdash q : A'\ p \sim \uparrow_X \mathsf{Var}\,\lvert\Gamma\rvert \in [X]$	by ind. hyp. on the third part
$d := \uparrow_X \mathsf{Var}\,\lvert\Gamma\rvert$	abbreviation
$\Gamma.A' \vdash \mathsf{App}\ (t\ p)\ q : B\ (p, q) \sim$	
$\quad f \cdot d \in [F\ d]$	by def. of the logical relation
$\Gamma.A' \vdash \mathsf{App}\ (t\ p)\ q$	
$R_{\lvert\Gamma.A\rvert} \downarrow_{F\ d} f \cdot d : B\ (p, q)$	by ind. hyp.
$\Gamma.A' \vdash \mathsf{App}\ (t\ p)\ q =$	
$\quad R_{\lvert\Gamma.A\rvert} \downarrow_{F\ d} f \cdot d : B$	by (conv)
$\Gamma \vdash \lambda(\mathsf{App}\ (t\ p)\ q) =$	
$\quad \lambda(R_{\lvert\Gamma.A\rvert} \downarrow_{F\ d} f \cdot d) : \mathsf{Fun}\ A'\ B$	by congruence
$\Gamma \vdash t = \lambda(\mathsf{App}\ (t\ p)\ q) : \mathsf{Fun}\ A'\ B$	by (eta)
$\Gamma \vdash t = R_{\lvert\Gamma\rvert} \downarrow_{\mathsf{Fun}\ X\ F} f : \mathsf{Fun}\ A'\ B$	by trans.

3.

a) $\mathsf{Sing}\ d\ X$:

$\Gamma \vdash A = \{a\}_B$	by hypothesis
$\Gamma \vdash B \sim X \in T$	by hypothesis
$\Gamma \vdash t : B \sim d \in [X]$	by hypothesis
$\Gamma \vdash B \sim X \in T$	by monotonicity 43
$\Delta \vdash t\ p^i : B\ p^i \sim d \in [X]$	by monotonicity 43
$\Delta \vdash A\ p^i = \{a\ p^i\}_{B\ p^i}$	by congruence

b) $\mathsf{Fun}\ X\ F$:

$\Delta \vdash s : A'\ p^i \sim d' \in [X]$	hypothesis	(*)
$\Delta \vdash s = R_{\lvert\Delta\rvert} \downarrow_X d' : A'\ p^i$	by ind. hyp. on (*)	
$\Delta \vdash \mathsf{App}\ (t\ p^i)\ s =$		
$\quad \mathsf{App}\ ((R_{\lvert\Gamma\rvert}\ d)\ p^i)\ (R_{\lvert\Delta\rvert}\ (\downarrow_X d')) : B\ (p^i, s)$	by congruence	
$R_{\lvert\Delta\rvert}\ \mathsf{App} d \downarrow_X d' =$		
$\quad \mathsf{App}\ ((R_{\lvert\Gamma\rvert}\ d)\ p^i)\ (R_{\lvert\Delta\rvert}\ (\downarrow_X d'))$	by definition	
$\Delta \vdash \mathsf{App}\ (t\ p^i)\ s : B\ (p^i, s) \sim$		
$\quad \uparrow_{F\ d'} \mathsf{App}\ d \downarrow_X d' \in [F\ d']$	by ind. hyp.	

□

In order to finish the proof of soundness we have to prove that each well-typed term (and each well-formed type) is logically related to its denotation; with that aim we extend the definition of logical relations to substitutions and prove the fundamental theorem of logical relations.

Definition 48 (Logical relation for substitutions).

1. $\Gamma \vdash \sigma : \diamond \sim d \in \mathbf{1}$ always holds.

2. $\Gamma \vdash (\sigma, t) : \Delta.A \sim (d, d') \in \coprod X\,(d \mapsto [F\ d])$ iff $\Gamma \vdash \sigma : \Delta \sim d \in X$, $\Gamma \vdash A\,\sigma \sim F\ d \in T$, and $\Gamma \vdash t : A\,\sigma \sim d' \in [F\ d]$.

By the way this relation is defined, the counterparts of 42, 43, and 44 are easily proved by induction on the co-domain of the substitutions.

Remark 19. If $\Gamma \vdash \gamma = \delta : \Delta$, and $\Gamma \vdash \gamma : \Delta \sim d \in X$, then $\Gamma \vdash \delta : \Delta \sim d \in X$.

Remark 20. If $\Gamma \vdash \delta : \Delta \sim d \in X$, then for any $\Theta \leqslant^{i} \Gamma$, $\Theta \vdash \delta\,p^{i} : \Delta \sim d \in X$.

Remark 21. If $\Gamma \vdash \gamma : \Delta \sim d \in X$, and $d = d' \in X$, then $\Gamma \vdash \gamma : \Delta \sim d' \in X$.

Theorem 46 (Fundamental theorem of logical relations). Let $\Delta \vdash \delta : \Gamma \sim d \in [\![\Gamma]\!]$.

1. If $\Gamma \vdash A$, then $\Delta \vdash A\,\delta \sim [\![A]\!]d \in T$;

2. if $\Gamma \vdash t : A$, then $\Delta \vdash t\,\delta : A\,\delta \sim [\![t]\!]d \in [[\![A]\!]d]$; and

3. if $\Gamma \vdash \gamma : \Theta$ then $\Delta \vdash \gamma\,\delta : \Theta \sim [\![\gamma]\!]d \in [\![\Theta]\!]$.

Proof. We note that for terms we show only the cases when the last rule used was the introductory rule, or the rule for introducing elements in singletons; for the case of the conversion rule, we can conclude by i.h., and lemma 42.

1. Types. We show only the case for (FUN-F).

$\Delta \vdash s : A'\,p^{i} \sim e \in [X]$	hypothesis	(*)
$\Theta \vdash \delta\,p^{i} : \Gamma \sim d \in [\![\Gamma]\!]$	By Lem. 20	(†)
$\Theta \vdash (\delta\,p^{i}, s) : \Gamma.A \sim (d, e) \in [\![\Gamma.A]\!]$	From (*) and (†)	
$\Theta \vdash B\,(\delta\,p^{i}, s) \sim [\![B]\!](d, e) \in T$	by ind. hyp. on $\Gamma.A \vdash B$ and using42 and 44	

2. Terms. We show the case for application (FUN-EL) and for (N_n-E). The case for abstraction (FUN-I) is analogous to (FUN-F).

   a) (FUN-EL)

$\Gamma \vdash \mathsf{App}\ t\ r : B\,(\mathsf{id}, r)$	hypothesis
$\Delta \vdash r\,\delta : A\,\delta \sim [\![r]\!]d \in [[\![A]\!]d]$	by ind. hyp. (*)
$\Delta \vdash t\,\delta : \mathsf{Fun}\,A\,B\,\delta \sim [\![t]\!]d \in [[\![\mathsf{Fun}\,A\,B]\!]d]$	by ind. hyp. (†)
$\Delta \vdash \mathsf{App}\,(t\,\delta)\,(r\,\delta) : B\,(\mathsf{id}, r\,\delta) \sim$ $[\![t]\!]d \cdot [\![r]\!]d \in [[\![B]\!](d, [\![r]\!]d)]$	by def. of log. rel. for (†) with (*)
$\Delta \vdash (\mathsf{App}\ t\ r)\,\delta : B\,(\mathsf{id}, r\,\delta) \sim$ $[\![\mathsf{App}\ t\ r]\!]d \in [[\![B]\!](d, [\![r]\!]d)]$	by Lem. 42 and Lem. 44

b) (N_n-E)

$\Gamma \vdash \mathsf{case}^B\ t_0 \cdots t_{n-1}\ t : B\,(\mathsf{id}, t)$	hypothesis
$\Delta \vdash t\,\delta \colon \mathsf{N}_n \sim [\![t]\!]d \in [\mathsf{N}_n]$	by inversion and i.h.
$\Delta \vdash t\,\delta = \mathsf{R}_{\lvert\Delta\rvert}\,[\![t]\!]d \colon \mathsf{N}_n$	by Lem. 45
$\Delta \vdash t_i\,\delta \colon B\,(\delta, c_i) \sim$	
$\quad [\![t_i]\!]d \in [[\![B]\!](d, [\![t]\!]d)]$	by inversion and i.h.
If $\mathsf{R}_{\lvert\Delta\rvert}\,[\![t]\!]d \equiv c_i$:	
$\quad \Delta \vdash (\mathsf{case}\ B\ t_0 \cdots t_{n-1}\ c_i)\,\delta = t_i\,\delta \colon B\,(\mathsf{id}, t)$	by subst.
$\quad \Delta \vdash (\mathsf{case}\ B\ t_0 \cdots t_{n-1}\ c_i)\,\delta \colon B\,(\mathsf{id}, t) \sim$	
$\qquad [\![\mathsf{case}^B\ t_0 \cdots t_{n-1}\ t\ ]\!]d \in [[\![B\,(\mathsf{id}, t)]\!]d]$	by Lem. 42 and 44
If $\mathsf{R}_{\lvert\Delta\rvert}\,[\![t]\!]d \in \mathrm{Ne}$:	
$\quad \Delta.\mathsf{N}_n \vdash B\,(\delta\,p, q) = \mathsf{R}_{\lvert\Delta\rvert+1} \Downarrow [\![B]\!](d, \mathsf{Var}\,\lvert\Delta\rvert)$	
$\quad \Delta \vdash t_i\,\delta = \mathsf{R}_{\lvert\Delta\rvert}\,[\![t_i]\!]d \colon B\,(\delta, c_i)$	by Lem. 45
$\quad t_i' := \mathsf{R}_{\lvert\Delta\rvert}\,[\![t_i]\!]d$	abbreviation
$\quad t' := \mathsf{R}_{\lvert\Delta\rvert} \downarrow_{[\![B]\!](d, c_i)} [\![t]\!]d$	abbreviation
$\quad B' := \mathsf{R}_{\lvert\Delta\rvert+1}\,[\![B]\!](d, \mathsf{Var}\,\lvert\Delta\rvert)$	abbreviation
$\quad e := \mathsf{case}\ B'\ t_0' \cdots t_{n-1}'\ t'$	abbreviation
$\quad \Delta \vdash (\mathsf{case}\ B\ t_0 \cdots t_{n-1}\ t)\,\delta =$	
$\qquad e : B\,(\delta, t)$	congruence
$\quad \Delta \vdash (\mathsf{case}\ B\ t_0 \cdots t_{n-1}\ t)\,\delta \colon B\,(\delta, t) \sim$	
$\qquad \uparrow_{[\![B]\!](d, [\![t]\!]d)} e \in [\![B]\!](d, [\![t]\!]d)$	by Lem. 45 and 43
$\qquad \Delta \vdash (\mathsf{case}\ B\ t_0 \cdots t_{n-1}\ c_i)\,\delta \colon B\,(\mathsf{id}, t) \sim$	
$\qquad [\![\mathsf{case}^B\ t_0 \cdots t_{n-1}\ t\ ]\!]d \in [[\![B\,(\mathsf{id}, t)]\!]d]$	by Lem. 42 and 44

3. Substitutions. Only the proof for (ext-subs) is shown.

$\Gamma \vdash \Theta.A \colon (\gamma, t)$	hypothesis	
$\Delta \vdash \gamma\,\delta \colon \Theta \sim [\![\gamma]\!]d \in [\![\Theta]\!]$	by ind. hyp.	(*)
$\Delta \vdash t\,\delta \colon (A\,\gamma)\,\delta \sim [\![t]\!]d \in [[\![A\,\gamma]\!]d]$		(†)
$([\![\gamma]\!]d, [\![t]\!]d) \in \coprod [\![\Theta]\!]\,(e \mapsto [[\![A\,\gamma]\!]e])$	from (*) and (†)	
$\Delta \vdash (\gamma, t)\,\delta \colon \Theta.A \sim [\![(\gamma, t)]\!]d \in [\![\Theta.A]\!]$	by Lem. 19 and Lem. 21	

□

We define for each context Γ an element ρ_Γ of D. This environment will be used to define the normalisation function.

Definition 49 (Canonical environment). We define ρ_Γ by induction on Γ as follows:

$$\rho_\diamond = \top$$
$$\rho_{\Gamma.A} = (d', \uparrow_{[\![A]\!]d'} \mathsf{Var}\,n) \qquad \text{where } n = |\Gamma|, \text{ and } d' = \rho_\Gamma.$$

By an immediate induction on contexts we can check the following.

Lemma 47. If $\Gamma \vdash$ then $\Gamma \vdash \mathsf{id}_\Gamma : \Gamma \sim \rho_\Gamma \in [\![\Gamma]\!]$.

Proof. By induction on $\Gamma \vdash$; we show only the inductive case. Let $\Gamma.A \vdash$.

$d := \rho_\Gamma$	definition	
$\Gamma \vdash \mathsf{id} : \Gamma \sim d \in [\![\Gamma]\!]$	by inversion and i.h.	(*)
$\Gamma.A \vdash \mathsf{id}\ p : \Gamma \sim d \in [\![\Gamma]\!]$	from (*) by Rem. 20	(†)
$\Gamma.A \vdash p : \Gamma \sim d \in [\![\Gamma]\!]$	from (†) by Rem. 19	(**)
$\Gamma.A \vdash A\ p \sim [\![A]\!]d \in T$	by inversion and Thm. 46	
$\Gamma.A \vdash q : A\ p \sim\uparrow [\![A]\!]\mathsf{dVar}\, n \in [[\![A]\!]d]$	by Thm. 46	
$\Gamma.A \vdash (p, q) : \Gamma.A \sim$		
$\quad (d, \uparrow [\![A]\!]\mathsf{dVar}\, n) \in \coprod [\![\Gamma]\!]\, (e \mapsto [[\![A]\!]e])$	by Def. 48	(‡)
$\Gamma.A \vdash \mathsf{id} : \Gamma.A \sim \rho_{\Gamma.A} \in [\![\Gamma.A]\!]$	from (‡) by Rem. 19	

□

Main results

Now we can define concretely the normalisation function as the composition of reification with normalisation after evaluation under the canonical environment. The following corollaries just instantiate previous lemmata and theorems concluding correctness of NbE.

Definition 50 (Normalisation algorithm). Let $\Gamma \vdash A$, and $\Gamma \vdash t : A$.

$$\mathbf{nbe}_\Gamma(A) = \mathsf{R}_{|\Gamma|} \Downarrow [\![A]\!]\rho_\Gamma$$
$$\mathbf{nbe}^A_\Gamma(t) = \mathsf{R}_{|\Gamma|} \downarrow_{[\![A]\!]\rho_\Gamma} [\![t]\!]\rho_\Gamma$$

Notice that if we instantiate Thm. 46 with ρ_Γ, then a well-typed term t under Γ will be logically related to its denotation. Finally, using the key lemma 45 we conclude correctness for NbE.

Corollary 48. Let $\Gamma \vdash A$, and $\Gamma \vdash t : A$, then by fundamental theorem of logical relations (and Lem. 42),

1. $\Gamma \vdash A \sim [\![A]\!]\rho_\Gamma \in T$; and
2. $\Gamma \vdash t : A \sim [\![t]\!]\rho_\Gamma \in [[\![A]\!]\rho_\Gamma]$,

Corollary 49 (Soundness of NbE). By way of Lem. 45, it follows immediately

1. $\Gamma \vdash A = \mathbf{nbe}(A)$, and
2. $\Gamma \vdash t = \mathbf{nbe}(t) : A$.

We have now a decision procedure for judgmental equality; for deciding $\Gamma \vdash t = t' : A$, put both terms in normal formal and check if they are syntactically equal.

Corollary 50. If $\Gamma \vdash A$, and $\Gamma \vdash A'$, then we can decide $\Gamma \vdash A = A'$. Also if $\Gamma \vdash t : A$, and $\Gamma \vdash t' : A$, we can decide $\Gamma \vdash t = t' : A$.

As a byproduct we can conclude that type constructors are injective; this result is exploited in the next section where we introduce the type-checking algorithm.

Remark 22. By expanding definitions, we easily check

1. $\mathbf{nbe}_\Gamma(\mathsf{Fun}\,A\,B) = \mathsf{Fun}\,(\mathbf{nbe}_\Gamma(A))\,(\mathbf{nbe}_{\Gamma.A}(B))$, and
2. $\mathbf{nbe}_\Gamma(\{a\}_A) = \{\mathbf{nbe}^A_\Gamma(a)\}_{\mathbf{nbe}_\Gamma(A)}$.

Corollary 51 (Injectivity of $\mathsf{Fun}\_\_$ and of $\{\_\}\_$). If $\Gamma \vdash \mathsf{Fun}\,A\,B = \mathsf{Fun}\,A'\,B'$, then $\Gamma \vdash A = A'$, and $\Gamma.A \vdash B = B'$. Also $\Gamma \vdash \{t\}_A = \{t'\}_{A'}$, then $\Gamma \vdash A = A'$, and $\Gamma \vdash t = t' : A$.

Main results for λ^{Irr} The normalisation function is an homomorphism also for constructors of λ^{Irr}. Moreover, the type-constructor for dependent sums is also injective.

Remark 23.

1. $\mathbf{nbe}_\Gamma(\Sigma\,A\,B) = \Sigma\,\mathbf{nbe}_\Gamma(A)\,\mathbf{nbe}_{\Gamma.A}(B)$;
2. $\mathbf{nbe}^{\Sigma\,A\,B}_\Gamma((t,b)) = (\mathbf{nbe}^A_\Gamma(t), \mathbf{nbe}^{B\,(\mathsf{id},t)}_\Gamma(b))$;
3. $\mathbf{nbe}(\mathsf{suc}(t)) = \mathsf{suc}(\mathbf{nbe}(t))$.
4. $\mathbf{nbe}([A]) = [\mathbf{nbe}(A)]$.

Corollary 52. If $\Gamma \vdash \Sigma\,A\,B = \Sigma\,A'\,B'$, then $\Gamma \vdash A = A'$, and $\Gamma.A \vdash B = B'$.

4.4 Type-checking algorithm

In this section, we define a couple of judgements that represent a bidirectional type checking algorithm for terms in normal form; its implementation in Haskell can be found in the appendix. The algorithm is similar to previous ones [4, 37], in that it proceeds by analysing the possible types for each normal form, and succeeds only if the type's shape matches the one required by the introduction rule of the term. The only difference is introduced by the presence of singleton types; now we should take into account that a normal form can also have a singleton as its type.

This situation can be dealt in two possible ways; either one checks that the deepest tag of the normalised type (see Def. 52) has the form of the type of the introductory rule; or one adds a rule for checking any term against singleton types. The first approach requires to have more rules (this is due to the combination of singletons and a universe). We take the second approach, which requires to compute the eta-long normal form of the type before type-checking. We also note that the proof of completeness is more involved, because now the algorithm is not only driven by the term being checked, but also by the type.

Our algorithm depends on having a *good* normalisation function; note that this function does not need to be based on normalisation by evaluation. Also note that the second point asks for having correctness and completeness of the normalisation function.

Definition 51 (Good normalisation function).

1. $\mathbf{nbe}(\{a\}_A) = \{\mathbf{nbe}(a)\}_{\mathbf{nbe}(A)}$, and $\mathbf{nbe}(\mathsf{Fun}\, A\, B) = \mathsf{Fun}\, \mathbf{nbe}(A)\mathbf{nbe}(B)$;
2. $\mathbf{nbe}_\Gamma(A) = \mathbf{nbe}_\Gamma(B)$ if and only if $\Gamma \vdash A = B$, and $\mathbf{nbe}^A_\Gamma(t) = \mathbf{nbe}^A_\Gamma(t')$, if and only if $\Gamma \vdash t = t' : A$.

From these properties we can prove the injectivity of Fun which is crucial for completeness of type checking λ-abstractions.

Type-checking λ^{Sing}

In this section, let $V, V', W, v, v', w \in \mathrm{Nf}$, and $k \in \mathrm{Ne}$. For obtaining the deepest tag of a singleton type, we define an operation on types, which is essentially the same as the one defined by Aspinall [14].

Definition 52 (Singleton's tag).

$$\overline{V} = \begin{cases} \overline{W} & \text{if } V \equiv \{w\}_W \\ V & \text{otherwise.} \end{cases}$$

The predicates for type-checking are defined mutually inductively, together with the function for inferring types.

Definition 53 (Type-checking and type-inference). We define three mutually inductive algorithmic judgements

$\Gamma \downarrow V$ in context Γ, normal type V checks

$\Gamma \vdash v \Leftarrow V$ in context Γ, normal term v checks against type V

$\Gamma \vdash k \Rightarrow V$ in context Γ, the type of neutral term k is inferred as V.

All three judgements presuppose and maintain the invariant the input Γ is a well-formed context. The procedures $\Gamma \downarrow V$ and $\Gamma \vdash v \Leftarrow V$ expect their inputs V and v in β-normal form. Inference $\Gamma \vdash k \Rightarrow V$ expects a neutral term k and returns its principal type V in long normal form.
Well-formedness checking of types $\Gamma \downarrow V$.

$$\frac{}{\Gamma \downarrow U} \qquad \frac{\Gamma \downarrow V \qquad \Gamma.V \downarrow W}{\Gamma \downarrow \mathsf{Fun}\, V\, W} \qquad \frac{\Gamma \downarrow V \qquad \Gamma \vdash v \Leftarrow \mathbf{nbe}(V)}{\Gamma \downarrow \{v\}_V} \qquad \frac{\Gamma \vdash k \Leftarrow U}{\Gamma \downarrow k}$$

Type checking terms $\Gamma \vdash v \Leftarrow V$.

$$\frac{\Gamma \vdash V \Leftarrow U \qquad \Gamma.V \vdash W \Leftarrow U}{\Gamma \vdash \mathsf{Fun}\, V\, W \Leftarrow U} \qquad \frac{\Gamma.V \vdash v \Leftarrow W}{\Gamma \vdash \lambda v \Leftarrow \mathsf{Fun}\, V\, W}$$

$$\frac{\Gamma \vdash V \Leftarrow U \qquad \Gamma \vdash v \Leftarrow \mathbf{nbe}(V)}{\Gamma \vdash \{v\}_V \Leftarrow U} \qquad \frac{\Gamma \vdash v \Leftarrow V' \qquad \Gamma \vdash v' = v : V'}{\Gamma \vdash v \Leftarrow \{v'\}_{V'}}$$

$$\frac{\Gamma \vdash k \Rightarrow V' \qquad \Gamma \vdash \overline{V'} = V}{\Gamma \vdash k \Leftarrow V}\ V \not\equiv \{w\}_W$$

Type inference $\Gamma \vdash \mathsf{k} \Rightarrow V$.

$$\frac{}{\Gamma.A_i \ldots A_0 \vdash \mathsf{v}_i \Rightarrow \mathbf{nbe}(A_i\ \mathsf{p}^{i+1})}$$

$$\frac{\Gamma \vdash \mathsf{k} \Rightarrow V \qquad \Gamma \vdash \overline{V} = \mathsf{Fun}\, V'\, W \qquad \Gamma \vdash \nu \Leftarrow V'}{\Gamma \vdash \mathsf{App}\, \mathsf{k}\, \nu \Rightarrow \mathbf{nbe}(W\,(\mathsf{id}, \nu))}$$

Type checking dependent function types with a bidirectional algorithm is well-understood [37, 76]; let us illustrate briefly how it works for singleton types, by considering the type checking problem $\Gamma \vdash \mathsf{q} \Leftarrow \{\mathsf{zero}\}_{\mathsf{Nat}}$, where $\Gamma = \{\mathsf{zero}\}_{\mathsf{Nat}}$. Here is a skeletal derivation of this judgement, which is at the same time an execution trace of the type checker:

$$\dfrac{\dfrac{\overline{\Gamma \vdash \mathsf{q} \Rightarrow \{\mathsf{zero}\}_{\mathsf{Nat}}} \qquad \Gamma \vdash \overline{\{\mathsf{zero}\}_{\mathsf{Nat}}} = \mathsf{Nat}}{\Gamma \vdash \mathsf{q} \Leftarrow \mathsf{Nat}} \qquad \Gamma \vdash \mathsf{q} = \mathsf{zero} \colon \mathsf{Nat}}{\Gamma \vdash \mathsf{q} \Leftarrow \{\mathsf{zero}\}_{\mathsf{Nat}}}$$

Since the type to check against is a singleton, the algorithm proceeds by checking $\{\mathsf{zero}\}_{\mathsf{Nat}} \vdash \mathsf{q} \Leftarrow \mathsf{Nat}$ and $\{\mathsf{zero}\}_{\mathsf{Nat}} \vdash \mathsf{q} = \mathsf{zero} \colon \mathsf{Nat}$. Now the type of the neutral q is inferred and its tag compared to the given type Nat; as the tag is also Nat, the check succeeds. The remaining equation $\{\mathsf{zero}\}_{\mathsf{Nat}} \vdash \mathsf{q} = \mathsf{zero} \colon \mathsf{Nat}$ is derivable by (sing-eq-el). Of course, the equations are checked by the $\mathbf{nbe}(\_)$ function; for example, by using our own function for normalisation we have $\mathbf{nbe}^{\mathsf{Nat}}_{\{\mathsf{zero}\}_{\mathsf{Nat}}}(\mathsf{q}) = \mathsf{zero} = \mathbf{nbe}^{\mathsf{Nat}}_{\{\mathsf{zero}\}_{\mathsf{Nat}}}(\mathsf{zero})$.

Theorem 53 (Correctness of type-checking).

1. If $\Gamma \downarrow V$, then $\Gamma \vdash V$.
2. If $\Gamma \vdash \nu \Leftarrow V$, then $\Gamma \vdash \nu \colon V$.
3. If $\Gamma \vdash \mathsf{k} \Rightarrow V$, then $\Gamma \vdash \mathsf{k} \colon V$.

Proof. By simultaneous induction on $\Gamma \downarrow V$, $\Gamma \vdash \nu \Leftarrow V$, and $\Gamma \vdash V \Rightarrow \mathsf{k}$.

1. Types:
   - the case for $\mathsf{Fun}\, V\, W$ is also obtained directly from the derivations we get using the i.h. on $\Gamma \downarrow V$, and $\Gamma.V \downarrow W$; and use them for deriving $\Gamma \vdash \mathsf{Fun}\, V\, W$
   - for $\{\nu\}_V$, we can apply the same reasoning as before: by i.h. on $\Gamma \downarrow V$, and $\Gamma \vdash \nu \Leftarrow \mathbf{nbe}(V)$ we know that there are, respectively, derivations with conclusions $\Gamma \vdash V$, and $\Gamma \vdash \nu \colon V$; from which we can conclude $\Gamma \vdash \{\nu\}_V$
   - here we'll consider the three cases when V is a neutral term, because the reasoning is the same. By i.h. on $\Gamma \vdash V \Leftarrow \mathsf{U}$, we have a derivation with conclusion $\Gamma \vdash V \colon \mathsf{U}$; hence we use (u-el).
2. Terms:

- let $V = \mathsf{U}$, and $v = \mathsf{Fun}\, V' W$. By i.h. $\Gamma \vdash V' \colon \mathsf{U}$, and $\Gamma.V' \vdash W \colon \mathsf{U}$, and using both derivations we can derive $\Gamma \vdash \mathsf{Fun}\, V W \colon \mathsf{U}$.
- consider $V = \mathsf{U}$, and $v = \{v'\}_{V'}$. by i.h. on $\Gamma \vdash V' \Leftarrow \mathsf{U}$, and $\Gamma \vdash v' \Leftarrow \mathbf{nbe}(V)$, we have $\Gamma \vdash V \colon \mathsf{U}$, and $\Gamma \vdash v' \colon \mathbf{nbe}(V)$, and using conversion we derive $\Gamma \vdash v' \colon V$; and these are the premises we need to show $\Gamma \vdash \{v'\}_V \colon \mathsf{U}$.
- $V = \mathsf{Fun}\, V' W$, and $v = \lambda v'$: we have $\Gamma.V' \vdash v' \Leftarrow W$. From this we can conclude by i.h. $\Gamma.V' \vdash v' \colon W$; and this is the key premise for concluding $\Gamma \vdash \lambda v' \colon \mathsf{Fun}\, V' W$.
- $V = \{w\}_W$: by hypothesis we know $\Gamma \vdash w \colon W$, and $\Gamma \vdash v \Leftarrow W$, and $\Gamma \vdash w = v \colon W$; by the i.h. on the second one we get $\Gamma \vdash v \colon W$; then we can conclude using (SING-I).
- $v = \mathsf{k} \in \mathrm{Ne}$, and $V \not\equiv \{w\}_W$: let $\Gamma \vdash \mathsf{k} \Rightarrow V'$, then we distinguish the cases when V' is a singleton, and when V' is not a singleton. In the latter case, the derivation is obtained directly from the correctness of type-inference. In the first case we use the rule (SING-EL), with the derivation obtained by i.h. and then we conclude with conversion.

3. Inference:

   - for $\mathsf{q}\, \mathsf{p}^{\mathsf{i}}$, if $\mathsf{i} = 0$, then we use (HYP), and conversion; if $\mathsf{i} > 0$, then we have a derivation with conclusion $\Gamma \vdash \mathsf{q} \colon A_{\mathsf{i}}\, \mathsf{p}$, and clearly $\Gamma \vdash \mathsf{p}^{\mathsf{i}} \colon \Gamma.A_{\mathsf{i}} \ldots . A_0$, hence by (SUBS-TERM), we have $\Gamma.A_{\mathsf{i}} \ldots . A_0 \vdash \mathsf{q}\, \mathsf{p}^{\mathsf{i}} \colon A_{\mathsf{i}}\, \mathsf{p}^{\mathsf{i}+1}$, we conclude by correctness of $\mathbf{nbe}(\_)$ and by conversion.
   - by i.h. we have derivations with conclusions $\Gamma \vdash \mathsf{k} \colon V'$, with $\overline{V'} = \mathsf{Fun}\, V W$, hence we have a derivation $\Gamma \vdash \mathsf{k} \colon \mathsf{Fun}\, V W$ (using (SING-EL) if necessary) and $\Gamma \vdash v \colon V$, hence by the rule (FUN-EL), we have $\Gamma \vdash \mathsf{App}\, \mathsf{k}\, v \colon W\, (\mathsf{id}, v)$. We conclude by conversion and correctness of $\mathbf{nbe}(\_)$.

□

In order to prove completeness we define a lexicographic order on pairs of terms and types, in this way we can make induction over the term, and the type.

Definition 54. Let $v, v' \in \mathrm{Nf}$, and $A, A' \in \mathsf{Type}(\Gamma)$, then $(v, A) \prec (v', A')$ is the lexicographic order on $\mathrm{Nf} \times \mathsf{Type}(\Gamma)$. The corresponding orders are $v \prec v'$ iff v is an immediate sub-term of v'; and $A \prec^{\Gamma} A'$, iff $\mathbf{nbe}(A') \equiv \{w\}_{\mathbf{nbe}(A)}$.

Theorem 54 (Completeness of type-checking).

1. If $\Gamma \vdash V$, then $\Gamma \downarrow V$.
2. If $\Gamma \vdash v \colon A$, then $\Gamma \vdash v \Leftarrow \mathbf{nbe}(A)$.
3. If $\Gamma \vdash \mathsf{k} \colon A$, and $\Gamma \vdash \mathsf{k} \Rightarrow V'$, then $\Gamma \vdash \overline{\mathbf{nbe}(A)} = \overline{V'}$.

Proof. We prove simultaneously all the points. The first point is by induction on the structure of the type. In the last two points we use well-founded induction on the order $\prec$.

1. Types:
   - $\Gamma \vdash \mathsf{Fun}\, V'\, W$; by inversion we know $\Gamma \vdash V'$, and $\Gamma.V' \vdash W$; hence by i.h. we have respectively $\Gamma \downarrow V'$, and $\Gamma.V' \downarrow W$.
   - $V = \{v\}_{V'}$: by inversion we have $\Gamma \vdash V'$, and $\Gamma \vdash v\colon \mathbf{nbe}(V')$, hence by i.h. we have both $\Gamma \downarrow V'$, and $\Gamma \vdash v \Leftarrow V'$.
   - $\Gamma \vdash k$, we have to show $\Gamma \downarrow k$. By lemma 29, we know $\Gamma \vdash k\colon \mathsf{U}$; hence by i.h. we have $\Gamma \vdash k \Rightarrow A$, and $\Gamma \vdash A = \mathsf{U}$, hence $\Gamma \vdash k \Leftarrow \mathsf{U}$.
2. Terms: We omit the trivial cases, e.g. (U, A); we have re-arranged the order of the cases for the sake of clarity.
   - $v = \mathsf{Fun}\, V'\, W$:
     a) either $\Gamma \vdash A = \mathsf{U}$, $\Gamma \vdash V'\colon \mathsf{U}$, and $\Gamma.V' \vdash W\colon \mathsf{U}$; hence, by i.h. we know both $\Gamma \vdash V' \Leftarrow \mathsf{U}$, and $\Gamma.V' \vdash W \Leftarrow \mathsf{U}$; hence we can conclude $\Gamma \vdash \mathsf{Fun}\, V\, W \Leftarrow \mathsf{U}$.
     b) Or $\Gamma \vdash A = \{a\}_{A'}$, $\Gamma \vdash v\colon A'$, and $\Gamma \vdash v = a\colon A'$, hence by i.h. we know $\Gamma \vdash v \Leftarrow \mathbf{nbe}(B)$, by conversion we also have and transitivity of the equality $\Gamma \vdash \mathbf{nbe}(a) = v\colon \mathbf{nbe}(B)$, hence $\Gamma \vdash v \Leftarrow \{\mathbf{nbe}(a)\}_{\mathbf{nbe}(B)}$.
   - $v = \{v'\}_V$:
     a) $\Gamma \vdash V\colon \mathsf{U}$, and $\Gamma \vdash v'\colon V$. From those derivations we have by i.h. $\Gamma \vdash V \Leftarrow \mathsf{U}$, and $\Gamma \vdash v' \Leftarrow \mathbf{nbe}(V)$, respectively; from which we conclude $\Gamma \vdash \{v'\}_V \Leftarrow \mathsf{U}$
     b) $\Gamma \vdash A = \{a\}_{A'}$, with $\Gamma \vdash v\colon A'$, and $\Gamma \vdash v = a\colon A'$, hence by i.h. we know $\Gamma \vdash v \Leftarrow \mathbf{nbe}(B)$. We can also derive $\Gamma \vdash \mathbf{nbe}(a) = v\colon \mathbf{nbe}(B)$, hence $\Gamma \vdash v \Leftarrow \{\mathbf{nbe}(a)\}_{\mathbf{nbe}(B)}$.
   - $v = \lambda v'$
     a) $\Gamma \vdash V = \mathsf{Fun}\, A'\, B$, and $\Gamma.A' \vdash v'\colon B$; from this we can conclude $\Gamma.\mathbf{nbe}(A') \vdash v'\colon B$ by ind. hyp. we get $\Gamma.\mathbf{nbe}(A') \vdash v' \Leftarrow \mathbf{nbe}(B)$; therefore $\Gamma \vdash \lambda v' \Leftarrow \mathsf{Fun}\, \mathbf{nbe}(A')\, \mathbf{nbe}(B')$.
     b) Or $\Gamma \vdash A = \{a\}_{A'}$, $\Gamma \vdash v\colon A'$, and $\Gamma \vdash v = a\colon A'$, hence by i.h. we know $\Gamma \vdash v \Leftarrow \mathbf{nbe}(B)$, by conversion we also have and transitivity of the equality $\Gamma \vdash \mathbf{nbe}(a) = v\colon \mathbf{nbe}(B)$, hence $\Gamma \vdash v \Leftarrow \{\mathbf{nbe}(a)\}_{\mathbf{nbe}(B)}$.
   - $v \in \mathrm{Ne}$: then we do case analysis on $\mathbf{nbe}(A)$.
     a) If $\mathbf{nbe}(A) = \{w\}_W$, then by soundness of $\mathbf{nbe}(\_)$, and conversion we have $\Gamma \vdash k\colon \{w\}_W$; and by inversion of singletons we have $\Gamma \vdash k\colon W$, and also $\Gamma \vdash k = w\colon W(*)$. Clearly $(k, W) \prec (k, A)$, hence we can apply the inductive hypothesis and conclude $\Gamma \vdash k \Leftarrow W$; from that and $(*)$, we conclude $\Gamma \vdash k \Leftarrow \{w\}_W$, i.e., $\Gamma \vdash k \Leftarrow \mathbf{nbe}(A)$.
     b) If $V \not\equiv \{w\}_W$, then $\overline{V} \equiv V$. We use the last clause for concluding $\Gamma \vdash k \Leftarrow \mathbf{nbe}(A)$; but we need to show that if $\Gamma \vdash k \Rightarrow V'$, then $\Gamma \vdash \overline{V} = \overline{V'}$; we show this in the next point.
3. Inference: let $\Gamma \vdash k\colon A$, $\Gamma \vdash k \Rightarrow V'$, and $V = \mathbf{nbe}(A)$. Show $\Gamma \vdash \overline{V} = \overline{V'}$.

- let us consider first the case when $V = \{w\}_W$; by inversion we have derivations $\Gamma \vdash k : W$, and $\Gamma \vdash k = w : W$. Hence by i.h. we know that $\Gamma \vdash \overline{V'} = \overline{W}$, and $\overline{W} = \overline{\{w\}_W}$.
- Now we consider the case when V is not a singleton, and $k = q\,p^i$; this case is trivial because by inversion we know that $\Gamma \vdash V = \mathbf{nbe}((\Gamma!i)\,p^{i+1})$.
- the last case to consider is $k = \mathsf{App}\,k'\,v$ and V not a singleton. By inversion we know $\Gamma \vdash \mathsf{App}\,k\,v : B\,(\mathsf{id}, v)$, and $\Gamma \vdash k : \mathsf{Fun}\,A\,B$, hence $\Gamma \vdash k : \mathsf{Fun}\,\mathbf{nbe}(A)\,\mathbf{nbe}(B)$, and $\Gamma \vdash v : A$, hence $\Gamma \vdash v : \mathbf{nbe}(A)$. By i.h. we know that if $\Gamma \vdash k \Rightarrow V'$, then $\overline{V'} = \mathsf{Fun}\,\mathbf{nbe}(A)\,\mathbf{nbe}(B)$, and also $\Gamma \vdash v \Leftarrow \mathbf{nbe}(A)$. Hence we can conclude $\Gamma \vdash \mathsf{App}\,k\,v \Rightarrow \mathbf{nbe}(\mathbf{nbe}(B)\,(\mathsf{id}, v))$. And $\Gamma \vdash \mathbf{nbe}(\mathbf{nbe}(B)\,(\mathsf{id}, v)) = \mathbf{nbe}(B\,(\mathsf{id}, v))$ (by correctness of the $\mathbf{nbe}(\_)$ algorithm).

□

Calculus λ^{Irr} with proof irrelevance

We give additional rules for type-checking and type-inference algorithms for the constructs added in Sec. 4.1. Remember that we distinguished two calculi: the calculus $(\vdash^\star)$ has rules (N_0-TM) and (PRF-TM); while $(\vdash)$ lacks those rules.

Definition 55 (Type-checking and type-inference). Σ-types.

$$\frac{\Gamma \downarrow V \qquad \Gamma.V \downarrow W}{\Gamma \downarrow \Sigma\,V\,W} \qquad \frac{\Gamma \vdash V \Leftarrow U \qquad \Gamma.V \vdash W \Leftarrow U}{\Gamma \vdash \Sigma\,V\,W \Leftarrow U}$$

$$\frac{\Gamma \vdash v \Leftarrow V \qquad \Gamma \vdash v' \Leftarrow \mathbf{nbe}(W\,(\mathsf{id}_\Gamma, v))}{\Gamma \vdash (v, v') \Leftarrow \Sigma\,V\,W}$$

$$\frac{\Gamma \vdash k \Rightarrow \Sigma\,V\,W}{\Gamma \vdash \mathsf{fst}\,k \Rightarrow V} \qquad \frac{\Gamma \vdash k \Rightarrow \Sigma\,V\,W}{\Gamma \vdash \mathsf{snd}\,k \Rightarrow \mathbf{nbe}(W\,(\mathsf{id}_\Gamma, \mathsf{fst}\,k))}$$

Natural numbers.

$$\frac{}{\Gamma \downarrow \mathsf{Nat}} \qquad \frac{}{\Gamma \vdash \mathsf{Nat} \Leftarrow U} \qquad \frac{}{\Gamma \vdash \mathsf{zero} \Leftarrow \mathsf{Nat}} \qquad \frac{\Gamma \vdash v \Leftarrow \mathsf{Nat}}{\Gamma \vdash \mathsf{suc}(v) \Leftarrow \mathsf{Nat}}$$

$$\frac{\begin{array}{c}\Gamma.\mathsf{Nat} \downarrow V \qquad \Gamma \vdash k \Rightarrow \mathsf{Nat} \qquad \Gamma \vdash v \Leftarrow \mathbf{nbe}(V\,(\mathsf{id}_\Gamma, \mathsf{zero})) \\ \Gamma \vdash v' \Leftarrow \mathsf{Fun}\,\mathsf{Nat}\,(\mathsf{Fun}\,V\,\mathbf{nbe}(V\,(p\,p, \mathsf{suc}((q\,p)))))\end{array}}{\Gamma \vdash \mathsf{natrec}(V, v, v', k) \Rightarrow \mathbf{nbe}(V\,(\mathsf{id}, k))}$$

Finite types.

$$\frac{}{\Gamma \downarrow N_n} \qquad \frac{}{\Gamma \vdash N_n \Leftarrow U} \qquad \frac{i < n}{\Gamma \vdash c_i^n \Leftarrow N_n}$$

$$\frac{\Gamma.N_n \downarrow V \qquad \Gamma \vdash k \Rightarrow N_n \qquad \Gamma \vdash v_i \Leftarrow \mathbf{nbe}(V\,(\mathsf{id}_\Gamma, c_i^n))}{\Gamma \vdash \mathsf{case}^n\ V\ v_0 \cdots v_{n-1}\ k \Rightarrow \mathbf{nbe}(V\,(\mathsf{id}, k))}$$

Proof types.

$$\frac{\Gamma \downarrow V}{\Gamma \downarrow [V]} \qquad \frac{\Gamma \vdash V \Leftarrow \mathsf{U}}{\Gamma \vdash [V] \Leftarrow \mathsf{U}} \qquad \frac{\Gamma \vdash v \Leftarrow V}{\Gamma \vdash [v] \Leftarrow [V]}$$

$$\frac{\Gamma \vdash \mathsf{k} \Rightarrow [V] \qquad \Gamma.V \vdash v \Leftarrow \mathbf{nbe}(W\,\mathsf{p}) \qquad \Gamma.V.V\mathsf{p} \vdash v\mathsf{p} = v(\mathsf{pp},\mathsf{q}) : W\mathsf{pp}}{\Gamma \vdash v\,\mathsf{where}^{W}\,\mathsf{k} \Rightarrow W}\ \Gamma \downarrow W$$

We do not show the proof for correctness, because nothing is to be gained from it; suffice it to say that we can prove correctness with respect to $(\vdash^\star)$.

Theorem 55. The type-checking algorithm is sound with respect to the calculus $\vdash^\star$.

Proof. By simultaneous induction on the derivability of the type-checking judgements. □

It is clear that the given rules are not complete for checking $(\vdash^\star)$, because there is no rule for checking $\Gamma \vdash \star \Leftarrow A$. Note that it is not possible to have a sound and complete type-checking algorithm with respect to $(\vdash^\star)$, for it would imply the decidability of type-inhabitation. Since type checking happens always *before* normalisation, we can still use a good normalisation function with respect to the calculus $(\vdash^\star)$ for normalising types or deciding equality. Indeed, if the term to type-check does not contain $\star$, the need of checking $\Gamma \vdash \star \Leftarrow V$ will never arise; this is clearly seen by verifying that only sub-terms are type-checked in the premises.

Theorem 56. The type-checking algorithm is complete with respect to the calculus $(\vdash)$.

Proof. By simultaneous induction on the normal form of types and terms, using inversion on the typing judgement and correctness of $\mathbf{nbe}(\_)$. □

Corollary 57. The type-checking algorithm is correct (by Thm. 55 and Cor. 33) and complete with respect to the calculus $(\vdash)$.

4.5 Normalisation by evaluation

```
type Type   =  Term
data Term   =  U                          -- universe
           |  Fun Type Type               -- dependent function space
           |  Singl Term Type             -- singleton type ($\{a\}_A$)
           |  App Term Term               -- application
           |  Lam Term                    -- abstraction
           |  Q                           -- variable
           |  Sub Term Subst              -- substitution
           |  Sigma Type Type             -- dependent pair type
           |  Fst Term                    -- first projection
           |  Snd Term                    -- second projection
           |  Pair Term Term              -- dependent pair
           |  Nat                         -- naturals
           |  Zero                        -- 0
           |  Suc Term                    -- +1
           |  Natrec Type Term Term Term  -- elimination for Nat
           |  Prf Type                    -- proof irrelevance types
```

```
            |  Box Term                    -- a term in Prf
            |  Star                       -- canonical element in Prf
            |  Where Type Term Term       -- |Box| elimination
            |  Enum Int                   -- |Enum n| has n elements
            |  Const Int Int              -- |Const n i|, the |i|th element
            |  Case Int Type [Term] Term   -- elimination for |Enum n|
               deriving (Eq,Show)

data Subst  =  E                          -- empty substitution
            |  Is                         -- identity substitution
            |  Ext Subst Term             -- extension
            |  P                          -- weakening
            |  Comp Subst Subst           -- composition
               deriving (Eq,Show)

type DT     =  D                          -- semantic types
data D      =  T                          -- terminal object
            |  Ld (D -> D)                 -- function
            |  FunD DT (D -> DT)           -- dependent function type
            |  UD                         -- universe
            |  SingD D DT                 -- singleton type
            |  Vd Int                     -- free variable
            |  AppD D D                   -- neutral application
            |  SumD DT (D -> DT)           -- dependent sum
            |  PairD D D                  -- context comprehension
            |  FstD D                     -- first projection
            |  SndD D                     -- second projection
            |  NatD                       -- natural number type
            |  ZeroD                      -- 0
            |  SucD D                     -- +1
            |  NatrecD (D -> DT) D D D    -- recursion on neutrals
            |  PrfD DT                    -- proof type
            |  StarD                      -- don't care
            |  EnumD Int                  -- enumeration type
            |  ConstD Int Int             -- constants in |EnumD|
            |  CaseD Int (D -> DT) [D] D  -- elimination on neutrals

type Ctx    =  [Type]

pi1, pi2 ::  D -> D
pi1  (PairD d d')  =  d
pi1  StarD         = StarD
pi1  k             =  FstD k
pi2  (PairD d d')  =  d'
pi2  StarD         = StarD
pi2  k             =  SndD k

ap :: D -> D -> D
ap (Ld f) d = f d
ap StarD  _ = StarD

neutralD :: D -> Bool
neutralD (Vd _)              = True
neutralD (AppD _ _)          = True
neutralD (FstD _)            = True
neutralD (SndD _)            = True
neutralD (NatrecD _ _ _ _)   = True
neutralD (CaseD _ _ _ _)     = True
neutralD StarD               = True
neutralD _                   = False
```

```
natrec :: (D -> DT) -> D -> D -> D -> D
natrec b z s StarD            = StarD
natrec b z s ZeroD            = z
natrec b z s (SucD e)         = (s `ap` e) `ap` (natrec b z s e)
natrec b z s d | neutralD d   = up  (b d)
                                    (NatrecD  (\e -> downT (b e))
                                              (down (b ZeroD) z)
                                              downSuc
                                              d)
           where downSuc = down  (FunD  NatD
                                        (\n -> FunD (b n)
                                        (\e -> b (SucD n))))
                                 s

downs :: Int -> (D -> DT) -> [D] -> Int -> [D]
downs  _  _  []      _  = []
downs  n  f  (d:ds)  i  = down (f (ConstD n i)) d : downs n f ds (i+1)

constD  :: Int -> Int -> D -> Bool
constD  n  i  (ConstD m j)  =  m == n && i == j
constD  _  _  _             =  False

caseD :: Int -> (D -> DT) -> [D] -> D -> D
caseD n b ds StarD                                          =  StarD
caseD n b ds (ConstD m i) | n == m && i < n                 =  ds!!i
caseD n b ds d |  neutralD d &&
                  and [ constD n i (ds!!i) | i <- [0..n-1]] =  up (b d) d
caseD n b ds d |  neutralD d                                =  up (b d)
                                  (CaseD n  (\e -> downT (b e))
                                            (downs n b ds 0)
                                            d)
up :: DT -> D -> D
up (SingD a x)    k  = a
up (FunD a f)     k  = Ld (\d -> up (f d) (AppD k (down a d)))
up (SumD a f)     k  = PairD  (up a (FstD k))
                              (up (f (up a (FstD k))) (SndD k))
up (PrfD a)       k  = StarD
up (EnumD 0)      k  = StarD
up (EnumD 1)      k  = ConstD 1 0
up d              k  = k

down :: DT -> D -> D
down UD           d  = downT d
down (SingD a x)  d  = down x a
down (FunD a f)   d  = Ld (\e -> down (f (up a e)) (d `ap` (up a e)))
down (SumD a b)   d  = PairD (down a (pi1 a)) (down (b (pi1 d)) (pi2 d))
down (PrfD a)     d  = StarD
down (EnumD 1)    d  = ConstD 1 0
down d            e  = e

downT :: DT -> DT
downT (SingD a x)    = SingD  (down x a)  (downT x)
downT (FunD a f)     = FunD   (downT a)   (\d -> downT (f (up a d)))
downT (SumD a b)     = SumD   (downT a)   (\d -> downT (b (up a d)))
downT (PrfD a)       = PrfD   (downT a)
downT d              = d

readback :: Int -> D -> Term
readback i UD                = U
readback i (FunD a f)        = Fun       (readback i a)
                                         (readback (i+1) (f (Vd i)))
readback i (SingD a x)       = Singl     (readback i a) (readback i x)
```

```
readback i (Ld f)             = Lam       (readback (i+1) (f (Vd i)))
readback i (Vd n)             = mkvar     (i-n-1)
readback i (AppD k d)         = App       (readback i k) (readback i d)
readback i (FstD d)           = Fst       (readback i d)
readback i (SndD d)           = Snd       (readback i d)
readback i (PairD d e)        = Pair      (readback i d) (readback i e)
readback i (SumD a b)         = Sigma     (readback i a)
                                          (readback (i+1) (b (Vd i)))
readback i NatD               = Nat
readback i ZeroD              = Zero
readback i (SucD e)           = Suc       (readback i e)
readback i (NatrecD b z s e)  = Natrec    (Fun Nat  (readback (i+1) (b (Vd i))))
                                          (readback i z)
                                          (readback i s)
                                          (readback i e)
readback i (PrfD d)            = Prf      (readback i d)
readback i StarD               = Star
readback i (EnumD n)           = Enum n
readback i (ConstD n j)        = Const n  j
readback i (CaseD n b ds d)    = Case n   (readback (i+1) (b (Vd i)))
                                          (map (readback i) ds)
                                          (readback i d)

-- Evaluation

type Env = D

eval :: Term -> Env -> D
eval U                 d = UD
eval (Fun t f)         d = FunD (eval t d) (\d' -> eval f (PairD d d'))
eval (Singl t a)       d = SingD (eval t d) (eval a d)
eval (Lam t)           d = Ld (\d' -> eval t (PairD d d'))
eval (App t r)         d = (eval t d) `ap` (eval r d)
eval Q                 d = pi2 d
eval (Sub t s)         d = eval t (evalS s d)

eval (Sigma t r)       d = SumD (eval t d) (\e -> eval r (PairD d e))
eval (Fst t)           d = pi1 (eval t d)
eval (Snd t)           d = pi2 (eval t d)
eval (Pair t r)        d = PairD (eval t d) (eval r d)

eval Nat               d = NatD
eval Zero              d = ZeroD
eval (Suc t)           d = SucD (eval t d)
eval (Natrec b z s t)  d = natrec  (\e -> eval b (PairD d e))
                                   (eval z d)
                                   (eval s d)
                                   (eval t d)


eval (Prf t)           d = PrfD (eval t d)
eval (Box t)           d = StarD
eval Star              d = StarD
eval (Where t b p)     d = eval b (PairD d StarD)
eval (Enum n)          d = EnumD n
eval (Const n i)       d = ConstD n i
eval (Case n b ts t)   d = caseD n  (\e -> eval b (PairD d e))
                                    (map ((flip eval) d) ts)
                                    (eval t d)

evalS :: Subst -> Env -> Env
evalS E            d = T
```

```
evalS Is          d = d
evalS (Ext s t)   d = PairD (evalS s d) (eval t d)
evalS P           d = pi1 d
evalS (Comp s s') d = (evalS s . evalS s') d

nbe :: Type -> Term -> Term
nbe ty t = readback 0 (down (eval ty T) (eval t T))

nbeTy :: Type -> Type
nbeTy ty = readback 0 (downT (eval ty T))

nbeOpen :: Ctx -> Type -> Term -> Term
nbeOpen ctx ty t  = readback n (down (eval ty env) (eval t env))
    where  n      = length ctx
           env    = mkenv n ctx



nbeOpenTy :: Ctx -> Type -> Type
nbeOpenTy ctx ty  = readback n (downT (eval ty env))
    where  n      = length ctx
           env    = mkenv n ctx


mkenv :: Int -> Ctx -> Env
mkenv 0 []        = T
mkenv n (t:ts)    = PairD d' (up td (Vd (n-1)))
    where  d'     = mkenv (n-1) ts
           td     = eval t d'

mkvar :: Int -> Term
mkvar n  | n == 0     = Q
         | otherwise  = Sub Q (subs (n-1))

subs n  | n == 0      = P
subs n  | otherwise   = Comp P (subs (n-1))
```

4.6 Type-checking algorithm

Type checking algorithm for normal forms, and type inference algorithm for neutral terms.

Checking well-formedness of types

```
chkType :: Ctx -> Type -> Bool
chkType ts  U                     = True
chkType ts  (Fun t r)             = chkType ts t && chkType (t:ts) r
chkType ts  (Singl a t)           = chkType ts t && chkTerm ts t a
chkType ts  (Sigma t r)           = chkType ts t && chkType (t:ts) r
chkType ts  Nat                   = True
chkType ts  (Prf t)               = chkType ts t
chkType ts  (Enum n)              = True
chkType ts  Q                     = chkNeTerm ts U Q
chkType ts  w@(Sub Q s)           = chkNeTerm ts U w
chkType ts  w@(App k v)           = chkNeTerm ts U w
chkType ts  w@(Fst k)             = chkNeTerm ts U w
chkType ts  w@(Snd k)             = chkNeTerm ts U w
chkType ts  w@(Natrec t' v v' k)  = chkNeTerm ts U w
```

```
chkType _   _                    = False
```

Checking the types of terms

```
sgSub :: Term -> Term -> Term
sgSub t t' = Sub t (Ext Is t')


chkTerm :: Ctx -> Type -> Term -> Bool
chkTerm ts  U            (Fun t t')     =  chkTerm ts U t &&
                                           chkTerm (t:ts) U t'
chkTerm ts  U            (Singl e t)    =  chkTerm ts U t &&
                                           chkTerm ts t e
chkTerm ts  U            (Sigma t t')   =  chkTerm ts U t &&
                                           chkTerm (t:ts) U t'
chkTerm ts  U             Nat           =  True
chkTerm ts  (Fun t t')   (Lam e)        =  chkTerm (t:ts) t' e
chkTerm ts  (Singl e t)  e'             =  chkTerm ts (nbeOpenTy ts t) e' &&
                                           (nbeOpen ts e t) == (nbeOpen ts e' t)
chkTerm ts  (Sigma t r)  (Pair e e')    =  chkTerm ts t e &&
                                           chkTerm ts (nbeOpenTy ts (sgSub r e)) e'
chkTerm ts  Nat          Zero           =  True
chkTerm ts  Nat          (Suc t)        =  chkTerm ts Nat t
chkTerm ts  (Prf t)      (Box e)        =  chkTerm ts t e
chkTerm ts  (Enum n)     (Const m i)    =  m == n && i < n
chkTerm ts  t            e | neutral e  =  chkNeTerm ts t e
chkTerm _   _            _              =  False

neutral :: Term -> Bool
neutral Q                   = True
neutral (Sub Q s)           = True
neutral (App k v)           = True
neutral (Fst k)             = True
neutral (Snd k)             = True
neutral (Natrec t' v v' k)  = True
neutral (Case n b ts t)     = True
neutral (Where t b p)       = True
neutral _                   = False

erase :: Type -> Type
erase (Singl e t)  = erase t
erase t            = t

maybeEr :: Maybe Type -> Maybe Type
maybeEr = maybe Nothing (Just . erase)

chkNeTerm :: Ctx -> Type -> Term -> Bool
chkNeTerm ts t e =  case  maybeEr (infType ts e) of
                    Just t'  -> t == t'
                    Nothing  -> False
```

Inferring the types of neutral terms

```
nbeType :: Ctx -> Type -> Maybe Type
nbeType ctx t = Just (nbeOpenTy ctx t)
```

```
infType :: Ctx -> Term -> Maybe Type
infType (t:ts) Q            =  nbeType (t:ts) (Sub t P)
infType ts     (Sub Q s)    =  case infType (infCtx ts s) Q of
                               Just t -> nbeType ts (Sub t s)
                               _ -> Nothing

infType ts (App e e')       =  case maybeEr (infType ts e) of
                                 Just (Fun t t') ->
                                     if chkTerm ts t e'
                                     then nbeType ts (sgSub t' e')
                                     else Nothing
                                 _ -> Nothing

infType ts (Fst e)          =  case maybeEr (infType ts e) of
                                 Just (Sigma t t') ->  Just t
                                 _ -> Nothing
infType ts (Snd e)          =  case maybeEr (infType ts e) of
                                 Just (Sigma t t') -> nbeType ts (sgSub t' (Fst e))
                                 _ -> Nothing
infType ts (Natrec t v w k) =  case maybeEr (infType ts k) of
                               Just Nat -> if
                                   chkType (Nat:ts) t &&
                                   chkTerm ts (nbeOpenTy ts (sgSub t Zero)) v &&
                                   chkTerm (Nat:ts)
                                            (Fun (sgSub t Q)
                                                 (sgSub t (Suc (Sub Q P)))) w
                                   then nbeType ts (sgSub t k)
                                   else Nothing
                               _ -> Nothing
infType ts (Where t b k)    =  case maybeEr (infType ts k) of
                               Just (Prf t') -> if chkType ts t &&
                                                   chkTerm (t':ts) t b &&
                                                   nbeOpen ts' w
                                                      (Sub b (Ext (subs 1) Q)) ==
                                                   nbeOpen ts' w (Sub b P)
                                                   then Just t
                                                   else Nothing
                                   where ts' = Sub t' P:t':ts
                                         w   = Sub t (subs 1)
                               _ -> Nothing
infType ts (Case n b cs k)  =  case maybeEr (infType ts k) of
                               Just (Enum m) -> if m == n &&
                                                    chkType (Enum n:ts) b &&
                                                    chkList ts n b 0 cs
                                                then nbeType ts (sgSub b k)
                                                else Nothing
                               _ -> Nothing

infType _ _                 =  Nothing

chkList :: Ctx -> Int -> Type -> Int -> [Term] -> Bool
chkList ts  _  _  _  []     =  True
chkList ts  n  b  i  (e:es) =  chkTerm ts (nbeOpenTy ts (sgSub b (Const n i))) e &&
                               chkList ts n b (i+1) es

infCtx :: Ctx -> Subst -> Ctx
infCtx  (t:ts)  P          = ts
infCtx  (t:ts)  (Comp P s) = infCtx ts s
```

Pure Type Systems

5

In this chapter we show the equivalence between two presentations of PTSs that differ in their notion of equality in typing. One presentation is more akin to MLTT where equality is defined by rules of the typing system; the more common approach is to take equality as the convertibility relation defined among raw terms. The former is known as judgemental equality and is convenient for theoretical considerations [87, 110]; the latter is known as external equality and is often used in implementations. Given an implementation of a type-system based on the untyped notion of equality, it is desirable to have theoretical results ensuring the correctnes of the system.. If both systems are equivalent — every deduction in one of them can be mimicked on the other — then one can transfer theoretical results in the version with typed equality to the system with untyped conversion.

In this chapter, we establish such an equivalence for a certain class of PTSs, by exploiting some of the results of NbE we have obtained in the previous chapters. The main contribution of this chapter is in the proof method; this result has been already settled for every PTS by Siles[104] for every PTS, cf. Sec. 5.2. We have formalised in Agda several syntactical properties of both presentations of PTSs. The Agda files can be found in `http://cs.famaf.unc.edu.ar/~mpagano/thesis/pts`; the file names are relative to that URL.

5.1 Formal systems

As we already explained PTSs capture a whole class of type-systems by parameterising the syntax and the typing rules over *signatures*.

Definition 56 (PTS Signature). A signature $S = (S, A, R)$ is given by a set S, a binary relation A over S, and a ternary relation R over S.

Elements in S are called *sorts* and roughly correspond to universes. The relation $A \subseteq S^2$ is called the set of *axioms* and can be thought as defining a hierarchy of universes. The ternary relation $R \subseteq S^3$, which is called the set of *rules*, prescribes which function spaces can be formed in the system.

As far as we know, our presentation of PTSs is new, although the variations that we add were already considered separately. The three aspects in which we depart from usual PTSs are: explicit substitutions, de Bruijn indices, and abstractions without domain annotations. PTSs with explicit substitutions were considered by Bloo [27]; Muñoz [92] studied one particular PTSs with explicit substitutions and de Bruijn indices. Barthe and Sørensen [20] introduced domain-free PTSs and studied their meta-theory.

The set of terms and substitutions is shared between both presentations of PTSs.

Definition 57 (Raw terms). Let $S = (S, A, R)$ be a signature. The syntax of pre-terms and pre-substitutions are defined by:

$$\begin{aligned} \textit{Terms} \ni t, t' &::= \mathsf{q} \mid t\,\sigma \mid s \mid \mathsf{Fun}\, t\, t' \mid \mathsf{App}\, t\, t' \mid \lambda t \\ \textit{Substs} \ni \sigma, \sigma' &::= \mathsf{p} \mid \langle\rangle \mid \langle \sigma, t\rangle \mid \sigma\,\sigma' \mid \mathsf{id}\ . \end{aligned}$$

In the next sections we introduce the formal systems corresponding to both presentations of PTSs: λ^{σ} has an untyped notion of equality; and $\lambda^{\sigma=}$ features typed equality.

PTSs with untyped equality (λ^{σ})

The first family of type systems has an untyped notion of equality. It corresponds to the equivalence and congruence closure of the reductions $\to_{\beta}$ and $\to_{x}$. This equivalence relation on terms induces an equivalence relation on contexts.

Beta-reduction	**Congruence**
$\mathsf{App}\,(\lambda t)\,t' \to_{\beta} t\,\langle \mathsf{id}, t'\rangle$	$(\mathsf{Fun}\,A\,B)\,\sigma \to_{x} \mathsf{Fun}\,(A\,\sigma)\,(B\,\langle \sigma\,\mathsf{p}, \mathsf{q}\rangle)$
Substitutions	$t\,(\sigma\,\delta) \to_{x} (t\,\sigma)\,\delta$
$(\sigma\,\delta)\,\gamma \to_{x} \sigma\,(\delta\,\gamma)$	$t\,\mathsf{id} \to_{x} t$
$\sigma\,\mathsf{id} \to_{x} \sigma$	$\mathsf{q}\,\langle \sigma, t\rangle \to_{x} t$
$\mathsf{id}\,\sigma \to_{x} \sigma$	$(\lambda t)\,\sigma \to_{x} \lambda(t\,\langle \sigma\,\mathsf{p}, \mathsf{q}\rangle)$
$\langle\rangle\,\sigma \to_{x} \langle\rangle$	$(\mathsf{App}\,t\,t')\,\sigma \to_{x} \mathsf{App}\,(t\,\sigma)\,(t'\,\sigma)$
$\mathsf{p}\,\langle \sigma, t\rangle \to_{x} \sigma$	$s\,\sigma \to_{x} s$
$\langle \sigma, t\rangle\,\delta \to_{x} \langle \sigma\,\delta, t\,\delta\rangle$	

The typing rules of λ^{σ}, in Figs. 5.1 and 5.2, are basically the same as those in [92] without metavariables. In particular, we also have two rules for substitutions: (sub-tm) and (sub-sort). The latter rule is needed because otherwise we cannot resolve substitutions in the type when it is a top-sort. This problem is already discussed by Bloo [27]; his substitution rule uses explicit substitution on the subject, but implicit substitution in the type.

There is no rule like (conv-subs) neither in Bloo[27] nor in Muñoz[92]. In chapters 4 and 3 of this thesis the rule was not formulated explicitly, because it is a derived rule of the corresponding GATs. Without (conv-subs) it is not clear how to prove *context conversion*: if $\Gamma \vdash t : A$ and $\Gamma \to_{\beta} \Gamma'$, then $\Gamma' \vdash t : A$. For proving that result we need to prove the same property for substitutions; but that is not possible because in the typing rule for the identity substitution the target and source context should be syntactically equal.

Properties of λ^{σ} The inversion lemma of typing judgements, called generation lemma in the PTSs literature, can be proved by induction on derivations. A *type* is any sort or a term which can be typed with a sort.

Lemma 58 (Inversion of typing).

1. If $\Gamma \vdash s : C$, then there exists $(s, s') \in A$ and $C \equiv_{\beta x} s'$.
2. If $\Gamma \vdash \mathsf{q} : C$, then there exists $s \in S$, and $\Gamma' \vdash A : s$, $\Gamma = \Gamma.A$, and $C \equiv_{\beta x} A\,\mathsf{p}$.

$$\frac{}{\diamond \vdash}\ (\textsc{empty-ctx}) \qquad \frac{\Gamma \vdash \qquad \Gamma \vdash A : s}{\Gamma, A \vdash}\ (\textsc{ext-ctx})$$

$$\frac{\Gamma \vdash A : s \qquad \Gamma, A \vdash B : s'}{\Gamma \vdash \mathsf{Fun}\, A\, B : s''}\ (s, s', s'') \in R\ (\textsc{fun-f}) \qquad \frac{}{\Gamma \vdash c : s}\ (c, s) \in A\ (\textsc{axiom})$$

$$\frac{\Gamma \vdash \mathsf{Fun}\, A\, B : s'' \qquad \Gamma, A \vdash t : B}{\Gamma \vdash \lambda t : \mathsf{Fun}\, A\, B}\ (\textsc{fun-i}) \qquad \frac{\Gamma \vdash t : \mathsf{Fun}\, A\, B \qquad \Gamma \vdash r : A}{\Gamma \vdash \mathsf{App}\, t\, r : B\, \langle \mathsf{id}, r \rangle}\ (\textsc{fun-el})$$

$$\frac{\Delta \vdash A : s \qquad \Delta \vdash t : A \qquad \Gamma \vdash \sigma : \Delta}{\Gamma \vdash t\, \sigma : A\, \sigma}\ (\textsc{sub-tm}) \qquad \frac{\Delta \vdash A : s \qquad \Gamma \vdash \sigma : \Delta}{\Gamma \vdash A\, \sigma : s}\ (\textsc{sub-sort})$$

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash B : s \qquad A \equiv_{\beta x} B}{\Gamma \vdash t : B}\ (\textsc{conv}) \qquad \frac{\Gamma \vdash A : s}{\Gamma, A \vdash q : A\, p}\ (\textsc{hyp})$$

Figure 5.1: Rules for contexts and terms λ^σ

$$\frac{}{\Gamma \vdash \mathsf{id} : \Gamma}\ (\textsc{id-subs}) \qquad \frac{}{\Gamma \vdash \langle\rangle : \diamond}\ (\textsc{empty-subs}) \qquad \frac{\Gamma \vdash \qquad \Gamma \vdash A : s}{\Gamma, A \vdash p : \Gamma}\ (\textsc{fst-subs})$$

$$\frac{\Gamma \vdash \sigma : \Delta \qquad \Delta \vdash A : s \qquad \Gamma \vdash t : A\, \sigma}{\Gamma \vdash \langle \sigma, t \rangle : \Delta, A}\ (\textsc{ext-subs})$$

$$\frac{\Gamma \vdash \delta : \Theta \qquad \Theta \vdash \sigma : \Delta}{\Gamma \vdash \sigma\, \delta : \Delta}\ (\textsc{comp-subs}) \qquad \frac{\Gamma \vdash \sigma : \Delta \qquad \Delta' \equiv_{\beta x} \Delta}{\Gamma \vdash \sigma : \Delta}\ (\textsc{conv-subs})$$

Figure 5.2: Rules for substitutions of λ^σ

3. If $\Gamma \vdash \mathsf{Fun}\, A\, B : C$, then there exist $(s_0, s_1, s_2) \in R$, and $\Gamma \vdash A : s_0$, $\Gamma.A \vdash B : s_1$, and $C \equiv_{\beta\times} s_2$.

4. If $\Gamma \vdash \lambda t : C$, then there exist $A, B \in \mathit{Terms}$, $(s_0, s_1, s_2) \in R$, $\Gamma \vdash A : s_0$, $\Gamma.A \vdash B : s_1$, $\Gamma.A \vdash t : B$, and $C \equiv_{\beta\times} \mathsf{Fun}\, A\, B$.

5. If $\Gamma \vdash \mathsf{App}\, t\, r : C$, then there exist $A, B \in \mathit{Terms}$, $\Gamma \vdash t : \mathsf{Fun}\, A\, B$, $\Gamma \vdash r : A$, and $C \equiv_{\beta\times} B\, \langle \mathsf{id}, r \rangle$.

Proof. See `PtsInv.agda`. □

Lemma 59 (Type validity). If $\Gamma \vdash t : A$, then either $A \in S$ or there exists $s \in S$ and $\Gamma \vdash A : s$.

Proof. See `InvType` in `PtsInv.agda`. □

The formal proof of the substitution lemma is easier by extending subtitutions to act on contexts; we refer to the Agda formalisation for those definitions and the proofs of several properties. Note that $\langle \mathsf{id}, r \rangle$ is well-typed under $\Gamma.A$; given that substitution we can define $\widehat{\langle \mathsf{id}, r \rangle}$ which can be typed as $\Gamma.\Delta\, \langle \mathsf{id}, r \rangle \vdash \widehat{\langle \mathsf{id}, r \rangle} : \Gamma.A.\Delta$.

Lemma 60 (Substitution lemma). Let $\Gamma.A.\Delta \vdash$ and $\Gamma \vdash r : A$. Let $\Gamma' = \Gamma.\Delta\, \langle \mathsf{id}, r \rangle$.

1. If $\Gamma.A.\Delta \vdash t : s$, then $\Gamma' \vdash t\, \widehat{\langle \mathsf{id}, r \rangle} : s$.

2. If $\Gamma.A.\Delta \vdash B : s$ and $\Gamma.A.\Delta \vdash t : B$, then $\Gamma' \vdash t\, \widehat{\langle \mathsf{id}, r \rangle} : B\, \widehat{\langle \mathsf{id}, r \rangle}$.

3. If $\Gamma.A.\Delta \vdash \sigma : \Sigma$, then $\Gamma' \vdash \sigma\, \widehat{\langle \mathsf{id}, r \rangle} : \Sigma$.

Proof. See `PtsCtxCnv.agda`. □

Lemma 61 (Context Conversion). Let $\Gamma \equiv_{\beta\times} \Gamma'$.

1. $\Gamma' \equiv_{\beta\times} \Gamma$.

2. If $\Gamma \vdash t : A$, then $\Gamma' \vdash t : A$.

3. If $\Gamma \vdash \sigma : \Delta$, then $\Gamma' \vdash \sigma : \Delta$.

Proof.

1. See `CtxEqSym` in `PtsCtxCnv.agda`.

2. See `ctxTerm` in `PtsCtxCnv.agda`.

3. See `ctxSub` in `PtsCtxCnv.agda`.

□

$$\frac{}{\diamond \vdash_e} \text{(EMPTY-CTX)} \qquad \frac{\Gamma \vdash_e \quad \Gamma \vdash_e A : s}{\Gamma, A \vdash_e} \text{(EXT-CTX)}$$

$$\frac{\Gamma \vdash_e A : s \;\; \Gamma, A \vdash_e B : s' \;\; (s, s', s'') \in R}{\Gamma \vdash_e \mathsf{Fun}\, A\, B : s''} \text{(FUN-F)} \qquad \frac{(c, s) \in A}{\Gamma \vdash_e c : s} \text{(AXIOM)}$$

$$\frac{\Gamma \vdash_e \mathsf{Fun}\, A\, B : s'' \;\; \Gamma, A \vdash_e t : B}{\Gamma \vdash_e \lambda t : \mathsf{Fun}\, A\, B} \text{(FUN-I)} \qquad \frac{\Gamma \vdash_e t : \mathsf{Fun}\, A\, B \;\; \Gamma \vdash_e r : A}{\Gamma \vdash_e \mathsf{App}\; t\; r : B\, \langle \mathsf{id}, r \rangle} \text{(FUN-EL)}$$

$$\frac{\Gamma \vdash_e A : s}{\Gamma, A \vdash_e q : A\, p} \text{(HYP)} \qquad \frac{\Gamma \vdash_e t : A \qquad \Gamma \vdash_e A = B : s}{\Gamma \vdash_e t : B} \text{(CONV)}$$

$$\frac{\Delta \vdash_e A : s \;\; \Gamma \vdash_e \sigma : \Delta \;\; \Delta \vdash_e t : A}{\Gamma \vdash_e t\, \sigma : A\, \sigma} \text{(SUBS-TM)} \qquad \frac{\Gamma \vdash_e \sigma : \Delta \;\; \Delta \vdash_e A : s}{\Gamma \vdash_e A\, \sigma : s} \text{(SUBS-SORT)}$$

Figure 5.3: Typing rules for contexts and terms of $\lambda^{\sigma=}$

PTSs with typed equality ($\lambda^{\sigma=}$)

In $\lambda^{\sigma=}$ equality is axiomatised as in Martin-Löf type theory, i.e. by a system of axioms typeable under some context. To distinguish judgements of λ^{σ} from those of $\lambda^{\sigma=}$, we have a distinct turnstile ($\vdash_e$) for the latter. The typing rules are presented in Figs. 5.3 and 5.4; axioms are shown in Figs. 5.5, 5.6, and 5.7.

For $\lambda^{\sigma=}$ we need to introduce an additional form of judgements, that of conversion between contexts. It is enough to have reflexivity for the empty context (we skip that rule) and the following one for extended contexts:

$$\frac{\vdash_e \Gamma = \Gamma' \qquad \Gamma \vdash_e A : s \qquad \Gamma' \vdash_e B : s \qquad \Gamma \vdash_e A = B : s}{\vdash_e \Gamma.A = \Gamma'.B} \text{(EXT-EQ-CTX)}$$

The notion of normal forms (and neutrals) is only important for $\lambda^{\sigma=}$, because it is used in the model construction of Sec. 5.3:

$$\mathrm{Ne} \ni k ::= q \mid q\, p^{i+1} \mid \mathsf{App}\; k\, \nu$$
$$\mathrm{Nf} \ni \nu, V, W ::= s \mid \mathsf{Fun}\, V\, W \mid \lambda\nu \mid k \ .$$

Properties of $\lambda^{\sigma=}$ We have proved all the analogous results of λ^{σ} for $\lambda^{\sigma=}$. For proving inversion of typing, we introduce some predicates.

Definition 58.

$\Gamma \vdash_e t \approx t'$ This predicate says that t and t' can be proved equal with typed equality, but by changing the types in transitivity. It is axiomatised by the following clauses:

$$\frac{}{\Gamma \vDash_e \mathsf{id} : \Gamma}\ (\textsc{id-subs}) \qquad \frac{\Gamma \vDash_e \qquad \Gamma \vDash_e A : s}{\Gamma, A \vDash_e \mathsf{p} : \Gamma}\ (\textsc{fst-subs}) \qquad \frac{}{\Gamma \vDash_e \langle\rangle : \diamond}\ (\textsc{empty-subs})$$

$$\frac{\Gamma \vDash_e \sigma : \Delta \qquad \Delta \vDash_e A : s \qquad \Gamma \vDash_e t : A\,\sigma}{\Gamma \vDash_e \langle \sigma, t\rangle : \Delta, A}\ (\textsc{ext-subs})$$

$$\frac{\Gamma \vDash_e \delta : \Theta \qquad \Theta \vDash_e \sigma : \Delta}{\Gamma \vDash_e \sigma\,\delta : \Delta}\ (\textsc{comp-subs}) \qquad \frac{\Gamma \vdash \sigma : \Delta \qquad \vDash_e \Delta' = \Delta}{\Gamma \vdash \sigma : \Delta}\ (\textsc{conv-subs})$$

Figure 5.4: Typing rules for substitutions of $\lambda^{\sigma=}$

$$\begin{aligned}
(\mathsf{Fun}\,A\,B)\,\sigma &= \mathsf{Fun}\,(A\,\sigma)\,(B\,\langle \sigma\,\mathsf{p}, \mathsf{q}\rangle) & (\sigma\,\delta)\,\gamma &= \sigma\,(\delta\,\gamma)\\
c\,\sigma &= c & \sigma\,\mathsf{id} &= \sigma\\
t\,(\sigma\,\delta) &= (t\,\sigma)\,\delta & \mathsf{id}\,\sigma &= \sigma\\
t\,\mathsf{id} &= t & \langle\rangle\,\sigma &= \langle\rangle\\
\mathsf{q}\,\langle \sigma, t\rangle &= t & \mathsf{p}\,\langle \sigma, t\rangle &= \sigma\\
(\lambda t)\,\sigma &= \lambda(t\,\langle \sigma\,\mathsf{p}, \mathsf{q}\rangle) & \langle \sigma, t\rangle\,\delta &= \langle \sigma\,\delta, t\,\delta\rangle\\
(\mathsf{App}\ t\ t')\,\sigma &= \mathsf{App}\,(t\,\sigma)\,(t'\,\sigma)\\
\mathsf{App}\,(\lambda t)\,r &= t\,\langle \mathsf{id}, r\rangle
\end{aligned}$$

Figure 5.5: Axioms for terms and substitutions

1. $\Gamma \vDash_e t \approx t$.
2. If $\Gamma \vDash_e t = t' : A$, then $\Gamma \vDash_e t \approx t'$.
3. If $\Gamma \vDash_e t \approx t'$ and $\Gamma \vDash_e t' = t'' : A$, then $\Gamma \vDash_e t \approx t''$.

$\Gamma \vDash_e a \sim b$ This predicate indicates that either a and b are the same sort or $\Gamma \vDash_e a \approx b$.

Lemma 62 (Inversion of typing).

1. If $\Gamma \vDash_e s : C$, then there exists $(s, s') \in A$ and $\Gamma \vDash_e C \sim s'$.
2. If $\Gamma \vDash_e q : C$, then there exists $s \in S$, $\Gamma' \vDash_e$ and $A \in \mathit{Terms}$, such that $\Gamma' \vDash_e A : s$, $\Gamma = \Gamma.A$, and $\Gamma \vDash_e C \approx A\,\mathsf{p}$.
3. If $\Gamma \vDash_e \mathsf{Fun}\,A\,B : C$, then there exist $(s_0, s_1, s_2) \in R$, $\Gamma \vDash_e A : s_0$, $\Gamma.A \vDash_e B : s_1$, and $\Gamma \vDash_e C \sim s_2$.
4. If $\Gamma \vDash_e \lambda t : C$, then there exist $A, B \in \mathit{Terms}$, $(s_0, s_1, s_2) \in R$, $\Gamma \vDash_e A : s_0$, $\Gamma.A \vDash_e B : s_1$, $\Gamma.A \vDash_e t : B$, and $\Gamma \vDash_e C \approx \mathsf{Fun}\,A\,B$.
5. If $\Gamma \vDash_e \mathsf{App}\ t\ r : C$, then there exist $A, B \in \mathit{Terms}$, $\Gamma \vDash_e t : \mathsf{Fun}\,A\,B$, $\Gamma \vDash_e r : A$, and $\Gamma \vDash_e C \approx B\,\langle \mathsf{id}, r\rangle$,

$$\frac{\Gamma \vdash_e t : A}{\Gamma \vdash_e t = t : A}\ (\textsc{pts-refl}) \qquad \frac{\Gamma \vdash_e t = t' : A}{\Gamma \vdash_e t' = t : A}\ (\textsc{pts-sym})$$

$$\frac{\Gamma \vdash_e t = t' : A \qquad \Gamma \vdash_e t' = t'' : A}{\Gamma \vdash_e t = t'' : A}\ (\textsc{pts-trans})$$

$$\frac{\Gamma \vdash_e A = B : s \qquad \Gamma, A \vdash_e C = D : s' \qquad (s, s', s'') \in R}{\Gamma \vdash_e \mathsf{Fun}\, A\, C = \mathsf{Fun}\, B\, D : s''}\ (\textsc{prod-eq})$$

$$\frac{\Gamma, A \vdash_e t = t' : B \qquad \Gamma \vdash_e A : s \qquad \Gamma, A \vdash_e B : s' \qquad (s, s', s'') \in R}{\Gamma \vdash_e \lambda t = \lambda t' : \mathsf{Fun}\, A\, B}\ (\textsc{lam-eq})$$

$$\frac{\Gamma \vdash_e t_0 = t_0' : \mathsf{Fun}\, A\, B \qquad \Gamma \vdash_e t_1 = t_1' : A}{\Gamma \vdash_e \mathsf{App}\, t_0\, t_1 = \mathsf{App}\, t_0'\, t_1' : B\, \langle \mathsf{id}, t_1 \rangle}\ (\textsc{app-eq})$$

$$\frac{\Delta \vdash_e A : s \qquad \Delta \vdash_e t = t' : A \qquad \Gamma \vdash_e \sigma = \sigma' : \Delta}{\Gamma \vdash_e t\, \sigma = t'\, \sigma' : A\, \sigma}\ (\textsc{subs-eq})$$

$$\frac{\Gamma \vdash_e t = t' : s \qquad \Gamma \vdash_e \sigma = \sigma' : \Delta}{\Gamma \vdash_e t\, \sigma = t'\, \sigma' : s}\ (\textsc{subs-eq-s})$$

$$\frac{\Gamma \vdash_e t = t' : A \qquad \Gamma \vdash_e A = B : s}{\Gamma \vdash_e t = t' : B}\ (\textsc{conv-eq})$$

Figure 5.6: Equality is a congruence for term

$$\frac{\Gamma \vdash_e \sigma : \Delta}{\Gamma \vdash_e \sigma = \sigma : \Delta}\ (\textsc{s-refl}) \qquad \frac{\Gamma \vdash_e \sigma = \sigma' : \Delta}{\Gamma \vdash_e \sigma' = \sigma : \Delta}\ (\textsc{s-sym})$$

$$\frac{\Gamma \vdash_e \sigma = \sigma' : \Delta \qquad \Gamma \vdash_e \sigma' = \sigma'' : \Delta}{\Gamma \vdash_e \sigma = \sigma'' : \Delta}\ (\textsc{s-trans})$$

$$\frac{\Gamma \vdash_e \delta = \delta' : \Theta \qquad \Theta \vdash_e \sigma = \sigma' : \Delta}{\Gamma \vdash_e \sigma\, \delta = \sigma'\, \delta' : \Delta}\ (\textsc{comp-eq})$$

$$\frac{\Delta \vdash_e A : s \qquad \Gamma \vdash_e \sigma = \sigma' : \Delta \qquad \Gamma \vdash_e t = t' : A\, \sigma}{\Gamma \vdash_e \langle \sigma, t \rangle = \langle \sigma', t' \rangle : \Delta, A}\ (\textsc{ext-eq})$$

Figure 5.7: Equality is a congruence for substitutions

Proof. See `EPtsGen.agda`. □

Using the same strategy as for λ^{σ} we proved substitution lemma for $\lambda^{\sigma=}$. We do not repeat all the definitions needed for this proof.

Lemma 63 (Substitution lemma). Let $\Gamma.A.\Delta \vdash_{e}$ and $\Gamma \vdash_{e} t : A$; let $\Gamma' = \Gamma.\Delta\,\langle \mathsf{id}, r\rangle$.

1. If $\Gamma.A.\Delta \vdash_{e} t : s$, then $\Gamma' \vdash_{e} t\,\widehat{\langle \mathsf{id}, r\rangle} : s$.
2. If $\Gamma.A.\Delta \vdash_{e} B : s$ and $\Gamma.A.\Delta \vdash_{e} t : B$, then $\Gamma' \vdash_{e} t\,\widehat{\langle \mathsf{id}, r\rangle} : B\,\widehat{\langle \mathsf{id}, r\rangle}$.
3. If $\Gamma.A.\Delta \vdash_{e} \sigma : \Sigma$, then $\Gamma' \vdash_{e} \sigma\,\widehat{\langle \mathsf{id}, r\rangle} : \Sigma$.
4. If $\Gamma.A.\Delta \vdash_{e} t = t' : s$, then $\Gamma' \vdash_{e} t\,\widehat{\langle \mathsf{id}, r\rangle} = t'\,\widehat{\langle \mathsf{id}, r\rangle} : s$.
5. If $\Gamma.A.\Delta \vdash_{e} B : s$ and $\Gamma.A.\Delta \vdash_{e} t = t' : B$, then $\Gamma' \vdash_{e} t\,\widehat{\langle \mathsf{id}, r\rangle} = t'\,\widehat{\langle \mathsf{id}, r\rangle} : B\,\widehat{\langle \mathsf{id}, r\rangle}$.
6. If $\Gamma.A.\Delta \vdash_{e} \sigma = \sigma' : \Sigma$, then $\Gamma' \vdash_{e} \sigma\,\widehat{\langle \mathsf{id}, r\rangle} = \sigma'\,\widehat{\langle \mathsf{id}, r\rangle} : \Sigma$.

Proof. See `EPtsSubs.agda`. □

The only meta-theorem for $\lambda^{\sigma=}$ that we have not yet formalised in Agda is context conversion; we present the proof.

Lemma 64 (Context conversion). Let $\Gamma.A.\Delta \vdash_{e} J$ and $\Gamma \vdash_{e} A = B : s$, then $\vdash_{e} \Gamma.A.\Delta = \Gamma.B.\Delta$ and $\Gamma.B.\Delta \vdash_{e} J$.

Proof. We prove the two points simultaneously; the first, by induction on the length of Δ; and the second, by induction on derivations. We show some cases.

1. (ext-ctx): The premises are $\Gamma.A.\Delta \vdash_{e}$ and $\Gamma.A.\Delta \vdash_{e} C : s$; the conclusion is $\Gamma.A.\Delta.C \vdash_{e}$. By i.h. on the first premise we have $\Gamma.B.\Delta \vdash_{e}$ and $\vdash_{e} \Gamma.A.\Delta = \Gamma.B.\Delta$. By i.h. on the second premise, $\Gamma.B.\Delta \vdash_{e} C : s$. From those three facts, we conclude $\Gamma.A.\Delta.C \vdash_{e}$, by (ext-ctx); and $\vdash_{e} \Gamma.A.\Delta.C = \Gamma.B.\Delta.C$, by (ext-eq-ctx), for applying this rule we deduce, by (refl), $\Gamma.A.\Delta \vdash_{e} C = C : s$.
2. (id-subs): the premise is $\Gamma.A.\Delta \vdash_{e}$; by i.h. we know $\Gamma.B.\Delta \vdash_{e}$ and $\vdash_{e} \Gamma.A.\Delta = \Gamma.B.\Delta$. From the first, we derive $\Gamma.B.\Delta \vdash_{e} \mathsf{id} : \Gamma.B.\Delta$, and by (conv-subs) we conclude $\Gamma.B.\Delta \vdash_{e} \mathsf{id} : \Gamma.A.\Delta$.

□

Postulating context conversion, we have formally proved type validity and equality validity. The predicate $\Gamma \vdash_{e} A$ means that either $A \in S$ or there exists $s \in S$ and $\Gamma \vdash_{e} A : s$.

Lemma 65 (Type validity). If $\Gamma \vdash_{e} t : A$, then $\Gamma \vdash_{e} A$.

Proof. See `InvTy` in `EPtsValidity.agda`. □

Lemma 66 (Equality validity). If $\Gamma \vdash_{e} t = t' : A$, then $\Gamma \vdash_{e} A$, $\Gamma \vdash_{e} t : A$ and and $\Gamma \vdash_{e} t' : A$.

Proof. See `InvEqTy` in `EPtsValidity.agda`. □

5.2 Equivalence between λ^{σ} and $\lambda^{\sigma=}$

Now that we have formally introduced the two families of PTSs, we can precisely state the equivalence.

Theorem 67 (Equivalence between λ^{σ} and $\lambda^{\sigma=}$ [104]).

1. $\Gamma \vDash_{\mathsf{e}} t : A$ iff $\Gamma \vdash t : A$.
2. $\Gamma \vDash_{\mathsf{e}} t = t' : A$ iff $\Gamma \vdash t : A, \Gamma \vdash t' : A,$ and $t \equiv_{\beta x} t'$.

This theorem was conjectured by Geuvers [59] for PTSs with (η). Adams [9] proved the equivalence for functional PTSs. Later, Herbelin and Siles [105] extended and formalised Adams' result to semi-full systems; Siles finally settled in his PhD thesis [104] the equivalence for every PTS. Already in Geuvers' thesis it is recognised that the direction from λ^{σ} to $\lambda^{\sigma=}$ is easy; in fact Adams proved that direction for every PTS. The key result to prove the other implication depends on *subject reduction* for $\lambda^{\sigma=}$: *if* $\Gamma \vDash_{\mathsf{e}} t : A$ *and* $t \equiv_{\beta x} t'$, *then* $\Gamma \vDash_{\mathsf{e}} t' : A$. In sections 5.3 and 5.4 we use the machinery of NbE to prove a slightly weaker, but strong enough, version of that lemma for a class of PTSs. In contrast with our semantical approach, Adams and Siles results are based on a typed parallel reduction for PTSs with typed equality.

From $\lambda^{\sigma=}$ *to* λ^{σ} The proof that any derivation in $\lambda^{\sigma=}$ has a corresponding derivation in λ^{σ} is relatively straightforward and can be proved for every PTS. The formal statement of the theorem is the following. We have formally proved it in Agda.

Theorem 68.

1. If $\Gamma \vDash_{\mathsf{e}}$, then $\Gamma \vdash$.
2. If $\vDash_{\mathsf{e}} \Gamma = \Gamma'$, then $\Gamma \equiv_{\beta x} \Gamma'$.
3. If $\Gamma \vDash_{\mathsf{e}} t : A$, then $\Gamma \vdash t : A$.
4. If $\Gamma \vDash_{\mathsf{e}} \sigma : \Delta$, then $\Gamma \vdash \sigma : \Delta$.
5. If $\Gamma \vDash_{\mathsf{e}} t = t' : A$, then $\Gamma \vdash t : A$, $\Gamma \vdash t' : A$, and $t \equiv_{\beta x} t'$.
6. If $\Gamma \vDash_{\mathsf{e}} \sigma = \sigma' : \Delta$, then $\Gamma \vdash \sigma : \Delta$, $\Gamma \vdash \sigma' : \Delta$, and $\sigma \equiv_{\beta x} \sigma'$.

Proof. See `EPtsToPts.agda`. □

5.3 Semantics for $\lambda^{\sigma=}$

The model of $\lambda^{\sigma=}$ are parameterised by the signature of the PTS. Given a signature $S = (S, A, R)$, let D_S be the least solution for

$$\mathsf{D} \approx \mathbb{O} \oplus \mathsf{D} \times \mathsf{D} \oplus [\mathsf{D} \to \mathsf{D}] \oplus \mathsf{D} \times [\mathsf{D} \to \mathsf{D}] \oplus \mathit{Var}_{\perp} \oplus \mathsf{D} \times \mathsf{D} \oplus S_{\perp} \ .$$

Besides the convention for naming elements of D_S inherited from previous chapters, we also write $s \in \mathsf{D}$ for $s \in S$. In the following we omit the subscript S from D_S.

Definition 59 (Readback function).

$$\begin{aligned}
\mathsf{R}_j\ s &= s \\
\mathsf{R}_j\ (\mathsf{Fun}\ X\ F) &= \mathsf{Fun}\ (\mathsf{R}_j\ X)\ (\mathsf{R}_{j+1}\ (F\ \mathsf{Var}\ j)) \\
\mathsf{R}_j\ (\mathsf{App}\ d\ d') &= \mathsf{App}\ (\mathsf{R}_j\ d)\ (\mathsf{R}_j\ d') \\
\mathsf{R}_j\ (\mathsf{lam}\ f) &= \lambda(\mathsf{R}_{j+1}\ f(\mathsf{Var}\ j)) \\
\mathsf{R}_j\ (\mathsf{Var}\ i) &= \begin{cases} q & \text{if } j \leqslant i+1 \\ q\,p^{j-(i+1)} & \text{if } j > i+1 \end{cases}
\end{aligned}$$

As in previous chapters, we identify in D the set of semantical neutral values and semantical normal forms by taking the inverse image of R:

$$\mathsf{Ne} = \bigcap_{i\in\mathbb{N}} \{d \in D \mid \mathsf{R}_i\ d \in \mathsf{Ne}\} \quad \text{and} \quad \mathsf{Nf} = \bigcap_{i\in\mathbb{N}} \{d \in D \mid \mathsf{R}_i\ d \in \mathsf{Nf}\}\ .$$

Denoting types and sorts

Since our presentation of PTSs does not include (η), we can base our interpretation of sorts and types as we interpreted universes and types in Chap. 3: one has a universe of *types* and a function $[\_]$ mapping elements of that universe to subsets of the domain. The universe of types and the function $[\_]$ were introduced following the schema of inductive-recursive definitions. A key result for the normalisation algorithm was that *types* itself and the interpretation of each type were saturated, cf. Lem. 16 and Lem. 38.

Let us analyse informally which elements of D should be in the subset denoting types. By rule (Axiom) we know that sorts are among the possible *types*; so, S should be included in the universe *types*. Note also that terms typed with a sort can also be types; which suggests to add any element of $[s]$ to *types*, for each $s \in S$. The definition of $[s] \subseteq D$ depends both on axioms and rules of the system: axioms tell us that if $(s', s) \in A$, then $s' \in [s]$. The set R of rules indicates when one element of the form $\mathsf{Fun}\ d\ f$ should be a member of $[s]$. As we add $\mathsf{Fun}\ d\ f$ to *types* as soon as it is in some $[s]$, we have to define $[\mathsf{Fun}\ d\ f]$.

Our use, in previous chapters, of the schema of inductive-recursive definitions for *types* and $[\_]$ was justified by the existence of a well-founded order on universes. There are PTSs for which such an order cannot exists; these are impredicative PTSs. In particular, if we consider the type-system *type-in-type* given by the signature $\lambda^\star = (\{\star\}, \{(\star,\star)\}, \{(\star,\star,\star)\})$ we would have the following clause:

$$\mathsf{Fun}\ d\ f \in [\star],\ \text{if } d \in [\star] \text{ and } f\ e \in [\star] \text{ for all } e \in [d]\ .$$

This is a typical issue of impredicative definitions: to decide if some element belongs to $[\star]$ we should test a condition over all the elements in that very same set. For concreteness one can consider $\mathsf{Fun}\ \star\ f \in [\star]$, for some $f \in [D \to D]$.

We can safely use the inductive-recursive schema by restricting our attention to predicative PTSs as captured by the following definition.

Definition 60 (Predicative PTSs). A specification $S = (S, A, R)$ is called *predicative* if there is a well-founded order over the sorts, $\preccurlyeq \subseteq S^2$, such that:

- if $(s_1, s_2) \in A$, then $s_1 \prec s_2$;

- if $(s_1, s_2, s_3) \in R$, then $s_1 \preccurlyeq s_3$ and $s_2 \preccurlyeq s_3$.

The next definition introduces subsets of D for modelling predicative PTSs. In order to understand it, one should think that first one defines [s] inductively for every minimal element $s \in S$ (minimal with respect to the underlying order making the signature predicative).

Definition 61. Let $S = (S, A, R)$ be a predicative signature. We define simultaneously $T \subseteq \mathsf{D}$ and $[d] \subseteq \mathsf{D}$, for every $d \in T$.

$$\frac{}{\mathsf{Ne} \subseteq T} \qquad \frac{}{S \subseteq T}$$

$$\frac{(s, s') \in A}{s \in [s']} \qquad \frac{s \in T}{[s] \subseteq T}$$

$$\frac{(s, s', s'') \in R \qquad d \in [s] \qquad f\,e \in [s'], \text{for all } e \in [d]}{\mathsf{Fun}\,d\,f \in [s'']}$$

$$\frac{d \in T}{\mathsf{Ne} \subseteq [d]} \qquad \frac{g\,e \in [f\,e], \text{for all } e \in [d]}{\mathsf{lam}\,g \in [\mathsf{Fun}\,d\,f]}$$

The well-founded order over S gives us a way to reason about elements of T. In fact, it induces a well-founded order $\sqsubseteq$ over $[s]$, for every $s \in S$, and over T:

- minimal elements with respect of $\sqsubseteq$ are elements in Ne and minimal elements with respect to $\preccurlyeq$,
- $s \sqsubseteq s'$ if $s \preccurlyeq s'$, and
- for $(s, s', s'') \in R$, then $d \sqsubseteq \mathsf{Fun}\,d\,f$ for each $d \in [s]$ and $f\,e \sqsubseteq \mathsf{Fun}\,d\,f$, for all $e \in [d]$.

The first result we prove using well-founded induction over T is that every subset denoting types, that is $[d]$ for every element $d \in T$, is saturated: let $X \subseteq \mathsf{D}$, X is called *saturated* if $\mathsf{Ne} \subset X \subset \mathsf{Nf}$.

Lemma 69. For all $d \in T$, $[d]$ is saturated.

Proof. By well-founded induction over $\sqsubseteq$.

1. Let d be a minimal element, then $[d] = \mathsf{Ne}$.
2. If d is not minimal, then either $d = s$ or $d = \mathsf{Fun}\,d'\,f$.
   a) for $d = s$: since $S \subseteq \mathsf{Nf}$, we only consider $\mathsf{Fun}\,d'\,f \in [s]$ arising from some rule $(s_0, s_1, s) \in R$. By i.h. on s_0, $d' \in \mathsf{Nf}$; and by i.h. on d', $\mathsf{Ne} \subseteq [d']$; therefore $f\,(\mathsf{Var}\,i) \in [s_1]$ for every $\mathsf{Var}\,i$. By i.h. on s_1, $f\,(\mathsf{Var}\,i) \in \mathsf{Nf}$. Therefore $\mathsf{Fun}\,d'\,f \in \mathsf{Nf}$.
   b) for $d = \mathsf{Fun}\,d'\,f$: if $e \in [\mathsf{Fun}\,d'\,f]$, then either $e \in \mathsf{Ne}$ or $e = \mathsf{lam}\,g$. We consider only this last case: by i.h. on d' we know $\mathsf{Ne} \subseteq [d']$; therefore, for every $\mathsf{Var}\,i$, $g\,(\mathsf{Var}\,i) \in [f\,(\mathsf{Var}\,i)]$. By i.h. on $f\,(\mathsf{Var}\,i)$ we know $g\,(\mathsf{Var}\,i) \in \mathsf{Nf}$.

□

Corollary 70. *T is saturated: if $d \in T$, then there is some $e \in T$, such that $d \in [e]$.*

Interpretation and Soundness The missing pieces of the model are the interpretation of terms and the satisfaction of judgements by the model. As in previous chapters, we introduce two functions $[\![\_]\!]^s\_$ and $[\![\_]\!]^t\_$.

$$
\begin{aligned}
[\![\_]\!]^t\_ &: \mathit{Terms} \times D \to D \\
[\![s]\!]^t d &= s \\
[\![q]\!]^t d &= \mathsf{snd}\ \ d \\
[\![\mathsf{App}\ t\ u]\!]^t d &= [\![t]\!]^t d \cdot [\![u]\!]^t d \\
[\![\lambda t]\!]^t d &= \mathsf{lam}\,(d' \mapsto [\![t]\!]^t(d, d')) \\
[\![t\,\gamma]\!]^t d &= [\![t]\!]^t([\![\gamma]\!]^s d) \\
[\![\mathsf{Fun}\ A\ B]\!]^t d &= \mathsf{Fun}\ ([\![A]\!]^t d)\ (e \mapsto [\![B]\!](d, e))
\end{aligned}
\qquad
\begin{aligned}
[\![\_]\!]^s\_ &: \mathit{Substs} \times D \to D \\
[\![\langle\rangle]\!]^s d &= \top \\
[\![\mathsf{id}]\!]^s d &= d \\
[\![(\gamma, t)]\!]^s d &= ([\![\gamma]\!]^s d, [\![t]\!]^t d) \\
[\![p]\!]^s d &= \mathsf{fst}\ \ d \\
[\![\gamma\ \delta]\!]^s d &= [\![\gamma]\!]^s([\![\delta]\!]^s d)
\end{aligned}
$$

Projections fst and snd are defined as in Chap. 3; the evaluation operator $\_ \cdot \_$ is, as in Chap. 3, defined for semantical neutrals because we do not consider (η):

$$
d \cdot e = \begin{cases} f\ e & \text{if } d = \mathsf{lam}\ f \\ \mathsf{App}\ d\ e & \text{if } d \in \mathsf{Ne} \\ \bot & \text{otherwise} \end{cases} .
$$

Remark 24. Notice that if $d \in [\mathsf{Fun}\ X\ F]$ and $e \in [X]$, then $d \cdot e \in [F\ e]$.

Since the interpretation is given for pre-terms and pre-substitutions, we can prove that the interpretation model the untyped equality.

Lemma 71. *If $t \equiv_{\beta x} t'$ and $d \in D$, then $[\![t]\!]d = [\![t']\!]d$.*

Proof. By induction on $t \equiv_{\beta x} t'$; note that it is enough to prove that the interpretation models the reduction relations. We show some cases.

1. (beta):

$$
\begin{aligned}
[\![\mathsf{App}\ (\lambda t)\ t']\!]d = [\![\lambda t]\!]d \cdot [\![t']\!]d &= \mathsf{lam}\,(e \mapsto [\![t]\!](d, e)) \cdot [\![t']\!]d \\
&= [\![t]\!](d, [\![t']\!]d) = [\![t]\!]([\![\langle \mathsf{id}, t'\rangle]\!]d) = [\![t\ \langle \mathsf{id}, t'\rangle]\!]d
\end{aligned}
$$

2. (fun-subs):

$$
\begin{aligned}
[\![(\mathsf{Fun}\ A\ B)\ \sigma]\!]d &= [\![\mathsf{Fun}\ A\ B]\!]([\![\sigma]\!]d) \\
&= \mathsf{Fun}\ ([\![A]\!]([\![\sigma]\!]d))\ (e \mapsto [\![B]\!]([\![\sigma]\!]d, e)) \\
&= \mathsf{Fun}\ ([\![A]\!]([\![\sigma]\!]d))\ (e \mapsto [\![B]\!]([\![\sigma]\!]([\![p]\!](d, e)), e)) \\
&= \mathsf{Fun}\ ([\![A]\!]([\![\sigma]\!]d))\ (e \mapsto [\![B]\!]([\![\sigma\ p]\!](d, e), [\![q]\!](d, e))) \\
&= \mathsf{Fun}\ ([\![A]\!]([\![\sigma]\!]d))\ (e \mapsto [\![B]\!]([\![\langle \sigma\ p, q\rangle]\!](d, e)))
\end{aligned}
$$

□

The semantics of judgements uses the set T and the mapping $\lfloor\_\rfloor : T \to \mathcal{P}(D)$. If $\Gamma \vdash t : A$, we want $[\![A]\!]^t d \in T$ and, be that the case, $[\![t]\!]^t d \in \lfloor [\![A]\!]^t d \rfloor$. In the following definition we state formally the validity of judgements in the model and introduce, at the same time, the semantics of well-formed contexts.

Definition 62 (Validity).

1. a) $\diamond \vDash$;
   b) $\top \in [\![\diamond]\!]$.
2. a) $\Gamma, A \vDash$ iff $\Gamma \vDash A : s$;
   b) If $d \in [\![\Gamma]\!]$ and $d' \in [[\![A]\!]d]$, then $(d, d') \in [\![\Gamma, A]\!]$.
3. $\vDash \Gamma = \Gamma'$ iff $d \in [\![\Gamma]\!]$ iff $d \in [\![\Gamma]\!]'$.
4. $\Gamma \vDash t : A$ iff
   a) A is a top-sort, say s: $\Gamma \vDash$ and $[\![t]\!]d \in [s]$, for all $d \in [\![\Gamma]\!]$.
   b) $\Gamma \vdash A : s$: $\Gamma \vDash A : s$ and $[\![t]\!]d \in [[\![A]\!]d]$, for all $d \in [\![\Gamma]\!]$.
5. $\Gamma \vDash \delta : \Delta$ iff $\Gamma \vDash$, $\Delta \vDash$ and $[\![\delta]\!]d \in [\![\Delta]\!]$, for all $d \in [\![\Gamma]\!]$.
6. $\Gamma \vDash t = t' : A$ iff $\Gamma \vDash t : A$ and $[\![t]\!]d = [\![t']\!]d$, for all $d \in [\![\Gamma]\!]$.
7. $\Gamma \vDash \delta = \delta' : \Delta$ iff $\Gamma \vDash \delta : \Delta$ and $[\![\delta]\!]d = [\![\delta']\!]d$, for all $d \in [\![\Gamma]\!]$.

Theorem 72 (Soundness of the model). If $\Gamma \vdash_{\mathsf{e}} J$, then $\Gamma \vDash J$.

Proof. By induction on $\Gamma \vdash_{\mathsf{e}} J$; note that the last two items are proved by the inductive hypothesis and using Lem. 71.

1. (EMPTY-CTX): trivial.
2. (EXT-CTX) $\Gamma.A \vdash_{\mathsf{e}}$: by i.h. on $\Gamma \vdash_{\mathsf{e}} A : s$.
3. (AXIOM) $\Gamma \vdash_{\mathsf{e}} s' : s$: by i.h. $\Gamma \vDash$ and by definition of the model $s \in [s']$.
4. (FUN-F) $\Gamma \vdash_{\mathsf{e}} \mathsf{Fun}\, A\, B : s_2$: by i.h. $\Gamma \vDash$, $\Gamma \vDash A : s_0$; also by i.h. $\Gamma.A \vDash B : s_1$. From the last two conditions we see that $\mathsf{Fun}\, [\![A]\!]d\, e \mapsto [\![B]\!](d, e) \in [s_2]$ for any $d \in [\![\Gamma]\!]$.
5. (FUN-I) $\Gamma \vdash_{\mathsf{e}} \lambda t : \mathsf{Fun}\, A\, B$: by i.h. $\Gamma \vDash$, $\Gamma \vDash A : s_0$, $\Gamma.A \vDash B : s_1$, and $\Gamma.A \vDash t : B$. The last condition implies $[\![t]\!](d, e) \in [[\![B]\!](d, e)]$ for any $d \in [\![\Gamma]\!]$ and $e \in [[\![A]\!]d]$; therefore $[\![\lambda t]\!]d \in [[\![\mathsf{Fun}\, A\, B]\!]d]$ for any $d \in [\![\Gamma]\!]$.
6. (FUN-EL) $\Gamma \vdash_{\mathsf{e}} \mathsf{App}\, t\, r : B\, (\mathsf{id}, r)$: by i.h. $\Gamma \vDash t : \mathsf{Fun}\, A\, B$ and $\Gamma \vDash r : A$; therefore for any $d \in [\![\Gamma]\!]$, $[\![r]\!]d \in [[\![A]\!]d]$. By Rem. 24, $[\![t]\!]d \cdot [\![r]\!]d \in [[\![B]\!](d, [\![r]\!]d)]$.
7. (SUB-TM) $\Gamma \vdash_{\mathsf{e}} t\, \sigma : A\, \sigma$: by i.h. $\Gamma \vDash \sigma : \Delta$. For $d \in [\![\Gamma]\!]$, then $[\![\sigma]\!]d \in [\![\Delta]\!]$; therefore by i.h. on $\Delta \vdash_{\mathsf{e}} t : A$, we have $[\![t]\!]([\![\sigma]\!]d) \in [[\![A]\!]([\![\sigma]\!]d)]$.
8. (SUB-SORT) $\Gamma \vdash_{\mathsf{e}} A\, \sigma : s$: as in the previous item.
9. (HYP) $\Gamma.A \vdash_{\mathsf{e}} q : A\, p$: $(d, e) \in [\![\Gamma.A]\!]$ if $d \in [\![\Gamma]\!]$ and $e \in [[\![A]\!]d]$; that is enough to conclude, because $[\![A\, p]\!](d, e) = [\![A]\!]d$ and $[\![q]\!](d, e) = e$.
10. (CONV) $\Gamma \vdash_{\mathsf{e}} t : B$: by i.h. $[\![t]\!]d \in [[\![A]\!]d]$ for any $d \in [\![\Gamma]\!]$; by i.h. $[\![A]\!]d = [\![B]\!]d$, therefore $[\![t]\!]d \in [[\![B]\!]d]$.
11. (EXT-SUBS) $\Gamma \vdash_{\mathsf{e}} (\sigma, t) : \Delta.A$: for $d \in [\![\Gamma]\!]$, by i.h. $[\![\sigma]\!]d \in [\![\Gamma]\!]$ and $[\![t]\!]d \in [[\![A\, \sigma]\!]d]$.

12. (comp-subs) $\Gamma \models_e \sigma\,\delta : \Delta$: using the same reasoning as for (sub-tm).

13. (fst-subs) $\Gamma.A \models_e \mathsf{p} : \Gamma$: using the same reasoning as for (hyp).

□

By soundness we know that each typed term is interpreted as an element of the denotation of the type; moreover we know that judgmentally equal elements have the same denotation. As a corollary we know that they have the same normal form.

Corollary 73 (Completeness of normalisation). If $\Gamma \vdash t = t' : A$ and $d \in [\![\Gamma]\!]$, then $\mathsf{R}_j\,([\![t]\!]d) \equiv \mathsf{R}_j\,([\![t']\!]d)$, for any $j \in \mathbb{N}$.

5.4 Correctness of Nbe

In this section we prove that Fun is injective; i.e., if $\Gamma \models_e \mathsf{Fun}\,A\,B = \mathsf{Fun}\,A'\,B' : C$, then $\Gamma \models_e A = A' : s'$ and $\Gamma, A \models_e B = B' : s''$. As in previous chapters we use logical relations to show that each typeable term is judgmentally equal to its normal form as computed by our proposed **nbe** function. Remember that for predicative PTSs we have a well-founded order on T; this order is used to define the logical relations and to prove all their results.

The logical relations are slightly different than those for Martin-Löf type theory: for PTSs there is only one kind of logical relations relating terms typable with A under context Γ with elements of $[X]$ with $X \in T$:

$$\Gamma \models_e \_ : A \sim \_ \in [X] \subseteq \{t \mid \Gamma \vdash t : A\} \times [X] \ .$$

Definition 63 (Logical Relations). Let $X \in T$ and $\Gamma \models_e A : s$.

1. $X \in \mathsf{Ne}$: $\Gamma \models_e t : A \sim d' \in [X]$ iff for all weakening $\Delta \leqslant^i \Gamma$:
   a) $\Delta \models_e A\,\mathsf{p}^i = \mathsf{R}_{|\Delta|}\,X : s$
   b) $\Delta \models_e t\,\mathsf{p}^i = \mathsf{R}_{|\Delta|}\,d' : A\,\mathsf{p}^i$
2. $X = s \in S$: $\Gamma \models_e t : A \sim d \in [X]$ iff
   a) if $d \in \mathsf{Ne} \cup S$: $\Delta \models_e t\,\mathsf{p}^i = \mathsf{R}_{|\Delta|}\,d : s$
   b) if $d = \mathsf{Fun}\,X'\,F$: there exists $(s_0, s_1, s) \in R$ and
      i. $\Gamma \models_e t = \mathsf{Fun}\,A'\,B : s$
      ii. $\Gamma \models_e A' : s_0 \sim X' \in [s_0]$, and
      iii. $\Delta \models_e B\,\langle \mathsf{p}^i, r\rangle : s_1 \sim F\,e \in [s_1]$, for all $\Delta \models_e r : A'\,\mathsf{p}^i \sim e \in [X']$.
3. $X = \mathsf{Fun}\,X'\,F$: $\Gamma \models_e t : A \sim d \in [\mathsf{Fun}\,X'\,F]$ if and only if
   a) there exists $(s_0, s_1, s) \in R$ and $\Gamma \models_e A = \mathsf{Fun}\,A'\,B : s$, $\Gamma \models_e A' : s_0 \sim X' \in [s_0]$,
   b) for all $\Delta \leqslant^i \Gamma$ and $\Delta \models_e r : A'\,\mathsf{p}^i \sim d' \in [X]$, $\Delta \models_e \mathsf{App}\,(t\,\mathsf{p}^i)\,r : B\,(\mathsf{p}^i, r) \sim d \cdot d' \in [F\,d']$;
   c) i. if $d = \mathsf{lam}\,f$, then $\Gamma \vdash t = \lambda\,t' : A$,
      ii. if $d \in \mathsf{Ne}$, then $\Delta \vdash (t\,\mathsf{p}^i) = \mathsf{R}_{|\Delta|}\,d : A\,\mathsf{p}^i$.

The following lemma is proved analogously as we did in previous chapters: we use (SYM) and (TRANS) on the main conditions of the definition of the logical relation. We skip its proofs.

Lemma 74 (Preservation of the logical relation by judgemental equality). If $\Gamma \vDash_e t : A \sim d \in [X]$ and $\Gamma \vDash_e t = t' : A$, then $\Gamma \vDash_e t' : A \sim d \in [X]$. Moreover, if $\Gamma \vDash_e t : A \sim d \in [X]$ and $\Gamma \vDash_e A = B : s$, then $\Gamma \vDash_e t : B \sim d \in [X]$. □

Lemma 75 (Monotonicity of logical relations). If $\Gamma \vDash_e t : A \sim d \in [X]$ and $\Delta \leqslant^i \Gamma$, then $\Delta \vDash_e t\, p^i : A\, p^i \sim d \in [X]$.

Lemma 76 (Derivability of equality of reified elements). Let $\Gamma \vDash_e t : A \sim d \in [X]$ and $\Delta \leqslant^i \Gamma$, then $\Delta \vDash_e t\, p^i = R_{|\Delta|}\, d : R_{|\Delta|}\, X$. Conversely, if $\Gamma \vDash_e A : s \sim X \in [s]$, $k \in \mathsf{Ne}$ and $\Delta \vDash_e t\, p^i = R_{|\Delta|}\, k : A\, p^i$, for all $\Delta \leqslant^i \Gamma$, then $\Gamma \vDash_e t : A \sim k \in [X]$.

Proof. (By induction on $X \in T$.)

1. $X \in \mathsf{Ne}$: both parts are obtained from the hypothesis, because those are the condition to be logically related.

2. $X = s \in S$: by induction on $d \in [s]$: The first part for $d \in \mathsf{Ne}$ and $d \in S$ are obtained directly by definition. Moreover the second part is trivial, because the hypothesis is the main condition in the definition of the logical relation.

   We show only the first part for $d = \mathsf{Fun}\, X'\, F$: by i.h. on $\Gamma \vDash_e A : s_0 \sim X' \in [s_0]$ we have $\Delta \vDash_e A\, p^i = R_{|\Delta|}\, X' : s_0$. By i.h. on $\Delta \vDash_e B\, (p^i, r) : s_1 \sim F\, e \in [s_1]$. On the other hand we have $\Delta' \vDash_e q\, p^j = R_{|\Delta|}\, (\mathsf{Var}\, |\Gamma|) : A\, p^j$, for all $\Delta' \leqslant^j \Gamma.A$, so by i.h. on the second part, we know $\Gamma.A \vDash_e q : A\, p \sim \mathsf{Var}\, |\Gamma| \in [X']$. From which we conclude $\Gamma.A \vDash_e B\, (p, q) = R_{|\Delta|}\, (F\, (\mathsf{Var}\, |\Gamma|)) : s_1$. Using (PROD-EQ) we conclude the equality.

3. $X = \mathsf{Fun}\, X'\, F$: For the first part, we only show the case when $d = \mathsf{lam}\, f$.

   a) If $d = \mathsf{lam}\, f$: by definition of the logical relation $\Gamma \vDash_e t = \lambda t' : A$. As was shown in the previous point we have $\Gamma.A' \vDash_e q : A'\, p \sim \mathsf{Var}\, |\Gamma| \in [X']$; by definition of the logical relation and Lem. 74 we know $\Gamma.A' \vDash_e \mathsf{App}\, ((\lambda t')\, p)\, q : B\, (p, q) \sim \mathsf{lam}\, f \cdot (\mathsf{Var}\, |\Gamma|) \in [F\, (\mathsf{Var}\, |\Gamma|)]$. By Lem. 74 and by definition of application $\Gamma.A' \vDash_e t' : B \sim f\, (\mathsf{Var}\, |\Gamma|) \in [F\, (\mathsf{Var}\, |\Gamma|)]$; by i.h. $\Gamma.A' \vDash_e t' = R_{|\Gamma|+1}\, (f\, (\mathsf{Var}\, |\Gamma|)) : B$. By (CONG-ABS) $\Gamma \vDash_e \lambda t' = \lambda (R_{|\Gamma|+1}\, (f\, (\mathsf{Var}\, |\Gamma|))) : \mathsf{Fun}\, A'\, B$; and by definition of read-back $\Gamma \vDash_e \lambda t' = R_{|\Gamma|}\, (\mathsf{lam}\, f) : \mathsf{Fun}\, A'\, B$.

   b) For the second point we only need to prove $\Delta \vDash_e \mathsf{App}\, t\, r : B\, (p^i, r) \sim k \cdot d' \in [F\, d']$, for any $\Delta \leqslant^i \Gamma$ and $\Delta \vDash_e r : A'\, p^i \sim d' \in [X']$. By Lem. 69, $d' \in \mathsf{Nf}$, therefore $k \cdot d' = \mathsf{App}\, k\, d' \in \mathsf{Ne}$. We prove that by i.h. on the second part. By i.h. we know $\Delta' \vDash_e r\, p^j = R_{|\Delta'|}\, d' : A'\, p^{i+j}$, for any $\Delta' \leqslant^j \Delta$ and also, from the main hypothesis and (CONV-EQ), $\Delta' \vDash_e t\, p^{i+j} = R_{|\Delta'|}\, k : \mathsf{Fun}\, (A'\, p^{i+j})\, (B\, (p^{(i+j)+1}, q))$, therefore we can use (APP-CONG) to conclude

   $$\Delta' \vDash_e \mathsf{App}\, (t\, p^{i+j})\, (r\, p^j) = \mathsf{App}\, (R_{|\Delta'|}\, k)\, (R_{|\Delta'|}\, d') : B\, (p^{i+j}, r\, p^j)\ .\ \square$$

We now introduce logical relations between substitutions and environments modelling contexts. As in previous chapters, these relations are defined by recursion on the codomain of substitutions.

Definition 64 (Logical relation for substitutions). If $\Gamma \vDash_e$ and $\Delta \vDash_e$, we define a logical relation $\Gamma \vDash_e \_ : \Delta \sim \_ \in [\![\Delta]\!] \subseteq \{\sigma \mid \Gamma \vDash_e \sigma : \Delta\} \times \{d \mid d \in [\![\Delta]\!]\}$.

1. $\Gamma \vDash_e \sigma : \diamond \sim d \in [\![\diamond]\!]$.
2. $\Gamma \vDash_e \sigma : \Delta.A \sim (d, d') \in [\![\Delta.A]\!]$ iff $\Gamma \vDash_e p\,\sigma : \Delta \sim d \in [\![\Delta]\!]$ and $\Gamma \vDash_e q\,\sigma : A\,(p\,\sigma) \sim d' \in [[\![A]\!]d]$.

By induction on the codomain of substitutions we can prove easily preservation of the relations by judgemental equality of substitutions and weakening.

Lemma 77. Let $\Gamma \vDash_e \sigma : \Delta \sim d \in [\![\Delta]\!]$.

1. If $\Gamma \vDash_e \sigma = \sigma' : \Delta$, then $\Gamma \vDash_e \sigma' : \Delta \sim d \in [\![\Delta]\!]$.
2. If $\Theta \leqslant^i \Gamma$, then $\Theta \vDash_e \sigma\,p^i : \Delta \sim d \in [\![\Delta]\!]$.

Proof. (By induction on $\Delta \vDash_e$.)

1. The first point is trivial for $\diamond$. For $\Delta.A$ we can apply the i.h. on $\Gamma \vDash_e p\,\sigma = p\,\sigma' : \Delta$ and $\Gamma \vDash_e p\,\sigma : \Delta \sim d \in [\![\Delta]\!]$; the other part is obtained by using Lem. 74 on $\Gamma \vDash_e q\,\sigma = q\,\sigma' : A\,(p\,\delta)$ and $\Gamma \vDash_e q\,\delta : A\,(p\,\delta) \sim d' \in [[\![A]\!]d]$.
2. The second part is also trivial for $\diamond$. The second part for $\Delta.A$ is obtained by using the i.h. and Lem. 75.

□

At this point we can define an environment logically related with the identity substitution.

Definition 65 (Canonical environment). By induction on $\Gamma \vDash_e$ we define an environment ρ_Γ which models Γ and is logically related with the identity substitution id.

1. For $\diamond$ by (empty-ctx): $\rho_\diamond = \top$.
2. For $\Gamma.A$ by (ext-ctx): let $n = |\Gamma|$; then by i.h. $\rho_\Gamma \in [\![\Gamma]\!]$ and $\Gamma \vDash_e \mathsf{id} : \Gamma \sim \rho_\Gamma \in [\![\Gamma]\!]$; by both parts of Lem. 77 $\Gamma.A \vDash_e p\,\mathsf{id} : \Gamma \sim \rho_\Gamma \in [\![\Gamma]\!]$. On the other hand $\Delta \vDash_e q\,p^i = R_{|\Delta|}\,(\mathsf{Var}\,n) : A\,p^i$, for all $\Delta \leqslant^i \Gamma.A$; so, by Lem. 76, we have $\Gamma.A \vDash_e q : A\,p \sim \mathsf{Var}\,n \in [[\![A]\!]\rho_\Gamma]$ and by Lem. 74 $\Gamma.A \vDash_e q\,\mathsf{id} : A\,(p\,\mathsf{id}) \sim \mathsf{Var}\,n \in [[\![A]\!]\rho_\Gamma]$. So, we conclude $\Gamma.A \vDash_e \mathsf{id} : \Gamma.A \sim (\rho_\Gamma, \mathsf{Var}\,n) \in [\![\Gamma.A]\!]$.

Note that the formal statement of the fundamental theorem should have changed the order of the quantification between logically related substitutions and derivations.

Theorem 78 (Fundamental theorem of logical relations).

1. If $s \in S$, s is a top-sort, $\Gamma \vDash_e t : s$, and $\Delta \vDash_e \delta : \Gamma \sim d \in [\![\Gamma]\!]$, then $\Delta \vDash_e t\,\delta : s \sim [\![t]\!]d \in [s]$.
2. If $\Gamma \vDash_e t : A$ and $\Delta \vDash_e \delta : \Gamma \sim d \in [\![\Gamma]\!]$, then $\Delta \vDash_e t\,\delta : A\,\delta \sim [\![t]\!]d \in [[\![A]\!]d]$, providing A is not a top-sort.
3. If $\Gamma \vDash_e \gamma : \Theta$ and $\Delta \vDash_e \delta : \Gamma \sim d \in [\![\Gamma]\!]$, then $\Delta \vDash_e \gamma\,\delta : \Theta \sim [\![\gamma]\!]d \in [\![\Theta]\!]$.

Proof. (By induction on typing derivations.)

1. The first two points are shown by the following cases.

   a) $\Gamma \vDash_e s' : s$ by (AXIOM): we have $[\![s]\!]d = s$, since $(s, s') \in A$ we also have $s \in [s']$, and the reification of s its s itself. On the other hand we can use (SUB-SORT) to conclude $\Delta' \vDash_e (s\,\delta)\,p^i = s : s'$, for any $\Delta' \leqslant^i \Delta$.

   b) $\Gamma \vDash_e \mathsf{Fun}\,A\,B : s_3$ by (FUN-F): first we need to prove that there exists $A', B' \in \mathit{Terms}$ such that $\Delta \vDash_e \mathsf{Fun}\,A\,B\,\delta = \mathsf{Fun}\,A'\,B' : s_3$; we get this by (FUN-SUB): $A' = A\,\delta$ and $B' = B\,(\delta\,p, q)$. The second point to prove is $\Delta \vDash_e A\,\delta : s_1 \sim [\![A]\!]d \in [s_1]$, is obtained by i.h. on $\Gamma \vDash_e A : s_1$. Finally, let $\Delta' \vDash_e r : (A\,\delta)\,p^i \sim e \in [[\![A]\!]d]$; then, by definition of logical relations for substitutions, $\Delta' \vDash_e (\delta\,p^i, r) : \Gamma.A \sim (d, e) \in [\![\Gamma.A]\!]$. By i.h. on $\Gamma.A \vDash_e B : s_2$, we have $\Delta' \vDash_e B\,(\delta\,p^i, r) : s_2 \sim [\![B]\!](d, e) \in [s_2]$; by Lem. 74 we conclude $\Delta' \vDash_e B\,(\delta\,p, q)\,(p^i, r) : s_2 \sim [\![B]\!](d, e) \in [s_2]$.

   c) $\Gamma \vDash_e \lambda t : \mathsf{Fun}\,A\,B$ by (FUN-I): The point 3(c)i is easily obtained by using (ABS-SUB). The main point is proved in a similar way as we proved the second condition in the previous rule. First, we note $\mathsf{App}\,\lambda(t\,(\delta\,p, q))\,p^i\,r = t\,(\delta\,p^i, r)$ by applying successively the following rules (APP-CONG), (COMP-SUBS), (BETA), (ASS-SUBS), (DIST-SUBS), (FST-SUBS), (SND-SUBS). Now let $\Delta' \leqslant^i \Delta$ and $\Delta' \vDash_e r : (A\,\delta)\,p^i \sim e \in [[\![A]\!]d]$; so by i.h. on $\Gamma.A \vDash_e t : B$ we know $\Delta' \vDash_e t\,(\delta\,p^i, r) : B\,(p^i, r) \sim [\![t]\!](d, e) \in [([\![B]\!]d)\,e]$, where $[\![\lambda t]\!]d \cdot e = [\![t]\!](d, e)$.

   d) $\Gamma.A \vDash_e q : A\,q$ by (HYP): by inversion on the hypothesis, $\Delta \vDash_e \delta : \Gamma \sim d \in [\![\Gamma]\!]$, we know $\delta = (\delta', t)$ and $d = (d', e)$; since $q\,(\delta', t) = t$ and $[\![q]\!](d', e) = e$; the only point to prove $\Delta \vDash_e t : A\,\delta' \sim e \in [[\![A]\!]d']$ comes again from the hypothesis.

   e) $\Gamma \vDash_e t\,\sigma : A\,\sigma$ by (SUBS-TM): this is easily seen to hold by i.h. on both premises.

   f) $\Gamma \vDash_e A\,\sigma : s$ by (SUBS-S): as in the previous point by i.h.

   g) $\Gamma \vDash_e t : B$ by (CONV): we use the i.h. and Lem. 74.

2. For substitutions:

   a) $\Gamma \vDash_e \langle\rangle : \diamond$ by (EMPTY-SUBS): since $[\![\langle\rangle]\!]d = \top$ and $\top \in [\![\diamond]\!]$, then by definition of logical relations $\Delta \vDash_e \langle\rangle\,\delta : \diamond \sim \top \in [\![\diamond]\!]$.

   b) $\Gamma \vDash_e \mathsf{id} : \Gamma$ by (ID-SUBS): by using Lem. 77 on the main hypothesis and $\Gamma \vDash_e \mathsf{id}\,\delta = \delta : \Delta$.

   c) $\Gamma \vDash_e (\sigma, t) : \Theta.A$ by (EXT-SUBS): we apply the i.h. on both premises, $\Gamma \vDash_e \sigma : \Theta$ and $\Gamma \vDash_e t : A\,\sigma$, to get, respectively, $\Delta \vDash_e \sigma\,\delta : \Theta \sim [\![\sigma]\!]d \in [\![\Theta]\!]$ and $\Delta \vDash_e t\,\delta : (A\,\sigma)\,\delta \sim [\![t]\!]d \in [[\![A\,\sigma]\!]d]$. By using Lem. 77 and Lem. 74 on them we conclude $\Delta \vDash_e p\,(\sigma\,\delta, t\,\sigma) : \Theta \sim [\![\sigma]\!]d \in [\![\Theta]\!]$ and $\Delta \vDash_e q\,(\sigma\,\delta, t\,\sigma) : A\,(\sigma\,\delta) \sim [\![t]\!]d \in [[\![A\,\sigma]\!]d]$.

   d) $\Gamma \vDash_e \sigma\,\gamma : \Theta$ by (COMP-SUBS): by applying the i.h. twice: first on $\Gamma \vDash_e \gamma : \Sigma$ with the main hypothesis to get $\Delta \vDash_e \delta\,\gamma : \Sigma \sim [\![\gamma]\!]d \in [\![\Sigma]\!]$; and then with that substitution we can apply the i.h. on $\Sigma \vDash_e \sigma : \Theta$ to get $\Delta \vDash_e \sigma\,(\gamma\,\delta) : \Theta \sim [\![\sigma]\!]([\![\gamma]\!]d) \in [\![\Theta]\!]$.

   e) $\Gamma.A \vDash_e p : \Gamma$ by (FST-SUBS): by definition on logical relations $d = (e, e')$ and $\Delta \vDash_e p\,\delta : \Gamma \sim d \in [\![\Gamma]\!]$. □

Given a derivation $\Gamma \vDash_{\mathsf{e}} t : A$, we can instantiate the fundamental theorem of logical relations with the identity substitution and the canonical environment defined in Def. 65; so we obtain $\Gamma \vDash_{\mathsf{e}} t\,\mathsf{id} : A\,\mathsf{id} \sim [\![t]\!]\rho_\Gamma \in [[\![A]\!]\rho_\Gamma]$. By Lem. 76, we conclude $\Gamma \vDash_{\mathsf{e}} t\,\mathsf{id} = \mathsf{R}_{|\Gamma|}\,([\![t]\!]\rho_\Gamma) : A\,\mathsf{id}$; by using (sym), (trans), and (cong-eq) we conclude $\Gamma \vDash_{\mathsf{e}} t = \mathsf{R}_{|\Gamma|}\,([\![t]\!]\rho_\Gamma) : A$. As in previous chapters, this result leads us to the normalisation function.

Definition 66 (Normalisation function).

$$\mathbf{nbe}^{\Gamma}(t) = \mathsf{R}_{|\Gamma|}\,([\![t]\!]\rho_\Gamma)$$

Theorem 79 (Correctness of NbE). If $\Gamma \vDash_{\mathsf{e}} t : A$, then $\Gamma \vDash_{\mathsf{e}} t : A \sim [\![t]\!]\rho_\Gamma \in [[\![A]\!]\rho_\Gamma]$ and $\Gamma \vDash_{\mathsf{e}} t = \mathbf{nbe}^{\Gamma}(t) : A$.

Proof. By Thm. 78, $\Gamma \vDash_{\mathsf{e}} t\,\mathsf{id} : A\,\mathsf{id} \sim [\![t]\!]\rho_\Gamma \in [[\![A]\!]\rho_\Gamma]$ and by Lem. 74, $\Gamma \vDash_{\mathsf{e}} t : A \sim [\![t]\!]\rho_\Gamma \in [[\![A]\!]\rho_\Gamma]$ □

We deduce injectivity of Fun by inverting the equality judgement and using the fundamental theorem of logical relations and the preservation of the logical relations by judgemental equality.

Theorem 80 (Injectivity of Fun). If $\Gamma \vDash_{\mathsf{e}} \mathsf{Fun}\,A\,B = \mathsf{Fun}\,A'\,B' : C$, then $\Gamma \vDash_{\mathsf{e}} A = A' : s$ and $\Gamma.A \vDash_{\mathsf{e}} B = B' : s'$.

Proof. If $\Gamma \vDash_{\mathsf{e}} \mathsf{Fun}\,A\,B = \mathsf{Fun}\,A'\,B' : s$, by inversion of equality and Thm. 79 $\Gamma \vDash_{\mathsf{e}} \mathsf{Fun}\,A\,B : s \sim [\![\mathsf{Fun}\,A\,B]\!]\rho_\Gamma \in [[\![s]\!]\rho_\Gamma]$. By inversion of the logical relation we know $\Gamma \vDash_{\mathsf{e}} A : s_0 \sim [\![A]\!]\rho_\Gamma \in [[\![s_0]\!]\rho_\Gamma]$ and $\Gamma.A \vDash_{\mathsf{e}} B : s_1 \sim [\![B]\!]\rho_{\Gamma.A} \in [[\![s_1]\!]\rho_{\Gamma.A}]$. By Lem. 74 we also know $\Gamma \vDash_{\mathsf{e}} \mathsf{Fun}\,A'\,B' : s \sim [\![\mathsf{Fun}\,A\,B]\!]\rho_\Gamma \in [s]$, and by inversion we also know $\Gamma \vDash_{\mathsf{e}} A' : s_0 \sim [\![A]\!]\rho_\Gamma \in [s_0]$ and $\Gamma.A \vDash_{\mathsf{e}} B : s_1 \sim [\![B]\!]\rho_{\Gamma.A} \in [s_1]$. By Lem. 76 on the corresponding related elements we conclude, by rules (sym) and (trans), $\Gamma \vDash_{\mathsf{e}} A = A' : s_0$ and $\Gamma.A \vDash_{\mathsf{e}} B = B' : s_1$. □

5.5 From λ^{σ} to $\lambda^{\sigma=}$

In this last section we prove subject reduction for $\lambda^{\sigma=}$ using correctness of NbE; thus we can complete the equivalence proof between λ^{σ} and $\lambda^{\sigma=}$.

Lemma 81.

1. If $\Gamma \vDash_{\mathsf{e}} t : A$ and $t \equiv_{\beta x} t'$, then $\Gamma \vDash_{\mathsf{e}} t : A \sim [\![t']\!]\rho_\Gamma \in [[\![A]\!]\rho_\Gamma]$.
2. If $\Gamma \vDash_{\mathsf{e}} \sigma : \Delta$ and $\sigma \equiv_{\beta x} \sigma'$, then $\Gamma \vDash_{\mathsf{e}} \sigma : \Delta \sim [\![\sigma']\!]\rho_\Gamma \in [\![\Delta]\!]$.

Proof. By Thm. 79 on the first hypothesis $\Gamma \vDash_{\mathsf{e}} t : A \sim [\![t]\!]\rho_\Gamma \in [[\![A]\!]\rho_\Gamma]$ and by Lem. 71, $[\![t]\!]\rho_\Gamma = [\![t']\!]\rho_\Gamma$, therefore $\Gamma \vDash_{\mathsf{e}} t : A \sim [\![t']\!]\rho_\Gamma \in [[\![A]\!]\rho_\Gamma]$. □

Adams obtained, as a result of his proof method for the equivalence of PTSs, subject reduction for PTSs with typed equality: *If* $\Gamma \vDash_{\mathsf{e}} t : A$ *and* $t \equiv_{\beta x} t'$, *then* $\Gamma \vDash_{\mathsf{e}} t = t' : A$. We can prove a weaker version of subject reduction which is enough to complete the equivalence proof. Note that we equate t with the normal form of t', instead of t' itself.

Corollary 82 (Subject reduction for $\lambda^{\sigma=}$). If $\Gamma \vDash_{\mathsf{e}} t : A$ and $t \equiv_{\beta x} t'$, then $\Gamma \vDash_{\mathsf{e}} t = \mathbf{nbe}_\Gamma(t') : A$.

Finally we can prove the second part of Thm. 67; the formal proof of the next theorem assumes as a postulate the last corollary. Since that corollary depends on the results of the previous two sections, the next theorem only applies to predicative PTSs.

Theorem 83.

1. If $\Gamma \vdash$, then $\Gamma \vDash_{\mathsf{e}}$.
2. If $\Gamma \vdash t : A$, then $\Gamma \vDash_{\mathsf{e}} t : A$.
3. If $\Gamma \vdash \sigma : \Delta$, then $\Gamma \vDash_{\mathsf{e}} \sigma : \Delta$.

Proof. See `PtsToEpts.agda`. We explain the case for (CONV): the premises are $\Gamma \vdash t : A$, $\Gamma \vdash B : s$ and $A \equiv_{\beta x} B$. By i.h. on the first premise, we know $\Gamma \vDash_{\mathsf{e}} t : A$, and by i.h. on the second premise, $\Gamma \vDash_{\mathsf{e}} B : s$. By type validity on $\Gamma \vDash_{\mathsf{e}} t : A$, there is a sort s', such that $\Gamma \vDash_{\mathsf{e}} A : s'$. By Thm. 79 $\Gamma \vDash_{\mathsf{e}} A = \mathbf{nbe}(A) : s'$; therefore by (CONV), $\Gamma \vDash_{\mathsf{e}} t : \mathbf{nbe}(A)$. From $\Gamma \vDash_{\mathsf{e}} B : s$ and $A \equiv_{\beta x} B$, by Cor. 82 we know $\Gamma \vDash_{\mathsf{e}} B = \mathbf{nbe}_{\Gamma}(A) : s$. Now we can apply (CONV) again to get $\Gamma \vDash_{\mathsf{e}} t : B$. □

Corollary 84. If $\Gamma \vdash t : A$, $\Gamma \vdash t' : A$, and $t \equiv_{\beta x} t'$, then $\Gamma \vDash_{\mathsf{e}} t = t' : A$.

Proof. By Thm. 83 we have $\Gamma \vDash_{\mathsf{e}} t : A$ and $\Gamma \vDash_{\mathsf{e}} t' : A$. By Cor. 82, we know $\Gamma \vDash_{\mathsf{e}} t = \mathbf{nbe}_{\Gamma}(t') : A$; and, by Thm. 79 $\Gamma \vDash_{\mathsf{e}} t' = \mathbf{nbe}_{\Gamma}(t') : A$. We conclude by (SYM) and (TRANS). □

Conclusion

6

In this last chapter I would like first to show how NbE can be used to add axioms like commutativity for some operators. In the second section I mention some further considerations that have not been addressed in the thesis and possible future work.

6.1 Commutativity in Martin-Löf type theory

Introduction

In Chap. 4 we have used NbE to define a type-checker for MLTT extended with singleton types and also proof-irrelevant types. In this section, we adapt the NbE algorithm for a type axiomatising an abelian monoid. It is interesting to consider this type because, as we explained in the introduction 1.3, one cannot normalise terms by orienting the equality axioms.

From a practical point of view adding commutativity can alleviate the burden in the formalisation of proofs. We can illustrate this point by considering the following basic lemma of abelian monoids.

Lemma 85. Let $M = \langle M, +, 0\rangle$ be an (additive) abelian monoid. Let $a, b, x \in M$ be such that $a + x = 0$ and $b + x = 0$, then $a = b$.

Proof. $a = a + 0 = a + (b + x) = a + (x + b) = (a + x) + b = 0 + b = b$ □

To state and prove that lemma in type theory we first need to postulate a type representing (intensional) equality with one constructor for reflexivity and also transitivity. We then postulate the type for the monoid and the addition operation, the null element, and the properties of an abelian monoid (associativity, commutativity, and congruence of addition with respect to intensional equality). Then we can write the same equational reasoning as in the informal proof. In contrast, if we have the monoid and its axioms internalised in the system, we only need to postulate intensional equality (with transitivity) and congruence of addition. But instead of the whole proof we can just write: $a = a + (b + x) = b$. All the other steps are dealt by the normalisation algorithm.

In the next sections we consider the extension of the core of the type theory of Sec. 4.1, that is a universe of small types and dependent products. We skip the rules for the type-checking algorithm for the new type as they introduce no difficulty; the contribution of this section is the decision procedure for equality.

Syntax

We add a type for an abelian monoid with no other constant than the null element.

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathsf{M}}\ (\textsc{mon-type}) \qquad \frac{\Gamma \vdash}{\Gamma \vdash \mathsf{M} : \mathsf{U}}\ (\textsc{mon-u-i}) \qquad \frac{\Gamma \vdash}{\Gamma \vdash 0 : \mathsf{M}}\ (\textsc{mon-ztm}) \qquad \frac{\Gamma \vdash t : \mathsf{M} \qquad \Gamma \vdash t' : \mathsf{M}}{\Gamma \vdash t + t' : \mathsf{M}}\ (\textsc{mon-plus})$$

Axioms The axioms on the left column are those corresponding to the resolution of substitution; in the right column we list the equational theory for the abelian monoid.

$$\begin{aligned} \mathsf{M}\,\gamma &= \mathsf{M} & \qquad t + 0 &= t \\ 0\,\gamma &= 0 & \qquad s + t &= t + s \\ (t + t')\,\gamma &= (t\,\gamma) + (t'\,\gamma) & \qquad (r + s) + t &= r + (s + t) \end{aligned}$$

Remark 25 (Inversion).

1. if $\Gamma \vdash 0 : A$, then $\Gamma \vdash A = \mathsf{M}$; and
2. if $\Gamma \vdash s + t : A$, then $\Gamma \vdash A = \mathsf{M}$, $\Gamma \vdash s : \mathsf{M}$, and $\Gamma \vdash t : \mathsf{M}$.

Remember that our decision procedure for equality consisted on normalising terms and comparing syntactically equivalence of normal forms. It is clear that we need to extend the set of normal forms of Chap. 4 to include the new constructors concerning the monoid. To deal with associativity, sums are right-leaning; this accounts for the addition of a new non-terminal to the grammar.

Definition 67 (Normal forms).

$$\begin{aligned} \mathrm{Lhs} \ni l &::= q \mid q\,p^{i+1} \mid \mathsf{App}\ l\ v \\ \mathrm{Ne} \ni k &::= l \mid l + k \\ \mathrm{Nf} \ni v, V, W &::= k \mid \mathsf{U} \mid \mathsf{M} \mid 0 \mid \lambda v \mid \mathsf{Fun}\ V\ W \end{aligned}$$

Note that two normal forms generated by this grammar can be syntactically different, although judgementally equal. For example, $q\,p + q$ and $q + q\,p$, say in the context $\Gamma = \mathsf{M}.\mathsf{M}$. Instead of complicating the definition of normal forms, we will compare for syntactical equivalence terms which are in *canonical form*. This canonical forms will be based on an order between constructors.

Definition 68 (Order between normal forms). We define a total order on the terminals of the grammar: $q < q\,p^1 < \ldots < q\,p^i < q\,p^{i+1} < \ldots < \mathsf{App} \ < + < \mathsf{U} < \mathsf{M} < 0 < \lambda < \mathsf{Fun}$.

The order on normal forms is the induced lexicographic order using the order on terminals. It is clear that a sum in normal form has the following shape: $l_0 + (l_1 + (\ldots + l_n)\ldots)$. Such a sum is said to be in canonical form if lighter terms are to the left: $i \leqslant j$, implies $l_i \leqslant l_j$. In general, for $v \in \mathrm{Nf}$, v is in canonical form if every sum occurring in v is in canonical form.

The following functions rearrange operands in a sum, by flattening them and ordering them by the order between terms.

Definition 69. Let $\vec{k}$ be a tuple of neutral terms. Let $\_ \triangleright \_$ be the consing operation for tuples: $l \triangleright \langle k_0, \ldots, k_{n-1} \rangle = \langle l, k_0, \ldots, k_{n-1} \rangle$.

Merging of tuples

$$\langle\rangle * \vec{k} = \vec{k}$$
$$\vec{k} * \langle\rangle = \vec{k}$$
$$\langle l_1, \ldots, l_n\rangle * \langle l'_1, \ldots, l'_m\rangle = \begin{cases} l_1 \triangleright (\langle l_2, \ldots, l_n\rangle * \langle l'_1, l'_2, \ldots, l'_m\rangle) & \text{if } l_1 \leqslant l'_1 \\ l'_1 \triangleright (\langle l_1, l_2, \ldots, l_n\rangle * \langle l'_2, \ldots, l'_m\rangle) & \text{otherwise} \end{cases}$$

Flattening of neutrals

$$\mathit{flat}\ k = \begin{cases} \langle l\rangle & \text{if } k \equiv l \\ \langle l\rangle * \mathit{flat}\ k' & \text{if } k \equiv l + k' \end{cases}$$

Un-flattening

$$\mathit{unflat}\ \vec{l} = \begin{cases} l_0 & \text{if } \vec{l} = \langle l_0\rangle \\ l_0 + \mathit{unflat}\ \langle l_1, \ldots, l_n\rangle & \text{if } \vec{l} = \langle l_0, l_1, \ldots, l_n\rangle \end{cases}$$

To turn a normal form in canonical normal form v, we simply canonicalise all the sums occurring in v.

Definition 70 (Canonical transformation).

$$\begin{aligned} \mathrm{can}(q) &= q \\ \mathrm{can}(q\, p^i) &= q\, p^i \\ \mathrm{can}(\mathsf{App}\ l\ v) &= \mathsf{App}\ \mathrm{can}(l)\ \mathrm{can}(v) \\ \mathrm{can}(l + k) &= \mathit{unflat}\ (\langle \mathrm{can}(l)\rangle * \mathit{flat}\ (\mathrm{can}(k))) \\ \mathrm{can}(\mathsf{U}) &= \mathsf{U} \\ \mathrm{can}(\mathsf{M}) &= \mathsf{M} \\ \mathrm{can}(\lambda v) &= \lambda\, \mathrm{can}(v) \\ \mathrm{can}(\mathsf{Fun}\ V\ W) &= \mathsf{Fun}\ \mathrm{can}(V)\ \mathrm{can}(W) \end{aligned}$$

Remark 26. Let $\Gamma \vdash v : A$. Since the only change we do with $\mathrm{can}(\_)$ is rearranging terms in sums, it is clear that $\Gamma \vdash v = \mathrm{can}(v) : A$ (and the same for types). From this we conclude that if $\Gamma \vdash v : A$, $\Gamma \vdash v' : A$, and $\mathrm{can}(v) \equiv \mathrm{can}(v')$, then $\Gamma \vdash v = v' : A$.

NbE: correctness and completeness

We extend the semantic domain to interpret the new terms: $D = \ldots \oplus \{M\}_\perp \oplus \{0\}_\perp \oplus D^*_\perp$; an element $d \in D^*$ is a finite tuple written $\langle d_0, \ldots, d_{n-1}\rangle \in D$. Accordingly we add new clauses for the readback function:

$$\mathsf{R}_i\, M = M \qquad \mathsf{R}_i\, 0 = 0$$
$$\mathsf{R}_i\, \langle d_0, \ldots, d_n\rangle = \begin{cases} \mathsf{R}_i\, d_0 & \text{if } n = 0 \\ \mathsf{R}_i\, d_0 + \mathsf{R}_i\, \langle d_1, \ldots, d_n\rangle & \text{otherwise} \end{cases}$$

Note that we keep the readback function simple and the generated sums are right-leaning (if the tuple is flat). We need to adapt the definition of the PERs for semantical neutral values and normal forms, to take into account the symmetry of addition.

Definition 71 (PER for neutrals, and normal forms).

- $d = d' \in \mathsf{Ne}$ iff $\mathsf{R}_i\ d$ and $\mathsf{R}_i\ d'$ are defined, and $\mathrm{can}(\mathsf{R}_i\ d) \equiv_{\mathrm{Ne}} \mathrm{can}(\mathsf{R}_i\ d')$, for all $i \in \mathbb{N}$;
- $d = d' \in \mathsf{Nf}$ iff $\mathsf{R}_i\ d$ and $\mathsf{R}_i\ d'$ are defined, and $\mathrm{can}(\mathsf{R}_i\ d) \equiv_{\mathrm{Nf}} \mathrm{can}(\mathsf{R}_i\ d')$, for all $i \in \mathbb{N}$.

We add a new PER over D to interpret the type M; accordingly we also add a new clause to the universes U.

Definition 72 (Adapting the PER model).

PER for the monoid

$$\mathsf{Ne} \subseteq M \qquad \text{and} \qquad 0 = 0 \in M\ .$$

New clause for U

$$\mathsf{M} = \mathsf{M} \in U \qquad \text{and} \qquad [\mathsf{M}] = M\ .$$

η-expansion and normalisation

$$\uparrow_{\mathsf{M}} k = k \qquad \downarrow_{\mathsf{M}} d = d \qquad \Downarrow \mathsf{M} = \mathsf{M}\ .$$

Remark 27. It is easy to see that M is saturated; therefore Lem. 38 is valid in the extended model.

To interpret the addition operation, we build tuples but removing occurrences of 0. Note that if the two arguments are flat tuples, the result is also a flat tuple.

Definition 73.

$$\begin{gathered} 0 \dot{+} d = d \quad d \dot{+} 0 = d \\ \langle d_0, \ldots, d_{n-1} \rangle \dot{+} \langle d'_0, \ldots, d'_{m-1} \rangle = \langle d_0, \ldots, d_{n-1}, d'_0, \ldots, d'_{m-1} \rangle \\ \langle d_0, \ldots, d_{n-1} \rangle \dot{+} d' = \langle d_0, \ldots, d_{n-1}, d' \rangle \\ d \dot{+} \langle d'_0, \ldots, d'_{m-1} \rangle = \langle d, d'_0, \ldots, d'_{m-1} \rangle \\ d \dot{+} d' = \langle d, d' \rangle \end{gathered}$$

Definition 74 (Semantic equations).

$$[\![\mathsf{M}]\!] d = \mathsf{M} \quad [\![0]\!] d = 0 \quad [\![t + t']\!] d = [\![t]\!] d \dot{+} [\![t']\!] d$$

The following lemma is the key to prove soundness.

Lemma 86. The operator $\_ \dot{+} \_$ is associative and commutative with respect to the PER M. Let $c = c', d = d', e = e' \in M$,

1. $0 \dot{+} d = d' \in M$ and $d \dot{+} 0 = d' \in M$.
2. $d \dot{+} e = e' \dot{+} d' \in M$,
3. $c \dot{+} (d \dot{+} e) = (c' \dot{+} d') \dot{+} e' \in M$, and

Proof.

1. By definition.
2. The only interesting case is when both $d = d' \in \mathsf{Ne}$ and $e = e' \in \mathsf{Ne}$; note also that it is enough to consider flat tuples: $d = \langle d_0, \ldots, d_n \rangle$, $d' = \langle d'_0, \ldots, d'_n \rangle$, $e = \langle e_0, \ldots, e_m \rangle$, and $e' = \langle e'_0, \ldots, e'_m \rangle$. In that case for any $i \in \mathbb{N}$ and every element k of any of the tuples, $\mathsf{R}_i\, k$ is defined and is a neutral element. Therefore $\mathrm{can}(\mathsf{R}_i\, d_0 + (\mathsf{R}_i\, d_1 + (\ldots(\mathsf{R}_i\, e_{m-1} + \mathsf{R}_i\, e_m)\ldots)))$ has the same summands than $\mathrm{can}(\mathsf{R}_i\, e'_0 + (\mathsf{R}_i\, e'_1 + (\ldots(\mathsf{R}_i\, d'_{n-1} + \mathsf{R}_i\, d'_n)\ldots)))$ and in the same order.
3. By an argument analogous to that of the previous point.

□

Remark 28. The model is sound, cf. Thm. 34. This is easy after proving Lem. 86. By Rem. 27, if $\Gamma \vdash t = t' \colon \mathsf{M}$, then $\mathrm{can}((\mathsf{R}_i\, (\llbracket t \rrbracket d))) \equiv \mathrm{can}((\mathsf{R}_i\, (\llbracket t' \rrbracket d)))$, for any $d \in \llbracket \Gamma \rrbracket$ and $i \in \mathbb{N}$.

Logical relations The definition of logical relation for the monoid is analogous to the definition of logical relations for data types in Def.47.

Definition 75 (Logical relation for monoid).

1. $\Gamma \vdash A \sim \mathsf{M}$, if and only if $\Gamma \vdash A = \mathsf{M}$.
2. $\Gamma \vdash t \colon A \sim d \in [\mathsf{M}]$, if and only if $\Gamma \vdash A \sim \mathsf{M}$, and for all $\Delta \leqslant^i \Gamma$, $\Delta \vdash t\, \mathsf{p}^i = \mathsf{R}_{|\Delta|}\, d \colon A\, \mathsf{p}^i$.

All the technical lemmata corresponding to logical relations are easily proved for the new clauses. We recall only the most important results. The first one is that if a syntactical term is related with an element on some PER, then the term is judgementally equal to the reification of the semantical value. Note that for the monoid this lemma is trivial to prove.

Lemma 87 (Derivability of equality of reified elements). Let $\Gamma \vdash A \sim X \in T$, and $\Gamma \vdash t \colon A \sim d \in [X]$, then $\Gamma \vdash t = \mathsf{R}_{|\Gamma|}\, (\downarrow_A d) \colon A$.

The following lemma is useful to prove the fundamental theorem.

Lemma 88. If $\Gamma \vdash t \colon A \sim d \in M$ and $\Gamma \vdash t' \colon A \sim d' \in M$, then $\Gamma \vdash t + t' \colon A \sim d \dot{+} d' \in M$.

Proof. By induction on $d, d' \in M$. If $d = 0$, then we have $\Gamma \vdash t + t' = t' \colon A$ and $d \dot{+} d' = d'$; therefore by Lem. 42 $\Gamma \vdash t + t' \colon A \sim d \dot{+} d' \in M$. The case for $d' = 0$ is analogous. If $d, d' \in \mathsf{Ne}$, then by Lem. 87 $\Delta \vdash t\, \mathsf{p}^i = \mathsf{R}_{|\Delta|}\, d \colon A\, \mathsf{p}^i$ and $\Delta \vdash t'\, \mathsf{p}^i = \mathsf{R}_{|\Delta|}\, d' \colon A\, \mathsf{p}^i$; therefore, by first using (conv) to type both equalities with M, then using congruence of addition, and finally (conv) again to type the additions with the original type $A\, \mathsf{p}^i$, we get $\Delta \vdash (t + t')\, \mathsf{p}^i = \mathsf{R}_{|\Delta|}\, (d \dot{+} d') \colon A\, \mathsf{p}^i$. That is the condition for $\Gamma \vdash t + t' \colon A \sim d \dot{+} d' \in M$. □

Remember that the fundamental theorem of logical relations states that every term is logically related with its denotation. We recast it here only for the monoid.

Theorem 89. Let $\Gamma \vdash \delta \colon \Delta \sim d \in \llbracket \Delta \rrbracket$, then if $\Delta \vdash t \colon \mathsf{M}$, then $\Gamma \vdash t\, \delta \colon \mathsf{M}\, \delta \sim \llbracket t \rrbracket d \in M$.

Proof. This theorem is proved by induction on derivations; we show the case for $\Delta \vdash t + t' : M$, when the last rule used was (plus-tm). Let $\Gamma \vdash \delta : \Delta \sim d \in [\![\Delta]\!]$. By i.h. we know $\Gamma \vdash t\,\delta : M\,\delta \sim [\![t]\!]d \in M$ and $\Gamma \vdash t'\,\delta : M\,\delta \sim [\![t']\!]d \in M$. By Lem. 88, $\Gamma \vdash t\,\delta + t'\,\delta : M\,\delta \sim [\![t]\!]d \dot{+} [\![t']\!]d \in M$. By definition of the semantics of addition and by Lem. 42 we have $\Gamma \vdash (t + t')\,\delta : M\,\delta \sim [\![t + t']\!]d \in M$. □

Note that as a corollary of Lem. 87 and Thm. 89 with the canonical environment introduced in Def. 49, we have correctness of normalisation.

Corollary 90. If $\Gamma \vdash t : A$, then $\Gamma \vdash t = R_{|\Gamma|}\,(\downarrow_{[\![A]\!]\rho_\Gamma}\,([\![t]\!]\rho_\Gamma)) : A$.

We change the definition of the normalisation function in order to get canonical normal forms; the definition of the canonical environment remains as in Def. 49.

Definition 76. Let $\Gamma \vdash A$ and $\Gamma \vdash t : A$.

$$\mathbf{nbe}_\Gamma(A) = \mathrm{can}(R_{|\Gamma|}\,(\Downarrow [\![A]\!]\rho_\Gamma))$$
$$\mathbf{nbe}^A_\Gamma(t) = \mathrm{can}(R_{|\Gamma|}\,(\downarrow_{[\![A]\!]\rho_\Gamma}\,[\![t]\!]\rho_\Gamma))$$

The normalised term can still be proved equal to its canonical normal form by virtue of Rem. 26.

Corollary 91 (Correctness of NbE). If $\Gamma \vdash t : A$, then $\Gamma \vdash t = \mathbf{nbe}^A_\Gamma(t) : A$.

6.2 Further work

Correctness of Haskell implementation of NbE The correctness of the implementation of the type-checking algorithm for Martin-Löf type theory presented in Chap. 4 depends on having a proof showing that the Haskell implementation of NbE corresponds to the mathematical function for which we proved completeness and soundness. For that reason an implementation of a type-checker using the method suggested in Chap. 4 would be more trustworthy if one has a proof of the computational correctness of the Haskell program. I know of two papers that can be used to guide such a proof: Filinski and Rhode [55] proved a similar result for untyped NbE and its implementation in an ML-like language; Dybjer and Kuperberg [53] also proved the correctness of a Haskell implementation of NbE for untyped combinatory logic.

PTS with explicit substitutions It would be interesting to compare our presentation of PTS with explicit substitutions to PTS with those of Bloo and Muñoz. Moreover, it is needed to study more deeply the meta-theory of λ^σ and its relationship with traditional PTS, that is with named variables and implicit substitutions.

We expect that one can use analogous semantical methods as those that we used to prove injectivity of Fun to get an equivalence result for λ^σ and $\lambda^{\sigma=}$ with (η) rule.

More abstract definition of NbE There is some smell of repetition among the chapters of this thesis; this call for a deeper study to see if it is possible to define NbE more abstractly and prove the main properties (completeness and soundness) without repeating over and over the same proofs each time a rule

is added (or dropped). For example, Thierry Coquand suggested to consider NbE λ^{Π} without rule (cong-abs); instead of redoing all the work from scratch, it would be more interesting to have an abstract definition and extract the proof for that system by using the appropiate parameters in the general framework. One possibility for getting such an abstract result is to understand NbE in a categorical setting.

Using NbE to ease the use of dependent types In Sec. 6.1 we discussed how we can use NbE to decide equality for an abelian monoid and presented an example of how it can simplify the burden for formalising a (fairly trivial) proof. It will be interesting to have a proof-checker, even a prototype, with several extensions like commutativity. Ideally one should formalise some mathematical proof in such a system and in some other proof-assistant in order to compare the *de Bruijn factor* [117] between both formalisations.

Bibliography

[1] Martin Abadi, Luca Cardelli, Pierre-Louis Curien and Jean Jacques Lévy. 'Explicit substitutions'. In: *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. San Francisco, California, United States: ACM, 1990, pp. 31–46. ISBN: 0-89791-343-4. DOI: 10.1145/96709.96712 (cit. on pp. 11, 17).

[2] Andreas Abel, Klaus Aehlig and Peter Dybjer. 'Normalization by Evaluation for Martin-Löf Type Theory with One Universe'. In: *Proceedings of the 23rd Conference on the Mathematical Foundations of Programming Semantics (MFPS XXIII), New Orleans, LA, USA, 11-14 April 2007*. Ed. by Marcelo Fiore. Elsevier, 2007, pp. 17–39 (cit. on pp. 13, 14).

[3] Andreas Abel, Thierry Coquand and Peter Dybjer. 'Normalization by Evaluation for Martin-Löf Type Theory with Typed Equality Judgements'. In: *Proc. of the 22nd IEEE Symp. on Logic in Computer Science (LICS 2007)*. IEEE Computer Soc. Press, 2007, pp. 3–12. DOI: 10.1109/LICS.2007.33 (cit. on pp. 13, 14).

[4] Andreas Abel, Thierry Coquand and Peter Dybjer. 'On the Algebraic Foundation of Proof Assistants for Intuitionistic Type Theory'. In: *Proc. of the 9th Int. Symp. on Functional and Logic Programming, FLOPS 2008*. Ed. by Jacques Garrigue and Manuel V. Hermenegildo. Vol. 4989. Lect. Notes in Comput. Sci. Springer, 2008, pp. 3–13. ISBN: 978-3-540-78968-0. DOI: 10.1007/978-3-540-78969-7_2 (cit. on pp. 13, 88).

[5] Andreas Abel, Thierry Coquand and Peter Dybjer. 'Verifying a Semantic βη-Conversion Test for Martin-Löf Type Theory'. In: *Proc. of the 9th Int. Conf. on Mathematics of Program Construction, MPC 2008*. Ed. by Philippe Audebaud and Christine Paulin-Mohring. Vol. 5133. Lect. Notes in Comput. Sci. Springer, 2008, pp. 29–56. ISBN: 978-3-540-70593-2. DOI: 10.1007/978-3-540-70594-9_4 (cit. on pp. 13, 14).

[6] Andreas Abel, Thierry Coquand and Miguel Pagano. 'A Modular Type-Checking Algorithm for Type Theory with Singleton Types and Proof Irrelevance'. In: *TLCA*. Ed. by Pierre-Louis Curien. Vol. 5608. Lecture Notes in Computer Science. Springer, 2009, pp. 5–19. ISBN: 978-3-642-02272-2. DOI: 10.1007/978-3-642-02273-9_3 (cit. on p. 14).

[7] Andreas Abel, Thierry Coquand and Miguel Pagano. 'A Modular Type-Checking Algorithm for Type Theory with Singleton Types and Proof Irrelevance'. In: *Logical Methods in Computer Science* 7.2:4 (May 2011), pp. 1–57. DOI: 10.2168/LMCS-7(2:4)2011 (cit. on p. 14).

[8] Samson Abramsky and Achim Jung. 'Domain Theory'. In: *Handbook of Logic in Computer Science*. Oxford University Press, 1994, pp. 1–168 (cit. on p. 23).

[9] Robin Adams. 'Pure type systems with judgemental equality'. In: *Journal of Functional Programming* 16.2 (2006), pp. 219–246. DOI: 10.1017/S0956796805005770 (cit. on pp. 14, 109).

[10] Klaus Aehlig, Florian Haftmann and Tobias Nipkow. 'A Compiled Implementation of Normalization by Evaluation'. In: *TPHOLs*. Ed. by Otmane Aït Mohamed, César Muñoz and Sofiène Tahar. Vol. 5170. Lecture Notes in Computer Science. Springer, 2008, pp. 39–54. ISBN: 978-3-540-71065-3. DOI: 10.1007/978-3-540-71067-7_8 (cit. on p. 13).

[11] Klaus Aehlig and Felix Joachimski. 'Operational aspects of untyped Normalisation by Evaluation'. In: *Mathematical Structures in Computer Science* 14.4 (2004), pp. 587–611. DOI: 10.1017/S096012950400427X (cit. on pp. 13, 27).

[12] Thorsten Altenkirch, Peter Dybjer, Martin Hofmann and Philip J. Scott. 'Normalization by Evaluation for Typed Lambda Calculus with Coproducts'. In: *Proc. of the 16th IEEE Symp. on Logic in Computer Science (LICS 2001)*. IEEE Computer Soc. Press, 2001, pp. 303–310 (cit. on pp. 13, 61).

[13] Thorsten Altenkirch, Martin Hofmann and Thomas Streicher. 'Categorical Reconstruction of a Reduction Free Normalization Proof'. In: *Category Theory and Computer Science*. Ed. by David H. Pitt, David E. Rydeheard and Peter Johnstone. Vol. 953. Lecture Notes in Computer Science. Springer, 1995, pp. 182–199. ISBN: 3-540-60164-3 (cit. on p. 13).

[14] David Aspinall. 'Subtyping with Singleton Types'. In: *Computer Science Logic, 8th Int. Wksh., CSL '94*. Ed. by Leszek Pacholski and Jerzy Tiuryn. Vol. 933. Lect. Notes in Comput. Sci. Springer, 1995, pp. 1–15. ISBN: 3-540-60017-5. DOI: 10.1007/BFb0022243 (cit. on pp. 28, 55, 65, 89).

[15] Steven Awodey and Andrej Bauer. 'Propositions as [Types]'. In: *J. Log. Comput.* 14.4 (2004), pp. 447–471. DOI: 10.1093/logcom/14.4.447 (cit. on pp. 61, 62).

[16] Vincent Balat, Roberto Di Cosmo and Marcelo P. Fiore. 'Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums'. In: *Proc. of the 31st ACM Symp. on Principles of Programming Languages, POPL 2004*. Ed. by Neil D. Jones and Xavier Leroy. ACM Press, 2004, pp. 64–76. ISBN: 1-58113-729-X. DOI: 10.1145/964001.964007 (cit. on p. 61).

[17] Henk Barendregt. 'Lambda calculi with types'. In: *Handbook of Logic in Computer Science*. Ed. by Samson Abramsky, D. M. Gabbay and T. S. E. Maibaum. Oxford University Press, 1992, pp. 117–309 (cit. on p. 3).

[18] Bruno Barras. *Coq en Coq*. Research Report RR-3026. Projet COQ. INRIA, 1996. URL: http://hal.inria.fr/inria-00073667/PDF/RR-3026.pdf (cit. on p. 13).

[19] Bruno Barras. 'Auto-validation d'un système de preuves avec familles inductives'. Thèse de Doctorat. Université Paris 7, Nov. 1999 (cit. on p. 13).

[20] Gilles Barthe and Morten Heine Sørensen. 'Domain-free pure type systems'. In: *J. Funct. Program.* 10.5 (2000), pp. 417–452. URL: http://journals.cambridge.org/action/displayAbstract?aid=59753 (cit. on p. 101).

[21] Michael Beeson. 'Formalizing constructive mathematics: Why and how?' In: *Constructive Mathematics*. Ed. by Fred Richman. Vol. 873. Lecture Notes in Mathematics. Springer Berlin / Heidelberg, 1981, pp. 146–190. DOI: 10.1007/BFb0090733 (cit. on p. 1).

[22] Michael J. Beeson. 'Problematic principles in constructive mathematics'. In: *Logic Colloquium '80*. Vol. 108. Studies in Logic and the Foundations of Mathematics. Amsterdam: North-Holland, 1980, pp. 11–55. DOI: 10.1016/S0049-237X(09)70502-6 (cit. on p. 4).

[23] Stefano Berardi. 'Type Dependence and Constructive Mathematics'. PhD thesis. Carnegie Mellon University and Torino University, 1989. URL: http://www.di.unito.it/~stefano/Berardi-PhDThesis.rtf (cit. on p. 3).

[24] Stefano Berardi. 'About the sets-as-propositions embedding of HOL in CC'. Feb. 2004 (cit. on p. 55).

[25] Ulrich Berger, Matthias Eberl and Helmut Schwichtenberg. 'Normalisation by Evaluation'. In: *Prospects for Hardware Foundations*. Ed. by Bernhard Möller and J. V. Tucker. Vol. 1546. Lecture Notes in Computer Science. Springer, 1998, pp. 117–137. ISBN: 3-540-65461-5. DOI: 10.1007/3-540-49254-2_4 (cit. on p. 13).

[26] Ulrich Berger and Helmut Schwichtenberg. 'An Inverse of the Evaluation Functional for Typed lambda-calculus'. In: *Proceedings, Sixth Annual IEEE Symposium on Logic in Computer Science, 15-18 July, 1991, Amsterdam, The Netherlands*. IEEE Computer Society, 1991, pp. 203–211. DOI: 10.1109/LICS.1991.151645 (cit. on pp. 12, 13, 27, 71).

[27] Roel Bloo. 'Pure type systems with explicit substitution'. In: *Mathematical. Structures in Comp. Sci.* 11.1 (2001), pp. 3–19. ISSN: 0960-1295. DOI: 10.1017/S096012950000325X (cit. on pp. 101, 102).

[28] Ana Bove and Peter Dybjer. 'Dependent Types at Work'. In: *LerNet ALFA Summer School*. Ed. by Ana Bove, Luís Soares Barbosa, Alberto Pardo and Jorge Sousa Pinto. Vol. 5520. Lecture Notes in Computer Science. Springer, 2008, pp. 57–99. ISBN: 978-3-642-03152-6. DOI: 10.1007/978-3-642-03153-3_2 (cit. on p. 6).

[29] Kim Bruce and John C. Mitchell. 'PER models of subtyping, recursive types and higher-order polymorphism'. In: *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. Albuquerque, New Mexico, United States: ACM, 1992, pp. 316–327. ISBN: 0-89791-453-8. DOI: 10.1145/143165.143230 (cit. on p. 65).

[30] Nicolaas Govert de Bruijn. *AUTOMATH, a language for mathematics*. T.H. Department of Mathematics, Eindhoven University of Technology, Nov. 1968 (cit. on p. 3).

[31] Nicolaas Govert de Bruijn. 'Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem'. In: *Indagationes Mathematicae (Proceedings)* 75.5 (1972), pp. 381 –392. ISSN: 1385-7258. DOI: 10.1016/1385-7258(72)90034-0 (cit. on p. 11).

[32] Nicolaas Govert de Bruijn. *Some extensions of Automath : the AUT-4 family*. Tech. rep. OAI Repository of the Technische Universiteit Eindhoven (TU/e) (Netherlands), 1994. URL: http://library.tue.nl/csp/dare/LinkToRepository.csp?recordnumber=597613 (cit. on p. 55).

[33] Felice Cardone and J. Roger Hindley. 'Lambda-Calculus and Combinators in the 20th Century'. In: *Logic from Russell to Church*. Ed. by Dov M. Gabbay and John Woods. Vol. 5. Handbook of the History of Logic. North-Holland, 2009, pp. 723 –817. DOI: 10.1016/S1874-5857(09)70018-4 (cit. on p. 1).

[34] John Cartmell. 'Generalised algebraic theories and contextual categories'. In: *Annals of Pure and Applied Logic* (1986), pp. 32–209. DOI: 10.1016/0168-0072(86)90053-9 (cit. on p. 17).

[35] Alonzo Church. 'A set of postulates for the foundation of logic'. In: *The Annals of Mathematics* 33.2 (Apr. 1932), pp. 346–366. DOI: 10.2307/2371045 (cit. on pp. 1, 2).

[36] Alonzo Church. 'A Formulation of the Simple Theory of Types'. In: *J. Symb. Logic* 5.2 (June 1940), pp. 56–68 (cit. on pp. 1, 2).

[37] Thierry Coquand. 'An Algorithm for Type-Checking Dependent Types'. In: *Proc. of the 3rd Int. Conf. on Mathematics of Program Construction, MPC '95*. Vol. 26. Sci. Comput. Program. 1–3. Elsevier, May 1996, pp. 167–177 (cit. on pp. 13, 88, 90).

[38] Thierry Coquand. 'Type Theory'. In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Spring 2010. 2010 (cit. on p. 1).

[39] Thierry Coquand and Peter Dybjer. 'Intuitionistic model constructions and normalization proofs'. In: *Mathematical. Structures in Comp. Sci.* 7.1 (1997), pp. 75–94. ISSN: 0960-1295. DOI: 10.1017/S0960129596002150 (cit. on p. 13).

[40] Thierry Coquand, Peter Dybjer, Erik Palmgren and Anton Setzer. *Type-theoretic Foundations of Constructive Mathematics*. Tech. rep. Types Summer School, 2005 (cit. on p. 4).

[41] Thierry Coquand and Gerard Huet. 'The calculus of constructions'. In: *Inf. Comput.* 76.2-3 (1988), pp. 95–120. ISSN: 0890-5401. DOI: 10.1016/0890-5401(88)90005-3 (cit. on p. 3).

[42] Thierry Coquand, Randy Pollack and Makoto Takeyama. 'A Logical Framework with Dependently Typed Records'. In: *Fundamenta Informaticae* 65.1-2 (2005), pp. 113–134 (cit. on pp. 28, 55, 65).

[43] Judicaël Courant. 'Strong Normalization with Singleton Types'. In: *Intersection Types and Related Systems (ITRS 2002)*. Vol. 70. Electr. Notes in Theor. Comp. Sci. 1. Elsevier, 2002 (cit. on p. 60).

[44] Guy Cousineau, Pierre-Louis Curien and Michel Mauny. 'The categorical abstract machine'. In: *Science of Computer Programming* 8.2 (1987), pp. 173 –202. ISSN: 0167-6423. DOI: 10.1016/0167-6423(87)90020-7 (cit. on p. 18).

[45] Guy Cousineau, Pierre-Louis Curien and Bernard Robinet, eds. *Combinators and Functional Programming Languages, Thirteenth Spring School of the LITP, Val d'Ajol, France, May 6-10, 1985, Proceedings*. Vol. 242. Lecture Notes in Computer Science. Springer, 1986. ISBN: 3-540-17184-3 (cit. on p. 17).

[46] Karl Crary. 'A Syntactic Account of Singleton Types via Hereditary Substitution'. In: *4th Int. Wksh. on Logical Frameworks and Meta-languages: Theory and Practice (LFMTP 2009)*. Ed. by James Cheney and Amy Felty. ACM Press, 2009, pp. 21–29. ISBN: 978-1-60558-529-1. DOI: 10.1145/1577824.1577829 (cit. on p. 60).

[47] Pierre-Louis Curien. *Categorical combinators, sequential algorithms, and functional programming*. Progress in theoretical computer science. Birkhäuser, 1993. ISBN: 9780817636548 (cit. on p. 17).

[48] H. B. Curry. 'Functionality in Combinatory Logic'. In: *Proceedings of the National Academy of Sciences, U.S.A.* 20 (1934), pp. 584–590 (cit. on p. 2).

[49] Haskell B. Curry. 'Modified basic functionality in combinatory logic'. In: *Dialectica* 23.2 (1969), pp. 83–92. ISSN: 1746-8361. DOI: 10.1111/j.1746-8361.1969.tb01183.x (cit. on p. 2).

[50] Haskell B. Curry and Robert Feys. *Combinatory logic*. North-Holland publishing Company, 1958 (cit. on p. 2).

[51] Peter Dybjer. 'Internal Type Theory'. In: *TYPES '95: Selected papers from the International Workshop on Types for Proofs and Programs*. London, UK: Springer-Verlag, 1996, pp. 120–134. ISBN: 3-540-61780-9 (cit. on pp. 39, 40).

[52] Peter Dybjer. 'A General Formulation of Simultaneous Inductive-Recursive Definitions in Type Theory'. In: *J. Symb. Logic* 65.2 (2000), pp. 525–549 (cit. on pp. 44, 69).

[53] Peter Dybjer and Denis Kuperberg. 'Formal neighbourhoods, combinatory Böhm trees, and untyped normalization by evaluation'. In: *Ann. Pure Appl. Logic* 163.2 (2012), pp. 122–131. DOI: 10.1016/j.apal.2011.06.021 (cit. on p. 126).

[54] Andrzej Filinski. 'Normalization by Evaluation for the Computational Lambda-Calculus'. In: *TLCA*. 2001, pp. 151–165. DOI: 10.1007/3-540-45413-6_15 (cit. on pp. 13, 27).

[55] Andrzej Filinski and Henning Korsholm Rohde. 'A Denotational Account of Untyped Normalization by Evaluation'. In: *FoSSaCS*. Ed. by Igor Walukiewicz. Vol. 2987. Lecture Notes in Computer Science. Springer, 2004, pp. 167–181. ISBN: 3-540-21298-1. DOI: 10.1007/978-3-540-24727-2_13 (cit. on pp. 26, 126).

[56] Marcelo Fiore. 'Semantic analysis of normalisation by evaluation for typed lambda calculus'. In: *PPDP '02: Proceedings of the 4th ACM SIGPLAN international conference on Principles and practice of declarative programming*. Pittsburgh, PA, USA: ACM, 2002, pp. 26–37. ISBN: 1-58113-528-9. DOI: 10.1145/571157.571161 (cit. on p. 13).

[57] Daniel Fridlender and Miguel Pagano. *PTS with Typed Equality and Explicit Substitutions*. Presentation at Types for Proofs and Programs 2011. Sept. 2011 (cit. on p. 15).

[58] François Garillot and Benjamin Werner. 'Simple Types in Type Theory: Deep and Shallow Encodings'. In: *Theorem Proving in Higher Order Logics, TPHOLs 2007*. Ed. by Klaus Schneider and Jens Brandt. Vol. 4732. Lect. Notes in Comput. Sci. Springer, 2007, pp. 368–382. ISBN: 978-3-540-74590-7. DOI: 10.1007/978-3-540-74591-4_27 (cit. on p. 26).

[59] Herman Geuvers. 'Logics and Type Systems'. PhD thesis. Katholieke Universiteit Nijmegen, Sept. 1993 (cit. on pp. 14, 109).

[60] Herman Geuvers. 'Proof assistants: History, ideas and future'. English. In: *Sadhana* 34 (1 2009), pp. 3–25. ISSN: 0256-2499. DOI: 10.1007/s12046-009-0001-5 (cit. on p. 6).

[61] Jean-Yves Girard. 'Une extension de l'interpretation de Godel a l'analyse, et son application a l'elimination des coupures dans l'analyse et la theorie des types'. In: *Second Scandinavian Logic Symposium*. Ed. by Jens Erik Fenstad. Studies in Logic and the Foundations of Mathematics 63. North-Holland, 1971, pp. 63–92 (cit. on p. 3).

[62] Benjamin Grégoire and Xavier Leroy. 'A compiled implementation of strong reduction'. In: *Proc. of the 7th ACM SIGPLAN Int. Conf. on Functional Programming (ICFP '02)*. Vol. 37. SIGPLAN Notices 9. ACM Press, Sept. 2002, pp. 235–246. ISBN: 1-58113-487-8. DOI: 10.1145/581478.581501 (cit. on p. 23).

[63] Robert Harper, Furio Honsell and Gordon Plotkin. 'A framework for defining logics'. In: *Journal ofACM Press* 40.1 (1993), pp. 143–184. ISSN: 0004-5411. DOI: 10.1145/138027.138060 (cit. on p. 3).

[64] Robert Harper and Frank Pfenning. 'On equivalence and canonical forms in the LF type theory'. In: *ACM Trans. Comput. Logic* 6.1 (2005), pp. 61–101. ISSN: 1529-3785. DOI: 10.1145/1042038.1042041 (cit. on p. 13).

[65] John Harrison. 'HOL Light: An Overview'. In: *TPHOLs*. Ed. by Stefan Berghofer, Tobias Nipkow, Christian Urban and Makarius Wenzel. Vol. 5674. Lecture Notes in Computer Science. Springer, 2009, pp. 60–66. ISBN: 978-3-642-03358-2. DOI: 10.1007/978-3-642-03359-9_4 (cit. on p. 2).

[66] Leon Henkin. 'Completeness in the Theory of Types'. In: *The Journal of Symbolic Logic* 15 (1950), pp. 81–91. ISSN: 00224812. URL: http://www.jstor.org/stable/2266967 (cit. on p. 12).

[67] Arend Heyting. *Intuitionism, an Introduction*. 3rd ed. Amsterdam: North-Holland, 1971. ISBN: 0-7204-2239-6 (cit. on p. 4).

[68] Roger J. Hindley. 'The Principal Type-Scheme of an Object in Combinatory Logic'. In: *Transactions of the American Mathematical Society* 146 (1969), pp. 29–60. ISSN: 00029947. URL: http://www.jstor.org/stable/1995158 (cit. on p. 2).

[69] William Howard. 'The formulae-as-types notion of construction'. In: *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. Ed. by Haskell Curry, Jonathan Seldin and Roger Hindley. Academic Press, 1980, pp. 479–490. ISBN: 0123490502 (cit. on pp. 3, 4).

[70] Gérard P. Huet. 'Cartesian closed Categories and Lambda- calculus'. In: *Combinators and Functional Programming Languages*. 1985, pp. 123–135. DOI: 10.1007/3-540-17184-3_43 (cit. on p. 25).

[71] Fairouz Kamareddine, Twan Laan and Rob Nederpelt. *A Modern Perspective on Type Theory From its Origins Until Today*. Vol. 29. Applied Logic Series. Kluwer Academic Publishers, May 2005. ISBN: 1402023340 (cit. on p. 1).

[72] Delia Kesner. 'A Theory of Explicit Substitutions with Safe and Full Composition'. In: *Logical Methods in Computer Science* 5.3 (2009). DOI: 10.2168/LMCS-5(3:1)2009 (cit. on p. 11).

[73] Stephen C. Kleene and John Barkley Rosser. 'The Inconsistency of Certain Formal Logics'. In: *The Annals of Mathematics* 36.3 (July 1935), pp. 630–636 (cit. on p. 2).

[74] J. Lambek and P. J. Scott. *Introduction to higher order categorical logic*. New York, NY, USA: Cambridge University Press, 1988. ISBN: 0-521-35653-9 (cit. on pp. 4, 25).

[75] Daniel K. Lee, Karl Crary and Robert Harper. 'Towards a Mechanized Metatheory of Standard ML'. In: *Proc. of the 34th ACM Symp. on Principles of Programming Languages, POPL 2007*. Ed. by Martin Hofmann and Matthias Felleisen. ACM Press, 2007, pp. 173–184. ISBN: 1-59593-575-4. DOI: 10.1145/1190216.1190245 (cit. on p. 55).

[76] Andreas Löh, Conor McBride and Wouter Swierstra. 'A Tutorial Implementation of a Dependently Typed Lambda Calculus'. In: *Fundamenta Informaticae* 102.2 (2010), pp. 177–207. DOI: 10.3233/FI-2010-304 (cit. on p. 90).

[77] Zhaohui Luo. 'An Extended Calculus of Constructions'. PhD thesis. Department of Computer Science, University of Edinburgh, June 1990. URL: http://www.cs.rhul.ac.uk/~zhaohui/THESIS90.ps (cit. on p. 13).

[78] Zhaohui Luo. *Computation and reasoning: a type theory for computer science*. International series of monographs on computer science. Clarendon Press, 1994. ISBN: 9780198538356 (cit. on pp. 30, 44).

[79] Odalric-Ambrym Maillard. *Proof-irrelevance, strong-normalisation in Type-Theory and PER*. Tech. rep. Chalmers Institute of Technology, 2006 (cit. on p. 61).

[80] Per Martin-Löf. 'An Intuitonistic Theory of Types: Predicative Part'. English. In: ed. by H. E. Rose and J. C. Shepherdson. North-Holland Pub. Co., 1975, pp. 73–118. ISBN: 0444106421 (cit. on pp. 13, 50).

[81] Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984 (cit. on pp. 4, 59).

[82] Per Martin-Löf. 'On the Meanings of the Logical Constants and the Justification of the Logical Laws'. In: *Nordic Journal of Philosophical Logic* 1.1 (1996), pp. 11–60 (cit. on p. 5).

[83] Per Martin-Löf. 'An intuitionistic theory of types'. In: Oxford logic guides. reprinted version of an unpublished report from 1972. Clarendon Press, 1998. ISBN: 9780198501275 (cit. on pp. 1, 3, 39).

[84] The Agda Team. *The Agda Wiki*. URL: http://wiki.portal.chalmers.se/agda/ (cit. on p. 3).

[85] The Coq development team. *The Coq proof assistant reference manual*. Version 8.0. 2004. URL: http://coq.inria.fr (cit. on p. 3).

[86] Robin Milner. 'A theory of type polymorphism in programming'. In: *Journal of Computer and System Sciences* 17.3 (1978), pp. 348–375. DOI: 10.1016/0022-0000(78)90014-4 (cit. on p. 3).

[87] Alexandre Miquel and Benjamin Werner. 'The Not So Simple Proof-Irrelevant Model of CC'. In: *TYPES*. Ed. by Herman Geuvers and Freek Wiedijk. Vol. 2646. Lecture Notes in Computer Science. Springer, 2002, pp. 240–258. ISBN: 3-540-14031-X (cit. on p. 101).

[88] John C. Mitchell. 'A type-inference approach to reduction properties and semantics of polymorphic expressions (summary)'. In: *LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming*. Cambridge, Massachusetts, United States: ACM, 1986, pp. 308–319. ISBN: 0-89791-200-4. DOI: 10.1145/319838.319872 (cit. on p. 28).

[89] John C. Mitchell. 'Type Systems for Programming Languages'. In: *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*. 1990, pp. 365–458 (cit. on p. 65).

[90] John C. Mitchell. *Foundations of programming languages*. Cambridge, MA, USA: MIT Press, 1996. ISBN: 0-262-13321-0 (cit. on p. 31).

[91] John C. Mitchell and Eugenio Moggi. 'Kripke-style models for typed lambda calculus'. In: *Annals of Pure and Applied Logic* 51.1-2 (1991), pp. 99 –124. ISSN: 0168-0072. DOI: 10.1016/0168-0072(91)90067-V (cit. on p. 31).

[92] César Muñoz. *Dependent types and explicit substitutions*. Tech. rep. Institute for Computer Applications in Science and Engineering (ICASE), 1999 (cit. on pp. 101, 102).

[93] Tobias Nipkow, Lawrence C. Paulson and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Vol. 2283. LNCS. Springer, 2002 (cit. on p. 2).

[94] Bengt Nordström, Kent Petersson and Jan M. Smith. *Programming in Martin Löf's Type Theory: An Introduction*. Clarendon Press, Oxford, 1990 (cit. on p. 59).

[95] Nicolas Oury and Wouter Swierstra. 'The power of Pi'. In: *ICFP*. Ed. by James Hook and Peter Thiemann. ACM, 2008, pp. 39–50. ISBN: 978-1-59593-919-7 (cit. on p. 1).

[96] Frank Pfenning. 'On a Logical Foundation for Explicit Substitutions'. In: *TLCA*. Ed. by Simona Ronchi Della Rocca. Vol. 4583. Lecture Notes in Computer Science. Slides available at http://www.cs.cmu.edu/~fp/talks/rdp07-talk.pdf. Springer, 2007, p. 1. ISBN: 978-3-540-73227-3. DOI: 10.1007/978-3-540-73228-0_1 (cit. on p. 11).

[97] Randy Pollack. 'The Theory of LEGO'. PhD thesis. Department of Computer Science, University of Edinburgh, 1994. URL: http://homepages.inf.ed.ac.uk/rpollack/export/thesis.ps.gz (cit. on p. 13).

[98] John C. Reynolds. 'Towards a Theory of Type Structure'. In: *Lecture Notes in Computer Science*. Vol. 19. New York: Springer-Verlag, 1974, pp. 408–425 (cit. on p. 3).

[99] John C. Reynolds. 'What do types mean?: from intrinsic to extrinsic semantics'. In: New York, NY, USA: Springer-Verlag New York, Inc., 2003, pp. 309–327. ISBN: 0-387-95349-3 (cit. on p. 27).

[100] Giovanni Sambin and Jan M. Smith. *Twenty-five years of constructive type theory: proceedings of a congress held in Venice, October 1995*. Oxford logic guides. Clarendon Press, 1998. ISBN: 9780198501275.

[101] Giovanni Sambin and Silvio Valentini. 'Building up a toolbox for Martin-Löf's type theory: subset theory'. In: Oxford logic guides. Clarendon Press, 1998, pp. 221–244. ISBN: 9780198501275 (cit. on p. 55).

[102] Dana Scott. *Continuous Lattices*. Tech. rep. Oxford University, 1971 (cit. on p. 23).

[103] Natarajan Shankar and Sam Owre. 'Principles and Pragmatics of Subtyping in PVS'. In: *WADT '99: Selected papers from the 14th International Workshop on Recent Trends in Algebraic Development Techniques*. London, UK: Springer-Verlag, 2000, pp. 37–52. ISBN: 3-540-67898-0 (cit. on p. 55).

[104] Vincent Siles. 'Investigation on the typing of equality in type systems'. PhD thesis. École Polytechnique, Nov. 2010. URL: http://www.cse.chalmers.se/~siles/papers/thesis.pdf (cit. on pp. 14, 15, 101, 109).

[105] Vincent Siles and Hugo Herbelin. 'Equality Is Typable in Semi-full Pure Type Systems'. In: *LICS*. IEEE Computer Soc. Press, 2010, pp. 21–30. ISBN: 978-0-7695-4114-3. DOI: 10.1109/LICS.2010.19 (cit. on pp. 14, 109).

[106] Michael B. Smyth and Gordon D. Plotkin. 'The Category-Theoretic Solution of Recursive Domain Equations'. In: *SIAM Journal on Computing* 11.4 (1982), pp. 761–783. DOI: 10.1137/0211062 (cit. on p. 23).

[107] Matthieu Sozeau. 'Subset Coercions in Coq'. In: *Types for Proofs and Programs, Int. Wksh., TYPES 2006*. Ed. by Thorsten Altenkirch and Conor McBride. Vol. 4502. Lect. Notes in Comput. Sci. Springer, 2007, pp. 237–252. ISBN: 978-3-540-74463-4. DOI: 10.1007/978-3-540-74464-1_16 (cit. on p. 55).

[108] Christopher A. Stone and Robert Harper. 'Extensional equivalence and singleton types'. In: *ACM Trans. Comput. Logic* 7.4 (2006), pp. 676–722. ISSN: 1529-3785. DOI: 10.1145/1183278.1183281 (cit. on p. 13).

[109] Christopher A. Stone and Robert Harper. 'Extensional Equivalence and Singleton Types'. In: *ACM Trans. Comput. Logic* 7.4 (2006), pp. 676–722. ISSN: 1529-3785. DOI: 10.1145/1166109.1166112 (cit. on p. 55).

[110] Thomas Streicher. 'Correctness and Completeness of a Categorical Semantics of the Calculus of Constructions'. PhD thesis. Universität Passau, Passau, West Germany, June 1989 (cit. on p. 101).

[111] Morten Heine Sørensen and Paweł Urzyczyin. *Lectures on the Curry-Howard Isomorphism*. Vol. 149. Studies in Logic and the Foundations of Mathematics. Elsevier, 2006 (cit. on p. 1).

[112] Alvaro Tasistro. *Formulation of Martin-Löf's theory of types with explicit substitutions*. Published in [113]. 1993 (cit. on p. 11).

[113] Alvaro Tasistro. 'Substitution, Record Types and Subtyping in Type Theory, with Applications to the Theory of Programming'. PhD thesis. Göteborg University and Chalmers Institute of Technology, 1997 (cit. on p. 138).

[114] Jan Terlouw. *Een nadere bewijstheoretische analyse van GSTTs*. Tech. rep. University of Nijmegen, 1989 (cit. on p. 3).

[115] Anne S. Troelstra. *History of Constructivism in the Twentieth Century*. ITLI Prepublication Series ML-91-05. University of Amsterdam, 1991 (cit. on p. 4).

[116] Benjamin Werner. 'On the strength of proof-irrelevant type theories'. In: *Logical Methods in Computer Science* 4 (2008) (cit. on p. 55).

[117] Freek Wiedijk. 'The De Bruijn Factor'. In: *TPHOLs 2000: Supplemental Proceedings*. Ed. by M. Aargaard, J. Harrison and T. Schubert. OGI Technical Report CSE 00-009. Oregon Graduate Institute, Portland, USA. July 2000, pp. 213–230. URL: http://www.cs.ru.nl/~freek/factor/factor.pdf (cit. on p. 127).

[118] Freek Wiedijk, ed. *The Seventeen Provers of the World, Foreword by Dana S. Scott*. Vol. 3600. Lecture Notes in Computer Science. Springer, 2006. ISBN: 3-540-30704-4 (cit. on p. 6).

[119] Freek Wiedijk. 'Formal Proof – Getting Started'. In: *Notices of the American Mathematical Society* 55 (2008), pp. 1408–1414 (cit. on p. 6).

This thesis was typeset in pdfLaTeX.

The design is based on the memoir class, with some personal modifications.

The body text is set on the typeface “Johannes Kepler”, designed by Christophe Caignaert.

Mathematical content is set on “Euler”, by Hermann Zapf, and “Computer Modern”, by Donald Knuth, as provided by the euler-vm package.

Programs are listed on “Inconsolata” by Raph Levien.

Rules and derivations are typeset using Paul Taylor’s prooftree and Didier Rémy’s mathpartir.