Habilitationsschrift

Normalization by Evaluation

Dependent Types and Impredicativity

Andreas Abel

Institut für Informatik

Ludwig-Maximilians-Universität München

Abstract

Normalization by evaluation (NbE) is a technique to compute the normal form of a lambda-term, i. e., an expression of a pure functional programming language. While evaluation is only concerned with computing closed expressions, normalization also applies to function bodies, thus, needs to compute with open expressions containing free variables. NbE reduces normalization to evaluation of expressions in a residualizing model, i. e., a computational structure that has extra base values which are unknowns or computations blocked by unknowns.

Normalization by evaluation, while not under this name, has been used by Martin-Löf [1975] to prove normalization and decidability of type checking for his predicative intuitionistic type theory with a weak notion of term equality that is not closed under function abstraction. Independently, normalization by evaluation has been discoverd by Berger and Schwichtenberg [1991] as a tool to implement a normalizer for simply-typed lambda-calculus with a strong (extensional, "η") notion of term equality. Normalization invokes the evaluator of the host programming language, leading to a concise and efficient normalization procedure.

In this thesis, we join the two strands of research by developing NbE for dependent type theory with extensional term equality that does consider normalization of function bodies. Further, we extend NbE to impredicative type systems such as System F, F^ω, and the Calculus of Constructions. As NbE is a tool to connect syntax (terms) to semantics (values), the study of NbE enables us to semantically prove meta-theoretical properties of the considered type systems such as decidability of term equality, which entails the decidability of type checking in the presence of dependent types. For dependent types with universes and extensional (η) equality at the level of types, this is a novel result.

The study of NbE provides us with new insights into the nature of η-expansion which we apply to dependent type theory with singleton types. Further, we formulate the concept of *candidate space* to equip semantic types with extra structure that allows us to show the correctness of NbE. Finally, we introduce typed Kripke structures to unify semantic arguments by logical relations.

The results underlying this thesis were obtained partly in collaboration with Klaus Aehlig, Thierry Coquand, Peter Dybjer, and Miguel Pagano, as far as the design of the NbE algorithm and its application to predicative type theory are concerned.

Publications Included in this Thesis

1. A. Abel, K. Aehlig, and P. Dybjer. Normalization by evaluation for Martin-Löf type theory with one universe. In M. Fiore, editor, *Proceedings of the 23rd Conference on the Mathematical Foundations of Programming Semantics (MFPS XXIII), New Orleans, LA, USA, 11-14 April 2007*, volume 173 of *Electronic Notes in Theoretical Computer Science*, pages 17–39. Elsevier, 2007a

2. A. Abel, T. Coquand, and P. Dybjer. Normalization by evaluation for Martin-Löf Type Theory with typed equality judgements. In *22nd IEEE Symposium on Logic in Computer Science (LICS 2007), 10-12 July 2007, Wroclaw, Poland, Proceedings*, pages 3–12. IEEE Computer Society Press, 2007b

3. A. Abel, T. Coquand, and M. Pagano. A modular type-checking algorithm for type theory with singleton types and proof irrelevance. *Logical Methods in Computer Science*, 7(2:4):1–57, May 2011

4. A. Abel. Weak $\beta\eta$-normalization and normalization by evaluation for System F. In I. Cervesato, H. Veith, and A. Voronkov, editors, *15th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning, LPAR 2008, 22-27 November 2008, Doha, Qatar, Proceedings*, volume 5330 of *Lecture Notes in Artificial Intelligence*, pages 497–511. Springer-Verlag, 2008

5. A. Abel. Typed applicative structures and normalization by evaluation for System Fω. In E. Grädel and R. Kahle, editors, *Computer Science Logic, 23rd international Workshop, CSL 2009, 18th Annual Conference of the EACSL, Coimbra, Portugal, September 7-11, 2009. Proceedings*, volume 5771 of *Lecture Notes in Computer Science*, pages 40–54. Springer-Verlag, 2009a. ISBN 978-3-642-04026-9

6. A. Abel. Towards Normalization by Evaluation for the $\beta\eta$-Calculus of Constructions. In M. Blume, N. Kobayashi, and G. Vidal, editors, *Functional and Logic Programming, 10th International Symposium, FLOPS 2010, Sendai, Japan, April 19-21, 2010. Proceedings*, volume 6009 of *Lecture Notes in Computer Science*, pages 224–239. Springer-Verlag, 2010a. ISBN 978-3-642-12250-7

My contribution to papers 1 and 2 amounts to roughly 70%, to paper 3 roughly 35%.

Contents

1 Introduction **1**
1.1 How to Read this Thesis . . . 2
1.2 Acknowledgments . . . 4

2 Simple Types: From Evaluation to Normalization **5**
2.1 Evaluation . . . 5
2.2 Normalization . . . 8
2.3 Normalization by Evaluation . . . 10
2.4 Variable Handling . . . 12
2.5 Liftable Terms . . . 13
2.6 Soundness of Normalization by Evaluation . . . 15
2.7 Summary . . . 16

3 Untyped Normalization-By-Evaluation and Type Assignment **17**
3.1 Untyped NbE Using Domains . . . 18
3.2 Untyped NbE Using Partial Applicative Structures . . . 20
3.3 Type-Assignment System T . . . 22
3.4 Candidate Spaces and Normalization for Type-Assignment System T . . 25
3.5 Explicit Substitutions and β-Equality . . . 25
3.6 Extensionality and Partial Equivalence Relations . . . 27
3.7 Typed Candidate Spaces and Completeness of NbE . . . 31
3.8 Restoring Curried Constants . . . 34
3.9 Kripke Logical Relations and Soundness of NbE . . . 34
3.10 Summary . . . 37

4 Dependent Types **39**
4.1 A Full Dependently-Typed Language . . . 40
4.2 Type Values, Reflection and Reification . . . 43
4.3 Dependent Function Space and Universes . . . 44
4.4 A PER Model . . . 45
4.5 Dependently-Typed Candidate Spaces and Completeness of NbE . . . 47
4.6 Dependent Function Space on Groupoids . . . 48
4.7 Kripke Logical Relations for Dependent Types and Soundness of NbE . 51
4.8 Summary . . . 53

5 Impredicativity **55**
5.1 System F Syntax . . . 57
5.2 System F Type Semantics via Candidate Space . . . 58
5.3 Abstract Evaluation and the Fundamental Lemma for System F . . . 62

5.4 Normalization by Evaluation for System F 62
5.5 System F^ω, the Calculus of Constructions, and Beyond 65

6 Summary, Related Work, and Perspectives 67
6.1 Intrinsic Typing and NbE 68
6.2 On NbE for the Extensional Treatment of Finite Choice 70
6.3 Applications of NbE 71
6.4 Future Perspectives on NbE 71

Index of Notations 73

Bibliography 79

1 Introduction

Type systems have emerged as a paradigm to structure modern programming languages, but long before, they have been introduced into logical theories such as Church's simple theory of types that serve as foundations for mathematics. A strong correspondence between propositions and types, and between proofs and programs coined as *Curry-Howard-isomorphism* intimately connects logics and programming languages and inspires research and design of new programming languages and logics. Martin-Löf introduced the Curry-Howard counterpart of the quantifiers of predicate logic into functional programming and is considered the pioneer of dependent type theory, a unified language for programming and developing constructive mathematics. In dependent type theory, propositions are identified with types, and proofs with programs, and checking a proof for validity amounts to checking that a program has a certain type.

The distinctive feature of dependent type theory is that types can be defined by computation on values. Expressions that compute the same value have to be considered equal, and, in dependent type theory, two types that compute the same type value are considered equal and thus, classify the same programs. Algorithms for type-checking have to take type equality into account, and decidability of type-checking rests on the decidability of equality. When Martin-Löf presented his *Intuitionistic Theory of Types* (ITT) [1975] he accompanied it by a proof that typing is decidable. The decidability relies on *normalization* of all well-typed expressions of ITT, which means that all well-typed expressions reduce to a value, called the normal form of the expression, which is uniquely determined by the expression. Martin-Löf demonstrates normalization by a semantic argument in a constructive meta-language. Proofs in constructive meta-languages implicitly describe algorithms, and the algorithm underlying Martin-Löf's argument has been later coined *normalization by evaluation* (NbE).

Independently, when implementing the proof system Minlog in the functional language Scheme, Berger and Schwichtenberg [1991] discovered an algorithm for normalization of simply-typed lambda-calculus that is summarized by the slogan *normalization by evaluation*. A typed lambda-term is first embedded as program in the host language Scheme and then evaluated. Then, the evaluated Scheme program is quoted, arriving at a lambda-term in normal form. The quotation is described as *inverse of the evaluation functional*. A detailed description of the NbE algorithms as composition of evaluation and quotation is given in Chapter 2.

As NbE does not rely on the native evaluation and quotation features of Scheme, but can be adapted to any host programming language, it soon found wide-spread interest in the programming language research community. Its attraction comes from the elegant idea to reuse evaluation of the host language (originally Scheme) to implement evaluation for the object language (originally the simply-typed lambda calculus). For one, it was discovered that NbE corresponds to type-directed partial evaluation [Danvy, 1996]. For two, NbE presents an efficient normalization algorithm and even serves as

a framework for normalization by compilation [Grégoire and Leroy, 2002]. Finally, due to its semantic character, it proved a nice tool to structure the reasoning about meta-theoretical properties of type systems such as normalization of well-typed expressions and decidability of type equality [Martin-Löf, 1975].

This thesis continues the investigation of NbE as method to establish the decidability of dependent type theory. Martin-Löf [1975] considered typing up to a notion of type equality that is quite weak when relating functions. In essence, it is too *intensional* as it does not equate functions whose bodies are equal w. r. t. computation, their bodies must be literally identical. In contrast, modern implementations of dependent type theories, such as the successful proof assistant Coq [INRIA, 2012] and the dependently typed programming language Agda [Norell, 2007], rely on a more expressive, *extensional* equality for functions called η-equality. Two functions are extensionally equal if their applications to a fresh variable are already extensionally equal. Herein, a any value of function type is considered a function, thus the notion of equality is *typed*.

Using NbE, the author and the coauthors of the papers underlying this thesis have been able to prove the decidability of type checking in ITT with typed (η-)equality. But the formulation and verification of NbE for dependently-typed systems is interesting in itself, and the present work is the first description of NbE for a full-blown dependently typed language [Abel et al., 2007a,b]. During the study of NbE for dependent types we were also able to clarify η-expansion for singleton types [Abel et al., 2011]. The results developed in the cited paper may serve as theoretical foundation for the language Agda which is based on predicative universes.

The Coq system is based on the Calculus of Inductive Constructions (CIC) which adds an impredicative universe of propositions to ITT. To study the meta-theory of impredicative type theories we have adapted NbE to the impredicative System F [Abel, 2008], F^ω [Abel, 2009a] and the dependently-typed Calculus of Constructions (CoC) [Abel, 2010a]. These systems are subsystems of the CIC, and understanding of impredicativity in theses systems paves the way for the full CIC. While we have not fully proven the decidability of CoC, we were the first to prove termination and completeness of NbE for a system with both impredicativity and dependent types.

1.1 How to Read this Thesis

This thesis introduces the reader to NbE and semantic techniques to reason about type systems and the correctness of NbE. It develops the necessary tools to study NbE for dependent and impredicative type theories, summarizing the main ideas of the publications underlying this thesis. We have tried to be gentle in the exposition, focusing on the explanation of concepts rather than the proof details. For those, the reader is referred to the original publications. The remainder of this thesis is structured as follows:

Chapter 2 presents normalization by evaluation in a simple instance, a toy programming language with natural numbers and higher-order functions, known as System T (Section 2.1). On the way, we explain what we mean by definitional term equality, normal form, soundness and completeness of normalization, reflection of syntax into semantics, and reification of semantics into syntax (Sections 2.2 and 2.3). We give an

overview over the different solutions to the problem of generating fresh variable names during reification (Section 2.4) and pick one of them (Section 2.5). The introductory part closes with proof of the correctness of NbE by a Kripke logical relation connecting syntactic terms with semantic objects (Section 2.6).

Chapter 3 introduces untyped NbE (Section 3.1), generalizes it to partial applicative structures (Section 3.2) and then revisits System T from a type-assignment perspective. The purpose of this chapter of the thesis is to introduce—in a gentle setting— most of the technical tools we need for the analysis of NbE for dependent types. In particular, we apply untyped evaluation to System T and explain how it models System T's typing relation (Section 3.3). We introduce the concept of candidate spaces to extend our reasoning about evaluation reasoning about NbE (Section 3.4). Explicit substitutions (Section 3.5) and partial equivalences are introduced to model definitional equality (Section 3.6). Then we add type-directed reflection and reification on the value level to untyped NbE to obtain complete normalization for η-equality (Section 3.7). Finally, we generalize the Kripke logical relation for the soundness proof of NbE to arbitrary models of typed lambda-calculus we call type structures (Section 3.9).

Chapter 4 summarizes Abel et al. [2007a,b, 2011] and applies type-assignment NbE as developed in Chapter 3 to dependent types. First we enrich System T with dependent types and a hierarchy of predicative universes that allows us to compute types in dependency on values (Section 4.1), arriving at language PTT. In the adaption of NbE to dependent types, special attention is paid to reflection and reification which are now directed by type-values (Section 4.2). An inductive definition of valid types, which is the foundation of reasoning about PTT, is given in Section 4.3. Refining this inductive definition by an equivalence relation on valid types we arrive at a model of PTT that interprets types and universes as partial equivalence relations (PERs) on values (Section 4.4). An extension of our concept of candidate spaces allows us to reason about the completeness of NbE (Section 4.5). A hiherto unpublished view on PERs as subsets of groupoids appears in Section 4.6. Soundness of NbE is finally established by the dependently-typed Kripke logical relation of Section 4.7. The chapter concludes with a summary of the results and a short discussion on singleton types (Section 4.8).

Chapter 5 applies NbE to impredicative type systems, presenting Abel [2008] in terms of ideas from Abel [2009a] and outlining the results of Abel [2010a]. The exposition focuses on System F, a typed lambda-calculus with impredicative polymorphism (Section 5.1). A general notion of model for System F is developed that can be instantiated to prove soundness and completeness of NbE, but also subsumes traditional normalization proofs (Sections 5.2 and 5.3). Nbe for System F and its correctness are then explained in Section 5.4. Section 5.5 concludes the chapter with a discussion on the combination of impredicativity with higher kinds and dependent types.

Chapter 6 summarizes the contributions of this thesis. We discuss alternative, intrinsically typed approaches to NbE (Section 6.1) and NbE algorithms for extensional treatment of disjunction (Section 6.2). A review of applications of NbE (Section 6.3) demonstrates that NbE has secured its standing as efficient implementation of normalization in proof assistants and may enjoy a bright future (Section 6.4).

1.2 Acknowledgments

The author received valuable feedback from Andrew Pitts, Theo Winterhalter, and Francisco Ferreira which helped to correct the content and improve the presentation of this thesis.

2 Simple Types: From Evaluation to Normalization

2.1 Evaluation

Evaluation is the process of running a program and obtaining a result in the end, the *value*. This typically involves the compilation of the program into a machine language and the execution of the machine code on the machine. There are numerous machine architectures and machine languages, both machines implemented in hardware and machines implemented in software. We shall not be concerned with details of machines and compilation, but shall treat evaluation on a more abstract level. We shall specify the evaluation of a small programming language (our *object* language) in terms of a language that already features advanced concepts like functions and application (the *host* or *meta* language), which allows us to express evaluation succinctly. Such a host language could be a mathematical language such as set theory or type theory, or an already implemented programming language.

Evaluation lets us assign *meaning* to a program. If we type a program into an interpreter and it prints back its value, for instance a number, then the meaning of this program is that number. The meaning of a program could also be a function, not a concrete value such as a number; in this case, the typical interpreter will not print a result that adequately describes the meaning of the program—typically it may just tell us that the result is a function and as such, unprintable. We will see how such an interpreter could be modified to print back code that represents the function, but let us not get ahead of us. In the following, we will describe the meaning of a program by interpretation in mathematical language, which already has a meaningful concept of function.

We consider a functional programming language, Gödel's *System T* [Gödel, 1958], with natural numbers, primitive recursion,[1] and functions of higher order. The natural numbers are presented in unary form, introduced by constructors zero and suc. Primitive recursion $\mathsf{rec} : \mathsf{Rec}_T$ with $\mathsf{Rec}_T = T \to (\mathsf{N} \to T \to T) \to \mathsf{N} \to T$allows us to define values of type T by recursion over a natural number. Application of function r to argument s is written as juxtaposition $r\,s$. New functions $\lambda x t$ are constructed by abstracting variable x in term t, the body of the function. Each term t comes with its type T in a typing contexts Γ which assigns types to the variables that occur free in t;

[1] Recursion in System T is actually higher-order primitive recursion which has the *computational power* ε_0 of Peano Arithmetic (PA) [Peano, 1889], and not the primitive recursion of Primitive Recursive Arithmetic (PRA) [Skolem, 1923] which has the computational power ω^ω of LOOP-Programs [Schöning, 2003]. The Ackermann function can be easily coded by recursion into higher type $T = \mathsf{N} \to \mathsf{N}$. Yet recursion in System T has the *definition form* of primitive recursion, so we stick to the name.

$\boxed{S, T \in \mathsf{Ty}}$ Types.

$$\frac{}{\mathsf{N} \in \mathsf{Ty}}$$ type of natural numbers

$$\frac{S \in \mathsf{Ty} \qquad T \in \mathsf{Ty}}{S \to T \in \mathsf{Ty}}$$ function type.

$\boxed{\Gamma \in \mathsf{Cxt}}$ Typing contexts.

$$\frac{}{() \in \mathsf{Cxt}}$$ empty context

$$\frac{\Gamma \in \mathsf{Cxt} \qquad x \notin \Gamma}{(\Gamma, x\!:\!T) \in \mathsf{Cxt}}$$ extension by mapping variable x to type T.

$\boxed{c \in \mathsf{Cst}^T}$ Constants of type T, written $\boxed{c : T}$.

$$\frac{}{\mathsf{zero} : \mathsf{N}}$$ zero

$$\frac{}{\mathsf{suc} : \mathsf{N} \to \mathsf{N}}$$ successor function

$$\frac{}{\mathsf{rec} : T \to (\mathsf{N} \to T \to T) \to \mathsf{N} \to T}$$ primitive recursion into type T.

$\boxed{r, s, t \in \mathsf{Tm}_\Gamma^T}$ Well-typed terms, written $\boxed{\Gamma \vdash t : T}$.

$$\frac{c : T}{\Gamma \vdash c : T}$$ constant

$$\frac{(x\!:\!T) \in \Gamma}{\Gamma \vdash x : T}$$ variable

$$\frac{\Gamma, x\!:\!S \vdash t : T}{\Gamma \vdash \lambda x t : S \to T}$$ function abstraction

$$\frac{\Gamma \vdash r : S \to T \qquad \Gamma \vdash s : S}{\Gamma \vdash r\, s : T}$$ function application.

Figure 2.1: A small functional language: Gödel's System T.

we write $\Gamma \vdash t : T$ to express that t is a term of type T in context Γ. A type is either the base type N of natural numbers or a function type with domain S and codomain T, which are types themselves. We only consider finite type expressions, but of arbitrary order (nesting of function spaces).

System T can be directly interpreted in a meta language that has functions $f \in A \to B$ of arbitrary order and natural numbers $n \in \mathbb{N}$. Such a language could be classical set-theory, but for reasons becoming apparent later, we prefer a meta language with *computable* functions such as Martin-Löf Type Theory [Nordström et al., 1990]. Application of function f to argument a in the meta-language is written $f(a)$. Let further Set denote a universe of small sets and $\mathbb{N}_1$ denote the one-element type containing only the empty tuple (). We assume the meta language has a concept of primitive recursion with the following behavior:

$$\begin{array}{llllll} \mathsf{rec}\ (A \in \mathsf{Set}) & (z \in A) & (s \in \mathbb{N} \to A \to A) & (n \in \mathbb{N}) & \in & A \\ \mathsf{rec}\ (A) & (z) & (s) & (0) & = & z \\ \mathsf{rec}\ (A) & (z) & (s) & (n+1) & = & s(n)(\mathsf{rec}(A)(z)(s)(n)) \end{array}$$

We interpret types $[\![T]\!]$, contexts $[\![\Gamma]\!]$, and terms $[\![\Gamma \vdash t : T]\!]$ of System T as follows.

$$\begin{array}{llll} [\![T]\!] & & \in & \mathsf{Set} \\ [\![\mathsf{N}]\!] & & = & \mathbb{N} \\ [\![S \to T]\!] & & = & [\![S]\!] \to [\![T]\!] \\ \\ [\![\Gamma]\!] & & \in & \mathsf{Set} \\ [\![()]\!] & & = & \mathbb{N}_1 \\ [\![\Gamma, x\!:\!S]\!] & & = & [\![\Gamma]\!] \times [\![S]\!] \\ \\ [\![(x\!:\!T) \in \Gamma]\!] & (\rho \in [\![\Gamma]\!]) & \in & [\![T]\!] \\ [\![(x\!:\!T) \in (\Gamma, x\!:\!T)]\!] & (\rho, a) & = & a \\ [\![(x\!:\!T) \in (\Gamma, y\!:\!S)]\!] & (\rho, a) & = & [\![(x\!:\!T) \in \Gamma]\!](\rho) \quad \text{for } y \neq x \\ \\ [\![c : T]\!] & & \in & [\![T]\!] \\ [\![\mathsf{zero} : \mathsf{N}]\!] & & = & 0 \\ [\![\mathsf{suc} : \mathsf{N} \to \mathsf{N}]\!] & (n) & = & n+1 \\ [\![\mathsf{rec} : \mathsf{Rec}_T]\!] & & = & \mathsf{rec}([\![T]\!]) \\ \\ [\![\Gamma \vdash t : T]\!] & (\rho \in [\![\Gamma]\!]) & \in & [\![T]\!] \\ [\![\Gamma \vdash c : T]\!] & (\rho) & = & [\![c : T]\!] \\ [\![\Gamma \vdash x : T]\!] & (\rho) & = & [\![(x\!:\!T) \in \Gamma)]\!](\rho) \\ [\![\Gamma \vdash \lambda x t : S \to T]\!] & (\rho)(a) & = & [\![\Gamma, x\!:\!S \vdash t : T]\!](\rho, a) \\ [\![\Gamma \vdash r\, s : T]\!] & (\rho) & = & [\![\Gamma \vdash r : S \to T]\!](\rho)([\![\Gamma \vdash s : S]\!](\rho)) \end{array}$$

Now if we have a closed program $\vdash t : \mathsf{N}$ that computes a natural number, we can obtain its value as $[\![\vdash t : \mathsf{N}]\!]() \in \mathbb{N}$.

Looking closely, our interpretation does not really "explain" what natural numbers and functions are, since it just maps object-level concepts to the corresponding meta-level concepts.[2] However, it does explain what variables are: placeholders for values, or,

[2] In this respect, its Tarskian semantics at its best and subject to Girard's critique [2001].

more precisely, projections out of value tuples ρ which are valuations of typing contexts Γ.

In the following, for $\Gamma \vdash t : T$ we allow ourselves to write $[\![t]\!]$ instead of $[\![\Gamma \vdash t : T]\!]$.

2.2 Normalization

Normalization is the process of transforming a program into another program, called the *normal form* of the first program. Normalization should increase the "entropy" of the program while leaving its meaning intact. The term "entropy" is here used as an analogy to entropy in thermodynamics; for instance, mixing a hot fluid with a cold one will preserve the total thermal energy of the two fluids but increase the entropy of the system. Computing the normal form might collapse some internal structure of the program into a "simpler" one which leads to the same value. For instance, the expressions $1+1$ and 2 differ but have the same value, and normalization will turn $1+1$ into 2, increasing entropy by removing structure.

Formally, let $\mathsf{nf}(t)$ denote the normal form of typed term $\Gamma \vdash t : T$. Then we expect the following soundness properties to hold:

1. $\Gamma \vdash \mathsf{nf}(t) : T$ (well-typedness of normal form),
2. $[\![\mathsf{nf}(t)]\!] = [\![t]\!]$ (preservation of meaning), and
3. $\mathsf{nf}(\mathsf{nf}(t)) = \mathsf{nf}(t)$ (idempotency).

These soundness properties do not express our intuition that normalization should increase the entropy; for instance, the identity function fulfills all three properties. We would wish for a completeness property: if two programs have the same meaning (we also say they are *semantically equal*), then they have the same normal form, or in symbols, $[\![t]\!] = [\![t']\!]$ implies $\mathsf{nf}(t) = \mathsf{nf}(t')$. This wish can be granted for closed programs of base type $\vdash t, t' : \mathsf{N}$, because their meaning is just a natural number, and this number can be returned as normal form. However, for the interesting general case $\Gamma \vdash t, t' : T$ of open terms (the context Γ is non-empty) or functional terms (T is a function type), our wish cannot be fulfilled. The interpretation of open or functional terms are (higher-order) functions whose equality is undecidable.[3]

If we remove natural numbers and primitive recursion from our language, restricting to simply-typed lambda calculus, then program equivalence becomes decidable, and we can introduce the syntactic relation $\Gamma \vdash t = t' : T$ of $\beta\eta$-equivalence between typed terms t, t' that holds iff $[\![t]\!] = [\![t']\!]$ [Friedman, 1975]. Restoring natural numbers and extending $\beta\eta$-equivalence with rules characterizing the computational behavior of primitive recursion we arrive at the relation of *definitional equality* given in Figure 2.2. Definitional equality for System T is not complete, but decidable, as we will show. Since it encompasses equality of pure simply-typed lambda terms and equality of closed natural number terms, it is a reasonable equality concept.

Definitional equality makes use of a capture-avoiding substitution operation $t[s/x]$ which is meaning-preserving in the sense that

$$[\![\Gamma \vdash t[s/x] : T]\!](\rho) = [\![\Gamma, x\!:\!S \vdash t : T]\!](\rho, [\![\Gamma \vdash s : S]\!](\rho)).$$

[3] Undecidability of Gödel's System T is related to the undecidability of arithmetic [Gödel, 1931].

Computation rules (β).

$$\frac{\Gamma, x{:}S \vdash t : T \qquad \Gamma \vdash s : S}{\Gamma \vdash (\lambda x t)\, s = t[s/x] : T}$$

$$\frac{\Gamma \vdash z : T \qquad \Gamma \vdash s : \mathsf{N} \to T \to T}{\Gamma \vdash \mathsf{rec}\, z\, s\, \mathsf{zero} = z : T}$$

$$\frac{\Gamma \vdash z : T \qquad \Gamma \vdash s : \mathsf{N} \to T \to T \qquad \Gamma \vdash n : \mathsf{N}}{\Gamma \vdash \mathsf{rec}\, z\, s\, (\mathsf{suc}\, n) = s\, n\, (\mathsf{rec}\, z\, s\, n) : T}$$

Function extensionality (η).

$$\frac{\Gamma \vdash t : S \to T}{\Gamma \vdash \lambda x.\, t\, x = t : S \to T}$$

Compatibility rules.

$$\frac{c : T}{\Gamma \vdash c = c : T}$$

$$\frac{(x{:}T) \in \Gamma}{\Gamma \vdash x = x : T}$$

$$\frac{\Gamma, x{:}S \vdash t = t' : T}{\Gamma \vdash \lambda x t = \lambda x t' : S \to T}$$

$$\frac{\Gamma \vdash r = r' : S \to T \qquad \Gamma \vdash s = s' : S}{\Gamma \vdash r\, s = r'\, s' : T}$$

Equivalence rules (reflexivity, symmetry, transitivity).

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash t = t : T}$$

$$\frac{\Gamma \vdash t = t' : T}{\Gamma \vdash t' = t : T}$$

$$\frac{\Gamma \vdash t_1 = t_2 : T \qquad \Gamma \vdash t_2 = t_3 : T}{\Gamma \vdash t_1 = t_3 : T}$$

Figure 2.2: Definitional Equality for System T.

As a consequence, definitional equality is sound, i. e., $\Gamma \vdash t = t' : T$ implies $[\![\Gamma \vdash t : T]\!] = [\![\Gamma \vdash t' : T]\!]$. While semantic equality cannot be decided, definitional equality shall be decided by normalization, so we aim at the following ties between definitional equality and normalization.

1. $\Gamma \vdash \mathsf{nf}(t) = t : T$ (soundness).

2. If $\Gamma \vdash t = t' : T$ then $\mathsf{nf}(t) = \mathsf{nf}(t')$ (completeness).

Soundness says that normalization is compatible with definitional equality, and completeness says that definitionally equal terms have identical normal forms.

We normalize a term t by repeatedly applying the computation (β) and extensionality (η) equations from left to right to one of t's subterms. This process is known as *reduction* and it terminates in the normal form of t [Tait, 1967]. However, Berger and Schwichtenberg [1991] discovered a method to obtain a normalization function as instance of the evaluation function, and we will investigate it in the next section.

2.3 Normalization by Evaluation

Normalization can be viewed as evaluation of *open* terms. Under this view, a variable changes its role subtly from *place holder for values* to *unknown*.[4] Evaluating an expression with unknowns in general does not return a value but another, possibly simplified, expression with unknowns. Unknowns can *block* evaluation in these two cases:

1. An unknown function blocks application.

2. An unknown number blocks recursion.

Normal terms can be defined mutually with *neutral terms*,[5] which are blocked terms in normal form, by the rules in Figure 2.3.
Normalization-by-evaluation (NbE) works as follows:

1. We extend the interpretation of types such that the base type N contains the neutral terms of type N.

2. We define *reflection* functions $\uparrow^T$ that map neutral terms of type T to semantic objects in $[\![T]\!]$.

3. We define *reification* functions $\downarrow^T$ that map semantic objects in $[\![T]\!]$ to normal forms of type T. [6]

4. We obtain the normal form of a term $\Gamma \vdash t : T$ by reifying the value of t obtained in a environment of reflected variables.

[4] A bound variable is a place holder for some value, but a free variable should be considered as an unknown value.

[5] The notion of neutral term is due to Altenkirch et al. [1995] and should not be confused with Girard's notion of neutrality [1989] which is used in strong normalization proofs using reducibility candidates. Girard's concept is different but shares the property that substitution of a neutral for a variable does not create a β-redex.

[6] Berger and Schwichtenberg called reification an "inverse of the evaluation functional".

$\boxed{v \in \mathsf{Nf}^T_\Gamma}$ normal terms, written $\boxed{\Gamma \vdash v \leftleftarrows T}$.

$$\frac{}{\Gamma \vdash \mathsf{zero} \leftleftarrows \mathsf{N}} \qquad \frac{\Gamma \vdash v \leftleftarrows \mathsf{N}}{\Gamma \vdash \mathsf{suc}\, v \leftleftarrows \mathsf{N}} \qquad \frac{\Gamma \vdash u \rightrightarrows \mathsf{N}}{\Gamma \vdash u \leftleftarrows \mathsf{N}} \qquad \frac{\Gamma, x\!:\!S \vdash v \leftleftarrows T}{\Gamma \vdash \lambda x v \leftleftarrows S \to T}$$

$\boxed{u \in \mathsf{Ne}^T_\Gamma}$ neutral terms, written $\boxed{\Gamma \vdash u \rightrightarrows T}$.

$$\frac{(x\!:\!T) \in \Gamma}{\Gamma \vdash x \rightrightarrows T} \qquad \frac{\Gamma \vdash u \rightrightarrows S \to T \qquad \Gamma \vdash v \leftleftarrows S}{\Gamma \vdash u\, v \rightrightarrows T}$$

$$\frac{\Gamma \vdash v_z \leftleftarrows T \qquad \Gamma \vdash v_s \leftleftarrows \mathsf{N} \to T \to T \qquad \Gamma \vdash u \rightrightarrows \mathsf{N}}{\Gamma \vdash \mathsf{rec}\, v_z\, v_s\, u \rightrightarrows T}$$

Figure 2.3: Normal and neutral terms for System T.

Let us give a preliminary NbE algorithm as follows:

$$\begin{array}{lll}
[\![\mathsf{N}]\!] & = & \mathsf{Nf}^{\mathsf{N}} \\
[\![S \to T]\!] & = & [\![S]\!] \to [\![T]\!] \\
\\
\uparrow^T & \in & \mathsf{Ne}^T \to [\![T]\!] \\
\uparrow^{\mathsf{N}} \quad (u) & = & u \\
\uparrow^{S\to T}(u)(a \in [\![S]\!]) & = & \uparrow^T(u\, v) \qquad \text{where } v = \downarrow^S(a) \\
\\
\downarrow^T & \in & [\![T]\!] \to \mathsf{Nf}^T \\
\downarrow^N \quad (v) & = & v \\
\downarrow^{S\to T}(f) & = & \lambda x.\downarrow^T(f(a)) \qquad \text{where } a = \uparrow^S(x) \text{ and } x \text{ “fresh”} \\
\\
\uparrow^\Gamma & \in & [\![\Gamma]\!] \\
\uparrow^{()} & = & () \\
\uparrow^{\Gamma,x:S} & = & (\uparrow^\Gamma, \uparrow^S(x)) \\
\\
\mathsf{nf}^T_\Gamma & \in & \mathsf{Tm}^T_\Gamma \to \mathsf{Nf}^T_\Gamma \\
\mathsf{nf}^T_\Gamma \quad (t) & = & \downarrow^T([\![t]\!](\uparrow^\Gamma))
\end{array}$$

Since we have changed the semantic type of natural numbers, we have to update the definition of primitive recursion as well:

$$\begin{array}{lll}
\mathsf{rec}\ (T) & \in & [\![\mathsf{N} \to (\mathsf{N} \to T \to T) \to \mathsf{N} \to T]\!] \\
\mathsf{rec}\ (T)\ (z)\ (s)\ (\mathsf{zero}) & = & z \\
\mathsf{rec}\ (T)\ (z)\ (s)\ (\mathsf{suc}\, v) & = & s(v)(\mathsf{rec}(T)(z)(s)(v)) \\
\mathsf{rec}\ (T)\ (z)\ (s)\ (u) & = & \uparrow^T(\mathsf{rec}\, v_z\, v_s\, u) \text{ where } v_z = \downarrow^T(z) \text{ and } v_s = \downarrow^{\mathsf{N}\to T\to T}(s) \\
\\
[\![\mathsf{rec} : \mathsf{Rec}\, T\,]\!] & = & \mathsf{rec}(T)
\end{array}$$

Our preliminary algorithm is just a sketch, since we have been sloppy about variables in two aspects: We have omitted the contexts Γ in several places when mentioning Ne and Nf, and we have required that x should be a fresh variable without explaining what we mean by that. The purpose of the freshness condition for x in $\lambda x.\downarrow^T(f(\uparrow^S x))$ is to avoid capturing occurrences of variable x that morally stem from the reification of function f, not its argument $\uparrow^S x$. However, since f is a semantic object, the notion of variable occurrence in f is a priori undefined, and a definition via reification of f is problematic since we need the freshness condition to properly reify f. We shall address these issues in the next section.

Even with being confined to the above sketch, we can understand how normalization-by-evaluation works: Normalizing a functional term is performed by reifying its underlying function applied to an unknown of its domain. An unknown of functional type is a neutral term u that acts as a function which reifies its argument to a value v and returns the blocked application $u\,v$. If we recurse on an unknown natural number, the code of the recursive function is produced. In this sense, if we apply a function to an unknown it returns its own code.

NbE is complete for checking definitional equality. This follows from its definition $\downarrow^T(\llbracket t\rrbracket(\uparrow^\Gamma))$ because definitionally equal terms have the same interpretation. Soundness of NbE, $\Gamma \vdash t = \mathsf{nf}(t) : T$, follows by a logical relation between syntax and semantics, which we will study after having made NbE precise.

2.4 Variable Handling

In the literature, there are several approaches to the handling of variables and the freshness condition in NbE.

- *Gensym.* An implementation of NbE can obtain a fresh variable by taking one out of a global store. This side effect has no direct mathematical meaning and obstructs reasoning for the correctness of NbE. Filinski [2001] and Barral [2008] have formalized this approach using monads, arriving at a monadic NbE algorithm and a correctness proof using Kripke monadic logical relations.

- *Trial reification* [Garillot and Werner, 2007]. To find a variable that is fresh with regard to a function f we apply it to a dummy unknown. After reification, we have access to the variables actually used by f and we can pick an unused variable x. Then we properly reify f using variable x. This is a simple solution, albeit unfeasible because of exponential computational complexity.

- If we work in a *two-level lambda-calculus* [Danvy, 1996, Vestergaard, 2001a, Aehlig and Joachimski, 2004, Abel et al., 2007b] we have α-conversion both at the "lower" level of syntax and the "upper" level of "semantics". Picking a fresh variable is unproblematic since we can get the set of used variables at both levels. This approach might be criticized for departing from actual implementations of NbE which cannot make use of such trickery.

- Pitts [2010] specifies NbE in a meta-language of *nominal sets* and implements NbE in his programming language *Fresh Objective Caml*. The notion of name

and freshness are primitive in nominal logics and the accompanying programming language and apply to *all* objects, even (set-theoretical) functions, not only syntax. In nominal style, the “problem” of freshness is non-existent, or, to put it the other way round, nominal style exists to obliviate the problem of freshness.

- *Term families* [Berger and Schwichtenberg, 1991, Berger et al., 2003, Filinski, 1999] allow to systematically rename bound variables in order to make room for a new bound variable. Term families rely on a *de Bruijn level* representation of lambda-terms which uses integers $k \in \mathbb{N}$ for variable names and numbers lambda-bound variables consecutively from outside in. In the term family approach, base types are interpreted as function spaces $[\![\mathsf{N}]\!] = \mathbb{N} \to \mathsf{Nf}^{\mathsf{N}}$, where the integer argument denotes the difference by which to shift all de Bruijn levels upward.

- *Liftable terms* [Aehlig and Joachimski, 2004, Abel et al., 2007a] are similiar to term families only that they rely on a *de Bruijn index* representation. Not the bound, but the *free* variables are shifted upwards to make room for a binder. This style will be presented in detail in Section 2.5.

- *Locally nameless* term representations go back to Pollack [1994] and implement the Barendregt convention that bound variables should be distinct from free variables. In locally nameless style, bound variables are represented as de Bruijn indices and free variables as de Bruijn levels or by name. We have used this style in [Abel et al., 2011, Abel, 2010a] and present it in Chapter 3.

- Coquand [1994] derives NbE from *Kripke semantics* meaning that all semantic objects are relative to a typing context, can access this context to pick a fresh variable, and can be lifted to an extended context. Kripke semantics is often combined with de Bruijn index representations [Danielsson, 2007] and can be generalized to presheaf models [Altenkirch et al., 1995].

- *Contextual reification* [Abel et al., 2008, Abel, 2008, 2009a] uses de Bruijn levels in the semantics and names in the syntax. Quotation works with a set of used names and can pick a fresh name avoiding this set. Contextual reification is similar to the locally nameless style and can also be viewed as a light-weight variant of the Kripke approach. It confines the “Kripke” to the correctness proof of NbE while the actual algorithm does not work with context extensions or morphisms.

This list might not be comprehensive, but already gives an impression of the variety of NbE styles.

2.5 Liftable Terms

When we look at reflection and reification, we note that

1. in reification at function type $\downarrow^{S \to T}(f)$, we need a context Γ of variables currently in scope to pick a fresh variable $x \not\in \Gamma$,

2. this fresh variable is reflected into the semantics as value $a = \uparrow^{S}(x)$,

3. in reflection at function type $\uparrow^{S\to T}(u)$, a value a might be reified to a normal form $v = \downarrow^S(a)$ in a context Γ' different from the one where it was created (in fact Γ' can only be an extension of Γ, but this is not clear a priori).

These issues can be addressed by using *liftable* neutrals in the semantics of base types, i. e., neutrals that can be used in a different context than created. We set

$$\begin{array}{lcl} \mathsf{Nf}^T & = & (\Gamma \in \mathsf{Cxt}) \to \mathsf{Nf}^T_\Gamma \\ \mathsf{Ne}^T & = & (\Gamma \in \mathsf{Cxt}) \to (\mathsf{Ne}^T_\Gamma \uplus \{\bot\}) \end{array}$$

and we use $\hat{v}$ to denote a liftable normal term and $\hat{u}$ for a liftable neutral term. A liftable neutral $\hat{u}$ may be undefined for some context Γ, thus, we can have $\hat{u}(\Gamma) = \bot$. Application of or to an undefined neutral yields again the undefined neutral, $\bot\, v = \bot$ and $\mathsf{rec}\, v_z\, v_s\, \bot = \bot$. Application of liftable terms is overloaded as $(\hat{u}\,\hat{v})(\Gamma) = \hat{u}(\Gamma)\;\hat{v}(\Gamma)$ and liftable constants are simply defined by $\hat{c}(\Gamma) = c$. The base type Nat of natural numbers is inductively defined by the rules

$$\frac{}{\mathsf{zero} \in \mathsf{Nat}} \qquad \frac{n \in \mathsf{Nat}}{\mathsf{suc}\, n \in \mathsf{Nat}} \qquad \frac{\hat{u} \in \mathsf{Ne}^{\mathsf{N}}}{\hat{u} \in \mathsf{Nat}}$$

to allow us embedding of liftable neutrals and induction on natural numbers for the recursor. We can reify elements of Nat to normal forms recursively:

$$\begin{array}{lcll} \downarrow^{\mathsf{Nat}} & \in & \mathsf{Nat} \to \mathsf{Nf}^{\mathsf{N}} & \\ \downarrow^{\mathsf{Nat}}(\mathsf{zero}) & = & \widehat{\mathsf{zero}} & \\ \downarrow^{\mathsf{Nat}}(\mathsf{suc}\, n) & = & \widehat{\mathsf{suc}}\ \downarrow^{\mathsf{Nat}}(n) & \\ \downarrow^{\mathsf{Nat}}(\hat{u})(\Gamma) & = & \mathsf{zero} & \text{if } \hat{u}(\Gamma) = \bot \\ \downarrow^{\mathsf{Nat}}(\hat{u})(\Gamma) & = & \hat{u}(\Gamma) & \text{otherwise} \end{array}$$

The updated NbE algorithm now reads as follows:

$$\begin{array}{lcll} [\![\mathsf{N}]\!] & = & \mathsf{Nat} & \\ [\![S \to T]\!] & = & [\![S]\!] \to [\![T]\!] & \\ & & & \\ \uparrow^T & \in & \mathsf{Ne}^T \to [\![T]\!] & \\ \uparrow^{\mathsf{N}}(\hat{u}) & = & \hat{u} & \\ \uparrow^{S\to T}(\hat{u})(a \in [\![S]\!]) & = & \uparrow^T(\hat{u}\,\hat{v}) & \text{where } \hat{v} = \downarrow^S(a) \\ & & & \\ \downarrow^T & \in & [\![T]\!] \to \mathsf{Nf}^T & \\ \downarrow^{\mathsf{N}}(n) & = & \downarrow^{\mathsf{Nat}}(n) & \\ \downarrow^{S\to T}(f)(\Gamma) & = & \lambda x.\, \downarrow^T(f(a))(\Gamma, x{:}S) & \text{where } a = \uparrow^S(\hat{x}^S_\Gamma) \\ & & & \\ \uparrow^\Gamma & \in & [\![\Gamma]\!] & \\ \uparrow^{()} & = & () & \\ \uparrow^{\Gamma,x:S} & = & (\uparrow^\Gamma, \uparrow^S(\hat{x}^S_\Gamma)) & \\ & & & \\ \mathsf{nf}^T_\Gamma & \in & \mathsf{Tm}^T_\Gamma \to \mathsf{Nf}^T_\Gamma & \\ \mathsf{nf}^T_\Gamma(t) & = & \downarrow^T([\![t]\!](\uparrow^\Gamma))(\Gamma) & \end{array}$$

The description uses a *liftable variable* $\hat{x}^S_\Gamma$ in two places. It denotes the variable x created in context $\Gamma, x : S$ and usable in extended contexts $\Gamma, x : S, \Gamma'$. Formally, we have to define its value for all contexts—we shall return dummy value $\bot$ if it is used in contexts which are not extensions of its creation context.

$$\begin{array}{lcll} \hat{x}^S_\Gamma & \in & \mathsf{Ne}^S & \\ \hat{x}^S_\Gamma(\Gamma, x{:}S, \Gamma') & = & x & \\ \hat{x}^S_\Gamma(\Gamma') & = & \bot & \text{if } \Gamma' \text{ is not an extension of } \Gamma. \end{array}$$

What remains to be done is to update the primitive recursor to work on liftable terms in case of a neutral natural number:

$$\begin{array}{lcl} \mathsf{rec}\ (T) & \in & [\![\mathsf{N} \to (\mathsf{N} \to T \to T) \to \mathsf{N} \to T]\!] \\ \mathsf{rec}\ (T)\ (z)\ (s)\ (\mathsf{zero}) & = & z \\ \mathsf{rec}\ (T)\ (z)\ (s)\ (\mathsf{suc}\, n) & = & s(n)(\mathsf{rec}(T)(z)(s)(n)) \\ \mathsf{rec}\ (T)\ (z)\ (s)\ (\hat{u}) & = & \uparrow^T(\widehat{\mathsf{rec}}\, \hat{v}_z\, \hat{v}_s\, \hat{u}) \text{ where } \hat{v}_z = \downarrow^T(z) \text{ and } \hat{v}_s = \downarrow^{\mathsf{N} \to T \to T}(s) \end{array}$$

In the following, we adopt the notation $\downarrow^T_\Gamma(a)$ as shorthand for $\downarrow^T(a)(\Gamma)$.

2.6 Soundness of Normalization by Evaluation

We now turn to the problem of soundness of normalization $\Gamma \vdash t = \mathsf{nf}(t) : T$, which for NbE reads

$$\Gamma \vdash t = \downarrow^T_\Gamma a : T \text{ where } a = [\![t]\!](\uparrow^\Gamma).$$

We obtain soundness from a Kripke logical relation $\boxed{\Gamma \vdash t : T \circledR a}$ between a typed term t and a value $a \in [\![T]\!]$. The logical relation is constructed to imply $\Gamma \vdash t = \downarrow^T_\Gamma a : T$. Soundness follows after we establish $\Gamma \vdash t : T \circledR [\![t]\!](\uparrow^\Gamma)$ via the *fundamental lemma* of logical relations.

We say context Γ' extends context Γ, written $\boxed{\Gamma' \leq \Gamma}$, if $\Gamma' = \Gamma, \Delta$ for some context Δ. Our relation $\Gamma \vdash t : T \circledR a$ is defined by induction on type T as follows:

$$\begin{array}{lcl} \Gamma \vdash t : \mathsf{N} \circledR \hat{v} & \iff & \forall \Gamma' \leq \Gamma.\ \ \Gamma' \vdash t = \hat{v}(\Gamma') : \mathsf{N} \\ \Gamma \vdash r : S \to T \circledR f & \iff & \forall \Gamma' \leq \Gamma.\ \ \Gamma' \vdash s : S \circledR a \implies \Gamma' \vdash r\, s : T \circledR f(a) \end{array}$$

The logical relation is sandwiched between reflection and reification, more precisely, by induction on type T we can prove the implications

$$\begin{array}{rcl} (\forall \Gamma' \leq \Gamma.\ \ \Gamma' \vdash u = \hat{u}(\Gamma') : T) & \implies & \Gamma \vdash u : T \circledR \uparrow^T(\hat{u}) \\ \Gamma \vdash t : T \circledR a & \implies & \forall \Gamma' \leq \Gamma.\ \ \Gamma' \vdash t = \downarrow^T_{\Gamma'}(a) : T. \end{array}$$

A consequence of the first implication is that variables are logical related to their reflections, we have $\Gamma, x : T \vdash x : T \circledR \uparrow^T(\hat{x}^T_\Gamma)$ because $\Gamma' \vdash x = \hat{x}^T_\Gamma(\Gamma') : T$ for all $\Gamma' \leq (\Gamma, x{:}T)$.

To prove the fundamental lemma, we have to extend the logical relations to substitutions $\boxed{\Gamma \vdash \sigma : \Delta}$ and environments. Parallel substitutions are introduced by the rules

$$\frac{}{\Gamma \vdash () : ()} \qquad \frac{\Gamma \vdash \sigma : \Delta \qquad \Gamma \vdash s : S}{\Gamma \vdash (\sigma, s/x) : \Delta, x{:}S}$$

and the application of a substitution $t[\sigma]$ is an operation which enjoys the typing

$$\frac{\Gamma \vdash \sigma : \Delta \qquad \Delta \vdash t : T}{\Gamma \vdash t[\sigma] : T}$$

A Kripke logical relation $\boxed{\Gamma \vdash \sigma : \Delta \circledR \rho}$ between a substitution σ and an environment $\rho \in [\![\Delta]\!]$ and a semantic typing judgment $\boxed{\Gamma \vDash t : T}$ are constructed as follows:

$$\begin{array}{lcl}
\Gamma \vdash () : () \circledR () & :\Longleftrightarrow & \text{true} \\
\Gamma \vdash (\sigma, s/x) : (\Delta, x{:}S) \circledR (\rho, a) & :\Longleftrightarrow & \Gamma \vdash \sigma : \Delta \circledR \rho \text{ and } \Gamma \vdash s : S \circledR a \\
\\
\Gamma \vDash t : T & :\Longleftrightarrow & \forall \Delta, \sigma, \rho.\ \ \Delta \vdash \sigma : \Gamma \circledR \rho \\
 & & \qquad \Longrightarrow\ \Delta \vdash t[\sigma] : T \circledR [\![t]\!](\rho)
\end{array}$$

By induction on $\Gamma \vdash t : T$ we prove $\Gamma \vDash t : T$ — this is called the fundamental lemma of logical relations. For the identity substitution $\Gamma \vdash \mathsf{id} : \Gamma$ we have $\Gamma \vdash \mathsf{id} : \Gamma \circledR \uparrow^{\Gamma}$, thus, the fundamental lemma implies $\Gamma \vdash t : T \circledR [\![t]\!](\uparrow^{\Gamma})$ and finally the soundness $\Gamma \vdash t = \downarrow^{T}_{\Gamma}([\![t]\!](\uparrow^{\Gamma})) : T$ of NbE.

2.7 Summary

We have been introduced to NbE for simple types and *intrinsic typing*, i. e., types are defined first and terms are never considered in their raw, untyped form. The approach relies very much on the definition of type interpretation by induction on the syntax of types. For this reason, it does not scale directly to dependent types and impredicativity. While it served us well for a prime exposition, we switch to an *extrinsic*, type-assignment, *terms first!* style that allows us to formulate NbE uniformly for many type systems.

3 Untyped Normalization-By-Evaluation and Type Assignment

Describing a language as typed terms, as we have done in Section 2.1, has the advantage that we never have to look at terms that have no meaning, such as the term $\Omega = (\lambda x.\, x\, x)\, (\lambda x.\, x\, x)$. As a consequence, the denotation $[\![t]\!]$ is a priori well-defined for any term t.

But typed terms have also disadvantages: For one, computation is oblivious of types—the value of a term t does not depend on its type T. In practice, machines that execute computations are untyped, or at best, softly typed. Types give assurance that nothing "goes wrong" [Milner, 1978], but it is sufficient that we type-check a program before its execution or compilation, we can then throw away types. Thus, it makes sense to adopt a *type assignment* perspective: terms get assigned a type rather than *having* a fixed type from the beginning.[1]

Another problem with typed terms surfaces as we leave the world of simple types and embark on a journey into polymorphic and dependent types. Untyped terms allow us to separate concerns: we can first specify operations on untyped terms and then prove typing properties of these operations. Typed terms do not give us this flexibility; everything has to be well-typed from the beginning. Attempts to formulate NbE for dependently typed terms have not fully succeeded so far. Initial progress has been made by Altenkirch and Chapman [2009] who formalize a typed normalizer for a version of System T with explicit substitutions. Chapman [2009] has extended this approach to Martin-Löf's dependently-typed logical framework with one (albeit empty) universe and presented a partial formalization in Agda. Danielsson [2007] has formalized NbE for the same type theory in a precursor of Agda. An early, influential work is Catarina Coquand's formalization of NbE [1994] using Kripke function spaces, however, it is also confined to simple types.

Using a type-assignment perspective, we have succeeded to describe normalization by evaluation for both dependent types and impredicative polymorphism. The key tool is an underlying evaluation mechanism for untyped lambda calculus into a set of values D which forms an applicative structure. NbE amounts to adding the type-directed reflection and reification functions and logical relations that prove soundness and completeness of NbE. In this chapter, we study NbE for untyped lambda-calculus as the basic mechanism.

[1] The view that terms have a fixed type seems to be the heritage of the Curry-Howard-Correspondence that maps proofs to terms and propositions to types. It does not make much sense to consider a proof without the proposition it tries to proof; thus, propositions come before proofs in the world of logics. In the world of computation, programs can be meaningfully considered without their types, hence, programs come before their types.

3.1 Untyped NbE Using Domains

In the following we consider untyped lambda-calculus with variables represented as *de Bruijn indices* [de Bruijn, 1972]. An index is a natural number indicating how many λs stand between[2] the variable and its binding λ. The terms $t \in \mathsf{Exp}$, neutral terms $u \in \mathsf{Ne}$ and normal terms $v \in \mathsf{Nf}$ are given by the following grammar:

$$
\begin{array}{llcll}
\mathsf{Exp} & \ni & r, s, t & ::= & \mathsf{v}_i & \text{the } i\text{th variable} \\
 & & & \mid & \lambda t & \text{abstracting the 0th variable in } t \\
 & & & \mid & r\, s & \text{applying } r \text{ to } s. \\
\\
\mathsf{Ne} & \ni & u & ::= & \mathsf{v}_i \mid u\, v & \beta\text{-normal non-abstraction term} \\
\mathsf{Nf} & \ni & v & ::= & u \mid \lambda v & \beta\text{-normal term.}
\end{array}
$$

The term $\lambda x.\, x\,(\lambda y.\, y\, x)$ is represented by $\lambda.\, 0\,(\lambda.\, 0\, 1)$ in de Bruijn notation. De Bruijn indices should not be confused with *de Bruijn levels* which number the λ-bound variables from outside in. In de Bruijn level representation, the same term is written $\lambda 0.\, 0\,(\lambda 1.\, 1\, 0)$ which is obtained by replacing the variable x by 0 and y by 1.

Untyped lambda-calculus can be interpreted in a Scott domain $\mathsf{D} \cong [\mathsf{D} \to \mathsf{D}]$ which is a set D that is isomorphic to the set $[\mathsf{D} \to \mathsf{D}]$ of continuous functions on D [Stoy, 1977]. We obtain untyped NbE by extending D with a suitable representation of neutrals. The following "grammar" resembles data type declarations in functional languages and corresponds to a system of two solvable domain equations defining D and D^{ne}:

$$
\begin{array}{llcll}
\mathsf{D} & \ni & a, b, d & ::= & \mathsf{Abs}\ \ (f \in [\mathsf{D} \to \mathsf{D}]) & \text{function value} \\
 & & & \mid & \mathsf{Up}\ \ (e \in \mathsf{D}^{\mathsf{ne}}) & \text{neutral value} \\
\\
\mathsf{D}^{\mathsf{ne}} & \ni & e & ::= & \mathsf{Level}(k \in \mathbb{N}) & \text{de Bruijn level} \\
 & & & \mid & \mathsf{App}\ \ (e \in \mathsf{D}^{\mathsf{ne}},\ d \in \mathsf{D}) & \text{neutral application.}
\end{array}
$$

We write x_k for $\mathsf{Level}(k)$ and $e\, d$ for $\mathsf{App}(e, d)$. For the semantic world we have chosen de Bruijn levels instead of indices because a level is like a new constant, whereas an index shifts its meaning according to the context it is considered. Our choice avoids complications in the treatment of names such as the need to build liftings into the semantics or to employ a Kripke semantics in the first place. Our approach resembles the *locally nameless* representation of λ-terms [Pollack, 1994] and we have successfully applied it to NbE for dependent types [Abel et al., 2008, 2011, Abel, 2010a].

Application on D is a continuous function defined by cases as follows:

$$
\begin{array}{lcl}
_ \cdot _ & \in & [\mathsf{D} \to [\mathsf{D} \to \mathsf{D}]] \\
\mathsf{Abs}(f) \cdot d & = & f(d) \\
\mathsf{Up}(e) \cdot d & = & \mathsf{Up}(e\, d)
\end{array}
$$

For environments $\rho \in \mathsf{Env} = \mathbb{N} \to \mathsf{D}$, update $(\rho, d) \in \mathsf{Env}$ is defined as $(\rho, d)(0) = d$ and

[2] "Between" refers to the view of a term as a tree, not its textual representation.

$(\rho, d)(i+1) = \rho(i)$. Using environments, we define the evaluation function as usual.

$$
\begin{array}{lcll}
[\![_]\!] & \in & \mathsf{Exp} \to [\mathsf{Env} \to \mathsf{D}] & \\
[\![\mathsf{v}_i]\!]\,(\rho) & = & \rho(i) & \\
[\![\lambda t]\!]\,(\rho) & = & \mathsf{Abs}(f) & \text{where } f(d) = [\![t]\!](\rho, d) \\
[\![r\,s]\!](\rho) & = & [\![r]\!]\rho \cdot [\![s]\!]\rho. &
\end{array}
$$

To turn evaluation into normalization, we define a family of *read-back*[3] functions $\mathsf{R}^{\mathsf{nf}}_n$ that convert values $d \in \mathsf{D}$ that stem from terms with at most n free indices into a β-normal form. It is mutually defined with a family of read-back functions $\mathsf{R}^{\mathsf{ne}}_n$ for neutral values $e \in \mathsf{D}^{\mathsf{ne}}$.

$$
\begin{array}{lcl}
\mathsf{R}^{\mathsf{nf}}_n & \in & \mathsf{D} \to \mathsf{Nf} \\
\mathsf{R}^{\mathsf{nf}}_n(\mathsf{Abs}(f)) & = & \lambda.\, \mathsf{R}^{\mathsf{nf}}_{n+1}(f(\mathsf{x}_n)) \\
\mathsf{R}^{\mathsf{nf}}_n(\mathsf{Up}(e)) & = & \mathsf{R}^{\mathsf{ne}}_n(e)
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{R}^{\mathsf{ne}}_n & \in & \mathsf{D}^{\mathsf{ne}} \to \mathsf{Ne} \\
\mathsf{R}^{\mathsf{ne}}_n(\mathsf{x}_k) & = & \mathsf{v}_{n-(k+1)} \\
\mathsf{R}^{\mathsf{ne}}_n(e\,d) & = & \mathsf{R}^{\mathsf{ne}}_n(e)\, \mathsf{R}^{\mathsf{nf}}_n(d)
\end{array}
$$

The number n is needed to convert a de Bruijn level x_k into its corresponding de Bruijn index v_i. If we consider a context of n variables, de Bruijn levels appear in ascending order form left to right (0 is the variable that was introduced first) while de Bruijn indices appear in descending order (0 is variable that was introduced last).

level	x_0	x_1	$\dots$	x_k	$\dots$	$\mathsf{x}_{n-(i+1)}$	$\dots$	x_{n-2}	x_{n-1}
index	v_{n-1}	v_{n-2}	$\dots$	$\mathsf{v}_{n-(k+1)}$	$\dots$	v_i	$\dots$	v_1	v_0

Level and index of a variable add up to $n-1$, thus $\mathsf{v}_{n-(k+1)}$ is the index corresponding to the kth level. Symmetrically, $\mathsf{x}_{n-(i+1)}$ is the level corresponding to the ith index, and we define an initial environment ρ_n for the evaluation of an expression with free indices below n as

$$
\begin{array}{lcl}
\rho_n & \in & \mathsf{Env} \\
\rho_n(i) & = & \mathsf{Up}\left(\mathsf{x}_{n-(i+1)}\right).
\end{array}
$$

While a well-scoped de Bruijn index i is strictly below n, it is technically convenient[4] if $\rho_n(i)$ is defined also for $i \geq n$, even if it returns garbage. Thus, we consider here, and throughout this thesis, subtraction as truncating subtraction on the natural numbers: $n - i = 0$ if $i \geq n$. That we never encounter an ill-scoped de Bruijn index or level during NbE, i. e., that truncation is actually never necessary, is a consequence of the soundness theorem (Section 3.9).

[3] A variant of our read-back procedure has been introduced by Grégoire and Leroy [2002]. Read-back is often called *quotation* [Coquand, 1994, Coquand and Dybjer, 1997, Altenkirch et al., 1995].

[4] If we considered subtraction on $\mathbb{Z}$, we would either have to allow negative de Bruijn indices and levels, or let R^{ne} and ρ_n be partial functions, requiring a partial application on Ne. The definition of the candidate space $\bot, \top$ in Section 3.4 would be less direct.

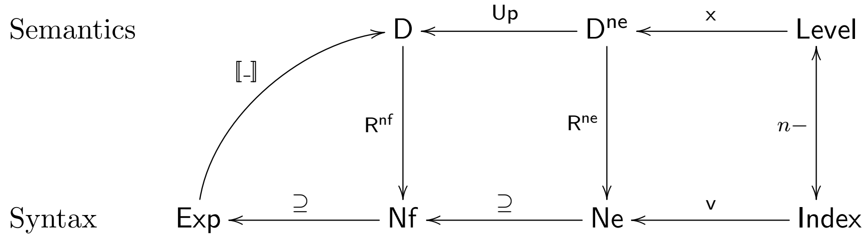

Figure 3.1: Untyped NbE in locally nameless style.

A family of (partial) normalization functions for the untyped lambda calculus is finally obtained by

$$\begin{array}{lcl} \mathsf{nf}_n & \in & \mathsf{Exp} \to \mathsf{Nf} \\ \mathsf{nf}_n(t) & = & \mathsf{R}^{\mathsf{nf}}_n(\llbracket t \rrbracket(\rho_n)). \end{array}$$

$\mathsf{nf}_n(t)$ returns the normal form of term t with indices $< n$ if such a normal form exists. We are not going to prove this here but refer to the works of Aehlig and Joachimski [2004] and Filinski and Rohde [2004]. We are not actually interested in untyped NbE per se, but we use it as a tool to study normalizing type assignment systems. Figure 3.1 summarizes our formulation of untyped NbE.

3.2 Untyped NbE Using Partial Applicative Structures

Domains are a nice tool to give semantics to programming languages, but they are not a primitive notion of set theory or type theory. In fact, the construction of a reflexive domain $\mathsf{D} \cong [\mathsf{D} \to \mathsf{D}]$ is quite involved, requiring a framework of complete partial orders, embedding-projection pairs, limits and so on [Schmidt, 1986]. It turns out that evaluation into domains is just a special case of evaluation into *partial applicative structures*.

For our purposes, a partial applicative structure[5] is a set D with partial application operation $_ \cdot _ \in \mathsf{D} \times \mathsf{D} \rightharpoonup \mathsf{D}$ and partial evaluation operation $\llbracket _ \rrbracket(_) \in \mathsf{Exp} \times \mathsf{Env} \rightharpoonup \mathsf{D}$ where $\mathsf{Env} = \mathbb{N} \rightharpoonup \mathsf{D}$. These operation must satisfy the following axioms:

$$\begin{array}{lcl} \llbracket \mathsf{v}_i \rrbracket\ (\rho) & \doteq & \rho(i) \\ \llbracket \lambda t \rrbracket\ (\rho) \cdot d & \doteq & \llbracket t \rrbracket(\rho, d) \\ \llbracket r\ s \rrbracket(\rho) & \doteq & \llbracket r \rrbracket(\rho) \cdot \llbracket s \rrbracket(\rho) \end{array}$$

We read these equations as: if one side is defined, so is the other, and then both are equal (Kleene equivalence $\doteq$). By definition, these laws hold for evaluation into domains. However, the more abstract framework of partial applicative structures also accommodates a *defunctionalized*[6] variant of evaluation into domains. If we look closely, we see

[5]Our notion of partial applicative structure is analogous to Barendregt's *syntactical applicative structure* [1984] or Mitchell's *environment models* [1996].

[6]Defunctionalized interpreters are due to Reynolds [1972] and are used by Ager et al. [2003] to systematically develop abstract normalization machines.

that all semantic functional values stem from the evaluation of a lambda-abstraction. This allows us a positive formulation of D using *closures* $(\underline{\lambda}t)\rho$ [Landin, 1964] that does not rely on domain theory.

$$\begin{array}{lcllll} \mathsf{D} & \ni & a, b, d, f & ::= & (\underline{\lambda}t)\rho & \text{closure (function value)} \\ & & & \mid & e & \text{neutral value} \\ \\ \mathsf{D}^{\mathsf{ne}} & \ni & e & ::= & \mathsf{x}_k \mid e\,d & \\ \mathsf{Env} & = & \mathbb{N} \to \mathsf{D} & & & \end{array}$$

Besides replacing continuous function with closures, nothing has changed, except that we leave constructor $\mathsf{Up} \in \mathsf{D}^{\mathsf{ne}} \to \mathsf{D}$ implicit. Application and evaluation are now defined mutually.

$$\begin{array}{lcl} (\underline{\lambda}t)\rho \cdot d & \doteq & [\![t]\!](\rho, d) \\ e \cdot d & = & e\,d \\ \\ [\![\mathsf{v}_i]\!](\rho) & = & \rho(i) \\ [\![\lambda t]\!](\rho) & = & (\underline{\lambda}t)\rho \\ [\![r\,s]\!](\rho) & \doteq & [\![r]\!](\rho) \cdot [\![s]\!](\rho) \end{array}$$

Still, we are using partial functions which might be a convenient tool of set theory but are not a primitive notion in type theory. A principled method [Bove et al., 2013] to construct a partial function in type theory is to first define the graph of the function inductively and then construct the "partial" function by recursion over its inductive graph.[7] The graphs of application and evaluation are given by the mutual inductive relations $\boxed{f \cdot a \searrow b}$ and $\boxed{[\![t]\!](\rho) \searrow a}$.

$$\frac{[\![t]\!](\rho, a) \searrow b}{(\underline{\lambda}t)\rho \cdot a \searrow b} \qquad \frac{}{e \cdot d \searrow e\,d}$$

$$\frac{}{[\![\mathsf{v}_i]\!](\rho) \searrow \rho(i)} \qquad \frac{}{[\![\lambda t]\!](\rho) \searrow (\underline{\lambda}t)\rho} \qquad \frac{[\![r]\!](\rho) \searrow f \qquad [\![s]\!](\rho) \searrow a \qquad f \cdot a \searrow b}{[\![r\,s]\!](\rho) \searrow b}$$

Since read-back of closures triggers the evaluation of the function body, the read-back functions are partial and we define their graphs $\boxed{\mathsf{R}^{\mathsf{nf}}_n\, d \searrow v}$ and $\boxed{\mathsf{R}^{\mathsf{ne}}_n\, e \searrow u}$ inductively as well.

$$\frac{[\![t]\!](\rho, \mathsf{x}_n) \searrow b \qquad \mathsf{R}^{\mathsf{nf}}_{n+1}\, b \searrow v}{\mathsf{R}^{\mathsf{nf}}_n\, (\underline{\lambda}t)\rho \searrow \lambda v} \qquad \frac{\mathsf{R}^{\mathsf{ne}}_n\, e \searrow u}{\mathsf{R}^{\mathsf{nf}}_n\, e \searrow u}$$

$$\frac{}{\mathsf{R}^{\mathsf{ne}}_n\, \mathsf{x}_k \searrow \mathsf{v}_{n-(k+1)}} \qquad \frac{\mathsf{R}^{\mathsf{ne}}_n\, e \searrow u \qquad \mathsf{R}^{\mathsf{nf}}_n\, d \searrow v}{\mathsf{R}^{\mathsf{ne}}_n\, e\,d \searrow u\,v}$$

Finally, the relation describing NbE is given by relation composition:

$$\mathsf{nf}_n(t) \searrow v \;:\Longleftrightarrow\; [\![t]\!](\rho_n) \searrow d \text{ and } \mathsf{R}^{\mathsf{nf}}_n\, d \searrow v.$$

[7] In fact, this "partial function" is a total function, but its domain is not the full product of argument types.

From here we pursue the following path: We consider type assignment systems for untyped terms that guarantee for typable terms $\Gamma \vdash t : T$ the normalization of evaluation and read-back. This then allows justifies the totality of the a priori partial NbE procedure.

Notes. In relational formulation, evaluation coincides with Kahn's natural semantics [1987], also called big-step operational semantics. An analogous path from a partial evaluation operation to an inductive relation has been followed by Altenkirch and Chapman [2009], including a read-back function they call *quote*—albeit for typed terms and for the sake of formalization in Agda.

3.3 Type-Assignment System T

In this section we shall reconstruct System T typing from the operational semantics given by our evaluation relation. First, let us extend expression and value language and evaluation by call-by-value natural numbers and primitive recursion. For now, we relinquish curried constants in favor of term constructors for successor and recursion.

$$\begin{array}{llllll} \mathsf{Exp} & \ni & r, s, t & ::= & \cdots \mid \mathsf{zero} \mid \mathsf{suc}(t) & \mid \mathsf{rec}(t_z, t_s, t_n) \\ \mathsf{Nf} & \ni & v & ::= & \cdots \mid \mathsf{zero} \mid \mathsf{suc}(v) & \\ \mathsf{Ne} & \ni & u & ::= & \ldots & \mid \mathsf{rec}(v_z, v_s, u) \\ \\ \mathsf{D} & \ni & a, b, d, f & ::= & \cdots \mid \mathsf{zero} \mid \mathsf{suc}(d) & \\ \mathsf{D}^{\mathsf{ne}} & \ni & e & ::= & \ldots & \mid \mathsf{rec}(d_z, d_s, e) \end{array}$$

Figure 3.2 introduces a new 4-ary relation $\boxed{\mathsf{rec}(d_z, d_s, d_n) \searrow d}$ for the execution of recursion and completes the operational semantics, now definitely call-by-value.

Evaluation is now partial not only because of non-termination, but also because of illegal operations; for instance, application of a number to an argument is undefined, as well as recursion over a closure instead of a number. In the following, we identify sets of values on which application and recursion are well-behaved and terminating. These set of values are *semantic types*.

A semantic type $\mathcal{A}$ is (for now) a predicate on D which we conceive as a subset $\mathcal{A} \subseteq \mathsf{D}$. The semantic type $\mathcal{N}at \subseteq \mathsf{D}$ of natural numbers is defined inductively by the rules

$$\frac{}{\mathsf{zero} \in \mathcal{N}at} \qquad \frac{d \in \mathcal{N}at}{\mathsf{suc}(d) \in \mathcal{N}at}.$$

For two semantic types $\mathcal{A}, \mathcal{B}$ the semantic function space $\mathcal{A} \to \mathcal{B}$ is defined by

$$\mathcal{A} \to \mathcal{B} = \{f \in \mathsf{D} \mid \forall a \in \mathcal{A}.\ \exists b \in \mathcal{B}.\ f \cdot a \searrow b\}$$

Let us introduce some suggestive, abbreviating notation for the statement that one of our operational relations produces a result in some semantic type. We write

$$\begin{array}{llll} f \cdot a & \in \mathcal{B} & \text{iff} & \exists b \in \mathcal{B}.\ f \cdot a \searrow b \\ \mathsf{rec}(d_z, d_s, d_n) & \in \mathcal{B} & \text{iff} & \exists b \in \mathcal{B}.\ \mathsf{rec}(d_z, d_s, d_n) \searrow b \\ [\![t]\!](\rho) & \in \mathcal{B} & \text{iff} & \exists b \in \mathcal{B}.\ [\![t]\!](\rho) \searrow b \end{array}$$

$\boxed{\mathsf{rec}(d_z, d_s, d_n) \searrow d}$ Primitive recursion.

$$\frac{}{\mathsf{rec}(d_z, d_s, \mathsf{zero}) \searrow d_z} \qquad \frac{\mathsf{rec}(d_z, d_s, d_n) \searrow a \qquad d_s \cdot d_n \searrow f \qquad f \cdot a \searrow b}{\mathsf{rec}(d_z, d_s, \mathsf{suc}(d_n)) \searrow b}$$

$$\frac{}{\mathsf{rec}(d_z, d_s, e) \searrow \mathsf{rec}(d_z, d_s, e)}$$

$\boxed{[\![t]\!](\rho) \searrow d}$ Extension of evaluation.

$$\frac{}{[\![\mathsf{zero}]\!](\rho) \searrow \mathsf{zero}} \qquad \frac{[\![t]\!](\rho) \searrow d}{[\![\mathsf{suc}(t)]\!](\rho) \searrow \mathsf{suc}(d)}$$

$$\frac{[\![t_z]\!](\rho) \searrow d_z \qquad [\![t_s]\!](\rho) \searrow d_s \qquad [\![t_n]\!](\rho) \searrow d_n \qquad \mathsf{rec}(d_z, d_s, d_n) \searrow d}{[\![\mathsf{rec}(t_z, t_s, t_n)]\!](\rho) \searrow d}$$

$\boxed{\mathsf{R}^{\mathsf{nf}}_n\, d \searrow v}$ and $\boxed{\mathsf{R}^{\mathsf{ne}}_n\, e \searrow u}$: extended read-back.

$$\frac{}{\mathsf{R}^{\mathsf{nf}}_n\, \mathsf{zero} \searrow \mathsf{zero}} \qquad \frac{\mathsf{R}^{\mathsf{nf}}_n\, d \searrow v}{\mathsf{R}^{\mathsf{nf}}_n\, \mathsf{suc}(d) \searrow \mathsf{suc}(v)}$$

$$\frac{\mathsf{R}^{\mathsf{nf}}_n\, d_z \searrow v_z \qquad \mathsf{R}^{\mathsf{nf}}_n\, d_s \searrow v_s \qquad \mathsf{R}^{\mathsf{ne}}_n\, e \searrow u}{\mathsf{R}^{\mathsf{ne}}_n\, \mathsf{rec}(d_z, d_s, e) \searrow \mathsf{rec}(v_z, v_s, u)}$$

Figure 3.2: Extensions to evaluation and read-back.

and analogously for the read-back and normalization relations. In the new notation, $f \in \mathcal{A} \to \mathcal{B}$ iff $f \cdot a \in \mathcal{B}$ for all $a \in \mathcal{A}$. We can prove the following "typing rules" for values; these are actually just implications which we write suggestively in rule format.

$$\frac{\forall a \in \mathcal{A}.\ \llbracket t \rrbracket(\rho, a) \in \mathcal{B}}{(\underline{\lambda} t)\rho \in \mathcal{A} \to \mathcal{B}} \qquad \frac{f \in \mathcal{A} \to \mathcal{B} \qquad a \in \mathcal{A}}{f \cdot a \in \mathcal{B}}$$

$$\frac{d_z \in \mathcal{A} \qquad d_s \in \mathcal{N}at \to \mathcal{A} \to \mathcal{A} \qquad d_n \in \mathcal{N}at}{\mathsf{rec}(d_z, d_s, d_n) \in \mathcal{A}}$$

The first two implications are immediate, the third requires an induction on $d_n \in \mathcal{N}at$.

We can now recover the typing rules for System T from the semantics. To this end, we interpret syntactic types T as semantic types $\mathcal{A}$ and typing contexts Γ as sets of environments ρ. Note that in our de Bruijn index representation, a typing context Γ is just a list of types.

$$\begin{array}{lcl} \llbracket T \rrbracket & \subseteq & \mathsf{D} \\ \llbracket \mathsf{N} \rrbracket & = & \mathcal{N}at \\ \llbracket S \to T \rrbracket & = & \llbracket S \rrbracket \to \llbracket T \rrbracket \\ \\ \llbracket \Gamma \rrbracket & \subseteq & \mathsf{Env} \\ \llbracket () \rrbracket & = & \mathsf{Env} \\ \llbracket \Gamma, S \rrbracket & = & \{(\rho, d) \mid \rho \in \llbracket \Gamma \rrbracket \text{ and } d \in \llbracket \mathcal{S} \rrbracket\} \end{array}$$

Semantically, a term t has type T in context Γ iff evaluating t in any environment $\rho \in \llbracket \Gamma \rrbracket$ results in a value of semantic type $\llbracket T \rrbracket$.

$$\boxed{\Gamma \vDash t : T} \ :\Longleftrightarrow\ \forall \rho \in \llbracket \Gamma \rrbracket.\ \ \llbracket t \rrbracket(\rho) \in \llbracket T \rrbracket$$

Using the value typing rules, we can immediately verify the following semantic typing "rules". Let context look-up $\Gamma(i)$ be defined by $(\Gamma, S)(0) = S$ and $(\Gamma, S)(i+1) = \Gamma(i)$.

$$\frac{\Gamma(i) = T}{\Gamma \vDash \mathsf{v}_i : T} \qquad \frac{\Gamma, S \vDash t : T}{\Gamma \vDash \lambda t : S \to T} \qquad \frac{\Gamma \vDash r : S \to T \qquad \Gamma \vDash s : S}{\Gamma \vDash r\, s : T}$$

$$\frac{}{\Gamma \vDash \mathsf{zero} : \mathsf{N}} \qquad \frac{\Gamma \vDash t : \mathsf{N}}{\Gamma \vDash \mathsf{suc}(t) : \mathsf{N}}$$

$$\frac{\Gamma \vDash t_z : T \qquad \Gamma \vDash t_s : \mathsf{N} \to T \to T \qquad \Gamma \vDash t_n : \mathsf{N}}{\Gamma \vDash \mathsf{rec}(t_z, t_s, t_n) : T}$$

Thus, we have derived type assignment $\boxed{\Gamma \vdash t : T}$ for System T which we define as the smallest relation closed under the above six rules. Note that neutral values and read-back have not played any role yet, but they will when we show the totality of NbE for type-assignment System T.

3.4 Candidate Spaces and Normalization for Type-Assignment System T

While we have established that the evaluation of a typable term is well-defined, we do not immediately obtain normalization. We need a stronger property: a typable term evaluates to a *reifiable* value. Note that read-back may trigger evaluation of a function body by application to a fresh de Bruijn level, and we have not established that de Bruijn levels inhabit semantic types.

Usually, if one wants more results from a model, one needs to ask more of semantic types. So far our semantic types are sets of values, now we will ask that they contain all de Bruijn levels and contain only values we can read back into normal forms. "Asking more" of semantic types, which are sometimes called *candidates* [Girard et al., 1989], can be abstractly formulated by restricting semantic types to a *candidate space* [Abel, 2009a]. In our context, a candidate space consists of two sets $\bot \subseteq \top \subseteq \mathsf{D}$ satisfying

$$\begin{array}{rcl} \bot & \subseteq & \top \to \bot \\ \bot \to \top & \subseteq & \top. \end{array}$$

A set $\mathcal{A}$ *inhabits* the candidate space if $\bot \subseteq \mathcal{A} \subseteq \top$. Co- and contravariance properties of the semantic function space immediately yield that $\mathcal{A} \to \mathcal{B}$ inhabits the candidate space if $\mathcal{A}$ and $\mathcal{B}$ do. Thus, for fixed $\bot$ and $\top$ we can redefine the set of semantic types to those subsets of D that inhabit the space spanned by $\bot$ and $\top$. A suitable candidate space for untyped NbE is given by

$$\begin{array}{rcl} \top & = & \{d \mid \forall n \exists v \in \mathsf{Nf}.\ \ \mathsf{R}^{\mathsf{nf}}_n\, d \searrow v\} \\ \bot & = & \{e \mid \forall n \exists u \in \mathsf{Ne}.\ \ \mathsf{R}^{\mathsf{ne}}_n\, e \searrow u\}. \end{array}$$

Now every semantic type contains only reifiable values, and all de Bruijn levels, since $\mathsf{R}^{\mathsf{ne}}_n\, \mathsf{x}_k \searrow \mathsf{v}_{n-(k+1)}$ (we round negative indices up to 0). Natural numbers have to be extended to include the reifiable neutrals:

$$\frac{}{\mathsf{zero} \in \mathcal{N}at} \qquad \frac{d \in \mathcal{N}at}{\mathsf{suc}(d) \in \mathcal{N}at} \qquad \frac{e \in \bot}{e \in \mathcal{N}at}.$$

It follows that $\bot \subseteq [\![T]\!]$, thus $\rho_n \in [\![\Gamma]\!]$ if n is the length of Γ. Now $\Gamma \vdash t : T$ implies $[\![t]\!](\rho_n) \in [\![T]\!] \subseteq \top$, thus $\mathsf{R}^{\mathsf{nf}}_n\, ([\![t]\!](\rho_n)) \searrow v$ for some $v \in \mathsf{Nf}$, and we have established the normalization of typable terms. However, neither its soundness nor completeness with regard to System T equality rules (see Fig. 2.2) have been demonstrated yet, so let us work on that in the next section.

3.5 Explicit Substitutions and β-Equality

Normalization should be complete for β-equality: $(\lambda t)\, s$ should have the same normal form as $t[s]$, which denotes the usual substitution of s for de Bruijn index 0 in t. This is already the case if they have the same semantic value, however, with substitution as an *operation* we cannot show this. Consider, for instance, $t = \lambda 1$ and $s = \lambda 0$. Then

$$\begin{array}{lcl} [\![(\lambda t)s]\!](\rho) & = & (\underline{\lambda} t)\rho \cdot [\![s]\!](\rho) = [\![t]\!](\rho, [\![s]\!]\rho) = (\underline{\lambda} 1)(\rho, (\underline{\lambda} 0)\rho) \\ [\![t[s]]\!](\rho) & = & [\![\lambda\lambda 0]\!](\rho) = (\underline{\lambda}\lambda 0)\rho. \end{array}$$

These two values are not equal[8] as objects of D. This can be remedied by switching to explicit substitutions [Abadi et al., 1991] which naturally underlie a category-theoretic treatment of type theory [Dybjer, 1996, Hofmann, 1997]. With explicit substitutions, we change the evaluation order for $[\![t[s]]\!](\rho)$: Instead of substituting s into t and then evaluating the result in environment ρ, the substitution $[s]$ is evaluated in ρ to a new environment $(\rho, [\![s]\!](\rho))$ which is then used to evaluate t. This matches the evaluation order for $[\![(\lambda t)s]\!](\rho)$.

The literature knows different formulations of explicit substitutions;[9] we follow Abadi et al. [1991]:

$$\begin{array}{llcll} \mathsf{Exp} & \ni \; r, s, t & ::= & \cdots \mid t\,\sigma & \text{application of substitution} \\ \\ \mathsf{Subst} & \ni \; \sigma, \tau & ::= & \uparrow & \text{index shift by 1} \\ & & \mid & \mathsf{id} & \text{identity substitution} \\ & & \mid & \sigma\,\tau & \text{substitution composition} \\ & & \mid & (\sigma, s) & \text{substitution extension.} \end{array}$$

The substitution $\uparrow$ (short arrow!) increases all free de Bruijn indices in a term by 1. The identity substitution id does nothing, as expected, and substitution composition $\sigma\,\tau$ has the effect of first substituting with σ and then with τ. Substitution extension (σ, s) is the syntactic analog of environment extension (ρ, a). We write the substitution of s for the 0th index (id, s) as just $[s]$.

Formally, the meaning of substitutions is given by its operational semantics $\boxed{[\![\sigma]\!](\rho) \searrow \rho'}$:

$$\frac{}{[\![\uparrow]\!](\rho, a) \searrow \rho} \qquad \frac{}{[\![\mathsf{id}]\!](\rho) \searrow \rho} \qquad \frac{[\![\tau]\!](\rho) \searrow \rho' \qquad [\![\sigma]\!](\rho') \searrow \rho''}{[\![\sigma\,\tau]\!](\rho) \searrow \rho''}$$

$$\frac{[\![\sigma]\!](\rho) \searrow \rho' \qquad [\![s]\!](\rho) \searrow a}{[\![(\sigma, s)]\!](\rho) \searrow (\rho', a)} \qquad \frac{[\![\sigma]\!](\rho) \searrow \rho' \qquad [\![t]\!](\rho') \searrow a}{[\![t\,\sigma]\!](\rho) \searrow a}$$

The last rule extends the evaluation of expressions to the case of substitution application.

Semantic substitution typing $\boxed{\Gamma \vDash \sigma : \Delta}$ is defined as

$$\Gamma \vDash \sigma : \Delta \;\; :\Longleftrightarrow \;\; \forall \rho \in [\![\Gamma]\!].\;\; [\![\sigma]\!](\rho) \in [\![\Delta]\!]$$

leading to the following new semantic typing rules.

$$\frac{}{\Gamma, S \vDash \uparrow : \Gamma} \qquad \frac{}{\Gamma \vDash \mathsf{id} : \Gamma} \qquad \frac{\Gamma_1 \vDash \tau : \Gamma_2 \qquad \Gamma_2 \vDash \sigma : \Gamma_3}{\Gamma_1 \vDash \sigma\,\tau : \Gamma_3}$$

$$\frac{\Gamma \vDash \sigma : \Delta \qquad \Gamma \vDash s : S}{\Gamma \vDash (\sigma, s) : \Delta, S} \qquad \frac{\Gamma \vDash \sigma : \Delta \qquad \Delta \vDash t : T}{\Gamma \vDash t\,\sigma : T}$$

[8] Although these values are not identical, they are extensionally equal: Both representations of the constant identity function $\lambda x \lambda y.y$. It is not hopeless to try and justify substitution-as-operation, but switching to explicit substitutions seems consequent since we use parallel substitution already in the soundness argument for NbE.

[9] Such as Nadathur's suspension calculus [2002] or Martin-Löf's substitution calculus [1992] or variants that fuse identity and shifting substitution, e. g., Abel and Pientka [2010].

Now we are ready to show that our semantics models β-equality. To this end, we define semantic term equality $\boxed{\Gamma \vDash t = t' : T}$ and semantic substitution equality $\boxed{\Gamma \vDash \sigma = \sigma' : \Delta}$ as follows:

$$\begin{array}{lcl} a = a' \in \mathcal{A} & :\Longleftrightarrow & a \in \mathcal{A} \text{ and } a' \in \mathcal{A} \text{ and } a = a' \\ \Gamma \vDash t = t' : T & :\Longleftrightarrow & \forall \rho \in [\![\Gamma]\!].\ \ [\![t]\!](\rho) = [\![t']\!](\rho) \in [\![T]\!] \\ \\ \rho = \rho' \in [\![\Delta]\!] & :\Longleftrightarrow & \rho \in [\![\Delta]\!] \text{ and } \rho' \in [\![\Delta]\!] \text{ and } \rho = \rho' \\ \Gamma \vDash \sigma = \sigma' : \Delta & :\Longleftrightarrow & \forall \rho \in [\![\Gamma]\!].\ \ [\![\sigma]\!](\rho) = [\![\sigma']\!](\rho) \in [\![\Delta]\!]. \end{array}$$

Figure 3.3 lists valid inferences for term equality $\Gamma \vDash t = t' : T$, yet we have written them as rules for a syntactic judgement $\Gamma \vdash t = t' : T$. Figure 3.4 does the same for substitution equality, which is shown to be a congruence relation: it has equivalence rules and all compatibility rules.

Term equality is only a weak congruence; we are missing the extensionality laws given in Figure 3.5. The reason is that our semantic equality is too intensional for functions. At function type, we only ask for equality of closures, yet we could ask for extensional equality of functions. We shall strengthen our semantics in this way in the next section.

3.6 Extensionality and Partial Equivalence Relations

To model extensional function equality, we need to equip our semantic types $\mathcal{A} \subseteq \mathsf{D}$ with an equivalence relation $=_{\mathcal{A}}$, such that for all $f, f' \in \mathcal{A} \to \mathcal{B}$,

$$f =_{\mathcal{A}\to\mathcal{B}} f' \iff \forall a, a' \in \mathcal{A}.\ \ a =_{\mathcal{A}} a' \implies f \cdot a =_{\mathcal{B}} f' \cdot a'.$$

A set with an equivalence relation is called a *setoid*.[10] However, in our case we deal with a subset (predicate) of D with an equivalence relation. It turns out that this is just a *partial equivalence relation* (PER) on D, i. e., a symmetric and transitive relation $\mathcal{A} \subseteq \mathsf{D} \times \mathsf{D}$. Membership of $a \in \mathcal{A}$ is defined as $(a, a) \in \mathcal{A}$; and if $(a, b) \in \mathcal{A}$ then by symmetry and transitivity both $a \in \mathcal{A}$ and $b \in \mathcal{A}$.

PERs have a groupoid structure [Abel, 2009a]. A groupoid (in the algebraic sense) is a set with a total inverse operation a^{-1} and a *partial* concatenation operation $a * b$. A set of pairs $\mathsf{D} \times \mathsf{D}$ makes a groupoid with $(a, b)^{-1} = (b, a)$ and $(a, b) * (b, c) = (a, c)$. The PERs on D correspond exactly to the subgroupoids of $\mathsf{D} \times \mathsf{D}$, with symmetry corresponding to inversion and transitivity to concatenation.

The view of PERs as subgroupoids lets us reuse our definition of semantic function space for relations. First note that a partial applicative structure D induces a partial applicative structure on $\mathsf{D}^2 = \mathsf{D} \times \mathsf{D}$ with

$$\begin{array}{lcl} (f, f') \cdot (a, a') & = & (f \cdot a,\ f' \cdot a') \\ [\![t]\!](\rho, \rho') & = & ([\![t]\!](\rho),\ [\![t]\!](\rho')) \text{ where } (\rho, \rho')(i) = (\rho(i), \rho'(i)). \end{array}$$

[10] Setoids go back to Bishop [1967] who called a collection of elements a *preset* and a preset with equivalence relation a *set*.

Computation rules (β).

$$\frac{\Gamma, S \vdash t : T \qquad \Gamma \vdash s : S}{\Gamma \vdash (\lambda t)\, s = t[s] : T}$$

$$\frac{\Gamma \vdash t_z : T \qquad \Gamma \vdash t_s : \mathsf{N} \to T \to T}{\Gamma \vdash \mathsf{rec}(t_z, t_s, \mathsf{zero}) = t_z : T} \qquad \frac{\Gamma \vdash t_z : T \qquad \Gamma \vdash t_s : \mathsf{N} \to T \to T \qquad \Gamma \vdash t_n : \mathsf{N}}{\Gamma \vdash \mathsf{rec}(t_z, t_s, \mathsf{suc}(t_n)) = t_s\ t_n\ \mathsf{rec}(t_z, t_s, t_n) : T}$$

Substitution resolution rules.

$$\frac{\Gamma(i) = T}{\Gamma, S \vdash \mathsf{v}_i \uparrow = \mathsf{v}_{i+1} : T} \qquad \frac{\Gamma \vdash t : T}{\Gamma \vdash t\, \mathsf{id} = t : T}$$

$$\frac{\Gamma \vdash \sigma : \Delta \qquad \Gamma \vdash s : S}{\Gamma \vdash \mathsf{v}_0\, (\sigma, s) = s} \qquad \frac{\Gamma \vdash \sigma : \Delta \qquad \Gamma \vdash s : S \qquad \Delta(i) = T}{\Gamma \vdash \mathsf{v}_{i+1}\, (\sigma, s) = \mathsf{v}_i\, \sigma : T}$$

Substitution propagation rules.

$$\frac{\Gamma \vdash \sigma : \Delta \qquad \Delta \vdash r : S \to T \qquad \Delta \vdash s : S}{\Gamma \vdash (r\, s)\, \sigma = (r\, \sigma)\, (s\, \sigma) : T} \qquad \frac{\Gamma_1 \vdash \tau : \Gamma_2 \qquad \Gamma_2 \vdash \sigma : \Gamma_3 \qquad \Gamma_3 \vdash t : T}{\Gamma \vdash (t\, \sigma)\, \tau = t\, (\sigma\, \tau) : T}$$

$$\frac{\Gamma \vdash \sigma : \Delta}{\Gamma \vdash \mathsf{zero}\, \sigma = \mathsf{zero} : \mathsf{N}} \qquad \frac{\Gamma \vdash \sigma : \Delta \qquad \Delta \vdash t : \mathsf{N}}{\Gamma \vdash \mathsf{suc}(t)\, \sigma = \mathsf{suc}(t\, \sigma) : \mathsf{N}}$$

$$\frac{\Gamma \vdash \sigma : \Delta \qquad \Delta \vdash t_z : T \qquad \Delta \vdash t_s : \mathsf{N} \to T \to T \qquad \Delta \vdash t_n : \mathsf{N}}{\Gamma \vdash \mathsf{rec}(t_z, t_s, t_n)\, \sigma = \mathsf{rec}(t_z\, \sigma, t_s\, \sigma, t_n\, \sigma) : T}$$

Compatibility rules.

$$\frac{\Gamma(i) = T}{\Gamma \vdash \mathsf{v}_i = \mathsf{v}_i : T} \qquad \frac{\Gamma \vdash r = r' : S \to T \qquad \Gamma \vdash s = s' : S}{\Gamma \vdash r\, s = r'\, s' : T}$$

$$\frac{}{\Gamma \vdash \mathsf{zero} = \mathsf{zero} : \mathsf{N}} \qquad \frac{\Gamma \vdash t = t' : \mathsf{N}}{\Gamma \vdash \mathsf{suc}(t) = \mathsf{suc}(t') : \mathsf{N}}$$

$$\frac{\Gamma \vdash t_z = t_z' : T \qquad \Gamma \vdash t_s = t_s' : \mathsf{N} \to T \to T \qquad \Gamma \vdash t_n = t_n' : \mathsf{N}}{\Gamma \vdash \mathsf{rec}(t_z, t_s, t_n) = \mathsf{rec}(t_z', t_s', t_n') : T}$$

$$\frac{\Gamma \vdash \sigma = \sigma' : \Delta \qquad \Delta \vdash t = t' : T}{\Gamma \vdash t\, \sigma = t'\, \sigma' : T}$$

Equivalence rules.

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash t = t : T} \qquad \frac{\Gamma \vdash t = t' : T}{\Gamma \vdash t' = t : T} \qquad \frac{\Gamma \vdash t_1 = t_2 : T \qquad \Gamma \vdash t_2 = t_3 : T}{\Gamma \vdash t_1 = t_3 : T}$$

Figure 3.3: $\boxed{\Gamma \vdash t = t' : T}$ Valid term equality rules.

Computation and category laws.

$$\frac{\Gamma \vdash \sigma : \Delta \qquad \Gamma \vdash s : S}{\Gamma \vdash \uparrow(\sigma, s) = \sigma : \Delta} \qquad \frac{\Gamma \vdash \sigma : \Delta}{\Gamma \vdash \mathsf{id}\,\sigma = \sigma : \Delta} \qquad \frac{\Gamma \vdash \sigma : \Delta}{\Gamma \vdash \sigma\,\mathsf{id} = \sigma : \Delta}$$

$$\frac{\Gamma_4 \vdash \sigma_3 : \Gamma_3 \qquad \Gamma_3 \vdash \sigma_2 : \Gamma_2 \qquad \Gamma_2 \vdash \sigma_1 : \Gamma_1}{\Gamma_4 \vdash (\sigma_1\,\sigma_2)\,\sigma_3 = \sigma_1\,(\sigma_2\,\sigma_3) : \Gamma_1}$$

Extensionality.

$$\frac{}{\Gamma, S \vdash \mathsf{id} = (\uparrow, \mathsf{v}_0) : \Gamma, S}$$

Propagation.

$$\frac{\Gamma \vdash \tau : \Gamma' \qquad \Gamma' \vdash \sigma : \Delta \qquad \Gamma' \vdash s : S}{\Gamma \vdash (\sigma, s)\,\tau = (\sigma\,\tau, s\,\tau) : \Delta, S}$$

Compatibility.

$$\frac{}{\Gamma, S \vdash \uparrow = \uparrow : \Gamma} \qquad \frac{}{\Gamma \vdash \mathsf{id} = \mathsf{id} : \Gamma} \qquad \frac{\Gamma \vdash \sigma = \sigma' : \Delta \qquad \Gamma \vdash s = s' : S}{\Gamma \vdash (\sigma, s) = (\sigma', s') : \Delta, S}$$

$$\frac{\Gamma_1 \vdash \sigma = \sigma' : \Gamma_2 \qquad \Gamma_2 \vdash \tau = \tau' : \Gamma_3}{\Gamma_1 \vdash \sigma\,\tau = \sigma'\,\tau' : \Gamma_3}$$

Equivalence rules.

$$\frac{\Gamma \vdash \sigma : \Delta}{\Gamma \vdash \sigma = \sigma : \Delta} \qquad \frac{\Gamma \vdash \sigma = \sigma' : \Delta}{\Gamma \vdash \sigma' = \sigma : \Delta} \qquad \frac{\Gamma \vdash \sigma_1 = \sigma_2 : \Delta \qquad \Gamma \vdash \sigma_2 = \sigma_3 : \Delta}{\Gamma \vdash \sigma_1 = \sigma_3 : \Delta}$$

Figure 3.4: $\boxed{\Gamma \vdash \sigma = \sigma' : \Delta}$ Valid substitution equality rules.

Weak function extensionality (ξ).

$$\frac{\Gamma, S \vdash t = t' : T}{\Gamma \vdash \lambda t = \lambda t' : S \to T}$$

Function extensionality (η).

$$\frac{\Gamma \vdash t : S \to T}{\Gamma \vdash t = \lambda.\, (t \uparrow)\, \mathsf{v}_0 : S \to T}$$

Substitution propagation under λ.

$$\frac{\Gamma \vdash \sigma : \Delta \qquad \Delta, S \vdash t : T}{\Gamma \vdash (\lambda t)\, \sigma = \lambda.\, t\, (\sigma \uparrow, \mathsf{v}_0) : S \to T}$$

Figure 3.5: Extensionality laws.

We marry the concept of applicative structure and groupoid, obtaining *applicative groupoids*, by taking the laws of both structures plus the following distributivity laws:

$$\begin{array}{lcll} (f \cdot a)^{-1} & = & f^{-1} \cdot a^{-1} & \text{if } f \cdot a \text{ def.} \\ (f \cdot a) * (f' \cdot a') & = & (f * f') \cdot (a * a') & \text{if } f \cdot a,\ f' \cdot a',\ f * f',\ a * a' \text{ def.} \end{array}$$

With unchanged definition of the semantic function space as $\mathcal{A} \to \mathcal{B} = \{f \in \mathsf{D}^2 \mid \forall a \in \mathcal{A}.\ f \cdot a \in B\}$, we easily prove that if $\mathcal{A}$ and $\mathcal{B}$ are subgroupoids of D^2, so is the function space $\mathcal{A} \to \mathcal{B}$. We shall revert to the synonymous terminology "PER" in the following. Also, we use the suggestive notation $a = a' \in \mathcal{A}$ for $(a, a') \in \mathcal{A}$.

The PER of natural numbers $\mathcal{N}at$ is given inductively by the following rules.

$$\frac{}{\mathsf{zero} = \mathsf{zero} \in \mathcal{N}at} \qquad \frac{a = a' \in \mathcal{N}at}{\mathsf{suc}(a) = \mathsf{suc}(a') \in \mathcal{N}at} \qquad \frac{\forall n.\ \ \mathsf{R}^{\mathsf{ne}}_n e = \mathsf{R}^{\mathsf{ne}}_n e' \in \mathsf{Ne}}{e = e' \in \mathcal{N}at}$$

A PER of environments $\boxed{\rho = \rho' \in [\![\Delta]\!]}$ can be defined pointwise as follows; we also redefine term and substitution equality.

$$\begin{array}{lcl} \rho = \rho' \in [\![\Delta]\!] & :\Longleftrightarrow & \forall i.\ \Delta(i) = T \implies \rho(i) = \rho'(i) \in [\![T]\!] \\ \Gamma \vDash t = t' : T & :\Longleftrightarrow & \forall \rho = \rho' \in [\![\Gamma]\!].\ [\![t]\!](\rho) = [\![t']\!](\rho') \in [\![T]\!] \\ \Gamma \vDash \sigma = \sigma' : \Delta & :\Longleftrightarrow & \forall \rho = \rho' \in [\![\Gamma]\!].\ [\![\sigma]\!](\rho) = [\![\sigma']\!](\rho') \in [\![\Delta]\!] \end{array}$$

The extensional term equality now models the extensionality laws of Figure 3.5.

However, we are still missing a piece: Semantically equal values do not reify to the same normal form; we are not producing η-long normal forms. For example, consider the second-order type $T = (\mathsf{N} \to \mathsf{N}) \to \mathsf{N} \to \mathsf{N}$ and the semantically equal terms $\Gamma \vdash \lambda 0 = \lambda\lambda.\, 1\, 0 : T$. Both terms are in β-normal form and invariant under (untyped)

normalization. Yet, we would like both to normalize to the long form $\lambda\lambda.\,1\,0$, to be able to completely decide term equality by normalization. To this end, we have to employ type-directed reification again.

3.7 Typed Candidate Spaces and Completeness of NbE

In typed NbE, η-expansion is built into reification and reflection. There, function types are modeled as space of actual functions, thus, there is no concept of neutral at function type. A variable that is reflected into the semantics *has* to become a function, thus, it is semantically η-expanded.

For untyped NbE, the situation is different. Neutral values inhabit each semantic type, hence, we have to take extra efforts to get η-expansion, and thus, unique and η-long normal forms. We have employed two different solutions to η-expansion. In Abel et al. [2008], Abel [2008, 2009a] we have replaced the untyped read-back function by a type-directed *contextual reification* procedure. Type and context, which amount to the types of the read-back term and its free variables, are used to produce an η-long form. While contextual reification does the job, it has a drawback: carrying the context around makes the semantics more complicated and we need a Kripke model.

Coquand had the idea to integrate type-directed reflection and reification process into the semantic world, at the level of values.[11] "Reflection" $\uparrow^T \in \mathsf{D} \to \mathsf{D}$ accounts for the η-expansion of free variables, and "reification" $\downarrow^T \in \mathsf{D} \to \mathsf{D}$ for the η-expansion of values at function type. The idea has been originally formulated for domain-NbE [Abel et al., 2011]:

$$\begin{array}{lll} \uparrow^T & \in & [\mathsf{D} \to \mathsf{D}] \\ \uparrow^{S \to T}(e) & = & \mathsf{Abs}(g) \quad \text{where } g(a) = \uparrow^T(\mathsf{App}(e, \downarrow^S a)) \\ \uparrow^{\mathsf{N}}(e) & = & e \end{array}$$

$$\begin{array}{lll} \downarrow^T & \in & [\mathsf{D} \to \mathsf{D}] \\ \downarrow^{S \to T}(f) & = & \mathsf{Abs}(g) \quad \text{where } g(e) = \downarrow^T(f(\uparrow^S e)) \\ \downarrow^{\mathsf{N}}(\mathsf{zero}) & = & \mathsf{zero} \\ \downarrow^{\mathsf{N}}(\mathsf{suc}(a)) & = & \mathsf{suc}(\downarrow^{\mathsf{N}}(a)) \\ \downarrow^{\mathsf{N}}(e) & = & e \end{array}$$

$$\begin{array}{lll} \uparrow^\Gamma & \in & \mathsf{Env} \\ \uparrow^{()} & = & () \\ \uparrow^{\Gamma,S} & = & (\uparrow^\Gamma, \uparrow^S \mathsf{x}_{|\Gamma|}) \end{array}$$

Normalization becomes $\mathsf{nf}^T_\Gamma(t) = \mathsf{R}^{\mathsf{nf}}_{|\Gamma|}(\downarrow^T(\llbracket t \rrbracket(\uparrow^\Gamma)))$. A slight change is necessary to

[11] This idea is to some extent already present in Danvy's work. Danvy [1996] presents a two-level lambda calculus where reflection and reification mitigate between the two levels. This corresponds to our two levels of *values*: D forms the upper level, and $\mathsf{D}^{\mathsf{ne}}/\mathsf{D}^{\mathsf{nf}}$ form the lower level.

Also, Danvy et al. [2001] implement type-directed reflection and reification for a higher-order presentation of typed normal and neutral (there: atomic) values, using the Haskell type class system. However, they do not present variables x_k in the semantics, and thus, have no read-back procedure: They describe their higher-order abstract syntax as "write-only".

recursion: we have to annotate rec with the type of the result of recursion, in order to η-expand in the case of neutral.

$$\begin{array}{lcl} \mathsf{rec}_T & \in & [\mathsf{D} \times \mathsf{D} \times \mathsf{D} \to \mathsf{D}] \\ \mathsf{rec}_T(a_z, a_s, \mathsf{zero}) & = & a_z \\ \mathsf{rec}_T(a_z, a_s, \mathsf{suc}(a_n)) & = & a_s \cdot a_n \cdot \mathsf{rec}_T(a_z, a_s, a_n) \\ \mathsf{rec}_T(a_z, a_s, \uparrow^{\mathsf{N}} e) & = & \uparrow^T \mathsf{rec}_T(\downarrow^T a_z,\ \downarrow^{\mathsf{N} \to T \to T} a_s,\ e) \end{array}$$

Defunctionalizing the resulting type-aware NbE algorithm, we realize that functional values now have two sources: Besides evaluated λs, there are also reflected variables at function type. These become part of our value grammar, and we also introduce a new syntactic category D^{nf} as the target of reification.

$$\begin{array}{llll} \mathsf{D} & \ni\ a, b, f & ::= & (\underline{\lambda} t)\rho \mid \uparrow^T e \mid \mathsf{zero} \mid \mathsf{suc}(a) \\ \mathsf{D}^{\mathsf{ne}} & \ni\ e & ::= & \mathsf{x}_k \mid e\, d \mid \mathsf{rec}_T(d_z, d_s, e) \\ \mathsf{D}^{\mathsf{nf}} & \ni\ d & ::= & \downarrow^T a \end{array}$$

Reflection and reification themselves have become just markers, indicating that the actual η-expansion still has to be performed during application or reification.

$$\begin{array}{llcl} _ & \cdot\ _ & \in & \mathsf{D} \times \mathsf{D} \to \mathsf{D} \\ (\underline{\lambda} t)\rho & \cdot\ a & = & [\![t]\!](\rho, a) \\ \uparrow^{S \to T} e & \cdot\ a & = & \uparrow^T (e \downarrow^S a) \end{array}$$

$$\begin{array}{lcl} \mathsf{R}^{\mathsf{nf}}_n & \in & \mathsf{D}^{\mathsf{nf}} \to \mathsf{Nf} \\ \mathsf{R}^{\mathsf{nf}}_n\ (\downarrow^{S \to T} f) & = & \lambda.\ \mathsf{R}^{\mathsf{nf}}_{n+1}(\downarrow^T (f \cdot \uparrow^S \mathsf{x}_n)) \\ \mathsf{R}^{\mathsf{nf}}_n\ (\downarrow^{\mathsf{N}}\ \mathsf{zero}) & = & \mathsf{zero} \\ \mathsf{R}^{\mathsf{nf}}_n\ (\downarrow^{\mathsf{N}}\ \mathsf{suc}(a)) & = & \mathsf{suc}(\mathsf{R}^{\mathsf{nf}}_n(\downarrow^{\mathsf{N}} a)) \\ \mathsf{R}^{\mathsf{nf}}_n\ (\downarrow^{\mathsf{N}}\ \uparrow^{\mathsf{N}} e) & = & \mathsf{R}^{\mathsf{ne}}_n(e) \end{array}$$

Read-back of neutrals $\mathsf{R}^{\mathsf{ne}}_n \in \mathsf{D}^{\mathsf{ne}} \to \mathsf{Ne}$ is unchanged, and so is evaluation. Figure 3.6 summarizes defunctionalized type-assignment NbE. It is straightforward to turn the new functional presentation of application and reification into inductive relations $f \cdot a \searrow b$ and $\mathsf{R}^{\mathsf{nf}}_n\, d \searrow v$.

We shall now revisit candidate spaces for the typed NbE. The PERs $\bot \subseteq \mathsf{D}^{\mathsf{ne}} \times \mathsf{D}^{\mathsf{ne}}$ and $\top \subseteq \mathsf{D}^{\mathsf{nf}} \times \mathsf{D}^{\mathsf{nf}}$ characterize the normal values we can read back to syntax, but they are no longer semantic types, since they are not relations on D.

$$\begin{array}{lcl} d = d' \in \top & :\Longleftrightarrow & \forall n.\ \ \mathsf{R}^{\mathsf{nf}}_n d = \mathsf{R}^{\mathsf{nf}}_n d' \in \mathsf{Ne} \\ e = e' \in \bot & :\Longleftrightarrow & \forall n.\ \ \mathsf{R}^{\mathsf{ne}}_n e = \mathsf{R}^{\mathsf{ne}}_n e' \in \mathsf{Nf} \end{array}$$

Formally, the PERs $\bot, \top$ are no longer a candidate space, but we can recover properties that capture the essence of a candidate space. The following implications, written as inference rules, hold for $\bot, \top$:

$$\frac{}{\mathsf{x}_k = \mathsf{x}_k \in \bot} \qquad \frac{e = e' \in \bot \qquad d = d' \in \top}{e\, d = e'\, d' \in \bot}$$

$$\frac{e = e' \in \bot}{\downarrow^{\mathsf{N}} \uparrow^{\mathsf{N}} e = \downarrow^{\mathsf{N}} \uparrow^{\mathsf{N}} e' \in \top} \qquad \frac{\forall e = e' \in \bot.\ \ \downarrow^T (f \cdot \uparrow^S e) = \downarrow^T (f' \cdot \uparrow^S e') \in \top}{\downarrow^{S \to T} f = \downarrow^{S \to T} f' \in \top}$$

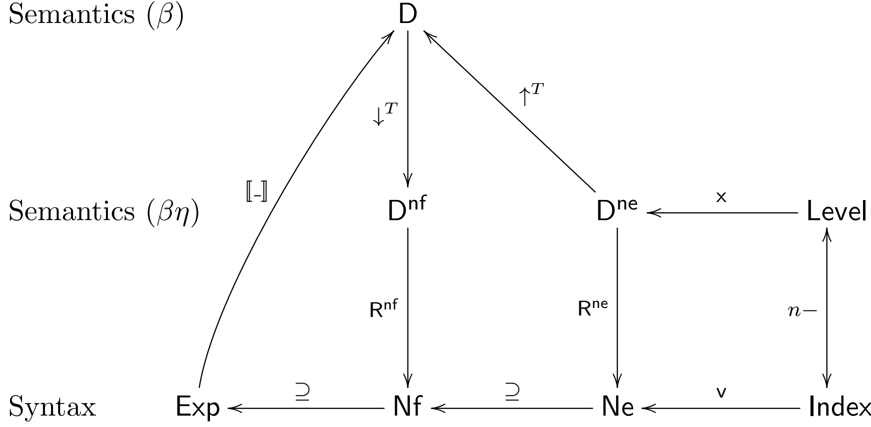

Figure 3.6: Type-assignment NbE in locally nameless style.

Further, we derive the following compatibility rules for natural numbers and recursion:

$$\frac{e = e' \in \bot \qquad d_z = d'_z \in \top \qquad d_s = d'_s \in \top}{\mathsf{rec}_T(d_z, d_s, e) = \mathsf{rec}_T(d'_z, d'_s, e') \in \bot}$$

$$\frac{}{\downarrow^{\mathsf{N}} \mathsf{zero} = \downarrow^{\mathsf{N}} \mathsf{zero} \in \top} \qquad \frac{\downarrow^{\mathsf{N}} a = \downarrow^{\mathsf{N}} a' \in \top}{\downarrow^{\mathsf{N}} \mathsf{suc}(a) = \downarrow^{\mathsf{N}} \mathsf{suc}(a') \in \top}$$

All these inferences are derived using the definition of the read-back functions R^{ne} and R^{nf}, and allow us to reason abstractly about well-defined neutral and normal values.

A *typed candidate space* consists of PERs $\underline{T}, \overline{T}$ for each type T such that $\underline{\mathsf{N}} \subseteq \overline{\mathsf{N}}$ for base type N and

$$\begin{array}{rcl} \underline{S \to T} & \subseteq & \overline{S} \to \underline{T} \\ \underline{S} \to \overline{T} & \subseteq & \overline{S \to T}. \end{array}$$

We say a syntactic type T *realizes* a PER $\mathcal{A}$, written $T \Vdash \mathcal{A}$, if $\underline{T} \subseteq \mathcal{A} \subseteq \overline{T}$. Since $S \Vdash \mathcal{A}$ and $T \Vdash \mathcal{B}$ imply $S \to T \Vdash \mathcal{A} \to \mathcal{B}$, we infer $T \Vdash [\![T]\!]$ from a suitable denotation $\mathsf{N} \Vdash [\![\mathsf{N}]\!]$ of base types.

A concrete typed candidate space for NbE is given by

$$\begin{array}{rcl} a = a' \in \overline{T} & :\Longleftrightarrow & \downarrow^T a = \downarrow^T a' \in \top \\ \uparrow^T e = \uparrow^T e' \in \underline{T} & :\Longleftrightarrow & e = e' \in \bot. \end{array}$$

The candidate space laws follow straightforwardly from the inference rules for $\bot, \top$. To realize $T \Vdash [\![T]\!]$, we slightly redefine $\mathcal{N}at$ as the least PER above $\underline{\mathsf{N}}$ that is closed under zero and suc.

$$\frac{}{\mathsf{zero} = \mathsf{zero} \in \mathcal{N}at} \qquad \frac{a = a' \in \mathcal{N}at}{\mathsf{suc}(a) = \mathsf{suc}(a') \in \mathcal{N}at} \qquad \frac{e = e' \in \bot}{\uparrow^{\mathsf{N}} e = \uparrow^{\mathsf{N}} e' \in \mathcal{N}at}$$

The concrete candidate space models $\uparrow^\Gamma = \uparrow^\Gamma \in [\![\Gamma]\!]$. A consequence of term equality $\Gamma \vdash t = t' : T$ is now that $[\![t]\!](\uparrow^\Gamma) = [\![t']\!](\uparrow^\Gamma) \in [\![T]\!]$, which by $[\![T]\!] \subseteq \overline{T}$ implies $\downarrow^T [\![t]\!](\uparrow^\Gamma) = \downarrow^T [\![t']\!](\uparrow^\Gamma) \in \mathsf{Nf}$ and, thus, $\mathsf{nf}^T_\Gamma(t) = \mathsf{nf}^T_\Gamma(t')$. We have proven that $\beta\eta$-equal terms have the same normal form, meaning that NbE is complete for term equality.

3.8 Restoring Curried Constants

So far, we have treated suc and rec as term constructors $\mathsf{suc}(t)$ and $\mathsf{rec}_T(t_z, t_s, t_n)$ and not as just function symbols that can be partially applied. However, we can implement the latter view by representing the partially applied functions suc, rec_T, $\mathsf{rec}_T\, t_z$ and $\mathsf{rec}_T\, t_z\, t_s$ as values in the semantics.

$$
\begin{array}{llll}
\mathsf{Cst} & \ni & c & ::= \ \mathsf{zero} \mid \mathsf{suc} \mid \mathsf{rec}_T \\
\mathsf{Exp} & \ni & r, s, t & ::= \ \mathsf{v}_i \mid \lambda t \mid r\, s \mid c \\
\mathsf{Nf} & \ni & v & ::= \ u \mid \lambda v \mid \mathsf{zero} \mid \mathsf{suc}\, v \\
\mathsf{Ne} & \ni & u & ::= \ \mathsf{v}_i \mid u\, v \mid \mathsf{rec}_T\, v_z\, v_s\, u \\
\mathsf{D} & \ni & a, b, f & ::= \ (\underline{\lambda} t)\rho \mid \uparrow^T e \mid c \mid \mathsf{suc}(a) \mid \mathsf{rec}_T(a_z) \mid \mathsf{rec}_T(a_z, a_s) \\
\mathsf{D}^{\mathsf{ne}} & \ni & e & ::= \ \mathsf{x}_k \mid e\, d \mid \mathsf{rec}_T(d_z, d_s, e) \\
\mathsf{D}^{\mathsf{nf}} & \ni & d & ::= \ \downarrow^T a
\end{array}
$$

The concepts of normality and neutrality are unaffected by this extension, only evaluation of constants and application of partially applied constants have to be implemented, in the obvious way,

$$
\begin{array}{lll}
[\![c]\!] \qquad\qquad (\rho) & = & c \\
\mathsf{suc} \qquad\qquad \cdot a & = & \mathsf{suc}(a) \\
\mathsf{rec}_T \qquad\qquad \cdot a_z & = & \mathsf{rec}_T(a_z) \\
\mathsf{rec}_T(a_z) \qquad \cdot a_s & = & \mathsf{rec}_T(a_z, a_s) \\
\mathsf{rec}_T(a_z, a_s) \cdot a_n & = & \mathsf{rec}_T(a_z, a_s, a_n)
\end{array}
$$

with $\mathsf{rec}_T(a_z, a_s, a_n) \in \mathsf{D} \times \mathsf{D} \times \mathsf{D} \to \mathsf{D}$ the partial function that executes primitive recursion as defined before.

3.9 Kripke Logical Relations and Soundness of NbE

In the following, we revisit the proof of soundness of normalization, i. e., $\Gamma \vdash t = \mathsf{nf}(t) : T$. Using the shorthand $\downarrow^T_\Gamma t$ for $\mathsf{R}^{\mathsf{nf}}_{|\Gamma|} \downarrow^T t$, the statement of soundness expands to $\Gamma \vdash t = \downarrow^T_\Gamma [\![t]\!](\uparrow^\Gamma) : T$, and we prove it by a logical relation between well-typed terms $\Gamma \vdash t : T$ and values $a \in [\![T]\!]$. This logical relation is *Kripke* which in our context means that it is preserved under context extensions. Since we are using a de Bruijn representation, we have to shift the de Bruijn indices when we extend the context, and this is performed by a *shifting* or *weakening substitution* $\boxed{\sigma : \Gamma' \leq \Gamma}$, a substitution σ that is composed of shifts $\uparrow$.

$$
\frac{}{\mathsf{id} : \Gamma \leq \Gamma} \qquad \frac{\sigma : \Gamma' \leq \Gamma}{\sigma \uparrow : \Gamma', S \leq \Gamma}
$$

In terminology of Kripke semantics, Γ is called a *world* and $\sigma : \Gamma' \leq \Gamma$ the (proof-relevant) *future* relation, stating that Γ' is a future of the world Γ. A world-indexed family of sets $\mathcal{A}$ is *Kripke* if it is monotone in the sense that $a \in \mathcal{A}_\Gamma$ implies $\sigma(a) \in \mathcal{A}_{\Gamma'}$ where $\sigma(a)$ is defined suitably to transport elements a into the future.

For example, the family Tm^T—where $\mathsf{Tm}^T_\Gamma = \{t \mid \Gamma \vdash t : T\}$ the set of terms t that can be assigned type T in context Γ—is Kripke with transport $\sigma(t) = t\,\sigma$ given by shifting. Also, the family $\overline{T}$ defined by

$$\overline{T}_\Gamma = \{\ (t,a) \in \mathsf{Tm}^T_\Gamma \times \mathsf{D} \ \mid\ \Gamma' \vdash t\,\sigma = \downarrow^T_{\Gamma'} a : T \text{ for all } \sigma : \Gamma' \leq \Gamma\ \}$$

is Kripke with transport $\sigma(t,a) = (t\,\sigma, a)$.

We shall now introduce *Kripke typed applicative structures*, or short, *type structures* D^T_Γ [Abel, 2009a] which subsume sets of typed terms, sets of values, PERs, and Kripke logical relations and allow us to prove a general form of the fundamental lemma that can be instantiated to show soundness of NbE as well as validity of typing in PER models (which is part of the completeness proof).

A type structure is a type-indexed Kripke family D^T of sets with a family of partial application operations app, constant interpretations cst, and term interpretations $[\![_]\!]$.

$$\begin{array}{lll} \mathsf{app}^{S\to T}_\Gamma & \in & D^{S\to T}_\Gamma \times D^S_\Gamma \to D^T_\Gamma \\ \mathsf{cst}_\Gamma(c) & \in & D^T_\Gamma \qquad\qquad\qquad \text{for } c : T. \end{array}$$

We write $f \cdot a$ for $\mathsf{app}^{S\to T}_\Gamma(f,a)$ and $\mathsf{cst}(c)$ for $\mathsf{cst}_\Gamma(c)$. Let the type of D-environments D^Δ_Γ be defined by recursion on Δ:

$$\begin{array}{lll} D^{()}_\Gamma & = & () \\ D^{\Delta,S}_\Gamma & = & D^\Delta_\Gamma \times D^S_\Gamma \end{array}$$

Typing and laws for term interpretation $[\![_]\!]$ are then given by:

$$\begin{array}{lll} [\![_]\!]\ \ (_) & \in & \mathsf{Tm}^T_\Gamma \times D^\Gamma_\Delta \to D^T_\Delta \\ [\![c]\!]\ \ (\rho) & = & \mathsf{cst}(c) \\ [\![v_i]\!]\ (\rho) & = & \rho(i) \\ [\![r\,s]\!](\rho) & = & [\![r]\!]\rho \cdot [\![s]\!]\rho \\ [\![\lambda t]\!]\ (\rho) \cdot a & = & [\![t]\!](\rho, a) \end{array}$$

A simple example for a type structure are typed terms $D^T = \mathsf{Tm}^T$. Then app is just term application, cst is the identity, D^Δ_Γ are parallel substitutions $\Gamma \vdash \sigma : \Delta$ with $() : \Gamma \leq ()$ the empty substitution, and evaluation $[\![t]\!](\sigma) = t\,\sigma$ is just application of substitution. Another simple example is the type structure of untyped values $D^T_\Gamma = \mathsf{D}$, with partial application and evaluation as introduced in the previous section.

A *type substructure* E of D is a system of Kripke subsets $E^T_\Gamma \subseteq D^T_\Gamma$ that is closed under the operations of the type structure D. A typical type substructure we might want to consider is $\mathsf{WN}^T_\Gamma \subseteq \mathsf{Tm}^T_\Gamma$, the weakly normalizing terms. If we show that WN is indeed a type substructure of Tm, which implies that application is a total function on well-typed weakly normalizing terms, then we have proven normalization for typed

lambda-calculus. However, establishing the totality of application is non-trivial, it is the essence of the normalization proof. One usually proceeds by strengthening the requirement of weak normalization to that of *computability* [Tait, 1967] aka *reducibility* Girard et al. [1989]. The reducible terms RED^T_Γ form in turn a type substructure of WN, with $\mathsf{RED}^{S\to T}_\Gamma$ defined as the set of terms that act as functions from RED^S_Γ to RED^T_Γ.

To capture the reducibility argument, but also PERs and logical relations for the correctness of NbE, we look at special type substructures we call *induced*. For a given type structure D let $\widehat{D}$ be the family of predicates over D. Using subset notation, we let $\mathcal{A} \in \widehat{D}^T$ iff $\mathcal{A}_\Gamma \subseteq D^T_\Gamma$ for all worlds Γ. We want to define an extensional function space $\mathcal{A} \to \mathcal{B} \in \widehat{D}^{S\to T}$ for $\mathcal{A} \in \widehat{D}^S$ and $\mathcal{B} \in \widehat{D}^T$. The naive pointwise definition $(\mathcal{A} \to \mathcal{B})_\Gamma = \{f \mid \forall a \in \mathcal{A}_\Gamma.\ f \cdot a \in \mathcal{B}_\Gamma\}$ is not Kripke, because the function space is contravariant in domain $\mathcal{A}$. Instead, we employ the *Kripke function space*

$$(\mathcal{A} \to \mathcal{B})_\Gamma = \{\ f \in D^{S\to T}_\Gamma \ \mid\ \forall \sigma : \Gamma' \leq \Gamma,\ a \in \mathcal{A}_{\Gamma'}.\ \ \sigma(f) \cdot a \in \mathcal{B}_{\Gamma'}\ \}$$

with built-in monotonicity. An *induced type substructure* of D is a system $E^T_\Gamma \subseteq D^T_\Gamma$ of subsets such that $\mathsf{cst}(c) \in E^T_\Gamma$ for $c : T$ and

$$E^{S\to T} = E^S \to E^T,$$

i. e., at function type we have the full Kripke function space. We have freedom to choose E^{N} at base type, and this choice spans or *induces* the structure E.

The *fundamental lemma* for Kripke logical relations now simply states that any induced $E \subseteq D$ is actually a type substructure of D. This statement includes the well-definedness of application, which is guaranteed by $E^{S\to T} = E^S \to E^T$, and the well-definedness of evaluation $[\![t]\!] \in E^\Gamma_\Delta \to E^T_\Delta$, which is proven by induction on the typing derivation $\Gamma \vdash t : T$.

As an example application of the fundamental lemma, we consider evaluation of closed typed terms. Our basic type structure are the closed terms $\mathsf{Tm}^T_{()}$ being trivially Kripke since world-independent. Application is total, and interpretation $[\![_]\!] \in \mathsf{Tm}^T_\Gamma \to \mathsf{Tm}^\Gamma_{()} \to \mathsf{Tm}^T_{()}$ closes an open term by substituting closed terms of appropriate type for its free variables. Our goal is to show that every closed term of type N evaluates to a natural number. To this end, we define an induced type structure $\mathsf{RED}^T \subseteq \mathsf{Tm}^T_{()}$, which is necessarily also world-independent, by setting

$$\mathsf{RED}^{\mathsf{N}} = \{t \in \mathsf{Tm}^{\mathsf{N}}_{()} \mid (\!|t|\!) \searrow \mathsf{suc}^n(\mathsf{zero}) \text{ for some } n \in \mathbb{N}\}.$$

Here $(\!|t|\!)$ evaluates closed term t into the set of values D, and it should not be confused with the interpretation $[\![_]\!]$ of open terms into $\mathsf{Tm}_{()}$. At function type we get $\mathsf{RED}^{S\to T} = \{r \in \mathsf{Tm}^{S\to T}_{()} \mid \forall s \in \mathsf{RED}^S.\ r\,s \in \mathsf{RED}^T\}$, which guarantees totality of application. The fundamental lemma now states that RED is a type substructure of $\mathsf{Tm}_{()}$, which entails that the interpretation $[\![t]\!]$ of term $\Gamma \vdash t : T$ is a map from RED^Γ to RED^T. In particular, the interpretation of a closed term of base type $\vdash t : \mathsf{N}$, which is $[\![t]\!]() = t$ the term itself, inhabits RED^N, thus, $(\!|t|\!) \searrow \mathsf{suc}^n(\mathsf{zero})$, *quod erat demonstrandum.*

We shall now apply the fundamental lemma to our original problem, the soundness of NbE. First, we introduce *Kripke candidate spaces* as Kripke families $\underline{T}_\Gamma, \overline{T}_\Gamma$ with

$\underline{\mathsf{N}}_\Gamma \subseteq \overline{\mathsf{N}}_\Gamma$ and

$$\begin{array}{rcl} \underline{S \to T}_\Gamma & \subseteq & (\overline{S} \to \underline{T})_\Gamma \\ (\underline{S} \to \overline{T})_\Gamma & \subseteq & \overline{S \to T}_\Gamma . \end{array}$$

In essence, these are candidate spaces w. r. t. the Kripke function space and pointwise entailment.

Let $D^T_\Gamma = \mathsf{Tm}^T_\Gamma \times \mathsf{D}$ the product of the two type structures of typed terms Tm and untyped values D. A Kripke candidate space on D is defined by

$$\begin{array}{rcl} \overline{T}_\Gamma & = & \{\ (t, a) \quad \mid \forall \sigma : \Gamma' \leq \Gamma. \quad \Gamma' \vdash t\,\sigma = \mathsf{R}^{\mathsf{nf}}_{|\Gamma'|} \downarrow^T a \ :\ T\ \} \\ \underline{T}_\Gamma & = & \{\ (t, \uparrow^T e) \mid \forall \sigma : \Gamma' \leq \Gamma. \quad \Gamma' \vdash t\,\sigma = \mathsf{R}^{\mathsf{ne}}_{|\Gamma'|}\, e \quad :\ T\ \}. \end{array}$$

We interpret types $[\![T]\!]_\Gamma \subseteq D^T_\Gamma$ as sets of term-value pairs. This interpretation is sometimes called *gluing* model [Coquand and Dybjer, 1997, Altenkirch et al., 1995, Abel, 2009a], because it glues syntax (a term) to semantics (a value).

$$\begin{array}{rcl} [\![T]\!] & \in & \widehat{D}^T \\ [\![\mathsf{N}]\!] & = & \overline{\mathsf{N}} \\ [\![S \to T]\!] & = & [\![S]\!] \to [\![T]\!]. \end{array}$$

Since $[\![T]\!]$ is an induced type substructure of D^T, by the fundamental lemma $(t\,\sigma, [\![t]\!](\rho)) \in [\![T]\!]_\Delta$ for any $(\sigma, \rho) \in [\![\Gamma]\!]_\Delta$. In particular, $(t, [\![t]\!](\uparrow^\Gamma)) \in [\![T]\!]_\Gamma \subseteq \overline{T}_\Gamma$. Finally, by definition of the $\overline{T}$ we conclude $\Gamma \vdash t = \downarrow^T_\Gamma [\![t]\!](\uparrow^\Gamma) : T$, the soundness of NbE.

3.10 Summary

We have now completed our development of NbE for type-assignment System T, in preparation for its extension to dependent types and impredicativity. Let us recapitulate the concepts we have reviewed or introduced:

1. *Untyped NbE* [Aehlig and Joachimski, 2004] is an algorithm for β-normalization that captures the intuition and practice that evaluation is independent of types. It can be implemented directly using recursive data types and reasoned about using domain theory [Filinski and Rohde, 2004]. It is universal for all typed lambda-calculi if they are viewed as *type assignment* systems for untyped lambda calculus.

2. *Defunctionalization* [Reynolds, 1972] removes domains, leading to a mutual positive definition of values and environments that has a direct representation in type theory or set theory. It also leads to a mutual definition of application and evaluation as partial functions. Analyzing these partial functions as inductive functional relations takes us from NbE to big-step operational (aka natural) semantics.

3. Modeling types as *sets of values* allows us to semantically reconstruct the typing rules for System T.

4. The restriction of semantic types to a *candidate space* $(\bot, \top)$ proves termination of untyped NbE for simple-type assignment.

5. *Weak term equality* is justified by our subset model when formulated with *explicit substitutions*.

6. *Extensional term equality* requires us to model types as *partial equivalence relations* with extensional function equality.

7. NbE delivers η-long normal forms as we integrate type-directed *reflection* $\uparrow^T$ and *reification* $\downarrow^T$ in defunctionalized form at the level of values. Going from values to long normal forms is separated into semantic η-expansion $\downarrow^T$ followed by read-back into syntax. To show completeness of NbE, we make candidate spaces *typed*, leading to a connection between semantic type and type expression called *realization*.

8. Soundness of NbE finally follows by a Kripke logical relation which we formulate as *type substructure* which inhabits a *Kripke candidate space*. In this second model of the System T typing rules, semantic types are Kripke families of relations between terms and values.

4 Dependent Types

Dependent types allow types to depend on values. We distinguish two kinds of dependencies:

1. Dependent types as *refinements*. In this approach, dependent typing is a refinement of simple (or polymorphic) typing. Each dependent type refines an underlying simple type. For instance, the type of vectors of length n refines the type of lists. The type of numbers below n refines the type of natural numbers. Finally, the type of a safe projection function that takes a number n, a vector of length n, and index $i < n$ and returns the ith element of the vector refines the underlying simple type of a function that takes a number, a list, another number, and possibly returns an element.

Typical examples of refinement type systems are the dependent systems of the lambda cube [Barendregt, 1991] which refine their non-dependent counterpart. For instance, λP, which is roughly the Edinburgh logical framework LF [Harper et al., 1993], refines the simply-typed lambda-calculus.[1] Likewise, the pure Calculus of Constructions [Coquand and Huet, 1988] is a refinement of System F^ω [Girard, 1972]. Other refinement type systems are Dependent ML (DML) [Xi and Pfenning, 1999] and indexed types [Zenger, 1997].

Refinement dependencies are *erasable*. A dependently-typed term can also be assigned the underlying simple (or polymorphic) type. Normalization of terms in a refinement dependent type system can be inherited, via erasure, from the normalization of the corresponding non-dependent system.[2] Thus, refinement dependencies do not add computational power or proof-theoretical strength.[3]

Extending NbE to refinement dependent types is not a challenging task. We simply erase the dependencies and then run NbE for simple types. The shape of the η-long form is determined by its simple type, refinements only affect whether a term was well-typed in the first place. Challenges await us, however, for *full dependent types*.

2. Full dependent types allow us to type terms that have no underlying simple or polymorphic type. The prime example is C's `printf` function whose number and types of arguments depend on its first argument, the format string. There is no single simple type for `printf`, but Augustsson [1999] presents its type in the full dependently-typed language Cayenne.

[1] That LF is just a refinement of the STLC has been exploited heavily in the formal justification of LF [Harper and Pfenning, 2005]: its metatheory is based on an algorithm directed by *simple types* which checks LF terms for $\beta\eta$-equality. The technique of hereditary substitutions [Watkins et al., 2003] for LF-terms, which is directed by their underlying *simple type*, lead to the design of Canonical LF [Harper and Licata, 2007].

[2] For instance, Geuvers [1994] proves normalization for the Calculus of Constructions by erasure to System F^ω.

[3] The computational power or proof-theoretical strength of a system is the highest ordinal needed to prove the termination of all functions definable in this system [Setzer, 1998].

The distinctive feature of full dependent types is the definition of types by computation on a value, for instance, by case distinction or recursion. This feature is sometimes called *large elimination* [Altenkirch, 1994, Werner, 1992] or *strong elimination* [Paulin-Mohring, 1993]. In Chapter 2, we have used large eliminations at the level of the meta language, as we have defined the meaning $[\![T]\!]$ of a type expression T by recursion on T. In this instance, the type expression T plays the role of the value, and the denotation $[\![T]\!] \in \mathsf{Set}$ the type depending on this value.

With types depending on values, computation happens no longer only on the term level, but also on the type level. The meaning of a type should not change under computation, for instance, the type of vectors of length $1 + 1$ should be the type of vectors of length 2.[4] This calls for a notion of evaluation $(\!|T|\!) = A$ of type expressions T into type values A. Type values should be sufficiently evaluated such that their *shape* is apparent, i. e., it is clear whether they represent a function type or a base type.[6] The shape of a type is needed for η-expansion; thus, reflection and reification are directed by type values rather than type expressions. This leads to a definition of NbE for dependently typed terms $\Gamma \vdash t : T$ by

$$\mathsf{nf}^T_\Gamma(t) = \mathsf{R}^{\mathsf{nf}}_{|\Gamma|} \downarrow^A ((\!|t|\!) \uparrow^\Gamma) \text{ where } A = (\!|T|\!) \uparrow^\Gamma .$$

The main challenge in the study of dependently-typed NbE is its semantic justification. With the tools developed in Chapter 3, we are ready to face it.

4.1 A Full Dependently-Typed Language

We consider an extension of System T by dependent function types and predicative universes [Martin-Löf, 1975]. For the sake of brevity, let us refer to this language as PTT, for *predicative type theory*. PTT is a version of Martin-Löf Type Theory [Nordström et al., 1990] cut down to essentials, natural numbers and functions, but equipped with a predicative universe hierarchy Set_k ($k \in \mathbb{N}$) for a streamlined presentation of computation at type level.

As displayed in Figure 4.1, we conceive the language of PTT as untyped lambda calculus Exp in de Bruijn representation with explicit substitutions and a set of constants Cst that lets us express numbers, primitive recursive functions, but also the type of natural numbers N, dependent function types $\mathsf{Fun}\, S\, \lambda T$, and universes Set_k. Fusing term and type language into a common expression language is convenient for full dependent types, as it allows for a single evaluation function for both terms and types.

The form of explicit substitutions $\sigma \in \mathsf{Subst}$ and contexts $\Gamma \in \mathsf{Cxt}$ is unchanged. The β-normal expressions Nf now include also types in normal form. The dependent function type $\mathsf{Fun}\, S\, \lambda T$ is a de Bruijn encoding of $\mathsf{Fun}\, S\, (\lambda x T)$ which is often more

[4] In mathematical practice, 1+1 is *identified* with 2, and the statement $\mathsf{Vec}(1{+}1) = \mathsf{Vec}(2)$ tautological, maybe even alienating, if insisted upon. However, as we consider the semantics of type theory, which is a possible foundation of mathematics, we a priori distinguish the expressions $1 + 1$ and 2 even if they have the same value. It took Russell and Whitehead [1910–1913] more than 300 pages[5] of their *Principia Mathematica* to prove $1 + 1 = 2$, certainly an immortal joke in the history of mathematics.

[5] On page 379 of the first edition of volume I they write: “From this proposition it will follow, when arithmetical addition has been defined, that 1+1=2.”

[6] This means that type values should at least be in *weak head normal form*.

$$
\begin{array}{llll}
\mathsf{Cst} & \ni c & ::= & \mathsf{N} \mid \mathsf{zero} \mid \mathsf{suc} \mid \mathsf{rec} \mid \mathsf{Fun} \mid \mathsf{Set}_k \qquad (k \in \mathbb{N}) \\
\mathsf{Exp} & \ni r, s, t, R, S, T & ::= & c \mid \mathsf{v}_i \mid \lambda t \mid r\, s \mid t\, \sigma \\
\mathsf{Subst} & \ni \sigma, \tau & ::= & \uparrow \mid \mathsf{id} \mid \sigma\, \tau \mid (\sigma, s) \\
\mathsf{Cxt} & \ni \Gamma, \Delta & ::= & () \mid \Gamma, S \\
\mathsf{Nf} & \ni v, w, V, W & ::= & u \mid \mathsf{Fun}\, V\, W \mid \lambda v \mid \mathsf{N} \mid \mathsf{zero} \mid \mathsf{suc}\, v \mid \mathsf{Set}_k \\
\mathsf{Ne} & \ni u, U & ::= & \mathsf{v}_i \mid u\, v \mid \mathsf{rec}_V\, v_z\, v_s\, u
\end{array}
$$

Figure 4.1: Expressions and normal forms of PTT.

legibly written as $(x : S) \to T$. We write $S \to T$ for the non-dependent function space $\mathsf{Fun}\, S\, \lambda. T \uparrow$. To grasp this definition, note that shifting $T \uparrow$ introduces a fake dependency on the 0th variable, which is then abstracted by the λ. Indeed, we have $(\lambda. T \uparrow)\, s = (T \uparrow)\, (\mathsf{id}, s) = T\, (\uparrow (\mathsf{id}, s)) = T\, \mathsf{id} = T$.

Figure 4.2 presents the type assignment rules for PTT. There are several interesting things to note: First, since types contain terms, they are no longer automatically well-formed. The judgement $\boxed{\Gamma \vdash T}$ classifies a type T as well-formed if it inhabits one of the type universes Set_k. Further, contexts are also not automatically well-formed, judgement $\boxed{\vdash \Gamma}$ ensures this. The empty context is of course well-formed, and an extended context Γ, T is well-formed if Γ is well-formed and T is well-formed in Γ. These rules allow later assumptions in Γ to depend on earlier assumptions.

A typical use of this dependency is polymorphism, as in the judgement, written with variable names, $X : \mathsf{Set}_0,\ x : X\ \vdash x : X$. In de Bruijn representation, this becomes the unreadable $\mathsf{Set}_0, \mathsf{v}_0 \vdash \mathsf{v}_0 : \mathsf{v}_1$. The context is well-formed since the type Set_0 of the first assumption is well-formed, and the type v_0 of the second assumption, which refers to the first assumption, is also well-formed since $\mathsf{Set}_0 \vdash \mathsf{v}_0 : \mathsf{Set}_0$. Now the right hand side $\mathsf{v}_0 : \mathsf{v}_1$ of the judgement says that the second assumption is typed by the first assumption. On the right hand side, the type v_0 of the second assumption appears now under two assumptions and not under one, so it has to be shifted to v_1.

The need to shift de Bruijn indices of types when lifting them out of the context is accounted for in variable look-up $\boxed{\Gamma(i) = S}$. In the typing judgement $\boxed{\Gamma \vdash t : T}$ for terms t, dependency is reflected in three places. First, a substitution σ applied to a term t needs also be applied to its type T, since the type also depends on the context. Secondly, when applying a dependent function $r : \mathsf{Fun}\, S\, R$ to an argument s, the codomain R needs also be applied to s, since the codomain itself is a function from domain S to types. Finally, we have a subsumption rule expressing that any term which can be assigned type T can also be assigned type T' as long as $\boxed{\Gamma \vdash T \leq T'}$ meaning that T is a subtype of T' in context Γ. Subtyping is restricted to universe subsumption $\mathsf{Set}_k \leq \mathsf{Set}_l$ and type equality $\boxed{\Gamma \vdash T = T'}$ which means that T and T' are $\beta\eta$-equal expressions inhabiting some universe.

Consider the type assignment $\boxed{c : T}$ for constants. The base type of natural numbers

$\boxed{c : T}$ Constant c can be assigned type T.

$$\begin{array}{lllllll} \mathsf{zero} & : & \mathsf{N} & & \mathsf{N} & : & \mathsf{Set}_k & (k \in \mathbb{N}) \\ \mathsf{suc} & : & \mathsf{N} \to \mathsf{N} & & \mathsf{Fun} & : & \mathsf{Fun}\,\mathsf{Set}_i\,\lambda.\,(\mathsf{v}_0 \to \mathsf{Set}_j) \to \mathsf{Set}_k & (i, j \leq k) \\ \mathsf{rec} & : & \mathsf{Rec}_k \quad (k \in \mathbb{N}) & & \mathsf{Set}_i & : & \mathsf{Set}_j & (i < j) \end{array}$$

$\boxed{\vdash \Gamma}$ Context Γ is well-formed.

$$\frac{}{\vdash ()} \qquad \frac{\vdash \Gamma \qquad \Gamma \vdash T}{\vdash \Gamma, T}$$

$\boxed{\Gamma \vdash T}$ Type T is well-formed in context Γ. $\Gamma \vdash T$Type T is well-formed in context Γ

$$\Gamma \vdash T \;\; :\Longleftrightarrow \;\; \Gamma \vdash T : \mathsf{Set}_k \text{ for some } k \in \mathbb{N}$$

$\boxed{\Gamma(i) = S}$ De Bruijn index i has type S in context Γ.

$$\frac{}{(\Gamma, S)(0) = S\!\uparrow} \qquad \frac{\Gamma(i) = S}{(\Gamma, S)(i+1) = S\!\uparrow}$$

$\boxed{\Gamma \vdash t : T}$ Term t has type T in context Γ.

$$\frac{\vdash \Gamma \qquad c : T}{\Gamma \vdash c : T} \qquad \frac{\Gamma \vdash \sigma : \Delta \qquad \Delta \vdash t : T}{\Gamma \vdash t\,\sigma : T\,\sigma}$$

$$\frac{\vdash \Gamma \qquad \Gamma(i) = S}{\Gamma \vdash \mathsf{v}_i : S} \qquad \frac{\Gamma, S \vdash t : T}{\Gamma \vdash \lambda t : \mathsf{Fun}\,S\,\lambda T} \qquad \frac{\Gamma \vdash r : \mathsf{Fun}\,S\,R \qquad \Gamma \vdash s : S}{\Gamma \vdash r\,s : R\,s}$$

$$\frac{\Gamma \vdash t : T \qquad \Gamma \vdash T \leq T'}{\Gamma \vdash t : T'}$$

$\boxed{\Gamma \vdash T \leq T'}$ Type T is a subtype of T' in context Γ.

$$\frac{k \leq l}{\Gamma \vdash \mathsf{Set}_k \leq \mathsf{Set}_l} \qquad \frac{\Gamma \vdash T = T'}{\Gamma \vdash T \leq T'}$$

$\boxed{\Gamma \vdash T = T'}$ Types T and T' are equal in context Γ.

$$\Gamma \vdash T = T' \;\; :\Longleftrightarrow \;\; \Gamma \vdash T = T' : \mathsf{Set}_k \text{ for some } k \in \mathbb{N}$$

Figure 4.2: Type assignment for PTT.

N inhabits all type universes, and the code Set_i for a universe inhabits all higher universes, resulting in a *cumulative* hierarchy[7] which is also reflected by subtyping. The typing of function type constructor Fun is easier to understand if expanded to a rule:

$$\frac{\Gamma \vdash S : \mathsf{Set}_i \qquad \Gamma, S \vdash T : \mathsf{Set}_j}{\Gamma \vdash \mathsf{Fun}\, S\, \lambda T : \mathsf{Set}_k}\; i, j \leq k$$

This rule captures *predicativity* by requiring that the universe level of a function type is at least the level of domain and codomain. Finally, we define the family of types Rec_k for primitive recursion as the de Bruijn encoding of

$$(P : \mathsf{N} \to \mathsf{Set}_k) \to P\,\mathsf{zero} \to ((x\!:\!\mathsf{N}) \to P\,x \to P\,(\mathsf{suc}\,x)) \to (x\!:\!\mathsf{N}) \to P\,x$$

which is the type of generalized induction for natural numbers. In the form of a rule, we have:

$$\frac{\Gamma \vdash P : \mathsf{N} \to \mathsf{Set}_k \qquad \Gamma \vdash z : P\,\mathsf{zero} \qquad \Gamma, \mathsf{N}, (P\,\mathsf{v}_0) \vdash s : P\,(\mathsf{suc}\,\mathsf{v}_1) \qquad \Gamma \vdash n : \mathsf{N}}{\Gamma \vdash \mathsf{rec}_P\, z\, (\lambda\lambda s)\, n : P\, n}.$$

Since $P\,n$ can be *large*, i. e., not just a base type or higher-order function type, but a universe, rec allows us to define types by recursion on natural numbers. For instance, the type $\mathsf{N}^n \to \mathsf{N}$ of an n-ary curried function is coded as $\mathsf{rec}_{\lambda \mathsf{Set}_0}\, \mathsf{N}\, (\lambda\lambda.\, \mathsf{N} \to \mathsf{v}_0)\, n$.

We have presented the typing rules as an initial orientation in PTT, but defer the equality rules until we can justify them by a model. To spoil the excitement, let us remark that they correspond to the ones of System T, modulo dependent typing.

4.2 Type Values, Reflection and Reification

Evaluation $(\!|_|\!)$ for PTT must include type expressions, thus, we add the type constants to D (subsumed under c), and (partial) applications of the function type constructor Fun. Also, to get closer to expression syntax, we present partial applications of suc, rec, and Fun in curried style. $\mathsf{Base} \subset \mathsf{D}$ classifies the type values that represent the base types: the type of natural numbers N, universes Set_k, and neutral types $\uparrow^{\mathsf{Set}_k} E$.

$$\begin{array}{llll}
\mathsf{D} & \ni & a, b, f, A, F & ::= \; (\underline{\lambda} t)\rho \mid \uparrow^A e \mid c \mid \mathsf{suc}\, a \mid \mathsf{rec}_A \mid \mathsf{rec}_A\, a_z \mid \mathsf{rec}_A\, a_z\, a_s \\
 & & & \quad\;\; \mid \; \mathsf{Fun}\, A \mid \mathsf{Fun}\, A\, F \\
\mathsf{Base} & \ni & B & ::= \; \mathsf{N} \mid \mathsf{Set}_k \mid \uparrow^{\mathsf{Set}_k} E \\
\mathsf{Env} & \ni & \rho & ::= \; () \mid \rho, a \\
\mathsf{D}^{\mathsf{ne}} & \ni & e, E & ::= \; \mathsf{x}_k \mid e\, d \mid \mathsf{rec}_D\, d_z\, d_s\, e \\
\mathsf{D}^{\mathsf{nf}} & \ni & d, D & ::= \; \downarrow^A a
\end{array}$$

The non-dependent function type $A_1 \to A_2$ can be defined as $\mathsf{Fun}\, A_1\, (\underline{\lambda} \mathsf{v}_1)((), A_2)$. Non-dependency is witnessed by $(\underline{\lambda} \mathsf{v}_1)((), A_2) \cdot a = (\!|\mathsf{v}_1|\!)(((), A_2), a) = (\!|\mathsf{v}_0|\!)((), A_2) = A_2$.

A significant change w. r. t. simple types is that reflection and reification are now directed by type *values* A. This has profound a impact on the application and read-back

[7] Cumulative universe hierarchies are part of the type-theoretic proof assistants Coq [Coquand, 1986, INRIA, 2012] and NuPrl [Constable et al., 1986, Allen, 1987].

operations. Application of a reflected neutral $(\uparrow^{\mathsf{Fun}\,A\,F} e)$ at dependent function type leads to instantiation of the function codomain. η-expansion happens here necessarily piece-wise.

$$
\begin{array}{lcll}
(\uparrow^{\mathsf{Fun}\,A\,F} e) \cdot a & = & \uparrow^{F \cdot a}(e\,d) & \text{where } d = \downarrow^A a \\
\mathsf{Fun} \quad \cdot A & = & \mathsf{Fun}\,A & \\
\mathsf{Fun}\,A \quad \cdot F & = & \mathsf{Fun}\,A\,F &
\end{array}
$$

$$
\begin{array}{lcll}
\mathsf{R}^{\mathsf{nf}}_n \downarrow^{\mathsf{Fun}\,A\,F} f & = & \lambda.\ \mathsf{R}^{\mathsf{nf}}_{n+1} \downarrow^{F \cdot a}(f \cdot a) & \text{where } a = \uparrow^A \mathsf{x}_k \\
\mathsf{R}^{\mathsf{nf}}_n \downarrow^{\mathsf{Set}_k} \mathsf{N} & = & \mathsf{N} & \\
\mathsf{R}^{\mathsf{nf}}_n \downarrow^{\mathsf{Set}_k} \mathsf{Set}_i & = & \mathsf{Set}_i & \\
\mathsf{R}^{\mathsf{nf}}_n \downarrow^{\mathsf{Set}_k} \mathsf{Fun}\,A\,F & = & \mathsf{Fun}\ (\mathsf{R}^{\mathsf{nf}}_n \downarrow^{\mathsf{Set}_k} A)\ (\mathsf{R}^{\mathsf{nf}}_n \downarrow^{A \to \mathsf{Set}_k} F) & \\
\mathsf{R}^{\mathsf{nf}}_n \downarrow^{B} \uparrow^{B'} e & = & \mathsf{R}^{\mathsf{ne}}_n e &
\end{array}
$$

When reading back reified neutrals $\downarrow^B \uparrow^{B'} e$ at base type, we allow $B \neq B'$. For one, this saves us a pointless equality check during read-back, for two, since we have cumulative universes, the neutral $\downarrow^{\mathsf{Set}_j} \uparrow^{\mathsf{Set}_i} e$ is legal, at least for $j \geq i$.

The default valuation $\uparrow^\Gamma$ of context Γ by de Bruijn levels reflected at their type now makes use of type evaluation:

$$
\begin{array}{lcll}
\uparrow^{()} & = & () & \\
\uparrow^{\Gamma,S} & = & \rho, \uparrow^{(\!|S|\!)\rho} \mathsf{x}_{|\Gamma|} & \text{where } \rho = \uparrow^\Gamma
\end{array}
$$

The PERs of quotable neutral $e = e' \in \bot$ and normal $d = d' \in \top$ values are type-independent, thus unchanged.

$$
\begin{array}{lcl}
d = d' \in \top & :\Longleftrightarrow & \forall n \exists v \in \mathsf{Nf}.\ \ \mathsf{R}^{\mathsf{nf}}_n\, d \searrow v \text{ and } \mathsf{R}^{\mathsf{nf}}_n\, d' \searrow v \\
e = e' \in \bot & :\Longleftrightarrow & \forall n \exists u \in \mathsf{Ne}.\ \ \mathsf{R}^{\mathsf{ne}}_n\, e \searrow u \text{ and } \mathsf{R}^{\mathsf{ne}}_n\, e' \searrow u
\end{array}
$$

However, the “introduction rules” for $\top$ have to be adapted to dependent types, and we get formation rules for normal type values in $\top$.

$$
\frac{e = e' \in \bot}{\downarrow^{B_1} \uparrow^{B_2} e = \downarrow^{B_1'} \uparrow^{B_2'} e' \in \top}
\qquad
\frac{}{\downarrow^{\mathsf{Set}_k} \mathsf{N} = \downarrow^{\mathsf{Set}_k} \mathsf{N} \in \top}
\qquad
\frac{}{\downarrow^{\mathsf{Set}_k} \mathsf{Set}_i = \downarrow^{\mathsf{Set}_k} \mathsf{Set}_i \in \top}
$$

$$
\frac{\forall e = e' \in \bot.\ \ \downarrow^{F \cdot \uparrow^A e}(f \cdot \uparrow^A e) = \downarrow^{F' \cdot \uparrow^{A'} e'}(f' \cdot \uparrow^{A'} e') \in \top}{\downarrow^{\mathsf{Fun}\,A\,F} f = \downarrow^{\mathsf{Fun}\,A'\,F'} f' \in \top}
$$

The last rule suggests an application operation $(\downarrow^{\mathsf{Fun}\,A\,F} f)(e) = \downarrow^{F \cdot a}(f \cdot a)$ where $a = \uparrow^A e$ for normal function values. With the appropriate notion of function space, it would witness $\bot \to \top \subseteq \top$. In our development, this application operation does not have a formal status, however, it does in formulations of reification on Scott domains [Abel et al., 2011].

4.3 Dependent Function Space and Universes

A distinctive feature of *predicative* type theory is that its semantics can be defined inductively, from below, without reference to syntactic typing.[8] In the following we

[8] In contrast, the impredicative quantification of System F or the CoC, cannot be defined inductively.

revisit the subset model, where semantic types $\mathcal{A} \in \mathcal{P}(\mathsf{D})$ are modeled as sets of values, for dependent types.

Let $\mathcal{A} \in \mathcal{P}(\mathsf{D})$ and $\mathcal{F} \in \mathcal{A} \to \mathcal{P}(\mathsf{D})$, meaning that $\mathcal{F}(a) \subseteq \mathsf{D}$ for each $a \in \mathcal{A}$. The dependent function space $\Pi\,\mathcal{A}\,\mathcal{F} \in \mathcal{P}(\mathsf{D})$ is defined as

$$\Pi\,\mathcal{A}\,\mathcal{F} = \{f \in \mathsf{D} \mid \forall a \in \mathcal{A}.\;\; f \cdot a \in \mathcal{F}(a)\}.$$

Let $\mathcal{N}at \in \mathcal{P}(\mathsf{D})$ be the least set that containts zero and $\uparrow^{\mathsf{N}} e$ for $e \in \bot$ and is closed under application of successor suc.

We will now inductively define a sequence $\mathcal{S}et_0, \mathcal{S}et_1, \ldots$ of *universes* which are semantic types themselves and contain valid codes for types. At the same, time, we define the *extension* functions $\mathcal{E}l_i \in \mathcal{S}et_i \to \mathcal{P}(\mathsf{D})$ that map valid type codes to semantic types. The joint definition of $\mathcal{S}et_i$ and $\mathcal{E}l_i$ is called an *inductive-recursive* definition [Dybjer, 2000].[9]

$$\frac{}{\mathsf{N} \in \mathcal{S}et_k} \qquad \mathcal{E}l_k(\mathsf{N}) = \mathcal{N}at$$

$$\frac{j < k}{\mathsf{Set}_j \in \mathcal{S}et_k} \qquad \mathcal{E}l_k(\mathsf{Set}_j) = \mathcal{S}et_j$$

$$\frac{A \in \mathcal{S}et_k \qquad \forall a \in \mathcal{E}l_k(A).\; F \cdot a \in \mathcal{S}et_k}{\mathsf{Fun}\,A\,F \in \mathcal{S}et_k} \qquad \mathcal{E}l_k(\mathsf{Fun}\,A\,F) = \Pi\,\mathcal{A}\,\mathcal{F}$$

$$\text{where} \quad \mathcal{A} = \mathcal{E}l_k(A), \quad \mathcal{F}(a \in \mathcal{A}) = \mathcal{E}l_k(F \cdot a)$$

This is indeed a cumulative universe hierarchy: For $i \leq j$ and $A \in \mathcal{S}et_i$ we can prove $A \in \mathcal{S}et_j$ and $\mathcal{E}l_j(A) = \mathcal{E}l_i(A)$. Thus, a reasonable limit $\mathcal{S}et_\omega = \bigcup_{i \in \mathbb{N}} \mathcal{S}et_i$ exists with extension function $\mathcal{E}l_\omega(A) = \mathcal{E}l_i(A)$ for $A \in \mathcal{S}et_i$, and we can use extension $\mathcal{E}l_\omega(A)$ uniformly at all universe levels.

The subset model can be used to justify weak β-equality for PTT, however, the absence of extensionality (ξ) does not even allow us to infer $\mathsf{Fun}\,S\,\lambda T = \mathsf{Fun}\,S'\,\lambda T'$ from $S = S'$ and $T = T'$. Therefore, we shall immediately proceed to construct an extensional model based on PERs.

4.4 A PER Model

We return to model semantic types as PERs in order to justify extensional term equality. Along the way, we also add the neutral types to our universes to prepare for reasoning about NbE.

Let Rel denote the set of relations on D and $\mathsf{Per} \subseteq \mathsf{Rel}$ the PERs on D. Let $\mathcal{A} \in \mathsf{Rel}$. By abuse of notation, we write $a \in \mathcal{A}$ if a is in the domain of relation $\mathcal{A}$, meaning that $(a, a') \in \mathcal{A}$ or $(a', a) \in \mathcal{A}$ for some $a' \in \mathsf{D}$. For PERs $\mathcal{A}$, the statement $a \in \mathcal{A}$ is equivalent to $(a, a) \in \mathcal{A}$, since a PER is reflexive on its domain.

[9] This inductive-recursive definition can be reduced to an inductive definition, for details see Abel and Coquand [2007], Abel et al. [2007a, 2008].

Writing $\mathcal{F} \in \mathcal{A} \to \mathsf{Rel}$ shall include the requirement that $\mathcal{F}$ respects relation $\mathcal{A}$, meaning that $\mathcal{F}(a) = \mathcal{F}(a')$ for all $(a, a') \in \mathcal{A}$. The dependent function space $\Pi\,\mathcal{A}\,\mathcal{F} \in \mathsf{Rel}$ is then given by

$$\Pi\,\mathcal{A}\,\mathcal{F} = \{(f, f') \mid \forall (a, a') \in \mathcal{A}.\ (f \cdot a,\ f' \cdot a') \in \mathcal{F}(a)\}.$$

If $\mathcal{A}$ is a PER and $\mathcal{F} \in \mathcal{A} \to \mathsf{Per}$, then $\Pi\,\mathcal{A}\,\mathcal{F}$ is also a PER. Neutral types E are interpreted by PERs of neutral terms $\underline{E} = \{(\uparrow^{E_1} e_1, \uparrow^{E_2} e_2) \mid E_1 = E_2 = E \in \bot \text{ and } e_1 = e_2 \in \bot\}$. Using PER $\mathcal{N}at$ as defined just before Section 3.8, we construct a universe hierarchy $\mathcal{S}et_k$ of PERs analogously to the subset hierarchy in the last section.

$$\frac{E = E' \in \bot}{\uparrow^{\mathsf{Set}_k} E = \uparrow^{\mathsf{Set}_k} E' \in \mathcal{S}et_k} \qquad \mathcal{E}l_k(E) = \underline{E}$$

$$\frac{}{\mathsf{N} = \mathsf{N} \in \mathcal{S}et_k} \qquad \mathcal{E}l_k(\mathsf{N}) = \mathcal{N}at$$

$$\frac{j < k}{\mathsf{Set}_j = \mathsf{Set}_j \in \mathcal{S}et_k} \qquad \mathcal{E}l_k(\mathsf{Set}_j) = \mathcal{S}et_j$$

$$\frac{A = A' \in \mathcal{S}et_k \qquad \forall a = a' \in \mathcal{E}l_k(A).\ F \cdot a = F' \cdot a' \in \mathcal{S}et_k}{\mathsf{Fun}\,A\,F = \mathsf{Fun}\,A'\,F' \in \mathcal{S}et_k} \qquad \begin{aligned} \mathcal{E}l_k(\mathsf{Fun}\,A\,F) &= \Pi\,\mathcal{A}\,\mathcal{F} \\ \text{where}\quad \mathcal{A} &= \mathcal{E}l_k(A) \\ \mathcal{F}(a \in \mathcal{A}) &= \mathcal{E}l_k(F \cdot a) \end{aligned}$$

Simultaneously with the definition, we also prove that the universes $\mathcal{S}et_k$ are indeed PERs, and that the extension functions $\mathcal{E}l_k$ respect PER-equality. We write $\mathcal{T}ype$ for the limit Set_ω and $[_]$ for the extension function $\mathcal{E}l_\omega$.

In the following, let us write $(\!|_|\!)__ \in \mathsf{Exp} \times \mathsf{Env} \to \mathsf{D}$ for the partial function that performs evaluation of expressions into D. Type interpretation $[\![T]\!]\rho = [(\!|t|\!)\rho]$ of type expressions T as semantic types is the composition of evaluation $(\!|_|\!)$ and extension $[_]$.

Each well-formed context induces a PER on environments. We simultaneously define valid contexts $\models \Gamma$ and their semantic extensions $\rho = \rho' \in [\![\Gamma]\!]$ by induction on the length of Γ. Empty contexts $\models ()$ are trivially valid and relate the empty environment to itself, $() = () \in ()$.

$$\begin{aligned} \models \Gamma, T \quad &:\Longleftrightarrow \quad \models \Gamma \text{ and } \forall \rho = \rho' \in \Gamma.\ (\!|T|\!)\rho = (\!|T|\!)\rho' \in \mathcal{T}ype \\ (\rho, a) = (\rho', a') \in [\![\Gamma, T]\!] \quad &:\Longleftrightarrow \quad \rho = \rho' \in [\![\Gamma]\!] \text{ and } a = a' \in [\![T]\!]\rho \end{aligned}$$

Semantic typing and equality judgements are defined as for System T, only that we have to take special attention to ensure semantic well-formedness of types.

$$\begin{aligned} \Gamma \models T \quad &:\Longleftrightarrow \quad \models \Gamma \text{ and } \Gamma \models T : \mathsf{Set}_k \text{ for some } k \\ \Gamma \models t : T \quad &:\Longleftrightarrow \quad \Gamma \models t = t : T \\ \Gamma \models t = t' : T \quad &:\Longleftrightarrow \quad \Gamma \models T \text{ and } \forall \rho = \rho' \in [\![\Gamma]\!].\ (\!|t|\!)\rho = (\!|t'|\!)\rho' \in [\![T]\!]\rho \\ \Gamma \models \sigma : \Delta \quad &:\Longleftrightarrow \quad \Gamma \models \sigma = \sigma : \Delta \\ \Gamma \models \sigma = \sigma' : \Delta \quad &:\Longleftrightarrow \quad \models \Gamma \text{ and } \models \Delta \text{ and } \forall \rho = \rho' \in [\![\Gamma]\!].\ (\!|\sigma|\!)\rho = (\!|\sigma'|\!)\rho' \in [\![\Delta]\!] \end{aligned}$$

Semantic typing justifies the syntactic typing rules of Figure 4.2, and semantic equality the term equality rules of Figures 3.3 to 3.5, suitably modified for dependent types.[10]

4.5 Dependently-Typed Candidate Spaces and Completeness of NbE

A *dependently-typed candidate space* for a universe $\mathcal{U}$ is a pair of semantic types $\underline{A}, \overline{A}$ for each type code $A \in \mathcal{U}$ such that

1. $\underline{(\,)}$ and $\overline{(\,)}$ respect equality in $\mathcal{U}$, i. e., if $A = A' \in \mathcal{U}$ then $\underline{A} = \underline{A'}$ and $\overline{A} = \overline{A'}$,
2. $\underline{B} \subseteq \overline{B}$ for all base types $B \in \mathcal{U}$, and
3. for all function types $\mathsf{Fun}\,A\,F \in \mathcal{U}$ we have

$$\begin{array}{rcl} \underline{\mathsf{Fun}\,A\,F} & \subseteq & \Pi\,\overline{A}\,\underline{F} \\ & & \Pi\,\underline{A}\,\overline{F} \quad \subseteq \quad \overline{\mathsf{Fun}\,A\,F} \end{array}$$

where $\underline{F}$ and $\overline{F}$ are defined by

$$\begin{array}{rclcrcl} \underline{F} & \in & \overline{A} \to \mathsf{Per} & \qquad & \overline{F} & \in & \underline{A} \to \mathsf{Per} \\ \underline{F}(a) & = & \underline{F \cdot a} & & \overline{F}(a) & = & \overline{F \cdot a}. \end{array}$$

We say type code $A \in \mathcal{U}$ *realizes* a semantic type $\mathcal{A}$, written $A \Vdash \mathcal{A}$, if $\underline{A} \subseteq \mathcal{A} \subseteq \overline{A}$. By induction on $A \in \mathcal{S}et_k$ we prove that $A \Vdash [A]$.

The candidate space that guarantees completeness of NbE is given by the family of PERs

$$\begin{array}{rcl} a_1 = a_2 \in \overline{A} & :\Longleftrightarrow & \downarrow^{A_1} a_1 = \downarrow^{A_2} a_2 \in \top \text{ for all } A_1 = A_2 = A \in \mathcal{S}et_k \\ \uparrow^{A_1} e_1 = \uparrow^{A_2} e_2 \in \underline{A} & :\Longleftrightarrow & e = e' \in \bot \text{ and } A_1 = A_2 = A \in \mathcal{S}et_k \end{array}$$

for all k and $A \in \mathcal{S}et_k$. Condition 1 of dependently-typed candidate spaces, $\underline{(\cdot)}, \overline{(\cdot)} \in \mathcal{U} \to \mathsf{Per}$ holds since we built have invariance under type equality into the definition. Condition 2, $\underline{B} \subseteq \overline{B}$ for base types B, which are neutral types E, the natural number type N, and universes Set_i, is satisfied since $\mathsf{R}^{\mathsf{nf}}_n \downarrow^{B'} \uparrow^{B''} e = \mathsf{R}^{\mathsf{ne}}_n e$ for all neutrals $e \in \mathsf{D}^{\mathsf{ne}}$.

Condition 3 follows from condition 1 and the properties of reflection and reification at function types. To prove property $\underline{\mathsf{Fun}\,A\,F} \subseteq \Pi\,\overline{A}\,\underline{F}$, assume $e_1 = e_2 \in \bot$, hence, $\uparrow^{\mathsf{Fun}\,A_1\,F_1} e = \uparrow^{\mathsf{Fun}\,A_2\,F_2} e' \in \underline{\mathsf{Fun}\,A\,F}$ for $\mathsf{Fun}\,A_1\,F_1 = \mathsf{Fun}\,A_2\,F_2 = \mathsf{Fun}\,A\,F \in \mathcal{S}et_k$. Assume further $a_1 = a_2 \in \overline{A}$ and show $\uparrow^{F_1 \cdot a_1}(e_1 \downarrow^{A_1} a_1) = \uparrow^{F_2 \cdot a_2}(e_2 \downarrow^{A_2} a_2) \in \underline{F \cdot a_1}$. With $d_1 := \downarrow^{A_1} a_1$ and $d_2 := \downarrow^{A_2} a_2$ we have $d_1 = d_2 \in \top$, hence, $e_1\,d_1 = e_2\,d_2 \in \bot$. It follows that $\uparrow^{F_1 \cdot a_1}(e_1\,d_1) = \uparrow^{F_2 \cdot a_2}(e_2\,d_2) \in \underline{F \cdot a_1}$, since $F_1 \cdot a_2 = F_2 \cdot a_2 = F \cdot a_1 \in \mathcal{S}et_k$.

For property $\Pi\,\underline{A}\,\overline{F} \subseteq \overline{\mathsf{Fun}\,A\,F}$, assume $f_1 = f_2 \in \Pi\,\underline{A}\,\overline{F}$ and show $\downarrow^{\mathsf{Fun}\,A_1\,F_1} f_1 = \downarrow^{\mathsf{Fun}\,A_2\,F_2} f_2 \in \top$ for $\mathsf{Fun}\,A_1\,F_1 = \mathsf{Fun}\,A_2\,F_2 = \mathsf{Fun}\,A\,F \in \mathcal{S}et_k$. It is sufficient to assume

[10] Adapting term equality to dependent types concerns only the typing, not the untyped equations themselves. Dependent typing for term and substitution equality has been given in Abel et al. [2008], substitution typing for a de Bruijn representation in Abel et al. [2011], Abel [2010a].

$e_1 = e_2 \in \bot$ and prove $\downarrow^{F_1 \cdot a_1}(f_1 \cdot a_1) = \downarrow^{F_2 \cdot a_2}(f_2 \cdot a_2) \in \top$ for $a_1 = \uparrow^{A_1} e_1$ and $a_2 = \uparrow^{A_2} e_2$. Since $a_1 = a_2 \in \underline{A}$, the goal follows from $f_1 \cdot a_1 = f_2 \cdot a_2 \in \overline{F \cdot a_1}$.

In previous works [Abel et al., 2007a,b, 2011] we did not introduce the concept of typed candidate space, but proved these statements about reflection and reification by induction on $A = A' \in \mathcal{S}et_k$:

1. If $e = e' \in \bot$ then $\uparrow^{A} e = \uparrow^{A'} e' \in [A]$.
2. If $a = a' \in [A]$ then $\downarrow^{A} a = \downarrow^{A'} a' \in \top$.

Typed candidate spaces capture the essence of this proof, separating it from the induction on types, and thus, preparing for the treatment of impredicativity. However, note that the definition of typed candidate spaces relies on a semantic type equality $A = A' \in \mathcal{T}ype$.

Completeness of NbE now follows. First, we establish $\uparrow^{\Gamma} = \uparrow^{\Gamma} \in [\![\Gamma]\!]$. Then, by soundness of term equality, $\Gamma \vdash t = t' : T$ entails $a = a' \in [A]$ with $A = (\!|T|\!)(\uparrow^{\Gamma})$ and $a = (\!|t|\!)(\uparrow^{\Gamma})$ and $a' = (\!|t'|\!)(\uparrow^{\Gamma})$. Since $[A] \subseteq \overline{A}$, we infer $\downarrow^{A} a = \downarrow^{A} a' \in \top$ which entails $\mathsf{nf}(t) = \mathsf{nf}(t') \in \mathsf{Nf}$.

4.6 Dependent Function Space on Groupoids

A small nuisance in our model of dependent types is the redefinition of the dependent function space $\Pi\,\mathcal{A}\,\mathcal{F}$ for PERs. We would prefer to just have one definition of Π relative to an applicative structure, and instantiate it to subset and PER models. In Section 3.6 we noted that the PERs on D correspond to the subgroupoids of D^2, allowing us to obtain the PER function space as an instance of the concept of function space for applicative groupoids. In the following, we extend this development to the dependent function space.

Let $(G,\ _\cdot_,\ _^{-1},\ _*_)$ denote an applicative groupoid and Agd denote the set of applicative subgroupoids of G. Let $\mathcal{A} \in \mathsf{Agd}$. We say $\mathcal{F} \in G \to \mathsf{Agd}$ *respects* $\mathcal{A}$ and write $\mathcal{F} \in \mathcal{A} \to \mathsf{Agd}$ if $\mathcal{F}$ is oblivious of groupoid operations: $\mathcal{F}(a) = \mathcal{F}(a^{-1}) = \mathcal{F}(a * b)$ for all $a, b \in \mathcal{A}$ such that $a * b$ is defined. The intention behind this is to model invariance of $\mathcal{F}$ under PER-equality, with inversion a^{-1} corresponding to symmetry and composition $a * b$ corresponding to transitivity.

Given $\mathcal{A} \in \mathsf{Agd}$ and $\mathcal{F} \in \mathcal{A} \to \mathsf{Agd}$, the dependent function space $\Pi\,\mathcal{A}\,\mathcal{F}$ is defined by

$$\Pi\,\mathcal{A}\,\mathcal{F} = \{f \in G \mid \forall a \in \mathcal{A}.\ \ f \cdot a \in \mathcal{F}(a)\}$$

and we show $\Pi\,\mathcal{A}\,\mathcal{F} \in \mathsf{Agd}$ by the laws of applicative groupoids. For instance, $\Pi\,\mathcal{A}\,\mathcal{F}$ is closed under composition: Let $f, g \in \Pi\,\mathcal{A}\,\mathcal{F}$ such that $f * g$ is defined (in G) and show $f * g \in \Pi\,\mathcal{A}\,\mathcal{F}$. To this end, assume $a \in \mathcal{A}$ and show $(f * g) \cdot a \in \mathcal{F}(a)$. First, because $a * a^{-1} \in \mathcal{A}$ by the groupoid laws, we have $f \cdot (a * a^{-1}) \in \mathcal{F}(a * a^{-1}) = \mathcal{F}(a)$. Secondly, $g \cdot a \in \mathcal{F}(a)$. Note that by the distributivity laws of applicative groupoids, $(f \cdot (a * a^{-1})) * (g \cdot a) = (f * g) \cdot ((a * a^{-1}) * a) = (f * g) \cdot a$, since $f * g$ and $(a * a^{-1}) * a$ and $f \cdot (a * a^{-1})$ and $g \cdot a$ are all defined. Since $\mathcal{F}(a)$ is a subgroupoid of G, and $(f \cdot (a * a^{-1})) * (g \cdot a)$ is defined, $(f \cdot (a * a^{-1})) * (g \cdot a) \in \mathcal{F}(a)$, entailing $(f * g) \cdot a \in \mathcal{F}(a)$.

The dependent function space construction allows us to model pure predicative type theory in any applicative groupoid G with constants Set_k and Fun. Constants c in an

applicative groupoid are self-inverse and idempotent in the sense that $c^{-1} = c$ and $c*c = c$. By this, we capture the mapping of constants c to (c, c) in the "mother groupoid" D^2. In D^2 it also holds that different constants do not compose, i. e., $(c, c) * (c', c')$ is undefined for $c \neq c'$; adding this requirement would model the distinctiveness of constants in G.

The definition of semantic term equality $\Gamma \vDash t = t' : T$ calls for a refinement of applicative groupoids. In terms of a PER model, we express semantic equality via $(\!|t|\!)\rho = (\!|t'|\!)\rho' \in [\![T]\!]\rho$. Now, working in an applicative structure D^2 of value pairs, each interpreted term $(\!|t|\!)\rho$ is already a pair, but $[\![T]\!]\rho$ is a set of pairs and not of pairs of pairs. To address this, we add a partial *merge* operation $a \bowtie a'$ to applicative groupoids which on D^2 is a total operation given by $(a_1, a_2) \bowtie (b_1, b_2) = (a_1, b_2)$. In general, $\bowtie$ is not total; closing a PER $\mathcal{A}$ under $\bowtie$ *squashes*[11] the PER structure of $\mathcal{A}$ since it equates all elements which are in the domain of $\mathcal{A}$. The introduction of $\bowtie$ allows us to formulate semantic term equality in the following way: An environment ρ of value pairs can be viewed as a pair of environments (ρ_1, ρ_2) of single values. Then, $(\!|t|\!)\rho = ((\!|t|\!)\rho_1, (\!|t|\!)\rho_2)$ and $(\!|t|\!)\rho \bowtie (\!|t'|\!)\rho' = ((\!|t|\!)\rho_1, (\!|t'|\!)\rho'_2)$, thus, we can express a relationship between the interpretation of t and the interpretation of t'. The axiomatization of $\bowtie$ is inspired by its definition for D^2:

1. Extension of $*$: If $a * b$ is defined, then $a \bowtie b$ is defined and $a \bowtie b = a * b$.
2. Idempotency: $a \bowtie a$ is always defined and $a \bowtie a = a$.
3. Merge chaining (associativity and fusion): If $a \bowtie b$ and $b \bowtie c$ are defined, then $a \bowtie (b \bowtie c) = (a \bowtie b) \bowtie c = a \bowtie c$ are all defined and equal. Thus, the value of a merge chain depends only on its first and last element.
4. Inversion: If $a \bowtie b$ is defined then $(a \bowtie b)^{-1} = b^{-1} \bowtie a^{-1}$ are defined and equal.
5. Compatibility with composition:
    a) If $a \bowtie b$ and $b * c$ are defined, then $a \bowtie (b * c) = (a \bowtie b) * c = a \bowtie c$ are all defined and equal.
    b) If $a * b$ and $b \bowtie c$ are defined, then $a * (b \bowtie c) = (a * b) \bowtie c = a \bowtie c$ are all defined and equal.
6. Distribution over application:
    a) If $a \bowtie b$ and $f \cdot a$ and $f \cdot b$ are defined, then $f \cdot (a \bowtie b) = (f \cdot a) \bowtie (f \cdot b)$ are defined and equal.
    b) If $f \bowtie g$ and $f \cdot a$ and $g \cdot a$ are defined, then $(f \bowtie g) \cdot a = (f \cdot a) \bowtie (g \cdot a)$ are defined and equal.

We call an applicative groupoid with merge a *mergable applicative groupoid* (MAG). Since we intend substructures of D^2, which has a total merge operation, to have only partial merging, we cannot employ the default algebraic substructure concept which would then require that $a \bowtie b \in \mathcal{A}$ for all $a, b \in \mathcal{A}$. Instead, we consider *applicative*

[11] The merge operation $\bowtie$ could be useful in models of *squash types* that have been introduced to dependent type theory in the context of Nuprl Constable et al. [1986].

subgroupoids with merge $\mathcal{A} \in \mathsf{Mag}$ which are applicative subgroupoids of "mother" G and are MAGs, i. e., they fulfill the laws for merge given above.

On MAGs $\mathcal{A}$ we can define a PER structure $a \sim a' \in \mathcal{A}$ by

$$a \sim b \in \mathcal{A} \;:\Longleftrightarrow\; a, b, a \bowtie b \in \mathcal{A}.$$

This is indeed a PER. Transitivity is easy, it follows from the chaining laws for $\bowtie$. For symmetry, we have to derive $b \bowtie a \in \mathcal{A}$. Note that $(a \bowtie b)^{-1} = b^{-1} \bowtie a^{-1} \in \mathcal{A}$ by closure under inversion. Also $b * b^{-1}, a^{-1} * a \in \mathcal{A}$ by the subgroupoid laws. Since $\bowtie$ extends $*$, we have also $b \bowtie b^{-1}, a^{-1} \bowtie a \in \mathcal{A}$ and $b \bowtie b^{-1} \bowtie a^{-1} \bowtie a = b \bowtie a \in \mathcal{A}$ by merge chaining. Semantic term equality is then defined using the PER structure as

$$\Gamma \vDash t = t' : T \;:\Longleftrightarrow\; \forall \rho \sim \rho' \in [\![\Gamma]\!].\; (\!|t|\!)\rho \sim (\!|t'|\!)\rho' \in [\![T]\!]\rho.$$

With the introduction of $\bowtie$ we have to update the notion of respect. For $\mathcal{A} \in \mathsf{Mag}$ by $\mathcal{F} \in \mathcal{A} \to \mathsf{Mag}$ we now mean that $\mathcal{F}$ is also invariant under merge, so we have $\mathcal{F}(a) = \mathcal{F}(a^{-1}) = \mathcal{F}(a * b) = \mathcal{F}(a \bowtie b)$ whenever the mentioned MAG elements are defined. In this case, we also say that $\mathcal{F}$ is an *extension function*, a well-defined mapping of type codes to semantic types.

In the following let Mag more concretely denote the applicative subgroupoids with merge of D^2. Additional laws to integrate the groupoid and merge operations with recursion are necessary, but they can be defined in analogy to application.

Reflection and reification can be defined on D^2 as on D, using the pointwise construction of function type codes $\mathsf{Fun}\,(A, A')\,(F, F') = (\mathsf{Fun}\,A\,F, \mathsf{Fun}\,A'\,F')$ and reflected neutrals $\uparrow^{(A,A')}(e, e') = (\uparrow^{A} e, \uparrow^{A'} e')$ and all the other constructs of $\mathsf{D}, \mathsf{D}^{\mathsf{ne}}, \mathsf{D}^{\mathsf{nf}}$, excluding only function closures $(\underline{\lambda} t)\rho$. Read-back from value pairs to a term shall be undefined if the two individual values result in different terms, otherwise it is defined as for a single value.

A dependently-typed candidate space $\underline{(\cdot)}, \overline{(\cdot)}$ for a universe $\mathcal{U}$ requires now that $\underline{(\cdot)}, \overline{(\cdot)} \in \mathcal{U} \to \mathsf{Mag}$ are extension functions, $\underline{B} \subseteq \overline{B}$ for base types $B \in \mathcal{U}$ and

$$\begin{array}{rcl} \underline{\mathsf{Fun}\,A\,F} & \subseteq & \Pi\,\overline{A}\,\underline{F} \\ \Pi\,\underline{A}\,\overline{F} & \subseteq & \overline{\mathsf{Fun}\,A\,F} \end{array}$$

for function types $\mathsf{Fun}\,A\,F \in \mathcal{U}$. Herein $\underline{F}$ and $\overline{F}$ are required to be extension functions defined by

$$\begin{array}{rclcrcl} \underline{F} & \in & \overline{A} \to \mathsf{Mag} & \quad & \overline{F} & \in & \underline{A} \to \mathsf{Mag} \\ \underline{F}(a) & = & \underline{F \cdot a} & & \overline{F}(a) & = & \overline{F \cdot a}. \end{array}$$

This is only a sketch of the groupoid approach to PER models.[12] It seems worth to pursue it further in future work, because working uniformly with groupoid elements (instead of sometimes with values and sometimes with pairs of values) cuts down the design space for semantic concepts and the amount of details.

[12] Our (algebraic) groupoids model *definitional* equality and are not to be confused with the (categorical) groupoids of Hofmann and Streicher [1994] that model *propositional* equality.

4.7 Kripke Logical Relations for Dependent Types and Soundness of NbE

As demonstrated in Section 2.6, the soundness of NbE can be established by a Kripke logical relation between well-typed terms $\Gamma \vdash t : T$ and values $a \in [\![T]\!]$, defined by induction on type T. In case of dependent types, we cannot just do induction on type expressions, but we can induct on type values $A \in \mathcal{S}et_k$ instead. This induction is the privilege of *predicative* type theory where types are obtained from below. We recapitulate the proof given in Abel et al. [2007a,b, 2011].

By induction on $A \in \mathcal{S}et_k$ we define a relation $\boxed{\Gamma \vdash T \circledR A}$ between well-formed types $\Gamma \vdash T : \mathsf{Set}_k$ and type values $A \in \mathcal{S}et_k$ and a relation $\boxed{\Gamma \vdash t : T \circledR a \in A}$ between well-typed terms $\Gamma \vdash t : T$ and values $a \in [A]$. When using context extension $\sigma : \Gamma' \leq \Gamma$, we assume Γ' to be well-formed.

$$\begin{array}{ll} \Gamma \vdash R \circledR \mathsf{Fun}\, A\, F & \\ \quad :\iff & \Gamma \vdash R = \mathsf{Fun}\, S\, T \text{ for some } S, T \text{ and} \\ & \Gamma \vdash S \circledR A \text{ and} \\ & \forall \sigma : \Gamma' \leq \Gamma.\ \ \Gamma' \vdash s : S \circledR a \in A \implies \Gamma' \vdash (T\,\sigma)\, s \circledR F \cdot a \end{array}$$

$$\begin{array}{ll} \Gamma \vdash r : R \circledR f \in \mathsf{Fun}\, A\, F & \\ \quad :\iff & \Gamma \vdash R = \mathsf{Fun}\, S\, T \text{ for some } S, T \text{ and} \\ & \Gamma \vdash S \circledR A \text{ and} \\ & \forall \sigma : \Gamma' \leq \Gamma.\ \ \Gamma' \vdash s : S \circledR a \in A \implies \Gamma' \vdash (r\,\sigma)\, s : (T\,\sigma)\, s \circledR f \cdot a \in F \cdot a \end{array}$$

$$\begin{array}{ll} \Gamma \vdash T \circledR B & \\ \quad :\iff & \forall \sigma : \Gamma' \leq \Gamma.\ \ \Gamma' \vdash T\,\sigma = \downarrow^{\mathsf{Set}_k}_{\Gamma'} B : \mathsf{Set}_k \end{array}$$

$$\begin{array}{ll} \Gamma \vdash T : S \circledR A \in \mathsf{Set}_i & \\ \quad :\iff & \Gamma \vdash S = \mathsf{Set}_i \text{ and } \Gamma \vdash T \circledR A \end{array}$$

$$\begin{array}{ll} \Gamma \vdash t : T \circledR a \in B \quad (B \neq \mathsf{Set}_i) & \\ \quad :\iff & \forall \sigma : \Gamma' \leq \Gamma.\ \ \Gamma' \vdash t\,\sigma = \downarrow^{B}_{\Gamma'} a : T\,\sigma \end{array}$$

Having completed the definition, we can subsume the relation for types $\Gamma \vdash T \circledR A$ under the relation for terms $\Gamma \vdash T : \mathsf{Set}_i \circledR A \in \mathsf{Set}_i$ for an appropriate universe Set_i. The relation $\Gamma \vdash t : T \circledR a \in A$ is monotone in the sense that $\sigma : \Gamma' \leq \Gamma$ implies $\Gamma' \vdash t\,\sigma : T\,\sigma \circledR a \in A$. Further, it is closed under syntactic equality on the left, for both terms $\Gamma \vdash t' = t : T$ and type $\Gamma \vdash T' = T$ and it is closed under semantic equality on the right for values $a = a' \in [A]$ and type values $A = A' \in \mathcal{T}ype$. Further, we can show the "sandwiching" property we usually express by a candidate space. Let $\Gamma \vdash T \circledR A$.

1. If $\Gamma' \vdash t\,\sigma = \mathsf{R}^{\mathsf{ne}}_{|\Gamma'|}\, e : T\,\sigma$ for all $\sigma : \Gamma' \leq \Gamma$, then $\Gamma \vdash t : T \circledR \uparrow^{A} e \in A$.
2. If $\Gamma \vdash t : T \circledR a \in A$ and $\sigma : \Gamma' \leq \Gamma$ then $\Gamma' \vdash t\,\sigma = \mathsf{R}^{\mathsf{nf}}_{|\Gamma'|} \downarrow^{A} a : T\,\sigma$.

The path to soundness of NbE is straightforward from here. We extend the logical relation to environments $\boxed{\Gamma \vdash \sigma : \Delta \circledR \rho}$ and build semantic judgements $\boxed{\Gamma \vDash t : T}$ and

$\boxed{\Gamma \vDash \sigma : \Delta}$ which we prove by the fundamental lemma of logical relations.

$$
\begin{array}{lll}
\Gamma \vdash () : () \circledR () & :\Longleftrightarrow & \text{true} \\
\Gamma \vdash (\sigma, s) : (\Delta, S) \circledR (\rho, a) & :\Longleftrightarrow & \Gamma \vdash \sigma : \Delta \circledR \rho \text{ and } \Gamma \vdash s : S\,\sigma \circledR a \in (\!|S|\!)\rho \\
\Gamma \vDash t : T & :\Longleftrightarrow & \forall \Delta, \sigma, \rho.\ \ \Delta \vdash \sigma : \Gamma \circledR \rho \\
& & \qquad \Longrightarrow\ \Delta \vdash t\,\sigma : T\,\sigma \circledR (\!|t|\!)\rho \in (\!|T|\!)\rho \\
\Gamma \vDash \tau : \Gamma' & :\Longleftrightarrow & \forall \Delta, \sigma, \rho.\ \ \Delta \vdash \sigma : \Gamma \circledR \rho \\
& & \qquad \Longrightarrow\ \Delta \vdash \tau\,\sigma : \Gamma' \circledR (\!|\tau|\!)\rho
\end{array}
$$

Soundness of NbE follows as in Section 2.6, and we have

$$
\begin{array}{llll}
\Gamma \vdash t : T & \Longrightarrow & \Gamma \vDash t : T & \\
& \Longrightarrow & \Gamma \vdash t\,\mathsf{id} : T\,\mathsf{id} \circledR (\!|t|\!)\uparrow^{\Gamma} \in (\!|T|\!)\uparrow^{\Gamma} & \\
& \Longleftrightarrow & \Gamma \vdash t : T \circledR a \in A & \text{with } a = (\!|t|\!)\uparrow^{\Gamma} \text{ and } A = (\!|T|\!)\uparrow^{\Gamma} \\
& \Longrightarrow & \Gamma \vdash t = \mathsf{R}^{\mathsf{nf}}_{|\Gamma|} \downarrow^{A} a : T & \\
& \Longleftrightarrow & \Gamma \vdash t = \mathsf{nf}^{T}_{\Gamma} t : T &
\end{array}
$$

Remark. In the presence of the logical relation $\Gamma \vdash T \circledR A$ between syntactic types and type values, we can define a dependently-typed candidate space as follows:

$$
\begin{array}{rcll}
\overline{A}_{\Gamma} & = & \{\ (t, a) & \mid \forall A' = A \in \mathcal{T}ype.\ \ \forall \Gamma \vdash T \circledR A.\ \ \Gamma \vdash t : T \text{ and} \\
& & & \quad \forall \sigma : \Gamma' \leq \Gamma.\ \Gamma' \vdash t\,\sigma = \mathsf{R}^{\mathsf{nf}}_{|\Gamma'|} \downarrow^{A'} a : T\,\sigma\ \} \\
\underline{A}_{\Gamma} & = & \{\ (t, \uparrow^{A'} e) & \mid A' = A \in \mathcal{T}ype \text{ and } \forall \Gamma \vdash T \circledR A.\ \ \Gamma \vdash t : T \text{ and} \\
& & & \quad \forall \sigma : \Gamma' \leq \Gamma.\ \Gamma' \vdash t\,\sigma = \mathsf{R}^{\mathsf{ne}}_{|\Gamma'|} e : T\,\sigma\ \}
\end{array}
$$

Note how the presence of $\Gamma \vdash T \circledR A$ lets us ensure the well-typedness of t relative to type value A.

Let us demonstrate law $(\Pi\,\underline{A}\,\overline{F})_{\Gamma} \subseteq \overline{\mathsf{Fun}\,A\,F}_{\Gamma}$. Assume $(r, f) \in (\Pi\,\underline{A}\,\overline{F})_{\Gamma}$ and $\mathsf{Fun}\,A'\,F' = \mathsf{Fun}\,A\,F \in \mathcal{S}et_k$ and $\Gamma \vdash \mathsf{Fun}\,S\,T \circledR \mathsf{Fun}\,A\,F$. The restriction to function types is not a loss of generality, in case of $\mathsf{Fun}\,A'\,F'$ due to inversion on semantic type equality and for $\mathsf{Fun}\,S\,T$ due to the type conversion rule for syntactic typing. Further assume $\sigma : \Gamma' \leq \Gamma$ and let $n = |\Gamma'|$ and show $\Gamma' \vdash r\,\sigma = \mathsf{R}^{\mathsf{nf}}_{n} \downarrow^{\mathsf{Fun}\,A'\,F'} f : (\mathsf{Fun}\,S\,T)\,\sigma$. Let $a = \uparrow^{A'} \mathsf{x}_n$ and $\Delta = \Gamma', S\,\sigma$ and $\sigma' = \sigma\uparrow$ which entails $\sigma' : \Delta \leq \Gamma$. Note that $(\mathsf{v}_0, a) \in \underline{A}_{\Delta}$ because $\forall \tau : \Delta' \leq \Delta.\ \ \Delta' \vdash \mathsf{v}_0\,\tau = \mathsf{R}^{\mathsf{ne}}_{|\Delta'|} \mathsf{x}_n : S\,\sigma'\,\tau$. By instantiation of the assumption on (r, f) we obtain $((r\,\sigma')\,\mathsf{v}_0,\ f \cdot a) \in \overline{F \cdot a}_{\Delta}$. By definition with $F \cdot a = F' \cdot a \in \mathcal{T}ype$ it follows $\Gamma', S\sigma \vdash (r\,\sigma')\,\mathsf{v}_0 = \mathsf{R}^{\mathsf{nf}}_{n+1} \downarrow^{F' \cdot a} (f \cdot a) : (T\,\sigma')\,\mathsf{v}_0$. By λ-abstraction, $\Gamma' \vdash \lambda.\,(r\,\sigma\uparrow)\,\mathsf{v}_0 = \lambda.\,\mathsf{R}^{\mathsf{nf}}_{n+1} \downarrow^{F' \cdot a} (f \cdot a) : \mathsf{Fun}\,(S\sigma)\,\lambda.\,(T\,\sigma\uparrow)\,\mathsf{v}_0$. Since $\Gamma \vdash T : S \to \mathsf{Set}_k$ and therefore $\Gamma' \vdash T\,\sigma : S\,\sigma \to \mathsf{Set}_k$, we can apply η-equality to contract the type to $\mathsf{Fun}\,(S\,\sigma)\,(T\,\sigma)$. Similarly, since $\Gamma \vdash r : \mathsf{Fun}\,S\,T$ we have $\Gamma' \vdash r\,\sigma = \lambda.\,(r\,\sigma\uparrow)\,\mathsf{v}_0 : \mathsf{Fun}\,(S\,\sigma)\,(T\,\sigma)$ and can η-contract the left-hand side. Finally, we fold back the quoting step at function type. Thus, we arrive at our goal $\Gamma' \vdash r\,\sigma = \mathsf{R}^{\mathsf{nf}}_{n} \downarrow^{\mathsf{Fun}\,A'\,F'} f : (\mathsf{Fun}\,S\,T)\,\sigma$.

However, completing the program of Section 3.9 for dependent types is not yet a low-hanging fruit. We cannot directly adapt the *Kripke typed applicative structures* D^T_Γ to dependently types, since dependent types with de Bruijn indices change their denotation with Γ, violating $\sigma(_) : \mathcal{A}_\Gamma \to \mathcal{A}_{\Gamma'}$ for $\sigma : \Gamma' \leq \Gamma$, where $\mathcal{A} \in \widehat{D}^T$. In fact, writing $\widehat{D}^T$ does not even make sense, because a type expression T has meaning only relative to a context.

A way out could be *categories with families* [Dybjer, 1996]. Alternatively, we could switch from de Bruijn indices to named variables, which are absolute references into a context and do not change their meaning with context extensions. We explore this approach for System F in Section 5, but leave dependent types to future work.

4.8 Summary

How have dependent types affected the normalization-by-evaluation algorithm and its semantic justification?

First η-expansion for dependent types is much more complicated than for simple types. For simple types, we have the choice of η-expanding a non-normal term everywhere and obtaining an η-long normal form by β-normalization. Or we could β-normalize first and then η-expand to a long normal form. For dependent types, such a phase separation of β- and η-normalization does not work. η-expanding first cannot work since the arity of a function type can be determined only after β-normalization. For instance, the type $T(\mathsf{suc}\,\mathsf{zero})$ with T given by $T\,\mathsf{zero} = \mathsf{N}$ and $T\,(\mathsf{suc}\,n) = \mathsf{N} \to T\,n$ is a function type of arity 1, but we know this only after computing its β-normal form (at least the weak head normal form). On the other hand, η-expanding after β-normalization fails in the presence of singleton types $\langle t : T \rangle$ with η-expansion $u \longrightarrow t$ in case $u : \langle t : T \rangle$. For singleton types, η-expansion can trigger new β-reductions, for instance in $T(x)$ where variable x has type $\langle \mathsf{zero} : \mathsf{N} \rangle$.

NbE analyzes η-expansion into reflection, which corresponds to η-expansion of free variables on the inside of a term, and reification, which corresponds to an η-expansion on the outside of a term. The inner η-expansion is lazy and interleaved with computation (β-reduction), as witnessed by the law

$$(\uparrow^{\mathsf{Fun}\,A\,F} e) \cdot a = \uparrow^{F \cdot a}(e \downarrow^A a).$$

Integration of singleton types into our NbE framework is effortless and has been carried out in Abel et al. [2011]. The essence is a reification law

$$\uparrow^{\langle a : A \rangle} e = a$$

that performs η-expansion at singleton types at just the right time, meaning as soon as the value of $\uparrow^{\langle a:A \rangle} e$ is demanded by a computation, and with the guarantee that e's type has emerged as a singleton type at this point or will never do so. This way, we never miss a β-reduction that could be triggered by an η-expansion. An NbE-inspired strategy for type checking in the presence of dependent and singleton-types has been implemented as part of the prototypical language MiniAgda [Abel, 2010b].

For the semantic justification of dependently-typed NbE we could reuse a PER model which is standard except for the inclusion of neutrals at the base types. It is constructed

by an inductive-recursive definition of the valid semantic types; this inductive-recursive definition seems to be the essence of predicative dependent type theories. The model gets its specific touch by the dependently-typed candidate space that all of its types inhabit. This is shown by an extra induction on valid types. Induction on valid types lets us also construct a Kripke logical relation between syntax and semantics, which completes the correctness proof of NbE. Dependent types certainly complicate this logical relation, since syntactic types are context sensitive and need shifting of de Bruijn indices in case of context extension.

The models we have constructed justify not only NbE, but also the injectivity of the dependent function type constructor in the presence of universes and η-conversion on the type level [Abel et al., 2007b]. Before, this result had only been established for a formulation of type theory that confined η-conversion to terms Goguen [1994].[13] Type constructor injectivity is essential for efficiently checking type equality,[14] which is an integral part of any type checker for dependent types.

[13]Some formulations of type theory [Nordström et al., 1990, Altenkirch, 1994, Abel and Coquand, 2007] use an extension function El on the level of syntax, meaning that for $T : \mathsf{Set}$ they do not have $t : T$ but only $t : \mathsf{El}(T)$. Those variants confine $\beta\eta$-equality to terms t, thus, getting injectivity for type constructors for free. However, practical implementations such as Agda [Norell, 2007] and Coq [INRIA, 2012] spare the user the bureaucracy of El. This is very sensible, however, hardens the type theorist's job to prove consistency and decidability of type checking for these systems.

[14]Injectivity of Fun lets us reduce the problem $\mathsf{Fun}\, S\, T = \mathsf{Fun}\, S'\, T'$ to the problems $S = S'$ and $T = T'$.

5 Impredicativity

Impredicative type theories allow the formation of polymorphic types that can be instantiated by *all* other types. In predicative theories such as PTT, this is not allowed; a type like $(X : \mathsf{Set}_0) \to X \to X$, which is the type of the identity function polymorphic in all types $X : \mathsf{Set}_0$, is legal but not a member of Set_0 itself. A type that quantifies over types of a universe Set_i must live in a higher universe, at least Set_{i+1}. We get an impredicative universe Set_0 if we relax the formation rules of function types in Set_0 to

$$\frac{\Gamma \vdash S \qquad \Gamma, S \vdash T : \mathsf{Set}_0}{\Gamma \vdash \mathsf{Fun}\, S\, \lambda T : \mathsf{Set}_0}.$$

The difference to PTT is that S can now be any well-formed type, it does not have to be from Set_0. In particular, S can be Set_0 itself, as in the type of the polymorphic identity. Since Set_0 is not a member of Set_0 but Set_1, the domain of a impredicative polymorphic function can live in a higher universe than the whole function type.

An impredicative universe like the modified Set_0 cannot be defined inductively. Because its types, like $\mathsf{Fun}\, S\, \lambda T$, reference arbitrary types S in higher universes, we would need the definition of the higher universes before defining Set_0. This circularity prevents an inductive definition of valid types from below.

In fact, impredicativity is one step from inconsistency. Several paradoxes have been discovered in extensions of impredicative type theories and logics. By an adaption of Burali-Forti's paradox, Girard [1972] showed that System U^-, which has another impredicative universe stacked on the impredicative base universe, is inconsistent.[1] As a consequence, System U with a universe $\mathsf{Set} : \mathsf{Set}$ that contains a code for itself is inconsistent as well, since it simulates[2] System U^-. System U can be simulated in turn if we add impredicative strong existential types to type theory [Hook and Howe, 1986]. Although $\mathsf{Set} : \mathsf{Set}$ is logically inconsistent, it is computationally consistent (has subject reduction), has (incomplete) type checking [Coquand and Takeyama, 2000, Abel and Altenkirch, 2011] and can serve as the basis of a polymorphic programming language [Burstall and Lampson, 1984, Cardelli, 1986]. However, due to logical inconsistency, System U programs are not terminating in general [Hurkens, 1995], and the best we can offer is untyped NbE.

Impredicative quantification alone seems to be consistent;[3] there are several popular

[1] The proof of System U's inconsistency has later been simplified by Coquand [1986] and Hurkens [Hurkens, 1995, Barthe and Coquand, 2006].

[2] In fact, System U simulates any other pure type system (PTS) [Barendregt, 1991], it is the terminal object in the category of PTSs. PTT is simulated by erasing all universe levels $|\mathsf{Set}_i| = \mathsf{Set}$.

[3] Consistency proofs of impredicativity require impredicative principles such as complete lattices on the meta-level. As far as I know, a proof-theoretic reduction of impredicativity to finitary methods is unknown, as well as the proof-theoretic strength of impredicative systems in terms of ordinal analysis. Therefore, consistency of impredicativity is empirical—but in the end, so is the consistency of any foundation of mathematics, as the failure of Hilbert's program demonstrated.

impredicative logics and type systems, including Girard's System F^ω [1972] and System F [Reynolds, 1974], the Calculus of Constructions (CoC) [Coquand and Huet, 1988], the Calculus of Inductive Constructions (CIC) [Coquand and Paulin, 1988, Paulin-Mohring, 1993, Werner, 1994] underlying Coq [INRIA, 2012], and second- and higher-order predicate logic, to name a few.

NbE can be adapted to impredicative type systems, as we will demonstrate in the following, and can help to establish the meta-theoretic properties of these systems, like normalization, consistency, and decidability of type equality and type checking. We have considered NbE for System F [Abel, 2008], System F^ω [Abel, 2009a], and the CoC [Abel, 2010a] with inductive types in the impredicative universe. The main difficulty in constructing models for an impredicative universe is that induction on types in this universe is not available. Thus, we cannot define semantic types first and prove interesting properties of them later. We did so in the predicative case: we defined semantic types and later proved they inhabit a candidate space that lets us show soundness and/or completeness of NbE. For impredicative types, this is not possible, instead we have to first make up our mind and define a candidate space, yet without reference to the collection of valid semantic types. Then, we define the semantics of polymorphic types by quantifying over all candidates inhabiting this candidate space.

In Girard's normalization proof of System F [Girard et al., 1989], the space consists of the *reducibility candidates* $\mathcal{A} \in \mathsf{CR}$, which are closed under function space and arbitrary intersection (impredicativity here!). The interpretation $[\![T]\!]\rho$ of type T in an environment of reducibility candidates $\rho \in \mathsf{TyVar} \to \mathsf{CR}$ is then defined by:

$$
\begin{array}{rcl}
[\![X]\!]\rho & = & \rho(X) \\
[\![S \to T]\!]\rho & = & [\![S]\!]\rho \to [\![T]\!]\rho \\
[\![\forall X\, T]\!]\rho & = & \bigcap_{\mathcal{A} \in \mathsf{CR}} [\![T]\!](\rho, \mathcal{A}/X)
\end{array}
$$

A naive attempt to replace the "big" impredicative intersection by a "small" intersection over syntactic types, $[\![\forall X\, T]\!] = \bigcap_{S \in \mathsf{Ty}} [\![T[S/X]]\!]$, fails because the type $T[S/X]$ can be equal to $\forall XT$ (in case $S = \forall XT$ and $T = X$) or even bigger—thus, the interpretation function is ill-defined.[4]

The definition of interpretation lets us infer $[\![T]\!]\rho \in \mathsf{CR}$, so we know that each valid semantic type is a reducibility candidate, but we do not know more than this. If we need additional properties of semantic types, we have to refine the concept of CR beforehand. In the case of NbE, we want to know that elements of a semantic type $[\![A]\!]$ are reifiable at type A, thus, we have to design our candidate space accordingly. Or, to show soundness of NbE, we want to interpret types A as sets of term-value pairs such that the reified value is equal to the term. We cannot use the approach we used in Section 4.7 for dependent types since it presupposes a logical relation between syntactic and semantic types. Instead, we will adapt our concept of *type structure* from Section 3.9 to polymorphic types.

It turns out that the definition of greatest candidate $\overline{T}$ of type T as a Kripke set is inconvenient when T has free de Bruijn indices. For instance, in the simple case $\overline{T} =$

[4] However, this works for predicative System F where type variables X can only be instantiated by monotypes S that do not contain quantification $\forall$ over types.

Tm^T it does not hold that $\overline{T}_\Gamma \subseteq \overline{T}_{\Gamma'}$ since T appears in different contexts, thus, changes its meaning. We could use transport $\sigma : \Gamma' \leq \Gamma$ and have $\overline{T}_\Gamma \subseteq \overline{T\,\sigma}_{\Gamma'}$, interpreting semantics as presheafs, an approach apparently followed in Altenkirch et al. [1997]. Still, $\overline{T}$ by itself does not make sense, because to apply a transport to T we need to know in which world Γ we have to start. We deal with these issues by ditching de Bruijn indices and changing to named variables. Named variables keep their meaning under context extensions, thus, a semantic type $\overline{T}$ is a valid Kripke set. The same effect could be achieved with a de Bruijn level or locally nameless variable representation [Pollack, 1994].

The named approach to NbE has been studied in Abel et al. [2008], Abel [2008, 2009a]. It does not require nominal sets or logic Pitts [2010], only the quotation process has to be informed of the names already in use that should be avoided when generating a fresh name. In above works, we achieved this by *contextual reification*, a reification function from value to terms directed by both type and typing context. In the following, we refine this approach by keeping the separation of type-aware, context-oblivious reification $\downarrow^A$ and a read-back function that is responsible for picking fresh names and thus, must be informed about context length (in case of de Bruijn representations) or names of context-bound variables (in the named approach).

TyVar	$\ni$	X, Y			type variables
Ty	$\ni$	R, S, T	$::=$	$X \mid S \to T \mid \forall X T$	types
Var	$\ni$	x, y			term variables
Tm	$\ni$	r, s, t	$::=$	$x \mid \lambda x{:}S.\,t \mid r\,s$	terms (typed lambda calculus)
			$\mid$	$\Lambda X t \mid r\,S$	type abstraction and application
			$\mid$	$t\,\sigma$	explicit substitution
Subst	$\ni$	σ, τ	$::=$	$\mathsf{id} \mid \sigma\,\tau$	category of substitutions
			$\mid$	$(\sigma, s/x) \mid (\sigma, S/X)$	term and type substitutions
Cxt	$\ni$	Γ, Δ	$::=$	$()$	mixed context
			$\mid$	Γ, X	type variable decl. $(X \notin \lvert\Gamma\rvert)$
			$\mid$	$\Gamma, x{:}S$	term variable decl. $(x \notin \lvert\Gamma\rvert)$

Figure 5.1: System F syntax.

5.1 System F Syntax

We consider Church-style formulation of System F with separate syntactic categories for types and terms, but a mixture of term and type variable valuations in substitutions and contexts (see Figure 5.1). In contrast to the presentation in Abel [2008, Section 2], we are explicit about type variable bindings in contexts, and we consider explicit sub-

stitutions on terms.[5] However, on types we use substitution as an operation, $T[\sigma]$, so that type equality is just α-equality. We maintain the invariant that in contexts each variable can be declared at most once. The domain $|\Gamma|$ of context Γ be the list of variables declared in Γ.

Figure 5.2 presents typing for System F and the $\beta\eta$-axioms of term equality. There are also the congruence rules and rules for substitution equality and propagation of explicit substitutions, which we have omitted. They are similar to the ones presented in Figures 3.3 and 3.4 (modulo name representation) or the ones in Abel et al. [2008] (modulo dependent typing).

Weakening is implicit in all judgments: if a judgement holds in context Γ then it holds also for the extended context $\Gamma' \leq \Gamma$. Consequently, shifting is missing from the grammar of substitutions; it is simply id.

We write Ty_Γ for $\{T \in \mathsf{Ty} \mid \Gamma \vdash T\}$ and Tm_Γ^T for $\{t \in \mathsf{Tm} \mid \Gamma \vdash t : T\}$.

5.2 System F Type Semantics via Candidate Space

We shall now sketch the semantics of System F types in terms of *(applicative) System F structures* as given in Abel [2008, Section 3].

A family of sets Z_Γ indexed by well-formed contexts $\vdash \Gamma$ is called *Kripke* if $Z_\Gamma \subseteq Z_{\Gamma'}$ whenever $\Gamma' \leq \Gamma$. Examples of Kripke sets are the well-formed types $\mathsf{Ty}_\Gamma = \{T \in \mathsf{Ty} \mid \Gamma \vdash T\}$ and $\mathsf{Tm}_\Gamma^T = \{t \in \mathsf{Tm} \mid \Gamma \vdash t : T\}$, the terms of type T.

A *System F application structure* is a family D^T of Kripke sets for each type $T \in \mathsf{Ty}$ with a family of partial operations

$$\begin{array}{rcl} \mathsf{app}_\Gamma^{S \to T} & \in & D_\Gamma^{S \to T} \to D_\Gamma^S \to D_\Gamma^T \\ \mathsf{App}_\Gamma^{\forall X T} & \in & D_\Gamma^{\forall X T} \to (S \in \mathsf{Ty}_\Gamma) \to D_\Gamma^{T[S/X]}. \end{array}$$

When type and context indices are reconstructable or do not matter, we simply write $_ \cdot _$ for any of the application operations. System F application structures are the System F equivalent of the applicative aspect of System T's type structures as defined in Section 3.9. We will later complete System F application structures by an evaluation operation to *System F structures*. Let us for now drop the "System F" and simply say *application structures*.

A *total application structure* is one in which both application operations are total. Examples for total application structures are well-typed terms Tm_Γ^T and well-typed terms modulo term equality $\Gamma \vdash _ = _ : T$. (Partial) application structures include raw terms Tm and values D (defined later in analogy of System T values).

The pointwise product $(D \times E)_\Gamma^T = D_\Gamma^T \times E_\Gamma^T$ of two application structures forms a application structure with application defined componentwise. A *application substructure* $E \subseteq D$ is a family of subsets $E_\Gamma^T \subseteq D_\Gamma^T$ on with application inherited from D. The

[5] The choice of explicit substitutions is forced by the generality of our notion of *System F structure* (see Section 5.3). Since we do not ask for (weak) extensionality on the level of evaluation, β-equality is not modeled for substitution as operation; the counterexample in the introduction to Section 3.5) applies also to named variable representations.

System F's type level has no interesting notion of computation, thus, we can use substitution as operation there.

$\boxed{\Gamma \vdash T}$ Well-formed types.

$$\frac{X \in |\Gamma|}{\Gamma \vdash X} \qquad \frac{\Gamma \vdash S \qquad \Gamma \vdash T}{\Gamma \vdash S \to T} \qquad \frac{\Gamma, X \vdash T}{\Gamma \vdash \forall X T}$$

$\boxed{\vdash \Gamma}$ Well-formed contexts.

$$\frac{}{\vdash ()} \qquad \frac{\vdash \Gamma}{\vdash \Gamma, X} \qquad \frac{\vdash \Gamma \qquad \Gamma \vdash S}{\vdash \Gamma, x{:}S}$$

$\boxed{\Gamma \vdash t : T}$ Typing.

$$\frac{\vdash \Gamma \qquad \Gamma(x) = S}{\Gamma \vdash x : S} \qquad \frac{\Gamma, x{:}S \vdash t : T}{\Gamma \vdash \lambda x{:}S.\, t : S \to T} \qquad \frac{\Gamma \vdash r : S \to T \qquad \Gamma \vdash s : S}{\Gamma \vdash r\, s : T}$$

$$\frac{\Gamma, X \vdash t : T}{\Gamma \vdash \Lambda X t : \forall X T} \qquad \frac{\Gamma \vdash r : \forall X T \qquad \Gamma \vdash S}{\Gamma \vdash r\, S : T[S/X]} \qquad \frac{\Gamma \vdash \sigma : \Delta \qquad \Delta \vdash t : T}{\Gamma \vdash t\, \sigma : T[\sigma]}$$

$\boxed{\Gamma' \leq \Gamma}$ Context extension.

$$\frac{}{\Gamma \leq \Gamma} \qquad \frac{\Gamma' \leq \Gamma}{\Gamma', X \leq \Gamma} \qquad \frac{\Gamma' \vdash S \qquad \Gamma' \leq \Gamma}{\Gamma', x{:}S \leq \Gamma}$$

$$\frac{\Gamma' \leq \Gamma}{\Gamma', X \leq \Gamma, X} \qquad \frac{\Gamma' \leq \Gamma \qquad \Gamma \vdash S}{\Gamma', x{:}S \leq \Gamma, x{:}S}$$

$\boxed{\Gamma \vdash \sigma : \Delta}$ Substitution typing.

$$\frac{}{\Gamma \vdash \mathsf{id} : \Gamma} \qquad \frac{\Gamma_1 \vdash \tau : \Gamma_2 \qquad \Gamma_2 \vdash \sigma : \Gamma_3}{\Gamma_1 \vdash \sigma\, \tau : \Gamma_3} \qquad \frac{\Gamma \vdash \sigma : \Delta' \qquad \Delta' \leq \Delta}{\Gamma \vdash \sigma : \Delta}$$

$$\frac{\Gamma \vdash \sigma : \Delta \qquad \Delta \vdash S \qquad \Gamma \vdash s : S[\sigma]}{\Gamma \vdash (\sigma, s/x) : \Delta, x{:}S} \qquad \frac{\Gamma \vdash \sigma : \Delta \qquad \Gamma \vdash S}{\Gamma \vdash (\sigma, S/X) : \Delta, X}$$

$\boxed{\Gamma \vdash t = t' : T}$ Term equality.

$$\frac{\Gamma, x{:}S \vdash t : T \qquad \Gamma \vdash s : S}{\Gamma \vdash (\lambda x{:}S.\, t)\, s = t\,(s/x) : T} \qquad \frac{\Gamma \vdash t : S \to T}{\Gamma \vdash t = \lambda x{:}S.\, t\, x : S \to T}\; x \notin |\Gamma|$$

$$\frac{\Gamma, X \vdash t : T \qquad \Gamma \vdash S}{\Gamma \vdash (\Lambda X t)\, S = t\,(S/X) : T[S/X]} \qquad \frac{\Gamma \vdash t : \forall X T}{\Gamma \vdash t = \Lambda X.\, t\, X : \forall X T}\; X \notin |\Gamma|$$

Figure 5.2: System F typing and equality.

set of application substructures of D is denoted $\hat{D}$, and we write $\mathcal{A} \in \hat{D}^T$ for the Kripke family $\mathcal{A}_\Gamma \subseteq D^T_\Gamma$.

The essence of normalization proofs is to show that the a priori partial type structure with application and evaluation of normalizing terms is in fact total. In the following, we deliver the application part of the bill: a systematic method to construct a total System F application substructure induced by a candidate space.

On $\hat{D}$, we define the *Kripke function space* and *type abstraction*, in preparation for the specification of System F candidate spaces.

$$\begin{array}{lcl} (_ \to _)^{S\to T} & \in & \hat{D}^S \to \hat{D}^T \to \hat{D}^{S\to T} \\ (\mathcal{A} \to \mathcal{B})^{S\to T}_\Gamma & = & \{f \in D^{S\to T}_\Gamma \mid \forall \Gamma' \leq \Gamma, a \in \mathcal{A}_{\Gamma'}.\ f \cdot a \in \mathcal{B}_{\Gamma'}\} \\ (S._)^{\forall XT} & \in & \hat{D}^{T[S/X]} \to \hat{D}^{\forall XT} \\ (S.\mathcal{B})^{\forall XT}_\Gamma & = & \{f \in D^{\forall XT}_\Gamma \mid \forall \Gamma' \leq \Gamma.\ S \in \mathsf{Ty}_{\Gamma'} \implies f \cdot S \in \mathcal{B}_{\Gamma'}\} \\ \bigwedge^{\forall XT} & \in & ((S \in \mathsf{Ty}) \to \hat{D}^{T[S/X]}) \to \hat{D}^{\forall XT} \\ (\bigwedge^{\forall XT} \mathcal{F})_\Gamma & = & \bigcap_{S\in\mathsf{Ty}} (S.\mathcal{F}(S))^{\forall XT}_\Gamma \end{array}$$

$\bigwedge^{\forall XT}$ is a preliminary interpretation of universal quantification with only a small intersection (only over syntactic types). Note that X is a bound variable, in one case by $\forall$, in the other by the substitution operation, thus, it is subject to α-equality and does not pose a problem.[6] In the following, we drop the type superscripts from these semantic operators.

A *System F candidate space* is a family of two application substructures $\underline{T}, \overline{T} \in \hat{D}^T$ of D (formally $\underline{(\cdot)}, \overline{(\cdot)} \subseteq D$) that satisfy the following inclusion laws (pointwise at each world Γ):

$$\begin{array}{lcllcl} \underline{X} & \subseteq & \overline{X} & & & \\ \underline{S \to T} & \subseteq & \overline{S} \to \underline{T} & \qquad \underline{S} \to \overline{T} & \subseteq & \overline{S \to T} \\ \underline{\forall XT} & \subseteq & \bigwedge \underline{X.T} & \qquad \bigwedge \overline{X.T} & \subseteq & \overline{\forall XT} \end{array}$$

Herein, $\underline{X.T}_\Gamma(S) = \underline{T[S/X]}_\Gamma$ and $\overline{X.T}_\Gamma(S) = \overline{T[S/X]}_\Gamma$. We call $\underline{T}$ the *least candidate* and $\overline{T}$ the *greatest candidate* for type T.

For a semantic (proto)type $\mathcal{A} \in \hat{D}^T$ we say T *realizes* $\mathcal{A}$, written $\boxed{T \Vdash \mathcal{A}}$, if $\mathcal{A}$ inhabits the interval of the candidate space spanned by T, i. e., if $\underline{T} \subseteq \mathcal{A} \subseteq \overline{T}$. The proper interpretation of universal quantification in $\hat{D}$ is now given by

$$\begin{array}{lcl} \forall^{X.T} & \in & ((S \in \mathsf{Ty}) \to \hat{D}^S \to \hat{D}^{T[S/X]}) \to \hat{D}^{\forall XT} \\ \forall^{X.T}\mathcal{F} & = & \bigcap\limits_{S \Vdash \mathcal{A}} S.\mathcal{F}(S, \mathcal{A}) \end{array}$$

Note that the intersection here is big (impredicative), since we quantify over all candidates to define a new candidate.

[6]In formalizations, a locally nameless style is probably preferable to a named variable presentation. In local nameless style, bound variables like X vanish, obliviating the need for α-conversion.

We extend realizability to type substitutions σ and type environments $\rho \in \hat{D}^\sigma$ by setting

$$\sigma \Vdash^\Gamma \rho \;:\Longleftrightarrow\; \sigma(X) \Vdash \rho(X) \text{ for all } X \in |\Gamma|.$$

Herein, $\rho \in \hat{D}^\sigma$ if $\rho(X) \in \hat{D}^{\sigma(X)}$ for all $X \in |\Gamma|$. Let $\sigma \in \mathsf{Ty}^\Gamma$ mean that σ is a valid type substitution for Γ, i. e., $\sigma(X) \in \mathsf{Ty}$ for all $X \in |\Gamma|$.

Type interpretation $[\![_]\!](_,_) \in (T \in \mathsf{Ty}_\Gamma) \to (\sigma \in \mathsf{Ty}^\Gamma) \to (\rho \in \hat{D}^\sigma) \to \hat{D}^{T[\sigma]}$ is now defined by recursion on T:

$$\begin{array}{rcl} [\![X]\!](\sigma,\rho) & = & \rho(X) \\ [\![S \to T]\!](\sigma,\rho) & = & [\![S]\!](\sigma,\rho) \to [\![T]\!](\sigma,\rho) \\ [\![\forall XT]\!](\sigma,\rho) & = & \boldsymbol{\forall}^{X.T}(S,\mathcal{A}) \mapsto [\![T]\!]((\sigma,S/X),(\rho,\mathcal{A}/X)) \end{array}$$

Using the laws of the candidate space we prove that type interpretation is sound in the sense that $T[\sigma] \Vdash [\![T]\!](\sigma,\rho)$ if $\sigma \Vdash^\Gamma \rho$. Further, interpretation is compatible with substitution, i. e., we have

$$[\![T[\sigma]]\!](\tau,\rho) = [\![T]\!](\sigma\,\tau, [\![\sigma]\!](\tau,\rho))$$

for a pointwise extension $[\![\sigma]\!]$ of interpretation to type substitutions σ.

Pairs $A = (T,\mathcal{A})$ of a semantic type $\mathcal{A}$ and its realizer T form a System F type structure $\underline{\overline{D}}$ of candidates with function space and quantification.

$$\begin{array}{lcl} _ \to _ & \in & \underline{\overline{D}} \to \underline{\overline{D}} \to \underline{\overline{D}} \\ (S,\mathcal{A}) \to (T,\mathcal{B}) & = & (S \to T, \mathcal{A} \to \mathcal{B}) \\[1em] \boldsymbol{\forall} & \in & \underline{\overline{D}}^{\star\to\star} \to \underline{\overline{D}} \\ \boldsymbol{\forall}(X.T,\mathcal{F}) & = & (\forall XT, \boldsymbol{\forall}^{X.T}\mathcal{F}) \end{array}$$

Herein, $\underline{\overline{D}}^{\star\to\star} = (X \in \mathsf{TyVar}) \times (T \in \mathsf{Ty}) \times (((S,\mathcal{A}) \in \underline{\overline{D}}) \to \hat{D}^{T[S/X]})$, and not the naive $\underline{\overline{D}} \to \underline{\overline{D}}$, which is does not give enough information to ensure the well-formedness of $\boldsymbol{\forall}$. Note that the typing of $\to$ and $\boldsymbol{\forall}$ as operators on D uses the candidate space laws. Using $\underline{\overline{D}}$, we can view the pair $\sigma \Vdash^\Gamma \rho$ of matching environments as a single environment $\xi \in \underline{\overline{D}}^\Gamma$.

We finally have the tools to exhibit total application structures wrt. to a partial application structure D. Let $[\![T]\!]_\Gamma = (\xi \in \underline{\overline{D}}^\Gamma) \to [\![T]\!]\xi$. Then $[\![T]\!] \in \hat{D}^T$ is a total structure with application operations

$$\begin{array}{rcccl} \mathsf{app}_\Gamma^{S\to T}(f \in [\![S \to T]\!]_\Gamma,\, a \in [\![S]\!]_\Gamma)(\xi) & = & f \cdot a & \in & [\![T]\!]\xi \\ \mathsf{App}_\Gamma^{\forall XT}(f \in [\![\forall XT]\!]_\Gamma \;\;,\, S \in \mathsf{Ty}_\Gamma)\,(\xi) & = & f \cdot S & \in & [\![T[S/X]]\!]\xi. \end{array}$$

The well-formedness of type application in structure $[\![_]\!]$ relies on the substitution property of type interpretation.

A total *substructure* $[\![_]\!] \subseteq D$ is obtained if we choose a canonical type environment ξ^Γ for each context Γ. The substructure given by $[\![T]\!]_\Gamma = [\![T]\!]\xi^\Gamma$ shall be called *induced* by the underlying candidate space and the type valuation ξ^Γ. Induced application substructures are total since D's application on $[\![_]\!]$ is total by construction of the Kripke function space and universal quantification on $\hat{D}$.

A typical canonical environment for normalization proofs is $\xi_{\mathsf{id}}(X) = (X, \underline{X})$ which fits each context Γ.

5.3 Abstract Evaluation and the Fundamental Lemma for System F

A *System F structure* D is a System F application structure with a partial operation $(\!|_|\!) \in \mathsf{Tm}^T_\Gamma \to (\eta \in D^\Gamma_\Delta) \to D^{T[\eta]}_\Delta$ called *evaluation* which satisfies the following laws:

$$
\begin{array}{rcl}
(\!|x|\!)\eta & = & \eta(x) \\
(\!|r\,s|\!)\eta & = & (\!|r|\!)\eta \cdot (\!|s|\!)\eta \\
(\!|r\,S|\!)\eta & = & (\!|r|\!)\eta \cdot S[\eta] \\
(\!|\lambda x\!:\!S.\,t|\!)\eta \cdot a & = & (\!|t|\!)(\eta, a/x) \\
(\!|\Lambda X t|\!)\eta \cdot S & = & (\!|t|\!)(\eta, S/X)
\end{array}
$$

Herein, $\eta \in \mathsf{D}^\Gamma_\Delta$ is a D-environment satisfying $\eta(X) \in \mathsf{Ty}_\Delta$ for all type variables $X \in |\Gamma|$ and $\eta(x) \in D^{S[\eta]}_\Delta$ for all term variables $x \in |\Gamma|$. The equations are to be read as "if one side is defined, so is the other, and then both sides are equal".

Products $D \times E$ of System F structures are defined pointwise, as expected, and a System F substructure $E \subseteq D$ inherits interpretation from D, and if $(\!|t|\!)\eta$ is defined in D and $\eta \in E^\Gamma_\Delta$ is a valid E-environment, then $[\![t]\!]\eta \in E^{T[\eta]}_\Delta$ must be defined as well.

The *fundamental lemma of System F structures* now states that for any System F substructure structure $[\![_]\!] \subseteq D$ which is induced by a System F candidate space and canonical valuation ξ^Γ is total. Since application is total in induced structures, it remains to show that evaluation $(\!|t|\!)\eta$ is total. This follows from the more general statement that if $\eta \Vdash^\Gamma \rho$ then $(\!|t|\!)\eta \in [\![T]\!](\eta, \rho)$ which is proven by induction on $\Gamma \vdash t : T$, using the definition of type interpretation $[\![T]\!]$ and the evaluation laws.

5.4 Normalization by Evaluation for System F

In this section, we consider an algorithm of normalization by evaluation for System F. We adapt the developments of Chapter 3 to polymorphism and named variables. By now, we have gained sufficient familiarity with NbE to do this mechanically.

Values are as for simply-typed lambda calculus, only that we have additional polymorphic function closures $(\underline{\Lambda} X t)\eta$ and neutral type applications $e\,S$. Environments hold both values for term variables and type expressions for type variables.

$$
\begin{array}{llll}
\mathsf{D} & \ni & a, b, f & ::= \ (\underline{\lambda} x t)\eta \mid (\underline{\Lambda} X t)\eta \mid \uparrow^T e \\
\mathsf{D^{ne}} & \ni & e & ::= \ x \mid e\,d \mid e\,S \\
\mathsf{D^{nf}} & \ni & d & ::= \ \downarrow^T a \\
\mathsf{Env} & \ni & \eta & ::= \ () \mid (\eta, d/x) \mid (\eta, S/X) \\
\mathsf{Ne} & \ni & u & ::= \ x \mid u\,v \mid u\,S \\
\mathsf{Nf} & \ni & v & ::= \ u \mid \lambda x\!:\!S.\,v \mid \Lambda X v
\end{array}
$$

Evaluation, application and type application are defined again as mutually recursive partial functions.

$$
\begin{array}{lcl}
(\!|_|\!) & \in & \mathsf{Exp} \times \mathsf{Env} \to \mathsf{D} \\
(\!|x|\!)\ \eta & = & \eta(x) \\
(\!|\lambda x\!:\!S.\, t|\!)\eta & = & (\underline{\lambda} x t)\eta \\
(\!|\Lambda X t|\!)\ \eta & = & (\underline{\Lambda} X t)\eta \\
(\!|r\, s|\!)\ \eta & = & (\!|r|\!)\eta \cdot (\!|s|\!)\eta \\
(\!|r\, S|\!)\ \eta & = & (\!|r|\!)\eta \cdot S[\eta]
\end{array}
$$

Evaluation satisfies the laws of the partial System F structure D of values, given the following implementation of application.

$$
\begin{array}{lcl}
_ \cdot _ & \in & \mathsf{D} \times \mathsf{D} \to \mathsf{D} \\
(\underline{\lambda} x t)\eta \cdot a & = & (\!|t|\!)(\eta, a/x) \\
\uparrow^{S \to T} e \cdot a & = & \uparrow^{T} (e \downarrow^{S} a)
\end{array}
$$

$$
\begin{array}{lcl}
_ \cdot _ & \in & \mathsf{D} \times \mathsf{Ty} \to \mathsf{D} \\
(\underline{\Lambda} X t)\eta \cdot S & = & (\!|t|\!)(\eta, S/X) \\
\uparrow^{\forall X T} e \cdot S & = & \uparrow^{T[S/X]} (e\, S)
\end{array}
$$

Application also applies to reflected neutrals of function or polymorphic type. Note the impredicative character of reflection at type $\forall XT$: it is defined in terms of reflection at the potentially bigger type $T[S/X]$.

In contrast to quotation into de Bruijn terms, the (partial) read-back functions are now indexed by a set of taken names $|\Gamma|$ instead of just the number n of variables in scope. We assume we have a function $\mathsf{fresh}\,|\Gamma|$ that returns a hitherto unused term variable $x \notin |\Gamma|$ and $\mathsf{Fresh}\,|\Gamma|$ that does the same for a type variable X.

$$
\begin{array}{lcl}
\mathsf{R}^{\mathsf{ne}}_{|\Gamma|} & \in & \mathsf{D}^{\mathsf{ne}} \to \mathsf{Ne} \\
\mathsf{R}^{\mathsf{ne}}_{|\Gamma|}\ x & = & x \\
\mathsf{R}^{\mathsf{ne}}_{|\Gamma|}\ (e\, d) & = & (\mathsf{R}^{\mathsf{ne}}_{|\Gamma|} e)\ (\mathsf{R}^{\mathsf{nf}}_{|\Gamma|} d) \\
\mathsf{R}^{\mathsf{ne}}_{|\Gamma|}\ (e\, S) & = & (\mathsf{R}^{\mathsf{ne}}_{|\Gamma|} e)\ S
\end{array}
$$

$$
\begin{array}{lcll}
\mathsf{R}^{\mathsf{nf}}_{|\Gamma|} & \in & \mathsf{D}^{\mathsf{nf}} \to \mathsf{Nf} & \\
\mathsf{R}^{\mathsf{nf}}_{|\Gamma|} \downarrow^{X} e & = & \mathsf{R}^{\mathsf{ne}}_{|\Gamma|} e & \\
\mathsf{R}^{\mathsf{nf}}_{|\Gamma|} \downarrow^{S \to T} f & = & \lambda x\!:\!S.\ \mathsf{R}^{\mathsf{nf}}_{|\Gamma|,x} \downarrow^{T} (f \cdot \uparrow^{S} x) & \text{where } x = \mathsf{fresh}\,|\Gamma| \\
\mathsf{R}^{\mathsf{nf}}_{|\Gamma|} \downarrow^{\forall Y T} f & = & \Lambda X.\ \mathsf{R}^{\mathsf{nf}}_{|\Gamma|,X} \downarrow^{T[X/Y]} (f \cdot X) & \text{where } X = \mathsf{Fresh}\,|\Gamma|
\end{array}
$$

As read-back is an *algorithm*, we do not rely on implicit α-conversion for types. Instead, when reifying at polymorphic type $\forall YT$, we substitute the fresh free variable X for the bound variable Y in T. (This also lends itself nicely to locally-nameless style.)

In an implementation, we could try to use Y in place of the fresh name if $Y \not\in |\Gamma|$, since the user-chosen name Y would be a more meaningful than an arbitrary fresh identifier. A similar name-preserving strategy could be employed when reifying a closure $\downarrow^{S \to T} (\underline{\lambda} x t)\xi$. In case $\downarrow^{\forall Y T} (\underline{\Lambda} X t)\xi$ we have even two names to choose from.

At this point, another comment on name management is in order. Our set of used names $|\Gamma|$ is an instance of Barral's *name generation environment* [2008, Def. 2.20], or short, *name supply*. He discusses global name generation mechanisms like Scheme's `gensym`, but discards them because such imperative effects do not have a direct counterpart in the functional world of mathematics.[7] He then observes that local name generation is sufficient; such local environments can be implemented as reader or context monads which are basically functions taking the name supply as input. Berger and Schwichtenberg's term families [1991, 2003], which are functions from name supply to lambda-terms and interpret the base types, are an instance of such a context monad.[8] The Abel et al. [2011] approach to NbE, which we have adopted to a named term representation here, confines name generation to quotation which is the "last" step in the normalization procedure, not intermingled with evaluation in the way reflection and reification are. Thus, there is no need for name generation in the semantic part of the algorithm, concerning values and evaluation. We trade name generation semantics for a semantics enriched with neutrals, called *accumulators* in Grégoire and Leroy [2002]. By this choice, we abandon tag-free interpreters[9] for NbE, however, we lose the tag-free representation of function values anyway as soon as we take untyped NbE as basis. We consider the removal of name-generation from the semantics as a great conceptual simplification. For instance, it enables us to proof completeness of NbE without reference to contexts (see Sections 3.7 and 4.5). Contexts and names come only back in the soundness proof in the form of Kripke logical relations.

The definition of the NbE algorithm is unchanged w. r. t. Chapter 3, only that the canonical environment $\uparrow^\Gamma$ needs to provide a valuation for type variables, which are mapped to themselves.

$$
\begin{array}{lcl}
\uparrow^\Gamma & \in & \mathsf{Env} \\
\uparrow^{()} & = & () \\
\uparrow^{\Gamma, x:S} & = & (\uparrow^\Gamma, (\uparrow^S x)/x) \\
\uparrow^{\Gamma, X} & = & (\uparrow^\Gamma, X/X) \\
 & & \\
\mathsf{nf}^T_\Gamma & \in & \mathsf{Exp} \to \mathsf{Nf} \\
\mathsf{nf}^T_\Gamma(t) & = & \mathsf{R}^{\mathsf{nf}}_{|\Gamma|} \downarrow^T (\!| t |\!)(\uparrow^\Gamma)
\end{array}
$$

Showing soundness of NbE follows from the fundamental lemma for the System F

[7] But cf. Filinski [2001] who uses a state monad for name generation in NbE for the computational lambda-calculus.

[8] Filinski [1999], Dybjer and Filinski [2000] follow the term families approach, and present a weak HOAS (higher-order abstract syntax) wrapper for it.

[9] Tag-free interpreters use untagged values. Whether a value is a function or, e. g., a number, is determined by the typing and need not be checked during evaluation. The evaluator presented in Section 2.1 is tag-free in this sense.

substructure of $\mathsf{Tm}^T_\Gamma \times \mathsf{D}$ induced by

$$\begin{array}{lcl} \overline{T}_\Gamma & = & \{(t,a) \quad \in \mathsf{Tm}^T_\Gamma \times \mathsf{D} \mid \forall \Gamma' \leq \Gamma.\ \ \Gamma' \vdash t = \mathsf{R}^{\mathsf{nf}}_{|\Gamma'|} \downarrow^T a : T\} \\ \underline{T}_\Gamma & = & \{(t, \uparrow^T e) \in \mathsf{Tm}^T_\Gamma \times \mathsf{D} \mid \forall \Gamma' \leq \Gamma.\ \ \Gamma' \vdash t = \mathsf{R}^{\mathsf{ne}}_{|\Gamma'|}\ e \quad : T\}. \end{array}$$

Our work amounts to proving that these two Kripke sets indeed form a System F candidate space (see similar proof in Abel [2008]).

Completeness of NbE uses the fundamental lemma on the substructure of $\mathsf{D} \times \mathsf{D}$ given by the (context-independent) candidate space of PERs

$$\begin{array}{lcl} \overline{T} & = & \{(a,a') \qquad \in \mathsf{D}^2 \mid \forall \Gamma.\ \ \mathsf{R}^{\mathsf{nf}}_{|\Gamma|} \downarrow^T a = \mathsf{R}^{\mathsf{nf}}_{|\Gamma|} \downarrow^T a'\} \\ \underline{T} & = & \{(\uparrow^T e, \uparrow^T e') \in \mathsf{D}^2 \mid \forall \Gamma.\ \ \mathsf{R}^{\mathsf{ne}}_{|\Gamma|}\ e \quad = \mathsf{R}^{\mathsf{ne}}_{|\Gamma|}\ e'\}. \end{array}$$

Further we show by induction on $\Gamma \vdash t = t' : T$ that term equality is modeled by this substructure.

Correctness of Nbe shows that normal forms generated from a closed term t via two different name supplies Γ_1 and Γ_2 are α-equivalent: With $d_i = \downarrow^T (\!| t |\!) \uparrow^{\Gamma_i}$ we get by soundness $\Gamma_1 \vdash t = \mathsf{R}^{\mathsf{nf}}_{|\Gamma_1|} d_1 : T$ and $\Gamma_2 \vdash t = \mathsf{R}^{\mathsf{nf}}_{|\Gamma_2|} d_2 : T$, and by strengthening and transitivity, $\vdash \mathsf{R}^{\mathsf{nf}}_{|\Gamma_1|} d_1 = \mathsf{R}^{\mathsf{nf}}_{|\Gamma_2|} d_2 : T$. Since both terms are normal forms, they must be α-equivalent.[10] If we parameterize the readback functions R^{ne} and R^{nf} over *name supplies* instead of used names, we can even prove that the choice of fresh does not influence normal forms (up to α).

This completes our presentation of NbE for System F. The main investment has been a design of Kripke semantics based on a generic notion of candidates. It seems that as we move on to more expressive type systems like F$^\omega$ or the CoC, the main work is to clarify and formulate the semantic structures, while the actual NbE algorithm needs little modifications only.

5.5 System F$^\omega$, the Calculus of Constructions, and Beyond

System F$^\omega$ replaces System F's type expressions by *type constructors*, which are essentially simply-typed expressions over the constants $\to$ and $\forall$, only that in this case we say simply-*kinded* to avoid confusing the types of types constructors (i. e., kinds) with the types of terms.

Normalization in F$^\omega$ concerns both type constructors as well as terms. Normalization of type constructors is as easy as normalization of simply-typed lambda-calculus. In principle, we could restrict to normal type constructors in the first place, and use hereditary substitutions Watkins et al. [2003] to normalize the application of two normal type constructors, or the substitution of a normal type constructor into another normal type constructor. However, this would prevent us from extending F$^\omega$ with inductive kinds and types defined by recursion, such as present in LX [Crary and Weirich, 1999] or later versions of Haskell [Yorgey et al., 2012], since the method of hereditary substitutions does not scale to recursion. Furthermore, since we are implementing normalization for

[10] While this is intuitively clear, technically we must also prove *transitivity elimination* for judgemental equality.

lambda-terms anyway, we can use the same algorithm on the type level, if we use a single expressions language for terms and type expressions.

In Abel [2009a], we present NbE for F^ω type constructors and terms using contextual reification. For the formulation of abstract models of F^ω, we introduce *kind structures*, *type structures* and *term structures* (called *object structures* in Abel [2009a]). We exploit the stratification of F^ω, into these three layers; since types cannot depend on terms, and kinds not on types, we can define F^ω's semantics top down, first specifying the layer of kinds, then the layer of types, and finally the layer of terms. We define kind candidate spaces (in analogy to simply-typed candidate spaces) and (higher-kinded) type candidate spaces and instantiate them to obtain soundness and completeness results for type-level and term-level NbE. The full development and proofs can be found in Abel [2009b].

The pure Calculus of Constructions [Coquand and Huet, 1988] is an extension of F^ω by dependencies: types can depend on terms. However, in the absence of inductive types and recursion, dependencies are erasable, thus, NbE for the pure CoC is the almost the same as NbE for F^ω. It gets more interesting if we add an inductive type like the natural numbers and types defined by recursion on numbers [Werner, 1992]. Then dependencies can no longer be erased which significantly complicates the meta-theory. While dependencies cannot be erased in types, they can be erased in kinds, thus, every kind has a skeleton that corresponds to a simple kind. Werner [1992] exploits simple-kinding to define raw sets of semantic type constructors of higher kind. This definition allows to boot-strap the semantics of CoC+$\mathbb{N}$.

We adopt Werner's trick to define higher-kinded type candidate spaces for the CoC which enable us to prove termination and completeness of NbE for the CoC with large eliminations [Abel, 2010a]. However, it is quite a technical battle already, and soundness of NbE, which would require Kripke models for the CoC, remains open. Also, a general concept of *CoC structure* has not been formulated in the way we did for System F and F^ω.

If we add yet another universe to CoC, Werner's trick is no longer usable, since dependencies on the kind level can no longer be erased. Developing a semantic technique for this system is probably as hard as for the full Calculus of Inductive Constructions (CIC) [Coquand and Paulin, 1988], which has a countable hierarchy of predicative universes Type_i on top of the impredicative base universe Prop, and inductive types on each level. A justification of NbE for the CIC remains open.

6 Summary, Related Work, and Perspectives

In this thesis, we have demonstrated how to normalize expressions by evaluation in dependent and impredicative type systems. We have not followed the *intrinsic-typing* style as presented in Section 2.1 where *types come first* and terms are elements of types and can only be understood in connection with their type. Instead we have adopted a *extrinsic typing* or *type assignment* style where *terms come first* and can be assigned one or several types and be η-expanded w. r. t. the assigned type or not.

We describe evaluation in terms of *(syntactical) partial applicative structures* (see Section 3.2) which are partial and untyped versions of Mitchell's environment models [Mitchell, 1996, Section 4.5.3] and Barendregt's syntactical applicative structures [Barendregt, 1984, Section 5.3]. Partial applicative structures have many instances, subsuming many different approaches to value representation in NbE, such as

- meta-theoretic functions (e.g. set-theoretic functions) [Berger and Schwichtenberg, 1991],
- two-level lambda-calculus [Danvy, 1996, Vestergaard, 2001a, Aehlig and Joachimski, 2004, Abel et al., 2007b],
- Scott domains [Filinski, 1999, Filinski and Rohde, 2004, Abel et al., 2007a, 2011],
- closed or open normal forms,
- weak head normal forms or closures (Martin-Löf [1975], Coquand [1996], Altenkirch and Chapman [2009] and this work), and
- compiled code [Grégoire and Leroy, 2002, Boespflug et al., 2011, Aehlig et al., 2012].

Our type assignment approach to NbE is in essence an interleaving of evaluation and lazy reflection and reification on the level of values, followed by quotation that reads values back into η-long normal forms. Due to the lazy nature of reification, read-back triggers further evaluation of function bodies, which can involve further reflection and computation. Through the study of NbE we have gained new insights into η-expansion which is applied to the "inputs" (free variables) via reflection and to the "output" (evaluation result) via reification. This separation of η-expansion into two phases has enabled us to seamlessly integrate singleton types with dependent types [Abel et al., 2011]. Our treatment considerably simplifies Stone and Harper [2006], which depends on a non-standard notion of Kripke logical relations, and extends it to universes and large eliminations.

The completeness of NbE, meaning that two terms result in the same normal form if they are identified in the equational theory of the calculus, has been shown by modeling types as partial equivalence relations (PERs). We have analyzed PERs as subsets or predicates over groupoidal applicative structures, leading to simplified form of the fundamental lemma for logical relations. To standard (untyped) models of type theory we have added the necessary type assignment in form of *typed candidate spaces*, that as an instance allowed us to specify the desired properties of reflection and reification to obtain completeness of NbE. These candidate spaces have proven essential for the treatment of NbE in impredicative type systems where properties of semantic types have to be specified a priori since they cannot be proven by induction a posteriori.

The soundness of NbE for the impredicative systems F and F^ω has been obtained from a general semantics in *typed Kripke structures*. These general formulations allow for an algebraic formulation of the fundamental lemma that can be instantiated for both soundness and completeness of NbE. In the specification of typed Kripke structures we have made essential use of the stratification of F^ω expressions into kinds, types, and terms, which allowed us to define kind structures, type structures, and term structures, in this order [Abel, 2009b]. A suitable adaption of typed Kripke structures for dependent types is missing yet, thus, we have shown soundness of NbE for predicative dependent types by a Kripke logical relation defined by induction on semantic types. Such an induction is not available for impredicative dependent types, and the soundness of NbE for the Calculus of Constructions with large eliminations remains open [Abel, 2010a].

A major novel meta-theoretic result of our semantic studies of NbE for dependent types is the decidability of type checking in the presence of predicative universes and computation and η-equivalence on the type level [Abel et al., 2007b]. The key to this result is the injectivity of type constructors, which is hard to establish in the presence of a typed equality judgement, but follows from the inductive structure of our semantic type equality and the presence of sound quotation into syntax.

A consequence of our extrinsic, type-assignment approach is that we can model NbE for impredicative types in a meta-theory with only impredicative propositions. In contrast to an intrinsic approach [Altenkirch et al., 1996], impredicative types of System F are not modeled by impredicative sets, but by impredicative predicates on values. In terms of the Calculus of Inductive Constructions, we model impredicative Set by impredicative Prop. This seems to be in accord with the recent removal of impredicative Set from Coq in favor of a predicative universe of sets, to gain set-theoretical models [Lee and Werner, 2011].

6.1 Intrinsic Typing and NbE

Why have we dismissed intrinsic typing for our investigation of NbE for dependent types? The reasons for this choice where three-fold:

First, computation, and thus, evaluation, is mostly oblivious of types. We are executing algorithms on concrete objects such as integers on machines, and the result of the execution is dependent just on the code of the algorithm and the values of the involved objects. Types exist in compiled code only in the form of *tags* that indicate the *basic* type of an object, like whether it is an integer value, a function pointer, or, in

the case of a language with dynamic dispatch, which class the present object belongs to. Higher types are absent[1] from run-time code and are only used by the compiler to chose the layout for data structures, stack frames, and the like, and to catch errors at compile-time. For these reasons, it makes sense to consider normalization by untyped evaluation, and separate the type-sensitive aspects such as η-expansion from the basic computation mechanism.

The second reason for choosing type assignment is that in presence of dependent types, intrinsic typing forces us to work with Kripke function spaces already for the formulation of evaluation. Functions need to be Kripke functions, taking a context extension (or morphism) as additional argument [Coquand and Gallier, 1990, Coquand, 1994, Danielsson, 2007]. This is in contrast to our type-assignment development, where the basic evaluation mechanism is unchanged, and Kripke structures only appear in the justification of NbE.

Finally, from a practical perspective, intrinsically dependently-typed NbE is yet unattainable. Our current, mature programming languages that serve as host languages to implement terms, types, and normalization, are simply-typed or polymorphically typed and cannot express intrinsic dependent typing.[2] Prototypical dependently-typed languages such as Agda [Norell, 2007] and Coq [INRIA, 2012] can express intrinsic dependent typing, but it is not clear yet whether full intrinsic typing is feasible at all, although its a long-standing and actively pursued research area [Danielsson, 2007, Chapman, 2009, McBride, 2010, Benton et al., 2012]. Boot-strapping implementations such as *Agda in Agda* or *Coq in Coq*, which are common for non-dependently programming languages, remain a grand challenge, even if self-representation has been achieved for fragments of dependently-typed languages, such as Barras' work on Coq [Barras and Werner, 1997, Barras, 1999] or Marten's work on LF [Martens and Crary, 2012].

By now, intrinsically-typed NbE for dependent types has withstood the challenge even for a rigorous pen-and-paper formulation. The notable exception is Danielsson's work [2007], however, it does not cover large eliminations.

For impredicativity, some progress has been made by Altenkirch, Hofmann, and Streicher. They present NbE for a variable-free, combinatory version of System F [1996], and there is an unpublished paper extending their category-theoretical approach to System F with variables [1997]. They express their developments in models of impredicative extensional type theory, which liberates them of some technical battles they would have to fight in just intensional type theory, but also removes them from an actual formalization in a system like Coq with impredicative Set. As a consequence, while the goal is to extract NbE for System F from their development, currently they can only present a hand-crafted SML program with the prospect of verifying it through a logical relation with the category-theoretical development. Still, this work is impressive, and it is a pity it has not been completed.[3]

[1] In distributed code, complex types structures might be present for marshalling and unmarshalling data of arbitrary structure. However, it is not the *types* themselves, but their *run-time representation*, which are computed with in this case.

[2] Simply intrinsic typing can be expressed in later versions of Haskell by GADTs (generalized algebraic data types), which are type-indexed data types, but also in Scala and other strongly-typed languages.

[3] Vestergaard's try on NbE for System F suffered the same fate [2001b]. He gives a normalization algorithm for System F in terms of a two-level polymorphic lambda-calculus, but its normalization

The remaining literature on intrinsically-type NbE seems to be confined to simple types. Besides the formalized work of Coquand [1994] we would like to emphasize work on extensional normalization for disjoint sum types (coproducts).

6.2 On NbE for the Extensional Treatment of Finite Choice

Altenkirch, Dybjer, Hofmann, and Scott [2001] present a construction of NbE for coproducts in category theory, Danvy [1996] and Balat, DiCosmo, and Fiore [2004] outline an algorithm using delimited continuations, and Barral [2008] uses exceptions in his NbE algorithm. In the presence of full extensionality for sum types, the description of normal forms is already involved [Altenkirch et al., 2001], not mentioning getting intuitions about algorithms using continuations. Barral's algorithm probably the most intuitive one, yet it has not been verified.

Full extensionality for sums models the syntactic sum type as the disjoint tagged union, $[\![S+T]\!] = [\![S]\!] + [\![T]\!]$, there is no separate of concept of neutrals at disjoint sum type. Reflecting a variable x of sum type $S+T$ into the semantics has to inject it either into $[\![S]\!]$ or $[\![T]\!]$, thus we have to make a choice which of the alternatives is valid for x. Since the value of x is unknown, we have to place a case distinction over x into the normal form before its first use, and normalize the remainder of the term twice, once with x being the left injection of a new variable, once more the right injection. Inserting these case distinctions eagerly is not feasible in the presence of unknown functions y to a sum type, because we cannot case on $y\,a$ for every possible argument a of y. Barral's algorithm installs an exception handler each time a new variable y is introduced whose type ends in a sum type. If the value of the fully applied $y\,\vec{a}$ is demanded, an exception is thrown carrying the arguments $\vec{a}$. Upon its catch, a case distinction on $y\,\vec{a}$ is inserted, and NbE is restarted. Termination of this process is somehow intuitively clear, but probably hard to establish rigorously. The full theoretical and practical exploration of NbE for extensional disjoint sums remains a research challenge.

For the very restricted case of simply typed lambda-calculus with only booleans as base type, a sound and complete NbE algorithm has been given by Altenkirch and Uustalu [2004]. The situation is simpler here, because the graph of any function is finite. Thus, it is possible to describe functions by first-order data structures, and the authors use binary trees called decision trees. Extensional equality of functions is decidable, thus, completeness of NbE is easy to prove. The authors express doubts that their treatment scales to extensions by unknown or infinite base types.

Whether we want full extensionality for disjoint sums and finite types in practice is debatable anyway. Note that extensionality for booleans allows us to show *by normalization* that each unary boolean function f is identical to its double self composition, $f \circ f \circ f$. However, this identity is not entirely trivial, it requires us to reflect on the possible values of f. There are only four possible boolean functions: the constant functions, the identity, and negation. Negation has order 2, and the other three functions are idempotent, thus $f^3 = f$ holds. Getting this result by normalization is a bit "spooky", and it is unclear if it would not irritate the users of an interactive proof assistant such as Agda or Coq more than it would benefit them. Furthermore, it does not scale to

proof remains unfinished.

large finite types such as the machine integers with 2^{32} cases for each variable—note that extensional treatment of integers could verify any arithmetic equation over integer variables or integer function variables *by normalization*, quite unimaginable with classical computing equipment.

6.3 Applications of NbE

NbE is applied successfully in practice, in fact, the "invention" of NbE was triggered by an implementation task, and it has proven to be a useful framework for efficient normalization.

Berger and Schwichtenberg [1991] implemented the first NbE algorithm in Scheme, in order to normalize terms of the Minlog system [Berger et al., 2011] without having to implement capture-avoiding substitution. Later extended to term rewriting [Berger et al., 2003], NbE is still the algorithm underlying computation in Minlog.

Danvy [1996, 1999] has coined the term *type-directed partial evaluation* for NbE in the context of partial evaluation for simply-typed functional programming languages such as ML.

The Coq system [INRIA, 2012] requires an efficient normalization algorithm for type checking programs and proofs, especially for proofs by reflection. Gregoire and Leroy [2002] describe normalization for terms of the Calculus of Inductive Constructions, Coq's core language, by first compiling the term to byte code of an abstract machine, then running the compiled code, and finally reading back a term in normal form. The abstract machine has been extended by *accumulators*, which correspond to our *neutral values*. Their "compiled reduction" approach is an instance of untyped NbE, as remarked in Abel [2009a]. Boespflug, Dénès, and Grégoire [2011] present a refinement that does not need compilation to a modified abstract machine but can be directly translated to OCaml. They achieve a tag-less representation of function values by considering accumulators as functions of infinite arity with a null code pointer.

Boespflug [2010] has implemented a NbE-inspired proof checker for Dedukti, a dependently typed logical framework with rewriting at the level of terms and types ($\lambda\Pi$ modulo [Cousineau and Dowek, 2007]). He compiles Dedukti signatures into a HOAS representation of dependent function types to speed up type checking.

Aehlig et al. [2012] implement untyped NbE for evaluation of functional programs specified in Isabelle/HOL. Their version of NbE is proven sound, but not complete. Normalization includes proper term rewriting rules such as the associativity of append, and even rewriting rules which are not left-linear.

The NbE framework could be useful to compile Agda terms such that they can compute with open expressions. This would probably help overcome the current performance problems in the Agda type checker.

6.4 Future Perspectives on NbE

Untyped NbE has secured its spot in proof assistants based on higher-order logic (e. g., Isabelle [Aehlig et al., 2012]) and type theory (e. g., Coq [Boespflug et al., 2011]). Typed NbE is present in Minlog (Berger et al. [2003]), but dispensable for all systems that can

postpone η-expansion until β-normalization has been completed. As far as we know, only singleton types and the difficult extensional sum types violate η-postponement and are really dependent on typed NbE. Our prototypical language MiniAgda [Abel, 2010b] has singleton types, but not the more mature systems Agda and Coq.

Correctness of NbE for the Calculus of Inductive Constructions (CIC) remains open. Possibly existing realizability models [Werner, 1994, Sacchini, 2011] could be adapted to the needs of NbE. The result of such an enterprise would be nothing short of the decidability of type checking for the CIC, an important backing for the increasingly popular Coq proof assistant.

Index of Notations

$()$ — Empty typing context, substitution, or environment . . . 6, 7

$(S._)^{\forall XT}$ — Semantic inverse-instantiation operator . . . 60

$(\rho, a/x)$, (ρ, a) — Update of environment ρ at x/0th variable with value a . . . 18

$(\sigma, s/x)$, (σ, s) — Substitution extension . . . 26

$(\underline{\lambda} t)\rho$ — Function closure . . . 21

$(x : S) \to T$ — Dependent function type from S to T . . . 41

$[A]$ — Extension of type value A . . . 46

$[\mathsf{D} \to \mathsf{D}]$ — Continuous-function space . . . 18

$[s]$ — Singleton substititution s for 0 . . . 26

$[\![T]\!]\rho$, $[\![T]\!](\sigma, \rho)$ — Interpretation of expression T as semantic type . . . 46, 61

$\forall^{X.T}$ — Semantic type quantification . . . 60

Γ — Typing context . . . 6

$\Gamma' \leq \Gamma$ — Context Γ' extends context Γ . . . 15, 59

$\Gamma(x)$, $\Gamma(i)$ — Context lookup . . . 24, 42

$\Gamma \vDash T$ — (Semantically) valid type expression T . . . 46

$\Gamma \vDash \sigma : \Delta$ — Semantic substitution typing . . . 26

$\Gamma \vDash \sigma = \sigma' : \Delta$ — Semantic substitution equality . . . 27, 46

$\Gamma \vDash t : T$ — Semantic typing . . . 16, 24, 46, 51

$\Gamma \vDash t = t' : T$ — Semantic term equality . . . 27, 46, 49

$\Gamma \vdash T = T'$ — Type T is equal to T' in context Γ . . . 42

$\Gamma \vdash T \leq T'$ — Type T is a subtype of T' in context Γ . . . 42

$\Gamma \vdash T \circledR A$ — Kripke logical type expression-value relation . . . 51

$\Gamma \vdash \sigma : \Delta$ — Well-typed substitution σ . . . 15, 59

$\Gamma \vdash \sigma : \Delta \circledR \rho$ — Kripke logical substitution-environment relation . . . 16, 51

$\Gamma \vdash t : T \circledR a \in A$ — Kripke logical expression-value relation . . . 51

$\Gamma \vdash t : T \circledR a$ — Kripke logical relation between well-typed term t and value a . . . 15

$\Gamma \vdash t : T$ — Well-typed term, typing relation . . . 6, 42, 59

$\Gamma \vdash t = t' : T$ — Typed definitional equality of t and t' . . . 8, 28, 59

$\Gamma \vdash u \rightrightarrows T$ — Typed neutral term u . . . 11

$\Gamma \vdash v \leftleftarrows T$ — Typed normal term v . . . 11

Symbol	Description	Pages
$\Lambda X t$	Term denoting type abstraction	57
Π	Dependent function space (semantic type)	45, 46, 48
$_ * _$	Concatenation (groupoid operation)	27, 48
$_ \bowtie _$	Merge operation in applicative groupoids	49
$_ \cdot _$	Application of values (partial)	20, 32, 44, 48, 58, 63
$_^{-1}$	Inverse (groupoid operation)	27, 48
β	Computation law	9, 28
$\bot$	Least semantic type in candidate space	25, 32, 44
$\top$	Greatest semantic type in candidate space	25, 32, 44
η	Extensionality law	9, 30
$\forall X T$	Polymorphic type	57
$\lambda x t$, $\lambda x\!:\!S.\, t$, λt	Term denoting function abstraction	6, 18
$(\!| t |\!)\rho$	Interpretation of expression t as value	46, 62, 63
$\hat{D}^{\sigma}$	Set of D type environments	61
$\uparrow^{T}$	Reflection of neutral at type T	10, 31, 44
$\uparrow^{\Gamma}$	Canonical environment for context Γ	31, 44, 64
$\downarrow^{T}$	Reification of value at type T	10, 31, 44
$[\![T]\!]$	Interpretation of type T	7, 24
$[\![\Gamma]\!]$	Interpretation of context Γ	7, 24, 46
$[\![\Gamma \vdash t : T]\!]$	Interpretation of typed term t	7
$[\![\sigma]\!]$	Interpretation of substitution σ as environment	27
$[\![\sigma]\!](\rho) \searrow \rho'$	Evaluation of substitution σ to environment ρ'	26
$[\![t]\!]$	Interpretation of term t in meta-language	8, 19, 20, 35
$[\![t]\!](\rho) \searrow a$	Evaluation relation, inductively defined	21
$\uparrow$	Shifting substitution	26
$\mathsf{R}^{\mathsf{ne}}_{n}\, e \searrow u$	Read-back relation for neutrals, inductively defined	21
$\mathsf{R}^{\mathsf{nf}}_{n}\, d \searrow v$	Read-back relation, inductively defined	21
$\mathsf{nf}_{n}(t) \searrow v$	Normalization relation	22
$\mathsf{rec}(d_z, d_s, d_n) \searrow d$	Evaluation of recursion, inductively defined	23
$\overline{T}$, $\overline{A}$	Greatest semantic type candidate for type T or A	33, 37, 47, 50, 60, 65
$\underline{T}$, $\underline{A}$	Least semantic type candidate for type T or A	33, 37, 47, 50, 60, 65
$\Vdash$	Type expression $T \Vdash \mathcal{A}$ or value $A \Vdash \mathcal{A}$ realizes $\mathcal{A}$	33, 60
ρ	Environment	7, 16
$\rho = \rho' \in [\![\Delta]\!]$	PER of environments	30

Symbol	Description	Pages
σ	Substitution	15
$\sigma(a)$	Transport a via weakening σ into new world	35
$\sigma : \Gamma' \leq \Gamma$	Weakening substitution σ with $\Gamma' \vdash \sigma : \Gamma$	34
$\sigma\,\tau$	Substitution composition	26
$\to$	Function type constructor	6, 22, 30, 36, 60
$\models \Gamma$	(Semantically) valid typing context Γ	46
$\vdash \Gamma$	Context Γ is well-formed	42, 59
ξ	Weak function extensionality law	30
$a = a' \in \mathcal{A}$	Partial equivalence relation (PER) membership	30
$a \sim a' \in \mathcal{A}$	PER structure on MAGs	50
$c : T$	Constant c can be assigned type T	42
$f \cdot a \searrow b$	Application relation, inductively defined	21
$r\,S$	Term denoting type application	57
$r\,s$	Term denoting function application	6, 18
$t[\sigma]$	Application of substitution σ to term t	16
$t[s/x]$	Capture-avoiding substitution of s for x in t	8
$t\,\sigma$	Explicit substitution	26
a	Value	7
$\mathcal{A}$	Semantic type: value set/PER	22, 27, 45, 48
$\mathcal{A}_\Gamma$	Kripke semantic type $\mathcal{A}$ at world Γ	35, 60
Abs	Embedding of functions into value set D	18
Agd	Set of applicative groupoid	48
App	Application of neutral value to value	18
$\mathsf{App}_\Gamma^{\forall XT}$	Type application in type structure	58, 61
$\mathsf{app}_\Gamma^{S \to T}$	Application in type structure	35, 58, 61
Base	Set of base type values	43
c	Term constant (function or constructor symbol)	6, 48
Cst^T	Set of constants of type T	6, 41
cst_Γ	Interpretation of constants in type structure	35
Cxt	Set of typing contexts	6, 41, 57
D	Set or domain of values	18, 22, 32, 43, 62
d	(Normal) value	18
D_Γ^Δ	Set of D-environments	35, 62
D^{ne}	Set of neutral values	18, 32, 62
D^{nf}	Set of normal values	32, 62

D^T_Γ	(Kripke) type(d applicative) structure	35, 58, 62
$\widehat{D}^T$	Kripke predicates on D^T (pre-candidates for type T)	36, 60
e	Neutral value in D^{ne}	18
$\mathcal{E}l$	Extension function, interpreting type codes as semantic types	45
Env	Set of environments	18, 20, 43, 62
E^T_Γ	Type substructure	35
Exp	Set of untyped terms (expressions)	18, 22, 41
f	(Function) value	7
$\mathcal{F}$	Family of semantic types	45, 46, 48
$\mathsf{Fresh}\ \lvert\Gamma\rvert$	Fresh type variable generator	63
$\mathsf{fresh}\ \lvert\Gamma\rvert$	Fresh term variable generator	63
Fun	Dependent function type constructor	41
G	Applicative groupoid	48
id	Identity substitution	26
$\overline{\underline{D}}$	A System F structure of realized types	61
Mag	Set of applicative subgroupoids with merge	50
N	Natural number type	6
$\mathcal{N}at$	Semantic type of natural numbers	22, 25, 30, 45
Ne	Set of neutral terms	11, 14, 18, 22, 41, 62
Nf	Set of normal terms	11, 14, 18, 22, 41, 62
$\mathsf{nf}(t)$	Normal form of term t	8, 20, 31, 40, 64
$\mathbb{N}_1$	Unit set	7
$\mathcal{P}(\mathsf{D})$	Set of value sets	45
Per	Set of partial equivalence relations on values	45
Rec	Type of primitive recursor	5, 43
rec	Term denoting primitive recursion	6, 32, 34
Rel	Set of relations on values	45
R^{ne}	Read-back of neutral value as neutral term	19, 32, 63
R^{nf}	Read-back of (normal) value as normal term	19, 32, 44, 63
$\mathcal{S}et_k$	kth semantic type universe	45, 46
Set	Universe of small sets	7
Set_k	kth universe of types	41
Subst	Set of substitutions	41, 57
suc	Term denoting the successor function	6, 22, 34
T	Type expression	6, 57

t	Term	6
Tm_{Γ}^{T}	Set of terms of type T in context Γ	6, 35
Ty	Set of type expressions	6, 57, 58
$\mathcal{T}ype$	Universe of all types	46
TyVar	Set of type variables	57
u	Neutral term	11, 18
$\hat{u}$	Liftable neutral term	14
Up	Embedding of neutral values into value set D	18
$\mathcal{U}$	A universe of types	47
v	Normal term	11, 18
$\hat{v}$	Liftable normal term	14
v_i	De Bruijn index i	18
X	Type variable	57
x	Placeholder for a term variable	6
$\hat{x}_{\Gamma}^{S}$	Liftable variable	15
x_k	De Bruijn level k	18
zero	Term denoting number 0	6, 22, 34

Bibliography

M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

A. Abel. Weak $\beta\eta$-normalization and normalization by evaluation for System F. In I. Cervesato, H. Veith, and A. Voronkov, editors, *15th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning, LPAR 2008, 22-27 November 2008, Doha, Qatar, Proceedings*, volume 5330 of *Lecture Notes in Artificial Intelligence*, pages 497–511. Springer-Verlag, 2008.

A. Abel. Typed applicative structures and normalization by evaluation for System Fω. In E. Grädel and R. Kahle, editors, *Computer Science Logic, 23rd international Workshop, CSL 2009, 18th Annual Conference of the EACSL, Coimbra, Portugal, September 7-11, 2009. Proceedings*, volume 5771 of *Lecture Notes in Computer Science*, pages 40–54. Springer-Verlag, 2009a. ISBN 978-3-642-04026-9.

A. Abel. Typed applicative structures and normalization by evaluation for System F^{ω} (full version). http://www.tcs.ifi.lmu.de/~abel/fomegaNbe.pdf, 2009b.

A. Abel. Towards Normalization by Evaluation for the $\beta\eta$-Calculus of Constructions. In M. Blume, N. Kobayashi, and G. Vidal, editors, *Functional and Logic Programming, 10th International Symposium, FLOPS 2010, Sendai, Japan, April 19-21, 2010. Proceedings*, volume 6009 of *Lecture Notes in Computer Science*, pages 224–239. Springer-Verlag, 2010a. ISBN 978-3-642-12250-7.

A. Abel. MiniAgda: Integrating sized and dependent types. In A. Bove, E. Komendantskaya, and M. Niqui, editors, *Workshop on Partiality And Recursion in Interactive Theorem Provers (PAR 2010), Satellite Workshop of ITP'10 at FLoC 2010*, volume 43 of *Electronic Proceedings in Theoretical Computer Science*, pages 14–28, 2010b.

A. Abel and T. Altenkirch. A partial type checking algorithm for Type:Type. *Electronic Notes in Theoretical Computer Science*, 229(5):3–17, 2011. Proceedings of the Second Workshop on Mathematically Structured Functional Programming (MSFP 2008).

A. Abel and T. Coquand. Untyped algorithmic equality for Martin-Löf's logical framework with surjective pairs. *Fundamenta Informaticae*, 77(4):345–395, 2007. TLCA'05 special issue.

A. Abel and B. Pientka. Explicit substitutions for contextual type theory. In K. Crary and M. Miculan, editors, *5th International Workshop on Logical Frameworks and Meta-languages: Theory and Practice (LFMTP 2010), Edinburgh, Scotland, UK,*

July 14, 2010, volume 34 of *Electronic Proceedings in Theoretical Computer Science*, pages 5–20, 2010.

A. Abel, K. Aehlig, and P. Dybjer. Normalization by evaluation for Martin-Löf type theory with one universe. In M. Fiore, editor, *Proceedings of the 23rd Conference on the Mathematical Foundations of Programming Semantics (MFPS XXIII), New Orleans, LA, USA, 11-14 April 2007*, volume 173 of *Electronic Notes in Theoretical Computer Science*, pages 17–39. Elsevier, 2007a.

A. Abel, T. Coquand, and P. Dybjer. Normalization by evaluation for Martin-Löf Type Theory with typed equality judgements. In *22nd IEEE Symposium on Logic in Computer Science (LICS 2007), 10-12 July 2007, Wroclaw, Poland, Proceedings*, pages 3–12. IEEE Computer Society Press, 2007b.

A. Abel, T. Coquand, and P. Dybjer. Verifying a semantic $\beta\eta$-conversion test for Martin-Löf type theory. In P. Audebaud and C. Paulin-Mohring, editors, *Mathematics of Program Construction, 9th International Conference, MPC 2008, Marseille, France, July 15-18, 2008. Proceedings*, volume 5133 of *Lecture Notes in Computer Science*, pages 29–56. Springer-Verlag, 2008. ISBN 978-3-540-70593-2.

A. Abel, T. Coquand, and M. Pagano. A modular type-checking algorithm for type theory with singleton types and proof irrelevance. *Logical Methods in Computer Science*, 7(2:4):1–57, May 2011.

K. Aehlig and F. Joachimski. Operational aspects of untyped normalization by evaluation. *Mathematical Structures in Computer Science*, 14(4):587–611, Aug. 2004.

K. Aehlig, F. Haftmann, and T. Nipkow. A compiled implementation of normalisation by evaluation. *Journal of Functional Programming*, 22(1):9–30, 2012.

M. S. Ager, D. Biernacki, O. Danvy, and J. Midtgaard. A functional correspondence between evaluators and abstract machines. In *Proceedings of the 5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, 27-29 August 2003, Uppsala, Sweden*, pages 8–19. ACM Press, 2003. ISBN 1-58113-705-2.

S. Allen. *A Non-Type-Theoretic Semantics for Type-Theoretic Language*. PhD thesis, Cornell University, 1987.

T. Altenkirch. Proving strong normalization of CC by modifying realizability semantics. In H. Barendregt and T. Nipkow, editors, *Types for Proofs and Programs, International Workshop TYPES'93, Nijmegen, The Netherlands, May 24-28, 1993, Selected Papers*, volume 806 of *Lecture Notes in Computer Science*, pages 3–18. Springer-Verlag, 1994. ISBN 3-540-58085-9.

T. Altenkirch and J. Chapman. Big-step normalisation. *Journal of Functional Programming*, 19(3-4):311–333, 2009.

T. Altenkirch and T. Uustalu. Normalization by evaluation for $\lambda^{\rightarrow 2}$. In Y. Kameyama and P. J. Stuckey, editors, *Functional and Logic Programming, 7th International*

Symposium, FLOPS 2004, Nara, Japan, April 7-9, 2004, Proceedings, volume 2998 of *Lecture Notes in Computer Science*, pages 260–275. Springer-Verlag, 2004. ISBN 3-540-21402-X.

T. Altenkirch, M. Hofmann, and T. Streicher. Categorical reconstruction of a reduction free normalization proof. In D. H. Pitt, D. E. Rydeheard, and P. Johnstone, editors, *Category Theory and Computer Science, 6th International Conference, CTCS '95, Cambridge, UK, August 7-11, 1995, Proceedings*, volume 953 of *Lecture Notes in Computer Science*, pages 182–199. Springer-Verlag, 1995. ISBN 3-540-60164-3.

T. Altenkirch, M. Hofmann, and T. Streicher. Reduction-free normalisation for a polymorphic system. In *11th Annual IEEE Symposium on Logic in Computer Science (LICS'96), 27-30 July 1996, New Brunswick, New Jersey, Proceedings*, pages 98–106. IEEE Computer Society Press, 1996.

T. Altenkirch, M. Hofmann, and T. Streicher. Reduction-free normalisation for System F. Available from `http://www.cs.nott.ac.uk/~txa/publ/f97.pdf`, 1997.

T. Altenkirch, P. Dybjer, M. Hofmann, and P. J. Scott. Normalization by evaluation for typed lambda calculus with coproducts. In *16th IEEE Symposium on Logic in Computer Science (LICS 2001), 16-19 June 2001, Boston University, USA, Proceedings*, pages 303–310. IEEE Computer Society Press, 2001.

L. Augustsson. Cayenne - a language with dependent types. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming (ICFP '98), Baltimore, Maryland, USA, September 27-29, 1998*, volume 34 of *SIGPLAN Notices*, pages 239–250. ACM Press, 1999. ISBN 0-58113-024-4.

V. Balat, R. D. Cosmo, and M. P. Fiore. Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums. In N. D. Jones and X. Leroy, editors, *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, pages 64–76. ACM Press, 2004. ISBN 1-58113-729-X.

H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North Holland, Amsterdam, 1984.

H. Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, 1991.

F. Barral. *Decidability for non-standard conversions in lambda-calculus.* PhD thesis, Ludwig-Maximilians-University Munich, 2008.

B. Barras. *Auto-validation d'un système de preuves avec familles inductives.* PhD thesis, Université Paris 7, 1999.

B. Barras and B. Werner. Coq in Coq. Available on the WWW, 1997.

G. Barthe and T. Coquand. Remarks on the equational theory of non-normalizing pure type systems. *Journal of Functional Programming*, 16(2):137–155, 2006.

N. Benton, C.-K. Hur, A. Kennedy, and C. McBride. Strongly typed term representations in Coq. *Journal of Automated Reasoning*, 49(2):141–159, 2012.

U. Berger and H. Schwichtenberg. An inverse to the evaluation functional for typed λ-calculus. In *Sixth Annual Symposium on Logic in Computer Science (LICS '91), July, 1991, Amsterdam, The Netherlands, Proceedings*, pages 203–211. IEEE Computer Society Press, 1991.

U. Berger, M. Eberl, and H. Schwichtenberg. Term rewriting for normalization by evaluation. *Information and Computation*, 183(1):19–42, 2003.

U. Berger, K. Miyamoto, H. Schwichtenberg, and M. Seisenberger. Minlog - a tool for program extraction supporting algebras and coalgebras. In A. Corradini, B. Klin, and C. Cîrstea, editors, *Algebra and Coalgebra in Computer Science - 4th International Conference, CALCO 2011, Winchester, UK, August 30 - September 2, 2011. Proceedings*, volume 6859 of *Lecture Notes in Computer Science*, pages 393–399. Springer-Verlag, 2011. ISBN 978-3-642-22943-5.

E. Bishop. *Foundations of Constructive Analysis.* Academic Press, New York, 1967. ISBN 4-87187-714-0.

M. Boespflug. Conversion by evaluation. In M. Carro and R. Peña, editors, *Practical Aspects of Declarative Languages, 12th International Symposium, PADL 2010, Madrid, Spain, January 18-19, 2010. Proceedings*, volume 5937 of *Lecture Notes in Computer Science*, pages 58–72. Springer-Verlag, 2010. ISBN 978-3-642-11502-8.

M. Boespflug, M. Dénès, and B. Grégoire. Full reduction at full throttle. In J.-P. Jouannaud and Z. Shao, editors, *Certified Programs and Proofs - First International Conference, CPP 2011, Kenting, Taiwan, December 7-9, 2011. Proceedings*, volume 7086 of *Lecture Notes in Computer Science*, pages 362–377. Springer-Verlag, 2011. ISBN 978-3-642-25378-2.

A. Bove, A. Krauss, and M. Sozeau. Partiality and recursion in interactive theorem provers: An overview. *Mathematical Structures in Computer Science*, 2013. To appear.

R. M. Burstall and B. W. Lampson. A kernel language for abstract data types and modules. In G. Kahn, D. B. MacQueen, and G. D. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 1–50. Springer-Verlag, 1984. ISBN 3-540-13346-1.

L. Cardelli. Typechecking dependent types and subtypes. In M. Boscarol, L. C. Aiello, and G. Levi, editors, *Foundations of Logic and Functional Programming*, volume 306 of *Lecture Notes in Computer Science*, pages 45–57. Springer-Verlag, 1986. ISBN 3-540-19129-1.

J. Chapman. Type theory should eat itself. *Electronic Notes in Theoretical Computer Science*, 228:21–36, 2009. Proceedings of the International Workshop on Logical Frameworks and Metalanguages: Theory and Practice (LFMTP 2008).

R. L. Constable, S. F. Allen, M. Bromley, R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing mathematics with the Nuprl proof development system.* Prentice Hall, 1986. ISBN 978-0-13-451832-9.

C. Coquand. From semantics to rules: A machine assisted analysis. In E. Börger, Y. Gurevich, and K. Meinke, editors, *Computer Science Logic, 7th Workshop, CSL '93, Swansea, United Kingdom, September 13-17, 1993, Selected Papers*, volume 832 of *Lecture Notes in Computer Science*, pages 91–105. Springer-Verlag, 1994. ISBN 3-540-58277-0.

T. Coquand. An analysis of Girard's Paradox. In *Proceedings, Symposium on Logic in Computer Science, 16-18 June 1986, Cambridge, Massachusetts, USA*, pages 227–236. IEEE Computer Society, 1986.

T. Coquand. An algorithm for type-checking dependent types. In *Mathematics of Program Construction. Selected Papers from the Third International Conference on the Mathematics of Program Construction (July 17–21, 1995, Kloster Irsee, Germany)*, volume 26 of *Science of Computer Programming*, pages 167–177. Elsevier, May 1996.

T. Coquand and P. Dybjer. Intuitionistic model constructions and normalization proofs. *Mathematical Structures in Computer Science*, 7(1):75–94, 1997.

T. Coquand and J. Gallier. A proof of strong normalization for the theory of constructions using a kripke-like interpretation. In G. Huet and G. Plotkin, editors, *Proceedings of the First Workshop on Logical Frameworks*, May 1990.

T. Coquand and G. P. Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, 1988.

T. Coquand and C. Paulin. Inductively defined types—preliminary version. In P. Martin-Löf and G. Mints, editors, *Proceedings of COLOG '88*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66. Springer-Verlag, 1988.

T. Coquand and M. Takeyama. An implementation of Type : Type. In P. Callaghan, Z. Luo, J. McKinna, and R. Pollack, editors, *Types for Proofs and Programs, International Workshop, TYPES 2000, Durham, UK, December 8-12, 2000, Selected Papers*, volume 2277 of *Lecture Notes in Computer Science*, pages 53–62. Springer-Verlag, 2000. ISBN 3-540-43287-6.

D. Cousineau and G. Dowek. Embedding pure type systems in the lambda-pi-calculus modulo. In S. R. D. Rocca, editor, *Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007, Paris, France, June 26-28, 2007, Proceedings*, volume 4583 of *Lecture Notes in Computer Science*, pages 102–117. Springer-Verlag, 2007. ISBN 978-3-540-73227-3.

K. Crary and S. Weirich. Flexible type analysis. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP '99), Paris, France*, volume 34 of *SIGPLAN Notices*, pages 233–248. ACM Press, 1999. ISBN 1-58113-111-9.

N. A. Danielsson. A formalisation of a dependently typed language as an inductive-recursive family. In T. Altenkirch and C. McBride, editors, *Types for Proofs and Programs, International Workshop, TYPES 2006, Nottingham, UK, April 18-21, 2006, Revised Selected Papers*, volume 4502 of *Lecture Notes in Computer Science*, pages 93–109. Springer-Verlag, 2007. ISBN 978-3-540-74463-4.

O. Danvy. Type-directed partial evaluation. In H.-J. Boehm and G. L. S. Jr., editors, *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, pages 242–257. ACM Press, 1996. ISBN 0-89791-769-3.

O. Danvy. Type-directed partial evaluation. In J. Hatcliff, T. Æ. Mogensen, and P. Thiemann, editors, *Partial Evaluation – Practice and Theory, DIKU 1998 International Summer School, Copenhagen, Denmark, June 29 - July 10, 1998*, volume 1706 of *Lecture Notes in Computer Science*, pages 367–411. Springer-Verlag, 1999. ISBN 3-540-66710-5.

O. Danvy, M. Rhiger, and K. H. Rose. Normalization by evaluation with typed abstract syntax. *Journal of Functional Programming*, 11(6):673–680, 2001.

N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972.

P. Dybjer. Internal type theory. In S. Berardi and M. Coppo, editors, *Types for Proofs and Programs, International Workshop TYPES'95, Torino, Italy, June 5-8, 1995, Selected Papers*, volume 1158 of *Lecture Notes in Computer Science*, pages 120–134. Springer-Verlag, 1996. ISBN 3-540-61780-9.

P. Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *The Journal of Symbolic Logic*, 65(2):525–549, 2000.

P. Dybjer and A. Filinski. Normalization and partial evaluation. In G. Barthe, P. Dybjer, L. Pinto, and J. Saraiva, editors, *Applied Semantics, International Summer School, APPSEM 2000, Caminha, Portugal, September 9-15, 2000, Advanced Lectures*, volume 2395 of *Lecture Notes in Computer Science*, pages 137–192. Springer-Verlag, 2000. ISBN 3-540-44044-5.

A. Filinski. A semantic account of type-directed partial evaluation. In G. Nadathur, editor, *Principles and Practice of Declarative Programming, International Conference, PPDP'99, Paris, France, September 29 - October 1, 1999, Proceedings*, volume 1702 of *Lecture Notes in Computer Science*, pages 378–395. Springer-Verlag, 1999. ISBN 3-540-66540-4.

A. Filinski. Normalization by evaluation for the computational lambda-calculus. In S. Abramsky, editor, *Typed Lambda Calculi and Applications, 5th International Conference, TLCA 2001, Krakow, Poland, May 2-5, 2001, Proceedings*, volume 2044 of *Lecture Notes in Computer Science*, pages 151–165. Springer-Verlag, 2001. ISBN 3-540-41960-8.

A. Filinski and H. K. Rohde. A denotational account of untyped normalization by evaluation. In I. Walukiewicz, editor, *Foundations of Software Science and Computation Structures, 7th International Conference, FOSSACS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, volume 2987 of *Lecture Notes in Computer Science*, pages 167–181. Springer-Verlag, 2004. ISBN 3-540-21298-1.

H. Friedman. Equality between functionals. In *Symposium on Logic Held at Boston, 1972-73*, volume 453, pages 22–37. Springer-Verlag, 1975.

F. Garillot and B. Werner. Simple types in type theory: Deep and shallow encodings. In K. Schneider and J. Brandt, editors, *Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007, Kaiserslautern, Germany, September 10-13, 2007, Proceedings*, volume 4732 of *Lecture Notes in Computer Science*, pages 368–382. Springer-Verlag, 2007. ISBN 978-3-540-74590-7.

H. Geuvers. A short and flexible proof of strong normalization for the Calculus of Constructions. In B. N. P. Dybjer and J. Smith, editors, *Types for Proofs and Programs, Int. Workshop TYPES '94*, volume 996 of *Lecture Notes in Computer Science*, pages 14–38, Båstad, Sweden, 1994. Springer-Verlag.

J.-Y. Girard. *Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieur.* Thèse de Doctorat d'État, Université de Paris VII, 1972.

J.-Y. Girard. Locus solum: From the rules of logic to the logic of rules. *Mathematical Structures in Computer Science*, 11(3):301–506, 2001.

J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1989.

K. Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931.

K. Gödel. Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes. *Dialectica*, pages 280–287, 1958.

H. Goguen. *A Typed Operational Semantics for Type Theory.* PhD thesis, University of Edinburgh, August 1994. Available as LFCS Report ECS-LFCS-94-304.

B. Grégoire and X. Leroy. A compiled implementation of strong reduction. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002*, volume 37 of *SIGPLAN Notices*, pages 235–246. ACM Press, Sept. 2002. ISBN 1-58113-487-8.

R. Harper and D. Licata. Mechanizing metatheory in a logical framework. *Journal of Functional Programming*, 17(4–5):613–673, July 2007.

R. Harper and F. Pfenning. On equivalence and canonical forms in the LF type theory. *ACM Transactions on Computational Logic*, 6(1):61–101, 2005. ISSN 1529-3785.

R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association of Computing Machinery*, 40(1):143–184, Jan. 1993.

M. Hofmann. Syntax and semantics of dependent types. In A. M. Pitts and P. Dybjer, editors, *Semantics and Logics of Computation*, volume 14, pages 79–130. Cambridge University Press, Cambridge, 1997.

M. Hofmann and T. Streicher. The groupoid model refutes uniqueness of identity proofs. In *Ninth Annual IEEE Symposium on Logic in Computer Science (LICS'94), 4-7 July 1994, Paris, France, Proceedings*, pages 208–212. IEEE Computer Society Press, 1994.

J. G. Hook and D. J. Howe. Impredicative strong existential equivalent to Type:Type. Technical report, Cornell University, 1986.

A. J. C. Hurkens. A simplification of Girard's paradox. In M. Dezani-Ciancaglini and G. D. Plotkin, editors, *Typed Lambda Calculi and Applications, Second International Conference on Typed Lambda Calculi and Applications, TLCA '95, Edinburgh, UK, April 10-12, 1995, Proceedings*, volume 902 of *Lecture Notes in Computer Science*, pages 266–278. Springer-Verlag, 1995. ISBN 3-540-59048-X.

INRIA. *The Coq Proof Assistant Reference Manual.* INRIA, version 8.4 edition, 2012.

G. Kahn. Natural semantics. In F.-J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, editors, *STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany, February 19-21, 1987, Proceedings*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer-Verlag, 1987. ISBN 3-540-17219-X.

P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4): 308–320, Jan. 1964. ISSN 0010-4620 (print), 1460-2067 (electronic).

G. Lee and B. Werner. Proof-irrelevant model of CC with predicative induction and judgmental equality. *Logical Methods in Computer Science*, 7(4), 2011.

C. Martens and K. Crary. LF in LF: Mechanizing the metatheory of LF in Twelf. In *Logical Frameworks and Metalanguages: Theory and Practice*, 2012.

P. Martin-Löf. An intuitionistic theory of types: Predicative part. In H. E. Rose and J. C. Shepherdson, editors, *Logic Colloquium '73*, pages 73–118. North-Holland, 1975.

P. Martin-Löf. Substitution calculus. Unpublished notes from a lecture in Göteborg, November 1992.

C. McBride. Outrageous but meaningful coincidences: Dependent type-safe syntax and evaluation. In B. C. d. S. Oliveira and M. Zalewski, editors, *Proceedings of the ACM SIGPLAN Workshop on Generic Programming, WGP 2010, Baltimore, MD, USA, September 27-29, 2010*, pages 1–12. ACM Press, 2010. ISBN 978-1-4503-0251-7.

R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, Aug. 1978.

J. Mitchell. *Foundations of Programming Languages.* MIT Press, 1996.

G. Nadathur. The suspension notation for lambda terms and its use in metalanguage implementations. *Electronic Notes in Theoretical Computer Science*, 67:35–48, 2002.

B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin Löf's Type Theory: An Introduction.* Clarendon Press, Oxford, 1990.

U. Norell. *Towards a Practical Programming Language Based on Dependent Type Theory.* PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, Göteborg, Sweden, Sept. 2007.

C. Paulin-Mohring. Inductive definitions in the system Coq - rules and properties. In M. Bezem and J. F. Groote, editors, *Typed Lambda Calculi and Applications, International Conference on Typed Lambda Calculi and Applications, TLCA '93, Utrecht, The Netherlands, March 16-18, 1993, Proceedings*, volume 664 of *Lecture Notes in Computer Science*, pages 328–345. Springer-Verlag, 1993. ISBN 3-540-56517-5.

G. Peano. *Arithmetices principia: nova methodo exposito.* Fratres Bocca, 1889.

A. M. Pitts. Nominal System T. In M. V. Hermenegildo and J. Palsberg, editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 159–170. ACM Press, 2010. ISBN 978-1-60558-479-9.

R. Pollack. Closure under alpha-conversion. In H. Barendregt and T. Nipkow, editors, *Types for Proofs and Programs, International Workshop TYPES'93, Nijmegen, The Netherlands, May 24-28, 1993, Selected Papers*, volume 806 of *Lecture Notes in Computer Science*, pages 313–332. Springer-Verlag, 1994.

J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference, Boston, Massachusetts*, pages 717–740, Aug. 1972.

J. C. Reynolds. Towards a theory of type structure. In B. Robinet, editor, *Programming Symposium*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425, Berlin, 1974. Springer-Verlag.

J. L. Sacchini. *On Type-Based Termination and Pattern Matching in the Calculus of Inductive Constructions.* PhD thesis, INRIA Sophia-Antipolis and École des Mines de Paris, 2011.

D. A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, 1986. ISBN 0-205-10450-9.

U. Schöning. *Theoretische Informatik - kurzgefasst.* Spektrum Akademischer Verlag, 4. Auflage (korrig. Nachdruck 2003) edition, 2003. ISBN 3827410991.

A. Setzer. An introduction to well-ordering proofs in Martin-Löf's type theory. In G. Sambin and J. Smith, editors, *Twenty-five years of constructive type theory*, pages 245 – 263, Oxford, 1998. Clarendon Press.

T. Skolem. Begründung der elementaren Arithmetik durch die rekurrierende Denkweise ohne Anwendung scheinbarer Veränderlichen mit unendlichem Ausdehnungsbereich. *Videnskapsselskapets skrifter. 1. Matematisk-naturvidenskabelig klasse*, (6), 1923.

C. A. Stone and R. Harper. Extensional equivalence and singleton types. *ACM Transactions on Computational Logic*, 7(4):676–722, 2006. ISSN 1529-3785.

J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, MA, USA, 1977. ISBN 0262191474.

W. W. Tait. Intensional interpretations of functionals of finite type I. *The Journal of Symbolic Logic*, 32(2):198–212, 1967.

R. Vestergaard. The simple type theory of normalization by evaluation. *Electronic Notes in Theoretical Computer Science*, 57:163–183, 2001a.

R. Vestergaard. The polymorphic type theory of normalisation by evaluation. Available on the author's home page, 2001b.

K. Watkins, I. Cervesato, F. Pfenning, and D. Walker. A concurrent logical framework I: Judgements and properties. Technical report, School of Computer Science, Carnegie Mellon University, Pittsburgh, 2003.

B. Werner. A normalization proof for an impredicative type system with large eliminations over integers. In B. Nordström, K. Petersson, and G. Plotkin, editors, *Proceedings of the 1992 Workshop on Types for Proofs and Programs, Båstad, Sweden, June 1992*, pages 341–357, 1992.

B. Werner. *Une Théorie des Constructiones Inductives.* PhD thesis, Universite Paris 7, 1994.

A. N. Whitehead and B. Russell. *Principia Mathematica.* Cambridge University Press, 1910–1913.

H. Xi and F. Pfenning. Dependent types in practical programming. In *Conference Record of POPL 99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas*, pages 214–227, New York, NY, 1999.

B. A. Yorgey, S. Weirich, J. Cretin, S. L. P. Jones, D. Vytiniotis, and J. P. Magalhães. Giving haskell a promotion. In B. C. Pierce, editor, *Proceedings of TLDI 2012: The Seventh ACM SIGPLAN Workshop on Types in Languages Design and Implementation, Philadelphia, PA, USA, Saturday, January 28, 2012*, pages 53–66. ACM Press, 2012. ISBN 978-1-4503-1120-5.

C. Zenger. Indexed types. *Theoretical Computer Science*, 187(1-2):147–165, 1997.