Constructions, Inductive Types and Strong Normalization

Thorsten Altenkirch

Ph. D.
University of Edinburgh
1993

Abstract

This thesis contains an investigation of Coquand's Calculus of Constructions, a basic impredicative Type Theory. We review syntactic properties of the calculus, in particular decidability of equality and type-checking, based on the equality-as-judgement presentation. We present a set-theoretic notion of model, CC-structures, and use this to give a new strong normalization proof based on a modification of the realizability interpretation. An extension of the core calculus by inductive types is investigated and we show, using the example of infinite trees, how the realizability semantics and the strong normalization argument can be extended to non-algebraic inductive types. We emphasize that our interpretation is sound for *large eliminations*, e.g. allows the definition of sets by recursion. Finally we apply the extended calculus to a non-trivial problem: the formalization of the strong normalization argument for Girard's System F. This formal proof has been developed and checked using the LEGO system, which has been implemented by Randy Pollack. We include the LEGO files in the appendix.

Acknowledgements

In November 1988 I met Rod Burstall in the Cafe Hardenberg in Berlin, and told him that I would like to do a PhD in Edinburgh. He made this possible and provided scientific and spiritual help and assistance over all the time. I particularly appreciate his pragmatic and intuitive approach towards theoretical computer science.

I would not have learnt Type Theory so easily if I had not been able to use Randy Pollack's LEGO system. The discussions and talks in the LEGO club have been extremely valuable and I want to thank all its participants. I profited from many controversial discussions with my second supervisor Zhaohui Luo and I appreciate his demands towards a more rigorous presentation of my results. The interaction with Healfdene Goguen has also been extremely helpful, he often provided essential feedback by spotting weak points in my constructions.

The person who had the greatest impact on my scientific thinking is my friend and colleague Martin Hofmann, with whom I had uncountable discussions focussing on Types, Logic and Categories. It is hard not to be impressed by his brilliance.

I profited very much from the vibrant research environment at the Laboratory of Foundations of Computer Science. I would like to thank Alex Simpson and Claudio Hermida for many inspiring discussions about Category Theory and related topics.

The contact with other researchers which has been made possible by the Types BRA has been very inspiring. I would like to thank Thierry Coquand, Herman Geuvers, Stefano Berardi, Eike Ritter and Benjamin Werner for many interesting

discussions. I especially want to thank Peter Dybjer and Thomas Streicher for comments on a draft version of this thesis.

I would like to thank the SIEMENS AG for supporting me with a studentship and for providing valuable feedback during my visits in Neu-Perlach.

I also want to thank all the members, animals and friends of Bath Street Housing Coop. I certainly would not have enjoyed my time in Edinburgh so much if I had not lived at Bath Street.

Declaration

I hereby declare that this thesis has been composed by myself, that the work is my own, and the ideas and results that I do not attribute to others are due to myself.

Table of Contents

1. Introduction **9**

1.1 The essence of Type Theory 9

1.2 The Calculus of Constructions 12

1.3 Why semantics ? 14

1.4 Strong normalization 15

1.5 The use of categories 16

1.6 Inductive types 17

1.7 Formal proof 18

1.8 Overview 19

2. The Calculus of Constructions **21**

2.1 The judgement presentation of CC 21

2.2 Basic Properties 27

2.3 Reduction and decidability 29

2.3.1 General reduction 30

2.3.2 Curry reduction 32

2.3.3 Tight reduction 33

2.3.4 Decidability 36

2.4 Type reconstruction 39

2.5 The conversion presentation 44

3. Semantics and strong normalization for the core calculus **47**

3.1 Basic notations 47

3.2 CC-structures 50

3.3 Proof irrelevance semantics 58

3.4 Realizability interpretation 60

3.4.1 Partial combinatory algebras 61

3.4.2 Interpreting dependent types 63

3.4.3 Interpreting constructions 64

3.5 Strong Normalization 67

3.5.1 Properties of SN 68

3.5.2 Saturated λ-sets 70

4. Inductive types **76**

4.1 Definition of trees 78

4.2 D-set semantics 79

4.2.1 Formation- and introduction-rules 79

4.2.2 Elimination- and computation-rules 84

4.3 Saturated Λ-sets 87

4.3.1 Syntactic properties 87

4.3.2 Formation and introduction-rules 89

4.3.3 Elimination- and computation-rules 91

4.3.4 Strong normalization 93

5. Application: Proving SN for System F **95**

5.1 Using LEGO 96

5.1.1 The Type Theory 96
5.1.2 The Logic 97
5.1.3 Inductive types 99
5.2 A guided tour through the formal proof 102
5.2.1 The untyped λ-calculus 102
5.2.2 Strong Normalization 104
5.2.3 System F 105
5.2.4 Candidates 107
5.2.5 Proving strong normalization 110
5.3 Alternatives and extensions 112
5.3.1 Extracting a normalization function 112
5.3.2 Saturated Sets 113
5.3.3 Reduction for Church terms 114

6. Conclusions and further work **116**

A. General μ-types **118**

A.1 Telescopes 118
A.2 Syntax 119
A.3 Rules 120
A.4 Examples 122

B. LEGO code **125**

B.1 `load-simple.l` 126
B.2 `load-f.l` 127
B.3 `basic.l` 128

B.4 `lambda.l` 137

B.5 `snorm.l` 147

B.6 `simple.l` 152

B.7 `sn-simple.l` 155

B.8 `f.l` 158

B.9 `sn-f.l` 166

Bibliography **173**

Index of Symbols **180**

Chapter 1

Introduction

1.1. The essence of Type Theory

The main purpose of the formalization of reasoning is to increase the confidence into its results. This is reflected in different areas: in the engineering disciplines fixed schemes and methods have been developed to verify the safety of a design, e.g. the static safety of a bridge. In mathematics logical reasoning has been formalized to make precise what can be accepted as a proof.

However, the costs connected with a complete formalization are extremely high, the effort to to formally verify a construction is much higher than the one required for its initial development. This is the reason that, despite its benefits, complete formalization is rare. The most typical example for this is mathematics: being well trained in formal reasoning, most mathematician believe that their results could be completely formalized; however this is almost never done. [1]

An area where the formal validation of design would be most useful are complex information processing systems, i.e. computer systems. Their intended behaviour is completely understood and in principle there is no problem to formalize their design. On the other hand given the ever growing importance of those systems

[1] One of the few exceptions is the AUTOMATH project [dB80], which can be viewed as a direct predecessor of modern Type Theory.

and their increasing complexity improved validation procedures [2] are becoming more and more important.

Computer science is not only of interest for the proliferation of verification tasks it also offers an essentially new solution for a fundamental problem of any validation process: How do we know that a validation, i.e. a formal correctness proof is correct? The task of *checking* whether a formal verification is valid can be completely mechanized and be performed by a computer program with a qualitatively higher accuracy.

In this situation we are confronted with a new task: we have to design systems for formal reasoning, which can be completely formalized and implemented on a computer. This requires a particular attention to every syntactic detail of the system which is not known in conventional logic. It is of special importance that the system is conceptually simple and allows a compact and straightforward representation of the concepts required for the verification. Otherwise we will lose confidence in the results because the system may be unsound. We also want the system to be universal enough that we are able to use it for a wide spectrum of different verification tasks.

Although there is obviously not one simple answer to such a diverse set of requirements, we believe that the study of Type Theory is especially useful in this context. By Type Theory we understand a diversity of concepts and systems most typically expressed in the work initiated by Per Martin-Löf. [3]

To us the essence of modern Type Theory is the unification of two apparently different aspects of formal reasoning: the membership of an object in a collection and the relation between a proof and the proposition it verifies. Martin-Löf in [Mar75], p.73 expresses this as follows:

---

[2] This certainly includes testing. There are different inherent limitations in both strategies: verification and testing.

[3] Apart from Martin-Löf's own work e.g. [Mar75], [Mar84], good accounts can be found in [BCMS89] and [NPS90].

The language of the theory is richer than the language of traditional intuitionistic systems in permitting proofs to appear as parts of propositions so that the propositions of the theory can express properties of proofs (and not only individuals, like in first order predicate logic).

This is also called the *proposition as types paradigm.*

When verifying a simple program we observe that its verification follows its computational structure. Moreover we may observe that that the proof has a similar, although finer, structure as the object it verifies. Finally we may realize that the proof essentially corresponds to a computational object annotated with additional information. We have arrived at the proposition as types paradigm.

From a conventional point of view we may think that computational structures and the accompanying proof principles are separate. From a type-theoretic viewpoint we would say that a type-theoretic construct has a computational aspect. This essentially simplifies the design of a reasoning system.

It has been observed that Type Theory can be non-conservative over a corresponding conventional logic. [4] However, we do not consider this as a defect of Type Theory. It seems rather to be related Martin-Löf's statement about having a *richer theory.* This is already reflected in the fact that for example the axiom of choice is provable in Type Theory, whereas it has to be introduced as an explicit assumption in a conventional logic.

The close association of proof and program has also a number of practically important consequences. First of all there is the problem that a type-theoretic construction also contains computationally irrelevant parts. Although it is not yet clear to what degree the separation process can be automated, this does not seem to be a fundamental problem because the user can explicitly mark the computa-

[4] In the case of CC see [Geu89].

tionally irrelevant parts. [5] This corresponds to our view that a type-theoretic expression is a program annotated with additional information.

On the other hand it seems possible to use this additional information not only for verification but also to improve compilation. Having more information at compile-time makes it possible to rule out certain cases of runtime errors, i.e. we can omit runtime checks and thereby speed up execution. We also imagine finer type structures which make the resource use explicit; an example would be linear types which can be used to avoid garbage collection. [6]

1.2. The Calculus of Constructions

The Calculus of Constructions (CC) was introduced by T. Coquand and Huet ([CH88],[Coq85]). It can be viewed as a unification of Girard's impredicative system F^{ω} and dependent types, which are the base of Martin-Löf's Type Theory. When Martin-Löf initially proposed a Type Theory [Mar71] he also attempted to capture Girard's system. However, it turned out that this system was inconsistent because it was possible to encode System U in it. Subsequently Martin-Löf avoided this problem by restricting himself to a predicative theory. In a way we may consider CC as a fix to an early problem in the formulation of Type Theory.

Since its introduction CC received a lot of attention because it can be viewed as a basic (impredicative) Type Theory. Based on work by Berardi Barendregt [Bar92] investigated the syntactic fine structure of the calculus thereby relating simply typed λ-calculus ($\lambda^{\rightarrow}$), System F and F^{ω}, a core calculus of dependent types (the logical framework or LF) and CC in a cube (figure 1.2). Based on the cube the notion of *Pure Type Systems* (PTS) has been developed which is a syntactic generalization of the calculus of constructions and related systems [7]

---

[5] E.g. see [PM93a,PMW93].

[6] E.g. see [Wad91], but here only non-dependent types are considered.

[7] See [Geu93] or [Bar92].

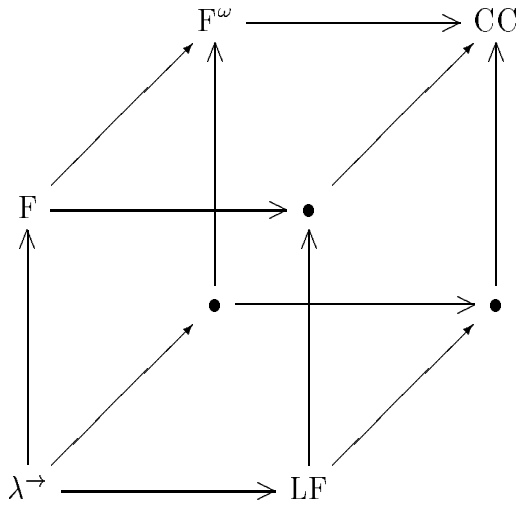


Figure 1–1: Barendregt's cube

Luo extends CC to the Extended Calculus of Constructions (ECC) by adding universes and Σ-types [Luo90]. This calculus is also the standard type-theory used in the LEGO system [LP92]. A related system is the specification language Gallina which is implemented in the Coq-system [D$^+$91].

The semantics of CC has been studied by a number of authors from a categorical point of view, e.g. see [HP89], in [Ehr89] the notion of a dictos is introduced, and in [Jac91] the more general notion of a CC-category is used. A very natural semantics based on the concept of Realizability is the ω-set semantics. In [Str89] a mild generalization (D-sets) is investigated in great detail and used to show some independence results. [8]

It should be noted that there are two different ways to present a type theory: a presentation based on conversion as in the presentation of PTS or a presentation where equality is a judgement, this is usually used in the presentation of Martin-Löf Type Theory. The conversion presentation is more compact and closer to an implementation whereas the judgement presentation can be better understood semantically. We consider the conversion presentation rather as a shorthand for

[8] For our own account in terms of CC-structures see section 3.4.

the judgement presentation. This is justified by the work in [Coq91] where an equivalence of the two presentations is established. [9]

1.3. Why semantics ?

Type Theory emphasizes the syntactic aspect of reasoning. It has been argued that the meaning of a system can be reduced to its syntactic presentation. Although this is a valid point of view, the first question one may raise is whether this reflects the way we *understand* a logical system. When we are learning or developing a new theory or extending an old one, we try to reflect our intuitive notion of mathematical truth and objects. To analyze a system semantically and to assign a *denotation* to syntactic objects is a way to make this precise. Certainly, as Martin-Löf pointed out [10], this is merely a translation, e.g. we translate Type Theory in Set Theory. However, we believe that such a translation can be quite meaningful and reflects our intuitive understanding of a system.

Here we also want to stress another use of semantics: semantic notions are extremely useful when we want to prove properties about our system, i.e. about our syntactic presentation.

A rule or an axiom scheme is derivable when it can be constructed just by a schematic application of the inference rules, i.e. by purely syntactic reasoning. Derivable rules or axiom schemes will be true in every sound interpretation by the very definition of soundness.

However, not many interesting properties are derivable. Usually we are not so much interested in properties which hold for all interpretations but more in some which are true for the syntax, i.e. the initial interpretation. Such properties are called *admissible*. A typical example for admissible properties are consistency

---

[9] Note that Coquand verifies this fact only for Martin-Löf Type Theory, i.e. a predicative theory.

[10] [Mar84], pp.69

properties: logical consistency (not all propositions are provable) or equational consistency (not all equalities hold).

One way to show consistency which can also be applied to other admissible properties is to construct a sound interpretation. If we want to establish logical soundness we just have to give a sound interpretation such that a particular type is denoted by the empty set (compare with theorem 3.3.4). The main effort goes into verifying that the interpretation is indeed sound. This justifies the slogan that *we use semantic methods to show properties of syntax.*

We will use this idea to give a new version of a Strong Normalization proof for CC which can also be extended to include inductive types with *large eliminations*. We will actually construct this strong normalization argument as a modification of the D-set semantics.

1.4. Strong normalization

The property of strong normalization will play the role of a red thread in this thesis. First we will use it to illustrate how syntactic properties can be proven by semantical construction. Then we will note that this view gives us a proof which can be easily generalized to a stronger system with inductive types and large eliminations. Finally we will use a strong normalization proof as an example of the development of a non-trivial theorem in the Type Theory we have presented.

Strong normalization is quite an essential property of a Type Theory: it does not only entail a number of decidability properties (decidability of equality and decidability of type checking) but also some other useful properties, e.g. note that the proof of the essential theorem 2.4.1 requires strong normalization. [11]

Less pragmatically strong normalization and its verification is interesting to us because it lies on the borderline of semantics and syntax and therefore, in a way, reflects the spirit of Type Theory.

---

[11] There are alternatives to strong normalization — see chapter 6, point 1.

It should be noted that the strong normalization proof for CC has been considered as notoriously difficult, e.g. see the historic account in [CG90]. We attempt to debunk this myth by clearly relating the SN proof to a standard model construction, i.e. the D-set semantics, and avoiding most of the usual syntactic complications. It is also interesting to note that in our development we avoid the use of Kripke-structures to present the semantics. Another feature of this construction is the possibility to extend it to the case of inductive types with large eliminations — compare this to the more conventional construction presented in [Wer92].

1.5. The use of categories

The machinery of Category Theory has been used and proven useful for the semantic investigation of Type Theories. E.g. this has been extensively studied in the work of Thomas Streicher [Str89] [12] and Bart Jacobs [Jac91]. Moreover in [Rit92] the categorical semantics of CC is used directly as a starting point for an implementation.

It has been noted [13] that the naive use of categorical notions does not necessarily produce a sound interpretation of the syntax. Often we have to introduce additional assumptions (e.g. split fibrations, equality instead of natural isomorphisms) to achieve this. We also note that constructing categorical interpretations often means that we have to *represent* element related concepts in terms of arrows. E.g. to model Π-types in the D-set semantics categorically we have to show that the pullback functor has a right adjoint. In our presentation we interpret Π-types as the subset of the set-theoretic dependent functions which have a realizer. Although equivalent to the categorical construction this presentation seems more natural.

---

[12] Streicher's thesis has been published as a monograph [Str91]. In future we will refer to the monograph only.

[13] E.g. see [ACCL90]

Given that our main motivation for semantics is to show admissible properties of the system and to do this we have to give sound interpretations, I decided not to use categories when constructing interpretations. However, it should be noted that the structures we define are heavily influenced by the corresponding categorical constructions.

I consider it as desirable to improve the categorical understanding of Type Theory and I believe that it should be possible to obtain a better agreement between categorical models and syntax. Some of the results presented here regarding inductive types in Type Theory should be generalized and rephrased in terms of Category Theory.

1.6. Inductive types

CC and related systems are called *pure* type systems because the only type constructor they contain is the function type or its generalization the Π-type. The systems used in practice are usually impure and have some mechanism to represent *inductive types*. Indeed, when looking at an example (like our own development in chapter 5) it turns out that inductive types become the workhorse of the development.

In an impredicative system like CC it is possible to encode inductive types (e.g. see [Alt90]). However, these encodings can be only used for computations, they lack the propositional *strength*, i.e. the induction axiom is not derivable. We could introduce an induction axiom as a logical assumption into the system but this would destroy the fundamental symmetry of Type Theory, because we have no computation rules for these axioms.

The notion of inductive types is well developed in Martin-Löf's Type Theory (e.g. see [Dyb91]). Here inductive types are considered as a primitive notion; induction and primitive recursion appear as two different aspects of the same type-theoretic concept. It seems sensible to integrate impredicativity and inductive types. In the context of the NuPRL system this has been investigated by Mendler [Men88]. In the context of intensional Type Theory this has been the subject of a

number of more recent investigations, e.g. see [CP89,Fu92,Gog93] — see also the discussion in chapter 4.

Inductive types roughly correspond to the `datatype` construct in ML or more formally can be viewed as the solution to some domain-equations. However, not every domain-equation corresponds to an inductive type: we restrict ourselves to either positive or strictly positive equations. Otherwise we would not only have a partial logic but also a propositionally inconsistent theory. However, as pointed out by Martin Hofmann [Hof93b] general domain equations can be investigated in a Type Theory which allows the definition of dependent types by recursion (*large eliminations*).

It is also interesting to observe that the notion of dependent inductive types or inductive families captures definitions by the initial semantics of a set of Horn clauses. [14] This is yet another example where the type-theoretic symmetry, i.e. the proposition as types paradigm, works very well.

Although inductive types only allow primitive recursion, the concept of general recursion can very well be represented in this framework as well-founded recursion, which can be defined using inductive types in a natural way. [15] It is interesting to note that this is a case where a higher-order (i.e. non-algebraic) inductive definition is essential.

1.7. Formal proof

Much of the material presented here has evolved around an experiment in proof-formalization: the formalization of the Strong Normalization argument for System F in LEGO (chapter 5). This raises the question whether the system in which this proof has been done is consistent. Although we do not answer this question in detail (e.g. verifying the proof checker formally), we develop sufficient material to

---

[14] Actually a generalization of Horn clauses to a higher-order case.

[15] See [Dyb92a], section 5.2.2.

see that the underlying Type Theory is consistent. There is yet another connection between the experiment and the material of the other chapters: by having done the formal development in such a great detail we have gathered some insights about Strong Normalization arguments, which were helpful when doing the proofs in chapters 3 and 4.

1.8. Overview

In chapter 2.1.2 we review the Calculus of Constructions. Our primary presentation is the judgement presentation using explicitly typed terms. Exploiting the strong normalization property we will prove a number of properties about the system and the equivalence of different presentations.

We continue with a semantic analysis of the basic calculus in chapter 3. For this purpose we will introduce LF-structures and CC-structures. We will then verify that CC-structures always give rise to a sound interpretation (theorem 3.2.10). We apply CC-structures to three different model constructions: the proof-irrelevance semantics section 3.3, the D-set semantics section 3.4 and the saturated Λ-set semantics section 3.5.2. The last one allows us to establish strong normalization as a corollary (corollary 3.5.12). A preliminary version of this work has been presented in [Alt93b].

Using the example of general trees, i.e. a non algebraic inductive type, we will show in chapter 4 how the D-sets semantics and the strong normalization argument can be extended to inductive types. We present a general notion of μ-types capturing most of the examples discussed here in the appendix, A.

In chapter 5 we apply CC extended by inductive types to a concrete example: the formal verification of a strong normalization proof for System F. This devel-

opment has been completely checked by the LEGO system [16] and the complete code can be found in the appendix B. This work has been presented in [Alt93a].

Let us summarize what we consider as the central points of this thesis:

1. We present a new strong normalization proof for CC based on a modification of the Realizability semantic.

2. We investigate decidability and type reconstruction for the judgement presentation of CC using the previous result.

3. We describe a new semantic structure to interpret CC (CC-structures) which is an alternative to categorical notions of model.

4. We extend realizability semantic and our strong normalization argument to inductive types (using a non-trivial example) with large eliminations.

5. We show the usefulness of the theory of CC extended by inductive types by completely formalizing a non-trivial example: the strong normalization proof for System F.

---

[16] For a discussion of the relation between the LEGO implementation and the Type Theory presented in this thesis see section 5.1.

Chapter 2

The Calculus of Constructions

In this chapter we will present CC and develop some of its metatheory. We diverge from the original presentation of CC (e.g. [CH88]) in that we use the equality-as-judgement presentation. We will exploit a semantic result which we are going to show in the next chapter — strong normalization of stripped typable terms (Curry terms) — to establish decidability and to justify a more implicit presentation (Church syntax). We will also show the equivalence of the judgement and the conversion presentation (for pure CC without the η-rule). We use a particular notion of reduction — *tight reduction* — which is essential to our approach. This presentation should be compared to [Rit92] where essentially the same goals are achieved using categorical combinators.

2.1. The judgement presentation of CC

In our presentation we largely follow [Str91], i.e.:

- We use equality as a judgement.

- We do not confuse types and terms, i.e. we avoid chains of colons. Therefore we have to introduce an explicit reflection operator El and differentiate between Π for types and $\forall$ for Set.

- We consider a calculus with explicitly typed application and show later that we can drop the annotations. This particularly simplifies the definition of an interpretation and is also exploited in the definition of tight reduction. [1]

We diverge form Streicher in the following ways:

- We use de-Bruijn-indices for the presentation of the system. However, we will exploit the usual convention that terms with named variables are a shorthand for the de-Bruijn-presentation.

- We also mark λ-abstractions with their codomain.

- We omit a number of structural rules, (see [Str91], pp. 160) like CONTTHIN, or CREFL because they are admissible in our presentation.

2.1.1. DEFINITION (Syntax). We define contexts ($\Gamma, \Delta \in \mathrm{Co}$), types ($\sigma, \tau, \rho \in \mathrm{Ty}$), terms ($M, N \in \mathrm{Tm}$) and constructions ($C, D \in \mathrm{Cn}$) as the union of types and terms as follows ($i, j, k \in \omega$):

$$\begin{array}{rcl} \mathrm{Co} & ::= & \bullet \mid \Gamma.\sigma \\ \mathrm{Ty} & ::= & \Pi\sigma.\tau \mid \mathrm{Set} \mid \mathrm{El}(M) \\ \mathrm{Tm} & ::= & i \mid \lambda\sigma(M)^{\tau} \mid \mathrm{app}^{\sigma.\tau}(M,N) \mid \forall\sigma.M \\ \mathrm{Cn} & ::= & \mathrm{Ty} \mid \mathrm{Tm} \end{array}$$

We always use de-Bruijn-indices [2] in our presentation of syntax: we represent variables by natural numbers which corresponds to the binding depth. By doing so we identify α-congruent terms. This choice also reflects our semantic intuition that variables are essentially projections out of a context. However, when presenting

---

[1] See the discussion in [Str91], pp. 177.

[2] They have been introduced in [dB72].

a particular term, we feel free to use variables with the obvious translation into de-Bruijn-indices, i.e we will use $\Pi x : \sigma.M[x]$, $\lambda x : \sigma(M[x])^\tau$ and $\forall x : \sigma.M[x]$. We also introduce the following abbreviations: $\sigma \to \tau \equiv \Pi\sigma.\tau^+$ and $\sigma \to A \equiv \forall\sigma.A^+$.

We introduce a whole bag of notations concerning weakening, substitution and contexts:

NOTATION. [Weakening and substitution] C^{+n} for weakening and $C[N]^n$ for substitution (see figure 2–1) — observe that $\sigma[N]^n \in \mathrm{Ty}$. The argument n represents the number of bound variables, if $n = 0$ then it will be omitted. We also define repeated weakening and parallel substitution for arbitrary constructions:

$$\begin{aligned} C^{\times 0} &= C \\ C^{\times(n+1)} &= (C^{\times n})^+ \\ C[\bullet] &= C \\ C[D\vec{D}] &= (C[D^{\times|\vec{D}|}])[\vec{D}] \end{aligned}$$

The intution behind M^+ is that all free variables are increased by one. This can be used to express the usual side conditions on free variables, e.g. instead of x is not free in M we use M^+ which entails that M cannot *see* the first free variable.

NOTATION. [Operations on contexts] We denote the concatenation of contexts by $\Gamma.\Delta$, which is defined:

$$\begin{aligned} \Gamma.\bullet &= \Gamma \\ \Gamma.(\Delta.\sigma) &= (\Gamma.\Delta).\sigma \end{aligned}$$

We define the length of a context $|\Gamma|$:

$$\begin{aligned} |\bullet| &= 0 \\ |\Gamma.\sigma| &= |\Gamma| + 1 \end{aligned}$$

and a projection operation $\Gamma(i)$ for all $i < |\Gamma|$:

$$\begin{aligned} \Gamma.\sigma(0) &= \sigma^+ \\ \Gamma.\sigma(i+1) &= \Gamma(i)^+ \end{aligned}$$

	Weakening	Substitution
Co	$\bullet^+ = \bullet$ $(\Gamma.\sigma)^+ = \Gamma^+.\sigma^{+\lvert\Gamma\rvert}$	$\bullet[N] = \bullet$ $(\Gamma.\sigma)[N] = \Gamma[N].\sigma[N]^{\lvert\Gamma\rvert}$
Ty	$(\Pi\sigma.\tau)^{+n} = \Pi\sigma^{+n}.\tau^{+(n+1)}$ $\mathrm{Set}^{+n} = \mathrm{Set}$ $\mathrm{El}(A)^{+n} = \mathrm{El}(A^{+n})$	$(\Pi\sigma.\tau)[N]^n = \Pi\sigma[N]^n.\tau[N]^{n+1}$ $\mathrm{Set}[N]^n = \mathrm{Set}$ $\mathrm{El}(A)[N]^n = \mathrm{El}(A[N]^n)$
Tm	$i^{+n} = \begin{cases} i+1 & \text{if } i \geq n \\ i & \text{otherwise} \end{cases}$ $(\lambda\sigma(M)^\tau)^{+n} = \lambda\sigma^{+n}(M^{+(n+1)})^{\tau^{+(n+1)}}$ $(\forall\sigma.M)^{+n} = \forall\sigma^{+n}.M^{+(n+1)}$ $(\mathrm{app}^{\sigma.\tau}(M,M'))^{+n} = \mathrm{app}^{\sigma^{+n}.\tau^{+(n+1)}}(M^{+n},M'^{+n})$	$i[N]^n = \begin{cases} i & \text{if } i < n \\ N^{+n} & \text{if } i = n \\ i-1 & \text{otherwise} \end{cases}$ $(\lambda\sigma(M)^\tau)[N]^n = \lambda\sigma[N]^n(M[N]^{n+1})^{\tau[N]^{n+1}}$ $(\forall\sigma.M[N]^n = \forall\sigma[N]^n.M[N]^{n+1}$ $(\mathrm{app}^{\sigma.\tau}(M,M'))[N]^n = \mathrm{app}^{\sigma[N]^n.\tau[N]^{n+1}}(M[N]^n,M'[N]^n)$

Figure 2–1: Weakening and substitution

We introduce the following judgements:

$\vdash \Gamma$	context validity,
$\Gamma \vdash \sigma$	type validity,
$\Gamma \vdash \sigma \simeq \tau$	type equality,
$\Gamma \vdash M : \sigma$	typing,
$\Gamma \vdash M \simeq N : \sigma$	equality.

We can also introduce context equality $\vdash \Gamma \simeq \Delta$ but it is not needed for the presentation of the system and therefore we consider it as a derived notion.

2.1.2. DEFINITION (Rules for Calculus of Constructions). The derivable judgement are the least relations introduced by the following rules: [3]

Context validity

$$\vdash \bullet \qquad \text{(EMPTY)}$$

$$\frac{\vdash \Gamma \qquad \Gamma \vdash \sigma}{\vdash \Gamma.\sigma} \qquad \text{(COMPR)}$$

Type validity

$$\frac{\Gamma.\sigma \vdash \tau}{\Gamma \vdash \Pi\sigma.\tau} \qquad \text{(PI)}$$

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathrm{Set}} \qquad \text{(SET)}$$

$$\frac{\Gamma \vdash A : \mathrm{Set}}{\Gamma \vdash \mathrm{El}(A)} \qquad \text{(EL)}$$

Type equality

$$\frac{\Gamma \vdash \sigma}{\Gamma \vdash \sigma \simeq \sigma} \qquad \text{(REFL)}$$

$$\frac{\Gamma \vdash \sigma \simeq \tau}{\Gamma \vdash \tau \simeq \sigma} \qquad \text{(SYM)}$$

$$\frac{\Gamma \vdash \sigma \simeq \tau \qquad \Gamma \vdash \tau \simeq \rho}{\Gamma \vdash \sigma \simeq \rho} \qquad \text{(TRANS)}$$

---

[3] We consider the rule notation just as a shorthand for implication. The fact that least relations exist follows from the fact that the rules correspond to Hornformulas.

$$\frac{\Gamma \vdash \sigma \simeq \sigma' \qquad \Gamma.\sigma \vdash \tau \simeq \tau'}{\Gamma \vdash \Pi\sigma.\tau \simeq \Pi\sigma'.\tau'} \quad (\textsc{Pi-Eq})$$

$$\frac{\Gamma \vdash A \simeq B : \mathrm{Set}}{\Gamma \vdash \mathrm{El}(A) \simeq \mathrm{El}(B)} \quad (\textsc{El-Eq})$$

$$\frac{\Gamma.\sigma \vdash A : \mathrm{Set}}{\Gamma \vdash \mathrm{El}(\forall\sigma.A) \simeq \Pi\sigma.\mathrm{El}(A)} \quad (\textsc{All-Elim})$$

Typing

$$\frac{\Gamma \vdash M : \sigma \qquad \Gamma \vdash \sigma \simeq \tau}{\Gamma \vdash M : \tau} \quad (\textsc{conv})$$

$$\frac{\vdash \Gamma \qquad i \leq |\Gamma|}{\Gamma \vdash i : \Gamma(i)} \quad (\textsc{var})$$

$$\frac{\Gamma.\sigma \vdash M : \tau}{\Gamma \vdash \lambda\sigma(M)^{\tau} : \Pi\sigma.\tau} \quad (\textsc{lam})$$

$$\frac{\Gamma \vdash M : \Pi\sigma.\tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash \mathrm{app}^{\sigma.\tau}(M, N) : \tau[N]} \quad (\textsc{app})$$

$$\frac{\Gamma.\sigma \vdash A : \mathrm{Set}}{\Gamma \vdash \forall\sigma.A : \mathrm{Set}} \quad (\textsc{all})$$

Equality

$$\frac{\Gamma \vdash M : \sigma}{\Gamma \vdash M \simeq M : \sigma} \quad (\textsc{refl})$$

$$\frac{\Gamma \vdash M \simeq N : \sigma}{\Gamma \vdash N \simeq M : \sigma} \quad (\textsc{sym})$$

$$\frac{\Gamma \vdash M \simeq N : \sigma \qquad \Gamma \vdash N \simeq O : \sigma}{\Gamma \vdash M \simeq O : \sigma} \quad (\textsc{trans})$$

$$\frac{\Gamma \vdash M \simeq N : \sigma \qquad \Gamma \vdash \sigma \simeq \tau}{\Gamma \vdash M \simeq N : \tau} \quad (\textsc{conv-eq})$$

$$\frac{\Gamma \vdash \sigma \simeq \sigma' \qquad \Gamma.\sigma \vdash \tau \simeq \tau' \qquad \Gamma.\sigma \vdash M \simeq M' : \tau}{\Gamma \vdash \lambda\sigma(M)^{\tau} \simeq \lambda\sigma'(M')^{\tau'} : \Pi\sigma.\tau} \quad (\textsc{lam-eq})$$

$$\frac{\Gamma \vdash \sigma \simeq \sigma' \qquad \Gamma.\sigma \vdash \tau \simeq \tau' \qquad \Gamma \vdash M \simeq M' : \Pi\sigma.\tau \qquad \Gamma \vdash N \simeq N' : \sigma}{\Gamma \vdash \mathrm{app}^{\sigma.\tau}(M, N) \simeq \mathrm{app}^{\sigma'.\tau'}(M', N') : \tau[N]} \quad (\textsc{app-eq})$$

$$\frac{\Gamma \vdash \sigma \simeq \sigma' \qquad \Gamma.\sigma \vdash A \simeq B : \mathrm{Set}}{\Gamma \vdash \forall\sigma.A \simeq \forall\sigma'.B : \mathrm{Set}} \quad (\textsc{all-eq})$$

$$\frac{\Gamma.\sigma \vdash M : \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash \mathrm{app}^{\sigma.\tau}(\lambda\sigma(M)^\tau, N) \simeq M[N] : \tau[N]} \quad (\textsc{beta-eq})$$

We call this system CC. [Str91] considers a system with an η-rule which we present as follows:

$$\frac{\Gamma \vdash M : \Pi\sigma.\tau}{\Gamma \vdash \lambda\sigma(\mathrm{app}^{\sigma^+.\tau^{+1}}(M^+, 0))^\tau \simeq M : \Pi\sigma.\tau} \quad (\textsc{eta-eq})$$

We call the extended system CC^η. In the following all properties will hold for CC and CC^η unless explicitly mentioned. We will discuss the problems with the η-rule later (see remark 2.3.19).

2.2. Basic Properties

In this section we collect a number of trivial but important observations about our presentation. The first important property of the system is that the rules are consistent with substitution and weakening.

This should be compared with the presentations in [Tro87] and [Str91]. One important difference is that we do not have weakening (or thinning) as a structural rules but as derived rules. This is motivated by the use of de-Bruijn-indices. It seems also natural if one wants to implement the syntax in Type Theory (e.g. compare with the presentation of System F in chapter 5).

2.2.1. Proposition (Weakening).

1.

$$\frac{\vdash \Gamma.\Delta \qquad \Gamma \vdash \tau}{\vdash \Gamma.\tau.\Delta^+}$$

2.

$$\frac{\Gamma.\Delta \vdash \sigma \qquad \Gamma \vdash \tau}{\Gamma.\tau.\Delta^+ \vdash \sigma^{+|\Delta|} \qquad \Gamma \vdash \tau} \qquad \frac{\Gamma.\Delta \vdash \sigma \simeq \sigma' \qquad \Gamma \vdash \tau}{\Gamma.\tau.\Delta^+ \vdash \sigma^{+|\Delta|} \simeq \sigma'^{+|\Delta|}}$$

3.

$$\frac{\Gamma.\Delta \vdash M : \sigma \qquad \Gamma \vdash \tau}{\Gamma.\tau.\Delta^{+} \vdash M^{+|\Delta|} : \sigma^{+|\Delta|}} \qquad \frac{\Gamma.\Delta \vdash M \simeq N : \sigma \qquad \Gamma \vdash \tau}{\Gamma.\tau.\Delta^{+} \vdash M^{+|\Delta|} \simeq N^{+|\Delta|} : \sigma^{+|\Delta|}}$$

Proof. (sketch) By induction over the structure of derivations.

2.2.2. PROPOSITION (Substitution).

1.

$$\frac{\vdash \Gamma.\tau.\Delta \qquad \Gamma \vdash N : \tau}{\vdash \Gamma.\Delta[N]}$$

2.

$$\frac{\Gamma.\tau.\Delta \vdash \sigma \qquad \Gamma \vdash N : \tau}{\Gamma.\Delta[N] \vdash \sigma[N]^{|\Delta|}} \qquad \frac{\Gamma.\tau.\Delta \vdash \sigma \simeq \sigma' \qquad \Gamma \vdash N \simeq N' : \tau}{\Gamma.\Delta[N] \vdash \sigma[N]^{|\Delta|} \simeq \sigma'[N']^{|\Delta|}}$$

3.

$$\frac{\Gamma.\tau.\Delta \vdash M : \sigma \qquad \Gamma \vdash N : \tau}{\Gamma.\Delta[N] \vdash M[N]^{|\Delta|} : \sigma[N]^{|\Delta|}} \qquad \frac{\Gamma.\tau.\Delta \vdash M \simeq M' : \sigma \qquad \Gamma \vdash N \simeq N' : \tau}{\Gamma.\Delta[N] \vdash M[N]^{|\Delta|} \simeq M'[N']^{|\Delta|} : \sigma[N]^{|\Delta|}}$$

Proof. (sketch) By induction over the structure of derivations, using lemma 2.2.1.

We also have the following relations between judgements:

2.2.3. LEMMA.

1. $\dfrac{\Gamma \vdash \sigma}{\vdash \Gamma}$

2. $\dfrac{\Gamma \vdash M : \sigma}{\Gamma \vdash \sigma}$

3. $\dfrac{\Gamma \vdash \sigma \simeq \tau}{\Gamma \vdash \sigma, \tau}$

4. $\dfrac{\Gamma \vdash M \simeq N : \sigma}{\Gamma \vdash M, N : \sigma}$

Proof. (sketch) By induction over the derivation. Note that for 2. (APP) we need (SUBST).

The following lemma states that the type- and context-formation rules are deterministic (i.e. invertible) and the term formation rules are invertible up-to type equality, i.e. the only non-determinism is caused by (CONV).

2.2.4. LEMMA.

1. $$\frac{\vdash \Gamma.\sigma}{\vdash \Gamma \qquad \Gamma \vdash \sigma}$$

2. $$\frac{\Gamma \vdash \Pi\sigma.\tau}{\Gamma.\sigma \vdash \tau}$$

3. $$\frac{\Gamma \vdash \mathrm{Set}}{\vdash \Gamma}$$

4. $$\frac{\Gamma \vdash \mathrm{El}(A)}{\Gamma \vdash A : \mathrm{Set}}$$

5. $$\frac{\Gamma \vdash i : \rho}{\Gamma \vdash \Gamma(i) \simeq \rho}$$

6. $$\frac{\Gamma \vdash \lambda\sigma(M)^{\tau} : \rho}{\Gamma.\sigma \vdash M : \tau \qquad \Gamma \vdash \Pi\sigma.\tau \simeq \rho}$$

7. $$\frac{\Gamma \vdash \mathrm{app}^{\sigma.\tau}(M, N) : \rho}{\Gamma \vdash M : \Pi\sigma.\tau \qquad \Gamma \vdash N : \sigma \qquad \Gamma \vdash \tau[N] \simeq \rho}$$

8. $$\frac{\Gamma \vdash \forall\sigma.A : \rho}{\Gamma.\sigma \vdash A : \mathrm{Set} \qquad \Gamma \vdash \mathrm{Set} \simeq \rho}$$

Proof. (sketch) All rules for $\vdash \Gamma, \Gamma \vdash \sigma, \Gamma \vdash M : \sigma$ are syntax-directed apart from (CONV).

2.3. Reduction and decidability

We will use a strong normalization result for Curry terms to show that conversion for typed terms is decidable. This is the base of a type checking algorithm. We also obtain *uniqueness of product formation* as a corollary, which we will also use to show that we can omit most of the type annotations.

2.3.1. General reduction

We will review a few general results about reduction here. [4] For this purpose assume any (countable) set T of terms and a relation $\rhd_X \in T \times T$ with the property that that we can enumerate the one-step reductions for every M.

Usually T will be defined inductively. When defining a reduction relation $\rhd_X$ we will omit the obvious structural rules and only give the reduction on redexes.

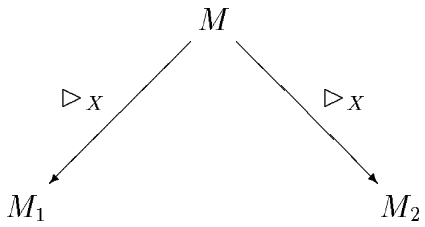
NOTATION. By $\rhd_X$ we refer to the one-step relation, i.e. precisely one redex is reduced. $\rhd_X^+$ is the transitive closure, $\rhd_X^*$ the transitive, reflexive closure and $\approx_X$ the transitive, reflexive, symmetric closure of $\rhd_X$.

2.3.1. DEFINITION. A term is *strongly normalizing* if all its reduction sequences wrt. $\rhd_X$ are finite, or more formally the set of strongly normalizing terms $\mathrm{SN}_X \subseteq T$ is the least set closed under the following rule

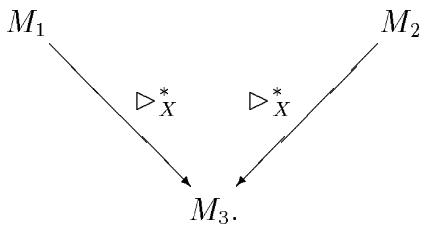
$$\frac{\forall_{N \in T} M \rhd_X N \to N \in \mathrm{SN}_X}{M \in \mathrm{SN}_X}$$

We say $\rhd_X$ is strongly normalizing if all terms are.

We say that $\rhd$ is *weakly Church-Rosser* if



implies that there exists a M_3 s.t.



$\rhd_X$ is Church-Rosser if we can replace $\rhd_X$ in the premise by $\rhd_X^*$.

$\rhd_X$ is *complete* if it is Church-Rosser and strongly normalizing.

---

[4] For a more extensive discussion see e.g. [Hue80]

Note that the above definition of SN_X directly justifies the *principle of Noetherian induction* by using the minimality in the definition of SN_X.

2.3.2. PROPOSITION.

1. *If $\rhd_X$ is weakly Church-Rosser and strongly normalizing then it is also Church-Rosser and therefore complete.*

2. *For any complete reduction $\approx_X$ is decidable by calculating the normal forms and comparing them.*

Proof.

1. By noetherian induction, compare to [Hue80], lemma 2.4.

2. Easy.

However, we are not able to use this proposition directly because our reduction is not globally strongly normalizing. Therefore we introduce a new congruence-relation by restricting reduction:

2.3.3. DEFINITION. $\rhd_X^{\mathrm{SN}} \in \mathrm{SN}_X \times \mathrm{SN}_X$ is the restriction of $\rhd_X$ to strongly normalizing terms. $\approx_X^{\mathrm{SN}} \in \mathrm{SN}_X \times \mathrm{SN}_X$ is the transitive symmetric closure of $\rhd_X^{\mathrm{SN}}$

2.3.4. COROLLARY. *If $\rhd_X$ is weakly Church-Rosser then*

1. *$\rhd_X^{\mathrm{SN}}$ is complete and*

2. *$\approx_X^{\mathrm{SN}}$ is decidable.*

Proof. Obvious by proposition 2.3.2.

Note that $\approx_X$ and $\approx_X^{\mathrm{SN}}$ do not necessarily coincide on strongly normalizing terms, i.e. there may be $M, N \in \mathrm{SN}_X$ s.t. $M \approx_X N$ but $M \not\approx_X^{\mathrm{SN}} N$.

2.3.2. Curry reduction

We call terms with no type information *Curry terms* (Λ with $M, N \in \Lambda$), they are defined as follows:

2.3.5. Definition.

$$\Lambda ::= i \mid MN \mid \lambda M \mid \forall M.$$

Weakening M^{+i} and substitution $M[N]^i$ is defined as for typed terms (in particular $\forall$ is a binder).

We use Curry terms and reduction on them to represent the *computational content* of a typed term.

There is an obvious forgetful function:

2.3.6. Definition (Stripping). We define a function $|\_| \in \mathrm{Tm} \to \Lambda$ by:

$$\begin{aligned} |i| &= i \\ |\lambda\sigma(M)^\tau| &= \lambda|M| \\ |\forall\sigma.M| &= \forall|M| \\ |\mathrm{app}^{\sigma,\tau}(M,N)| &= |M||N| \end{aligned}$$

This can be extended to types and contexts in an obvious way.

It should be obvious that stripping preserves weakening $|M^{+i}| = |M|^{+i}$ and substitution $|M[N]^i| = |M|[|N|]^i$.

2.3.7. Definition. $\rhd$ is the usual β-reduction:

$$(\lambda M)N \rhd M[N]$$

In the case of CC^η we also add:

$$\lambda(M^+0) \rhd_\eta M$$

We note that weakening and substitution preserves reduction in the following way:

2.3.8. LEMMA.

1. $\dfrac{M \rhd N}{M^{+i} \rhd N^{+i}}$

2. $\dfrac{M \rhd M'}{M[N]^i \rhd M'[N]^i}$

3. $\dfrac{N \rhd N'}{M[N]^i \rhd^* M[N']^i}$

Proof. By induction over the derivation of $\rhd$.

The following proposition is well known (the extension by $\forall$ is irrelevant):

2.3.9. PROPOSITION. $\rhd$,$\rhd_\eta$ *are Church-Rosser.*

Proof. See [Bar84], pp. 277.

We will exploit the following proposition which will be shown by a semantic construction in the next chapter:

2.3.10. PROPOSITION (Strong normalization). *If* $\Gamma \vdash M : \sigma$ *then* $|M| \in \mathrm{SN}$.

2.3.3. Tight reduction

In our definition of judgemental equality the equality of terms depends on the equality of their annotations. Therefore we introduce a notion of reduction on typed terms and types which we can use to decide the equality judgement. Note that we only allow reductions when the types coincide, here we diverge from [Str91], pp. 169. [5] We call this reduction *tight reduction* and refer to Streicher's

[5] Streicher does not investigate reduction in great detail. He assumes a very strong property (ibid.,p.169): *uniqueness of normal forms.* It seems rather unlikely that one can prove this property directly because it is non-trivial to show subject reduction for loose reduction. In particular this seems to require *uniqueness of product formation* (ibid, p.243) which he derives as a consequence of uniqueness of normal forms.

definition as *loose* reduction. This restriction is essential to show the *subject reduction property.*

2.3.11. DEFINITION (Tight reduction). We define $\rhd_{\mathrm{t}}, \rhd_{\mathrm{t}\eta} \subseteq \mathrm{Cn} \times \mathrm{Cn}$ as the reduction relations generated by the following rules:

$$\mathrm{El}(\forall\sigma.A) \rhd_{\mathrm{t}} \Pi\sigma.\mathrm{El}(A) \qquad (\textsc{all-red})$$

$$\mathrm{app}^{\sigma.\tau}(\lambda\sigma(M)^{\tau}, N) \rhd_{\mathrm{t}} M[N] \qquad (\textsc{beta-red})$$

$$\lambda\sigma(\mathrm{app}^{\sigma^{+}.\tau^{+1}}(M^{+}, 0))^{\tau} \rhd_{\mathrm{t}\eta} M \qquad (\textsc{eta-red})$$

Analogue to lemma 2.3.8 we have:

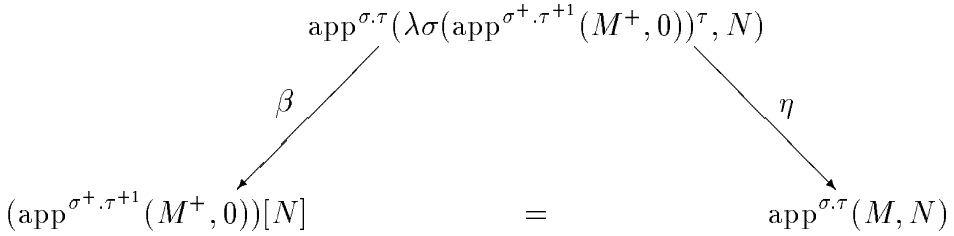
2.3.12. LEMMA.

1. $\dfrac{C \rhd_{\mathrm{t}} N}{C^{+i} \rhd_{\mathrm{t}} N^{+i}}$
2. $\dfrac{C \rhd_{\mathrm{t}} C'}{C[N]^{i} \rhd_{\mathrm{t}} C'[N]^{i}}$
3. $\dfrac{N \rhd_{\mathrm{t}} N'}{C[N]^{i} \rhd_{\mathrm{t}}^{*} C[N']^{i}}$

Proof. By induction over the derivation of $\rhd_{\mathrm{t}}$.

2.3.13. LEMMA. $\rhd_{\mathrm{t}}$ *is weakly Church-Rosser.*

Proof. By exhaustive case analysis. Let us just consider the case of an overlapping β and η-reduction here (which is a critical pair for the loose reduction):



All the other cases are completely straightforward using lemma 2.3.12.

Note that it is not obvious that $\rhd_t$ is Church-Rosser, indeed it is more likely that this is not the case because the rules are not left-linear. [6]

We will now show that the strong normalization property for Curry terms implies strong normalization for tight reduction. We do this by systematically blowing up terms such that every tight reduction can be mirrored by a reduction on the underlying Curry term.

We define first some auxiliary notions:

$$\begin{aligned} \bot &= \forall X : \mathrm{Set}.X \\ M(\sigma, N) &= \mathrm{app}^{\mathrm{Set}.\sigma^+}(\lambda\mathrm{Set}(M^+)^{\sigma^+}, N) \end{aligned}$$

2.3.14. Lemma.

1. $$\frac{\vdash \Gamma}{\Gamma \vdash \bot : \mathrm{Set}}$$
2. $$\frac{\Gamma \vdash M : \sigma \qquad \Gamma \vdash N : \mathrm{Set}}{\Gamma \vdash M(\sigma, N) \simeq M : \sigma}$$
3. $|M(\sigma, N)| \rhd |M|$

Proof. Easy.

2.3.15. Definition. We now define $\mathrm{blow} \in \mathrm{Cn} \to \mathrm{Cn}$:

$$\begin{aligned} \mathrm{blow}(\Pi\sigma.\tau) &= \mathrm{blow}(\sigma)(\mathrm{Set}, \mathrm{blow}(\tau)) \\ \mathrm{blow}(\mathrm{Set}) &= \bot \\ \mathrm{blow}(\mathrm{El}(A)) &= \mathrm{blow}(A) \\ \mathrm{blow}(i) &= i \\ \mathrm{blow}(\mathrm{app}^{\sigma.\tau}(M, N)) &= \mathrm{app}^{\sigma.\tau}(\mathrm{blow}(M), \mathrm{blow}(N))(\tau[N], \mathrm{blow}(\sigma))(\tau[N], \mathrm{blow}(\tau)) \\ \mathrm{blow}(\lambda\sigma(M)^\tau) &= \lambda\sigma(\mathrm{blow}(M))^\tau(\Pi\sigma.\tau, \mathrm{blow}(\sigma))(\Pi\sigma.\tau, \mathrm{blow}(\tau)) \\ \mathrm{blow}(\forall\sigma.A) &= \forall\sigma.\mathrm{blow}(A)(\mathrm{Set}, \mathrm{blow}(\sigma)) \end{aligned}$$

---

[6] Compare with Klop's counterexample, see [Bar84], pp. 403.

The idea behind the definition of blow is summarized in the following lemma:

2.3.16. LEMMA.

1. $$\frac{\Gamma \vdash \sigma}{\Gamma \vdash \mathrm{blow}(\sigma) : \mathrm{Set}}$$

2. $$\frac{\Gamma \vdash M : \sigma}{\Gamma \vdash \mathrm{blow}(M) : \sigma}$$

3. *If* $C \rhd_{\mathrm{t}} D$ *then* $|\mathrm{blow}(C)| \rhd^{+} |\mathrm{blow}(D)|$.

Proof.

1.,2. Just apply fact 2.3.14.

3. By induction over $\rhd_{\mathrm{t}}$. ...

2.3.17. COROLLARY.

1. $$\frac{\Gamma \vdash \sigma}{\sigma \in \mathrm{SN}_t}$$

2. $$\frac{\Gamma \vdash M : \sigma}{M \in \mathrm{SN}_t}$$

Proof. Just apply lemma 2.3.16.

2.3.4. Decidability

To derive decidability we have to show subject reduction and, alas, it is not clear how to show this property for CC^{η} therefore we will do this only for CC.

2.3.18. THEOREM (Subject reduction).

$$\frac{\Gamma \vdash \sigma \qquad \sigma \rhd_{\mathrm{t}} \tau}{\Gamma \vdash \tau \qquad \Gamma \vdash \sigma \simeq \tau} \qquad \frac{\Gamma \vdash M : \sigma \qquad M \rhd_{\mathrm{t}} N}{\Gamma \vdash N : \sigma \qquad \Gamma \vdash M \simeq N : \sigma}$$

Proof. Note that the first part of the conclusions can be inferred from the second by lemma 2.2.3(3,4).

By induction over $\rhd_{\mathrm{t}}$ exploiting lemma 2.2.4. Let's only consider the case of a β-reduction here:

$$\mathrm{app}^{\sigma.\tau}(\lambda\sigma(M)^{\tau}, N) \rhd_{\mathrm{t}} M[N]$$

From

$$\Gamma \vdash \mathrm{app}^{\sigma.\tau}(\lambda\sigma(M)^{\tau}, N) : \rho$$

we can infer (by lemma 2.2.4(6,8)):

$$\Gamma \vdash \rho \simeq \tau[N]$$
$$\Gamma.\sigma \vdash M : \tau$$
$$\Gamma \vdash N : \sigma$$

Now we only have to apply (BETA-EQ) and (CONV) to derive:

$$\Gamma \vdash \mathrm{app}^{\sigma.\tau}(\lambda\sigma(M)^{\tau}, N) \simeq M[N] : \rho$$

2.3.19. REMARK. What is the problem with η-reduction here? Consider:

$$\lambda\sigma(\mathrm{app}^{\sigma^+.\tau^{+1}}(M^+, 0))^{\tau} \rhd_{\mathrm{t}} M$$

Assume we know

$$\Gamma \vdash \lambda\sigma(\mathrm{app}^{\sigma^+.\tau^{+1}}(M^+, 0))^{\tau} : \rho$$

using lemma 2.2.4 we can conclude:

$$\Gamma.\sigma \vdash M^+ : \Pi\sigma^+.\tau^{+1}$$

but to apply (ETA-EQ) we need

$$\Gamma \vdash M : \Pi\sigma.\tau.$$

We certainly believe that the inverse of weakening, i.e. strengthening should be admissible, i.e.

$$\frac{\Gamma.\sigma \vdash M^+ : \tau^+}{\Gamma \vdash M : \tau}$$

However, it is not easy to verify this rule. The essential problem is that it is not true semantically, i.e. it fails in models with empty types. Therefore we cannot hope that we can derive this rule using only simple properties of substitution and weakening because they hold in all models.

However, the problem disappears when we turn around η-reduction. η-expansion seems the semantically appropriate notion because subject reduction becomes easily derivable, i.e. indeed it is true in every sound interpretation. But η-expansion has *syntactic* disadvantages: it is no longer strongly normalizing. It is also not clear how to reconstruct the type annotations.

It is now easy to derive the core lemma:

2.3.20. LEMMA.

$$\frac{\Gamma \vdash \sigma, \tau \qquad \sigma \approx_{\mathrm{t}}^{\mathrm{SN}} \tau}{\Gamma \vdash \sigma \simeq \tau} \qquad \frac{\Gamma \vdash M, N : \sigma \qquad M \approx_{\mathrm{t}}^{\mathrm{SN}} N}{\Gamma \vdash M \simeq N : \sigma}$$

Proof. We restrict ourselves to the second rule:

if We exploit that $\rhd_{\mathrm{t}}^{\mathrm{SN}}$ is Church-Rosser (corollary 2.3.4 and lemma 2.3.13) to derive that $M \rhd_{\mathrm{t}}^{\mathrm{SN}} L \lhd_{\mathrm{t}}^{\mathrm{SN}} N$. Now we only have to use theorem 2.3.18 to see that

$$\Gamma \vdash M \simeq L \simeq N : \sigma$$

and the result follows by (TRANS).

only if Obviously every conversion rule is mirrored by $\approx_{\mathrm{t}}$ and to see that it is indeed $\approx_{\mathrm{t}}^{\mathrm{SN}}$ we only have to use lemma 2.3.17.

2.3.21. COROLLARY (Decidability of equality).

1. *If* $\Gamma \vdash \sigma, \tau$ *it is decidable whether* $\Gamma \vdash \sigma \simeq \tau$.

2. *If* $\Gamma \vdash M, N : \sigma$ *it is decidable whether* $\Gamma \vdash M \simeq N : \sigma$

Proof. Follows directly from lemma 2.3.20 and corollary 2.3.4.

2.4. Type reconstruction

We have presented the calculus using a very explicit notation, this has been exploited for the definition of tight reduction and it will simplify the definition of an interpretation. Following Streicher [7] we can now show that we can *throw away the ladder on which we climbed* and omit types in applications and indeed also the codomain of applications. We will also describe a type reconstruction algorithm which is based on the decidability of equality (corollary 2.3.21).

Much of the syntactic development is based on the following property: [8]

2.4.1. THEOREM (Uniqueness of product formation).

$$\frac{\Gamma \vdash \Pi\sigma_1.\sigma_2 \simeq \Pi\tau_1.\tau_2}{\Gamma \vdash \sigma_1 \simeq \sigma_2 \qquad \Gamma.\sigma_1 \vdash \sigma_2 \simeq \tau_2}$$

Proof. By lemma 2.3.20 we know:

$$\Pi\sigma_1.\sigma_2 \simeq_t^{\mathrm{SN}} \Pi\tau_1.\tau_2$$

Using completeness of $\rhd_{\mathrm{t}}^{\mathrm{SN}}$ we have that there exists ρ s.t.

$$\Pi\sigma_1.\sigma_2 \rhd_{\mathrm{t}} \rho \lhd_{\mathrm{t}} \Pi\tau_1.\tau_2$$

From the definition of $\rhd_{\mathrm{t}}$ it is obvious that $\rho = \Pi\rho_1.\rho_2$ and

$$\sigma_1 \quad \rhd_{\mathrm{t}} \quad \rho_1 \quad \lhd_{\mathrm{t}} \tau_1$$
$$\sigma_2 \quad \rhd_{\mathrm{t}} \quad \rho_2 \quad \lhd_{\mathrm{t}} \tau_2$$

By applying theorem 2.3.18 we have:

$$\Gamma \vdash \sigma_1 \simeq \rho_1 \simeq \tau_1$$
$$\Gamma.\sigma_1 \vdash \sigma_2 \simeq \rho_2 \simeq \tau_2$$

and the result follows by (TRANS).

---

[7] [Str91], pp. 242

[8] Compare Lemma 4.8. [Str91],pp. 243

We will now define *Church terms* $\mathrm{Tm}^{\mathrm{Church}}$, from which all type information apart from the domain of a λ-abstraction and $\forall$ is removed, and a stripping operation.

2.4.2. DEFINITION (Church syntax).

$$\mathrm{Tm}^{\mathrm{Church}} ::= i \mid \lambda\sigma.M \mid M\ N \mid \forall\sigma.M$$

$\mathrm{Ty}^{\mathrm{Church}}$ and $\mathrm{Co}^{\mathrm{Church}}$ are defined as before, replacing Tm by $\mathrm{Tm}^{\mathrm{Church}}$.

We define $\|\_\| \in \mathrm{Tm} \to \mathrm{Tm}^{\mathrm{Church}}$ by primitive recursion:

$$\begin{aligned} \|i\| &= i \\ \|\lambda\sigma(M)^\tau\| &= \lambda\|\sigma\|.\|M\| \\ \|\forall\sigma.M\| &= \forall\|\sigma\|.\|M\| \\ \|\mathrm{app}^{\sigma,\tau}(M,N)\| &= \|M\|\,\|N\| \end{aligned}$$

This has to be extended to types and contexts in an obvious way.

We will show that Church terms determine an explicitly typed term and its type up to judgmental equality. [9]

2.4.3. LEMMA.

$$\frac{\Gamma \vdash M : \sigma \qquad \Gamma \vdash N : \tau \qquad \|M\| = \|N\|}{\Gamma \vdash \sigma \simeq \tau \qquad \Gamma \vdash M \simeq N : \sigma} \qquad \frac{\Gamma \vdash \sigma,\tau \qquad \|\sigma\| = \|\tau\|}{\Gamma \vdash \sigma \simeq \tau}$$

Proof. By induction over the structure of $\|M\|$ and $\|\sigma\|$. The cases for types are all straightforward (using induction hypotheses and lemma 2.2.4) and we will only discuss abstraction and application in detail here:

abstraction

Assume

$$\begin{aligned} M &= \lambda\sigma_1(M')^{\sigma_2} \\ N &= \lambda\tau_1(N')^{\tau_2} \\ \|M\| &= \|N\| \end{aligned}$$

[9] Compare Theorem 4.10. [Str91], pp. 245

Using lemma 2.2.4(6) we have

$$\Gamma.\sigma_1 \vdash M' : \sigma_2$$
$$\Gamma.\tau_1 \vdash N' : \tau_2$$
$$\Gamma \vdash \sigma \simeq \Pi\sigma_1.\sigma_2$$
$$\Gamma \vdash \tau \simeq \Pi\tau_1.\tau_2$$

Using the induction hypothesis we can derive:

$$\Gamma \vdash \sigma_1 \simeq \tau_1$$
$$\Gamma.\sigma_1 \vdash \sigma_2 \simeq \tau_2$$
$$\Gamma.\sigma_1 \vdash M \simeq N : \sigma_2$$

Hence by (PI-EQ) and (LAM-EQ) we have:

$$\Gamma \vdash \sigma \simeq \Pi\sigma_1.\sigma_2 \simeq \Pi\tau_1.\tau_2 \simeq \tau$$
$$\Gamma \vdash M = \lambda\sigma_1(M')^{\sigma_2} \simeq \lambda\tau_1(N')^{\tau_2} = N : \Pi\sigma_1.\sigma_2$$

application

Assume

$$\begin{aligned} M &= \mathrm{app}^{\sigma_1.\sigma_2}(M_1, M_2) \\ N &= \mathrm{app}^{\tau_1.\tau_2}(N_1, N_2) \\ \|M\| &= \|N\| \end{aligned}$$

Using lemma 2.2.4(7) we have

$$\Gamma \vdash M_1 : \Pi\sigma_1.\sigma_2$$
$$\Gamma \vdash N_1 : \Pi\tau_1.\tau_2$$
$$\Gamma \vdash M_2 : \sigma_1$$
$$\Gamma \vdash N_2 : \tau_1$$
$$\Gamma \vdash \sigma \simeq \sigma_2[M_1]$$
$$\Gamma \vdash \tau \simeq \tau_2[N_1]$$

Using the ind.hyp we can derive:

$$\Gamma \vdash \Pi\sigma_1.\sigma_2 \simeq \Pi\tau_1.\tau_2$$
$$\Gamma \vdash M_1 \simeq N_1 : \Pi\sigma_1.\sigma_2$$
$$\Gamma \vdash \sigma_1 \simeq \tau_1$$
$$\Gamma \vdash M_2 \simeq N_2 : \sigma_1$$

Using theorem 2.4.1 we also have:

$$\Gamma.\sigma_1 \vdash \sigma_2 \simeq \tau_2$$

Hence by lemma 2.2.2 and (APP-EQ) we have:

$$\Gamma \vdash \sigma \simeq \sigma_2[M_1] \simeq \tau_2[N_1] \simeq \tau$$
$$\Gamma \vdash M = \mathrm{app}^{\sigma_1,\sigma_2}(M_1, M_2) \simeq \mathrm{app}^{\tau_1,\tau_2}(N_1, N_2) = M : \sigma_2[M_1]$$

From the previous lemma we know that a typed term and its type is already determined by its underlying Church term. This justifies the use of Church terms to denote explicitely typed terms. Indeed, based on the decidability we can construct an algorithm which computes the explicitely typed version of a Church term and its type or signals an error if no such term exists, i.e. we have:

$$\begin{aligned}
\mathrm{TR}^{\mathrm{Co}} &\in \mathrm{Co}^{\mathrm{Church}} \to \mathrm{Co} \uplus \{\bot\} \\
\mathrm{TR}^{\mathrm{Ty}} &\in \mathrm{Co}^{\mathrm{Church}} \times \mathrm{Ty}^{\mathrm{Church}} \to \mathrm{Ty} \uplus \{\bot\} \\
\mathrm{TR}^{\mathrm{Tm}} &\in \mathrm{Co}^{\mathrm{Church}} \times \mathrm{Tm}^{\mathrm{Church}} \to (\mathrm{Tm} \times \mathrm{Ty}) \uplus \{\bot\}
\end{aligned}$$

with the following properties:

$$\mathrm{TR}^{\mathrm{Co}}(\Gamma) = \begin{cases} \Gamma' & \text{such that } \vdash \Gamma' \wedge \|\Gamma'\| = \Gamma \\ \bot & \text{if no solution exists.} \end{cases}$$

$$\mathrm{TR}^{\mathrm{Ty}}(\Gamma, \sigma) = \begin{cases} \sigma' & \text{such that } \Gamma' \vdash \sigma' \wedge \|\Gamma'\| = \Gamma \wedge \|\sigma'\| = \sigma \\ \bot & \text{if no solution exists.} \end{cases}$$

$$\mathrm{TR}^{\mathrm{Tm}}(\Gamma, M) = \begin{cases} (M', \sigma') & \text{such that } \Gamma' \vdash M' : \sigma' \wedge \|\Gamma'\| = \Gamma \wedge \|M'\| = M \wedge \|\sigma'\| = \sigma \\ \bot & \text{if no solution exists.} \end{cases}$$

We present this algorithm which works simply be structural recursion. We assume that $\mathrm{NF} \in \mathrm{Cn} \to \mathrm{Cn}$ computes the normal form of a construction which respect to $\rhd_{\mathrm{t}}$. To save space we adopt the convention that failures ($\bot$) are propagated.

$$
\begin{aligned}
\mathrm{TR}^{\mathrm{Co}}(\bullet) &= \bullet \\
\mathrm{TR}^{\mathrm{Co}}(\Gamma.\sigma) &= \mathrm{TR}^{\mathrm{Co}}(\Gamma).\mathrm{TR}^{\mathrm{Ty}}(\Gamma,\sigma) \\
\mathrm{TR}^{\mathrm{Ty}}(\Gamma,\Pi\sigma.\tau) &= \Pi\mathrm{TR}^{\mathrm{Ty}}(\Gamma,\sigma).\mathrm{TR}^{\mathrm{Ty}}(\Gamma.\sigma,\tau) \\
\mathrm{TR}^{\mathrm{Ty}}(\Gamma,\mathrm{Set}) &= \begin{cases} \mathrm{Set} & \text{if } \mathrm{TR}^{\mathrm{Co}}(\Gamma) \neq \bot \\ \bot & \text{otherwise} \end{cases} \\
\mathrm{TR}^{\mathrm{Ty}}(\Gamma,\mathrm{El}(A)) &= \begin{cases} \mathrm{El}(A') & \text{if } \mathrm{TR}^{\mathrm{Tm}}(\Gamma,A) = (A,\mathrm{Set}) \\ \bot & \text{otherwise} \end{cases} \\
\mathrm{TR}^{\mathrm{Tm}}(\Gamma,i) &= \begin{cases} (i,\mathrm{NF}(\Gamma'(i))) & \text{if } \mathrm{TR}^{\mathrm{Co}}(\Gamma) = \Gamma' \wedge i \leq |\Gamma| \\ \bot & \text{otherwise} \end{cases} \\
\mathrm{TR}^{\mathrm{Tm}}(\Gamma,MN) &= \begin{cases} (\mathrm{app}^{\sigma.\tau}(M',N'),\mathrm{NF}(\tau[N'])) & \text{if } \begin{array}{l} \mathrm{TR}^{\mathrm{Tm}}(\Gamma,M) = (M',\Pi\sigma.\tau) \wedge \\ \mathrm{TR}^{\mathrm{Tm}}(\Gamma,N) = (N',\sigma) \end{array} \\ \bot & \text{otherwise} \end{cases} \\
\mathrm{TR}^{\mathrm{Tm}}(\Gamma,\lambda\sigma.M) &= \begin{cases} (\lambda\sigma'(M')^{\tau},\Pi\sigma'.\tau) & \text{if } \mathrm{TR}^{\mathrm{Ty}}(\Gamma,\sigma) = \sigma' \wedge \mathrm{TR}^{\mathrm{Tm}}(\Gamma.\sigma,M) = (M',\tau) \\ \bot & \text{otherwise} \end{cases} \\
\mathrm{TR}^{\mathrm{Tm}}(\Gamma,\forall\sigma.M) &= \begin{cases} (\forall\sigma'.A',\mathrm{Set}) & \text{if } \mathrm{TR}^{\mathrm{Ty}}(\Gamma,\sigma) = \sigma' \wedge \mathrm{TR}^{\mathrm{Tm}}(\Gamma.\sigma,A) = (A',\mathrm{Set}) \\ \bot & \text{otherwise} \end{cases}
\end{aligned}
$$

We will not verify this algorithm in detail but note that all the types which are computed will always be in normal form and therefore as in corollary 2.3.21 judgemental equality is reflected by syntactic identity. It is also important to note that we will only compute a normal form when we can be sure that the type is valid and therefore strongly normalizing (lemma 2.3.17). Obviously, algorithm presented here is extremely inefficient: we may recompute the types of subterms or verify types or contexts several times and we use the ineffective tight reduction anyway.

2.5. The conversion presentation

We already mentioned the equality-as-conversion presentation of Type Theory. In this section we will exploit theorem 2.4.1 to show that the two presentation of CC coincide. We will encounter yet another problem with the η-rule. To simplify the presentation we will use the explicit syntax here.

We start by defining *loose reduction* which corresponds to the notion of reduction described in [Str91].

2.5.1. Definition (Loose reduction). We define $\rhd_{\mathrm{l}}, \rhd_{\mathrm{l}\eta} \subseteq \mathrm{Cn} \times \mathrm{Cn}$ as the reduction relations generated by the following rules:

$$\mathrm{El}(\forall\sigma.A) \rhd_{\mathrm{l}} \Pi\sigma.\mathrm{El}(A) \qquad \text{(ALL-RED)}$$

$$\mathrm{app}^{\sigma,\tau}(\lambda\sigma'(M)^{\tau'}, N) \rhd_{\mathrm{l}} M[N] \qquad \text{(BETA-RED)}$$

$$\lambda\sigma(\mathrm{app}^{\sigma^+,\tau^{+1}}(M^+, 0))^{\tau} \rhd_{\mathrm{l}\eta} M \qquad \text{(ETA-RED)}$$

The following fact is only true for $\rhd_{\mathrm{l}}$ not for $\rhd_{\mathrm{l}\eta}$:

2.5.2. Proposition. $\rhd_{\mathrm{l}}$ *is Church-Rosser.*

Proof. Straightforward extension of proposition 2.3.9.

Using theorem 2.4.1 we can now show subject reduction for loose reduction. Note that we needed to define tight reduction first because we used it to establish theorem 2.4.1. The following lemma relates loose and tight reduction:

2.5.3. Lemma (Tightening).

$$\frac{\Gamma \vdash \sigma \qquad \sigma \rhd_{\mathrm{l}} \tau}{\exists\sigma'.\Gamma \vdash \sigma \simeq \sigma' \qquad \sigma' \rhd_{\mathrm{t}} \tau} \qquad \frac{\Gamma \vdash M : \sigma \qquad M \rhd_{\mathrm{l}} N}{\exists M'.\Gamma \vdash M \simeq M' : \sigma \qquad M' \rhd_{\mathrm{t}} N}$$

Proof. By induction over the definition of $\rhd_{\mathrm{l}}$. The interesting cases are only β and η. We only consider β here: Assume

$$\Gamma \vdash M = \mathrm{app}^{\sigma,\tau}(\lambda\sigma'(M_1)^{\tau'}, M_2) : \rho$$

We have that $M \rhd_{\mathrm{l}} M_1[M_2]$. By using lemma 2.2.4(6,7) it can be easily established that:

$$\Gamma \vdash \Pi\sigma.\tau \simeq \Pi\sigma'.\tau'$$

and with theorem 2.4.1 we have that:

$$\Gamma \vdash \sigma \simeq \sigma'$$
$$\Gamma.\sigma \vdash \tau \simeq \tau'$$

and hence:

$$\Gamma \vdash M \simeq M' = \mathrm{app}^{\sigma,\tau}(\lambda\sigma(M_1)^{\tau}, M_2) : \rho$$

and obviously

$$M' \rhd_{\mathrm{t}} M_1[M_2].$$

2.5.4. Theorem (Subject reduction for $\rhd_{\mathrm{l}}$).

$$\frac{\Gamma \vdash \sigma \qquad \sigma \rhd_{\mathrm{l}} \tau}{\Gamma \vdash \tau \qquad \Gamma \vdash \sigma \simeq \tau} \qquad \frac{\Gamma \vdash M : \sigma \qquad M \rhd_{\mathrm{l}} N}{\Gamma \vdash N : \sigma \qquad \Gamma \vdash M \simeq N : \sigma}$$

Proof. Follows from theorem 2.3.18 with lemma 2.5.3.

We show the following lemma only for β:

2.5.5. Lemma.

$$\frac{\Gamma \vdash \sigma, \tau \qquad \sigma \approx_{\mathrm{l}} \tau}{\Gamma \vdash \sigma \simeq \tau} \qquad \frac{\Gamma \vdash M, N : \sigma \qquad M \approx_{\mathrm{l}} N}{\Gamma \vdash M \simeq N : \sigma}$$

Proof. Similar to 2.3.20, using proposition 2.5.2.

Note that although the simple minded proof fails for η it seems possible to adapt Geuver's work [Geu93] to our presentation. Another observation is that it is not so hard to show if we restrict ourself to $\approx_{\mathrm{l}}^{\mathrm{SN}}$ which is completely justified even from the view of an implementation.

We now define the conversion presentation of CC.

2.5.6. Definition (Conversion presentation of CC). We define the judgements $\vdash^{\approx} \Gamma$, $\Gamma \vdash^{\approx} \sigma$ and $\Gamma \vdash^{\approx} M : \sigma$ by modifying definition 2.1.2 as follows:

1. The rule (CONV) is replaced by

$$\frac{\Gamma \vdash^{\approx} M : \sigma \qquad \sigma \approx_{\mathrm{l}} \tau}{\Gamma \vdash^{\approx} M : \tau} \qquad \text{(CONV')}$$

2. All the rules regarding equality judgments are omitted.

2.5.7. THEOREM (Equivalence).

$$\frac{\vdash G}{\vdash^{\approx} G} \qquad \frac{\Gamma \vdash \sigma}{\Gamma \vdash^{\approx} \sigma} \qquad \frac{\Gamma \vdash M : \sigma}{\Gamma \vdash^{\approx} M : \sigma}$$

Proof. Follows directly from lemma 2.5.5.

It seems straightforward to combine this result with the one presented in the previous section to verify the equivalence to a presentation using conversion of Church-terms. This is already quite close to a PTS presentation. However, there are still some differences, in particular the presentation of the λ-rule, but the PTS presentation of the λ-rule seems questionable anyway. [10]

[10] It seems that the problem with the expansion-postponement property in [Pol92] are mainly caused by the way the λ-rule is presented for PTSes.

Chapter 3

Semantics and strong normalization for the core calculus

We will now analyze the system semantically and establish consistency and strong normalization. For this purpose we will introduce the notion of a CC-structure which is influenced by Henkin-models for simple types and by the categorical semantics of CC. We will show that every CC-structure gives rise to a sound interpretation and then rephrase well known model constructions in terms of CC-structures: the proof-irrelevance interpretation and the realizability semantics (D-sets). Finally we will present the saturated Λ-set model as a modification of the realizability semantics and derive strong normalization as a corollary.

3.1. Basic notations

We will review some set-theoretic notations and define what we understand by a *sound interpretation*.

NOTATION. If we write $a \in A$, $a = b$ we mean that A, a, b are defined and in the appropriate relation. We use $a = b \in A$ to say $a = b$ and $a, b \in A$.

We use $a \cong b$ to denote Kleene equality, i.e. either both a and b are defined and equal or they are both undefined.

Analogously by $a \underset{\sim}{\in} A$ we mean that if a is defined then A is defined and $a \in A$.

We also require some set-theoretic notation:

NOTATION. [Set-theoretic notations] Assume some encoding of pairs (a, b) together with projections π_1, π_2, s.t. $\pi_1(a, b) = a$, $\pi_2(a, b) = b$.

Application is a partial operation: $f(x)$ is defined if $\exists!_{(x,y)\in f}$ and then equal to y.

We will use nested pairs to represent sequences. ϵ is the empty sequence and if δ is a sequence and x an element then (δ, x) is a new sequence. If all elements of a sequence are in a set X then the sequence is element of X^*. We denote concatenation for nested pairs by juxtaposition, which can be defined recursively $\gamma\epsilon = \gamma$ and $\gamma(\delta, x) = (\gamma\delta, x)$. Often we will use the vector notation $\vec{a}$ to denote variables ranging over sequences. We denote the length of a sequence by $|\vec{a}|$.

3.1.1. DEFINITION (Operations on sets). Let A be a set and $\{B_a, B'_a\}_{a\in A}$ families of sets indexed by A. We define the following operations on sets:

$$\begin{aligned} \Sigma a \in A.B_a &= \{(a, b) \mid a \in A, b \in B_a\} \\ \Pi a \in A.B_a &= \{f \subseteq \Sigma a \in A.B_a \mid \forall_{a\in A}\exists!_{b\in B_a}(a, b) \in f\} \end{aligned}$$

As usual we write $A \to B$ for $\Pi a \in A.B$.

In all our models we use classes of sets with some additional (intensional) structure:

3.1.2. DEFINITION (Universe). A universe $\mathfrak{U}$ is a class together with an operation which assigns to every $X \in \mathfrak{U}$ a set $\overline{X}$ which we call *the extension* of X.

3.1.3. REMARK. The idea behind our definition of universes is that we want to consider classes of sets with some additional (intensional) structure which is not reflected in the enumeration of the elements. A typical example of this are the D-sets definition 3.4, where to every set we associate a realizability relation.

3.1.4. DEFINITION (Interpretation). An interpretation is given by a two universes $\mathfrak{U}_{\mathrm{Co}}$ and $\mathfrak{U}_{\mathrm{Ty}}$ and:

1. A partial assignment of elements of $\mathfrak{U}_{\mathrm{Co}}$ to contexts:

$$[\![\vdash \Gamma]\!] \in \mathfrak{U}_{\mathrm{Co}}$$

2. A partial assignment of families in $\mathcal{U}_{\mathrm{Ty}}$ to type validity judgements:

$$\{[\![\Gamma \vdash \sigma]\!]\gamma \in \mathfrak{U}_{\mathrm{Ty}}\}_{\gamma \in \overline{[\![\vdash \Gamma]\!]}}$$

3. A partial assignment of families of values to terms in a context:

$$\{[\![\Gamma \vdash M]\!]\gamma\}_{\gamma \in \overline{[\![\vdash \Gamma]\!]}}$$

Note that we assign meaning just to contexts, pairs of contexts and types and pairs of contexts and terms. That we use the notation for judgements (i.e. $\vdash$) is just to improve readability.

3.1.5. Definition (Soundness of an interpretation). We say an interpretation is sound, if the following holds:

1. If $\vdash \Gamma$ then $[\![\vdash \Gamma]\!]$ is defined.

2. If $\Gamma \vdash \sigma$ then $[\![\Gamma \vdash \sigma]\!]$ is defined.

3. If $\Gamma \vdash M : \sigma$ then

$$[\![\Gamma \vdash M]\!] \in \Pi\gamma \in \overline{[\![\vdash \Gamma]\!]}.[\![\Gamma \vdash \sigma]\!]\gamma.$$

4. If $\Gamma \vdash \sigma \simeq \tau$ then

$$[\![\Gamma \vdash \sigma]\!] = [\![\Gamma \vdash \tau]\!].$$

5. If $\Gamma \vdash M \simeq N : \sigma$ then

$$[\![\Gamma \vdash M]\!] = [\![\Gamma \vdash N]\!] \in \Pi\gamma \in \overline{[\![\vdash \Gamma]\!]}.[\![\Gamma \vdash \sigma]\!]\gamma.$$

Note that soundness of interpretation does not imply that we have a model (i.e. a CC-structure), because we only say something about the behaviour of

definable elements. However, it seems to be possible to obtain a model from a sound interpretation by restricting everything to definable elements.

This also implies that every valid type (or context) is defined and that every typable term is defined and in the appropriate set.

Note that we never have to verify any of the congruence rules : they are automatically true due to our choice to use equality in the semantics to interpret the judgements.

3.1.6. REMARK. Note that we really require the sets to be identical, which is stronger then the usual categorical formulation where we interpret equality of types by an isomorphism of objects. However, when interpreting syntax this leads to coherence problems, i.e. we have to be sure that every diagram which consists only of isomorphisms commutes. Here, we go another way and avoid coherence problems by building *a canonical choice* into the semantics, thus achieving equality and not just isomorphism. [1] However, note that the definition of an interpretation becomes slightly more clumsy, so maybe it would be better to have a good understanding of coherence for the Calculus of Constructions.

3.2. CC-structures

We will unify the model constructions presented here by introducing structures resembling Henkin models. We will do this in two steps: First we define LF-structures [2] which are sufficient to interpret dependent types with Π-types (products) and based on this we define CC-structures to interpret the Calculus of Constructions. LF-structures may be compared to closed comprehension categories (CComprC) [Jac91], and CC-structures to CComprCs with a (fibred) weakly reflective subcategory with a generic object (i.e. essentially a *dictos* [Ehr89]).

---

[1] This was proposed to me by Thomas Streicher. He used this idea already in [Str89].

[2] Here we use the term LF in a loose sense to refer to a basic typed λ-calculus with dependent types(see figure 1.2).

To introduce LF-structures we define a number of semantic operations on sets which reflect the syntactic operations in LF:

3.2.1. DEFINITION (Semantic operations). Let A be a set, $\{B_a, B'_a\}_{a\in A}$ and $\{C_p\}_{p\in(\Sigma a\in A.B_a)}$ families of sets.

projections Assume $f \in \Pi a \in A.B_a$ then

$$\begin{aligned} f^{+\{B'_a\}_a} &= p \in (\Sigma a \in A.B'_a) \mapsto f(\pi_1(p)) \\ &\in \Pi p \in (\Sigma a \in A.B'_a).B_{\pi_1(p)} \\ \mathrm{pr}_{A,\{B_a\}_a} &= p \in (\Sigma a \in A.B_a) \mapsto \pi_2(p) \\ &\in \Pi p \in (\Sigma a \in A.B_a).B_{\pi_1(p)} \end{aligned}$$

composition Assume $f \in \Pi a \in A.\Pi b \in B_a.C_{(a,b)}$ and $g \in \Pi a \in A.B_a$ then

$$\begin{aligned} f[g] &= a \in A \mapsto f(a)(g(a)) \\ &\in \Pi a \in A.C_{(a,g(a))} \end{aligned}$$

currying Assume $f \in \Pi p \in (\Sigma a \in A.B_a).C_p$:

$$\begin{aligned} \lambda(f) &= a \in A \mapsto b \in B_a \mapsto f(a,b) \\ &\in \Pi a \in A.\Pi b \in B_a.C_{(a,b)} \end{aligned}$$

LF-structures provide the basic semantic components to interpret a calculus with dependent types: We have **1** corresponding to an empty context and Σ for context comprehension; to any context and family of types over it we assign the set of sections (Sect) [3] which corresponds to the interpretations of typed terms in a context. Π corresponds to Π-types. The simplest LF-structure is the full, set-theoretic interpretation lemma 3.3.1.

3.2.2. DEFINITION (LF-structure).

$$\mathcal{L} = (\mathfrak{U}_{\mathrm{Co}}, \mathfrak{U}_{\mathrm{Ty}}, \mathbf{1}, \Sigma, \mathrm{Sect}, \Pi)$$

with

---

[3] Here our terminology is influenced by the categorical model constructions.

- $\mathfrak{U}_{\mathrm{Co}}, \mathfrak{U}_{\mathrm{Ty}}$ are universes.
- $\mathbf{1} \in \mathfrak{U}_{\mathrm{Co}}$.
- $\Sigma(X \in \mathfrak{U}_{\mathrm{Co}}, \{Y_x \in \mathfrak{U}_{\mathrm{Ty}}\}_{x \in \overline{X}}) \in \mathfrak{U}_{\mathrm{Co}}$ s.t. $\overline{\Sigma(X, \{Y_x\}_x)} = \Sigma x \in \overline{X}.\overline{Y_x}$
- $\mathrm{Sect}(X \in \mathfrak{U}_{\mathrm{Co}}, \{Y_x \in \mathfrak{U}_{\mathrm{Ty}}\}_{x \in \overline{X}}) \subseteq \Pi x \in \overline{X}.\overline{Y_x}$.
- $\Pi(X \in \mathfrak{U}_{\mathrm{Ty}}, \{Y_x \in \mathfrak{U}_{\mathrm{Ty}}\}_{x \in \overline{X}}) \in \mathfrak{U}_{\mathrm{Ty}}$ s.t. $\overline{\Pi(X, \{Y_x\}_x)} \subseteq \Pi x \in \overline{X}.\overline{Y_x}$

is an LF-structure if the following conditions are satisfied:

1. $\overline{\mathbf{1}}$ is a one-element set.
2. $$\frac{f \in \mathrm{Sect}(X, \{Z_x\}_x)}{f^{+\{\overline{Y_x}\}_x} \in \mathrm{Sect}(\Sigma(X, \{Y_x\}_x), \{Z_{\pi 1(p)}\}_p)}$$
3. $\mathrm{pr}_{\overline{X}, \{\overline{Y_x}\}_x} \in \mathrm{Sect}(\Sigma(X, \{Y_x\}_x), \{Y_{\pi 1(p)}\}_p)$
4. $$\frac{f \in \mathrm{Sect}(X, \{\Pi(Y_x, \{Z_{(x,y)}\}_{y \in Y_x})\}_x) \qquad g \in \mathrm{Sect}(X, \{Y_x\}_x)}{f[g] \in \mathrm{Sect}(X, Z_{(x,g(x))})}$$
5. $$\frac{f \in \mathrm{Sect}(\Sigma(X, \{Y_x\}_x), \{Z_p\}_p)}{\lambda(f) \in \mathrm{Sect}(X, \{\Pi(Y_x, \{Z_{(x,y)}\}_{y \in Y_x})\}_x)}$$

3.2.3. Remark (LF). We can interpret the logical framework, i.e. the calculus without the rules regarding Set,$\forall$ and El [4] in LF-structures in the following way: Assume $\mathcal{L} = (\mathfrak{U}_{\mathrm{Co}}, \mathfrak{U}_{\mathrm{Ty}}, \mathbf{1}, \Sigma, \mathrm{Sect}, \Pi)$ then

$$\begin{aligned}
[\![\vdash \bullet]\!]^{\mathcal{L}} &:\cong \mathbf{1} \\
[\![\vdash \Gamma.\sigma]\!]^{\mathcal{L}} &:\cong \Sigma([\![\vdash \Gamma]\!]^{\mathcal{L}}, [\![\Gamma \vdash \sigma]\!]^{\mathcal{L}}) \\
[\![\Gamma \vdash \Pi\sigma.\tau]\!]^{\mathcal{L}} &:\cong \{\Pi([\![\Gamma \vdash \sigma]\!]^{\mathcal{L}}\gamma, \{[\![\Gamma.\sigma \vdash \tau]\!]^{\mathcal{L}}(\gamma, x)\}_x)\}_{\gamma \in [\![\vdash \Gamma]\!]^{\mathcal{L}}} \\
[\![\Gamma.\sigma \vdash 0]\!]^{\mathcal{L}} &:\cong \mathrm{pr}_{[\![\vdash \Gamma]\!]^{\mathcal{L}}, [\![\Gamma \vdash \sigma]\!]^{\mathcal{L}}} \\
[\![\Gamma.\tau \vdash i+1]\!]^{\mathcal{L}} &:\cong ([\![\Gamma \vdash i : \sigma]\!]^{\mathcal{L}})^{+[\![\Gamma \vdash \tau]\!]^{\mathcal{L}}}
\end{aligned}$$

---

[4] Note, however, that we have to introduce some constants to have an interesting theory because there are no syntactic objects denotable in the core calculus as described.

$$[\![\Gamma \vdash \lambda\sigma(M)^\tau]\!]^{\mathcal{L}} \quad :\cong \quad \lambda([\![\Gamma.\sigma \vdash M : \tau]\!]^{\mathcal{L}})$$
$$[\![\Gamma \vdash \mathrm{app}^{\sigma,\tau}(M,N)]\!]^{\mathcal{L}} \quad :\cong \quad [\![\Gamma \vdash M : \Pi\sigma.\tau]\!]^{\mathcal{L}}[[\![\Gamma \vdash N : \sigma]\!]^{\mathcal{L}}]$$

Note that that for any $\Gamma \vdash M : \sigma$ we have $[\![\Gamma \vdash M]\!]^{\mathcal{L}} \in \mathrm{Sect}([\![\vdash \Gamma]\!]^{\mathcal{L}}, [\![\Gamma \vdash \sigma]\!]^{\mathcal{L}})$.

3.2.4. REMARK. In many cases one has $\mathfrak{U}_{\mathrm{Co}} = \mathfrak{U}_{\mathrm{Ty}}$ and $\mathrm{Sect}(X, \{Y_x\}_x) = \overline{\Pi(X, \{Y_x\}_x)}$. However, this is not reflected in the syntax:

- So far we have not introduced a notation for Σ-types.
- Even in the presence of Σ-types the equality introduced by substitution is syntactical identity whereas the equality introduced by contraction of projections is the judgemental equality [5]

This can also be compared with the difference between locally cartesian closed categories (LCCCs) and CComprCs.

We will now define CC-structure by adding additional structure to an LF-structure. To understand the motivation behind the following definition it is useful to have a look at a concrete structure, e.g. 3.3.1, see also 3.3.2.

3.2.5. DEFINITION (CC-structures).

$$\mathcal{C} = (\mathcal{L}, \mathfrak{M}, \mathrm{SET}, \mathrm{EL}, \mathrm{EL}^{-1}, \vartheta)$$

with

- $\mathcal{L} = (\mathfrak{U}_{\mathrm{Co}}, \mathfrak{U}_{\mathrm{Ty}}, \mathbf{1}, \Sigma, \mathrm{Sect}, \Pi)$ is an LF-structure.
- $\mathfrak{M} \subseteq \mathfrak{U}_{\mathrm{Ty}}$ is the subclass of *modest sets*.
- $\mathrm{SET} \in \mathfrak{U}_{\mathrm{Ty}}$.

---

[5] This leads naturally to a calculus with explicit substitutions as in [ACCL90].

- $\mathrm{EL}(A \in \overline{\mathrm{SET}}) \in \mathfrak{M}$ and $\mathrm{EL}^{-1}(A \in \mathfrak{M}) \in \overline{\mathrm{SET}}$.

- $\vartheta_{X \in \mathfrak{M}} \in \overline{X} \to \overline{\mathrm{EL}(\mathrm{EL}^{-1}(X))}$.

is a CC-structure if the following conditions are satisfied:

1. $\mathrm{Sect}(X, \mathrm{SET}) = \overline{X} \to \overline{\mathrm{SET}}$.

2. $\mathrm{EL}^{-1}(\mathrm{EL}(A)) = A$.

3. $\Pi(X, \{Y_x \in \mathfrak{M}\}_x \in \mathfrak{M}) \in \mathfrak{M}$.

4. ϑ_X is a bijection and

$$\frac{f \in \mathrm{Sect}(X, \{Y_x \in \mathfrak{M}\}_x)}{\vartheta \circ f \in \mathrm{Sect}(X, \{\mathrm{EL}(\mathrm{EL}^{-1}(Y_x))\}_x)}$$

3.2.6. Remark. Categorically, the conditions for CC can be expressed by saying that modest sets constitute a (fibred) reflective subcategory with a generic object. It should be noted that we do *not* require that EL^{-1} is inverse to EL. Indeed this is not the case in any of the constructions we are considering.

We will now assign an interpretation to every CC-structure and then verify that it is sound. For the following assume as given a CC-structure

$$\mathcal{C} = (\mathcal{L} = (\mathfrak{U}_{\mathrm{Co}}, \mathfrak{U}_{\mathrm{Ty}}, \mathbf{1}, \Sigma, \mathrm{Sect}, \Pi), \mathfrak{M}, \mathrm{SET}, \mathrm{EL}, \mathrm{EL}^{-1}, \vartheta).$$

We will first define two auxiliary operations:

$$\begin{aligned} \Theta(X \in \mathfrak{U}_{\mathrm{Ty}}) &= \begin{cases} \mathrm{EL}(\mathrm{EL}^{-1}(X) & \text{if } X \in \mathfrak{M} \\ X & \text{otherwise} \end{cases} \\ \tilde{\vartheta}_{X \in \mathfrak{U}_{\mathrm{Ty}}}(x \in \overline{X}) &= \begin{cases} \vartheta_X(x) & \text{if } X \in \mathfrak{M} \\ x & \text{otherwise} \end{cases} \\ &\in \overline{\Theta(X)} \end{aligned}$$

3.2.7. DEFINITION (Interpretation in CC-Structures). We define an interpretation as follows:

$$
\begin{aligned}
[\![\vdash \bullet]\!]^{\mathcal{C}} &:\cong \mathbf{1}\\
[\![\vdash \Gamma.\sigma]\!]^{\mathcal{C}} &:\cong \Sigma([\![\vdash \Gamma]\!]^{\mathcal{C}}, [\![\Gamma \vdash \sigma]\!]^{\mathcal{C}})\\
[\![\Gamma \vdash \Pi\sigma.\tau]\!]^{\mathcal{C}} &:\cong \{\Theta(\Pi([\![\Gamma \vdash \sigma]\!]^{\mathcal{C}}\gamma, \{[\![\Gamma.\sigma \vdash \tau]\!]^{\mathcal{C}}(\gamma, x)\}_x))\}_{\gamma \in [\![\vdash \Gamma]\!]^{\mathcal{C}}}\\
[\![\Gamma \vdash \mathrm{Set}]\!]^{\mathcal{C}} &:\cong \{\mathrm{SET}\}_{\gamma \in [\![\vdash \Gamma]\!]^{\mathcal{C}}}\\
[\![\Gamma \vdash \mathrm{El}(A)]\!]^{\mathcal{C}} &:\cong \{\mathrm{EL}([\![\Gamma \vdash A]\!]^{\mathcal{C}}\gamma)\}_{\gamma \in [\![\vdash \Gamma]\!]^{\mathcal{C}}}\\
[\![\Gamma.\sigma \vdash 0]\!]^{\mathcal{C}} &:\cong \mathrm{pr}_{[\![\vdash \Gamma]\!]^{\mathcal{C}}, [\![\Gamma \vdash \sigma]\!]^{\mathcal{C}}}\\
[\![\Gamma.\tau \vdash i+1]\!]^{\mathcal{C}} &:\cong ([\![\Gamma \vdash i]\!]^{\mathcal{C}})^{+[\![\Gamma \vdash \tau]\!]^{\mathcal{C}}}\\
[\![\Gamma \vdash \lambda\sigma(M)^{\tau}]\!]^{\mathcal{C}} &:\cong \tilde{\vartheta}_{[\![\Gamma \vdash \Pi\sigma.\tau]\!]^{\mathcal{C}}\gamma} \circ \lambda([\![\Gamma.\sigma \vdash M]\!]^{\mathcal{C}})\\
[\![\Gamma \vdash \mathrm{app}^{\sigma,\tau}(M,N)]\!]^{\mathcal{C}} &:\cong (\tilde{\vartheta}^{-1}_{[\![\Gamma \vdash \Pi\sigma.\tau]\!]^{\mathcal{C}}\gamma} \circ [\![\Gamma \vdash M]\!]^{\mathcal{C}})[[\![\Gamma \vdash N]\!]^{\mathcal{C}}]\\
[\![\Gamma \vdash \forall^{\sigma}.A]\!]^{\mathcal{C}} &:\cong \{\mathrm{EL}^{-1}(\Pi([\![\Gamma \vdash \sigma]\!]\gamma, \{\mathrm{EL}([\![\Gamma.\sigma \vdash A]\!]^{\mathcal{C}}(\gamma, x))\}_x))\}_{\gamma}
\end{aligned}
$$

Note that we use ϑ and Θ to coerce the interpretation of Sets to their canonical meanings. This technique is already used in [Str91] to give an interpretation of CC up-to-equality instead of merely up-to-isomorphism (which would impose a coherence problem).

We first have to verify that weakening and substitution are interpreted correctly:

3.2.8. LEMMA (Soundness of weakening). *For any* $\gamma \in \overline{[\![\vdash \Gamma]\!]^{\mathcal{C}}}$, $\gamma\delta \in \overline{[\![\vdash \Gamma.\Delta]\!]^{\mathcal{C}}}$ *and* $x \in \overline{[\![\Gamma \vdash \tau]\!]^{\mathcal{C}}\gamma}$ *we have*

$$
\begin{aligned}
[\![\Gamma.\Delta \vdash \sigma]\!]^{\mathcal{C}}\gamma\delta &\cong [\![\Gamma.\tau.\Delta^{+} \vdash \sigma^{+|\Delta|}]\!]^{\mathcal{C}}\gamma x\delta\\
[\![\Gamma.\Delta \vdash M]\!]^{\mathcal{C}}\gamma\delta &\cong [\![\Gamma.\tau.\Delta^{+} \vdash M^{+|\Delta|}]\!]^{\mathcal{C}}\gamma x\delta
\end{aligned}
$$

Proof. By induction over the structure of σ and M. Most cases follow straightforwardly by just applying definition 3.2.7 and if necessary the induction hypothesis. Let us therefore only consider the case $M = i$ here. Note that $[\![\Gamma \vdash i]\!]^{\mathcal{C}} = \pi_2 \circ \pi_1^i$. We have to distinguish the following cases (see definition 2–1):

$i >= |\Delta|$

$$\begin{aligned}
[\![\Gamma.\Delta \vdash i]\!]^{\mathcal{C}}\gamma\delta &= \pi_2(\pi_1^i(\gamma\delta)) \\
&= \pi_2(\pi_1^{i+1}(\gamma x\delta)) && \text{because } i >= |\Delta| \\
&= [\![\Gamma.\tau.\Delta^+ \vdash i+1 = i^{+|\Delta|}]\!]^{\mathcal{C}}\gamma x\delta
\end{aligned}$$

$i < |\Delta|$

$$\begin{aligned}
[\![\Gamma.\Delta \vdash i]\!]^{\mathcal{C}}\gamma\delta &= \pi_2(\pi_1^i(\gamma\delta)) \\
&= \pi_2(\pi_1^i(\gamma x\delta)) && \text{because } i < |\Delta| \\
&= [\![\Gamma.\tau.\Delta^+ \vdash i = i^{+|\Delta|}]\!]^{\mathcal{C}}\gamma x\delta
\end{aligned}$$

3.2.9. LEMMA (Soundness of substitution). *For any* $\gamma \in \overline{[\![\vdash \Gamma]\!]^{\mathcal{C}}}$ *and* $\gamma\delta \in \overline{[\![\vdash \Gamma.\Delta[N]]\!]^{\mathcal{C}}}$ *we have:*

$$\begin{aligned}
[\![\Gamma.\tau.\Delta \vdash \sigma]\!]^{\mathcal{C}}\gamma([\![\Gamma \vdash N]\!]^{\mathcal{C}}\gamma)\delta &\cong [\![\Gamma.\Delta[N] \vdash \sigma[N]^{|\Delta|}]\!]^{\mathcal{C}}\gamma\delta \\
[\![\Gamma.\tau.\Delta \vdash M]\!]^{\mathcal{C}}\gamma([\![\Gamma \vdash N]\!]^{\mathcal{C}}\gamma)\delta &\cong [\![\Gamma.\Delta[N] \vdash M[N]^{|\Delta|}]\!]^{\mathcal{C}}\gamma\delta
\end{aligned}$$

Proof. By induction over the structure of σ and M. The same remarks as for lemma 3.2.8 apply. Consider $M = i$ again. Note that:

$$[\![\Gamma.\tau.\Delta \vdash i]\!]^{\mathcal{C}}\gamma([\![\Gamma \vdash N]\!]^{\mathcal{C}}\gamma)\delta = \pi_2(\pi_1^i(\gamma([\![\Gamma \vdash N]\!]^{\mathcal{C}}\gamma)\delta))$$

We have two check three cases (see definition 2–1):

$i < |\Delta|$

$$\begin{aligned}
&\pi_2(\pi_1^i(\gamma([\![\Gamma \vdash N]\!]^{\mathcal{C}}\gamma)\delta)) \\
&= \pi_2(\pi_1^i(\gamma\delta) && \text{because } i < |\Delta| \\
&= [\![\Gamma.\Delta \vdash i = i[N]^{|\Delta|}]\!]^{\mathcal{C}}\gamma\delta
\end{aligned}$$

$i = |\Delta|$

$$\begin{aligned}
&\pi_2(\pi_1^i(\gamma([\![\Gamma \vdash N]\!]^{\mathcal{C}}\gamma)\delta)) \\
&= [\![\Gamma \vdash N]\!]^{\mathcal{C}}\gamma && \text{because } i = |\Delta| \\
&= [\![\Gamma \vdash N^{+|\Delta|} = i[N]]^{|\Delta|}]\!]^{\mathcal{C}}\gamma\delta && \text{by lemma 3.2.8}
\end{aligned}$$

$i > |\Delta|$

$$\begin{aligned}
&\pi_2(\pi_1^i(\gamma([\![\Gamma \vdash N]\!]^{\mathcal{C}}\gamma)\delta)) \\
&= \pi_2(\pi_1^{i-1}(\gamma\delta) && \text{because } i > |\Delta| \\
&= [\![\Gamma.\Delta \vdash i-1 = i[N]^{|\Delta|}]\!]^{\mathcal{C}}\gamma\delta
\end{aligned}$$

3.2.10. THEOREM (Soundness). *$[\![\vdash \Gamma]\!]^{\mathcal{C}}$, $[\![\Gamma \vdash \sigma]\!]^{\mathcal{C}}$ and $[\![\Gamma \vdash M]\!]^{\mathcal{C}}$ defines a sound interpretation of the calculus definition 3.1.5. In particular we have that:*

$$\frac{\Gamma \vdash M : \sigma}{[\![\Gamma \vdash M]\!]^{\mathcal{C}} \in \mathrm{Sect}[\![\vdash \Gamma]\!]^{\mathcal{C}}[\![\Gamma \vdash \sigma]\!]^{\mathcal{C}}}$$

$$\frac{\Gamma \vdash \sigma \simeq \tau}{[\![\Gamma \vdash \sigma]\!]^{\mathcal{C}} = [\![\Gamma \vdash \tau]\!]^{\mathcal{C}}}$$

$$\frac{\Gamma \vdash M \simeq N : \sigma}{[\![\Gamma \vdash M]\!]^{\mathcal{C}} = [\![\Gamma \vdash N]\!]^{\mathcal{C}}}$$

Proof. We show this by mutual induction over the derivations. Many cases are obvious, in particular:

- The definedness conditions (definition 3.1.5 1.,2.) follow directly from the premises.
- All congruence rules and (CONV) are direct consequences of the fact that we use semantic equality to interpret equality judgements.

(ALL-ELIM)

$$\begin{aligned}
&[\![\Gamma \vdash \mathrm{El}(\forall\sigma.\ A)]\!]^{\mathcal{C}}\gamma \\
&= \{\mathrm{EL}(\mathrm{EL}^{-1}(\Pi([\![\Gamma \vdash \sigma]\!]\gamma, \{\mathrm{EL}([\![\Gamma.\sigma \vdash A]\!]^{\mathcal{C}}(\gamma, x))\}_x)))\}_\gamma \\
&= \{\Theta(\Pi([\![\Gamma \vdash \sigma]\!]\gamma, \{\mathrm{EL}([\![\Gamma.\sigma \vdash A]\!]^{\mathcal{C}}(\gamma, x))\}_x))\}_\gamma \\
&= [\![\Pi\sigma.\mathrm{El}(A)]\!]^{\mathcal{C}}\gamma
\end{aligned}$$

(VAR) Just for $n = 0$:

$$\begin{aligned}
&[\![\Gamma.\sigma \vdash 0]\!]^{\mathcal{C}} \\
&= \mathrm{pr}_{[\![\vdash\Gamma]\!]^{\mathcal{C}}, [\![\Gamma\vdash\sigma]\!]^{\mathcal{C}}} \\
&\in \mathrm{Sect}(\Sigma([\![\vdash \Gamma]\!]^{\mathcal{C}}, [\![\Gamma \vdash \sigma]\!]^{\mathcal{C}})\{[\![\Gamma \vdash \sigma]\!]^{\mathcal{C}}\pi_1(\gamma)\}_\gamma) \\
&= \mathrm{Sect}([\![\vdash \Gamma.\sigma]\!]^{\mathcal{C}}, [\![\Gamma.\sigma \vdash \sigma^+]\!]^{\mathcal{C}}) && \text{lemma 3.2.8}
\end{aligned}$$

(LAM)

$$\begin{aligned}
&[\![\Gamma \vdash \lambda\sigma(M)^\tau]\!]^{\mathcal{C}} \\
&= \vartheta \circ \lambda([\![\Gamma.\sigma \vdash M]\!]^{\mathcal{C}}) \\
&\in \mathrm{Sect}([\![\Gamma]\!]^{\mathcal{C}}, \Theta(\Pi([\![\Gamma \vdash \sigma]\!]^{\mathcal{C}}, [\![\Gamma.\sigma \vdash \tau]\!]^{\mathcal{C}}))) && \text{ind.hyp.} \\
&= \mathrm{Sect}([\![\Gamma]\!]^{\mathcal{C}}, [\![\Gamma \vdash \Pi\sigma.\tau]\!]^{\mathcal{C}})
\end{aligned}$$

(APP)

$$\begin{aligned}
&[\![\Gamma \vdash \mathrm{app}^{\sigma.\tau}(M, N)]\!]^{\mathcal{C}} \\
&= (\tilde{\vartheta}^{-1}_{[\![\Gamma \vdash \Pi\sigma.\tau]\!]^{\mathcal{C}}\gamma} \circ [\![\Gamma \vdash M]\!]^{\mathcal{C}})[[\![\Gamma \vdash N]\!]^{\mathcal{C}}] \\
&\in \mathrm{Sect}([\![\Gamma]\!]^{\mathcal{C}}, \{[\![\Gamma.\sigma \vdash \tau]\!]^{\mathcal{C}}(\gamma, [\![\Gamma \vdash N]\!]^{\mathcal{C}}\gamma)\}_\gamma) && \text{ind.hyp.} \\
&= \mathrm{Sect}([\![\Gamma]\!]^{\mathcal{C}}, [\![\Gamma \vdash \tau[N]]\!]^{\mathcal{C}}) && \text{lemma 3.2.9}
\end{aligned}$$

(ALL) Note that properties 1 and 3 of definition 3.2.5 are required for this case.

(BETA)

$$\begin{aligned}
&[\![\Gamma \vdash \mathrm{app}^{\sigma.\tau}(\lambda\sigma(M)^\tau, N)]\!]^{\mathcal{C}}\gamma \\
&= (\lambda([\![\Gamma.\sigma \vdash M]\!]^{\mathcal{C}})[[\![\Gamma \vdash N]\!]^{\mathcal{C}}])\gamma \\
&= [\![\Gamma.\sigma \vdash M]\!]^{\mathcal{C}}(\gamma, [\![\Gamma \vdash N]\!]^{\mathcal{C}}\gamma) \\
&= [\![\Gamma \vdash M[N]]\!]^{\mathcal{C}}\gamma && \text{lemma 3.2.9}
\end{aligned}$$

3.3. Proof irrelevance semantics

The basic idea is that the most natural way to interpret types is to interpret type-theoretic constructs by their set-theoretic counterparts. In particular Π will be interpreted by Π on sets (3.1.1), i.e. as for simply typed λ-calculus one may call this a *full model*. Although this approach works fine for Martin-Löf Type Theory it has a shortcoming for impredicative systems: We have to identify all inhabitants of *sets*, i.e. we interpret Set (which we should rather call Prop here) as truth values and every inhabitant of a proposition will be mapped to a canonical one.

We note that the class of sets gives rise to an LF-structure:

3.3.1. LEMMA. *Let $\mathfrak{S}$ be the universe of sets with $\overline{X} = X$ and:*

- $\mathbf{1}_{\mathcal{S}} = \{\epsilon\}$.
- $\Sigma_{\mathcal{S}}(A, \{B_a\}_a) = \Sigma a \in A.B_a$.
- $\Pi_{\mathcal{S}}(A, \{B_a\}_a) = \Pi a \in A.B_a$.

$\mathcal{S} = (\mathfrak{S}, \mathfrak{S}, \mathbf{1}_{\mathcal{S}}, \Sigma_{\mathcal{S}}, \overline{\Pi}_{\mathcal{S}}, \Pi_{\mathcal{S}})$ *is an LF-structure.*

Proof. Follows directly from definition 3.2.1.

For the following let $\mathbf{2} = \{\emptyset, \mathbf{1}\}$ a canonical presentation of truth values, i.e. false $= \emptyset$ and true $= \mathbf{1}$. We call a set A with at most one element a *singleton* [6] , i.e. $\emptyset$ and $\mathbf{1}$ are singletons.

3.3.2. REMARK (Understanding CC-structures). This interpretation as simple as it is can be used to understand the motivation behind our definition of CC-structures and the definition of the interpretation 3.2.7: To interpret (REFL) we need that Π preserves singletons in the following sense: If $\{B_a\}_{a \in A}$ is a family of singletons then $\Pi a \in A.B_a$ is a singleton. However this is not sufficient to make the interpretation of (REFL) sound because we require this set to be *equal* to a canonical singleton (i.e. element of $\mathbf{2}$). This problem is solved by mapping every singleton set to a canonical one. This is expressed by $\Theta = \mathrm{EL} \circ \mathrm{EL}^{-1}$. This coercion operation is accompanied by a bijection of values which is given by θ. Now we can understand the use of θ in 3.2.7: Whenever a value in a single valued function space is constructed it has to be coerced to the canonical representation $\emptyset \in$ *true* by θ and whenever it is going to be applied the unique function it represents has to be recovered by θ^{-1}.

---

[6] Note the non-standard terminology here — usually singletons are considered to be non-empty.

3.3.3. THEOREM ($\mathcal{S}^+$ is a CC-structure). *Let $\mathfrak{M}_{\mathcal{S}}$ denote the class of singletons and:*

$$\mathrm{EL}_{\mathcal{S}}^{-1}(X) = \begin{cases} \text{false} & \text{if } X = \emptyset \\ \text{true} & \text{otherwise} \end{cases}$$

and $\vartheta_{\mathcal{S}X}(x) = \emptyset \in \text{true}$ (Note that $\vartheta_{\mathcal{S}}$ will be never applied if X is empty.).

$$\mathcal{S}^+ = (\mathcal{S}, \mathfrak{M}_{\mathcal{S}}, \mathbf{2}, X \mapsto X, \mathrm{EL}_{\mathcal{S}}^{-1}, \vartheta_{\mathcal{S}})$$

is a CC-structure.

Proof. Easy.

3.3.4. COROLLARY. *The calculus is logically sound.*

Proof. Consider

$$[\![\vdash \forall^{X:\mathrm{Set}} X : \mathrm{Set}]\!]^{\mathcal{S}} = \bigcap_{X \in \{\emptyset, \mathbf{1}\}} X = \emptyset$$

by soundness (3.2.10) we know that there can not be a term $\vdash M : \mathrm{El}(\forall^{X:\mathrm{Set}} X)$

3.4. Realizability interpretation

The essential shortcoming of the proof-irrelevance interpretation is that Set is equationally inconsistent, i.e. all equations between elements (aka proofs) hold. Indeed we cannot make this interpretation proof relevant and at the same time retain the full interpretation of Π-types. This is reflected in the slogan that *polymorphism (i.e. impredicativity) is not set-theoretic* (this has been studied in the case of System F in [Rey84]).

The solution to this problem is to restrict the elements of Π-types to dependent functions which can be *tracked* by some partial recursive function. This is reflected in the ω-Set semantics which can be generalized by using arbitrary *partial combinatory algebras*, which are called D-sets by Streicher. In [Str91] one can find a very comprehensive study of the D-set semantics using categorical tools. Here we will show that D-sets give rise to a CC-structure. Parts of this proof can be *reused* in the strong normalization proof.

3.4.1. Partial combinatory algebras

Let us repeat some fundamental definitions here:

3.4.1. DEFINITION (Partial equivalence relation (PER)). Let A be a set and $R \subseteq A \times A$. R is a *partial equivalence relation*, i.e. $R \in \mathrm{PER}(A)$ if R is symmetric and transitive. The domain of R is the subset of A where R is reflexive:

$$\mathrm{dom}(R) = \{a \in A \mid aRa\}.$$

3.4.2. DEFINITION (Quotient). We define the quotient of A wrt. R as

$$A/R = \{p \subseteq A \mid p \neq \emptyset \wedge \forall_{x,y \in A}(x \in p \wedge xRy) \rightarrow y \in p \wedge \forall_{x,y \in p} xRy\}.$$

3.4.3. DEFINITION (Partial combinatory algebra (PCA)).
$(D, \cdot, k, s)$ with

- D is a set.
- $\cdot$ is a partial function $D \times D \rightarrow D$.
- $k, s \in D$

is a *partial combinatory algebra* if the following holds:

1. $k \cdot x \cdot y = x$
2. $s \cdot x \cdot y$ is defined.
3. $s \cdot x \cdot y \cdot z \cong x \cdot z \cdot (y \cdot z)$.

A PCA is non-trivial iff D is not a singleton, which is equivalent to $k \neq s$.

3.4.4. DEFINITION (Language of PCAs). Let

$$\mathrm{Tm}^n_{\mathrm{PCA}} ::= i < n | S \mid K \mid \mathrm{Tm}^n_{\mathrm{PCA}} \mathrm{Tm}^n_{\mathrm{PCA}}$$

and $\cong^n_{\mathrm{PCA}} \subseteq \mathrm{Tm}^n_{\mathrm{PCA}} \times \mathrm{Tm}^n_{\mathrm{PCA}}$ the least PER generated by the three conditions for PCAs (interpreting definedness as reflexivity as before).

Let $I = SKK$ and $\mathrm{Tm}_{\mathrm{PCA}} = \mathrm{Tm}^0_{\mathrm{PCA}}$.

3.4.5. FACT. $(\mathrm{Tm}_{\mathrm{PCA}}, \_\,\_, K, S)$ is the initial PCA i.e. given any PCA D we have an (obvious) partial evaluation map: $(D, \cdot, k, s)$, we have

$$[\![\_]\!]_D \in \mathrm{Tm}_{\mathrm{PCA}} \to D$$

which is sound, i.e. preserves equality and definedness.

This can be extended to

$$[\![\_]\!]^n_D \in \mathrm{Tm}^n_{\mathrm{PCA}} \to D^n \to D$$

The following is the motivating example for PCAs:

3.4.6. FACT (Kleene application). Let $\{i\}j$ be the application of the ith partial recursive function to j. Then there are $s, k \in \omega$ such that $(\omega, \{\_\}\_, s, k)$ is a PCA.

We can now restate the famous theorem due to Schönfinkel in terms of PCAs:

3.4.7. THEOREM (Schönfinkel). *PCAs are functional complete, i.e. for every* $n \in \omega, M \in \mathrm{Tm}^{n+1}_{\mathrm{PCA}}$ *there is a* $\overline{\lambda}M \in \mathrm{Tm}^n_{\mathrm{PCA}}$ *such that*

$$(\overline{\lambda}M)N \cong M[N]$$

where$M[N]$ *denotes substitution.*

Proof. We define $\overline{\lambda}M$ by induction over the structure of terms:

$$\begin{aligned} \overline{\lambda}0 &= I \\ \overline{\lambda}(i+1) &= Ki \\ \overline{\lambda}(MN) &= S(\overline{\lambda}M)(\overline{\lambda}N) \\ \overline{\lambda}M &= M \qquad \text{otherwise} \end{aligned}$$

3.4.8. FACT (Pairing). We have $P, P_1, P_2 \in \mathrm{Tm}_{\mathrm{PCA}}$ such that

$$\begin{aligned} P_1(PMN) &= M \\ P_2(PMN) &= N \end{aligned}$$

Proof. Let

$$\begin{aligned} P &= \overline{\lambda}xy.\overline{\lambda}p.pxy \\ P_1 &= \overline{\lambda}p.p(\overline{\lambda}xy.x) \\ P_2 &= \overline{\lambda}p.p(\overline{\lambda}xy.y) \end{aligned}$$

and apply 3.4.7.

3.4.2. Interpreting dependent types

Using the well known material of the previous section we will now construct the LF-structure of D-sets:

3.4.9. DEFINITION (D-sets).
Let D be a set then a D-set X is a pair $(\overline{X}, \Vdash_X)$ with X is a set and $\Vdash_X \subseteq D \times \overline{X}$ s.t. $\forall_{x\in\overline{X}}\exists_{i\in D} i \Vdash_X x$. The class of D-sets together with the operation $\overline{X}$ which assigns a set to every D-set constitutes the universe $\mathfrak{D}$.

For the following we have to assume that there is a PCA $(D, \cdot, k, s)$. We will also abuse notation and confuse terms and elements of D, e.g. $\overline{\lambda}p.d_f(P_1 d) := [\![\overline{\lambda}p.d_f(P_1 p)]\!]^1_D(d_f)$.

3.4.10. LEMMA. *Assume $X \in \mathfrak{D}$, $\{Y_x \in \mathfrak{D}\}_{x\in\overline{X}}$ and let:*

$$\begin{aligned} \mathbf{1}_{\mathcal{D}} &= (\{\epsilon\}, D \times \{\epsilon\}) \\ \Sigma_{\mathcal{D}}(X, \{Y_x\}_{x\in\overline{X}}) &= (\Sigma_{x\in\overline{X}}\overline{Y_x}, \{(i,(x,y)) \mid P_1\, i \Vdash_X x \wedge P_2\, i \Vdash_{Y_x} y\}) \\ \Pi_{\mathcal{D}}(X, \{Y_x\}_{x\in\overline{X}}) &= (\{f \in \Pi_{x\in\overline{X}}\overline{Y_x} \mid \exists_{i\in D} i \Vdash_\Pi f\}, \Vdash_\Pi) \\ &\quad \textit{where } \Vdash_\Pi = \{(i,f) \mid \forall_{x\in\overline{X}}\forall_{j\in D} j \Vdash_X x \to i\, j \Vdash_{Y_x} f(x)\} \end{aligned}$$

$\mathcal{D} = (\mathfrak{D}, \mathfrak{D}, \mathbf{1}_{\mathcal{D}}, \Sigma_{\mathcal{D}}, \overline{\Pi}_{\mathcal{D}}, \Pi_{\mathcal{D}})$ is an LF-structure.

Proof. It is obvious that $\mathbf{1}_{\mathcal{D}}, \Pi_{\mathcal{D}}(X,Y) \in \mathfrak{D}$ to see that $\Sigma_{\mathcal{D}}(X,Y) \in \mathfrak{D}$, assume $d_x \Vdash_X x$, $d_y \Vdash_{Y_x} y$ then $P\, d_x\, d_y \Vdash_{\Sigma_{\mathcal{D}}(X,Y)} (x,y)$.

Now, let us check the conditions (cf. definition 3.2.2) - assume $\{Y'_x \in \mathfrak{D}\}_{x\in\overline{X}}$ and $\{Z_p\}_{p\in\overline{\Sigma_{\mathcal{D}}(X,Y)}}$:

1. $\overline{\mathbf{1}_{\mathcal{D}}} = \{\epsilon\}$ is a singleton set.

2. If $d_f \Vdash_{\Pi_{\mathcal{D}}(X,Y)} f$ then $\overline{\lambda}p.d_f(P_1 p) \Vdash_{\Pi_{\mathcal{D}}(\Sigma_{\mathcal{D}}(X,Y),\{Y'_{\pi 1(p)}\}_p)} f^{+\{\overline{Y_x}\}_x}$. [7]

3. $P_2 \Vdash_{\Pi_{\mathcal{D}}(\Sigma_{\mathcal{D}}(X,Y),\{Y_{\pi 1(p)}\}_p)} \mathrm{pr}_{\overline{X},\{\overline{Y_x}\}_x}$.

4. If $d_f \Vdash_{\Pi_{\mathcal{D}}(X,\{\Pi_{\mathcal{D}}(Y_x,\{Z_{(x,y)}\}_{y\in Y_x})\}_x)} f$, $d_g \Vdash_{\Pi_{\mathcal{D}}(X,\{Y_x\}_x)} g$ then

$$\overline{\lambda}x.d_f x(d_g x) = S d_f d_g \Vdash_{\Pi_{\mathcal{D}}(X,Z_{(x,g(x))})} f[g].$$

5. If $d_f \Vdash_{\Pi_{\mathcal{D}}(\Sigma_{\mathcal{D}}(X,\{Y_x\}_x),\{Z_p\}_p)} f$ then $\overline{\lambda}xy.d_f(Pxy) \Vdash_{\Pi_{\mathcal{D}}(X,\{\Pi_{\mathcal{D}}(Y_x,\{Z_{(x,y)}\}_y)\}_x)} \lambda(f)$

The essence of this proof is that every operation defined in 3.1.1 can be *tracked* by some λ-term.

3.4.3. Interpreting constructions

The notion of a modest D-set corresponds to the singleton sets in the proof-irrelevance interpretation and just as singletons can be canonically represented by elements of $\mathbf{2}$, modest sets can be represented by PERs. Note that the following definitions and the lemma do not require any computational structure on D. [8]

3.4.11. Definition. We call X modest, iff

$$\forall_{x,y\in\overline{X}}\forall_{i\in D} i \Vdash_X x \wedge i \Vdash_X y \to x = y,$$

we denote the (sub-)universe of modest D-sets by $\mathfrak{M}_{\mathcal{D}}$.

---

[7] We abuse notation slightly, i.e. $\overline{\lambda}p.d(P_1 d) := [\![\overline{\lambda}p.d_f(P_1 p)]\!]^1_D(d_f)$.

[8] This is important because this means we can reuse them for the strong normalization proof.

3.4.12. DEFINITION. Assume $X \in \mathfrak{M}_{\mathcal{D}}$, $R \in \mathrm{PER}(D)$:

$$\begin{aligned} \mathrm{EL}_{\mathcal{D}}(R) &= (D/R, \in) \\ \mathrm{EL}_{\mathcal{D}}^{-1}(X) &= \{(i,j) \mid \exists_{x \in \overline{X}} i \Vdash_X x \ \wedge j \Vdash_X x\} \\ \vartheta_{\mathcal{D}X}(x \in \overline{X}) &= \{i \mid i \Vdash_X x\} \end{aligned}$$

3.4.13. LEMMA.

1. $\mathrm{EL}_{\mathcal{D}}(R) \in \mathfrak{M}_{\mathcal{D}}$.

2. $\Pi_{\mathcal{D}}(X, \{Y_x \in \mathfrak{M}_{\mathcal{D}}\} x \in \overline{X}) \in \mathfrak{M}_{\mathcal{D}}$

3. $\mathrm{EL}_{\mathcal{D}}^{-1}(\mathrm{EL}_{\mathcal{D}}(R)) = R$

4. $\vartheta_{\mathcal{D}X} \in \overline{X} \to \overline{\mathrm{EL}_{\mathcal{D}}(\mathrm{EL}_{\mathcal{D}}^{-1}(X))}$ *is a bijection and* $i \Vdash_X x$ *iff* $i \Vdash_{\mathrm{EL}_{\mathcal{D}}(\mathrm{EL}_{\mathcal{D}}^{-1}(X))} \vartheta_X(x)$.

Proof.

1. Follows from the definition of quotients 3.4.2.

2. Assume $i \Vdash_{\Pi(X,Y)} f, g$, Now for all $x \in \overline{X}, j \Vdash_X x$ we have $i \cdot j \Vdash_{Y_x} f(x), g(x)$ and because Y_x is modest $f(x) = g(x)$ and therfore by extensionality $f = g$.

3. Expand definitions.

4. The preservation of realizers is easy to check and implies the first half of the proposition because all D-sets concerned are modest.

Now we have sufficient material to define the CC-structure $\mathcal{D}^+$:

3.4.14. THEOREM ($\mathcal{D}^+$ is a CC-structure). *Let*

$$\mathrm{SET}_{\mathcal{D}} = (\mathrm{PER}(D), D \times \mathrm{PER}(D))$$

then

$$\mathcal{D}^+ = (\mathcal{D}, \mathfrak{M}_\mathcal{D}, \mathrm{SET}_\mathcal{D}, \mathrm{EL}_\mathcal{D}, \mathrm{EL}_\mathcal{D}^{-1}, \vartheta_\mathcal{D})$$

is a CC-structure.

Proof. Follows by lemma 3.4.10 and 3.4.13. It is easy to see that SET_D fullfills condition 1 of definition 3.2.5.

3.4.15. COROLLARY. *The calculus is equationally sound.*

Proof. We use the Kleene-PCA (3.4.6). [9] Let

$$\begin{aligned} BOOL &= \forall^{X:Set} X \supset X \supset X \\ true &= \lambda X : Set.\lambda x, y : \mathrm{El}(X).x \\ false &= \lambda X : Set.\lambda x, y : \mathrm{El}(X).y \end{aligned}$$

It is easy to see that $\vdash BOOL : \mathrm{Set}, \vdash true, false : \mathrm{El}(BOOL)$. We have

$$\begin{aligned} [\![\vdash true : \mathrm{El}(BOOL)]\!]^\mathcal{D} &= \{i \mid \forall_{R\in \mathrm{PER}} \forall_{l\in\omega} \forall_{p\in\omega/R} \forall_{m\in p} \forall_{q\in\omega/R} \forall_{n\in q} i\cdot l\cdot m\cdot n \in p\} \\ [\![\vdash false : \mathrm{El}(BOOL)]\!]^\mathcal{D} &= \{i \mid \forall_{R\in \mathrm{PER}} \forall_{l\in\omega} \forall_{p\in\omega/R} \forall_{m\in p} \forall_{q\in\omega/R} \forall_{n\in q} i\cdot l\cdot m\cdot n \in q\} \end{aligned}$$

Let $t = \overline{\lambda} lmn.m$ and $f = \overline{\lambda} lmn.n$. By 3.4.7) we know $t\cdot l\cdot m\cdot n = m$ and $f\cdot l\cdot m\cdot n = n$. Certainly $t \in [\![\vdash true : \mathrm{El}(BOOL)]\!]^\mathcal{D}$ and $f \in [\![\vdash false : \mathrm{El}(BOOL)]\!]^\mathcal{D}$. To see that $t \notin [\![\vdash false : \mathrm{El}(BOOL)]\!]^\mathcal{D}$ choose $R = \{(i,i) \mid i \in \omega\}$ (the discrete PER), any l, $p = \{0\}, m = 0, q = \{1\}, n = 1\}$ then $i\cdot l\cdot m\cdot n \in p = 0 \notin q = \{1\}$. Therfore $[\![\vdash true : \mathrm{El}(BOOL)]\!]^\mathcal{D} \neq [\![\vdash false : \mathrm{El}(BOOL)]\!]^\mathcal{D}$ and by theorems 3.2.10 and 3.4.14 $\Gamma \vdash true \cong false : BOOL$ is not derivable.

3.4.16. REMARK. There is a simpler syntactic proof of this fact: we just have to realize that every equation which holds for typed terms is also true in the untyped λ-calculus, which is well known to be equationally consistent.

---

[9] Indeed this construction works for any consistent PCA.

There is also a syntactic proof for the logical soundness, using the strong normalization proof. However, proving SN is essentially a semantic construction, therefore it seems fair to say that there is no syntactic proof of this property.

3.4.17. REMARK. It is interesting to note that the proof irrelevance interpretation can be viewed as a special case of the D-set semantics by setting $D =$ the inconsistent PCA.

3.5. Strong Normalization

Strong normalization is a very essential property of the calculus which entails a number of important corollaries. This does not only include decidability of type checking but also facts which are not obviously related to reduction, like uniqueness of product formation and that constructors are one-to-one.

We will here first consider strong normalization in its pure form, i.e. that all pure lambda-terms typable in constructions are strongly normalizing. From this we will infer by a syntactic constructions that the judgemental reduction relation is strongly normalizing and Church-Rosser.

We want to emphasize here the fact that strong normalization proofs are essentially semantical which is reflected by obtaining a strong normalization argument as a modification of the realizability semantics. The development proceeds in a very similar way: where we have used properties of PCAs before we will use properties of the set of strongly normalizing terms here. It is also interesting to note that the properties of untyped reduction we have to verify are the same as for simply typed λ-calculus. [10]

[10] E.g. SN is *type closed*, as it is called in [Mit90].

3.5.1. Properties of SN

We will now verify in some detail the properties of SN for $\triangleright$ (e.g. definitions 2.3.5 (all statements about terms refer to curry terms), 2.3.1, 2.3.7) we need for the strong normalization proof. This is done so explicitly because the extensions to other notions of reduction follow the same pattern.

An important notion is *weak head reduction*: $\triangleright_{\mathbf{whd}} \subseteq \triangleright$ is the restriction of β reduction to the reduction of head-redexes not inside a λ-abstraction [11]. We can define $\triangleright_{\mathbf{whd}}$ inductively:

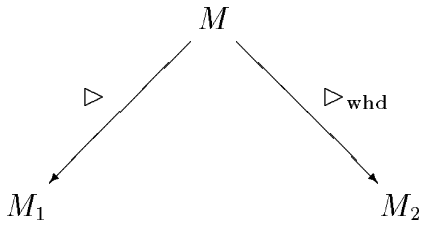
3.5.1. DEFINITION (Weak head reduction). $\triangleright_{\mathbf{whd}} \subseteq \Lambda \times \Lambda$ is the least relation closed under:

$$(\lambda M)N \triangleright_{\mathbf{whd}} M[N] \qquad \frac{M \triangleright_{\mathbf{whd}} M'}{MN \triangleright_{\mathbf{whd}} M'N}$$

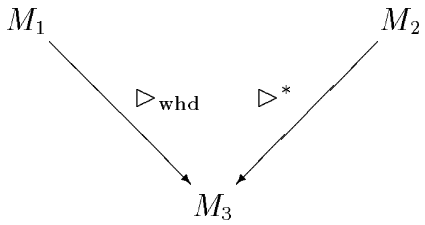
Note that this works for $\triangleright$ and $\triangleright_\eta$.

The following lemma, which is needed in the strong normalization argument, corresponds to a weak form of the standardization theorem.

3.5.2. LEMMA (Weak standardization). *If*



then either $M_1 = M_2$ *or there exists an* M_3 *s.t.*



[11] This is obviously deterministic.

Proof. Simple case analysis using lemma 2.3.8.

A strongly normalizing term which is not in constructor form (e.g. equal to λM) [12] in weak head normal form is called *void*. Here is another — syntactic — characterization of this set for $\beta\eta$-reduction:

3.5.3. Definition (Void). Void $\subseteq \Lambda$ is the smallest set closed under the following rules:

1. $i \in \text{Void}$.
2. $$\frac{M \in \text{Void} \qquad N \in \text{SN}}{MN \in \text{Void}}$$
3. $$\frac{M \in \text{SN}}{\forall M \in \text{Void}}$$

We can now summarize the syntactic properties of SN needed for the proof:

3.5.4. Lemma (Properties of SN).

1. $\text{Void} \subseteq \text{SN}$,

2. $$\frac{M, N, M[N] \in \text{SN}}{(\lambda M)N \in \text{SN}}$$

3. $$\frac{M' \rhd_{\text{whd}} M \qquad MN \in \text{SN}}{M'N \in \text{SN}}$$

Proof.

1. Follows directly from the definition of Void.

2. By (noetherian) induction over $M, N, M[N] \in \text{SN}$ it is easy to see that all one-step reducts of $(\lambda M)N$ are SN.

---

[12] Note that we do not consider $\forall$ as a constructor here.

3. By induction over $MN \in \mathrm{SN}$. We analyze one-step reducts of $M'N$, note that this can not be a β-redex because $M' \rhd_{\mathbf{whd}} M$. If N is reduced we just apply the induction hypothesis, if M was reduced we have to apply lemma 3.5.2 and then either apply the premise directly or use the induction hypothesis.

3.5.2. Saturated λ-sets

By applying definition 3.4.9 to Λ and Λ^* (sequences of λ terms) we obtain the universes of Λ-sets $\mathfrak{LAM}$ and Λ^*-sets $\mathfrak{LAM}^*$. We introduce the usual operations on Λ-sets essentially following lemma 3.4.10. Note, however, that one essential difference is that we use external sequences instead of encoding pairing internally. This reflects the dichotomy of contexts and types in the syntax.

3.5.5. DEFINITION. Assume $G \in \mathfrak{LAM}^*$, $\{Y_\gamma \in \mathfrak{LAM}\}_{\gamma\in\overline{G}}$, $X \in \mathfrak{LAM}$, $\{Z_x \in \mathfrak{LAM}\}_{x\in\overline{X}}$ and let:

$$
\begin{array}{rcl}
\mathbf{1}_\Lambda & = & (\{\epsilon\}, \{\epsilon\} \times \{\epsilon\}) \\
& \in & \mathfrak{LAM}^* \\
\Sigma_\Lambda(G, \{Y_\gamma\}_{\gamma\in\overline{G}}) & = & (\Sigma_{\gamma\in\overline{G}}\overline{Y_\gamma}, \{((\vec{M}, N), (\gamma, y)) \mid \vec{M} \Vdash_G \gamma \wedge N \Vdash_{Y_\gamma} y\}) \\
& \in & \mathfrak{LAM}^* \\
\mathrm{Sect}_\Lambda(G, \{Y_\gamma\}_{\gamma\in\overline{G}}) & = & \{f \in \Pi_{\gamma\in\overline{G}}\overline{Y_\gamma} \mid \exists_{M\in\Lambda} M \Vdash_{\mathrm{Sect}_\Lambda(G,\{Y_\gamma\}_\gamma)} f\} \\
\text{where } \Vdash_{\mathrm{Sect}_\Lambda(G,\{Y_\gamma\}_\gamma)} & = & \{(M, f) \mid \forall_{\gamma\in\overline{G}}\forall_{\vec{N}\in\Lambda^*} \vec{N} \Vdash_G \gamma \to M[\vec{N}] \Vdash_{Y_\gamma} f(\gamma)\} \\
\Pi_\Lambda(X, \{Z_x\}_{x\in\overline{X}}) & = & (\{f \in \Pi_{x\in\overline{X}}\overline{Z_x} \mid \exists_{M\in\Lambda} M \Vdash_{\Pi_\Lambda(X,\{Z_x\}_x)} f\}, \Vdash_{\Pi_\Lambda(X,\{Z_x\}_x)}) \\
& \in & \mathfrak{LAM} \\
\text{where } \Vdash_{\Pi_\Lambda(X,\{Z_x\}_x)} & = & \{(M, f) \mid \forall_{x\in\overline{X}}\forall_{N\in\Lambda} N \Vdash_X x \to MN \Vdash_{Z_x} f(x)\}
\end{array}
$$

However, $\mathfrak{LAM}$ and $\mathfrak{LAM}^*$ do not directly give rise to an LF-structure because we have not identified $\beta\eta$-equal Λ-terms. We will refrain from doing this and instead we identify a subclass of *saturated* Λ *sets* which can be interpreted as an LF- and CC-structure. A particular property of saturated Λ-sets is that all realizers are strongly normalizing.

3.5.6. DEFINITION. We call a Λ-set X saturated — $X \in \mathfrak{SAT}$ — iff the following conditions hold:

SAT1 Every realizer is strongly normalizing.

$$\forall_{M \Vdash_X x} M \in \mathrm{SN}$$

SAT2 There is a $\bot_X \in \overline{X}$ which is realized by every void term.

$$\forall_{M \in \mathsf{Void}} M \Vdash_X \bot_X$$

SAT3 The set of realizers for a certain element x is closed under weak head expansion inside SN:

$$\forall_{M \Vdash_X x} \forall_{M' \in \mathrm{SN}} (M' \rhd_{\mathbf{whd}} M) \to (M' \Vdash_X x)$$

This can be extended to $\mathfrak{LAM}^*$ by the following inductive definition:

1. $\mathbf{1}_\Lambda \in \mathfrak{SAT}^*$.
2. $$\frac{G \in \mathfrak{SAT}^* \qquad \{X_\gamma \in \mathfrak{SAT}\}_{\gamma \in \overline{G}}}{\Sigma_\Lambda(G, \{X_\gamma\}_{\gamma \in \overline{G}}) \in \mathfrak{SAT}^*}$$

3.5.7. REMARK. This is clearly influenced by the usual definition of saturated sets for non-dependent calculi, which we may phrase as follows:

A set $P \subseteq \Lambda$ is saturated if

1. $P \subseteq \mathrm{SN}$,
2. $\mathsf{Void} \subseteq P$,
3. $\forall_{M \in P} \forall_{M' \in \mathrm{SN}} (M' \rhd_{\mathbf{whd}} M) \to (M' \in P)$

It should be obvious that if $X \in \mathfrak{SAT}$ then the set of realizer $\{M \mid \exists_{x \in \overline{X}} M \Vdash_X x\}$ is saturated in the conventional sense.

$\mathbf{1}_\Lambda$ and Σ_Λ are operations on saturated Λ-sets by definition but it remains to show that this is also true for Π_Λ:

3.5.8. LEMMA. *Assume* $X \in \mathfrak{SAT}$, $\{Y_x \in \mathfrak{SAT}\}_{x \in \overline{X}}$ *then* $\Pi_\Lambda(X, \{Y_x\}_x) \in \mathfrak{SAT}$.

Proof.

SAT1 Assume $M \Vdash_{\Pi_\Lambda(X,\{Y_x\}_x)} f$, certainly $0 \Vdash_X \bot_X$ (**SAT2** for X). Now we know that $M0 \Vdash_{Y_{\bot_X}} f(\bot_X)$, therefore $M0 \in \mathrm{SN}$ (**SAT1** for Y_x), which implies $M \in \mathrm{SN}$.

SAT2 Assume $M \in \mathrm{Void}$, now for every $N \Vdash_X x$ we have that $MN \in \mathrm{Void}$ (**SAT1** for X and definition of Void) and therefore $MN \Vdash_{Y_x} \bot_{Y_x}$. This implies $M \Vdash_{\Pi_\Lambda(X,\{Y_x\}_x)} x \mapsto \bot_{Y_x}$, so we just set $\bot_{\Pi_\Lambda(X,\{Y_x\}_x)} = x \mapsto \bot_{Y_x}$.

SAT3 Assume $M \Vdash_{\Pi_\Lambda(X,\{Y_x\}_x)} f$, $M' \in \mathrm{SN}$ and $M' \rhd_{\mathbf{whd}} M$. For any $N \Vdash_X x$ we have that $MN \Vdash_{Y_x} f(x)$. By (APP-L) $M'N \rhd_{\mathbf{whd}} MN$ and by lemma 3.5.4 (3.) $M'N \in \mathrm{SN}$. Using **SAT3** for Y_x we have that $M'N \Vdash_{Y_x} f(x)$. Therefore we have established that $M' \Vdash_{\Pi_\Lambda(X,\{Y_x\}_x)} f$.

We come now to the central lemma about saturated Λ-sets:

3.5.9. LEMMA. $\mathcal{SAT} = (\mathfrak{SAT}^*, \mathfrak{SAT}, \mathbf{1}_\Lambda, \Sigma_\Lambda, \mathrm{Sect}_\Lambda, \Pi_\Lambda)$ *is an LF-structure.*

Proof. With lemma 3.5.8 we know that all operations are defined on saturated Λ-sets.

We have to verify the conditions for LF-structures (3.2.2). Assume $G \in \mathfrak{SAT}^*, \{X_\gamma, Y_\gamma \in \mathfrak{SAT}\}_{\gamma \in \overline{G}}, \{Z_\delta \in \mathfrak{SAT}\}_{\delta \in \Sigma \overline{G}.\{\overline{X_\gamma}\}_\gamma}$:

1. $\overline{\mathbf{1}_\Lambda} = \{\epsilon\}$ is a singleton set.

2. If $M \Vdash_{\mathrm{Sect}_\Lambda(G,\{Y_\gamma\}_\gamma)} f$ then $M^+ \Vdash_{\mathrm{Sect}_\Lambda(\Sigma_\Lambda(G,\{X_\gamma\}_\gamma),\{Y_{\pi_1(\delta)}\}\delta)} f^{+\{\overline{X_\gamma}\}_\gamma}$.

3. $0 \Vdash_{\mathrm{Sect}_\Lambda(\Sigma_\Lambda(G,\{Y_\gamma\}_\gamma),\{Y_{\pi 1(\delta)}\}_\delta)} \mathrm{pr}_{\overline{G},\{\overline{Y_\gamma}\}_\gamma}$.

4. If $M \Vdash_{\mathrm{Sect}_\Lambda(G,\{\Pi_\Lambda(X_\gamma,\{Z_{(\gamma,x)}\}_x)\}_\gamma)} f$, $N \Vdash_{\mathrm{Sect}_\Lambda(G,\{X_\gamma\}_\gamma)} g$ then

$$MN \Vdash_{\mathrm{Sect}_\Lambda(G,\{Z_{(\gamma,g(\gamma))}\}_\gamma)} f[g].$$

5. Assume $M \Vdash_{\mathrm{Sect}_\Lambda(\Sigma_\Lambda(G,\{X_\gamma\}_\gamma),\{Z_\delta\}_\delta)} f$.

   For any $\gamma \in \overline{G}$ and $\vec{N} \Vdash_G \gamma$ we have that $M[\vec{N}]^1 = M[\vec{N}0] \Vdash_{Z_{\gamma,\perp}} f(\gamma,\perp)$ (using SAT2) and therefore $M[\vec{N}]^1 \in \mathrm{SN}$ (by SAT1).

   Furthermore assume $x \in X_\gamma$ and $N' \Vdash_{X_\gamma} x$. We have that

$$M[\vec{N}]^1[N'] = M[\vec{N}N'] \Vdash_{Z_{(\gamma,x)}} f(\gamma,x)$$

   Knowing $M[\vec{N}]^1, N', M[\vec{N}]^1[N'] \in \mathrm{SN}$ we can apply lemma 3.5.4 (2.) to conclude that $(\lambda M[\vec{N}]^1)N' \in \mathrm{SN}$. We can now apply SAT3 because $(\lambda M)[\vec{N}]N' = \lambda M[\vec{N}]^1)N' \rhd_{\mathbf{whd}} M[\vec{N}]^1[N']$ to see that

$$(\lambda M)[\vec{N}]N \Vdash_{Z_{(\gamma,x)}} f(\gamma,x) = \lambda(f)(\gamma)(x)$$

   Therefore (by discharging the assumptions) we have that

$$\lambda M \Vdash_{\mathrm{Sect}_\lambda(G,\{\Pi_\lambda(X_\gamma,\{Z_{(\gamma,x)}\}_x)\}_\gamma)} \lambda(f).$$

Note that the restriction to saturated Λ-sets was only used for verifying the last condition which corresponds to the rule (LAM). At the same place in lemma 3.4.10 the closure under β-equality was used in an essential way.

To extend $\mathcal{SAT}$ to a CC-structure we exploit the fact that our construction for D-sets was independent from the computational structure:

3.5.10. THEOREM. *By applying definition 3.4.12 for $D = \Lambda$ we obtain $\mathfrak{M}_\Lambda, \mathrm{EL}_\Lambda, \mathrm{EL}_\Lambda^{-1}, \vartheta_\Lambda$. $\mathfrak{M}_\Lambda^{\mathrm{Sat}} = \mathfrak{M}_\Lambda \cap \mathfrak{SAT}$ is the universe of modest saturated Λ-sets. With*

$$\begin{aligned} \overline{\mathrm{SET}_\Lambda} &= \{R \in \mathrm{PER}(\Lambda) \mid \mathrm{EL}(R) \in \mathfrak{SAT}\} \\ \mathrm{SET}_\Lambda &= (\overline{\mathrm{SET}_\Lambda}, \mathrm{SN} \times \overline{\mathrm{SET}_\Lambda}) \end{aligned}$$

we have that

$$\mathcal{SAT}^+ = (\mathcal{SAT}, \mathfrak{M}_\Lambda^{\mathrm{Sat}}, \mathrm{SET}_\Lambda, \mathrm{EL}_\Lambda, \mathrm{EL}_\Lambda^{-1}, \vartheta_\Lambda)$$

is a CC-structure.

Proof. Note that SET_Λ has been defined in a way to ensure that for $R \in \overline{\mathrm{SET}_\Lambda}$ it holds that $\mathrm{EL}(R) \in \mathfrak{SAT}$. To see that $\mathrm{SET}_\Lambda \in \mathfrak{SAT}$ we have to apply lemma 3.5.4(1).

Condition 1 from definition 3.2.5 $\mathrm{Sect}(X, \mathrm{SET}_\Lambda) = \overline{X} \to \overline{\mathrm{SET}_\Lambda}$ holds trivially, just juse $\lambda x.x$ as a realizer. The other conditions follow from lemma 3.4.13.

To conclude strong normalization we have to show that the interpretation of a typable term is realized by its subject:

3.5.11. Lemma (Realizer lemma). *For any $\Gamma \vdash M : \sigma$ we have that*

$$|M| \Vdash_{\mathrm{Sect}_\Lambda([\![\Gamma]\!]^{\mathcal{SAT}^+}, [\![\Gamma \vdash \sigma]\!]^{\mathcal{SAT}^+})} [\![\Gamma \vdash M]\!]^{\mathcal{SAT}^+}$$

Proof. By induction over the derivation, for (VAR),(LAM) and (APP) this is already verified by the choice of realizers in the proof of lemma 3.5.9.

For (ALL) assume that

$$M \Vdash_{\mathrm{Sect}_\Lambda([\![\Gamma.\sigma]\!]^{\mathfrak{SAT}^+}, [\![\mathrm{SET}]\!]^{\mathfrak{SAT}^+})} [\![\Gamma.\sigma \vdash M]\!]^{\mathfrak{SAT}^+}$$

then

$$\forall M \Vdash_{\mathrm{Sect}_\Lambda([\![\Gamma]\!]^{\mathfrak{SAT}^+}, [\![\mathrm{Set}]\!]^{\mathfrak{SAT}^+})} [\![\Gamma \vdash \forall^\sigma M]\!]^{\mathfrak{SAT}^+}$$

because from the premise it follows that $\forall M \in \mathrm{SN}$ and every strongly normalizing terms realizes in SET_Λ.

3.5.12. Corollary (Strong normalization). *If $\Gamma \vdash M : \sigma$ then $|M| \in \mathrm{SN}$.*

Proof. From the properties of saturated Λ-sets it follows that

$$|\Gamma|, |\Gamma| - 1 \ldots 0 \Vdash_{[\![\Gamma]\!]^{\mathfrak{SAT}^+}} \bot, \bot \ldots \bot = \vec{\bot}$$

by lemma 3.5.11 it follows that

$$|M| = |M|[|\Gamma|, |\Gamma| - 1 \ldots 0] \Vdash_{[\![\Gamma \vdash \sigma]\!]^{\mathfrak{SAT}^+} \vec{\bot}} [\![\Gamma \vdash M]\!]^{\mathfrak{SAT}^+} \vec{\bot}$$

and therefore $M \in \mathrm{SN}$ by SAT1.

3.5.13. REMARK. It is natural to compare our approach with the categorical strong normalization proof by Hyland and Ong [HO93]. One obvious difference is that their approach is based on *modified* realizability, whereas the saturated Λ-sets can be viewed as a modification of standard realizability. Another difference is that they identify a structure on strongly normalizing terms which is weaker as PCAs (e.g. conditionally PCAs or C-PCAs), whereas we do not use PCAs at all but a notion of saturated sets.

Chapter 4

Inductive types

The main problem in adding inductive types to a Type Theory seems to be to find a notation which captures all interesting cases, yet is still consistent and comprehensible. There are a number of different proposals, e.g. [CP89], [Dyb91], [Dyb92a] and [PM93b] which are all similar in that they introduce a syntactically convenient notion. In [CP89] also a notion of inductive types based on strictly positive is proposed. This is followed up by Ore, who investigates a notion of inductive types for ECC based on functors [Ore92]. However it seems that this approach is pragmatically less interesting because the presentation requires an additional effort to encode the more convenient presentations. Ore also falls short of capturing families of inductive types which are essential for practical applications. Here we will refrain from proposing and investigating yet another *general* notion of inductive types but concentrate on showing in detail how a non-trivial example of an inductive type (general trees) can be added to the theory and how semantics and strong normalization can be extended. However, in appendix A we present a set of general rules, similar to the ones proposed in [Dyb92a].

Luo and Goguen propose a generalization of ECC called *Unifying Theory of Dependent Types* (UTT) [Luo92],[Gog93]. This calculus is based on a separation of types and proposition; therefore they propose to add inductive types only to the *predicative* levels of ECC and not to Prop which corresponds to Set in our presentation. Our approach is dual: because we want to exploit the propositions-as-types idea, we add inductive types *only* as set-constructors. We will not discuss

predicative universes here but we believe they can be added as inductive types following the proposal in [Dyb92b]. Another difference to UTT is that we emphasize the use of dependent inductive types on the impredicative level which are intentionally not included in UTT.

On a semantical level it seems obvious that inductive types can be modeled by initial T-algebras ala Hagino, [1] this is already sketched in [CP89]. This approach can also by extended to families by considering endofunctors in a slice. Ore in [Ore92] sketches an extension of the ω-set semantics for ECC to his functorial presentation of inductive types. However, this development is unsatisfactory because he never shows that his construction gives an initial T-algebra but he uses this fact in the definition of the semantics. It seems that to verify that his constructions is weakly initial one has to exploit the fact that all definable functors are internal, i.e. their effect on morphisms is tracked by a recursive function. Fu in [Fu92] investigates initial T-algebras in the ω-set semantics as well. However, he restricts himself to non-dependent, algebraic inductive types. Even in this restricted case his development does not seem to be convincing because he needs colimits of infinite chains for *decidable* [2] functors, which usually do not exists in the category of ω-sets. [3]

It should be possible to use the construction presented here to extend the D-set semantics to general T-algebras, where T is internal and preserves monomorphisms. However, in the spirit of the model construction in chapter 2.1.2 we will concentrate here on the concrete model construction and the direct interpretation

---

[1] See [Hag87], or [Alt90] for some examples.

[2] According to definition 5.3.8, p. 71, [Fu92] a decidable functors is one which preserves ω-sets with enumerable domain.

[3] The proof of proposition 5.3.9, p.71, ibid, seems incorrect: It is in general not possible to construct a *finite* Turing machine from *any* enumerable set of Turing machines. Indeed, we do not believe that even the colimit of the ω-chain generated by $T(X) = X+1$ exists, although this functor is clearly decidable and has an initial T-algebra.

of the syntax. Another important consideration is that we are using a construction which easily generalizes to saturated Λ-sets to extend the strong normalization argument.

An important feature of our approach to inductive types is that we allow *large eliminations*, e.g. the elimination constants can use an arbitrary family of types not just sets. This is not only essential to prove inequalities but also seems to extend the power of the theory considerably. [4] It is therefore interesting to note that we are able to interpret large eliminations in the D-set and the saturated Λ-set semantics, thereby extending consistency and strong normalization to this extension.

In the rest of this chapter we present general trees, discuss how they fit into the meta-theory developed in chapter 2.1.2 and then extend D-set and strong normalization.

4.1. Definition of trees

General trees are trees with an arbitrary set of leaves A and the branches indexed by another arbitrary set B. E.g. assume any Γ s.t. $\Gamma \vdash A, B : \mathrm{Set}$ then we define using the material presented in appendix A:

$$\mathrm{Tree} = \mu_{\mathrm{N}}(\{(A, \{\}), (\epsilon, \{B\})\})$$

We have the following derived rules:

$$\Gamma \vdash \mathrm{Tree} : \mathrm{Set} \qquad (\textsc{Tree-Form})$$

[4] E.g. the type-theoretic presentation of negative domain equations [Hof93b], the simple minded consistency arguments as presented in [CD93] and the universe constructions presented in [Dyb92b] essentially rely on this or a comparable mechanism.

$$\Gamma \vdash \mathrm{leaf} = \mathrm{C}^0_{\mathrm{Tree}} : A \to \mathrm{Tree}$$
$$\Gamma \vdash \mathrm{sup} = \mathrm{C}^1_{\mathrm{Tree}} : (B \to \mathrm{Tree}) \to \mathrm{Tree} \qquad \text{(TREE-INTRO)}$$

$$\frac{\Gamma.\mathrm{Tree} \vdash \sigma}{\begin{aligned}\Gamma \vdash \mathrm{R}^\sigma_{\mathrm{Tree}} : \quad & (\Pi a : A.\sigma[\mathrm{leaf}a]) \\ \to \quad & (\Pi f : B \to \mathrm{Tree}.(\Pi b : B.\sigma[fb]) \to \sigma[\mathrm{sup}f]) \\ \to \quad & (\Pi x : \mathrm{Tree}.\sigma[x])\end{aligned}} \qquad \text{(TREE-ELIM)}$$

$$\frac{\begin{array}{l}\Gamma.\mathrm{Tree} \vdash \sigma \\ \Gamma \vdash l : \Pi a : A.\sigma[\mathrm{leaf}a] \\ \Gamma \vdash s : \Pi f : B \to \mathrm{Tree}.(\Pi b : B.\sigma[fb]) \to \sigma[\mathrm{sup}f] \\ \Gamma \vdash a : A \\ \Gamma \vdash f : B \to \mathrm{Tree}\end{array}}{\begin{array}{l}\Gamma \vdash \mathrm{R}^\sigma_{\mathrm{Tree}} ls(\mathrm{leaf}a) \simeq la : \sigma[\mathrm{leaf}a] \\ \Gamma \vdash \mathrm{R}^\sigma_{\mathrm{Tree}} ls(\mathrm{sup}f) \simeq sf(\lambda x : B.\mathrm{R}^\sigma_{\mathrm{Tree}} ls(fb)) : \sigma[\mathrm{sup}f]\end{array}} \qquad \text{(TREE-COMP)}$$

4.2. D-set semantics

NOTATION. Given a $R \in \mathrm{PER}(D)$ we will denote the relation by $\sim_R$ for better readability, and by $\widetilde{R} = D/R = \overline{\mathrm{EL}(R)}$ the associated set of equivalence classes.

Given $R \in \mathrm{PER}(D)$ and $\{S_r \in \mathrm{PER}(D)\}_{r \in \widetilde{R}}$ we denote

$$\forall(R, \{S_r\}_r) = \mathrm{EL}^{-1}(\Pi_{\mathcal{D}}(\widetilde{R}, \{\widetilde{S_r}\}_r)) \in \mathrm{PER}(D)$$

For any $p \in \widetilde{\forall(R, \{S_r\}_r)}$ we have $\widehat{p} = \vartheta^{-1}(p) \in \overline{\Pi_{\mathcal{D}}(\widetilde{R}, \{\widetilde{S_r}\}_r)}$. Note that for any $d \in p$ we have $d \Vdash_{\Pi_{\mathcal{D}}(\widetilde{R}, \{\widetilde{S_r}\}_r)} \widehat{p}$.

In the special case of a constant family we just write $R \to S$.

4.2.1. Formation- and introduction-rules

We will now identify a PER $\mathrm{TREE}_{\mathcal{D}}$ to interpret Tree and give a sound interpretation of leaf and sup. For the following we assume a fixed $\gamma \in [\![\Gamma]\!]^{\mathcal{D}}$ and let $A = [\![\Gamma \vdash A]\!]^{\mathcal{D}}\gamma, B = [\![\Gamma \vdash B]\!]^{\mathcal{D}}\gamma$.

We will use the following codes for the constructors and recursor which are motivated by Parigot's encodings:

$$\begin{aligned} \mathrm{d}_{\mathrm{leaf}} &= \overline{\lambda}a.\overline{\lambda}ls.la \\ \mathrm{d}_{\mathrm{sup}} &= \overline{\lambda}f.\overline{\lambda}ls.sf \\ \mathrm{d}_{\mathrm{R}} &= \overline{\lambda}ls.Y(\overline{\lambda}rt.tl(\overline{\lambda}f.sf(\overline{\lambda}b.r(fb)))) \end{aligned}$$

where $Y = \overline{\lambda}f.(\overline{\lambda}x.f(xx))(\overline{\lambda}x.f(xx))$ or any other encoding of a fixpoint combinator s.t. $Yf \cong f(Yf)$ holds.

The following equations follow directly from the encoding:

4.2.1. Lemma.

$$\begin{aligned} \mathrm{d}_{\mathrm{R}}ls(\mathrm{d}_{\mathrm{leaf}}a) &= la \\ \mathrm{d}_{\mathrm{R}}ls(\mathrm{d}_{\mathrm{sup}}f) &\cong sf(\overline{\lambda}b.\mathrm{d}_{\mathrm{R}}ls(fb)) \end{aligned}$$

Moreover we have the following properties:

4.2.2. Lemma.

1. $\mathrm{d}_{\mathrm{leaf}}a \neq \mathrm{d}_{\mathrm{sup}}f$
2. $\dfrac{\mathrm{d}_{\mathrm{leaf}}a = \mathrm{d}_{\mathrm{leaf}}a'}{a = a'}$
3. $\dfrac{\mathrm{d}_{\mathrm{sup}}f = \mathrm{d}_{\mathrm{sup}}f'}{f = f'}$

Proof.

Ad 1: We have that for any $d, d' \in D$ $(\mathrm{d}_{\mathrm{leaf}}a)(\overline{\lambda}.d)(\overline{\lambda}.d') = d$ and $(\mathrm{d}_{\mathrm{sup}}f)(\overline{\lambda}.d)(\overline{\lambda}.d') = d'$, i.e. $\mathrm{d}_{\mathrm{leaf}}a = \mathrm{d}_{\mathrm{sup}}f$ implies $d = d'$.Therefore the equation cannot hold in any consistent PCA.

Ad 2,3: Note that $(\mathrm{d}_{\mathrm{leaf}}a)II = a$ and $(\mathrm{d}_{\mathrm{sup}}f)II = f$.

Note that we need the fact that the underlying PCA is consistent for the first clause. For the same reason this construction fails for the set-theoretic semantics.

We note that $\mathrm{PER}(D)$ w.r.t. the subset-ordering constitutes a complete lattice:

4.2.3. Lemma.

$$(\mathrm{PER}(D), \subseteq, \bigcap)$$

is a complete lattice.

Proof. Straightforward.

The glb can be defined in a standard way and does not correspond to set-theoretic union, but:

$$\overset{+}{\bigcup} P = \bigcap \{Q \mid \forall_{R \in P} R \subseteq Q\}$$

4.2.4. Remark. If we want to extend the construction to dependent μ-types we need the slightly more general result that for any indexing set I the set $I \to \mathrm{PER}(D)$ constitutes a complete lattice with respect to the pointwise inclusion. This is not hard to see and the rest of the construction works analogously.

We will now apply variant of Tarski's theorem [Tar55] to construct $\mathrm{TREE}_{\mathcal{D}}$.

4.2.5. Definition (Transfinite Iteration). Let $(L, \subseteq, \bigcap, \bigcup)$ be a complete lattice and Φ a monotone function on L, then we define for any ordinal α the transfinite iteration Φ^α as:

$$\begin{aligned} \Phi^0 &= \bigcap \emptyset \\ \Phi^{\alpha+1} &= \Phi(\Phi^\alpha) \\ \Phi^{\bigcup \beta} &= \bigcup \{\Phi^\alpha \mid \alpha \in \beta\} \end{aligned}$$

4.2.6. Theorem (Ordinal fixpoint theorem). *Let L be a complete lattice and Φ a monotone function as before. Then there exists an ordinal α_0 with $|\alpha_0| \leq |L|$ s.t. Φ^{α_0} is the least fixpoint of Φ, and we have:*

- $\forall_\alpha \Phi^\alpha \subseteq \Phi^{\alpha_0}$,
- $\forall_{\alpha \geq \alpha_0} \Phi(\Phi^\alpha) = \Phi^\alpha$,

Proof. See [Llo84], pp. 27.

4.2.7. Remarks.

- There is an obvious dualization of this theorem which is also proven in [Llo84]. This would be certainly useful to model coinductive types, however coinductive types are inherently difficult to handle in an intensional type theory.
- In Tarski's original proof [5] the least fixpoint is constructed as the glb of all pre-fixpoints

  $$\bigcap \{x \mid \Phi(x) \subseteq x\}$$

  If we only want to obtain a fixpoint then this construction is actually simpler then the iterative version. However, we will construct the interpretation of the recursor by another iteration parallel to the one above. It is not clear how to do this for Tarski's construction.

We define an operator $\Phi_{\mathcal{D}} \in \mathcal{P}(D \times D) \to \mathcal{P}(D \times D)$:

$$\Phi_{\mathcal{D}}(R) = \begin{array}{ll} & \{(\mathrm{d}_{\mathrm{leaf}} a, \mathrm{d}_{\mathrm{leaf}} a') \mid a \sim_A a'\} \\ \cup & \{(\mathrm{d}_{\mathrm{sup}} f, \mathrm{d}_{\mathrm{sup}} f') \mid f \sim_{B \to R} f'\} \end{array}$$

with the following properties:

4.2.8. Lemma.

1. *If* $d \sim_{\Phi_{\mathcal{D}}(R)} d'$ *then* **one** *of the following cases applies:*

   (a) $d = \mathrm{d}_{\mathrm{leaf}} a \wedge d' = \mathrm{d}_{\mathrm{leaf}} a' \wedge a \sim_A a'$

---

[5] See [LNS82] for a historical account of this *folk theorem*.

(b) $d = \mathrm{d}_{\mathrm{sup}} f \wedge d' = \mathrm{d}_{\mathrm{sup}} f' \wedge f \sim_{B \to R} f'$

2. $\Phi_{\mathcal{D}}(R \in \mathrm{PER}(D)) \in \mathrm{PER}(D)$

3. $$\frac{R \subseteq R'}{\Phi_{\mathcal{D}}(R) \subseteq \Phi_{\mathcal{D}}(R')}$$

Proof. 1. Just apply lemma 4.2.2.

2. We use the previous clause and the fact that A, $B \to R$ are PERs.

3. We apply 1. and note that $B \to R \subseteq B \to R'$.

We introduce the following abbreviations:

$$\begin{aligned} \mathrm{LEAF}_{\mathcal{D}}{}^{R} &= [\mathrm{d}_{\mathrm{leaf}}]_{A \to \Phi_{\mathcal{D}}(R)} \\ \mathrm{SUP}_{\mathcal{D}}{}^{R} &= [\mathrm{d}_{\mathrm{sup}}]_{(B \to R) \to \Phi_{\mathcal{D}}(R)} \end{aligned}$$

Clause 1. of lemma 4.2.8 implies that $\mathrm{LEAF}_{\mathcal{D}}{}^{R}$ and $\mathrm{SUP}_{\mathcal{D}}{}^{R}$ are disjoint and one-to-one, i.e.:

1. $\forall_{a \in \widetilde{A}} \forall_{f \in \widetilde{B \to R}} \widehat{\mathrm{LEAF}_{\mathcal{D}}}^{R}(a) \neq \widehat{\mathrm{SUP}_{\mathcal{D}}}^{R}(f)$

2. $$\forall_{a,a' \in \widetilde{A}} \frac{\widehat{\mathrm{LEAF}_{\mathcal{D}}}^{R}(a) = \widehat{\mathrm{LEAF}_{\mathcal{D}}}^{R}(a')}{a = a'}$$

3. $$\forall_{f,f' \in \widetilde{B \to R}} \frac{\widehat{\mathrm{SUP}_{\mathcal{D}}}^{R}(f) = \widehat{\mathrm{SUP}_{\mathcal{D}}}^{R}(f')}{f = f'}$$

4.2.9. Lemma. *There is an ordinal α_0 with $|\alpha_0| \leq |D|$ s.t. $\Phi_{\mathcal{D}}^{\alpha_0}$ is the least fixpoint of $\Phi_{\mathcal{D}}$ and we have that for all α: $\widetilde{\Phi_{\mathcal{D}}^{\alpha}} \subseteq \widetilde{\Phi_{\mathcal{D}}^{\alpha_0}}$.*

Proof. Follows from theorem 4.2.6 and lemma 4.2.8 (2,3). We have $\Phi_{\mathcal{D}}^{\alpha} \subseteq \Phi_{\mathcal{D}}^{\alpha_0}$ and by lemma 4.2.8(1.) we get that the inclusion holds on the level of classes.

We can now give a sound interpretation of the formation and introduction rules:

4.2.10. LEMMA. *The following is a sound interpretation of* Tree, leaf *and* sup*:*

$$\begin{aligned}
[\![\Gamma \vdash \mathrm{Tree}]\!]^{\mathcal{D}}\gamma &= \mathrm{TREE}_{\mathcal{D}} \\
&= \Phi_{\mathcal{D}}^{\alpha_0} \\
&\in [\![\Gamma \vdash \mathrm{Set}]\!]^{\mathcal{D}}\gamma \\
[\![\Gamma \vdash \mathrm{leaf}]\!]^{\mathcal{D}}\gamma &= \mathrm{LEAF}_{\mathcal{D}} \\
&= {\mathrm{LEAF}_{\mathcal{D}}}^{\mathrm{TREE}_{\mathcal{D}}} \\
&\in [\![\Gamma \vdash A \to \mathrm{Tree}]\!]^{\mathcal{D}}\gamma \\
[\![\Gamma \vdash \mathrm{sup}]\!]^{\mathcal{D}}\gamma &= \mathrm{SUP}_{\mathcal{D}} \\
&= {\mathrm{SUP}_{\mathcal{D}}}^{\mathrm{TREE}_{\mathcal{D}}} \\
&\in [\![\Gamma \vdash (B \to \mathrm{Tree}) \to \mathrm{Tree}]\!]^{\mathcal{D}}\gamma
\end{aligned}$$

Proof. Obviously we have that $\mathrm{TREE}_{\mathcal{D}} \in \overline{\mathrm{SET}} = \mathrm{PER}(D)$ and for the constructors we just have to exploit that $\Phi_{\mathcal{D}}(\mathrm{TREE}_{\mathcal{D}}) = \mathrm{TREE}_{\mathcal{D}}$ (lemma 4.2.9).

4.2.2. Elimination- and computation-rules

To interpret the recursor assume a family of D-sets $\{X_t\}_{t \in \mathrm{TREE}_{\mathcal{D}}}$ and

$$\begin{aligned}
l &\in \overline{\Pi_{\mathcal{D}}(A, \{X_{\widehat{\mathrm{LEAF}_{\mathcal{D}}(a)}}\}_a)} \\
s &\in \overline{\Pi_{\mathcal{D}}(B \to \mathrm{TREE}_{\mathcal{D}}, \{\Pi_{\mathcal{D}}(B, \{X_{\widehat{f(b)}}\}_b) \to X_{\widehat{\mathrm{SUP}_{\mathcal{D}}(f)}}\}_f)}
\end{aligned}$$

We also choose some realizer $\mathrm{d}_l \Vdash l$ and $\mathrm{d}_s \Vdash s$. First we will construct the set-theoretic part of the recursor by transfinite iteration for every α and then verify by transfinite induction that it is tracked by d_{R}.

We define $\Psi_{\mathcal{D}}(l,s)^\alpha \in \tilde{\Pi} t \in \widetilde{\Phi_{\mathcal{D}}^\alpha}.\overline{X_t}$ as follows:

$$\begin{aligned}
\Psi_{\mathcal{D}}(l,s)^0 &= \emptyset \\
\Psi_{\mathcal{D}}(l,s)^{\alpha+1}(t) &= \begin{cases} l(a) & \text{if } t = {\mathrm{LEAF}_{\mathcal{D}}}^{\Phi^\alpha}(a) \\ s(f, b \in \overline{B} \mapsto \Psi_{\mathcal{D}}(l,s)^\alpha(\widehat{f}(b))) & \text{if } t = {\mathrm{SUP}_{\mathcal{D}}}^{\Phi^\alpha}(f) \end{cases} \\
\Psi_{\mathcal{D}}(l,s)^{\bigcup^\beta}(t) &= \Psi_{\mathcal{D}}(l,s)^\alpha(t) \qquad \text{where } \alpha = \bigcap\{\alpha \in \beta \mid t \in \Phi^\alpha\}
\end{aligned}$$

To see that the X_t in the partial product above is always defined we have to remember that $\widetilde{\Phi^\alpha} \subseteq \widetilde{\mathrm{TREE}_{\mathcal{D}}}$ (lemma 4.2.9).

The case analysis for $\Psi_{\mathcal{D}}(l,s)^{\alpha+1}$ is deterministic and disjoint as a consequence of lemma 4.2.8.

Note that the successor step only defines a partial function because s only allows realizable functions for its second argument. We will now show that $\Psi^{\alpha}_{\mathcal{D}}$ is always total and has a uniform realizer:

4.2.11. LEMMA. *For all α and all $t \in \widetilde{\Phi^{\alpha}_{\mathcal{D}}}, \mathrm{d}_t \in t$ we have:*

1. $\Psi_{\mathcal{D}}(l,s)^{\alpha}(t) \in \overline{X_t}$

2. $\mathrm{d_R d}_l \mathrm{d}_s \mathrm{d}_t \Vdash_{X_t} \Psi_{\mathcal{D}}(l,s)^{\alpha}(t)$

Proof. By transfinite induction over α:

$\alpha = 0$ All clauses are trivial.

$\alpha = \alpha' + 1$

$t = \mathrm{LEAF}_{\mathcal{D}}(a)$

1. We know that $a \in \widetilde{A}$ and therefore: $\Psi_{\mathcal{D}}(l,s)^{\alpha+1}(t) = l(a) \in X_{\mathrm{LEAF}_{\mathcal{D}}(a)}$
2. By definition of $\mathrm{LEAF}_{\mathcal{D}}$: $\mathrm{d}_t = \mathrm{d}_{\mathrm{leaf}}\mathrm{d}_a$ with $\mathrm{d}_a \in a$. Using lemma 4.2.1 we can conclude:

$$\mathrm{d_R d}_l \mathrm{d}_s(\mathrm{d}_{\mathrm{leaf}}\mathrm{d}_a) = \mathrm{d}_l \mathrm{d}_a \Vdash l(a)$$

$t = \mathrm{SUP}_{\mathcal{D}}(f)$

1. We know that $f \in \widetilde{B \to \Phi^{\alpha}_{\mathcal{D}}}$. Assume $d_f \in f$ we can show using the induction hypothesis for 2.:

$$\mathrm{d}_h = \overline{\lambda} b.\mathrm{d_R d}_l \mathrm{d}_s(\mathrm{d}_f b) \Vdash_{\Pi_{\mathcal{D}}(B,\{X_{f(b)}\}_b)} h = b \in \overline{B} \mapsto \Psi_{\mathcal{D}}(l,s)^{\alpha}(\widehat{f}(b))$$

and $h \in \overline{\Pi_{\mathcal{D}}(B,\{X_{f(b)}\}_b)}$. Therefore:

$$\Psi_{\mathcal{D}}(l,s)^{\alpha+1}(t) = s(f,h) \in X_{\mathrm{SUP}_{\mathcal{D}}(f)}$$

2. We use lemma 4.2.1:

$$\begin{aligned} \mathrm{d_R d}_l \mathrm{d}_s(\mathrm{d_{sup}} \mathrm{d}_f) &= \mathrm{d}_s \mathrm{d}_f \mathrm{d}_h \\ &\Vdash s(f,h) \end{aligned}$$

$\alpha = \bigcup \beta$ Follows directly from the induction hypothesis.

We can now conclude that $\Psi_{\mathcal{D}}(l,s)^{\alpha_0}$ is a sound interpretation of the recursor:

4.2.12. Corollary.

1.

$$\begin{aligned} \Psi_{\mathcal{D}}(l,s)^{\alpha_0} &\in \overline{\widetilde{\Pi_{\mathcal{D}}(\mathrm{TREE}_{\mathcal{D}}, \{X_t\}_t)}} \\ \Psi_{\mathcal{D}}(l,s)^{\alpha_0}(\mathrm{LEAF}_{\mathcal{D}}(a)) &= l(a) \\ \Psi_{\mathcal{D}}(l,s)^{\alpha_0}(\mathrm{SUP}_{\mathcal{D}}(f)) &= s(f, b \in \overline{B} \mapsto \Psi_{\mathcal{D}}(l,s)^{\alpha_0}(\widehat{f}(b))) \end{aligned}$$

2.

$$\begin{aligned} [\![\Gamma \vdash \mathrm{R}^{\sigma}_{\mathrm{Tree}}]\!]^{\mathcal{D}}\gamma = & \quad l \in \overline{\Pi_{\mathcal{D}}(A, \{X_{\mathrm{LEAF}_{\mathcal{D}}(a)}\}_a)} \\ & \mapsto s \in \overline{\Pi_{\mathcal{D}}(B \to \mathrm{TREE}_{\mathcal{D}}, \{\Pi_{\mathcal{D}}(B, \{X_{\widehat{f}(b)}\}_b) \to X_{\widehat{\mathrm{SUP}_{\mathcal{D}}(f)}}\}_f)} \\ & \mapsto \Psi^{\alpha_0}_{\mathcal{D}}(l,s) \\ & \textit{with } X = [\![\Gamma.\mathrm{Tree} \vdash \sigma]\!]^{\mathcal{D}}\gamma. \end{aligned}$$

validates (Tree-Elim) *and* (Tree-Comp).

Proof. 1. The first line is a direct consequence of lemma 4.2.11. Note that $\Psi_{\mathcal{D}}(l,s)^{\alpha_0} = \Psi_{\mathcal{D}}(l,s)^{\alpha_0+1}$ and therefore the equations follow directly from the definition of $\Psi_{\mathcal{D}}(l,s)^{\alpha+1}$.

2. This is just a reformulation of 1.

4.2.13. Remark. It is interesting to note that this construction does not depend on the particular type of the recursor or on the choice of equations but only on the fact that the type of the recursor represents a *complete covering* and the right

hand side of the equations are *structurally smaller* then the left hand side — using the terminology of [Coq92b]. Therefore it seems straightforward to extend this semantics to the case of arbitrary pattern matching definitions as described ibid. This remark also applies to the development presented in the next section.

4.3. Saturated Λ-sets

The interpretation for the saturated Λ-set semantics proceeds in essentially the same way as for the D-set semantics. However, we have to extend some syntactic lemmas to trees.

4.3.1. Syntactic properties

We will first extend the notion of reduction, weak-head reduction and void terms in a consistent way. We extend the Curry terms (definition 2.3.5) by the following new constants:

$$\Lambda ::= \ldots \mid \mathrm{Tree} \mid \mathrm{leaf} \mid \mathrm{sup} \mid \mathrm{R}_{\mathrm{Tree}}$$

and the stripping map is extended by $|R^{\sigma}_{\mathrm{Tree}}| = \mathrm{R}_{\mathrm{Tree}}$ and is the identity for the other new cases. Reduction is extended by new β-rules:

$$\begin{array}{lcl} \mathrm{R}_{\mathrm{Tree}} M_l M_s (\mathrm{leaf}\, M_a) & \rhd & M_l M_a \\ \mathrm{R}_{\mathrm{Tree}} M_l M_s (\mathrm{sup} M_f) & \rhd & M_s M_f (\lambda b.\mathrm{R}_{\mathrm{Tree}} M_l M_s (M_f b)) \end{array} \qquad (\textsc{Tree-beta})$$

$\rhd_{\mathbf{whd}}$ includes (Tree-beta) and is closed under

$$\frac{N \rhd_{\mathbf{whd}} N'}{\mathrm{R}_{\mathrm{Tree}} M_l M_s N \rhd_{\mathbf{whd}} \mathrm{R}_{\mathrm{Tree}} M_l M_s N'} \qquad (\textsc{Tree-app})$$

which is valid for $\rhd$ automatically. We extend Void by the following clauses:

$$\frac{N \in \mathrm{Void} \qquad M_l, M_s \in \mathrm{SN}}{\mathrm{R}_{\mathrm{Tree}} M_l M_s N \in \mathrm{Void}}$$

$$\mathrm{Tree} \in \mathrm{Void}$$

By whnf we denote the partial function which assigns to every term its weak head normal form so it exists. Note that we do not need Church Rosser here because $\rhd_{\mathrm{whd}}$ is obviously deterministic.

The fact that we have extended the reduction in a coherent way reflects itself in the fact the we can extend lemma 3.5.2:

4.3.1. Lemma (Extended weak standardization). *lemma 3.5.2 holds for the extended system.*

Proof. The case analysis of lemma 3.5.2 has to be extended by the new cases generated by tree reduction. They are all straightforward.

Using the previous lemma we can show an extended version of lemma 3.5.4 with new clauses for the reduction of trees:

4.3.2. Lemma (Properties of SN).

1. $\mathrm{Void} \subseteq \mathrm{SN}$,
2. $$\frac{M, N, M[N] \in \mathrm{SN}}{(\lambda M)N \in \mathrm{SN}}$$
3. $$\frac{M' \rhd_{\mathrm{whd}} M \qquad MN \in \mathrm{SN}}{M'N \in \mathrm{SN}}$$
4. $$\frac{M_l M_a, M_s \in \mathrm{SN}}{\mathrm{R}_{\mathrm{Tree}} M_l M_s (\mathrm{leaf}\, M_a) \in \mathrm{SN}}$$
5. $$\frac{M_s M_f (\lambda b.\mathrm{R}_{\mathrm{Tree}} M_l M_s (M_f b)) \in \mathrm{SN}}{\mathrm{R}_{\mathrm{Tree}} M_l M_s (\mathrm{sup}\, M_f) \in \mathrm{SN}}$$
6. $$\frac{\mathrm{R}_{\mathrm{Tree}} M_l M_s M_u \in \mathrm{SN}, \qquad M_t \rhd_{\mathrm{whd}} M_u, \qquad M_t \in \mathrm{SN}}{\mathrm{R}_{\mathrm{Tree}} M_l M_s M_t \in \mathrm{SN}}$$

Proof. For the first three cases we only describe the changes to the proof of lemma 3.5.4:

1. We only have to consider the new clause for $\mathrm{R}_{\mathrm{Tree}}$ and observe that void terms are never constructors and therefore for $N \in \mathrm{Void}$ $\mathrm{R}_{\mathrm{Tree}} M_l M_s N$ cannot be a redex.

2. As for lemma 3.5.4.

3. Note that if $M' \rhd_{\mathbf{whd}} M$ then $M'N$ is not a redex. Therefore the argument from lemma 3.5.4 applies without change.

4. By induction over reductions of M_l, M_s, M_a.

5. By induction over reductions of M_l, M_s, M_f, M_a.

6. As in clause 3.: If $M_t \rhd_{\mathbf{whd}} M_u$ then $\mathrm{R}_{\mathrm{Tree}} M_l M_s M_t$ is not a redex. Therefore we have only to consider reductions in M_l, M_s, M_t. If M_t is reduced we need lemma 4.3.1 to apply the induction hypothesis.

4.3.2. Formation and introduction-rules

The definition of the interpretation function in the saturated Λ-set model follows the development of the D-set semantic very closely, i.e. in many places we have just to replace the CC-structure $\mathcal{D}$ by $\mathcal{SAT}$. Remember that

$$\overline{\mathrm{SET}_\Lambda} = \{R \in \mathrm{PER}(\Lambda) \mid \mathrm{EL}(R) \in \mathfrak{SAT}\}$$

plays the role of $\mathrm{PER}(D)$ now. In the following I will rather concentrate on the differences to the previous section.

We define operators $\Phi'_{\mathcal{SAT}}, \Phi_{\mathcal{SAT}} \in \mathcal{P}(\Lambda \times \Lambda) \to \mathcal{P}(\Lambda \times \Lambda)$:

$$\Phi'_{\mathcal{SAT}}(R) = \begin{array}{ll} & \{(\mathrm{leaf}\, M_a, \mathrm{leaf}\, M_{a'}) \mid M_a \sim_A M_{a'}\} \\ \cup & \{(\sup M_f, \sup M_{f'}) \mid M_f \sim_{B\to R} M_{f'}\} \end{array}$$

It is not the case that $\Phi'_{\mathcal{SAT}}(R)$ preserves saturatedness, therefore we define:

$$\Phi_{\mathcal{SAT}}(R) = \begin{array}{ll} & \{(N, N') \mid M, N \in \mathrm{SN} \wedge \mathrm{whnf}(N), \mathrm{whnf}(N') \in \mathrm{Void}\} \\ \cup & \{(M, N) \mid M, N \in \mathrm{SN} \wedge \mathrm{whnf}(M) \sim_{\Phi'_{\mathcal{SAT}}(R)} \mathrm{whnf}(N)\} \end{array}$$

4.3.3. LEMMA (Compare to 4.2.8). *1. If* $M \sim_{\Phi_{\mathcal{SAT}}(R)} M'$ *then* **one** *of the following cases applies:*

(a) $\mathrm{whnf}(M) = \mathrm{leaf}\, M_a \wedge \mathrm{whnf}(M') = \mathrm{leaf}\, M_{a'} \wedge M_a \sim_A M_{a'}$

(b) $\mathrm{whnf}(M) = \mathrm{sup} M_f \wedge \mathrm{whnf}(M') = \mathrm{sup} M_{f'} \wedge M_f \sim_{B\to R} M_{f'}$

(c) $\mathrm{whnf}(M), \mathrm{whnf}(M') \in \mathrm{Void}$

2. $\Phi_{\mathcal{SAT}}(R \in \overline{\mathrm{SET}_\Lambda}) \in \overline{\mathrm{SET}_\Lambda}$

3. $$\frac{R \subseteq R'}{\Phi_{\mathcal{SAT}}(R) \subseteq \Phi_{\mathcal{SAT}}(R')}$$

Proof.

1. Note that all $M \in \mathrm{dom}(\Phi'_{\mathcal{SAT}}(R))$ are non-void weak-head normal forms. Therefore $\Phi_{\mathcal{SAT}}$ does not confuse anything.

2. That $\Phi_{\mathcal{SAT}}(R) \in \mathrm{PER}(\Lambda)$ can be verified analgously 4.2.8. That $\Phi_{\mathcal{SAT}}$ is saturated follows directly from the definition.

3. As for 4.2.8.

We define:

$$\begin{aligned} \mathrm{LEAF}_{\mathcal{SAT}}{}^R &= [\mathrm{leaf}]_{A\to\Phi_{\mathcal{SAT}}(R)} \\ \mathrm{SUP}_{\mathcal{SAT}}{}^R &= [\mathrm{sup}]_{(B\to R)\to\Phi_{\mathcal{SAT}}(R)} \\ \perp^R &= \mathrm{Void} \in \widetilde{\Phi_{\mathcal{SAT}}(R)} \end{aligned}$$

As before $\mathrm{LEAF}_{\mathcal{SAT}}{}^R, \mathrm{SUP}_{\mathcal{SAT}}{}^R, \perp^R$ are disjoint and one-to-one.

4.3.4. LEMMA (Compare 4.2.9). *There is an ordinal* α_0 *with* $|\alpha_0| \leq |D|$ *s.t.* $\Phi^{\alpha_0}_{\mathcal{SAT}}$ *is the least fixpoint of* $\Phi_{\mathcal{SAT}}$ *and we have that for all* α: $\widetilde{\Phi^{\alpha}_{\mathcal{SAT}}} \subseteq \widetilde{\Phi^{\alpha_0}_{\mathcal{SAT}}}$.

Proof. Analogous to 4.2.9.

4.3.5. LEMMA. *The following is a sound interpretation of* Tree, leaf *and* sup*:*

$$\begin{aligned}
[\![\Gamma \vdash \mathrm{Tree}]\!]^{\mathcal{SAT}}\gamma &= \mathrm{TREE}_{\mathcal{SAT}} \\
&= \Phi^{\alpha_0}_{\mathcal{SAT}} \\
&\in [\![\Gamma \vdash \mathrm{Set}]\!]^{\mathcal{SAT}}\gamma \\
[\![\Gamma \vdash \mathrm{leaf}]\!]^{\mathcal{SAT}}\gamma &= \mathrm{LEAF}_{\mathcal{SAT}} \\
&= {\mathrm{LEAF}_{\mathcal{SAT}}}^{\mathrm{TREE}_{\mathcal{SAT}}} \\
&\in [\![\Gamma \vdash A \to \mathrm{Tree}]\!]^{\mathcal{SAT}}\gamma \\
[\![\Gamma \vdash \mathrm{sup}]\!]^{\mathcal{SAT}}\gamma &= \mathrm{SUP}_{\mathcal{SAT}} \\
&= {\mathrm{SUP}_{\mathcal{SAT}}}^{\mathrm{TREE}_{\mathcal{SAT}}} \\
&\in [\![\Gamma \vdash (B \to \mathrm{Tree}) \to \mathrm{Tree}]\!]^{\mathcal{SAT}}\gamma
\end{aligned}$$

Proof. Analogous to 4.2.10.

4.3.3. Elimination- and computation-rules

To interpret the recursor assume a family of saturated Λ-sets: $\{X_t\}_{t\in \mathrm{TREE}_{\mathcal{SAT}}}$ and

$$\begin{aligned}
l &\in \overline{\Pi_{\mathcal{SAT}}(A, \{X_{\widehat{\mathrm{LEAF}_{\mathcal{SAT}}}(a)}\}_a)} \\
s &\in \overline{\Pi_{\mathcal{SAT}}(B \to \mathrm{TREE}_{\mathcal{SAT}}, \{\Pi_{\mathcal{SAT}}(B, \{X_{\widehat{f}(b)}\}_b) \to X_{\widehat{\mathrm{SUP}_{\mathcal{SAT}}}(f)}\}_f)}
\end{aligned}$$

We also choose some $M_l \Vdash l$ and $M_s \Vdash s$.

We define $\Psi_{\mathcal{SAT}}(l,s)^\alpha \in \tilde{\Pi} t \in \widetilde{\Phi^\alpha_{\mathcal{SAT}}}.\overline{X_t}$ as follows:

$$\begin{aligned}
\Psi_{\mathcal{SAT}}(l,s)^0 &= \emptyset \\
\Psi_{\mathcal{SAT}}(l,s)^{\alpha+1}(t) &= \begin{cases} l(a) & \text{if } t = {\mathrm{LEAF}_{\mathcal{SAT}}}^{\Phi^\alpha}(a) \\ s(f, b \in \overline{B} \mapsto \Psi_{\mathcal{SAT}}(l,s)^\alpha(\widehat{f}(b))) & \text{if } t = {\mathrm{SUP}_{\mathcal{SAT}}}^{\Phi^\alpha}(f) \\ \bot_{X_\bot} & \text{if } t = \bot^{\Phi^\alpha_{\mathcal{SAT}}} \end{cases} \\
\Psi_{\mathcal{SAT}}(l,s)^{\bigcup^\beta}(t) &= \Psi_{\mathcal{SAT}}(l,s)^\alpha(t) \qquad \text{where } \alpha = \bigcap\{\alpha \in \beta \mid t \in \Phi^\alpha\}
\end{aligned}$$

4.3.6. LEMMA (Compare to 4.2.11). *For all α and all $t \in \widetilde{\Phi^\alpha_{\mathcal{D}}}$, $M_t \in t$ we have:*

1. $\Psi_{\mathcal{SAT}}(l,s)^\alpha(t) \in \overline{X_t}$

2. $R_{\text{Tree}} M_l M_s M_t \Vdash_{X_t} \Psi_{\mathcal{SAT}}(l,s)^{\alpha}(t)$

Proof. By transfinite induction over α, as before only the successor case is interesting:

$\alpha = \alpha' + 1$

$t = \text{LEAF}_{\mathcal{SAT}}(a)$

1. We know that $a \in \widetilde{A}$ and therefore: $\Psi_{\mathcal{SAT}}(l,s)^{\alpha+1}(t) = l(a) \in X_{\text{LEAF}_{\mathcal{SAT}}(a)}$
2. By definition of $\text{LEAF}_{\mathcal{SAT}}$: $\text{whnf}(M_t) = \text{leaf}\, M_a$ with $M_a \in a$. We reason by the length of the weak-head reduction of M_t
   n=0

$$\text{R}_{\text{Tree}} M_l M_s (\text{leaf}\, M_a) \rhd_{\textbf{whd}} M_l M_a \Vdash_{X_{\text{LEAF}_{\mathcal{SAT}}(a)}} l(a)$$

   by lemma 4.3.2 (4.) and SAT-3 for $X_{\text{LEAF}_{\mathcal{SAT}}(a)}$ we have:

$$\text{R}_{\text{Tree}} M_l M_s (\text{leaf}\, M_a) \Vdash_{X_{\text{LEAF}_{\mathcal{SAT}}(a)}} l(a)$$

   n=n'+1 In a similar fashion apply lemma 4.3.2 (6.) and SAT-3 for $X_{\text{LEAF}_{\mathcal{SAT}}(a)}$.

$t = \text{SUP}_{\mathcal{SAT}}(f)$

1. We know that $f \in B \widetilde{\rightarrow \Phi^{\alpha}_{\mathcal{SAT}}}$. Assume $M_f \in f$ we can show using the induction hypothesis for 2.:

   $M_h = \overline{\lambda} b.\text{R}_{\text{Tree}} M_l M_s (M_f b) \Vdash_{\Pi_{\mathcal{SAT}}(B,\{X_{f(b)}\}_b)} h = b \in \overline{B} \mapsto \Psi_{\mathcal{SAT}}(l,s)^{\alpha}(\widehat{f}(b))$

   and $h \in \overline{\Pi_{\mathcal{SAT}}(B,\{X_{f(b)}\}_b)}$. Therefore:

$$\Psi_{\mathcal{SAT}}(l,s)^{\alpha+1}(t) = s(f,h) \in X_{\text{SUP}_{\mathcal{SAT}}(f)}$$

2. As for the $\text{LEAF}_{\mathcal{SAT}}$ but using lemma 4.3.2 (5.) and (6.) this time.

4.3.7. COROLLARY (Compare to 4.2.12).

1.

$$\begin{aligned}
\Psi_{\mathcal{SAT}}(l,s)^{\alpha_0} &\in \overline{\Pi_{\mathcal{SAT}}(\widetilde{\mathrm{TREE}_{\mathcal{SAT}}}, \{X_t\}_t)} \\
\Psi_{\mathcal{SAT}}(l,s)^{\alpha_0}(\mathrm{LEAF}_{\mathcal{SAT}}(a)) &= l(a) \\
\Psi_{\mathcal{SAT}}(l,s)^{\alpha_0}(\mathrm{SUP}_{\mathcal{SAT}}(f)) &= s(f, b \in \overline{B} \mapsto \Psi_{\mathcal{SAT}}(l,s)^{\alpha_0}(\widehat{f}(b)))
\end{aligned}$$

2.

$$\begin{aligned}
&[\![\Gamma \vdash \mathrm{R}^{\sigma}_{\mathrm{Tree}}]\!]^{\mathcal{SAT}}\gamma = \\
&\quad l \in \overline{\Pi_{\mathcal{SAT}}(A, \{X_{\mathrm{LEAF}_{\mathcal{SAT}}(a)}\}_a)} \\
&\mapsto\ s \in \overline{\Pi_{\mathcal{SAT}}(B \to \mathrm{TREE}_{\mathcal{SAT}}, \{\Pi_{\mathcal{SAT}}(B, \{X_{\widehat{f}(b)}\}_b) \to X_{\widehat{\mathrm{SUP}_{\mathcal{SAT}}(f)}}\}_f)} \\
&\mapsto\ \Psi^{\alpha_0}_{\mathcal{SAT}}(l,s) \\
&\textit{with } X = [\![\Gamma.\mathrm{Tree} \vdash \sigma]\!]^{\mathcal{SAT}}\gamma.
\end{aligned}$$

validates (Tree-Elim) *and* (Tree-Comp).

Proof. As for 4.2.12.

4.3.4. Strong normalization

We can now derive strong normalization by extending the reasoning in section 3.5.

4.3.8. Lemma (Compare 3.5.11). *For any* $\Gamma \vdash M : \sigma$ *we have that*

$$|M| \Vdash_{\mathrm{Sect}_\Lambda([\![\Gamma]\!]^{\mathcal{SAT}^+}, [\![\Gamma \vdash \sigma]\!]^{\mathcal{SAT}^+})} [\![\Gamma \vdash M]\!]^{\mathcal{SAT}^+}$$

Proof. By induction over derivations. For the rules of the core calculus we can reuse the original proof using lemma 4.3.2 instead of 3.5.4. From the construction it should be obvious that:

$$\begin{aligned}
\mathrm{leaf} &\Vdash \mathrm{LEAF}_{\mathcal{SAT}} \\
\mathrm{sup} &\Vdash \mathrm{SUP}_{\mathcal{SAT}} \\
\mathrm{Tree} &\Vdash \mathrm{TREE}_{\mathcal{SAT}} \\
\mathrm{R}_{\mathrm{Tree}} &\Vdash [\![\mathrm{R}^{\sigma}_{\mathrm{Tree}}]\!]^{\mathcal{SAT}}
\end{aligned}$$

4.3.9. Corollary (Strong normalization). *If* $\Gamma \vdash M : \sigma$ *then* $|M| \in \mathrm{SN}$.

Proof. Analogous corollary 3.5.12.

4.3.10. REMARK. The extension of the results from chapter 2.1.2 is straightforward but laborious and will not be carried out here. The possible reductions in the annotations of the recursor make it necessary to extend the blowing-up map, definition 2.3.16.

Chapter 5

Application: Proving SN for System F

In the following we are going to apply the Type Theory developed in the previous chapters, i.e. CC extended by inductive sets [1] , to a non trivial problem: to formally verify the strong normalization result for System F as presented in [GLT89]. This development has been completely formalized in LEGO see the appendix B for a complete and commented listing.

Using Type Theory to verify results about Type Theory or λ-calculus has a twofold purpose: We can test the practicability of the system in an area which is well known to us and therefore can easily be communicated *and* we are able to verify properties of our implementation. This may seem circular at first but note that for the correctness of a type theoretic system implementation details are more essential than abstract meta-theoretic considerations. Another answer to this objection is the possibility of cross-verification. [2] Our choice of strong normalization can be justified with the essential role this problem or more generally the question of decidability plays for the implementation of Type Theories.

In the context of the BRA on logical frameworks there has been some related work, just to mention a few examples:

---

[1] Since this example was developed *before* the material in the previous chapters, there is a slight mismatch: At one point we will use an inductive type in a universe (`VEC`).

[2] This has recently been proposed by Dowek, see [Dow93].

- Berardi partially implemented strong normalization for System F in *pure* CC also using LEGO, [Ber91].

- C. Coquand verified the decidability of equality in the simply typed λ-calculus in ALF [Mag92], an implementation of Martin-Löf's Type Theory with pattern matching, [Coq92a].

- Huet implemented the the residual theory of β-reduction in pure λ-calculus in Gallina/Coq, [Hue93].

- McKinna and Pollack implemented large parts of the meta theory of Pure Type Systems in LEGO, [MP93].

5.1. Using LEGO

LEGO is a proof development system based on Type Theory which has been implemented by Randy Pollack. The main documentation for LEGO is [LP92], a good introduction can be found in [Hof92], where LEGO is used for program verification.

5.1.1. The Type Theory

The standard Type Theory used in the LEGO system is ECC. However we will not use the predicative type universes in any essential way and consider LEGO as an implementation of the calculus described in chapter 2. LEGO uses the Church presentation of terms but it goes even further and allows the suppression of type information if it can be automatically inferred. See 5–1 for a quick overview of the syntax but for more detailed information [LP92] should be consulted. Note that the judgement $\Gamma \vdash \sigma$ is represented in LEGO by a typing judgement and that LEGO does not distinguish between $\forall$ and Π or use El.

An essential element of the LEGO system is that we do not have to come up with a proof term ourselves but that we can generate one by using built-in proof

Our notation	LEGO notation	Remarks
$\lambda x : \sigma.M$	`[x:S]M`	explicit
	`[x|S]M`	implicit
$\Pi x : \sigma.\tau$	`{x:S}T`	explicit
$\mathrm{El}(\forall x : \sigma.T)$	`{x|S}M`	implicit
MN	`M N`	explicit
	`M|N`	implicit
Set	`Prop`	
	`Set`	defined
$(\Gamma) \vdash \sigma$	`S:Type(0)`	

Figure 5–1: Syntax of terms in LEGO

tactics. Therefore most of the constructions in appendix B are not presented as λ-terms directly but as the sequence of tactics by which they are generated. For details we refer again to [LP92].

5.1.2. The Logic

The basic logical connectives (`/\`,`\/`,`not` and `Ex`) are defined using an impredicative encoding see figure 5–2. However, instead of Leibniz Equality we define propositional equality `EQ` as an inductive type.

Theoretically, it would have been better to use inductive types for all logical connectives, because they come with stronger elimination rules and it seems a bit of a waste to introduce impredicativity just to encode basic logical connectives. However, the current `Refine` tactic of LEGO is tuned for the impredicative encodings.

It is also an essential disadvantage that we only have the weak eliminations when we want to use the proposition as types paradigm. Given that the proof on

Usual notation	**Lego notation**	**Definition**
$A \wedge B$	`A /\ B` *or* `and A B`	`{C|Prop}(A->B->C) -> C`
$A \vee B$	`A \/ B` *or* `or A B`	`{C|Prop}(A->C) -> (B->C) -> C`
$A \Rightarrow B$ $A \Leftrightarrow B$	`A -> B` `(iff A B)`	`{_:A}B` `(and (A->B) (B->A))`
$\bot$ (falsity) $\neg A$	`absurd` `not A`	`{C:Prop} C` `A -> absurd`
$\forall x \in A.P(x)$	`{x:A} (P x)` *or* `{x|A} (P x)`	*primitive*
$\exists x \in A.P(x)$	`Ex[a:A]P a` *or* `Ex P`	`{C|Prop}({a:A}(P a) -> C) -> C`
$x = y \in X$	`(EQ x y)`	*inductive type*

Figure 5–2: Logical connectives in Lego

which this development is based is not type-theoretic this turns out to be a minor problem, but see section 5.3.1.

5.1.3. Inductive types

When this work was done (August 1992) there was no mechanism in LEGO to define inductive types. [3] Therefore we introduce inductive types just by *assuming* the constants used and by adding the typed reduction rules to the system. Consider, as an example, the type of natural numbers (see appendix B.3. Using the notation defined in A we would define

$$\begin{aligned} \mathrm{Nat} &= \mu_{\mathrm{N}}(\{(A_0 = \epsilon, \{\}), (A_1 = \epsilon, \{B_{10} = \epsilon\})\}) \\ &: \mathrm{Set} \end{aligned}$$

This specification can be written in a more readable way by directly presenting the types of the canonical elements and at the same time introducing names for set former and canonical elements:

```
mu[Nat:Set](zero:Nat,succ:Nat->Nat)
```

This is translated into the following LEGO declarations:

```
$[Nat : Set]
$[zero : Nat]
$[succ : Nat->Nat]}

$[RecNat : {P:Nat->Type(0)}
           (P zero)
         ->({n:Nat}(P n)->(P (succ n)))
         -> {n:Nat}P n];
```

[3] This situation has changed now, due to the work by Claire Jones [Jon93].

```
[[P:Nat->Type(0)][z:P zero][f:{n:Nat}(P n)->(P (succ n))]
 [n:Nat]
    RecNat P z f zero ==> z
 || RecNat P z f (succ n) ==> f n (RecNat P z f n)];
```

Note that we simulate *large eliminations* by quantifying over `Type(0)` instead of `Prop`.

If we want to define a function over natural numbers, we may declare it first in an ML-like fashion:

```
add : Nat->Nat->Nat
rec add zero n = n
  | add (succ m) n = succ (add m n)
```

which can be (mechanically) translated into the following LEGO code using the recursor (here we use a derived non-dependent recursor `RecNatN`[4] to simplify the typing):

```
[RecNatN[C|Type(0)] = RecNat ([_:Nat]C)];
```

```
[add = RecNatN ([n:Nat]n)
               ([m:Nat][add_m:Nat->Nat][n:Nat]succ (add_m n))];
```

We can only define functions by primitive recursion in this way, but note that we get more than the usual primitive recursive functions because we have higher order functions.

It is interesting to consider inductive types with dependent constructors like the type of vectors:

---

[4]We adopt the convention that *R*`N` stands for the non-dependent version of recursor *R*.

```
[A:Set]mu[Vec:Nat->Set](v_nil:Vec zero,
                        v_cons:A->{n|Nat}(Vec n)->(Vec succ n))
```

or the family of finite sets:

```
mu[Fin:Nat->Set](f_zero:{n:Nat}Fin (succ n),
                 f_succ:{n|Nat}(Fin n)->(Fin (succ n)))
```

Vectors resemble lists but differ in that the length of the sequence is part of its type. Therefore we have `Vec: Set->Nat->Set` in contrast to `List : Set->Set`, i.e. `Vec A 3`[5] is the type of sequences of type `A` of length `3`. Finite sets are a representation of subsets of natural numbers less than a certain number, i.e. `Fin` n corresponds to $\{i \mid i < n\}$.

As already remarked we need the *large eliminations* to show inequalities, like the fourth Peano axiom: $\forall n.0 \neq n + 1$. When using dependent inductive types the proofs of these proposition also have a computational usage. An example is a *run-time-error-free* lookup function for vectors:[6]

```
v_nth : {n|Nat}(Fin n)->(Vec A n)->A

rec v_nth|(succ n) (f_zero n) (v_cons a _) = a
  | v_nth|(succ n) (f_succ i) (v_cons _ l) = v_nth|n i l
```

Using these error-free functions not only simplifies the verification of functions using vectors; it also allows, in principle a more efficient compilation of code involving dependent types.[7]

---

[5]The official LEGO syntax for this is `Vec A (succ (succ (succ zero)))`.

[6]I.e. `v_nth (f_succ (f_zero 3)) : (Vec A 5)->A` extracts the second element out of a sequence of five.

[7]The idea of using dependent types to avoid run-time-errors was first proposed by Healfdene Goguen to me.

Another use of dependent inductive types is the definition of predicates as the initial semantics of a set of Horn clauses. An example is the definition of the predicate $\leq$ (`LE`) for natural numbers:

```
mu[LE:Nat->Nat->Set](
   le0:{n:Nat}LE zero n,
   le1:{m,n|Nat}(LE m n)->(LE (succ m) (succ n)))
```

5.2. A guided tour through the formal proof

In the following I am going to explain the formalized proof. For more detailed information it may be worthwhile to study the LEGO code or better to run the proofs through the system.

5.2.1. The untyped λ-calculus

We define untyped λ-terms (`Tm`) using de Bruijn indices [dB72] as the following inductive type:

```
mu[Tm:Set](var : Nat->Tm,
           app : Tm->Tm->Tm,
           lam : Tm->Tm)
```

We define the operations weakening `weak : Nat->Tm->Tm` (corresponding to M^{+n}) (introduction of dummy variables) and substitution `subst : Nat->Tm->Tm->Tm` (corresponding to $M[N]^n$)by primitive recursion over the structure of terms using the appropriate recursor (see appendix B.3). The first parameter indicates the number of bound variables — `weak0` and `subst0` are defined as abbreviations, i.e. `subst0 M N` substitutes the free variable with index 0 in `M` by `N`.[8] We also define

[8] Although we only use `weak0` and `subst0` in the following definition we really have to *export* the general versions because we have to use them whenever we want to

repeated weakening `rep_weak0:Nat->Tm->Tm` which is needed to simulate parallel substitution.

In the course of the proof we need a number of facts about weakening and substitution:

```
Goal {l:Nat}{M,N:Tm}EQ (subst l (weak l M) N) M;

Goal {l',l:Nat}{M:Tm}(LE l' l)->
        (EQ (weak (succ l) (weak l' M))
            (weak l' (weak l M)));

Goal {l',l:Nat}{M,N:Tm}(LE l' l)->
        (EQ (subst (succ l) (weak l' M) N)
            (weak l' (subst l M N)));

Goal {l:Nat}{M,N1,N2:Tm}
        EQ (subst l (subst (succ l) M N1) N2)
           (subst l (subst l M (weak0 N2)) N1);

Goal {m,l:Nat}{M1,M2,N:Tm}
        EQ (subst m (subst (succ (add m l)) M1 N) (subst l M2 N))
           (subst (add m l) (subst m M1 M2) N);
```

It is interesting to note that those properties are often ignored in informal reasoning but they require quite an effort when formalizing it. However, we should not forget that technical errors, like omitting bound variables, can easily happen and make the whole reasoning unsound.

---

prove anything about substitution or weakening in general (i.e. for terms containing λ-abstractions).

Another observation is that it pays off that we use de Bruijn indices from now on because our reasoning will be essentially algebraic using the laws above. We never have to carry around and manipulate side conditions about free variables.

We define the one-step reduction relation[9] by the following inductive type:

```
mu[Step:Tm->Tm->Set](
      beta : {M,N:Tm}Step (app (lam M) N) (subst0 M N),
      app_l : {M,M',N:Tm}(Step M M')->(Step (app M N) (app M' N)),
      app_r : {M,M',N:Tm}(Step M M')->(Step (app N M) (app N M')),
      xi : {M,N:Tm}(Step M M')->(Step (lam M) (lam M')) )
```

This amounts to translating the usual Horn clauses defining the reduction relation into the constructors for an inductive type.

5.2.2. Strong Normalization

One of the main technical contributions which simplify the formalization of the proof is the definition of the predicate *strongly normalizing* by the following inductive type:[10]

```
mu[SN:Tm->Set](
    SNi : {M:Tm}({N:Tm}(Step M N)->(SN N))->(SN M))
```

In other words: we define `SN` as the set of elements for which `Step` is well founded.

More intuitively: `SN` holds for all normal forms because for them the premise of `SNi` is vacuously true. Now we can also show that all terms which reduce in

---

[9] We are going to define `Red` (the reflexive, transitive closure of `Step`) later (section 5.3.1). Note, however, that we never need it for the strong normalization proof.

[10] It is interesting to note that this inductive type is not algebraic or equivalently does not correspond to a specification by a set of Horn formulas. Compare this to the predicate `Acc` as defined in [Dyb92a], section 5.2.2.

one step to a normal form are SN and so on for an arbitrary number of steps. On the other hand these are all the terms for which SN holds because SN is defined inductively.

We will use the non-dependent version of the recursor [11]

```
RecSNN : {P:Tm->Type}
    ({M:Tm}({N:Tm}(Step M N)->SN N)->({N:Tm}(Step M N)->P N)->P M)
    ->{M|Tm}(SN M)->P M];
```

to simulate induction over the length of the longest reduction of a strongly normalizing term — in terms of [GLT89] this is induction over $\nu(M)$. Observe that we never have to formalize the concept of the length of a reduction or to define the partial function ν. [12] It is also interesting that the important property that SN is closed under reduction shows up as the destructor for this type (SNd).

5.2.3. System F

The type expressions of System F have essentially the same structure as untyped λ-terms. However, in contrast to the definition of Tm we will use a dependent type here, which makes the number of free variables explicit. This turns out to be useful when we define the semantic interpretation of types later.[13]

```
mu[Ty:Nat->Set](t_var : {n|Nat}(Fin n)->(Ty n),
                arr   : {n|Nat}(Ty n)->(Ty n)->(Ty n),
                pi    : {n|Nat}(Ty (succ n))->(Ty n) )
```

---

[11] This corresponds to the principle of Noetherian Induction [Hue80].

[12] Note that *bounded* and *noetherian* coincide for β-reduction because it is finitely branching (König's lemma).

[13] We could have used a dependent type for Tm as well, but we never need to reason about the number of free variables of an untyped term.

`Ty` i represents type expressions with i free variables.

When defining weakening and substitution for `Ty` we observe that the types actually tell us how these operations behave on free variables:

```
t_weak : {l:Nat}(Ty (add l n))->(Ty (succ (add l n)))
t_subst : {l:Nat}(Ty (add (succ l) n))->(Ty n)->(Ty (add l n))
```

Although these functions are operationally equivalent to `weak` and `subst` we have to put in more effort to implement them. We do this by deriving some special recursors from the standard recursor.[14]

We now define contexts and derivations as:

```
[Con[m:Nat] = Vec (Ty m)];

mu[Der:{m,n|Nat}(Con m n)->Tm->(Ty m)->Set](
        Var : {m,n|Nat}{G:Con m n}{i:Fin n}
                Der G (var (Fin2Nat i)) (v_nth i G)
        App : {m,n|Nat}{G|Con m n}{s,t|Ty m}{M,N|Tm}
                (Der G M (arr s t))
                -> (Der G N s)
                -> (Der G (app M N) t)
        Lam : {m,n|Nat}{G|Con m n}{s,t|Ty m}{M|Tm}
                (Der (v_cons s G) M t)
                -> (Der G (lam M) (arr s t))
        Pi_e: {m,n|Nat}{G|Con m n}{s:Ty (succ m)}{t:Ty m}{M|Tm}
                (Der G M (pi s))
                -> (Der G M (t_subst0 s t))
        Pi_i: {m,n|Nat}{G|Con m n}{s:Ty (succ m)}{M|Tm}
                (Der (v_map t_weak0 G) M s)
```

---

[14] It seems that we could save a lot of effort here by using Thierry Coquand's idea of considering definitions by pattern matching as primitive [Coq92b].

```
-> (Der G M (pi s)) )
```

In the rule `Var` we use `Fin` because this rule is only applicable to integers smaller than the length of the context. Here we have to coerce it to a natural number first (`Fin2Nat`) because `var` requires `Nat` as an argument.

`v_map t_weak0 G` means that all the types in `G` are weakened — this is equivalent to the usual side condition in the standard definition of Π-introduction. It is nice to observe how well the types of `t_subst0` and `t_weak0` fit for the definition of the rules.

5.2.4. Candidates

One of the essential insights about strong normalization proofs is that they require another form of induction than proofs of other properties of typed λ-calculi like the *subject reduction property* or the *Church-Rosser property*. We cannot show strong normalization just by induction over type derivations or by induction over the length of a reduction. They correspond to a model construction, i.e. they are essentially semantical.

The idea of *Candidates of Reducibility* can be summarized as follows:

1. Every Candidate only contains strongly normalizing terms.

2. For every operation on types we can define a semantic operation on sets of terms such which is closed under candidates. Another way to express this is to say that the Candidates constitute a sound interpretation.

3. Every term which has a type is also in the semantic interpretation of the type. [15]

Putting these things together we will obtain that every typable term is strongly normalizing.

---

[15] This corresponds to `Int_sound` in appendix B.9, which is actually a misnomer.

In the definition of *Candidates of Reducibility* `CR:(Tm->Set)->Set` we follow [GLT89]:[16]

```
[neutr[M:Tm] = {M':Tm}not (EQ (lam M') M)];

[P:Tm->Set]
[CR1 = {M|Tm}(P M)->(SN M)]
[CR2 = {M|Tm}(P M)->{N:Tm}(Step M N)->(P N)]
[CR3 = {M|Tm}(neutr M)->
         ({N:Tm}(Step M N)->(P N))->(P M)]
[CR = CR1 /\ CR2 /\ CR3];
Discharge P;
```

We define `neutr` as the set of terms which are not generated by the constructor for the arrow type — `lam`.[17] `CR1` places an upper bound on candidates: they may only contain strongly normalizing terms. `CR2` says that candidates have to be closed under reduction and `CR3` is essentially `SNi` restricted to neutral terms.

The essence of this definition lies in the possibility of proving the following lemmas:

`CR_var` *Candidates contain all variables*

```
{P:Tm->Set}(CR P)->{i:Nat}P (var i);
```

We need this not only for the following lemmas, but also for the final corollary when we want to deduce strong normalization from soundness for non-empty contexts.

---

[16] `[P:Tm->Set] ... Discharge P;` means that `P` is λ-abstracted from all definitions in between.

[17] If we generalize this to systems with inductive types we have to include their *constructors* as well.

This is a trivial consequence of CR3 because variables are neutral terms in normal form.

CR_SN *There is a candidate set*

```
CR SN
```

The choice is arbitrary but the simplest seems to be SN. The proof is trivial: just apply SNd for CR2 and SNi for CR3.

CR_ARR *Candidates are closed under the semantic interpretation of arrow types.*

```
{P,R:Tm->Set}(CR P)->(CR R)->(CR (ARR P R))
```

where

```
[ARR[P,R:Tm->Set] = [M:Tm]{N:Tm}(P N)->(R (app M N))];
```

The proof of CR3 for ARR P R is actually quite hard and requires an induction using RecSNN which corresponds to the reasoning using $\nu(N)$ in [GLT89].

CR_PI *Candidates are closed under the interpretation of* Π*-types*

```
{F:(Tm->Set)->(Tm->Set)}
       ({P:Tm->Set}(CR P)->(CR (F P)))
       -> (CR (PI F));
```

where

```
[PI[F:(Tm->Set)->(Tm->Set)] =
       [M:Tm]{P:Tm->Set}(CR P)->(F P M)];
```

At this point we really need impredicativity for the proof. However, it is interesting to observe how simple this lemma is technically: we do not apply any induction — we just have to show that CR is closed under arbitrary non-empty intersections.

Lam_Sound *The rule of arrow introduction (*Lam*) is semantically sound for candidate sets.*

```
{P,R:Tm->Set}(CR P)->(CR R)->
    {M:Tm}({N:Tm}(P N)->(R (subst0 M N)))
    ->(ARR P R (lam M));
```

Observe that we could not have proved this lemma for arbitrary subsets of SN. The proof requires a nested induction using RecSNN which corresponds to an induction over $\nu(M) + \nu(N)$.

5.2.5. Proving strong normalization

We now have all the ingredients for the proof, we just have to put them together.

We proceed by defining an interpretation function. Types are interpreted by functions from sequences of sets of terms to sets of terms, the length of the sequence depending on the number of free type variables:[18]

```
Int : {m|Nat}(Ty m)->(VEC (Tm->Set) m)->(Tm->Set)

rec Int|m (t_var i) = [v:VEC (Tm->Set) m]V_nth i v
  | Int|m (arr s t) = [v:VEC (Tm->Set) m]ARR (Int s v) (Int t v)
  | Int|m (pi t)    = [v:VEC (Tm->Set) m]
                            PI ([P:Tm->Set]Int t (V_cons P v))
```

We can show by a simple induction that every interpretation of a type preserves candidates (CR_Int) by exploiting CR_ARR and CR_PI .

---

[18] We have to use another type of vectors (large vectors) VEC:Nat->Type(0)->Type(0) instead of Vec:Nat->Set->Set. Unfortunately, this sort of polymorphism cannot be expressed in the current implementation of LEGO — i.e. we have to duplicate the definitions.

We extend this to an interpretation of judgements, i.e. pairs of types and contexts (`Mod`). The idea is that `Mod G M T v` holds iff by substituting all the variables in `M` by terms of the corresponding interpretation of the types in `G` we end up with an element of `Int T v`:[19]

```
Mod : {m,n|Nat}(Con m n)->Tm->(Ty m)->(VEC (Tm->Set) m)->Set

rec Mod m zero     empty        M T v = Int T v M
  | Mod m (succ n) (v_cons S G) M T v =
        {N:Tm}(Int S v N)->(Mod G (subst0 M (rep_weak0 N n)) T v)
```

We use `Mod` to state soundness (`Int_Sound`), i.e. that `Der G M T` implies `Mod G M T v` if all free type variables are interpreted by candidates:

```
{m,n|Nat}{G|Con m n}{M|Tm}{T|Ty m}(Der G M T)->
        {v:VEC (Tm->Set) m}({i:Fin m}CR (V_nth i v))
        -> (Mod G M T v);
```

The proof of soundness proceeds by induction over derivations. Essentially we only have to apply `Lam_sound` to show that the rule `Lam` is sound. The soundness of application `App` follows directly from the definition of `ARR`. To verify soundness for the rules which are particular to System F we do not need additional properties of `CR` but we have to verify that `t_weak` and `t_subst` are interpreted correctly with respect to `Int`. Again these intuitively simple lemmas are quite hard to show formally.

To conclude strong normalization:

```
{m,n|Nat}{G|Con m n}{M|Tm}{T|Ty m}(Der G M T)->(SN M)
```

---

[19] `rep_weak0` is the iterated application of `weak0`. It is necessary to apply weakening here because we do not get parallel substitution by a repeated application of `subst0`.

we have to put `Int_sound` and `CR_Int` together to show that every term is in the interpretation of a candidate; and by definition candidates only contain strongly normalizing terms. There are two technical complications: to show the theorem for terms with free term variables we exploit `CR_var`; to show it for a derivation with free type variables we have to supply a candidate — here we use `CR_SN`. Note that the choice is arbitrary but that it is essential that `CR` is not empty.

5.3. Alternatives and extensions

In the following we will discuss some possible variations on the implemented proofs which have not been completely formalized.

5.3.1. Extracting a normalization function

The proof not only tells us that every typable term is strongly normalizing but it is also possible to derive a function which computes the normal form. This seems to be a case where it is actually more straightforward to give a proof that every strongly normalizable term has a normal form than to program it directly.

To specify normalization we need a notion of reduction (`Red`) — which is just the transitive reflexive closure of `step`:

```
mu[Red:Tm->Tm->Set](
     r_refl : {M:Tm}(Red M M),
     step : {M1,M2,M3|Tm}(Step M1 M2)->(Red M2 M3)->(Red M1 M3) )
```

and we define the predicate *normal form*:

```
[nf[M:Tm] = {M':Tm}not (Step M M')]
```

Now we want to show `norm_lem`:

```
{M:Tm}(SN M)->Ex[M':Tm](Red M M')/\(nf M')
```

which can be done using RecSNN — it turns out that we need decidability of normal form as a lemma:

```
{M:Tm}(nf M)\/(Ex[M':Tm]Step M M')
```

Actually, this is even stronger, because it also gives us a choice of a reduct for terms not in normal form (this is the point where we specify the strategy of reduction).

If we use the strong sum to implement Ex, instead of the weak impredicative encoding, we can use norm_lem to derive:

```
norm    : {M:Tm}(SN M)->Tm
norm_ok : {M:Tm}{p:SN M}(Red M (norm M p)) /\ (nf (norm M p))
```

Note that at this point we exploit the propositions as types principle. Here it is a serious limitation to only have weak eliminations for logical connectives.

5.3.2. Saturated Sets

In many strong normalization proofs the notion of *Saturated Sets* is used instead of *Candidates of Reducibility* — in particular the definition of saturated λ-sets is clearly motivated by them. It is relatively easy to change the proof to use saturated sets: all we have to do is to replace CR by SAT and prove that it has the same properties as CR.

To formalize saturated set we first have to define weak head reduction and void terms, which are both inductively defined:

```
mu[W_Hd_Step:Tm->Tm->Set](
        wh_beta : {M,N:Tm}W_Hd_Step (app (lam M) N) (subst0 M N),
        wh_app_l : {M,M',N:Tm}(W_Hd_Step M M')
                        ->(W_Hd_Step (app M N) (app M' N)) )

mu[Void:Tm->Set](
        v_var: {i:Nat}Void (var i),
        v_app: {M,N|Tm}(Void M)->(SN N)->(Void (app M N)));
```

Now we can define SAT analogously to CR as:

```
[P:Tm->Set]
[SAT1 = {M|Tm}(P M)->(SN M)]
[SAT2 = {M|Tm}(Void M)->(P M)]
[SAT3 = {M,M'|Tm}(P M)->(W_Hd_Step M' M)
                ->(SN M')->(P M')];
[SAT = SAT1 /\ SAT2];
Discharge P;
```

Luo shows that CR P implies SAT P ([Luo90], page 95) and remarks that the converse does not hold because saturated sets do not have to be closed under reduction. An example is the set of all strongly normalizing terms whose weak head normal form is neutral or equal to $\lambda x.II$, which is saturated but not closed under reduction.

If we want to show CR_ARR and Lam_sound formally we have to use RecSNN in a manner similar to the original proof. The complexity of the proof seems to be roughly the same, although we have to define the transitive closure and show an additional lemma about reduction and substitution lemma 2.3.8(3) to show lemma 3.5.2. On the other hand the advantage of saturated sets is that it seems easier to generalize them to dependent types (as we have done in the definition of saturated Λ-sets).

5.3.3. Reduction for Church terms

We have only done the proofs for the Curry style systems — so one obvious question is how hard it would be to extend this proof to the Church style presentation, i.e. to terms with explicit type information. In the case of simply typed λ-calculus this is straightforward because every reduction on a typed term corresponds to one on its untyped counterpart and vice versa. However, this reasoning does not generalize to System F because here we have additional (second order) reductions on typed terms.

This problem is usually solved by arguing that the second order reductions are terminating anyway. Another way, maybe more amenable to formalization, would be to extend the notion of untyped terms and reduction:

```
mu[Tm:Set](...,
           T_Lam : Tm->Tm,
           T_App : Tm->Tm)

mu[Step:Tm->Tm->Set](...,
        Beta : {M:Tm}(Step (T_App (T_Lam M)) M) )
```

Note that `T_Lam` does not actually bind any term-variables but corresponds to second order abstraction for typed terms; analogously `T_App` is used as a dummy type application where the type is omitted.[20]

It does not seem hard to extend the proof to this notion of untyped terms. We have to extend the notion of neutrality, and the soundness of `Pi_i`, which was trivial so far, has to be proved as an additional property of `CR`. The result for Church terms now follows by observing that for the extended notion of untyped terms reductions coincide with the typed terms.

---

[20] It may just be a curiosity, but this version of untyped terms corresponds to (a special case of) partial terms. In [Pfe92] it is shown that type inference for partial terms is undecidable, which is still open for the usual notion of untyped terms.

Chapter 6

Conclusions and further work

Let us summarize the main points of this work by two slogans:

Proving properties of syntax by using semantics I hope that I have demonstrated this by deriving an extensible strong normalization proof from the realizability semantics of CC. Here we emphasize the use of semantics as a tool to analyze a system than as a vehicle of understanding. However, by doing this we challenge the view that systems can or should be reduced to their syntactical presentation.

Type Theory is useful for verification We have given some support for this slogan this by implementing the proof for the strong normalization of System F in LEGO. It should be noted that the structure of this proof was still very much influenced by the conventional formulation in predicate logic. In [CD93] it is shown how type-theoretic systems can be analyzed in a *type-theoretic* way.

I will close by mentioning some areas which deserve further attention:

1. In this thesis we followed the traditional approach to show strong normalization and to derive further metatheoretic properties (like uniqueness of product formation (theorem 2.4.1)) from this. The limits of this method show up already in the context of η-reduction (2.3.19). An alternative is

to directly give a procedure which assigns to every term a canonical normal form and show its correctness. This approach has been investigated in [Coq90],[BS91] and [CD93]. However, to our knowledge, it has not yet been extended to CC or to general inductive types.

2. The recent work by Dybjer [Dyb91,Dyb92a,Dyb92b] and T. Coquand [Coq92b] suggest a general formalism to present inductive definitions, structural recursion and proofs by induction. This formalism should include the sort of mutual definitions which are required for the definitions of universes and for the presentation of Type Theory in Type Theory. The (categorical) semantics and the metatheory (i.e. decidability) of this formalism needs to be worked out in some detail.

3. The inherent limitations of the intensional presentation of Type Theory often lead to complications: the axiom of extensionality is not provable and there is no notion of quotients in intensional Type Theory. A related problem is the computational use of co-induction when presenting co-inductive types. We believe that it is possible to overcome these problems and present a Type Theory which is essentially extensional but still regaining decidability. Recent progress has been made in this area due to the work of Hofmann [Hof93a].

4. The use of Type Theory as an integrated verification and programming-language should be investigated and supported by implementations. We believe that Type Theory should be directly used as a programming language instead of encoding other languages into it. Only this way the additional information available at compile time can be used most effectively.

Appendix A

General μ-types

We present an extension of the core calculus by a general notion of inductive types, essentially following [Dyb91]. Note that we leave out several possible generalizations such as mutual inductive definitions (e.g. see [Dyb92a]), or mutual inductive, recursive definitions ([Dyb92b]) or T. Coquand's generalization of elimination rules by pattern matching [Coq92b].

One essential difference to Dybjer's presentation is that we present μ-types in a closed theory, i.e. we introduce general rules which can be instantiated to particular cases. Another difference is that although one can consider our formation and introduction rules as *monomorphic* we introduce *polymorphic* elimination constants to allow for large eliminations without having to introduce another universe.

Note that we present the following rules using Church syntax. However, the explicit versions can be easily recovered by using the type-reconstruction algorithm (see section 2.4).

A.1. Telescopes

In order to present the general rules we need some notation regarding sequences of Sets.

$$\Gamma \vdash \vec{A} : \vec{\mathrm{Set}}$$

is a shorthand for:

$$\begin{aligned}
\Gamma &\vdash A_0 : \mathrm{Set} \\
\Gamma.\mathrm{El}(A_0) &\vdash A_1 : \mathrm{Set} \\
&\vdots \\
\Gamma.\mathrm{El}(A_0).\mathrm{EL}(A_1)\ldots\mathrm{El}(A_{|\vec{A}|-2}) &\vdash A_{|\vec{A}|-1} : \mathrm{Set}
\end{aligned}$$

Substitution and weakening can be easily extended to telescopes:

$$\begin{aligned}
\vec{A}^{+}n &= \{A_i^{+(n+i)}\}_{i\in|\vec{A}|} \\
\vec{A}[N]^n &= \{A_i[N]^{(n+i)}\}_{i\in|\vec{A}|}
\end{aligned}$$

We will also use $\vec{\mathrm{El}}(\vec{A}) = \mathrm{EL}(A_0).\mathrm{El}(A_1)\ldots\mathrm{El}(A_{|\vec{A}|-1})$. I.e.

$$\Gamma.\vec{\mathrm{El}}(\vec{A}) = \Gamma.\mathrm{EL}(A_0).\mathrm{El}(A_1)\ldots\mathrm{El}(A_{|\vec{A}|-1})$$

and

$$\Gamma \vdash \vec{N} : \vec{\mathrm{El}}(\vec{A})$$

means

$$\begin{aligned}
\Gamma &\vdash N_1 : \mathrm{El}(A_1) \\
\Gamma &\vdash N_2 : \mathrm{El}(A_2[N_1]) \\
&\ldots \\
\Gamma &\vdash N_{|\vec{A}|-1} : \mathrm{El}(A_{|\vec{A}|-1}[N_0, N_1, \ldots, N_{|\vec{N}|-1} = \vec{N}])
\end{aligned}$$

Π, λ and application can be easily generalized to telescopes as follows:

$$\begin{aligned}
\vec{\Pi}\vec{\mathrm{El}}(\vec{A})\sigma &= \Pi\mathrm{El}(A_0).\Pi\mathrm{El}(A_1)\ldots\Pi\mathrm{El}(A_{|\vec{A}|-1}).\sigma \\
\vec{\lambda}\vec{\mathrm{El}}(\vec{A}).M &= \lambda\mathrm{El}(A_0).\lambda\mathrm{El}(A_1)\ldots\lambda\mathrm{El}(A_{|\vec{A}|-1}).M \\
M\,\vec{N} &= MN_0N_1\ldots N_{|\vec{N}|-1}
\end{aligned}$$

A.2. Syntax

We will extend the syntax for terms by additional constants for: set formers (μ), constructors (C) and recursors (R). Every constant has a specification as its argument, which has a fairly complicated structure, specifying the types of all constructors. To simplify the presentation we introduce a special syntactic class $\mathrm{Mu}(S, T, U)$.

A.2.1. DEFINITION. We extend definition 2.1.1 by:

$$\begin{aligned} \mathrm{Mu} &::= (\mathrm{Tm}^*, (\mathrm{Tm}^*, \mathrm{Tm}^*, (\mathrm{Tm}^*, \mathrm{Tm}^*)^*)^*) \\ \mathrm{Tm} &::= \mu T \mid \mathrm{C}^i_T \mid \mathrm{R}^T_\sigma \end{aligned}$$

Note that we do not introduce any new types, similar to the monomorphic presentation of Type Theory.

A.2.2. DEFINITION (Weakening and substitution).

$$\begin{aligned} &(\vec{I}, \{(\vec{A}_i, \vec{P}_i, \{(\vec{B}_{ij}, \vec{Q}_{ij})\}_{j\in m_i})\}_{i\in n})^{+n} \\ &\quad = (\vec{I}^{+n}, \{(A_i^{+n}, P_i^{+n+|\vec{A}_i|}, \{(\vec{B}_{ij}^{+n+|\vec{A}_i|}, \vec{Q}_{ij}^{+n+|\vec{A}_i|+|\vec{B}_{ij}|})\}_{j\in m_i})\}_{i\in n}) \\ &(\vec{I}, \{(\vec{A}_i, \vec{P}_i, \{(\vec{B}_{ij}, \vec{Q}_{ij})\}_{j\in m_i})\}_{i\in n})[N]^n \\ &\quad = (\vec{I}[N]^n, \{(A_i[N]^n, P_i[N]^{n+|\vec{A}_i|}, \{(\vec{B}_{ij}[N]^{n+|\vec{A}_i|}, \vec{Q}_{ij}[N]^{n+|\vec{A}_i|+|\vec{B}_{ij}|})\}_{j\in m_i})\}_{i\in n}) \end{aligned}$$

Assume $T = (\vec{I}, \{(\vec{A}_i, \vec{P}_i, \{(\vec{B}_{ij}, \vec{Q}_{ij})\}_{j\in m_i})\}_{i\in n})$.

$$\begin{aligned} (\mu T)^{+n} &= \mu(T^{+n}) \\ (\mathrm{C}^i_T)^{+n} &= \mathrm{C}^i_{T^{+n}} \\ (\mathrm{R}^\sigma_T)^{+n} &= \mathrm{R}^{\sigma^{+n+|\vec{I}|+1}}_{T^{+n}} \\ (\mu T)[N]^n &= \mu(T[N]^n) \\ (\mathrm{C}^i_T)[N]^n &= \mathrm{C}^i_{T[N]^n} \\ (\mathrm{R}^\sigma_T)[N]^n &= \mathrm{R}^{\sigma[N]^{+n+|\vec{I}|+1}}_{T[N]^n} \end{aligned}$$

A.3. Rules

The formation rule contains a complete specification of all constructors. n is the number of constructors and m_i is the number of *recursive premises* of the ith constructor.

$$\frac{\begin{array}{l} n \in \omega \qquad \{m_i \in \omega\}_{i \in n} \\ \Gamma \vdash \vec{I} : \vec{\mathrm{Set}} \\ \left.\begin{array}{l} \Gamma \vdash \vec{A}_i : \vec{\mathrm{Set}} \\ \Gamma.\vec{\mathrm{El}}(\vec{A}_i) \vdash \vec{P}_i : \vec{\mathrm{El}}(\vec{I}^{+|\vec{A}_i|}) \\ \left.\begin{array}{l} \Gamma.\vec{\mathrm{El}}(\vec{A}_i) \vdash \vec{B}_{ij} : \vec{\mathrm{Set}} \\ \Gamma.\vec{\mathrm{El}}(\vec{A}_i).\vec{\mathrm{El}}(\vec{B}_{ij}) \vdash \vec{Q}_{ij} : \vec{\mathrm{El}}(\vec{I}^{+|\vec{A}_i|+|\vec{B}_{ij}|}) \end{array}\right\} j \in m_i \end{array}\right\} i \in n \end{array}}{\Gamma \vdash \mu(\vec{I}, \{(A_i, P_i, \{(\vec{B}_{ij}, \vec{Q}_{ij})\}_{j \in m_i})\}_{i \in n}) : \vec{\Pi}\vec{\mathrm{El}}(\vec{I}).\mathrm{Set}} \qquad \text{(MU-FORM)}$$

We omit the obvious but elaborated congruence rules. Let

$$T = (\vec{I}, \{(A_i, P_i, \{(\vec{B}_{ij}, \vec{Q}_{ij})\}_{j \in m_i})\}_{i \in n})$$

in the following rules s.t.

$$\Gamma \vdash \mu(T) : \vec{\Pi}\vec{\mathrm{El}}(\vec{I}).\mathrm{Set}$$

If $\vec{I} = \epsilon$ then $\mu(T)$ is a non-dependent type. In this case we also have that all $\vec{P}_i, \vec{Q}_{ij} = \epsilon$ therefore we introduce the following abbrevation:

$$\mu_{\mathrm{N}}(\{(A_i, \{\vec{B}_{ij}\}_{j \in m_i})\}_{i \in n}) = \mu(\epsilon, \{(A_i, \epsilon, \{(\vec{B}_{ij}, \epsilon)\}_{j \in m_i})\}_{i \in n})$$

If $\vec{I} \neq \epsilon$ then $\mu(T)$ is a dependent inductive type or an inductively defined family. If $m_i > 0$ for some i we call $\mu(T)$ *recursive*. If all $B_{ij} = \epsilon$ then $\mu(T)$ is algebraic.

We complete the definition of μ types by giving the introduction, elimination and computation rule.

$$\frac{i \in n}{\Gamma \vdash \mathrm{C}^i_T : \vec{\Pi}\vec{A}_i.(\{\vec{\Pi}\vec{\mathrm{El}}(\vec{B}_{ij}).\vec{\mathrm{El}}(\mu(T)\,\vec{Q}_{ij}\}_{j \in m_i}).\vec{\mathrm{El}}((\mu(T)\,\vec{P}_i)))^+} \qquad \text{(MU-INTRO)}$$

For the following rules we assume

$$\Gamma.\vec{x} : \vec{I}.\mu(T)\,\vec{x} \vdash \sigma$$

as a premise.

For the elimination and the computation rules we have to specify the recursion hypothesis introduced by the ith constructor: [1]

$$\delta_i = \begin{array}{ll} \vec{\Pi} & \vec{a} : \vec{\mathrm{El}}(\vec{A}_i) \\ . & \vec{b} : \{\vec{\Pi}\vec{B}_{ij}.\mu(T)\,\vec{Q}_{ij}\} \\ . & \{\vec{\Pi}\vec{c} : \vec{\mathrm{El}}(\vec{B}_{ij}[\vec{a}]).\sigma[\vec{\mathrm{El}}(\vec{Q}_{ij}[\vec{a}]).(b_j\,\vec{c})])\}_{j\in m_i} \\ . & \sigma[\vec{P}_i[\vec{a}].(\mathrm{C}^i_T\,\vec{a}\,\vec{b})] \end{array}$$

We can verify that $\Gamma \vdash \delta_i$.

$$\Gamma \vdash \mathrm{R}^\sigma_T : \vec{\Pi}\{\delta_i\}_{i\in n}.\vec{x} : \vec{\mathrm{El}}(\vec{I}).(\mu(T)\,\vec{x}).\sigma \qquad \text{(MU-ELIM)}$$

$$\frac{\begin{array}{ll} \Gamma \vdash D_k : \delta_k & \text{for } k \in n \\ i \in n & \\ \Gamma \vdash \vec{X} : \vec{\mathrm{El}}(\vec{A}_i) & \\ \Gamma \vdash Y_j : (\vec{\Pi}\vec{\mathrm{El}}(\vec{B}_{ij}).\vec{\mathrm{El}}(\mu(T)\,\vec{Q}_{ij}))[\vec{X}] & \text{for } j \in m_i \end{array}}{\Gamma \vdash \mathrm{R}^\sigma_T\,\vec{D}\,(\mathrm{C}^i_T\,\vec{X}\,\vec{Y}) \simeq D_i\,\vec{X}\,\vec{Y}\,\{\vec{\lambda}\vec{c}\colon \vec{B}_{ij}[\vec{X}].\mathrm{R}^\sigma_T\,\vec{D}\,(Y_j\,\vec{c})\}_{j\in m_j} : \sigma[\vec{P}_i[\vec{X}].(\mathrm{C}^i_T\,\vec{X}\,\vec{Y})]} \text{(MU-COMP)}$$

A.4. Examples

In the following only the *set formers* and the *constructors* are presented. All typing judgements refer to the empty context. When indexing the constructors we will confuse T and $\mu(T)$.

Σ sets

$$\begin{array}{rcl} \Sigma & = & \lambda X : \mathrm{Set}, Y : X \to \mathrm{Set}.\mu_{\mathrm{N}}(\{(A_0 = x : X.Yx, \{\})\}) \\ & : & \Pi X : \mathrm{Set}.(X \to \mathrm{Set}) \to \mathrm{Set} \end{array}$$

---

[1] Note that we are omitting the applications of the weakening operation to improve readability.

$$\begin{aligned}\mathrm{pair} &= \lambda X : \mathrm{Set}, Y : X \to \mathrm{Set}.\mathrm{C}^0_{\Sigma\, X\, Y} \\ &: \Pi X : \mathrm{Set}.\Pi Y : (X \to \mathrm{Set}) \to \mathrm{Set}.\Pi x : \mathrm{El}(X).\mathrm{El}(Y\, x) \to \Sigma\, X\, Y\end{aligned}$$

Disjoint union

$$\begin{aligned}+ &= \lambda X, Y : \mathrm{Set}.\mu_{\mathrm{N}}(\{(A_0 = X, \{\}), (A_1 = Y, \{\})\}) \\ &: \mathrm{Set} \to \mathrm{Set} \to \mathrm{Set} \\ \mathrm{inl} &= \lambda X, Y : \mathrm{Set}.\mathrm{C}^0_{+\, X\, Y} \\ &: \Pi X, Y : \mathrm{Set}.\mathrm{El}(X) \to +\, X\, Y \\ \mathrm{inr} &= \lambda X, Y : \mathrm{Set}.\mathrm{C}^1_{+\, X\, Y} \\ &: \Pi X, Y : \mathrm{Set}.\mathrm{El}(Y) \to +\, X\, Y\end{aligned}$$

Equality Sets

$$\begin{aligned}\mathrm{Id} &= \lambda X : \mathrm{Set}.\mu(I = X.X, \{(A_0 = x : X, P_0 = x.x, \{\})\}) \\ &: \Pi X : \mathrm{Set}.\mathrm{El}(X) \to \mathrm{El}(X) \to \mathrm{Set} \\ \mathrm{id} &= \lambda X : \mathrm{Set}.\mathrm{C}^0_{\mathrm{Id}\, X} \\ &: \Pi X : \mathrm{Set}.\Pi x : \mathrm{El}(X).\mathrm{Id}\, X\, x\, x\end{aligned}$$

Finite Sets

$$\begin{aligned}\{\} &= \mu_{\mathrm{N}}(\{\}) \\ &: \mathrm{Set} \\ \mathrm{T} &= \mu_{\mathrm{N}}(\{(A_0 = \epsilon, \{\})\}) \\ &: \mathrm{Set} \\ \mathrm{tt} &= \mathrm{C}^0_{\mathrm{T}} \\ &: \mathrm{El}(\mathrm{T})\end{aligned}$$

Natural Numbers

$$\begin{aligned}\mathrm{Nat} &= \mu_{\mathrm{N}}(\{(A_0 = \epsilon, \{\}), (A_1 = \epsilon, \{B_{10} = \epsilon\})\}) \\ &: \mathrm{Set}\end{aligned}$$

$$
\begin{aligned}
0 &= \mathrm{C}^0_{\mathrm{Nat}} \\
&: \mathrm{El}(\mathrm{Nat}) \\
\mathrm{succ} &= \mathrm{C}^1_{\mathrm{Nat}} \\
&: \mathrm{El}(\mathrm{Nat}) \to \mathrm{El}(\mathrm{Nat})
\end{aligned}
$$

Lists

$$
\begin{aligned}
\mathrm{List} &= \lambda X : \mathrm{Set}.\mu_{\mathrm{N}}(\{(A_0 = \epsilon, \{\}), (A_1 = X, \{B_{10} = \epsilon\})\}) \\
&: \mathrm{Set} \\
\mathrm{nil} &= \lambda X : \mathrm{Set}.\mathrm{C}^0_{\mathrm{List}\,X} \\
&: \Pi X : \mathrm{Set}.\mathrm{El}(\mathrm{List}\,X) \\
\mathrm{cons} &= \lambda X : \mathrm{Set}.\mathrm{C}^1_{\mathrm{List}\,X} \\
&: \Pi X : \mathrm{Set}.\mathrm{El}(X) \to \mathrm{El}(\mathrm{List}\,X) \to \mathrm{El}(\mathrm{List}\,X)
\end{aligned}
$$

Vectors

$$
\begin{aligned}
\mathrm{Vec} &= \lambda X : Set.\mu(I = \mathrm{Nat}, \left\{ \begin{array}{c} (A_0 = \epsilon, \{\}, P_0 = 0,), \\ (A_1 = n : \mathrm{Nat}.X, \{(B_{10} = \epsilon, Q_{10} = n)\}, P_1 = \mathrm{succ}\,n) \end{array} \right\}) \\
&: \Pi X : \mathrm{Set}.\mathrm{El}(\mathrm{Nat}) \to \mathrm{Set} \\
\mathrm{vnil} &= \lambda X : \mathrm{Set}.\mathrm{C}^0_{\mathrm{Vec}\,X} \\
&: \Pi X : \mathrm{Set}.\mathrm{El}(\mathrm{List}\,X\,0) \\
\mathrm{vcons} &= \lambda X : \mathrm{Set}.\mathrm{C}^1_{\mathrm{List}\,X} \\
&: \Pi X : \mathrm{Set}.\Pi n : \mathrm{El}(\mathrm{Nat}).\mathrm{El}(X) \to \mathrm{El}(\mathrm{Vec}\,X\,n) \to \mathrm{El}(\mathrm{Vec}\,X\,(\mathrm{succ}\,n))
\end{aligned}
$$

Ordinal notations

$$
\begin{aligned}
\mathrm{Ord} &= \mu_{\mathrm{N}}(\{(A_0 = \epsilon, \{\}), (A_1 = \epsilon, \{B_{10} = \epsilon\}), (A_2 = \epsilon, \{B_{20} = \mathrm{Nat}\})\}) \\
0' &= \mathrm{C}^0_{\mathrm{Ord}} \\
&: \mathrm{El}(\mathrm{Ord}) \\
\mathrm{succ}' &= \mathrm{C}^1_{\mathrm{Ord}} \\
&: \mathrm{El}(\mathrm{Ord}) \to \mathrm{El}(\mathrm{Ord}) \\
\mathrm{lim} &= \mathrm{C}^2_{\mathrm{Ord}} \\
&: (\mathrm{El}(\mathrm{Nat}) \to \mathrm{El}(\mathrm{Ord})) \to \mathrm{El}(\mathrm{Ord})
\end{aligned}
$$

Appendix B

LEGO code

The following contains the LEGO code for the Strong Normalization proof for simply type λ-calculus and System F.

The Lego files can be obtained on e-mail request (alti@dcs.ed.ac.uk) or by anonymous internet ftp from host `ftp.dcs.ed.ac.uk` (address 129.215.160.5), directory `pub/alti`, file `snorm.tar.Z` (this is a compressed tar file which contains a directory including a file `README` for further information). Here is a sample dialog:

```
   ftp ftp.dcs.ed.ac.uk
Name:  anonymous
Password:  your email-address
ftp> binary
ftp> cd pub/alti
ftp> get snorm.tar.Z
ftp> bye
```

After the file is transferred it should be uncompressed and then extracted using `tar`. For example: `zcat snorm.tar.Z | tar fox -`.

B.1. load-simple.l

```
(*
**
**         T. Altenkirch
**
**        Strong Normalisation for simply typed \lambda calculus
**
** This proof does not introduce any non-logical axioms. Assumptions are only
** used to model inductive types.
*)

Init XCC; Logic; [Set=Prop];
Include basic;
Include lambda;
Include snorm;
Include simple;
Include "sn-simple.l";
```

B.2. load-f.l

```
(*
**
**         T. Altenkirch
**
**        Strong Normalisation for System F
**
** This proof does not introduce any non-logical axioms. Assumptions are only
** used to model inductive types.
*)

Init XCC; Logic; [Set=Prop];
Include basic;
Include lambda;
Include snorm;
Include f;
Include "sn-f.l";
```

B.3. basic.l

```
(*
** some basic definitions
**
** these are taken from /home/alti/lego/newLib to keep the proof
** self contained.
*)

(* composition *)

[o[B,C|Type][f:B->C][A|Type][g:A->B] = [a:A]f (g a)];

(* iff is reflexiv & transitive *)

Goal {A:Prop}iff A A;
intros;andI;intros;Immed;intros;Immed;
Save iff_refl;

Goal {A,B,C:Prop}(iff A B)->(iff B C)->(iff A C);
intros;Refine H;intros;Refine H1;intros;andI;
intros;Refine H4;Refine H2;Immed;intros;Refine H3;Refine H5;Immed;
Save iff_trans;

(*
** Definition of Martin-L"of equality &
** the proof of some properties
*)

(* [A|Set]mu[EQ:A->A->Set](EQ_refl:{a:A}EQ a a) *)

[A|Set]
$[EQ : A->A->Set]
$[EQ_refl : {a:A}EQ a a]
$[RecEQ : {C:{a,b|A}(EQ a b)->Type}
              ({a:A}C (EQ_refl a))
              -> {a,b|A}{p:EQ a b}C p];

[[C:{a,b|A}(EQ a b)->Type][f0:{a:A}C (EQ_refl a)]
 [a:A]
   RecEQ C f0 (EQ_refl a) ==> f0 a];

Discharge A;

[A,B|Set];
```

```
[EQ_trans [a,b,c|A][p1:EQ a b][p2:EQ b c] =
        (RecEQ ([a,b|A][_:EQ a b](EQ b c) -> (EQ a c))
           ([a:A][p:EQ a c]p)
           p1) p2 : EQ a c];

[EQ_sym [a,b|A][p:EQ a b] =
        RecEQ ([a,b|A][_:EQ a b]EQ b a)
          ([a:A]EQ_refl a)
          p : EQ b a];

[EQ_resp [f:A->B][a,b|A][p:EQ a b] =
        RecEQ ([a,b|A][_:EQ a b]EQ (f a) (f b))
          ([a:A]EQ_refl (f a))
          p : EQ (f a) (f b)];

[EQ_rewrite [a,b|A][p:EQ a b][P:A->Type] =
        RecEQ ([a,b|A][_:EQ a b](P a)->(P b))
          ([a:A][p:P a]p)
          p : (P a) -> (P b)];
[EQ_rewrite'[a,b|A][p:EQ a b] = EQ_rewrite (EQ_sym p)];

Discharge A;

Configure Qrepl EQ EQ_rewrite EQ_sym;

Goal {A|Set}{c:A}{C:{a|A}(EQ a c)->Set}
                (C (EQ_refl c))->{a|A}{p:EQ a c}C p;
intros;
Claim  {a,c|A}{q:EQ a c}{C:{a|A}(EQ a c)->Set}(C (EQ_refl c))
  -> C q;
Refine ?+1; Refine H;
Refine RecEQ [a,c|A][q:EQ a c]
        {C:{a|A}(EQ a c)->Set}(C (EQ_refl c))->C q;
intros;Refine H1;
Save RecEQ1;

(*
** some basic datatypes
*)

(*** mu[Unit:Set](void:Unit) ***)

[Unit : Set]
[void : Unit]
```

```
[RecUnit : {P:Unit->Type}(P void)->{x:Unit}P x];

[[P:Unit->Type][f0:P void]
   RecUnit P f0 void ==> f0];

[RecUnitN[P|Type] = RecUnit ([_:Unit]P) : P->Unit->P];

(*** mu[Nat:Set](zero:Nat,succ:Nat->Nat) ***)

[Nat : Set]
[zero : Nat][succ : Nat->Nat]

[RecNat : {P:Nat->Type}
          (P zero)->({n:Nat}(P n)->(P (succ n)))
          -> {n:Nat}P n];

[[P:Nat->Type][z:P zero][f:{n:Nat}(P n)->(P (succ n))][n:Nat]
    RecNat P z f zero ==> z
 || RecNat P z f (succ n) ==> f n (RecNat P z f n)];

[RecNatN[C|Type] = RecNat ([_:Nat]C)];

(*  add : nat -> Nat -> Nat
rec add zero n = n
  | add (succ m) n = succ (add m n)
*)
[add = RecNatN ([n:Nat]n)
               ([m:Nat][add_m:Nat->Nat][n:Nat]succ (add_m n))];

Goal {m,n:Nat}EQ (add m (succ n)) (succ (add m n));
Refine RecNat [m:Nat]{n:Nat}EQ (add m (succ n)) (succ (add m n));
intros _;Refine EQ_refl;
intros;Refine EQ_resp succ;Refine H;
Save add_succ_lem;

(* mu[LE:Nat->Nat->Set](
        LE0:{n:Nat}LE zero n,
        LE1:{m,n|Nat}(LE m n) -> (LE (succ m) (succ n)))
*)

[LE:Nat->Nat->Set]
[LE0:{n:Nat}LE zero n]
[LE1:{m,n|Nat}(LE m n) -> (LE (succ m) (succ n))]
[RecLE:{C:{m,n|Nat}(LE m n)->Set}
```

```
           ({n:Nat}(C (LE0 n)))
        -> ({m,n|Nat}{p:LE m n}(C p) -> C (LE1 p))
        -> {m,n|Nat}{p:LE m n}C p];

[[C:{m,n|Nat}(LE m n)->Set][f0:{n:Nat}(C (LE0 n))]
 [f1:{m,n|Nat}{p:LE m n}(C p)->C (LE1 p)]
 [m,n:Nat][p:LE m n]
   RecLE C f0 f1 (LE0 n) ==> f0 n
|| RecLE C f0 f1 (LE1 p) ==> f1 p (RecLE C f0 f1 p)];

[RecLEN[C:Nat->Nat->Set] = RecLE ([m,n:Nat][_:LE m n]C m n)];

(* mu[Fin:Nat->Set](f_zero:{n:Nat}Fin (succ n),
                   f_succ:{n:Nat}(Fin n)->(Fin (succ n))) *)

[Fin : Nat -> Set]
[f_zero : {n:Nat}Fin (succ n)]
[f_succ : {n|Nat}(Fin n) -> (Fin (succ n))]
[RecFin : {P:{n|Nat}(Fin n)->Type}
             ({n:Nat}P (f_zero n)) ->
             ({n|Nat}{m:Fin n}(P m) -> (P (f_succ m))) ->
             {n|Nat}{m:Fin n}P m];

[[P:{n|Nat}(Fin n)->Type][f0:{n:Nat}P (f_zero n)]
 [f1:{n|Nat}{m:Fin n}(P m) -> (P (f_succ m))]
 [k:Nat][m:Fin k]
    RecFin P f0 f1 (f_zero k) ==> f0 k
 || RecFin P f0 f1 (f_succ m) ==> f1 m (RecFin P f0 f1 m)];

[RecFinN [C:Nat->Type] = RecFin ([n|Nat][_:Fin n]C n)
 : ({n:Nat}C (succ n))
 ->({n|Nat}(Fin n)->(C n)->C (succ n))
 ->{n|Nat}(Fin n)->C n];

[Fin2Nat = RecFinN ([_:Nat]Nat)
                ([_:Nat]zero)
                ([n|Nat][_:Fin n]succ)];

[RecFinZero [P:(Fin zero)->Type][i:Fin zero] =
        [P'[k|Nat] = RecNat ([m:Nat](Fin m)->Type)
                ([i:Fin zero]P i)
                ([m:Nat][_:(Fin m)->Type][_:Fin (succ m)]Unit) k]
        RecFin P' ([_:Nat]void) ([n|Nat][i:Fin n][_:P' i]void) i
 : P i];
```

```
[RecFinZeroN[P:Type] = RecFinZero ([_:Fin zero]P)
        : (Fin zero)->P];

[RecFinSucc [P:{n|Nat}(Fin (succ n))->Type]
        [f0:{n:Nat}P (f_zero n)][f1:{n|Nat}{m:Fin n}P (f_succ m)]
        [n|Nat][i:Fin (succ n)] =
        [P'[k|Nat] = RecNat ([m:Nat](Fin m)->Type)
                ([_:Fin zero]Unit)
                ([m:Nat][_:(Fin m)->Type]P|m) k]
        RecFin P' f0 ([n|Nat][i:Fin n][_:P' i]f1 i) i
 : P i];

[RecFinSuccN[P:Nat->Type] = RecFinSucc ([n|Nat][_:Fin (succ n)]P n)];

Goal {n|Nat}{P:(Fin (succ n))->Set}
        (P (f_zero n))
        ->({i:Fin n}P (f_succ i))
        ->{m:Fin (succ n)}P m;
intros;
Claim {n|Nat}{m:Fin (succ n)}{P:(Fin (succ n))->Set}
        (P (f_zero n))->({i:Fin n}P (f_succ i))->(P m);
Refine ?+1;Immed;
Refine RecFinSucc [n|Nat][m:Fin (succ n)]{P:(Fin (succ n))->Set}
          (P (f_zero n))->({i:Fin n}P (f_succ i))->(P m);
intros;Refine H2;
intros;Refine H3;
Save RecFin1;

[RecFin1N [n:Nat][P|Set] = RecFin1 ([_:Fin (succ n)]P)
 :  P->((Fin n)->P)->(Fin (succ n))->P];

(* [A:Set]mu[Vec:Nat->Set](v_nil:Vec zero,
                          v_cons:A->{n|Nat}(Vec n)->(Vec succ n)) *)

[Vec : Set->Nat -> Set]
[v_nil : {A:Set}Vec A zero]
[v_cons : {A|Set}A -> {n|Nat}(Vec A n) -> (Vec A (succ n))]

[RecVec : {A|Set}{P:{n|Nat}(Vec A n)->Type}
          (P (v_nil A)) ->
          ({a:A}{n|Nat}{l:Vec A n}(P l) -> (P (v_cons a l))) ->
          {n|Nat}{l:Vec A n}P l];

[[A:Set][P:{n|Nat}(Vec A n)->Set]
 [n:P (v_nil A)][f:{a:A}{n|Nat}{l:Vec A n}(P l) -> (P (v_cons a l))]
```

```
  [a:A][k:Nat][v:Vec A k]
    RecVec P n f (v_nil A) ==> n
 || RecVec P n f (v_cons a v) ==> f a v (RecVec P n f v)];

[A,B|Set];

[RecVecN [P:Nat->Type] = RecVec ([n|Nat][_:Vec A n]P n)
 : (P zero)->(A->{n|Nat}(Vec A n)->(P n)->P (succ n))->{n|Nat}(Vec A n)->P n];
[RecVecNN [P|Type] = RecVecN ([n:Nat]P)
 : P->(A->{n|Nat}(Vec A n)->P->P)->{n|Nat}(Vec A n)->P];

[RecVecSucc [P:{n|Nat}(Vec A (succ n))->Type]
        [f1:{a:A}{n|Nat}{v:Vec A n}P (v_cons a v)]
        [n|Nat][v:Vec A (succ n)] =
        [P' = RecNat ([i:Nat](Vec A i)->Type)
                ([_:Vec A zero]Unit)
                ([i:Nat][_:(Vec A i)->Type]P|i)]
        RecVec P' void ([a:A][j|Nat][v:Vec A j][_:P' j v]f1 a v) v : P v];
[RecVecSuccN [P:Nat->Type] = RecVecSucc ([n|Nat][_:Vec A (succ n)]P n)];

Goal {n|Nat}{P:(Vec A (succ n))->Set}
          ({a:A}{l:Vec A n}P (v_cons a l))
        ->{m:Vec A (succ n)}P m;
intros;
Claim {n|Nat}{l:Vec A (succ n)}{P:(Vec A (succ n))->Set}
        ({a:A}{l:Vec A n}P (v_cons a l)) -> (P l);
Refine ?+1;intros;Refine H;
Refine RecVecSucc [n|Nat][l:Vec A (succ n)]
        {P:(Vec A (succ n))->Set}({a:A}{l1:Vec A n}P (v_cons a l1))->P l;
intros;Refine H1;
Save RecVec1;

[v_hd = RecVecSuccN ([_:Nat]A) ([a:A][n|Nat][_:Vec A n]a)
      : {n|Nat}(Vec A (succ n)) -> A];
[v_tl = RecVecSuccN (Vec A) ([_:A][n|Nat][v:Vec A n]v)
      : {n|Nat}(Vec A (succ n)) -> (Vec A n)];

DischargeKeep A;

(* v_nth : {n|Nat}(Fin n)->(Vec A n)->A

rec v_nth|(succ n) (f_zero n) = v_hd|A|n
  | v_nth|(succ n) (f_succ i) = o (v_tl|A|n) (v_nth i)
*)
```

```
[v_nth = RecFinN ([n:Nat](Vec A n)->A) (v_hd|A)
                 ([n|Nat][_:Fin n][f:(Vec A n)->A]o f (v_tl|A|n))];

[v_map [f:A->B] =
  RecVec ([i|Nat][_:Vec A i]Vec B i)
         (v_nil B) ([a:A][i|Nat][_:Vec A i][v:Vec B i]v_cons (f a) v)];

[v_append =
  [P[n:Nat]={m|Nat}(Vec A m)->(Vec A (add n m))]
  RecVecN P ([m|Nat][v:Vec A m]v)
         ([a:A][n|Nat][_:Vec A n][f:P n]
          [m|Nat][v:Vec A m]v_cons a (f v))
  : {n|Nat}(Vec A n)->{m|Nat}(Vec A m)->(Vec A (add n m))];

(*

v_insert : {m,n|Nat}(Vec A m)->A->(Vec A n)->(Vec A (succ (add m n)))

v_insert v_nil a v2 = v_cons a v2
v_insert b::v1 a v2 = v_cons b (v_insert v1 a v2)

*)

[v_insert [m,n|Nat][v1:Vec A m][a:A][v2:Vec A n] =
         RecVecN ([m:Nat]Vec A (succ (add m n)))
           (v_cons a v2)
           ([b:A][m|Nat][v1:Vec A m][v_insert_v1:Vec A (succ (add m n))]
                 v_cons b v_insert_v1)
           v1];

Discharge A;

(* We repeat all definitions we did for Vec for large Vectors (VEC) ! *)

(* [A:Type(0)]mu[VEC:Nat->Type(0)](v_nil:VEC zero,
                            v_cons:A->{n|Nat}(VEC n)->(VEC succ n)) *)

[VEC : Type(0) ->Nat -> Type(0)]
[V_nil : {A:Type(0)}VEC A zero]
[V_cons : {A|Type(0)}A -> {n|Nat}(VEC A n) -> (VEC A (succ n))]

[RecVEC : {A|Type(0)}{P:{n|Nat}(VEC A n)->Type}
           (P (V_nil A)) ->
           ({a:A}{n|Nat}{l:VEC A n}(P l) -> (P (V_cons a l))) ->
           {n|Nat}{l:VEC A n}P l];
```

```
[[A:Set][P:{n|Nat}(VEC A n)->Set]
 [n:P (V_nil A)][f:{a:A}{n|Nat}{l:VEC A n}(P l) -> (P (V_cons a l))]
  [a:A][k:Nat][v:VEC A k]
    RecVEC P n f (V_nil A) ==> n
 || RecVEC P n f (V_cons a v) ==> f a v (RecVEC P n f v)];


[A,B|Type(0)];

[RecVECN [P:Nat->Type] = RecVEC ([n|Nat][_:VEC A n]P n)
 : (P zero)->(A->{n|Nat}(VEC A n)->(P n)->P (succ n))->{n|Nat}(VEC A n)->P n];
[RecVECNN [P|Type] = RecVECN ([n:Nat]P)
 : P->(A->{n|Nat}(VEC A n)->P->P)->{n|Nat}(VEC A n)->P];

[RecVECSucc [P:{n|Nat}(VEC A (succ n))->Type]
        [f1:{a:A}{n|Nat}{v:VEC A n}P (V_cons a v)]
        [n|Nat][v:VEC A (succ n)] =
        [P' = RecNat ([i:Nat](VEC A i)->Type)
                ([_:VEC A zero]Unit)
                ([i:Nat][_:(VEC A i)->Type]P|i)]
        RecVEC P' void ([a:A][j|Nat][v:VEC A j][_:P' j v]f1 a v) v : P v];
[RecVECSuccN [P:Nat->Type] = RecVECSucc ([n|Nat][_:VEC A (succ n)]P n)];

Goal {n|Nat}{P:(VEC A (succ n))->Set}
          ({a:A}{l:VEC A n}P (V_cons a l))
        ->{m:VEC A (succ n)}P m;
intros;
Claim {n|Nat}{l:VEC A (succ n)}{P:(VEC A (succ n))->Set}
        ({a:A}{l:VEC A n}P (V_cons a l)) -> (P l);
Refine ?+1;intros;Refine H;
Refine RecVECSucc [n|Nat][l:VEC A (succ n)]
        {P:(VEC A (succ n))->Set}({a:A}{l1:VEC A n}P (V_cons a l1))->P l;
intros;Refine H1;
Save RecVEC1;

[V_hd = RecVECSuccN ([_:Nat]A) ([a:A][n|Nat][_:VEC A n]a)
      : {n|Nat}(VEC A (succ n)) -> A];
[V_tl = RecVECSuccN (VEC A) ([_:A][n|Nat][v:VEC A n]v)
      : {n|Nat}(VEC A (succ n)) -> (VEC A n)];

DischargeKeep A;

[V_nth = RecFinN ([n:Nat](VEC A n)->A) (V_hd|A)
                ([n|Nat][_:Fin n][f:(VEC A n)->A]o f (V_tl|A|n))];
```

```
[V_map [f:A->B] =
  RecVEC ([i|Nat][_:VEC A i]VEC B i)
         (V_nil B) ([a:A][i|Nat][_:VEC A i][v:VEC B i]V_cons (f a) v)];


[V_append =
  [P[n:Nat]={m|Nat}(VEC A m)->(VEC A (add n m))]
  RecVECN P ([m|Nat][v:VEC A m]v)
         ([a:A][n|Nat][_:VEC A n][f:P n]
          [m|Nat][v:VEC A m]V_cons a (f v))
  : {n|Nat}(VEC A n)->{m|Nat}(VEC A m)->(VEC A (add n m))];

[V_insert [m,n|Nat][v1:VEC A m][a:A][v2:VEC A n] =
         RecVECN ([m:Nat]VEC A (succ (add m n)))
           (V_cons a v2)
           ([b:A][m|Nat][v1:VEC A m][V_insert_v1:VEC A (succ (add m n))]
                 V_cons b V_insert_v1)
           v1];

Discharge A;
```

B.4. lambda.l

```
(*
**
** untyped \lambda terms
**
*)



(*
** Definition of terms
*)

(* mu[Tm:Set](var : Nat -> Tm,
              app : Tm -> Tm -> Tm,
              lam : Tm -> Tm)
*)

$[Tm : Set]
$[var : Nat -> Tm]
$[app : Tm -> Tm -> Tm]
$[lam : Tm -> Tm]

$[RecTm : {P:Tm->Type}
           ({n:Nat}P (var n))
        -> ({M|Tm}(P M)->{N|Tm}(P N)->P (app M N))
        -> ({M|Tm}(P M)->P (lam M))
        -> {M:Tm}P M];

[[P:Tm->Type][var_:{n:Nat}P (var n)]
 [app_:{M|Tm}(P M)->{N|Tm}(P N)->P (app M N)]
 [lam_:{M|Tm}(P M)->P (lam M)]
 [n:Nat][M,N:Tm]
   RecTm P var_ app_ lam_ (var n) ==> var_ n
|| RecTm P var_ app_ lam_ (app M N) ==> app_ (RecTm P var_ app_ lam_ M)
                                            (RecTm P var_ app_ lam_ N)
|| RecTm P var_ app_ lam_ (lam M) ==> lam_ (RecTm P var_ app_ lam_ M)];

[RecTmN [P|Type] = RecTm ([_:Tm]P)
 : (Nat->P)->(Tm->P->Tm->P->P)->(Tm->P->P)->Tm->P];

(*
** Simple lemmas about terms
*)
```

```
Goal {i:Nat}{M,M':Tm}not (EQ (app M M') (var i));
Intros i M M' H;
Refine EQ_rewrite H
         (RecTmN ([_:Nat]absurd) ([_:Tm][_:Set][_:Tm][_:Set]Unit)
                 ([_:Tm][_:Set]Unit));
Refine void;
Save neq_var_app;

Goal {i:Nat}{M:Tm}not (EQ (lam M)  (var i));
Intros i M H;
Refine EQ_rewrite H
         (RecTmN ([_:Nat]absurd) ([_:Tm][_:Set][_:Tm][_:Set]Unit)
                 ([_:Tm][_:Set]Unit));
Refine void;
Save neq_var_lam;

Goal {N,M1,M2:Tm}not (EQ (lam N) (app M1 M2));
Intros N M1 M2 H;
Refine EQ_rewrite H
         (RecTmN ([_:Nat]Unit) ([_:Tm][_:Set][_:Tm][_:Set]absurd)
                 ([_:Tm][_:Set]Unit));
Refine void;
Save neq_lam_app;

Goal  {M1,M2,N1,N2:Tm}(EQ (app M1 M2) (app N1 N2))
                ->((EQ M1 N1)/\(EQ M2 N2));
intros;
Refine EQ_rewrite H
         (RecTmN ([_:Nat]Unit)
                 ([N1':Tm][_:Set][N2':Tm][_:Set](EQ M1 N1')/\(EQ M2 N2'))
                 ([_:Tm][_:Set]Unit));
andI;Refine EQ_refl;Refine EQ_refl;
Save inj_app;

Goal {M1,M2:Tm}(EQ (lam M1) (lam M2)) -> (EQ M1 M2);
intros;
Refine EQ_rewrite H
         (RecTmN ([_:Nat]Unit) ([_:Tm][_:Set][_:Tm][_:Set]Unit)
                 ([M2':Tm][_:Set]EQ M1 M2'));
Refine EQ_refl;
Save inj_lam;

(*
** weakening & substitution
*)
```

```
(* weak_var : Nat -> Nat -> Nat

rec weak_var zero     i = (succ i)
  | weak_var (succ l) zero = zero
  | weak_var (succ l) (succ i) = succ (weak_var l i)

*)

[weak_var =
  RecNatN succ
          ([_:Nat][weak_l:Nat->Nat]
           RecNatN zero ([i:Nat][_:Nat]succ (weak_l i)))];

(* weak : Nat->Tm->Tm

rec weak l (var i) = var (weak_var l i)
  | weak l (app M N) = app (weak l M) (weak l N)
  | weak l (lam M) = lam (weak (succ l) M)
*)

[weak[l:Nat][M:Tm] =
  RecTmN ([i:Nat][l:Nat]var (weak_var l i))
         ([_:Tm][weak_M:Nat->Tm]
          [_:Tm][weak_N:Nat->Tm]
            [l:Nat]app (weak_M l) (weak_N l))
         ([_:Tm][weak_M:Nat->Tm]
            [l:Nat]lam (weak_M (succ l)))
        M l]

[weak0 = weak zero : Tm -> Tm];

[rep_weak0[M:Tm] = RecNatN M ([_:Nat][M:Tm]weak0 M)];

Goal {n:Nat}EQ (rep_weak0 (var zero) n) (var n);
Refine RecNat [n:Nat]EQ (rep_weak0 (var zero) n) (var n);
Refine EQ_refl;
intros;Equiv EQ ? (weak0 (var n));Refine EQ_resp weak0;Refine H;
Save rep_weak0_lem;

(* subst_var : Nat->Nat->Tm->Tm

rec subst_var zero zero N = N
  | subst_var zero (succ i) N = var i
  | subst_var (succ l) zero N = var zero
```

```
  | subst_var (succ l) (succ i) N = weak0 (subst_var l i N)
*)

[subst_var =
  RecNatN (RecNatN ([N:Tm]N) ([i:Nat][_:Tm->Tm][_:Tm]var i))
          ([l:Nat][subst_var_l : Nat->Tm->Tm]
             RecNatN ([_:Tm]var zero)
                ([i:Nat][_:Tm->Tm][N:Tm]weak0 (subst_var_l i N)))];

(* subst : Nat->Tm->Tm->Tm

rec subst l (var i) N = subst_var l i N
  | subst l (app M M') N = app (subst l M) (subst l N)
  | subst l (lam M) N = lam (subst (succ l) M N)

*)

[subst[l:Nat][M,N:Tm] =
  RecTmN ([i:Nat][l:Nat]subst_var l i N)
         ([_:Tm][subst_M:Nat->Tm][_:Tm][subst_N:Nat->Tm]
                [l:Nat]app (subst_M l) (subst_N l))
         ([_:Tm][subst_M:Nat->Tm]
                [l:Nat]lam (subst_M (succ l)))
        M l];


[subst0 = subst zero];

(*
** substitution & weakening laws
*)

Goal {l:Nat}{M,N:Tm}EQ (subst l (weak l M) N) M;
intros;
Refine RecTm [M:Tm]{l:Nat}EQ (subst l (weak l M) N) M;
(* var *)
intros;
Refine RecNat [l:Nat]{i:Nat}EQ (subst l (weak l (var i)) N) (var i);
intros;Refine EQ_refl;
intros l2 IH i;
Refine RecNat [i:Nat]EQ (subst (succ l2) (weak (succ l2) (var i)) N) (var i);
Refine EQ_refl;
intros i1 _;
Equiv EQ (weak0 (subst l2 (weak l2 (var i1)) N)) (weak0 (var i1));
Refine EQ_resp weak0;Refine IH;
```

```
(* app *)
intros;Refine EQ_rewrite' (H l1) ([X:Tm]EQ (app X ?) ?);
Refine EQ_rewrite' (H1 l1) ([X:Tm]EQ (app ? X) ?);
Refine EQ_refl;
(* lam *)
intros;Refine EQ_rewrite' (H (succ l1)) ([X:Tm]EQ (lam X) ?);
Refine EQ_refl;
Save subst_weak_lem;

Goal {l',l:Nat}{M:Tm}(LE l' l)->
        (EQ (weak (succ l) (weak l' M))
            (weak l' (weak l M)));
intros;
Refine RecTm [M:Tm]{l',l:Nat}(LE l' l)->
         EQ (weak (succ l) (weak l' M)) (weak l' (weak l M));
(*** var ***)
intros i l1' l1 _;Refine EQ_resp var;
Refine RecLEN [l1',l1:Nat]{i:Nat}
         EQ (weak_var (succ l1) (weak_var l1' i))
            (weak_var l1' (weak_var l1 i));
(** LE0 **)
intros;Refine EQ_refl;
(** LE1 **)
intros l2' l2 _ IH;
Refine RecNat [i:Nat]
        EQ (weak_var (succ (succ l2)) (weak_var (succ l2') i))
           (weak_var (succ l2') (weak_var (succ l2) i));
(* i=0 *) Refine EQ_refl;
(* succ i *)
intros i' _;Refine EQ_resp succ;Refine IH;
(** **) Immed;
(*** app ***)
intros;
Refine EQ_rewrite ? ([X:Tm]EQ ? (app X ?));Next +1;Refine H1;Immed;
Refine EQ_rewrite ? ([X:Tm]EQ ? (app ? X));Next +1;Refine H2;Immed;
Refine EQ_refl;
(*** lam ***)
intros;
Refine EQ_rewrite ? ([X:Tm]EQ ? (lam X));Next +1;Refine H1;Refine LE1;Immed;
Refine EQ_refl;
(*** ***) Immed;
Save weak_weak_lem;

Goal {l',l:Nat}{M,N:Tm}(LE l' l)->
        (EQ (subst (succ l) (weak l' M) N)
```

```
            (weak l' (subst l M N)));
intros;
Refine RecTm [M:Tm]{l',l:Nat}(LE l' l)->
        (EQ (subst (succ l) (weak l' M) N)
            (weak l' (subst l M N)));
(*** var ***)
intros i l1' l1 _;
Refine RecLEN [l',l:Nat]{i:Nat}
        EQ (subst (succ l) (weak l' (var i)) N) (weak l' (subst l (var i) N));
(** LE0 **)
intros;Refine EQ_refl;
(** LE_1 **)
intros l2' l2 _ IH;
Refine RecNat [i:Nat]
        EQ (subst (succ (succ l2)) (weak (succ l2') (var i)) N)
           (weak (succ l2') (subst (succ l2) (var i) N));
(* i = 0 *) Refine EQ_refl;
(* succ i *)
intros i' _;
Refine EQ_trans;Next +2;Refine EQ_sym;Refine weak_weak_lem;Refine LE0;
Refine EQ_resp weak0;Refine IH;
(* *) Immed;
(*** app ***)
intros;
Refine EQ_rewrite ? ([X:Tm]EQ ? (app X ?));Next +1;Refine H1;Immed;
Refine EQ_rewrite ? ([X:Tm]EQ ? (app ? X));Next +1;Refine H2;Immed;
Refine EQ_refl;
(*** lam ***)
intros;
Refine EQ_rewrite ? ([X:Tm]EQ ? (lam X));Next +1;Refine H1;Refine LE1;Immed;
Refine EQ_refl;
(*** ***) Immed;
Save subst_weak_lem';

Goal {l:Nat}{M,N1,N2:Tm}
        EQ (subst l (subst (succ l) M N1) N2)
           (subst l (subst l M (weak0 N2)) N1);
intros;
Refine RecTm [M:Tm]{l:Nat}
  EQ (subst l (subst (succ l) M N1) N2) (subst l (subst l M (weak0 N2)) N1);
(*** var ***)
intros;
Refine RecNat [l:Nat]{i:Nat}
   EQ (subst l (subst (succ l) (var i) N1) N2)
      (subst l (subst l (var i) (weak0 N2)) N1);
```

```
(** l = 0 **)
Refine RecNat [i:Nat]
   EQ (subst zero (subst (succ zero) (var i) N1) N2)
      (subst zero (subst zero (var i) (weak0 N2)) N1);
(* i = 0 *)
Refine EQ_trans;Next +2;Refine EQ_sym;Refine subst_weak_lem;Refine EQ_refl;
(* succ i *)
intros;Refine EQ_trans;Next +1;Refine subst_weak_lem;Refine EQ_refl;
(** succ l **)
intros l2 IH;
Refine RecNat [i:Nat]
   EQ (subst (succ l2) (subst (succ (succ l2)) (var i) N1) N2)
      (subst (succ l2) (subst (succ l2) (var i) (weak0 N2)) N1);
(* i = 0 *)
Refine EQ_refl;
(* succ i *)
intros i _;
Refine EQ_trans;Next +1;Refine subst_weak_lem';Refine LE0;
Refine EQ_trans;Next +2;Refine EQ_sym;Refine subst_weak_lem';Refine LE0;
Refine EQ_resp weak0;Refine IH;
(* app *)
intros;Refine EQ_rewrite (H l1) ([X:Tm]EQ ? (app X ?));
Refine  EQ_rewrite (H1 l1) ([X:Tm]EQ ? (app ? X));Refine EQ_refl;
(* lam *)
intros;Refine EQ_rewrite (H (succ l1)) ([X:Tm]EQ ? (lam X));
Refine EQ_refl;
Save subst_subst_lem;

Goal {m,l:Nat}{M1,M2,N:Tm}
        EQ (subst m (subst (succ (add m l)) M1 N) (subst l M2 N))
           (subst (add m l) (subst m M1 M2) N);
intros m' l' M1' M2' N';
Refine RecTm [M1:Tm]{m,l:Nat}{M2,N:Tm}
        EQ (subst m (subst (succ (add m l)) M1 N) (subst l M2 N))
           (subst (add m l) (subst m M1 M2) N);
(*** var ***)
intros;
Refine RecNat [i:Nat]{m:Nat}
        EQ (subst m (subst (succ (add m l)) (var i) N) (subst l M2 N))
           (subst (add m l) (subst m (var i) M2) N);
(** i=0 **)
Refine RecNat [m:Nat]
        EQ (subst m (var zero) (subst l M2 N))
           (subst (add m l) (subst m (var zero) M2) N);
(* m=0 *) Refine EQ_refl;
```

```
(* succ m *) intros; Refine EQ_refl;
(** succ i **)
intros i IH;
Refine RecNat [m:Nat]
        EQ (subst m (subst (succ (add m l)) (var (succ i)) N) (subst l M2 N))
           (subst (add m l) (subst m (var (succ i)) M2) N);
(* m=0 *) intros; Refine subst_weak_lem;
(* succ m *)
intros i1 _;
Refine EQ_trans;Next +1;Refine subst_weak_lem';Refine LE0;
Refine EQ_trans;Next +2;Refine EQ_sym;Refine subst_weak_lem';Refine LE0;
Refine EQ_resp weak0;Refine IH;
(*** app ***)
intros;
Refine EQ_rewrite ? ([X:Tm]EQ ? (app X ?));Next +1;Refine H;
Refine EQ_rewrite ? ([X:Tm]EQ ? (app ? X));Next +1;Refine H1;
Refine EQ_refl;
(*** lam ***)
intros;
Refine EQ_rewrite ? ([X:Tm]EQ ? (lam X));Next +1;Refine H;
Refine EQ_refl;
Save subst_subst_lem';

(*
** one-step reduction
*)

(* mu[Step:Tm->Tm->Set](
        beta : {M,N:Tm}Step (app (lam M) N) (subst0 M N),
        app_l : {M,M',N:Tm}(Step M M')->(Step (app M N) (app M' N)),
        app_r : {M,M',N:Tm}(Step M M')->(Step (app N M) (app N M')),
        xi : {M,N:Tm}(Step M M')->(Step (lam M) (lam M')) )
*)

$[Step:Tm->Tm->Set]
$[beta : {M,N:Tm}Step (app (lam M) N) (subst0 M N)]
$[app_l : {M,M',N:Tm}(Step M M')->(Step (app M N) (app M' N))]
$[app_r : {M,M',N:Tm}(Step M M')->(Step (app N M) (app N M'))]
$[xi : {M,M':Tm}(Step M M')->(Step (lam M) (lam M'))]
$[RecStep:{P:{M,N|Tm}(Step M N)->Type}
        ({M,N:Tm}P (beta M N))
        -> ({M,M',N:Tm}{p:Step M M'}(P p)->(P (app_l M M' N p)))
        -> ({M,M',N:Tm}{p:Step M M'}(P p)->P (app_r M M' N p))
        -> ({M,M':Tm}{p:Step M M'}(P p)->P (xi M M' p))
        -> {M,N|Tm}{p:Step M N}P p];
```

```
[[P:{M,N|Tm}(Step M N)->Type]
 [beta_ : {M,N:Tm}P (beta M N)]
 [app_l_ : {M,M',N:Tm}{p:Step M M'}(P p)->P (app_l M M' N p)]
 [app_r_ : {M,M',N:Tm}{p:Step M M'}(P p)->P (app_r M M' N p)]
 [xi_ : {M,M':Tm}{p:Step M M'}(P p)->P (xi M M' p)]
 [M,M',N:Tm][p:Step M M']
   RecStep P beta_ app_l_ app_r_ xi_ (beta M N) ==> beta_ M N
|| RecStep P beta_ app_l_ app_r_ xi_ (app_l M M' N p)
       ==> app_l_ M M' N p (RecStep P beta_ app_l_ app_r_ xi_ p)
|| RecStep P beta_ app_l_ app_r_ xi_ (app_r M M' N p)
       ==> app_r_ M M' N p (RecStep P beta_ app_l_ app_r_ xi_ p)
|| RecStep P beta_ app_l_ app_r_ xi_ (xi M M' p)
       ==> xi_ M M' p (RecStep P beta_ app_l_ app_r_ xi_ p)];

[RecStepN[P:Tm->Tm->Type] = RecStep ([M,N|Tm][_:Step M N]P M N)
 : ({M,N:Tm}P (app (lam M) N) (subst0 M N))
   ->({M,M',N:Tm}(Step M M')->(P M M')->P (app M N) (app M' N))
   ->({M,M',N:Tm}(Step M M')->(P M M')->P (app N M) (app N M'))
   ->({M,M':Tm}(Step M M')->(P M M')->P (lam M) (lam M'))
   ->{M,N|Tm}(Step M N)->P M N];

(*
** Lemmas about Step
*)

Goal {M,lam_M':Tm}(Step (lam M) lam_M')
       ->Ex[M':Tm](Step M M') /\ (EQ lam_M' (lam M'));
intros;
Claim {lam_M,lam_M':Tm}(EQ (lam M) lam_M)->(Step (lam M) lam_M')
       ->Ex ([M':Tm]and (Step M M') (EQ lam_M' (lam M')));
Refine ?+1;Next +1;Refine EQ_refl;Immed;
intros;
Refine RecStepN [lam_M,lam_M':Tm](EQ (lam M) lam_M)
       ->Ex ([M':Tm]and (Step M M') (EQ lam_M' (lam M'')));
intros;Refine neq_lam_app;Next +3;Refine H3;
intros;Refine neq_lam_app;Next +3;Refine H5;
intros;Refine neq_lam_app;Next +3;Refine H5;
intros;Refine ExIntro;Next +1;andI;
Refine EQ_rewrite' ? ([X:Tm]Step X ?);Next +1;Refine inj_lam;Refine H5;
Refine H3;Refine EQ_refl;
Next +1;Refine H2;Refine EQ_refl;
Save step_lam;

Goal {l:Nat}{M,M',N:Tm}(Step M M')->(Step (subst l M N) (subst l M' N));
```

```
intros;
Refine RecStepN [M,M':Tm]{l:Nat}Step (subst l M N) (subst l M' N);
(* beta *)
intros;
Refine EQ_rewrite ? [X:Tm]Step ? X;Next +1;Refine subst_subst_lem';
Refine beta;
(* app_l *)
intros;Refine app_l;Refine H2;
(* app_r *)
intros;Refine app_r;Refine H2;
(* xi *)
intros;Refine xi;Refine H2;
(* End of cases *)
Immed;
Save step_subst;

[nf[M:Tm] = {M':Tm}not (Step M M')];

Goal {i:Nat}(nf (var i));
Intros i M H;
Claim {M,M'|Tm}(Step M M')-> not (EQ M (var i));
Refine ?+1;Next +3;Refine EQ_refl;Immed;
Refine RecStepN [M,M':Tm]not (EQ M (var i));
intros;Refine neq_var_app;
intros;Refine neq_var_app;
intros;Refine neq_var_app;
intros;Refine neq_var_lam;
Save nf_var;

(* examples *)

(*
[T0 = var zero][T1 = lam T0];
[T2 = lam (app (var zero) (var (succ zero)))];
weak0 T0; Normal VReg;
weak0 T1; Normal VReg;
weak0 T2; Normal VReg;
subst0 T2 T1; Normal VReg;
subst0 T2 T0; Normal VReg;
*)
```

B.5. snorm.l

```
(*
** Definition of SNORM
*)

(*
mu[SN:Tm->Set](
  SNi : {M:Tm}({N:Tm}(Step M N)->(SN N))->(SN M))
*)

$[SN:Tm->Set]
$[SNi : {M:Tm}({N:Tm}(Step M N)->(SN N))->(SN M)]
$[RecSN : {P:{M|Tm}(SN M)->Type}
        ({M:Tm}{F:{N:Tm}(Step M N)->(SN N)}
                ({N:Tm}{R:Step M N}P (F N R))
                -> (P (SNi M F)))
        -> {M|Tm}{p:SN M}P p];

[RHS_SN[P:{M|Tm}(SN M)->Type]
 [SNi_ : {M:Tm}{F:{N:Tm}(Step M N)->(SN N)}
                ({N:Tm}{R:Step M N}P (F N R))
                -> (P (SNi M F))]
 [M:Tm][F:{N:Tm}(Step M N)->(SN N)] =
 SNi_ M F ([N:Tm][R:Step M N]RecSN P SNi_ (F N R))];

[[P:{M|Tm}(SN M)->Type]
 [SNi_ : {M:Tm}{F:{N:Tm}(Step M N)->(SN N)}
                ({N:Tm}{R:Step M N}P (F N R))
                -> (P (SNi M F))]
 [M:Tm][F:{N:Tm}(Step M N)->(SN N)]
   RecSN P SNi_ (SNi M F) ==>
(*         SNi_ M F ([N:Tm][R:Step M N]RecSN P SNi_ (F N R)) *)
        RHS_SN P SNi_ M F];

[RecSNN [P:Tm->Type] = RecSN ([M|Tm][_:SN M]P M)
        : ({M:Tm}({N:Tm}(Step M N)->SN N)->({N:Tm}(Step M N)->P N)->P M)
          ->{M|Tm}(SN M)->P M];

(* lemmas about SN *)

Goal {M|Tm}(SN M)->{N:Tm}(Step M N)->(SN N);
intros M sn_M;
Refine RecSNN ([M:Tm]{N:Tm}(Step M N)->(SN N));
intros;Refine H;Refine H2;
```

```
Refine sn_M;
Save SNd;

Goal {M,N|Tm}(SN (app M N)) -> (SN M);
Intros;
Claim {MN|Tm}(SN MN)->{M,N|Tm}(EQ MN (app M N)) -> (SN M);
Refine ?+1;Next +1;Refine H;Next +1;Refine EQ_refl;
Refine RecSNN [MN|Tm]{M,N|Tm}(EQ MN (app M N))->SN M;
intros;Refine SNi;intros;Refine H2;
Next +1;Refine EQ_rewrite' H3 ([M:Tm]Step M ?+1);
Refine app_l;Next +1;Refine H4;Next +1;Refine EQ_refl;
Save SNapp;

Goal {M|Tm}{n:Nat}(SN (subst0 M (var n)))->(SN M);
intros;
Refine RecSNN [M':Tm]{M:Tm}(EQ M' (subst0 M (var n)))->(SN M);
intros;Refine SNi;intros;Refine H2;Next +2;Refine EQ_refl;
Refine EQ_rewrite' H3 ([X:Tm]Step X ?);Refine step_subst;
Refine H4;
(**)
Next +1;Refine H;Refine EQ_refl;
Save SNvar;

(*
** Definition of Candidates
*)

[neutr[M:Tm] = {M':Tm}not (EQ (lam M') M)];

[P:Tm -> Set]

[CR1 = {M|Tm}(P M)->(SN M)]
[CR2 = {M|Tm}(P M)->{N:Tm}(Step M N)->(P N)]
[CR3 = {M|Tm}(neutr M)->
         ({N:Tm}(Step M N)->(P N))->(P M)]
[CR = CR1 /\ CR2 /\ CR3];

Discharge P;

(*
** Candidates contain all variables
*)

Goal {P:Tm->Set}(CR P)->{i:Nat}P (var i);
Intros;
```

```
Refine snd (snd H);
(* neutral *)
Refine neq_var_lam;
(* is in nf *)
Intros;Refine nf_var;Next +2;Refine H1;
Save CR_nonEmpty;

(*
** SN is a candidate
*)

Goal CR SN;
andI;
Intros;Immed;
andI;
Intros;Refine SNd;Immed;
Intros;Refine SNi;Refine H1;
Save CR_SN;

(*
** ARR preserves candidates
*)

[ARR[P,R:Tm->Set] = [M:Tm]{N:Tm}(P N)->(R (app M N))];

Goal {P,R:Tm->Set}(CR P)->(CR R)->(CR (ARR P R));
intros P R CR_P CR_R;andI;
(* CR1 *)
Intros M H;Refine SNapp;Refine var zero;
Refine fst CR_R;Refine H;Refine CR_nonEmpty;Refine CR_P;
andI;
(* CR2 *)
Intros M H1 N H2 N1 H3;
Refine fst (snd CR_R);Next +2;Refine app_l;Next +1;Refine H2;
Refine H1;Refine H3;
(* CR3 *)
Intros M nM Hyp N P_N;
Refine snd(snd CR_R);
Intros _;Refine neq_lam_app;
Claim (P N)->{L:Tm}(Step (app M N) L)->(R L);
Refine ?+1;Refine P_N;
Refine RecSNN [N:Tm](P N)->{L:Tm}(Step (app M N) L)->R L;
intros N' _ IH P_N' L H1;
Claim {MN':Tm}(EQ MN' (app M N'))->(R L);
Refine ?+1;Next +1;Refine EQ_refl;Intros MN' EQ_MN';
```

```
Refine RecStepN [MN',L:Tm](EQ MN' (app M N'))->R L;
(* impossible *)
intros;Refine inj_app;Next +4;Refine H2;intros;Refine nM;
Next +1;Refine H3;
(* M was reduced *)
intros;Refine inj_app;Next +4;Refine H4;intros;
Refine Hyp;
Refine EQ_rewrite H5 ([M:Tm]Step M M');Refine H2;
Refine EQ_rewrite' H6 P;Refine P_N';
(* N was reduced *)
intros;Refine inj_app;Next +4;Refine H4;intros;
Refine snd (snd CR_R);Intros _;Refine neq_lam_app;
intros;Refine IH;Refine M';Refine EQ_rewrite H6 ([N':Tm]Step N' M');
Refine H2;Refine fst(snd CR_P);Refine N';Refine P_N';
Refine EQ_rewrite H6 ([N':Tm]Step N' M');Refine H2;
Refine EQ_rewrite H5 ([M:Tm]Step (app M M') N2);Refine H7;
(* impossible *)
intros;Refine neq_lam_app;Next +3;Refine H4;
(* end of cases *)
Next +1;Refine H1;Refine EQ_refl;
(* SN N *)
Refine fst (CR_P); Refine P_N;
Save CR_ARR;

(*
** Soundness of Lam
*)

Goal {P,R:Tm->Set}(CR P)->(CR R)->{M:Tm}
        ({N:Tm}(P N)->(R (subst0 M N)))
        ->(ARR P R (lam M));
Intros P R CR_P CR_R M H L P_L;
Refine RecSNN [M:Tm]{L:Tm}(P L)->
        ({N:Tm}(P N)->R (subst0 M N))->R (app (lam M) L);
intros M1 _ IH1 L1 P_L1 Hyp;
Refine RecSNN [L:Tm](P L)->(R (app (lam M1) L));
intros L2 _ IH2 P_L2;
(* Apply CR3 for R *)
Refine snd(snd CR_R);Intros _;Refine neq_lam_app;intros MM _;
(* Induction over Step *)
Refine RecStepN [lam_M1_L2,MM:Tm](EQ lam_M1_L2 (app (lam M1) L2))->(R MM);
(** beta **)
intros M1' L2' _;Refine inj_app;Next +4;Refine H4;intros;
Refine EQ_rewrite' ? ([X:Tm]R (subst0 X L2'));Next +1;
Refine inj_lam;Refine H5;Refine EQ_rewrite' H6 ([X:Tm]R (subst0 M1 X));
```

```
Refine Hyp;Refine P_L2;
(** app_l **)
intros lam_M1 lam_M1' L3 ___;Refine inj_app;Next +4;Refine H6;intros;
Refine step_lam;Next +2;Refine EQ_rewrite H7 ([X:Tm]Step X ?+2);Refine H4;
intros M1' _;Refine H9;intros;
Refine EQ_rewrite' H11 ([X:Tm]R (app X L3));
Refine IH1;Refine H10;Refine EQ_rewrite' H8 P;Refine P_L2;
intros;Refine fst (snd CR_R);Next +2;Refine step_subst;Next +1;Refine H10;
Refine Hyp;Refine H12;
(** app_r **)
intros L3 L3' lam_M2 ___;Refine inj_app;Next +4;Refine H6;intros;
Refine EQ_rewrite' H7 ([X:Tm]R (app X L3'));
Refine IH2;
Next +1;Refine fst (snd CR_P);Next +1;Refine P_L2;Refine ?+1;
Refine EQ_rewrite H8 ([X:Tm]Step X L3');Refine H4;
(** xi - impossible **)
intros;Refine neq_lam_app;Next +3;Refine H6;
(** END of CASES **)
Next +1; Refine H3; Refine EQ_refl;
Refine fst CR_P;Immed;
(* SN_M *) Refine SNvar zero;Refine fst CR_R;Refine H;Refine CR_nonEmpty;
Immed;
Save Lam_sound;

[PI[F:(Tm->Set)->(Tm->Set)] =
        [M:Tm]{P:Tm->Set}(CR P)->(F P M)];

Goal {F:(Tm->Set)->(Tm->Set)}
        ({P:Tm->Set}(CR P)->(CR (F P)))
        -> (CR (PI F));
intros;andI;
(* CR1 *) Intros;Refine (fst (H ? ?));Refine SN;Next +1;Refine H1;
Refine ?+1;Refine CR_SN;
andI;
(* CR2 *) Intros;Refine (fst (snd (H P H3)));Next +1;Refine H1;Immed;
(* CR3 *) Intros;Refine (snd (snd (H P H3)));Refine H1;intros;Refine H2;
Immed;
Save CR_PI;
```

B.6. simple.l

```
(*
** simply typed \lambda calculus
*)

(*
** Types
*)

(* mu[Ty:Set](ty0 : Ty, arr : Ty -> Ty -> Ty) *)

[Ty : Set]
[ty0 : Ty]
[arr : Ty -> Ty -> Ty];
[RecTy : {P:Ty->Type}(P ty0) ->
            ({s:Ty}(P s)->{t:Ty}(P t) -> (P (arr s t)))
         -> ({s:Ty}P s)];

[[P:Ty->Type][f0:P ty0][f1:{s:Ty}(P s)->{t:Ty}(P t) -> (P (arr s t))]
 [s,t:Ty]
   RecTy P f0 f1 ty0 ==> f0
|| RecTy P f0 f1 (arr s t) ==> f1 s (RecTy P f0 f1 s)
                                 t (RecTy P f0 f1 t)];

[RecTyN[P|Type] = RecTy ([_:Ty]P)
 : P->(Ty->P->Ty->P->P)->Ty->P];

(*
** Contexts
*)

[Con = Vec Ty]
[empty = v_nil Ty : Con zero];

(*
** Derivations
*)

(* mu[Der:{n|Nat}(Con n)->Tm->Ty->Set](
        Var : {n|Nat}{G:Con n}{i:Fin n}
                Der G (var (Fin2Nat i)) (v_nth i G)
        App : {n|Nat}{G|Con n}{s,t|Ty}{M,N|Tm}
                (Der G M (arr s t))
                -> (Der G N s)
```

```
                -> (Der G (app M N) t)
        Lam :  {n|Nat}{G|Con n}{s,t|Ty}{M|Tm}
                (Der (v_cons s G) M t)
                -> (Der G (lam M) (arr s t))
*)

$[Der : {n|Nat}(Con n)->Tm->Ty->Set]
$[Var : {n|Nat}{G:Con n}{i:Fin n}
                Der G (var (Fin2Nat i)) (v_nth i G)]
$[App : {n|Nat}{G|Con n}{s,t|Ty}{M,N|Tm}
                (Der G M (arr s t))
                -> (Der G N s)
                -> (Der G (app M N) t)]
$[Lam : {n|Nat}{G|Con n}{s,t|Ty}{M|Tm}
                (Der (v_cons s G) M t)
                -> (Der G (lam M) (arr s t))]
$[RecDer : {P:{n|Nat}{G|Con n}{M|Tm}{s|Ty}(Der G M s)->Type}
        ({n|Nat}{G:Con n}{i:Fin n}P (Var G i))
        -> ({n|Nat}{G|Con n}{s,t|Ty}{M,N|Tm}
                {d1:Der G M (arr s t)}(P d1)
                -> {d2:Der G N s}(P d2)
                -> P (App d1 d2))
        -> ({n|Nat}{G|Con n}{s,t|Ty}{M|Tm}
                {d:Der (v_cons s G) M t}(P d)
                -> P (Lam d))
        -> {n|Nat}{G|Con n}{M|Tm}{s|Ty}{d:Der G M s}P d];

[[P:{n|Nat}{G|Con n}{M|Tm}{s|Ty}(Der G M s)->Type]
 [Var_ : {n|Nat}{G:Con n}{i:Fin n}P (Var G i)]
 [App_ : {n|Nat}{G|Con n}{s,t|Ty}{M,N|Tm}
                {d1:Der G M (arr s t)}(P d1)
                -> {d2:Der G N s}(P d2)
                -> P (App d1 d2)]
 [Lam_ : {n|Nat}{G|Con n}{s,t|Ty}{M|Tm}
                {d:Der (v_cons s G) M t}(P d)
                -> P (Lam d)]
 [n:Nat][G:Con n][M,N|Tm][s,t:Ty][i:Fin n][d1:Der G M (arr s t)]
 [d2:Der G N s][d:Der (v_cons s G) M t]
   RecDer P Var_ App_ Lam_ (Var G i) ==> (Var_ G i)
|| RecDer P Var_ App_ Lam_ (App d1 d2) ==>
        App_ d1 (RecDer P Var_ App_ Lam_ d1)
             d2 (RecDer P Var_ App_ Lam_ d2)
|| RecDer P Var_ App_ Lam_ (Lam d) ==>
        Lam_ d (RecDer P Var_ App_ Lam_ d)];
```

```
[RecDerN[P:{n|Nat}(Con n)->Tm->Ty->Type] =
        RecDer [n|Nat][G|Con n][M|Tm][s|Ty][_:Der G M s]P G M s
: ({n|Nat}{G:Con n}{i:Fin n}P G (var (Fin2Nat i)) (v_nth i G))
->({n|Nat}{G|Con n}{s,t|Ty}{M,N|Tm}(Der G M (arr s t))->(P G M (arr s t))
        ->(Der G N s)->(P G N s)->P G (app M N) t)
->({n|Nat}{G|Con n}{s,t|Ty}{M|Tm}(Der (v_cons s G) M t)->(P (v_cons s G) M t)
        ->P G (lam M) (arr s t))
->{n|Nat}{G|Con n}{M|Tm}{s|Ty}(Der G M s)->P G M s
];
```

B.7. sn-simple.l

```
(*
**
** Strong normalization for simply typed lambda calculus
**
*)

(*
** Interpretation of types
*)

(* Int : Ty->(Tm->Set)

rec Int ty0 = SN
  | Int (arr s t) = ARR (Int s) (Int t)
*)

[Int = RecTyN SN
        ([_:Ty][Int_s:Tm->Set][_:Ty][Int_t:Tm->Set]ARR Int_s Int_t)];

Goal {t:Ty}CR (Int t);
Refine RecTy [t:Ty]CR (Int t);
Refine CR_SN;
intros;Refine CR_ARR;Immed;
Save CR_Int;

(*
** Interpretation of judgements
*)

(* Mod : {n|Nat}(Con n)->Tm->Ty->Set

rec Mod zero empty M T = (Int T) M
  | Mod (succ n) (v_cons S G) M T =
              {N:Tm}(Int S N)->(Mod G (subst0 M (rep_weak0 N n) T)

*)

[Mod[n|Nat][G:Con n][M:Tm][T:Ty] =
        RecVecNN
          ([M:Tm] (Int T) M)
          ([S:Ty][n|Nat][G:Con n][Mod_G:Tm->Prop]
                [M:Tm] {N:Tm}(Int S N)->(Mod_G (subst0 M (rep_weak0 N n))))
          G M];
```

```
(*
** Soundness
*)

Goal {n|Nat}{G|Con n}{M|Tm}{T|Ty}(Der G M T)->(Mod G M T);
intros;
Refine RecDerN [n|Nat][G:Con n][M|Tm][T|Ty]Mod G M T;
(*** Var ***)
intros;
Refine RecFin [n|Nat][i:Fin n]{G:Con n}Mod G (var (Fin2Nat i)) (v_nth i G);
(* var zero *)
Refine RecVecSucc [n:Nat][G:Con (succ n)]Mod G (var zero) (v_hd G);
Intros S n2 G2 N S_N;
Refine RecVec [n2|Nat][G2 : Vec Ty n2]Mod G2 (rep_weak0 N n2) S;
Intros; Refine S_N;
Intros;Refine EQ_rewrite' ? ([M:Tm]Mod l M S);
Next +1;Refine subst_weak_lem zero; Refine H1;
(* var (succ i) *)
intros;
Refine RecVec1 [G2 : Con (succ n2)]
        Mod G2 (var (Fin2Nat (f_succ m))) (v_nth (f_succ m) G2);
Intros;Refine H1;
(*** App ***)
intros;
Refine RecVec [n|Nat][G1:Con n]
        {M1,N:Tm}(Mod G1 M1 (arr s t))->(Mod G1 N s)->Mod G1 (app M1 N) t;
(* G = empty *)
intros;Refine H5;Refine H6;
(* G = S::G' *)
Intros S m G' _______;Refine H5;Refine H6;Refine H8;Refine H7;Refine H8;
Immed;
(*** Lam ***)
intros;
Refine RecVec [n1|Nat][G1:Con n1]
        {M1:Tm}(Mod (v_cons s G1) M1 t)->(Mod G1 (lam M1) (arr s t));
(* G = empty *)
intros;Refine Lam_sound;Refine CR_Int;Refine CR_Int;Refine H3;
(* G = S::G' *)
Intros;Refine H3;Intros;
Equiv
  Mod l (subst0 (subst (succ zero) M11 (rep_weak0 N n2)) (rep_weak0 N1 n2)) t;
Refine EQ_rewrite' ? ([M:Tm]Mod l M t);Next +1;
Refine subst_subst_lem zero; Refine H4; Immed;
(* END of CASES *)
```

```
Immed;Immed;
Save Int_sound;

(*
** Strong Normalisation
*)

Goal {n|Nat}{G|Con n}{M|Tm}{T|Ty}(Der G M T)->(SN M);
intros;
Claim (Mod G M T)->(SN M);Refine ?+1;Refine Int_sound;Immed;
Refine RecVec [n|Nat][G:Con n]{M:Tm}(Mod G M T)->(SN M);
(* G = empty *)
intros;Claim CR (Int T);Refine fst ?+1;Refine H1;Refine CR_Int;
(* G = S::G' *)
intros S m G' ___;
Refine SNvar m; Refine EQ_rewrite ? [X:Tm]SN (subst0 M1 X);
Next +1;Refine rep_weak0_lem;
Refine H1;Refine H2;Refine CR_nonEmpty;Refine CR_Int;
Save snorm;
```

B.8. f.l

```
(*
** System F
*)

(* mu[Ty:Nat->Set](t_var : {n|Nat}(Fin n) -> (Ty n),
                   arr   : {n|Nat}(Ty n) -> (Ty n) -> (Ty n),
                   pi    : {n|Nat}(Ty (succ n)) -> (Ty n) )
*)

$[Ty : Nat -> Set]
$[t_var : {n|Nat}(Fin n) -> (Ty n)]
$[arr : {n|Nat}(Ty n) -> (Ty n) -> (Ty n)]
$[pi : {n|Nat}(Ty (succ n)) -> (Ty n)];

$[RecTy : {P:{n|Nat}(Ty n)->Type}
           ({n|Nat}{p:Fin n} P (t_var p))
        -> ({n|Nat}{M:Ty n}(P M) -> {N:Ty n}(P N) -> (P (arr M N)))
        -> ({n|Nat}{M:Ty (succ n)}(P M) -> (P (pi M)))
        -> {n|Nat}{M:Ty n}P M];

[[P:{n|Nat}(Ty n)->Type]
 [f0:{n|Nat}{p:Fin n} P (t_var p)]
 [f1:{n|Nat}{M:Ty n}(P M) -> {N:Ty n}(P N) -> (P (arr M N))]
 [f2:{n|Nat}{M:Ty (succ n)}(P M) -> (P (pi M))]
 [n:Nat][p:Fin n][M,N:Ty n][O:Ty (succ n)]
   RecTy P f0 f1 f2 (t_var p) ==> f0 p
|| RecTy P f0 f1 f2 (arr M N) ==> f1 M (RecTy P f0 f1 f2 M)
                                    N (RecTy P f0 f1 f2 N)
|| RecTy P f0 f1 f2 (pi O) ==> f2 O (RecTy P f0 f1 f2 O)];

[RecTyN [P:Nat->Type] = RecTy ([n|Nat][_:Ty n]P n)
 : ({n|Nat}(Fin n)->P n)
 ->({n|Nat}(Ty n)->(P n)->(Ty n)->(P n)->P n)
 ->({n|Nat}(Ty (succ n))->(P (succ n))->P n)
 ->{n|Nat}(Ty n)->P n];

(*
** Special recursors for Ty
*)

Goal  {n:Nat}{P:{l:Nat}(Ty (add l n))->Set}
           ({l:Nat}{p:Fin (add l n)} P l (t_var p))
        -> ({l:Nat}{M:Ty (add l n)}(P l M)
```

```
                -> {N:Ty (add l n)}(P l N)
                -> (P l (arr M N)))
        -> ({l:Nat}{M:Ty (succ (add l n))}(P (succ l) M)
                -> (P l (pi M)))
        -> {l:Nat}{M:Ty (add l n)}P l M;
intros;
Claim {l'|Nat}{M':Ty l'}{l:Nat}{q:EQ l' (add l n)}
        P l (EQ_rewrite q Ty M');
Refine ?+1 M l (EQ_refl ?);
Refine RecTy [l'|Nat][M':Ty l']{l:Nat}{q:EQ l' (add l n)}
        P l (EQ_rewrite q Ty M');
(* t_var *)
intros;
Refine RecEQ1 (add l1 n)
        [n1:Nat][q:EQ n1 (add l1 n)]
        {p:Fin n1}P l1 (EQ_rewrite q Ty (t_var p));
Refine H;
(* arr *)
intros;
Refine RecEQ1 (add l1 n)
        [n1:Nat][q:EQ n1 (add l1 n)]
        {M1:Ty n1}(P l1 (EQ_rewrite q Ty M1))
         -> {N:Ty n1}(P l1 (EQ_rewrite q Ty N))
         -> (P l1 (EQ_rewrite q Ty (arr M1 N)));
Refine H1;
Refine H3;Refine H4;
(* pi *)
intros;
Refine RecEQ1 (add l1 n)
        [n1:Nat][q:EQ n1 (add l1 n)]
        {M1 : Ty (succ n1)}(P (succ l1) (EQ_rewrite (EQ_resp succ q) Ty M1))
        -> P l1 (EQ_rewrite q Ty (pi M1));
Refine H2;
Refine H3 (succ l1) (EQ_resp succ q);
Save RecTy1;

[RecTy1N [n:Nat][P:Nat->Set] = RecTy1 n ([l:Nat][_:Ty (add l n)]P l)
 : ({l:Nat}(Fin (add l n))->P l)
 ->({l:Nat}(Ty (add l n))->(P l)->(Ty (add l n))->(P l)->P l)
 ->({l:Nat}(Ty (succ (add l n)))->(P (succ l))->P l)
 ->{l:Nat}(Ty (add l n))->P l];

Goal  {n:Nat}{P:{l:Nat}(Ty (succ (add l n)))->Set}
           ({l:Nat}{p:Fin (succ (add l n))} P l (t_var p))
        -> ({l:Nat}{M:Ty (add (succ l) n)}(P l M)
```

```
                -> {N:Ty (add (succ l) n)}(P l N)
                -> (P l (arr M N)))
        -> ({l:Nat}{M:Ty (succ (succ (add l n)))}(P (succ l) M)
                -> (P l (pi M)))
        -> {l:Nat}{M:Ty (succ (add l n))}P l M;
intros;
Claim {l'|Nat}{M':Ty l'}{l:Nat}{q:EQ l' (succ (add l n))}
        P l (EQ_rewrite q Ty M');
Refine ?+1 M l (EQ_refl ?);
Refine RecTy [l'|Nat][M':Ty l']{l:Nat}{q:EQ l' (succ (add l n))}
        P l (EQ_rewrite q Ty M');
(* t_var *)
intros;
Refine RecEQ1 (succ (add l1 n))
        [n1:Nat][q:EQ n1 (succ (add l1 n))]
        {p:Fin n1}P l1 (EQ_rewrite q Ty (t_var p));
Refine H;
(* arr *)
intros;
Refine RecEQ1 (succ (add l1 n))
        [n1:Nat][q:EQ n1 (succ (add l1 n))]
        {M1:Ty n1}(P l1 (EQ_rewrite q Ty M1))
         -> {N:Ty n1}(P l1 (EQ_rewrite q Ty N))
         -> (P l1 (EQ_rewrite q Ty (arr M1 N)));
Refine H1;
Refine H3;Refine H4;
(* pi *)
intros;
Refine RecEQ1 (succ (add l1 n))
        [n1:Nat][q:EQ n1 (succ (add l1 n))]
        {M1 : Ty (succ n1)}(P (succ l1) (EQ_rewrite (EQ_resp succ q) Ty M1))
        -> P l1 (EQ_rewrite q Ty (pi M1));
Refine H2;
Refine H3 (succ l1) (EQ_resp succ q);
Save RecTy2;

[RecTy2N [n:Nat][P:Nat->Set] = RecTy2 n ([l:Nat][_:Ty (succ (add l n))]P l)
 : ({l:Nat}(Fin (succ (add l n)))->P l)
 ->({l:Nat}(Ty (add (succ l) n))->(P l)
 ->(Ty (add (succ l) n))->(P l)->P l)
 ->({l:Nat}(Ty (succ (succ (add l n))))->(P (succ l))->P l)
 ->{l:Nat}(Ty (succ (add l n)))->P l];

(*
** weakening & substitution for Types
```

```
*)

[n|Nat];

(* t_weak_var : {l:Nat}(Fin (add l n))->(Fin (succ (add l n)))

rec t_weak_var zero     i = (f_succ i)
  | t_weak_var (succ l) (f_zero (add l n)) = f_zero (succ (add l n))
  | t_weak_var (succ l) (f_succ i) = f_succ (t_weak_var l i)

*)

[t_weak_var =
  [P[l:Nat] = (Fin (add l n))->(Fin (succ (add l n)))]
  RecNat P (f_succ|n)
        ([l:Nat][t_weak_var_l:P l]
           RecFin1N (add l n)
              (f_zero (succ (add l n)))
              ([i:Fin (add l n)]f_succ (t_weak_var_l i)))
 : {l:Nat}(Fin (add l n))->(Fin (succ (add l n)))];

(* t_weak : {l|Nat}(Tm (add l n))->(Ty (succ (add l n)))

rec t_weak l (t_var i) = t_var (t_weak_var l i)
  | t_weak l (arr M N) = arr (t_weak l M) (t_weak l N)
  | t_weak l (abs M) = pi (t_weak (succ l) M)
*)

[t_weak =
  [P[l:Nat] = Ty (succ (add l n))]
  RecTy1N n P
        ([l:Nat][i:Fin (add l n)]t_var (t_weak_var l i))
        ([l:Nat][_:Ty (add l n)][t_weak_M:P l]
                [_:Ty (add l n)][t_weak_N:P l]arr t_weak_M t_weak_N)
        ([l:Nat][_:Ty (add (succ l) n)][t_weak_M:P (succ l)]pi t_weak_M)
  :  {l|Nat}(Ty (add l n))->Ty (succ (add l n))];

[t_weak0 = t_weak|zero
 : (Ty n) -> (Ty (succ n))];

DischargeKeep n;

(* t_subst_var : {l:Nat}(Fin (add (succ l) n))->(Ty n)->(Ty (add l n))

rec t_subst_var zero (f_zero (succ n)) = N
```

```
  | t_subst_var zero (f_succ i) = (t_var i)
  | t_subst_var (succ l) (f_zero (add (succ l) n)) =
        t_var (f_zero (add l n))
  | t_subst_var (succ l) (f_succ i) = t_weak0 (t_subst_var l i)
*)

[t_subst_var[l:Nat][i:Fin (add (succ l) n)][N:Ty n] =
  [P[l:Nat] = (Fin (add (succ l) n))->(Ty (add l n))]
  RecNat P
    (RecFin1N n N (t_var|n))
    ([l:Nat][t_subst_var_l:P l]
        RecFin1N (add (succ l) n)
           (t_var (f_zero (add l n)))
           ([i:Fin (add (succ l) n)]t_weak0 (t_subst_var_l i)))
   l i];

(* t_subst : {l:Nat}(Ty (add (succ l) n))->(Ty n)->(Ty (add l n))

rec t_subst l (var i) = t_subst_t_var l i
  | t_subst l (arr M1 M2) = arr (t_subst l M1) (t_subst l M2)
  | t_subst l (abs M) = abs (t_subst (succ l) M)
*)

[t_subst[l|Nat][M:Ty (add (succ l) n)][N:Ty n] =
  [P[l:Nat] = Ty (add l n)]
  RecTy2N n P
  ([l:Nat][i:Fin (succ (add l n))]t_subst_var l i N)
  ([l:Nat][_:Ty (succ (add l n))][t_subst_M1:P l]
          [_:Ty (succ (add l n))][t_subst_M2:P l]arr t_subst_M1 t_subst_M2)
  ([l:Nat][_:Ty (succ (add (succ l) n))][t_subst_M:P (succ l)]pi t_subst_M)
    l M
  : Ty (add l n)];

[t_subst0 = t_subst|zero : (Ty (succ n)) -> (Ty n) -> (Ty n)];

Discharge n;

(*
** Derivations
*)

[Con[m:Nat] = Vec (Ty m)];

(* mu[Der:{m,n|Nat}(Con m n)->Tm->(Ty m)->Set](
        Var : {m,n|Nat}{G:Con m n}{i:Fin n}
```

```
                Der G (var (Fin2Nat i)) (v_nth i G)
        App : {m,n|Nat}{G|Con m n}{s,t|Ty m}{M,N|Tm}
                (Der G M (arr s t))
                -> (Der G N s)
                -> (Der G (app M N) t)
        Lam : {m,n|Nat}{G|Con m n}{s,t|Ty m}{M|Tm}
                (Der (v_cons s G) M t)
                -> (Der G (lam M) (arr s t))
        Pi_e: {m,n|Nat}{G|Con m n}{s:Ty (succ m)}{t:Ty m}{M|Tm}
                (Der G M (pi s))
                -> (Der G M (t_subst0 s t))
        Pi_i: {m,n|Nat}{G|Con m n}{s:Ty (succ m)}{M|Tm}
                (Der (v_map t_weak0 G) M s)
                -> (Der G M (pi s)) )
*)

$[Der:{m,n|Nat}(Con m n)->Tm->(Ty m)->Set]
$[Var:  {m,n|Nat}{G:Con m n}{i:Fin n}
                Der G (var (Fin2Nat i)) (v_nth i G)]
$[App : {m,n|Nat}{G|Con m n}{s,t|Ty m}{M,N|Tm}
                (Der G M (arr s t))
                -> (Der G N s)
                -> (Der G (app M N) t)]
$[Lam : {m,n|Nat}{G|Con m n}{s,t|Ty m}{M|Tm}
                (Der (v_cons s G) M t)
                -> (Der G (lam M) (arr s t))]
$[Pi_e: {m,n|Nat}{G|Con m n}{s|Ty (succ m)}{M|Tm}
                (Der G M (pi s))
                -> {t:Ty m}(Der G M (t_subst0 s t))]
$[Pi_i: {m,n|Nat}{G|Con m n}{s|Ty (succ m)}{M|Tm}
                (Der (v_map (t_weak0|m) G) M s)
                -> (Der G M (pi s))];

$[RecDer : {P:{m,n|Nat}{G|Con m n}{M|Tm}{s|Ty m}(Der G M s)->Type}
        ({m,n|Nat}{G:Con m n}{i:Fin n}P (Var G i))
        -> ({m,n|Nat}{G|Con m n}{s,t|Ty m}{M,N|Tm}
                {d1:Der G M (arr s t)}(P d1)
                -> {d2:Der G N s}(P d2)
                -> P (App d1 d2))
        -> ({m,n|Nat}{G|Con m n}{s,t|Ty m}{M|Tm}
                {d:Der (v_cons s G) M t}(P d)
                -> P (Lam d))
        -> ({m,n|Nat}{G|Con m n}{s|Ty (succ m)}{M|Tm}
                {d:Der G M (pi s)}(P d)
                -> {t:Ty m}P (Pi_e d t))
```

```
        -> ({m,n|Nat}{G|Con m n}{s|Ty (succ m)}{M|Tm}
                {d:Der (v_map (t_weak0|m) G) M s}(P d)
                -> P (Pi_i d))
        -> {m,n|Nat}{G|Con m n}{M|Tm}{s|Ty m}{d:Der G M s}P d];

[RecDerN [P:{m,n|Nat}(Con m n)->Tm->(Ty m)->Type] =
  RecDer [m,n|Nat][G:Con m n][M:Tm][s:Ty m][_:Der G M s]P G M s
 : ({m,n|Nat}{G:Con m n}{i:Fin n}P G (var (Fin2Nat i)) (v_nth i G))
->({m,n|Nat}{G|Con m n}{s,t|Ty m}{M,N|Tm}(Der G M (arr s t))
        ->(P G M (arr s t))->(Der G N s)->(P G N s)->P G (app M N) t)
->({m,n|Nat}{G|Con m n}{s,t|Ty m}{M|Tm}(Der (v_cons s G) M t)
        ->(P (v_cons s G) M t)->P G (lam M) (arr s t))->
({m,n|Nat}{G|Con m n}{s|Ty (succ m)}{M|Tm}(Der G M (pi s))
        ->(P G M (pi s))->{t:Ty m}P G M (t_subst0 s t))
->({m,n|Nat}{G|Con m n}{s|Ty (succ m)}{M|Tm}(Der (v_map (t_weak0|m) G) M s)
        ->(P (v_map (t_weak0|m) G) M s)->P G M (pi s))
->{m,n|Nat}{G|Con m n}{M|Tm}{s|Ty m}(Der G M s)->P G M s];

[[P:{m,n|Nat}{G|Con m n}{M|Tm}{s|Ty m}(Der G M s)->Type]
 [Var_:{m,n|Nat}{G:Con m n}{i:Fin n}P (Var G i)]
 [App_:{m,n|Nat}{G|Con m n}{s,t|Ty m}{M,N|Tm}
                {d1:Der G M (arr s t)}(P d1)
                -> {d2:Der G N s}(P d2)
                -> P (App d1 d2)]
 [Lam_:{m,n|Nat}{G|Con m n}{s,t|Ty m}{M|Tm}
                {d:Der (v_cons s G) M t}(P d)
                -> P (Lam d)]
 [Pi_e_:{m,n|Nat}{G|Con m n}{s|Ty (succ m)}{M|Tm}
                {d:Der G M (pi s)}(P d)
                -> {t:Ty m}P (Pi_e d t)]
 [Pi_i_:{m,n|Nat}{G|Con m n}{s|Ty (succ m)}{M|Tm}
                {d:Der (v_map (t_weak0|m) G) M s}(P d)
                -> P (Pi_i d)]
 [m,n|Nat][G:Con m n][i:Fin n][s,t:Ty m][s':Ty (succ m)][M,N|Tm]
 [d1:Der G M (arr s t)][d2:Der G N s][d3:Der (v_cons s G) M t]
 [d4:Der G M (pi s')][d5:Der (v_map (t_weak0|m) G) M s']
   RecDer P Var_ App_ Lam_ Pi_e_ Pi_i_ (Var G i) ==> Var_ G i
|| RecDer P Var_ App_ Lam_ Pi_e_ Pi_i_ (App d1 d2) ==>
        App_ d1 (RecDer P Var_ App_ Lam_ Pi_e_ Pi_i_ d1)
             d2 (RecDer P Var_ App_ Lam_ Pi_e_ Pi_i_ d2)
|| RecDer P Var_ App_ Lam_ Pi_e_ Pi_i_ (Lam d3) ==>
        Lam_ d3 (RecDer P Var_ App_ Lam_ Pi_e_ Pi_i_ d3)
|| RecDer P Var_ App_ Lam_ Pi_e_ Pi_i_ (Pi_e d4 t) ==>
        Pi_e_ d4 (RecDer P Var_ App_ Lam_ Pi_e_ Pi_i_ d4) t
|| RecDer P Var_ App_ Lam_ Pi_e_ Pi_i_ (Pi_i d5) ==>
```

```
        Pi_i_ d5 (RecDer P Var_ App_ Lam_ Pi_e_ Pi_i_ d5)
];
```

B.9. sn-f.l

```
(*
**
** Strong normalization for System F
**
*)

(*
** Interpretation of types
*)

(* Int : {m|Nat}(Ty m)->(VEC (Tm->Set) m)->(Tm->Set)

rec Int m (t_var i) = [v:VEC (Tm->Set) m]V_nth i v
  | Int m (arr s t) = [v:VEC (Tm->Set) m]ARR (Int s v) (Int t v)
  | Int m (pi t)    = [v:VEC (Tm->Set) m]PI [P:Tm->Set]Int t (V_cons P v)
*)

[Int = RecTyN ([m:Nat](VEC (Tm->Set) m)->(Tm->Set))
        ([m|Nat][i:Fin m]
         [v:VEC (Tm->Set) m]V_nth i v)
        ([m|Nat][s:Ty m][Int_s:(VEC (Tm->Set) m)->(Tm->Set)]
                [t:Ty m][Int_t:(VEC (Tm->Set) m)->(Tm->Set)]
         [v:VEC (Tm->Set) m]ARR (Int_s v) (Int_t v))
        ([m|Nat][t:Ty (succ m)][Int_t:(VEC (Tm->Set) (succ m))->(Tm->Set)]
         [v:VEC (Tm->Set) m]PI [P:Tm->Set]Int_t (V_cons P v))];

Goal {m|Nat}{t:Ty m}{v:VEC (Tm->Set) m}
        ({i:Fin m}CR (V_nth i v))
        -> (CR (Int t v));
Refine RecTy ([m|Nat][t:Ty m]{v:VEC (Tm->Set) m}
        ({i:Fin m}CR (V_nth i v))
        -> (CR (Int t v)));
(* t_var *) intros;Refine H;
(* arr s t *) intros;Refine CR_ARR;Refine H;Refine H2;
Refine H1;Refine H2;
(* t_pi t *) intros;Refine CR_PI;intros;Refine H;
Refine RecFin1 [i:Fin (succ n)]CR (V_nth i (V_cons P v));
Refine H2;Refine H1;
Save CR_Int;

(*
** Correctness of weakening and substitution
*)
```

```
Goal {m,l:Nat}{t:Ty (add l m)}{v1:VEC (Tm->Set) l}{v2:VEC (Tm->Set) m}
        {M:Tm}{P:Tm->Set}
        iff (Int t (V_append v1 v2) M)
            (Int (t_weak t) (V_insert v1 P v2) M);
intros m;
Refine RecTy1 m [l:Nat][t:Ty (add l m)]
        {v1:VEC (Tm->Set) l}{v2:VEC (Tm->Set) m}{M:Tm}{P:Tm->Set}
        iff (Int t (V_append v1 v2) M)
            (Int (t_weak t) (V_insert v1 P v2) M);
(*** t_var ***)
intros;
Refine RecVEC [l|Nat][v1:VEC (Tm->Set) l]
        {p:Fin (add l m)}
           iff (Int (t_var p) (V_append v1 v2) M) (Int (t_weak (t_var p))
               (V_insert v1 P v2) M);
(** v1 = V_nil **)
intros;Refine iff_refl;
(** V_cons R v1 **)
intros R l v IH;
Refine RecFin1 [p:Fin (add (succ l1) m)]
        iff (Int (t_var p) (V_append (V_cons R v) v2) M)
            (Int (t_weak (t_var p)) (V_insert (V_cons R v) P v2) M);
(* p=f_zero *) Refine iff_refl;
(* f_succ p1 *) intros p1;Refine IH;
(*** arr ***)
intros l s IH_s t IH_t v1 v2 M1 P;andI;
(** fst **)
Intros;Refine fst (IH_t ? ? ? ?);Refine H;Refine snd (IH_s ? ? ? ?);Next +1;
Refine H1;
(** snd **)
Intros;Refine snd (IH_t ? ? ? ?);Next +1;Refine H;Refine fst (IH_s ? ? ? ?);
Refine H1;
(*** pi ***)
intros l s IH_s ____;andI;
(** fst **)
intros;Equiv Int (pi (t_weak|m|(succ l) s)) (V_insert v1 P v2) M;
Intros;Equiv Int (t_weak|m|(succ l) s) (V_insert (V_cons P1 v1) P v2) M;
Refine fst (IH_s ? ? ? ?);Refine H;Refine H1;
(** snd **)
Intros;Equiv Int s (V_cons P1 (V_append v1 v2)) M;
Refine snd (IH_s (V_cons P1 v1) ? ? ?);Next +1;Refine H;Refine H1;
Save t_weak_lem;

Goal {m|Nat}{t:Ty m}{v:VEC (Tm->Set) m}{M:Tm}
```

```
                (Int t v M)->
                {P:Tm->Set}Int (t_weak0 t) (V_cons P v) M;
intros;Refine fst (t_weak_lem ? ? ? (V_nil (Tm->Set)) ? ? ?);Immed;
Save t_weak_ok;

Goal {m,l:Nat}{s:Ty (succ (add l m))}{t:Ty m}
        {v1:VEC (Tm->Set) l}{v2:VEC (Tm->Set) m}{M:Tm}
        iff (Int s (V_insert v1 (Int t v2) v2) M)
            (Int (t_subst s t) (V_append v1 v2) M);
intros m;
Refine RecTy2 m [l:Nat][s:Ty (succ (add l m))]
        {t:Ty m}{v1:VEC (Tm->Set) l}{v2:VEC (Tm->Set) m}{M:Tm}
        iff (Int s (V_insert v1 (Int t v2) v2) M)
            (Int (t_subst s t) (V_append v1 v2) M);
(*** t_var ***)
intros;
Refine RecVEC [l|Nat][v1:VEC (Tm->Set) l]
        {p:Fin (succ (add l m))}
         iff (Int (t_var p) (V_insert v1 (Int t v2) v2) M)
             (Int (t_subst (t_var p) t) (V_append v1 v2) M);
(** v1 = V_nil **)
Refine RecFin1 [p:Fin (succ (add zero m))]
        iff (Int (t_var p) (V_cons (Int t v2) v2) M)
            (Int (t_subst (t_var p) t) v2 M);
(* p =f_zero *) Refine iff_refl;
(* f_succ p *) intros; Refine iff_refl;
(** V_cons R v **)
intros R l v IH;
Refine RecFin1 [p:Fin (succ (add (succ l1) m))]
        iff (Int (t_var p) (V_insert (V_cons R v) (Int t v2) v2) M)
            (Int (t_subst (t_var p) t) (V_append (V_cons R v) v2) M);
(* p=f_zero *) Refine iff_refl;
(* f_succ p1 *) intros p1;
Refine iff_trans;Next +2;Refine t_weak_lem ? ? ? (V_nil (Tm->Set)) ? ? ?;
Refine IH;
(*** arr ***)
intros l s IH_s t IH_t v1 v2 M1 P;andI;
(** fst **)
Intros;Refine fst (IH_t ? ? ? ?);Refine H;Refine snd (IH_s ? ? ? ?);Refine H1;
(** snd **)
Intros;Refine snd (IH_t ? ? ? ?);Refine H;Refine fst (IH_s ? ? ? ?);
Refine H1;
(*** pi ***)
intros l s IH_s ____;andI;
(** fst **)
```

```
Intros;Equiv Int (t_subst|m|(succ l) s t) (V_cons P (V_append v1 v2)) M;
Refine fst (IH_s ? (V_cons P v1) ? ?);;
Refine H;Refine H1;
(** snd **)
Intros;Refine snd (IH_s ? (V_cons P v1) ? ?);Refine H;Refine H1;
Save t_subst_lem;

Goal {m|Nat}{s:Ty (succ m)}{t:Ty m}{v:VEC (Tm->Set) m}{M:Tm}
          (Int s (V_cons (Int t v) v) M)
          -> (Int (t_subst0 s t) v M);
intros;Refine fst (t_subst_lem ? ? ? ? (V_nil Tm->Set) ? ?);Immed;
Save t_subst_ok;

(*
** Interpretation of judgements
*)

(* Mod : {m,n|Nat}(Con m n)->Tm->(Ty m)->(VEC (Tm->Set) m)->Set

rec Mod m zero     empty        M T v = Int T v M
  | Mod m (succ n) (v_cons S G) M T v =
               {N:Tm}(Int S v N)->(Mod G (subst0 M (rep_weak0 N n)) T v)

*)

[Mod[m,n|Nat][G:Con m n][M:Tm][T:Ty m][v:VEC (Tm->Set) m] =
        RecVecNN
          ([M:Tm]Int T v M)
          ([S:Ty m][n|Nat][G:Con m n][Mod_G:Tm->Set]
                [M:Tm]{N:Tm}(Int S v N)
                        ->(Mod_G (subst0 M (rep_weak0 N n))))
          G M];

*)
(*
** Soundness
*)

Goal {m,n|Nat}{G|Con m n}{M|Tm}{T|Ty m}(Der G M T)->
        {v:VEC (Tm->Set) m}({i:Fin m}CR (V_nth i v))
        -> (Mod G M T v);
intros;
Refine RecDerN [m,n|Nat][G:Con m n][M|Tm][T|Ty m]
        {v:VEC (Tm->Set) m}({i:Fin m}CR (V_nth i v)) -> (Mod G M T v);
(*** Var ***)
```

```
intros;
Refine RecFin [n|Nat][i:Fin n]{G1:Con m1 n}
        Mod G1 (var (Fin2Nat i)) (v_nth i G1) v1;
(* var zero *)
Refine RecVecSucc [n:Nat][G1:Con m1 (succ n)]
        Mod G1 (var zero) (v_hd G1) v1;
Intros S n2 G2 N S_N;
Refine RecVec [n2|Nat][G2:Con m1 n2]Mod G2 (rep_weak0 N n2) S v1;
Refine S_N;
Intros;Refine EQ_rewrite' ? ([M:Tm]Mod l M S v1);
Next +1;Refine subst_weak_lem zero;Refine H3;
(* var (succ i) *)
intros;
Refine RecVec1 [G2 : Con m1 (succ n2)]
        Mod G2 (var (Fin2Nat (f_succ m2))) (v_nth (f_succ m2) G2) v1;
Intros;Refine H3;
(*** App ***)
intros;
Refine RecVec [n|Nat][G1:Con m1 n]
        {M1,N:Tm}(Mod G1 M1 (arr s t) v1)->(Mod G1 N s v1)
                ->Mod G1 (app M1 N) t v1;
(* G = empty *)
intros;Refine H7;Refine H8;
(* G = S::G' *)
Intros;Refine H7;Refine H8;Refine H10;Refine H9;Refine H10;Refine H3;Refine H6;
Refine H5;Refine H6;
(*** Lam ***)
intros;
Refine RecVec [n1|Nat][G1:Con m1 n1]
        {M1:Tm}(Mod (v_cons s G1) M1 t v1)->(Mod G1 (lam M1) (arr s t) v1);
(* G = empty *)
intros;Refine Lam_sound;Refine CR_Int;Refine H4;Refine CR_Int;Refine H4;
Refine H5;
(* G = S::G' *)
Intros;Refine H5;Intros;Refine EQ_rewrite' ? ([M:Tm]Mod l M t v1);Next +1;
Refine subst_subst_lem zero; Refine H6; Immed;Refine H3;Refine H4;
(*** Pi_e ***)
intros;
Refine RecVec [n1|Nat][G1:Con m1 n1]
        {M1:Tm}(Mod G1 M1 (pi s) v1) -> (Mod G1 M1 (t_subst0 s t) v1);
(* G = empty *)
intros;Refine t_subst_ok;Refine H5;Refine CR_Int;Refine H4;
(* G = S::G' *)
Intros;Refine H5;Refine H6;Refine H7;
(* *) Refine H3;Refine H4;
```

```
(*** Pi_i ***)
intros;
Refine RecVec [n1|Nat][G1:Con m1 n1]
        {M1:Tm}
        ({P:Tm->Set}(CR P) ->(Mod (v_map (t_weak0|m1) G1) M1 s (V_cons P v1)))
        -> (Mod G1 M1 (pi s) v1);
(* G = empty *)
intros;Refine H5;
(* G = S::G' *)
Intros;Refine H5;intros;Refine H6;Refine H8;Refine t_weak_ok;Refine H7;
(* *) intros;Refine H3;
Refine RecFin1[i:Fin (succ m1)]CR (V_nth i (V_cons P v1));
Refine H5;
Refine H4;
(*** END of CASES ***)
Immed;
Save Int_Sound;

(*
** Strong Normalisation
*)

[SN_m = RecNat ([m:Nat]VEC (Tm->Set) m)
        (V_nil (Tm->Set))
        ([m:Nat][v:VEC (Tm->Set) m]V_cons SN v)];

Goal {m:Nat}{i:Fin m}CR (V_nth i (SN_m m));
Refine RecNat [m:Nat]{i:Fin m}CR (V_nth i (SN_m m));
Refine RecFinZero [i:Fin zero]CR (V_nth i (SN_m zero));
intros __;Refine RecFin1 [i:Fin (succ n)]CR (V_nth i (SN_m (succ n)));
Refine CR_SN;Refine H;
Save CR_SN_m;

Goal {m,n|Nat}{G|Con m n}{M|Tm}{T|Ty m}(Der G M T)->(SN M);
intros;
Claim (Mod G M T (SN_m m))->(SN M);Refine ?+1;Refine Int_Sound;
Refine H;Refine CR_SN_m;
Refine RecVec [n|Nat][G:Con m n]{M:Tm}(Mod G M T (SN_m m))->SN M;
(* G = empty *)
intros;Claim CR (Int T (SN_m m));Refine fst ?+1;Refine H1;Refine CR_Int;
Refine CR_SN_m;
(*  G = S::G' *)
intros S n1 G' ___;
Refine SNvar n1; Refine EQ_rewrite ? [X:Tm]SN (subst0 M1 X);
Next +1;Refine rep_weak0_lem;
```

```
Refine H1;Refine H2;Refine CR_nonEmpty;Refine CR_Int;Refine CR_SN_m;
Save snorm;
```

Bibliography

[ACCL90] M. Abadi, L. Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. In *ACM Conference on Principles of Programming Languages, San Francisco*, 1990.

[Alt90] Thorsten Altenkirch. Impredicative representations of categorical datatypes, thesis proposal, October 1990.

[Alt93a] Thorsten Altenkirch. A formalization of the strong normalization proof for System F in LEGO. In J.F. Groote M. Bezem, editor, *Typed Lambda Calculi and Applications*, LNCS 664, 1993.

[Alt93b] Thorsten Altenkirch. Yet another strong normalization proof for the Calculus of Constructions. In *Proceedings of El Vintermöte*, number 73 in Programming Methodology Group Reports. Chalmers University, Göteborg, 1993.

[Bar84] H.P. Barendregt. *The Lambda Calculus - Its Syntax and Semantics (Revised Edition)*. Studies in Logic and the Foundations of Mathematics. North Holland, 1984.

[Bar92] H.P. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science, Vol. 2*, pages 118 – 310. Oxford University Press, 1992.

[BCMS89] Roland Backhouse, Paul Chisholm, Grant Malcolm, and Erik Saaman. Do-it-yourself type theory. *Formal Aspects of Computing*, 1:19–84, 1989.

[Ber91] Stefano Berardi. Girard's normalisation proof in LEGO. unpublished draft, 1991.

[BS91] Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In *Proceedings of the Sixth Annual Symposium on Logic in Computer Science*, pages 202 – 211, 1991.

[CD93] Thierry Coquand and Peter Dybjer. Normalization algorithms, mechanical normalization and intuitionistic model constructions. In *Proceedings of El Vintermöte*, number 73 in Programming Methodology Group Reports. Chalmers University, Göteborg, 1993.

[CG90] Thierry Coquand and Jean Gallier. A proof of strong normalization for the theory of constructions using a Kripke-like interpretation. Informal Proceedings of the First Annual Workshop on Logical Frameworks, Antibes, 1990.

[CH88] Thierry Coquand and Gerard Huet. The calculus of constructions. *Information and Computation*, 76:95 – 120, 1988.

[Coq85] Thierry Coquand. *Une théorie des constructions.* PhD thesis, Université Paris VII, 1985.

[Coq90] Thierry Coquand. Metamathematical investigations of a calculus of constructions. In P.G. Oddifreddi, editor, *Logic and Computer Science*, pages 91 – 122. Academic Press, 1990.

[Coq91] Thierry Coquand. An algorithm for deciding conversion in type theory. In Huet and Plotkin, editors, *Logical Frameworks.* Cambridge University Press, 1991.

[Coq92a] Catarina Coquand. A proof of normalization for simply typed lambda calculus written in ALF. In *Workshop on Logical Frameworks.* BRA Types, 1992. Preliminary Proceedings.

[Coq92b] Thierry Coquand. Pattern matching with dependent types. In *Workshop on Logical Frameworks*, 1992. Preliminary Proceedings.

[CP89] Thierry Coquand and Christine Paulin. Inductively defined types. In Peter Dybjer et al., editors, *Proceedings of the Workshop on Programming Logic*, 1989. Preliminary version.

[D+91] Gilles Dowek et al. *The Coq Proof Assistant User's Guide.* INRIA-Rocquencourt — CNRS-ENS Lyon, 1991. Version 5.6.

[dB72] N.G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indag. Math.*, 34, 1972.

[dB80] N.G. de Bruijn. A survey of the project AUTOMATH. In J.P.Seldin and J.R.Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 589 – 606. Academic Press, 1980.

[Dow93] Gilles Dowek. Talk given at the BRA Types workshop in Nijmegen, 1993.

[Dyb91] Peter Dybjer. Inductive sets and families in Martin-Löf's type theory and their set-theoretic semantics. Technical Report Report 62, Programming Methodology Group, Chalmers University, 1991.

[Dyb92a] Peter Dybjer. Inductive families, 1992. Draft, to appear in FACS.

[Dyb92b] Peter Dybjer. Universes and a general notion of simultanous inductive-recursive definition in type theory. In *Workshop on Logical Frameworks.* BRA Types, 1992. Preliminary Proceedings.

[Ehr89] Thomas Ehrhard. Dictoses. In D.H. Pitt et al., editors, *Category Theory and Computer Science*, pages 213–223. Springer, 1989. LNCS 389.

[Fu92] Yuxi Fu. Topics in type theory. Technical report, Dep. of Computer Science; University of Manchester, 1992. Revised version of PhD thesis.

[Geu89] J.H. Geuvers. Theory of constructions is not conservative over higher order logic, 1989.

[Geu93] Herman Geuvers. *Logics and Type Systems.* PhD thesis, Katholieke Universiteit Nijmegen, 1993.

[GLT89] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types.* Cambridge University Press, 1989.

[Gog93] Healfdene Goguen. *Strong Normalization for a Type Theory with Inductive Data Types.* PhD thesis, University of Edinburgh, 1993. forthcoming.

[Hag87] Tatsuya Hagino. *A Categorical Programming Language.* PhD thesis, University of Edinburgh, September 1987.

[HO93] J.M.E. Hyland and C.-H. L. Ong. Modified realizability toposes and strong normalization proofs. In J.F. Groote M. Bezem, editor, *Typed Lambda Calculi and Applications*, LNCS 664, 1993.

[Hof92] Martin Hofmann. Formal development of functional programs in type theory — a case study. LFCS Report ECS-LFCS-92-228, University of Edinburgh, 1992.

[Hof93a] Martin Hofmann. Elimination of extensionality and quotient types in Martin-Löf type theory. Talk given at the BRA Types workshop in Nijmegen, 1993.

[Hof93b] Martin Hofmann. Implementing continutations in type theory. Experiment on the implementation of a breadth-first-search algorithm in LEGO, 1993.

[HP89] J.M.E. Hyland and A.M. Pitts. The theory of constructions: categorical semantics and topos-theoretic models. In J. Gray and A. Scedrov, editors, *Categories in Computer Science and Logic*, pages 137 – 199, 1989.

[Hue80] Gérard Huet. Confluent reductions. *JACM*, 27(4):797–821, 1980.

[Hue93] Gérard Huet. Residual theory in λ-calculus: A complete Gallina development. Paper presented at a workshop of the BRA Types in Turino, 1993.

[Jac91] Bart Jacobs. *Categorical Type Theory.* PhD thesis, Katholieke Universiteit Nijmegen, 1991.

[Jon93] Claire Jones. Datatypes in LEGO. Extension of the lego implementation by a tactic for inductive types, 1993.

[Llo84] J.W. Lloyd. *Foundations of Logic Programming.* Springer Verlag, 1984.

[LNS82] J.-L. Lassez, V.L. Nguyen, and E.A. Sonneberg. Fixed point theorems and semantics: A folk tale. *Information Processing Letters*, 14(3), 1982.

[LP92] Zhaohui Luo and Robert Pollack. The LEGO proof development system: A user's manual. LFCS report ECS-LFCS-92-211, University of Edinburgh, 1992.

[Luo90] Zhaohui Luo. *An Extended Calculus of Constructions.* PhD thesis, University of Edinburgh, 1990.

[Luo92] Zhaohui Luo. A unifying theory of dependent types: the schematic approach. Technical Report ECS-LFCS-92-202, LFCS, 1992.

[Mag92] Lena Magnusson. The new implementation of ALF. In *Workshop on Logical Frameworks*, 1992. Preliminary Proceedings.

[Mar71] Per Martin-Löf. A theory of types. Technical report, University of Stockholm, 1971.

[Mar75] Per Martin-Löf. An intutionistic theory of types: Predicative part. In *Logic Colloqioum 1973*, pages 73 – 118, 1975.

[Mar84] Per Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, 1984.

[Men88] N.P. Mendler. *Inductive Definition in Type Theory.* PhD thesis, Cornell University, 1988.

[Mit90] John Mitchell. Type systems for programming languages. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science.* North Holland, 1990.

[MP93] James McKinna and Robert Pollack. Pure type systems formalized. In J.F. Groote M. Bezem, editor, *Typed Lambda Calculi and Applications*, LNCS 664, 1993.

[NPS90] Bengt Nordström, Kent Petersson, and Jan Schmith. *Programming in Martin-Löf Type Theory.* Oxford University Press, 1990.

[Ore92] Christian-Emil Ore. The extended calculus of constructions with inductive types. *Information and Computation*, 1992.

[Pfe92] Frank Pfenning. On the undecidability of partial polymorphic type reconstruction. Technical Report CMU-CS-92-105, Carnegie Mellon University, January 1992.

[PM93a] Christine Paulin-Mohring. Extracting F_ω's programs from proofs in the calculus of constructions. In *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, 1993.

[PM93b] Christine Paulin-Mohring. Inductive definitions in the system Coq; rules and properties. In J.F. Groote M. Bezem, editor, *Typed Lambda Calculi and Applications*, LNCS 664, 1993.

[PMW93] Christine Paulin-Mohring and Benjamin Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15:607 – 640, 1993.

[Pol92] Randy Pollack. Typechecking in pure type systems. In *Workshop on Types for Proofs and Programs, Båstad, Sweden*, 1992. Preliminary Proceedings.

[Rey84] John Reynolds. Polymorphism is not set-theoretic. In *Proceedings of the International Symposium on Semantics of Data Types*, 1984. LNCS 173.

[Rit92] Eike Ritter. *Categorical Abstract Machines for Higher-Order Typed Lambda Calculi.* PhD thesis, University of Cambridge, 1992.

[Str89] Thomas Streicher. *Correctness and Completeness of a Categorical Semantics of the Calculus of Constructions.* PhD thesis, Universität Passau, Passau, West Germany, June 1989.

[Str91] Thomas Streicher. *Semantics of Type Theory.* Birkhäuser, 1991.

[Tar55] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math*, 5:285 – 309, 1955.

[Tro87] A. S. Troelstra. On the syntax of Martin-Löf's type theories. *Theoretical Computer Science*, 51:1–26, 1987.

[Wad91] Philip Wadler. Is there a use for linear logic. In *Conference on Partial Evaluation and Semantics-Based Program Manipulation.* ACM Press, 1991.

[Wer92] Benjamin Werner. A normalization proof for an impredicative type system with large eliminations over integers. In *Workshop on Logical Frameworks.* BRA Types, 1992. Preliminary Proceedings.

Index of Symbols

The following index contains symbols, their description, letters used for variables (where appropriate) and page numbers. Symbols which have more then one page number associated to them are overloaded.

$\_(\_)$	(Partial) set-theoretic application	48
$\Gamma(i)$	Context projection	23
$(\_,\_)$	Pairing	48
$\_[\_]\_$	Substitution	23,32
$[\![\_]\!]$	Interpretation	48
$[\![\_]\!]^{\mathcal{L}}$	Interpretation in an LF structure	52
$[\![\_]\!]^{\mathcal{C}}$	Interpretation in a CC-structure	54
$\_[\_]$	(Dependent) composition	51
$\_^{+}\_$	Weakening	23,32
$\_^{+}\_$	Semantic weakening	51
$\|\_\|$	Stripping	32,40
$\|\_\|$	Length of a sequence	48
$\|\Gamma\|$	Length of a context	23
$\Gamma.\sigma$	Context comprehension	22
A/R	Quotient	61
$\tilde{R}$	$= D/R$	79
$\vec{A}$	Notation for sequences	48
$\hat{p}$	$= \vartheta_{\mathcal{D}}^{-1}(p)$	79
$\forall\sigma.M$	Small Π-type	22
$\overline{X}$	Extension (Universe)	48
$\overline{X}$	Extension of a D-set	63

$\perp^R$			90
$\bullet$	Empty context		22
$\cong$	Kleene equality		47
$\underset{\sim}{\in}$	Partial $\in$		47
ϵ	Empty Sequence		48
$\rhd_X$	(One-Step) Reduction (general)		30
$\rhd_X^+$	Transitive closure of $\rhd_X$		30
$\rhd_X^*$	Transitive, reflexive closure of $\rhd_X$		30
$\approx_X$	Transitive, reflexive, symmetric closure of $\rhd_X$		30
$\rhd_X^{\mathrm{SN}}$	Restriction of $\rhd_X$		31
$\rhd\ (\rhd_\eta)$	Curry $\beta(\eta)$-reduction		32,87
$\rhd_{\mathrm{l}}\ (\rhd_{\mathrm{l}\eta})$	Loose reduction		44
$\rhd_{\mathrm{t}}\ (\rhd_{\mathrm{t}\eta})$	Tight reduction		34
$\rhd_{\mathbf{whd}}$	Weak head reduction		68,87
$\mathbf{1}$	Component of an LF-structure		51
$\vdash \Gamma$	Context validity		25
$\Gamma \vdash \sigma$	Type validity		25
$\Gamma \vdash \sigma \simeq \tau$	Type equality		25
$\Gamma \vdash M : \sigma$	Typing		25
$\Gamma \vdash M \simeq N : \sigma$	Equality		25
$\ldots \vdash^{\approx} \ldots$	Conversion presentation		45
$\mathrm{app}^{\sigma,\tau}(M,N)$	Typed application		22
$\mathrm{blow}(C)$	Blowing up		35
C_T^i	Constructors		120
Cn	Constructions	C, D	22
Co	(Pre)contexts	Γ, Δ	22
$\mathcal{D}$	Realizability interpretation (LF)		63
$\mathfrak{D}$	Universe of D-sets		63
$\mathcal{D}^+$	Realizability interpretation (CC)		65
$\mathrm{dom}(R)$	Domain of a PER		61
$\mathrm{d}_{\mathbf{leaf}}, \mathrm{d}_{\mathbf{sup}}, \mathrm{d}_{\mathrm{R}}$	Realizer for tree-semantics		80

$\mathrm{El}(A)$	Type constructor		22
$\vec{\mathrm{El}}(\vec{A})$	Telescope notation		119
$\mathrm{EL}, \mathrm{EL}^{-1}$	Component of a CC-structure		53
Λ	Curry terms	M, N	32,87
$\lambda(\_)$	Currying		51
$\lambda\sigma(M)^\tau$	Typed lambda abstraction		22
$\vec{\lambda}\Theta.M$	Iterated λ-abstraction		119
$\overline{\lambda}$	Schönfinkel abstraction		62
$\mathfrak{LAM}$	Universe of Λ-sets		70
$\mathfrak{LAM}^*$	Universe of Λ^*-sets		70
$\mathrm{LEAF}_{\mathcal{D}}$	Semantics of leaf (D-Sets)		84
$\mathrm{LEAF}_{\mathcal{SAT}}$	Semantics of leaf (saturated Λ-Sets)		90
Mu	Mu specification	T, U	120
μT	Type constructor		120
$\mu_{\mathrm{N}} T$			121
$\mathfrak{M}_{\mathcal{D}}$	Modest D-sets		64
$\mathfrak{M}_{\Lambda}^{\mathrm{Sat}}$	Modest saturated Λ sets		73
ω	natural numbers	i, j, k, m, n	
P, P_1, P_2	Pairing and projection combinators		62
π_1, π_2	Projections		48
$\mathrm{PER}(\_)$	Partial Equivalence Relations	R, S	61
$\Phi_{\mathcal{D}}$	Monotone operator (D-Sets)		82
$\Phi_{\mathcal{SAT}}$	Monotone operator (saturated Λ-Sets)		89
$\Psi_{\mathcal{D}}$	Approximations (D-Sets)		86
$\Psi_{\mathcal{SAT}}$	Approximations (saturated Λ-Sets)		91
$\mathrm{pr}_{\_}$	Projection function		51
$\Pi a \in A.B_a$	Set-theoretic Π		48
$\Pi(\_,\_)$	Component of an LF-structure		51
$\Pi\sigma.\tau$	Π-type		22
$\vec{\Pi}\Theta\sigma$	Iterated Π-type		119
R_T^σ	Recursor		120

$\Sigma a \in A.B_a$	Set-theoretic Σ		48
$\mathfrak{SAT}$	Universe of saturated Λ-sets		70
$\mathfrak{SAT}^*$	Universe of saturated Λ^*-sets		71
$\mathcal{SAT}$	Saturated Λ-set interpretation (LF)		72
$\mathcal{SAT}^+$	Saturated Λ-set interpretation (CC)		73
Sect	Component of an LF-structure		51
Set	Type of (small) sets	$A, B, C, \ldots$	22
$\vec{\text{Set}}$	Telescopes	$\vec{A}, \vec{B}, \vec{C}, \ldots$	118
SET	Component of a CC-structure		53
SN_X	Strongly normalizing		30
$\text{SUP}_{\mathcal{D}}$	Semantics of sup (D-Sets)		84
$\text{SUP}_{\mathcal{SAT}}$	Semantics of sup (saturated Λ-Sets)		90
$\vartheta_X(x)$	Component of a CC-structure		53
$\tilde{\vartheta}_X(x)$			54
$\Theta(X)$			54
Tm	(Pre)terms	M, N	22
$\text{Tm}^{\text{Church}}$	Church terms	M, N	40
Tm_{PCA}	PCA terms		61
TR	Type reconstruction		42
$\text{TREE}_{\mathcal{D}}$	D-Set interpretation of Tree		84
Ty	(Pre)types	σ, τ, ρ	22
$\mathfrak{U}$	A universe		48
$\mathfrak{U}_{\text{Co}}, \mathfrak{U}_{\text{Ty}}$	Components of LF-structures		51
Void	Void terms		69,87
whnf	Weak head normal form		88