Summer 2014

The semantic analysis of advanced programming languages

Harley D. Eades III
University of Iowa

THE SEMANTIC ANALYSIS OF ADVANCED PROGRAMMING LANGUAGES

by

Harley D. Eades III

A thesis submitted in partial fulfillment of the
requirements for the Doctor of Philosophy
degree in Computer Science
in the Graduate College of
The University of Iowa

August 2014

Thesis Supervisor: Associate Professor Aaron Stump

Graduate College
The University of Iowa
Iowa City, Iowa

CERTIFICATE OF APPROVAL

PH.D. THESIS

This is to certify that the Ph.D. thesis of

Harley D. Eades III

has been approved by the Examining Committee for the thesis requirement for the Doctor of Philosophy degree in Computer Science at the August 2014 graduation.

Thesis Committee: _____
Aaron Stump, Thesis Supervisor

_____
Cesare Tinelli

_____
Stephanie Weirich

_____
Gregory Landini

_____
Kasturi Varadarajan

To my lovely wife, Jenny Eades.

Program testing can be used to show the presence of bugs, but never to show their absence!

–Dijkstra (1970)

ACKNOWLEDGEMENTS

The first person I would like to acknowledge is my advisor Aaron Stump. He is one of the kindest and most intelligent people I have had the pleasure to work with, and without his guidance I would have never made it this far. I can only hope to acquire the insight and creativity you have when working on a research problem. Furthermore, I would like to thank him for introducing me to my research area in type theory and the foundations of functional programming languages.

Secondly, I would like to thank my wife, Jenny Eades, whose hard work literally made it possible for there to be food on our table and a roof over our heads. She is a remarkable person who I cherish; without her I would be lost.

I would like to thank my family, especially my parents, Harley and Judy Eades, for their support, and my brother, Steve Eades, who let Jenny and I sleep at his house when we visited over the course of the last five years. My family always reminded me to climb out of the office, and that it was okay to have fun outside of research.

Cesare Tinelli taught me about rigor and how to probe deep into a research article to get at the heart of the matter. I would like to thank him for that.

Stephanie Weirich is an amazing researcher with great ideas, and I learned a lot from her. I am very thankful for the time I got to spend working with her especially during the summer of 2012.

I would also like to thank all of the members of the University of Iowa Computational Logic Center. I have really enjoyed interacting with them all both in research

and personally. I learned a tremendous amount during our semester seminars.

Most of my research over the course of the last five years was part of the Trellys project. I learned a lot from every member of the project. So I would like to thank them all for their many conversations, especially during the yearly project meetings. These were extremely fun, and we always had great homemade pizza. If I forgot anyone, then I am sorry, and I thank you.

TABLE OF CONTENTS

LIST OF FIGURES ix

INTRODUCTION 1

PART A. BACKGROUND 6

CHAPTER 1. A BRIEF HISTORY OF TYPE THEORY 7

1.1 The Early Days (1900 - 1960) 7
1.2 Modern Type Theory 10

CHAPTER 2. THE COMPUTATIONAL TRINITY 32

2.1 Logic 33
2.2 Category Theory 36
2.3 Impact 40

CHAPTER 3. CLASSICAL TYPE THEORY 42

3.1 The $\lambda\mu$-Calculus 42
3.2 The $\lambda\Delta$-Calculus 47
3.3 Beautiful Dualities 51
3.3.1 The Duality of Computation 52
3.3.2 The Dual Calculus 59

CHAPTER 4. DEPENDENT TYPE THEORY 64

4.1 Martin-Löf's Type Theory 64
4.2 The Calculus of Constructions 74

CHAPTER 5. DEPENDENT TYPES IN PRACTICE 80

CHAPTER 6. METATHEORY OF TYPE THEORIES 86

6.1 Hereditary Substitution 88
6.2 Hereditary Substitution for STLC 89
6.3 Tait-Girard Reducibility 97
6.4 Logical Relations 102
6.4.1 Step-Indexed Logical Relations 103

PART B. DESIGN 113

CHAPTER 7. FREEDOM OF SPEECH 114

7.1 Syntax and Reduction Relation 119
7.2 Type System 125

CHAPTER 8. SEPARATION OF PROOF FROM PROGRAM 135

CHAPTER 9. DUALIZED LOGIC AND TYPE THEORY 150

9.1 Pinto and Uustalu's L 154
9.2 Dualized Intuitionistic Logic 156
9.3 Dualized Type Theory 159

PART C. BASIC SYNTACTIC ANALYSIS 166

CHAPTER 10. FREEDOM OF SPEECH 167

10.1 Basic Results 168
10.2 Type Preservation 174
10.3 Logical consistency 177

CHAPTER 11. DUALIZED LOGIC AND TYPE THEORY 201

11.1 Consistency of DIL 201
11.2 Completeness of DIL 204
11.3 Metatheory of DTT 230

PART D. NORMALIZATION BY HEREDITARY SUBSTITUTION 252

CHAPTER 12. STRATIFIED SYSTEM F AND BEYOND 253

12.1 Stratified System F 254
12.1.1 Basic Syntactic Lemmas 257
12.1.2 Ordering on Types 266
12.1.3 Hereditary Substitution 269
12.1.4 Main Properties 272
12.1.5 The Main Substitution Lemma 281
12.1.6 Concluding Normalization 284
12.2 Stratified System F^+ 287
12.2.1 Ordering on Types 290
12.2.2 Hereditary Substitution 292
12.2.3 Main Properties 294

12.2.4 Concluding Normalization 322
12.3 Dependent Stratified System $F^{=}$ 324
12.3.1 Basic Syntactic Results 325
12.3.2 Hereditary Substitution 334
12.3.3 Concluding Normalization 339

CHAPTER 13. THE $\lambda\Delta$-CALCULUS 344

13.1 Basic Syntactic Lemmas 344
13.2 An Extension 346
13.2.1 Problems with a Naive Extension 346
13.2.2 A Correct Extension 347
13.2.3 Main Properties 351
13.3 Concluding Normalization 366
13.4 Related Work 368

CONCLUSION 372

REFERENCES 375

LIST OF FIGURES

Figure

1 Syntax and reduction rules for the Church-style simply-typed λ-calculus 12

2 Typing Relation for the Church-style simply typed λ-calculus 12

3 Syntax and reduction rules for the Curry-style simply-typed λ-calculus 16

4 Typing relation for the Curry-style simply typed λ-calculus 16

5 Syntax and reduction rules for Gödel's system T 17

6 Typing Relation for the Gödel's system T 17

7 Syntax and reduction rules for system F 22

8 Typing relation for the system F 22

9 Syntax and reduction rules for SSF 25

10 Kinding relation for the SSF 26

11 Typing relation for the SSF 27

12 Syntax and reduction rules for system F^{ω} 29

13 Kinding rules of system F^{ω} 30

14 Typing relation for the system F^{ω} 30

15 Syntax and reduction rules for the $\lambda\mu$-calculus 43

16 Type-checking algorithm for the $\lambda\mu$-calculus 44

17 Syntax and reduction rules for the $\lambda\Delta$-calculus 48

18 Type-checking algorithm for the $\lambda\Delta$-calculus 49

19 The Syntax and Reduction Rules for the $\bar{\lambda}\mu\tilde{\mu}$-Calculus 57

20 The Typing Rules for the $\bar{\lambda}\mu\tilde{\mu}$-Calculus 57

21 Syntax of the Dual Calculus 59

22 Reduction Rules for the Dual Calculus 61

23 Typing Rules for the Dual Calculus 62

24 The syntax of Martin-Löf's Type Theory 65

25 Kinding for Martin-Löf's Type Theory 68

26 Validity for Martin-Löf's Type Theory 68

27 Typing Rules for Martin Löf's Type Theory 69

28 Equality for Martin-Löf's Type Theory 70

29 Syntax for the Separated Calculus of Constructions 77

30 Sorting Rules for the Separated Calculus of Constructions 77

31 Kinding Rules for the Separated Calculus of Constructions 78

32 Typing Rules for the Separated Calculus of Constructions 78

33 The Equality for the Separated Calculus of Constructions 79

34 Syntax and reduction rules for freedom of speech 120

35 Type-checking Rules for Logical Kinds 142

36 Type-checking Rules for Predicates 142

37 Type-checking Rules for Proofs 143

38 Type-checking Rules for Proofs Continued 144

39 Semantic Values 146

40 Syntax of L. 154

41 Inference Rules for L. 155

42 Syntax for DIL. 157

43 Inference Rules for DIL. 158

44 Reachability Judgment for DIL. 159

45 Syntax for DTT. 160

46 Type-Assignment Rules for DTT. 161

47 Reduction Rules for DTT. 162

48 Well-formed substitutions 179

49 Classical typing of DTT terms 240

50 Interpretations of types 242

51 Syntax, Reduction Rules, and Commuting Conversions for SSF^{+} . . . 288

52 Well-formedness of Contexts for SSF^{+} 288

53 SSF^{+} Kinding Rules 289

54 SSF^{+} Type-Assignment Rules 289

55 Hereditary Substitution Function for Stratified System F^{+} 292

56 Hereditary Substitution Function for Stratified System F^{+} Continued 293

57 Syntax of Terms, Types, and Kinds and Reduction Rules for $DSSF^{=}$. 325

58 $DSSF^{=}$ Kinding Rules 325

59 $DSSF^{=}$ Type-Assignment Rules 326

60 $DSSF^{=}$ Type Syntactic Equality 326

61 Hereditary Substitution Function for Stratified System $F^{=}$ 338

INTRODUCTION

There are two major problems growing in two areas. The first is in Computer Science, in particular software engineering. Software is becoming more and more complex, and hence more susceptible to software defects. Software bugs have two critical repercussions: they cost companies lots of money and time to fix, and they have the potential to cause harm.

The National Institute of Standards and Technology estimated that software errors cost the United State's economy approximately sixty billion dollars annually, while the Federal Bureau of Investigations estimated in a 2005 report that software bugs cost U.S. companies approximately sixty-seven billion a year [114, 137].

Software bugs have the potential to cause harm. In 2010 there were a approximately a hundred reports made to the National Highway Traffic Safety Administration of potential problems with the braking system of the 2010 Toyota Prius [25]. The problem was that the anti-lock braking system would experience a "short delay" when the brakes where pressed by the driver of the vehicle [135]. This actually caused some crashes. Toyota found that this short delay was the result of a software bug, and was able to repair the the vehicles using a software update [115]. Another incident where substantial harm was caused was in 2002 where two planes collided over Überlingen in Germany. A cargo plane operated by DHL collided with a passenger flight holding fifty-one passengers. Air-traffic control did not notice the intersecting traffic until less than a minute before the collision occurred. Furthermore, the on-board collision

detection system did not alert the pilots until seconds before the collision. It was officially ruled by the German Federal Bureau of Aircraft Accidents Investigation that the on-board collision detection was indeed faulty [99].

The second major problem affects all of science. Scientific publications are riddled with errors. A portion of these errors are mathematical. In 2012 Casey Klein et al. used specialized computer software to verify the correctness of nine papers published in the proceedings of the International Conference on Functional Programming (ICFP). Two of the papers where used as a control which where known to have been formally verified before. In their paper [77] they show that all nine papers contained mathematical errors. This is disconcerting especially since most researchers trust published work and base their own work off of these papers. Kline's work shows that trusting published work might result in wasted time for the researchers basing their work off of these error prone publications. Faulty research hinders scientific progress.

Both problems outlined above have been the focus of a large body of research over the course of the last forty years. These challenges have yet to be completed successfully. The work we present here makes up the foundations of one side of the programs leading the initiative to build theory and tools which can be used to verify the correctness of software and mathematics. This program is called program verification using dependent type theories. The second program is automated theorem proving. In this program researchers build tools called model checkers and satisfiability modulo-theories solvers. These tools can be used to model and prove properties of

large complex systems carrying out proofs of the satisfiability of certain constraints on the system nearly automatically, and in some cases fully automatically. As an example André Platzer and Edmund Clarke in 2009 used automated theorem proving to verify the correctness of the in flight collision detection systems used in airplanes. They actually found that there were cases where two planes could collide, and gave a way to fix the problem resulting in a fully verified algorithm for collision detection. That is he mathematically proved that there is no possible way for two planes to collide if the systems are operational [108]. Automated theorem provers, however, are tools used to verify the correctness of software externally to the programming language and compiler one uses to write the software. In contrast with verification using dependent types we wish to include the ability to verify software within the programming language being used to write the software. Both programs have their merits and are very fruitful and interesting.

Every formal language within this thesis has been formally defined in a tool called Ott [121]. In addition, the full Ott specification of every type theory defined with in this thesis can be found in the appendix in [49]. Ott is a tool for writing definitions of logics, programming languages, type theories, λ-calculi, and any other formal language that consists of syntax and inference-style rules. Ott generates a parser and a type checker which is used to check the accuracy of all objects definable with in the language given to Ott as input. Ott's strongest application is to check for syntax errors within research articles. Ott is a great example of a tool using the very theory we are presenting in this thesis. It clearly stands as a successful step towards

the solution of the second major problem outlined above.

This thesis consists of two major topics. The first topic is on the design of general purpose dependently-typed functional programming languages. This topic is covered in Part B (Design). The second topic is on the analysis of dependently-typed functional programming languages and various type theories. This topic is broken up into two parts: Part C (Basic Syntactic Analysis) and Part D (Normalization by Hereditary Substitution). It is the content of these parts that consists of novel research contributions. Specifically, the following list briefly outlines each contribution:

- The design and analysis of a core dependently-typed functional programming language with a new property called freedom of speech. See Chapter 7 and Chapter 10.

- The full design of a core dependently-typed functional programming language called Separation of Proof from Program (Sep^3) that remedies several short comings of the freedom of speech language. This is a full programming language that has been implemented, but this implementation is not a contribution of this thesis. See Chapter 8. Part of Sep^3's design as well as several real-world examples of verification carried out in Sep^3 were published in the special issue on advanced programming techniques for construction of robust, general and evolutionary programs [74].

- The design and analysis of a new logic and corresponding type theory called Dualized Intuitionistic Logic (DIL) and Dualized Type Theory (DTT) respectively.

We introduce a new completely symmetric syntax that makes for a beautiful definition of the two theories. See Chapter 9 and Chapter 11.

- We prove weak normalization of an entire family of predicative type theories based on Stratified System F using a proof technique called hereditary substitution. See Chapter 12. A slightly different version of the proof of normalization using hereditary substitution of Stratified System F given in this thesis was presented at the workshop on proof-search in type theories [50].

- Similarly, we show that hereditary substitution can be extended to prove normalization of a classical type theory called the $\lambda\Delta$-calculus. See Chapter 13. This work first appeared at the workshop on control operators and their semantics [52].

- The final contribution is the brief history of type theory given in Part A. There we try and highlight the significant contributions of type theory starting with Russell. This history is by no means complete, but we provide the complete definition of many significant type theories. This tries to provide a one stop shop for an introduction to the field.

PART A

BACKGROUND

CHAPTER 1

A BRIEF HISTORY OF TYPE THEORY

In this section we give a short history of type theory. This history will set the stage for the later development by illustrating the reasons type theories exist and are important, and by giving some definitions of well-known theories that make for good examples in later sections. We first start with the early days of type theory between the years of 1900 and 1960 during the time of Bertrand Russell and Alonzo Church. They are as we consider them the founding fathers of type theory. The history given here is presented in chronological order. This is not to be considered a complete history, but rather a glimpse at the highlights of the history of type theory. This is the least amount of history one must know to fully understand where we have been and where this line of research may be heading.

1.1 The Early Days (1900 - 1960)

In the early 1900's Bertrand Russell pointed out a paradox in naive set theory. The paradox states that if $H = \{x \mid x \notin x\}$ then $H \in H \iff H \notin H$. The problem Russell exploits is that the comprehension axiom of naive set theory is allowed to use impredicative-universal quantification. That is x in the definition of H could be instantiated with H, because we are universally quantifying over all sets. Russell called this vicious circularity, and he thought it made no sense at all. Russell plagued by this paradox needed a way of eliminating it. To avoid the paradox Russell, as he described in letters to Gottlob Frege [67, 66], considers sets as having a certain

level and such sets may only contain objects of lower level. Actually, in his letters to Frege he gives a brief description of what came to be called the ramified theory of types which is a generalization of the type theory we describe here. However, this less general type theory is enough to avoid Russell's logical paradoxes. These levels can be considered as types of objects and so Russell's theory became known as simple type theory. Now what does such a theory look like? Elliott Mendleson gave a nice and simple definition of the simple type theory and we summarize this in the following definition [91].

Definition 1.1.0.1.

Let U denote the universe of sets. We divide U as follows:

- *J^1 is the collection of individuals (objects of type 0).*
- *J^{n+1} is the collection of objects of type n.*

As mentioned above the simple type theory avoids Russell's paradox. Lets consider how this is accomplished. Take Russell's paradox and add types to it following Def. 1.1.0.1. We obtain if $H^n = \{x^{n-1} \mid x^{n-1} \notin x^{n-1}\}$ then $H^n \in H^n \iff H^n \notin H^n$. We can easily see that this paradox is false. H^n can only contain elements of type $n-1$ which excludes H^n.

Russell's simple type theory reveals something beautiful. It shows that to enforce a particular property over a collection of objects we can simply add types to the objects. This is the common theme behind all type theories. The property Russell wished to enforce was predicativity of naive set theory. Throughout this thesis we will see several different properties types can enforce. While ramified type theory

and simple type theory are the first defined type theories they however are not the formulation used throughout computer science. The most common formulations used are the varying formulations and extensions of Alonzo Church's simply typed theory and Haskell Curry's combinatory logic [33, 29] [1].

In 1932 Alonzo Church published a paper on a set of formal postulates which he thought could be used to get around Russell's logical paradoxes without the need for types [32]. In this paper he defines what we now call the λ-calculus. The original λ-calculus consisted of variables, predicates denoted $\lambda x.t$, and predicate application denoted $t_1\ t_2$. See the appendix in [49] for a complete definition of the λ-calculus. It was not until Stephen Kleene and John Rosser were able to show that the λ-calculus was inconsistent as a logic when Church had to embrace types [76]. To over come the logical paradoxes shown by Kleene and Rosser, Church, added types to his λ-calculus to obtain the simply typed λ-calculus [33, 12]. In the next section we give a complete definition of Church's simple type theory. The reason we postpone the definition of the simply typed λ-calculus is because we provide a modern formulation of the theory. So far we have summarized the beginnings of type theory starting with Russell, Curry, and Church. Some really great references on this early history and more can be found in [29, 37, 17]. We now move on to modern type theory where we will cover a large part of type theory as it stands today.

---

[1] While Church's simple type theory is the most common there are some other type theories that have become very common to use and extend. To name a few: Gödel's system T, Girard-Reynolds system F, Thierry Coquand's Calculus of Constructions, Per Martin-Löf's Type Theory, Michel Parigot's $\lambda\mu$-Calculus, and Philip Wadler's Dual Calculus.

1.2 Modern Type Theory

In this section we take a journey through modern type theory by presenting various important advances in the field. We will provide detailed definitions of each type theory considered. The reader may have noticed that the only definition of type theory we have provide is that a type theory is any theory in which one must enforce a property by organizing the objects of the theory into collections based on a notion of type. This is not at all a complete definition and this section will serve as a guide to a more complete definition. We do not give a complete general formal definition of a type theory, but we hope that it is discernible from this survey. The first type theory we define is the modern formulation of the simply typed λ-calculus.

The simply typed λ-calculus. There are three formulations of the simply typed λ-calculus. The first one is called Church style [60, 17, 33], the second is called Curry style [17, 117], and the third is in the form of a pure type system. We introduce pure type systems in Section 4.2. We define the first and the second formulations here beginning with the first. We will first define the type theories and then we will comment on the differences between the two theories. The first step in defining a type theory is to define its language or syntax. Following the syntax are several judgments assigning some meaning to the language. A judgment is a statement about the object language derived from a set of inference rules. In the following type theories we will derive two judgments: the reduction relation and the type-assignment relation. The syntax and reduction relation of the Church-style simply typed λ-calculus (STLC) is defined in Figure 1 where t ranges over syntactic

expressions called terms and T ranges over syntactic expressions called types. Terms consist of variables x, unary functions $\lambda x : T.t$ (called λ-abstractions) where x is bound in t, and function application denoted $t_1\, t_2$. Now types are variables X (we use variables as base types or constants just to indicate that we may have any number of constants), and function types denoted $T_1 \to T_2$ where we call T_1 the domain type and T_2 the range type. Note that if we remove the syntax for types from Figure 1 then we would obtain the (untyped) λ-calculus. The syntax defines what language is associated with the type theory. Additionally, the reduction rules describe how to compute with the terms. The Beta rule says that if a λ-abstraction $\lambda x : T.t$ is applied to some term t', then that term may be reduced to the term resulting from substituting t' for x in t which is the English interpretation for $[t'/x]t$. We call $[t'/x]t$ the capture avoiding substitution function . It is a meta-level function. That is, it is not part of the object language. In STLC the types and terms are disjoint, but in type theories the types are used to enforce particular properties on the terms . To enforce these properties we need a method for assigning types to terms. This is the job of what we will call the typing judgment , type-checking judgment, or type-assignment judgment[2]. A judgment is a statement about the object language derived from a set of inference rules. The typing judgment for STLC is defined in Figure 2. The typing judgment depends on a typing context Γ which for now can be considered as a list

[2]Throughout the literature one may find the typing judgment being called the typing algorithm, type-checking algorithm, or type-assignment algorithm. However, this is a particular case where the rules deriving the typing judgment are algorithmic in the sense that when deriving conclusions from the inference rules deriving the judgment there is always a deterministic choice on how to proceed.

Syntax:

$$\begin{array}{lll} T & ::= & X \mid T \to T' \\ t & ::= & x \mid \lambda x : T.t \mid t_1\ t_2 \end{array}$$

Full β-reduction:

$$\frac{}{(\lambda x : T.t)\ t' \rightsquigarrow [t'/x]t}\ \text{R\_Beta} \qquad \frac{t \rightsquigarrow t'}{\lambda x : T.t \rightsquigarrow \lambda x : T.t'}\ \text{R\_Lam}$$

$$\frac{t_1 \rightsquigarrow t_1'}{t_1\ t_2 \rightsquigarrow t_1'\ t_2}\ \text{R\_App1} \qquad \frac{t_2 \rightsquigarrow t_2'}{t_1\ t_2 \rightsquigarrow t_1\ t_2'}\ \text{R\_App2}$$

Figure 1. Syntax and reduction rules for the Church-style simply-typed λ-calculus

$$\frac{}{\Gamma, x : T, \Gamma' \vdash x : T}\ \text{Var} \qquad \frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x : T_1.t : T_1 \to T_2}\ \text{Lam}$$

$$\frac{\Gamma \vdash t_1 : T_1 \to T_2 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\ t_2 : T_2}\ \text{App}$$

Figure 2. Typing Relation for the Church-style simply typed λ-calculus

of ordered pairs consisting of a variable and a type. This list is used to keep track of the types of the free variables in a term . The grammar for context is as follows:

$$\Gamma \quad ::= \quad \cdot \mid x : T \mid \Gamma_1, \Gamma_2$$

Here the empty context is denoted $\cdot$ and context extension is denoted Γ_1, Γ_2.

The inference rules deriving the typing judgment are used to determine if a

term has a particular type. That is the term t has type T in context Γ if there is a derivation with conclusion $\Gamma \vdash t : T$ and beginning with axioms. Derivations are constructed in a goal directed fashion. We first match our desired conclusion with a rule that matches its pattern and then derives its premises bottom up. To illustrate this consider the following example.

Example 1.2.0.1.

Suppose $\Gamma \overset{\mathsf{def}}{\equiv} x : T_1 \to T_2, y : T_1$. *Then we apply each rule starting with its conclusion:*

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash x : T_1 \to T_2}\,\textsc{Var} \qquad \dfrac{}{\Gamma \vdash y : T_1}\,\textsc{Var}}{x : T_1 \to T_2, y : T_1 \vdash x\ y : T_2}\,\textsc{App}}{x : T_1 \to T_2 \vdash \lambda y : T_1.(x\ y) : T_1 \to T_2}\,\textsc{Lam}}{\cdot \vdash \lambda x : T_1 \to T_2.\lambda y : T_1.(x\ y) : ((T_1 \to T_2) \to T_1) \to T_2}\,\textsc{Lam}$$

The Curry-style simply typed λ-calculus is exactly Church-style simply type λ-calculus except there is no type annotations on λ-abstractions. That is we have $\lambda x.t$ instead of $\lambda x : T.t$ in the syntax for terms. This definition of STLC was an extension of Curry's work on combinator logic.

Now a large number of type theories can be either in Church-style or in Curry-style. The lack of typing annotations has a syntactic benefit. It prevents the programmer from having to fill in type annotations when defining functions. This can be very beneficial when defining complicated functions. Curry-style type theories also differ semantically. Church-style type theories contain annotations to enforce the assignment of exactly one type to any given term. Now Curry-style type theories do not contain annotations, thus any given term may have many different types. Consider

the identity function $\lambda x.x$ this function may have the type $\mathsf{Nat} \to \mathsf{Nat}$, but it can also be given the type $\mathsf{Bool} \to \mathsf{Bool}$. In fact, there are infinitely main types one can give the Curry-style identity function. We can also characterize this semantic difference by what John Reynolds called intrinsic and extrinsic meanings. Church-style theories give an intrinsic meaning to terms. This means that typeable terms are the only terms assigned a meaning. Thus, the identity function $\lambda x : T.x$ always has a meaning, because we can give it the type $T \to T$, but the function $\lambda x : T.x\,x$ has no meaning, because no matter how hard we try we can never give the correct type annotation T. Now Curry-style theories give an extrinsic meaning to terms which amounts to the same meaning we give un-type (or uni-typed) type theories. The identity function $\lambda x.x$ can be assigned the meaning that it is the identity function on the entire domain of values, not just the typeable ones. Note that we can give a Curry-style type theory both an extrinsic and an intrinsic semantics, but Church-style is always intrinsic [118]. A last remark is that type annotations can actually make giving an intrinsic meaning difficult conducting the meta-theory of various expressive type theories and programming languages, and thus removing annotations, but still maintaining an intrinsic semantics may make meta-theoretic reasoning less difficult. This is the benefit of using a type-annotation eraser function[3] to translate a Church-style type theory into a Curry-style type theory with an intrinsic semantics. This has been very beneficial in the study of dependent type theories.

[3]This is often called "type erasure", but the erasure is the image of the type-annotation eraser function which is the result of applying the eraser, and hence not the process of erasing.

The syntax and reduction relation of the Curry-style STLC is defined in Figure 3 and the typing judgment is defined in Figure 4. We call the typing judgment defined here an implicit typing paradigm. The fact that it is implicit shows up in the application typing rule App:

$$\frac{\Gamma \vdash t_1 : T_1 \to T_2 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\, t_2 : T_2}\ \textsc{App}$$

Recall that these rules are read bottom up. Until now we have considered the typing judgment as simply a checking procedure with the type as one of the inputs, but often this judgment is defined so that the type is computed and becomes an output. In theories like this the above rule causes some trouble. The type T_1 is left implicit that is by looking at only the conclusion of the rule one cannot tell what the value of T_1 must be. This problem also exists for the typing rule for λ-abstractions. This is, however, not a problem in Church style STLC because that type is annotated on functions. This suggest that for some Curry style type theories type construction is undecidable. Not all type theories have a Church style and a Curry style formulations. Thierry Coquand's Calculus of Constructions is an example of a type theory that is in the style of Church, but it is also unclear how to define a Curry style version. It is also unclear how to define a Church style version of the type theory of intersection types [17].

Gödel's system T. The two type theories we have considered above are not very expressive. In fact we cannot represent any decently complex functions on the naturals within them. This suggests it is quite predictable that extensions of STLC

Syntax:

$$\begin{array}{lcl} T & ::= & X \mid T \to T' \\ t & ::= & x \mid \lambda x.t \mid t_1\ t_2 \end{array}$$

Full β-reduction:

$$\frac{}{(\lambda x.t)\ t' \rightsquigarrow [t'/x]t}\ \text{R\_Beta} \qquad \frac{t \rightsquigarrow t'}{\lambda x.t \rightsquigarrow \lambda x.t'}\ \text{R\_Lam} \qquad \frac{t_1 \rightsquigarrow t_1'}{t_1\ t_2 \rightsquigarrow t_1'\ t_2}\ \text{R\_App1}$$

$$\frac{t_2 \rightsquigarrow t_2'}{t_1\ t_2 \rightsquigarrow t_1\ t_2'}\ \text{R\_App2}$$

Figure 3. Syntax and reduction rules for the Curry-style simply-typed λ-calculus

$$\frac{}{\Gamma, x : T, \Gamma' \vdash x : T}\ \text{Var} \qquad \frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x.t : T_1 \to T_2}\ \text{Lam} \qquad \frac{\Gamma \vdash t_1 : T_1 \to T_2 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\ t_2 : T_2}\ \text{App}$$

Figure 4. Typing relation for the Curry-style simply typed λ-calculus

would arise. The first of these is Gödel's system T. In this theory Gödel extends STLC with natural numbers and primitive recursion. In [60] the authors present system T with pairs and booleans, but we leave these out here for clarity. The big improvement is primitive recursion. The syntax and reduction relation are defined in Figure 5 and the type-checking relation is defined in Figure 6.

We can easily see from the definition of the language that this is a direct extension of STLC. Gödel extended the types of STLC with a type constant Nat

Syntax:

$$\begin{array}{lcl} T & ::= & \mathsf{Nat} \mid T \to T' \\ t & ::= & x \mid 0 \mid \mathsf{S} \mid \lambda x : T.t \mid t_1\ t_2 \mid \mathsf{R} \end{array}$$

Full β-reduction:

$$\frac{}{(\lambda x : T.t)\ t' \rightsquigarrow [t'/x]t}\ \text{R\_Beta} \qquad \frac{}{\mathsf{R}\ t_1\ t_2\ 0 \rightsquigarrow t_1}\ \text{R\_RecBase}$$

$$\frac{}{\mathsf{R}\ t_1\ t_2\ (\mathsf{S}\ t_3) \rightsquigarrow t_2\ (\mathsf{R}\ t_1\ t_2\ t_3)\ t_3}\ \text{R\_RecStep} \qquad \frac{t_1 \rightsquigarrow t_1'}{\mathsf{R}\ t_1\ t_2\ t_3 \rightsquigarrow \mathsf{R}\ t_1'\ t_2\ t_3}\ \text{R\_RecCong1}$$

$$\frac{t_2 \rightsquigarrow t_2'}{\mathsf{R}\ t_1\ t_2\ t_3 \rightsquigarrow \mathsf{R}\ t_1\ t_2'\ t_3}\ \text{R\_RecCong2} \qquad \frac{t_3 \rightsquigarrow t_3'}{\mathsf{R}\ t_1\ t_2\ t_3 \rightsquigarrow \mathsf{R}\ t_1\ t_2\ t_3'}\ \text{R\_RecCong3}$$

$$\frac{t \rightsquigarrow t'}{\lambda x : T.t \rightsquigarrow \lambda x : T.t'}\ \text{R\_Lam} \qquad \frac{t_1 \rightsquigarrow t_1'}{t_1\ t_2 \rightsquigarrow t_1'\ t_2}\ \text{R\_App1} \qquad \frac{t_2 \rightsquigarrow t_2'}{t_1\ t_2 \rightsquigarrow t_1\ t_2'}\ \text{R\_App2}$$

$$\frac{t \rightsquigarrow t'}{\mathsf{S}\ t \rightsquigarrow \mathsf{S}\ t'}\ \text{R\_Succ}$$

Figure 5. Syntax and reduction rules for Gödel's system T

$$\frac{}{\Gamma, x : T, \Gamma' \vdash x : T}\ \text{Var} \qquad \frac{}{\Gamma \vdash 0 : \mathsf{Nat}}\ \text{Zero} \qquad \frac{}{\Gamma \vdash \mathsf{S} : \mathsf{Nat} \to \mathsf{Nat}}\ \text{Succ}$$

$$\frac{}{\Gamma \vdash \mathsf{R} : T \to ((T \to (\mathsf{Nat} \to T)) \to (\mathsf{Nat} \to T))}\ \text{Rec}$$

$$\frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x : T_1.t : T_1 \to T_2}\ \text{Lam} \qquad \frac{\Gamma \vdash t_1 : T_1 \to T_2 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\ t_2 : T_2}\ \text{App}$$

Figure 6. Typing Relation for the Gödel's system T

which is the type of natural numbers. He then extended the terms with a constant term 0 denoting the natural number zero, a term S which is the successor function and finally a recursor R which corresponds to primitive recursion. The typing judgment is extended in the straightforward way. We only explain the typing rule for the recursor of system T. Consider the rule:

$$\frac{}{\Gamma \vdash \mathsf{R} : T \to ((T \to (\mathsf{Nat} \to T)) \to (\mathsf{Nat} \to T))} \text{ Rec}$$

We can think of R as a function which takes in a term of type T, which will be the base case of the recursor, and then a term of type $T \to (\mathsf{Nat} \to T)$, which is the step case of the recursion, and a second term of type Nat, which is the natural number index of the recursion, i.e. with each recursive call this number decreases. Finally, when given these inputs R will compute a term by recursion of type T. While the typing of R gives us a good picture of its operation the reduction rules for R give an even better one. The rule

$$\frac{}{\mathsf{R}\, t_1\, t_2\, 0 \rightsquigarrow t_1} \text{ R\_RecBase}$$

shows exactly that the first argument of R is the base case. Similarly, the rule

$$\frac{}{\mathsf{R}\, t_1\, t_2\, (\mathsf{S}\, t_3) \rightsquigarrow t_2\, (\mathsf{R}\, t_1\, t_2\, t_3)\, t_3} \text{ R\_RecStep}$$

shows how the step case is computed. The type of R tells us that its second argument must be a function which takes in the recursive call and the predecessor of the index of R. These two functions turn out to be all that is needed to compute all primitive recursive functions [60].

The authors of [60] consider system T to be a step forward computationally, but a step backward logically. We will see in Section 2 how type theories can be considered as logics, but for now it suffices to say that they claim that system T has no such correspondence. It turns out that system T is expressive enough to define every primitive recursive function. In fact we can encode every ordinal from 0 to ϵ_0 in system T. This is quite an improvement from STLC. We now pause to give a few example terms corresponding to interesting functions and some example computations.

Example 1.2.0.2.

Some interesting functions in system T:

Addition:

$$\mathbf{add}\, x\, y \overset{\mathsf{def}}{\equiv} \lambda x : \mathsf{Nat}.(\lambda y : \mathsf{Nat}.(\mathsf{R}\, x\, (\lambda z : \mathsf{Nat}.(\lambda w : \mathsf{Nat}.(\mathsf{S}\, z)))\, y))$$

Multiplication:

$$\mathbf{mult}\, x\, y \overset{\mathsf{def}}{\equiv} \lambda x : \mathsf{Nat}.(\lambda y : \mathsf{Nat}.(\mathsf{R}\, 0\, (\lambda z : \mathsf{Nat}.(\lambda w : \mathsf{Nat}.(\mathbf{add}\, x\, z)))\, y))$$

Exponentiation:

$$\mathbf{exp}\, x\, y \overset{\mathsf{def}}{\equiv} \lambda x : \mathsf{Nat}.(\lambda y : \mathsf{Nat}.(\mathsf{R}\, (\mathsf{S}\, 0)\, (\lambda z : \mathsf{Nat}.(\lambda w : \mathsf{Nat}.(\mathbf{exp}\, x\, z)))\, y))$$

Predecessor:

$$\mathbf{pred}\, x \overset{\mathsf{def}}{\equiv} \lambda x : \mathsf{Nat}.(\mathsf{R}\, 0\, (\lambda z : \mathsf{Nat}.(\lambda w : \mathsf{Nat}.w))\, x)$$

Example 1.2.0.3.

We give an example reduction of addition. We define natural numbers using constructor form, and define a more convenient syntax as follows: $\hat{1} \overset{\mathsf{def}}{\equiv} \mathsf{S}\, 0$, $\hat{2} \overset{\mathsf{def}}{\equiv} \mathsf{S}\, (\mathsf{S}\, 0)$, *etc. Now we provide the following reduction:*

$$
\begin{array}{lll}
\mathbf{add}\,\hat{2}\,\hat{3} & \rightsquigarrow^2 & \mathsf{R}\,\hat{2}\,(\lambda z : \mathsf{Nat}.(\lambda w : \mathsf{Nat}.(\mathsf{S}\,z)))\,\hat{3} \\
& \rightsquigarrow & ((\lambda z : \mathsf{Nat}.(\lambda w : \mathsf{Nat}.(\mathsf{S}\,z)))\,(\mathsf{R}\,\hat{2}\,(\lambda z : \mathsf{Nat}.(\lambda w : \mathsf{Nat}.(\mathsf{S}\,z)))\,\hat{2}))\,\hat{3} \\
& \rightsquigarrow & (\lambda w : \mathsf{Nat}.(\mathsf{S}\,(\mathsf{R}\,\hat{2}\,(\lambda z : \mathsf{Nat}.(\lambda w : \mathsf{Nat}.(\mathsf{S}\,z)))\,\hat{2})))\,\hat{3} \\
& \rightsquigarrow & \mathsf{S}\,(\mathsf{R}\,\hat{2}\,(\lambda z : \mathsf{Nat}.(\lambda w : \mathsf{Nat}.(\mathsf{S}\,z)))\,\hat{2}) \\
& \rightsquigarrow & \mathsf{S}\,(((\lambda z : \mathsf{Nat}.(\lambda w : \mathsf{Nat}.(\mathsf{S}\,z)))\,(\mathsf{R}\,\hat{2}\,(\lambda z : \mathsf{Nat}.(\lambda w : \mathsf{Nat}.(\mathsf{S}\,z)))\,\hat{1}))\,\hat{2}) \\
& \rightsquigarrow & \mathsf{S}\,((\lambda w : \mathsf{Nat}.(\mathsf{S}\,(\mathsf{R}\,\hat{2}\,(\lambda z : \mathsf{Nat}.(\lambda w : \mathsf{Nat}.(\mathsf{S}\,z)))\,\hat{1})))\,\hat{2}) \\
& \rightsquigarrow & \mathsf{S}\,(\mathsf{S}\,(\mathsf{R}\,\hat{2}\,(\lambda z : \mathsf{Nat}.(\lambda w : \mathsf{Nat}.(\mathsf{S}\,z)))\,\hat{1})) \\
& \rightsquigarrow & \mathsf{S}\,(\mathsf{S}\,((\lambda z : \mathsf{Nat}.(\lambda w : \mathsf{Nat}.(\mathsf{S}\,z)))\,(\mathsf{R}\,\hat{2}\,(\lambda z : \mathsf{Nat}.(\lambda w : \mathsf{Nat}.(\mathsf{S}\,z)))\,0)\,\hat{1})) \\
& \rightsquigarrow & \mathsf{S}\,(\mathsf{S}\,((\lambda z : \mathsf{Nat}.(\lambda w : \mathsf{Nat}.(\mathsf{S}\,z)))\,\hat{2}\,\hat{1})) \\
& \rightsquigarrow & \mathsf{S}\,(\mathsf{S}\,((\lambda w : \mathsf{Nat}.(\mathsf{S}\,\hat{2}))\,\hat{1})) \\
& \rightsquigarrow & \mathsf{S}\,(\mathsf{S}\,(\mathsf{S}\,\hat{2})) \\
& \equiv & \mathsf{S}\,(\mathsf{S}\,(\mathsf{S}\,(\mathsf{S}\,0))) \\
& \equiv & \hat{5}
\end{array}
$$

Notice that the example reduction given in Ex. 1.2.0.3 is terminating. A natural question one could ask is, are all functions definable in system T terminating? The answer is positive. There is a detailed proof of termination of system T in [60]. The proof is similar to how we show strong normalization for STLC in Section 6.3. Termination is in fact guaranteed by the types of system T – in fact it is guaranteed by the types of all the type theories we have seen up till now. Remember types are used to enforce certain properties and termination is one of the most popular properties types enforce.

Girard-Reynold's System F. System T extended STLC with primitive recursion, but it is not really that large of a leap forward, logically. However, a large leap was taken independently by a French logician named Jean-Yves Girard and an American computer scientist named John Reynolds. In 1971 Girard published his thesis which included a number of advances in type theory one of them being an extension of STLC with two new constructs [59, 60, 17]. In STLC we have term variables and binders for them called λ-abstractions. Girard added type variables

and binders for them. This added the ability to define a large class of truly universal functions. He named his theory system F, and went on to show that it has a beautiful correspondence with second order arithmetic [142]. He showed that everything definable in second order arithmetic is also definable in system F by defining a projection from system F into second order arithmetic. This shows that system F is a very powerful type theory both computationally and as we will see later logically. Later in 1974 Reynolds published a paper which contained a type theory equivalent to Girard's system F [116, 117]. Reynolds being in the field of programming languages was investigating polymorphism. That is the ability to define universal (or generic[4]) functions within a programming language. That is functions with generic types which can be instantiated with other types. For example, being able to write a generic fold operation which is polymorphic in the type of data the list can hold. In system T or STLC this was not possible. We would have to define a new fold for each type of list. Reynolds also showed that system F is equivalent to second order arithmetic, in a similar, although different, way Girard did [142].

The syntax for terms, types, and the reduction rules are defined in Figure 7 and the definition of the typing relation is defined in Figure 8. Similar to system T we can easily see that system F is an extension of STLC. Types now contain a new type $\forall X.T$ which binds the type variable X in the type T . This allows one to define more universal types allowing for the definition of single functions that can work on

[4] Throughout this thesis we will use the term "generic" to mean that terms or programs are written with the most abstract type possible. Try not to confuse this with generic programming in the sense used in the design of algorithms.

Syntax:

$$\begin{array}{lcl} T & ::= & X \mid T \to T' \mid \forall X.\, T \\ t & ::= & x \mid \lambda x : T.t \mid \Lambda X.t \mid t_1\ t_2 \mid t[T] \end{array}$$

Full β-reduction:

$$\frac{}{(\lambda x : T.t)\ t' \rightsquigarrow [t'/x]t}\ \text{R\_Beta} \qquad \frac{}{(\Lambda X.t)[T] \rightsquigarrow [T/X]t}\ \text{R\_TypeRed}$$

$$\frac{t \rightsquigarrow t'}{\lambda x : T.t \rightsquigarrow \lambda x : T.t'}\ \text{R\_Lam} \qquad \frac{t \rightsquigarrow t'}{\Lambda X.t \rightsquigarrow \Lambda X.t'}\ \text{R\_TypeAbs}$$

$$\frac{t_1 \rightsquigarrow t_1'}{t_1\ t_2 \rightsquigarrow t_1'\ t_2}\ \text{R\_App1} \qquad \frac{t_2 \rightsquigarrow t_2'}{t_1\ t_2 \rightsquigarrow t_1\ t_2'}\ \text{R\_App2}$$

$$\frac{t \rightsquigarrow t'}{t[T] \rightsquigarrow t'[T]}\ \text{R\_TypeApp}$$

Figure 7. Syntax and reduction rules for system F

$$\frac{}{\Gamma, x : T, \Gamma' \vdash x : T}\ \text{Var} \qquad \frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x : T_1.t : T_1 \to T_2}\ \text{Lam}$$

$$\frac{\Gamma, X \vdash t : T}{\Gamma \vdash \Lambda X.t : \forall X.\, T}\ \text{TypeAbs} \qquad \frac{\Gamma \vdash t_1 : T_1 \to T_2 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\ t_2 : T_2}\ \text{App}$$

$$\frac{\Gamma \vdash t : \forall X.\, T'}{\Gamma \vdash t[T] : [T/X]T'}\ \text{TypeApp}$$

Figure 8. Typing relation for the system F

data of multiple different types. Terms are extended with two new terms the $\Lambda X.t$ and $t[T]$. The former is the introduction form for the $\forall$-type while the latter is the elimination form for the $\forall$-type. The former binds the type variable X in t similarly to the λ-abstraction. The latter is read, "instantiate the type of term t with the type T." The typing rules make this more apparent. The formulation of system F we present here is indeed Church style so terms do contain type annotations. We need a reduction rule to eliminate the bound variable in $\Lambda X.t$ with an actual type much like application for λ-abstractions. Hence, we extended the reduction rules of STLC with a new rule R_TypeRed which does just that. We next consider some example functions in system F.

Example 1.2.0.4.

Example functions with their types in system F:

Identity:
Type: $\forall X.(X \to X)$
Term: $\Lambda X.\lambda x : X.x$
Pairs:
Type: $\forall X.(\forall Y.(X \to (Y \to (\mathsf{PAIR_{TY}}\ X\ Y))))$
Term: $\Lambda X.\Lambda Y.\lambda x : X.(\lambda y : Y.\Lambda Z.(\lambda z : X \to (Y \to Z).((z\ x)\ y)))$
First Projection:
Type: $\forall X.(\forall Y.((\mathsf{PAIR_{TY}}\ X\ Y) \to X))$
Term: $\Lambda X.\Lambda Y.(\lambda p : \mathsf{PAIR_{TY}}\ X\ Y.((p[X])\ (\lambda x : X.\lambda y : Y.x)))$
Second Projection:
Type: $\forall X.(\forall Y.((\mathsf{PAIR_{TY}}\ X\ Y) \to Y))$
Term: $\Lambda X.\Lambda Y.(\lambda p : \mathsf{PAIR_{TY}}\ X\ Y.((p[Y])\ (\lambda x : X.\lambda y : Y.y)))$
Natural Number n:
Type: $\forall X.((X \to X) \to (X \to X))$
Term: $\Lambda X.(\lambda s : (X \to X).(\lambda z : X.(s^n\ z)))$

Note that in the previous example we used the definition

$$\mathsf{PAIR_{TY}}\ X\ Y \stackrel{\mathsf{def}}{\equiv} \forall Z.((X \to (Y \to Z)) \to Z)$$

for readability. We could have gone even further than natural numbers and pairs by

defining addition, multiplication, exponentiation, and even primitive recursion, but we leave those to the interested reader. For more examples, see [60]. The encodings we use are the famous Church encodings of pairs and natural numbers. What is remarkable about the encoding of natural numbers is that they act as function iteration. That is for any function f from any type X to X and value v of type X we have $n\,[X]\,f\,v \rightsquigarrow^* f^n v$, where n is the term n in the above table.

There is one important property of TypeApp which the reader should take notice of. Notice that there are no restrictions on what types T ranges over. That is there is nothing preventing T from being $\forall X.T'$. This property is known as impredicativity and system F is an impredicative system. The reader may now be questioning whether or not this type theory is terminating. That is can we use impredicativity to obtain a looping term? The answer was settled negatively by Girard and we will see how he proved this in Section 6. The possibility of writing a looping term in this theory depends on the ability to be able find a closed inhabitant of the type $\forall X.X$. We call a term closed if all of its variables are bound. An inhabitant of a type T is a term with type T. Such a term could be given the type $T_1 \to T_2$ and T_1 which would allow us to write a looping term. However, it is impossible to define a closed term of type $\forall X.X$.

Stratified System F. Russell called impredicativity vicious circularity and found it appalling. He actually took steps to remove it from his type theories all together. To remove impredicativity – that is enforce predicativity – from his type theories he added a second level of types which were used to organize the types of

his theory. This organization made it impossible to instantiate a type with itself. Predicative systems are less expressive than impredicative systems [83]. This means that there are functions definable in an impredicative theory which are not definable in its predicative version. In [83, 45] Daniel Leivant and Norman Danner define and analyze a predicative version of Reynolds-Girard's system F called Stratified System F (SSF). They show that SSF is substantially weaker than system F. In fact we will discuss the fact that SSF can be proven terminating by a much simpler proof technique then system F suggesting that it is indeed weaker in Section 6.1. The syntax and reduction rules for SSF are defined in Figure 9, kinding rules in Figure 10, and typing rules in Figure 11. The objective of SSF is to enforce the property of

Syntax:

$$\begin{array}{lcl} K & ::= & *_1 \mid *_2 \mid \ldots \\ T & ::= & X \mid T \to T' \mid \forall X : *_p . T \\ t & ::= & x \mid \lambda x : T . t \mid \Lambda X : *_p . t \mid t_1\ t_2 \mid t[T] \end{array}$$

Full β-reduction:

$$\frac{}{(\lambda x : T . t)\ t' \rightsquigarrow [t'/x]t}\ \text{R\_Beta} \qquad \frac{}{(\Lambda X : *_p . t)[T] \rightsquigarrow [T/X]t}\ \text{R\_TypeRed}$$

$$\frac{t \rightsquigarrow t'}{\lambda x : T . t \rightsquigarrow \lambda x : T . t'}\ \text{R\_Lam} \qquad \frac{t \rightsquigarrow t'}{\Lambda X : *_p . t \rightsquigarrow \Lambda X : *_p . t'}\ \text{R\_TypeAbs}$$

$$\frac{t_1 \rightsquigarrow t_1'}{t_1\ t_2 \rightsquigarrow t_1'\ t_2}\ \text{R\_App1} \qquad \frac{t_2 \rightsquigarrow t_2'}{t_1\ t_2 \rightsquigarrow t_1\ t_2'}\ \text{R\_App2} \qquad \frac{t \rightsquigarrow t'}{t[T] \rightsquigarrow t'[T]}\ \text{R\_TypeApp}$$

Figure 9. Syntax and reduction rules for SSF

$$\frac{}{\Gamma, X : *_p, \Gamma' \vdash X : *_p} \text{ K\_Var} \qquad \frac{\Gamma \vdash T_1 : *_p \quad \Gamma \vdash T_2 : *_q}{\Gamma \vdash T_1 \to T_2 : *_{\mathsf{max}(p,q)}} \text{ K\_Arrow}$$

$$\frac{\Gamma, X : *_p \vdash T : *_q}{\Gamma \vdash \forall X : *_p . T : *_{\mathsf{max}(p+1,q)}} \text{ K\_Forall}$$

Figure 10. Kinding relation for the SSF

predicativity on the types of system F. To accomplish this Leivant took the same path as Russell in that he added a second layer of typing to system F. This second layer is known as the kind level. Kinds are the types of types. The kinds of SSF are the elements of the syntactic category K in the syntax for SSF. These are simply all the natural numbers. We call these type levels. To stratify the types of system F we use kinding rules to organize the types into levels making sure that polymorphic types reside in a higher level than the types allowed to instantiate these polymorphic types. The kinding rules are pretty straightforward. The one of interest is

$$\frac{\Gamma, X : *_p \vdash T : *_q}{\Gamma \vdash \forall X : *_p . T : *_{\mathsf{max}(p+1,q)}} \text{ K\_Forall.}$$

This is the rule which enforces predicativity. It does this by making sure the level of $\forall X : *_p . T$ is at a larger level than X. This works, because all the types we instantiate this type with must have the same level as X. We can easily see that $p < max(p+1, q)$ for all p and q. Hence, resulting in the enforcement of our desired property.

$$\frac{\Gamma \vdash T : *_p}{\Gamma, x : T, \Gamma' \vdash x : T}\ \text{Var} \qquad \frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x : T_1.t : T_1 \to T_2}\ \text{Lam}$$

$$\frac{\Gamma \vdash t_1 : T_1 \to T_2 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\ t_2 : T_2}\ \text{App} \qquad \frac{\Gamma, X : *_p \vdash t : T}{\Gamma \vdash \Lambda X : *_p.t : \forall X : *_p.T}\ \text{TypeAbs}$$

$$\frac{\Gamma \vdash t : \forall X : *_p.T' \quad \Gamma \vdash T : *_p}{\Gamma \vdash t[T] : [T/X]T'}\ \text{TypeApp}$$

Figure 11. Typing relation for the SSF

A understandable question one could ask at this point is, are predicative theories expressive enough to capture advanced mathematical reasoning, and real-world programming? Unfortunately there is no correct answer at this time. This is a debatable question. Some believe predicative systems are enough and that impredicative systems are too paradoxical [54]. In fact Hermann Weyl proposed a predicativist foundation of mathematics. In his book [144] he developed a predicative analysis using stratification to enforce predicativity. He goes on to show that a substantial amount of mathematics can be done predictively.

I believe that impredicativity is not something that should be abolished, but embraced. It gives theories more expressive power in an elegant way. This power comes at a cost that reasoning about impredicative theories is more complex then predicative theories, but this we think is to be expected. However, we do believe that impredicativity needs to be better understood. At least in a computational light.

System F^{ω}. In Girard's thesis [59] Girard extends the type language of system F with a copy of STLC. This type theory is called system F^{ω}. The syntax and reduction rules are in Figure 12, the kinding rules in Figure 13, and the typing rules in Figure 14. There are two kinds denoted Type and $K_1 \to K_2$. The formers inhabitants are well-formed types, while the latter's inhabitants are type level functions whose inputs are types and outputs are types. There are only three forms of well-formed types: variables, arrow types, and $\forall$-types. The additional members of the syntactic category for types are used to compute types. These are λ-abstractions denoted $\lambda X : K . T$ and applications denoted $T_1\ T_2$. Note that in general these are not types. They are type constructors. However, applications may be considered a type when $T_1\ T_2$ has type Type, but this is not always the case, because STLC allows for partial applications of functions.

The ability to compute types is known as type computation. Type-level computation adds a lot of power. It can be used to write generic function specifications. We mentioned above that system F allows one to write functions with more generic types which allows one to define term level functions once and for all. Type level computation increases this ability. In fact module systems can be encoded in system F^{ω} [122]. There is one drawback though. Since terms are disjoint from types we obtain a lot of duplication. For example, we need two copies of the natural numbers: one at the type level and one at the term level. This is unfortunate. A fix for this problem is to unite the term and type level allowing for types to depend on terms. This is called dependent type theory and is the subject of Section 4. Using dependent

Syntax:

$$\begin{array}{lcl} K & ::= & \mathsf{Type} \mid K \to K' \\ T & ::= & X \mid T \to T' \mid \forall X : K.T \mid \lambda X : K.T \mid T_1\ T_2 \\ t & ::= & x \mid \lambda x : T.t \mid \Lambda X : K.t \mid t_1\ t_2 \mid t[T] \end{array}$$

Full β-reduction:

$$\frac{}{(\lambda x : T.t)\ t' \rightsquigarrow [t'/x]t}\ \text{R\_Beta} \qquad \frac{}{(\Lambda X : K.t)[T] \rightsquigarrow [T/X]t}\ \text{R\_TypeRed}$$

$$\frac{t \rightsquigarrow t'}{\lambda x : T.t \rightsquigarrow \lambda x : T.t'}\ \text{R\_Lam} \qquad \frac{t \rightsquigarrow t'}{\Lambda X : K.t \rightsquigarrow \Lambda X : K.t'}\ \text{R\_TypeAbs}$$

$$\frac{t_1 \rightsquigarrow t_1'}{t_1\ t_2 \rightsquigarrow t_1'\ t_2}\ \text{R\_App1} \qquad \frac{t_2 \rightsquigarrow t_2'}{t_1\ t_2 \rightsquigarrow t_1\ t_2'}\ \text{R\_App2}$$

$$\frac{t \rightsquigarrow t'}{t[T] \rightsquigarrow t'[T]}\ \text{R\_TypeApp} \qquad \frac{}{(\lambda X : K.T)\ T' \rightsquigarrow [T'/X]T}\ \text{TR\_TypeBeta}$$

$$\frac{T \rightsquigarrow T'}{\lambda X : K.T \rightsquigarrow \lambda X : K.T'}\ \text{TR\_TypeLam} \qquad \frac{T_1 \rightsquigarrow T_1'}{T_1\ T_2 \rightsquigarrow T_1'\ T_2}\ \text{TR\_TypeApp1}$$

$$\frac{T_2 \rightsquigarrow T_2'}{T_1\ T_2 \rightsquigarrow T_1\ T_2'}\ \text{TR\_TypeApp2}$$

Type β-equality:

$$\frac{}{T \equiv_\beta T}\ \text{Eq\_Refl} \qquad \frac{T_2 \equiv_\beta T_1}{T_1 \equiv_\beta T_2}\ \text{Eq\_Sym}$$

$$\frac{T_1 \equiv_\beta T_2 \quad T_2 \equiv_\beta T_3}{T_1 \equiv_\beta T_3}\ \text{Eq\_Trans} \qquad \frac{T_1 \equiv_\beta T_2}{\lambda X : K.T_1 \equiv_\beta \lambda X : K.T_2}\ \text{Eq\_Lam}$$

$$\frac{T_1 \equiv_\beta T_1' \quad T_2 \equiv_\beta T_2'}{T_1\ T_2 \equiv_\beta T_1'\ T_2'}\ \text{Eq\_App} \qquad \frac{T_1 \equiv_\beta T_1' \quad T_2 \equiv_\beta T_2'}{T_1 \to T_2 \equiv_\beta T_1' \to T_2'}\ \text{Eq\_Imp}$$

$$\frac{T_1 \equiv_\beta T_2}{\forall X : K.T_1 \equiv_\beta \forall X : K.T_2}\ \text{Eq\_Forall} \qquad \frac{}{(\lambda X : K.T_2)\ T_1 \equiv_\beta [T_1/X]T_2}\ \text{Eq\_Beta}$$

Figure 12. Syntax and reduction rules for system F^ω

$$\frac{}{\Gamma, X : K, \Gamma' \vdash X : K}\ \text{K\_Var} \qquad \frac{\Gamma \vdash T_1 : \mathsf{Type} \quad \Gamma \vdash T_2 : \mathsf{Type}}{\Gamma \vdash T_1 \to T_2 : \mathsf{Type}}\ \text{K\_Arrow}$$

$$\frac{\Gamma, X : K \vdash T : \mathsf{Type}}{\Gamma \vdash \forall X : K.T : \mathsf{Type}}\ \text{K\_Forall} \qquad \frac{\Gamma, X : K_1 \vdash T : K_2}{\Gamma \vdash \lambda X : K_1.T : K_1 \to K_2}\ \text{K\_Lam}$$

$$\frac{\Gamma \vdash T_1 : K_1 \to K_2 \quad \Gamma \vdash T_2 : K_1}{\Gamma \vdash T_1\ T_2 : K_2}\ \text{K\_App}$$

Figure 13. Kinding rules of system F^ω

$$\frac{\Gamma \vdash T : \mathsf{Type}}{\Gamma, x : T, \Gamma' \vdash x : T}\ \text{Var} \qquad \frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x : T_1.t : T_1 \to T_2}\ \text{Lam}$$

$$\frac{\Gamma, X : K \vdash t : T_2}{\Gamma \vdash \Lambda X : K.t : \forall X : K.T}\ \text{TypeAbs} \qquad \frac{\Gamma \vdash t_1 : T_1 \to T_2 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\ t_2 : T_2}\ \text{App}$$

$$\frac{\Gamma \vdash t : \forall X : K.T'}{\Gamma \vdash t[T] : [T/X]T'}\ \text{TypeApp} \qquad \frac{T_1 \equiv_\beta T_2 \quad \Gamma \vdash T_2 : \mathsf{Type} \quad \Gamma \vdash t : T_1}{\Gamma \vdash t : T_2}\ \text{Conv}$$

Figure 14. Typing relation for the system F^ω

types and type-level computation we could amongst other things define and use only a single copy of the natural numbers.

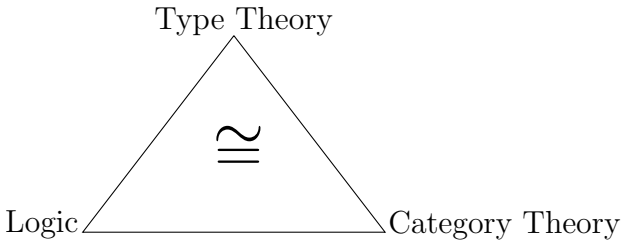
Logically, through the computational trinity (see Section 2) system F^{ω} corresponds to higher-order logic, because we are able to define predicates of higher type. This is quite a large logical leap forward from System F which corresponds to second order predicate logic.

Throughout this section we took a brief journey into modern type theory. We defined each of the most well-known type theories that are at the heart of the vast majority of existing research in type theory and foundations of functional programming languages. This was by no means a complete history, but whose aim was to give the reader a nice introduction to the field.

CHAPTER 2

THE COMPUTATIONAL TRINITY

The Merriam-Webster dictionary defines "computation" as "the act or action of computing : calculation", "the use or operation of a computer", "system of reckoning", or "an amount computed". These meanings suggest computation is nothing more than the process of mathematical calculation, but computation is so much more than this. In fact there are three perspectives of computation:



Each offering a unique position for studying computational structure. The figure above illustrates that type theory, category theory, and logic are equals where the symbol in the middle can be read as "isomorphic to." That is all three fields look very different, but can be treated as equivalent. Type theories – as we have seen above – or typed λ-calculi are essentially the study of functions where types enforce some properties on these functions. Now as it turns out category theory is basically the abstract study of mathematical structures using the abstraction of a function called a morphism. Hence, in hindsight it is not surprising that type theory and category theory are equals each offering a unique perspective of computation. Less intuitive is the connection between these two fields and that of logic. Call-

ing this beautiful relationship the computational trinity is non-standard. In fact I am proposing that this terminology become standard. The standard names for this relationship is the Curry-Howard correspondence (or isomorphism) or the proofs-as-programs propositions-as-types correspondence. The first pays tribute to Haskell Curry and William Howard. As we will see both Curry and Howard did have a hand in making this ternary relationship explicit, but they were not the only ones. Hence, this former name is unsatisfactory. The second only signifies the connection between logic and type theory; it does not mention category theory. Thus, it is unsatisfactory. Therefore, a better name for this relationship must become standard and I propose the computational trinity. Robert Harper calls this connection the "Holy Trinity" and the three way connection given above computational trinitarianism, but we have chosen to not use religious metafors in computational research. Thus, we propose the name "computational trinity" to emphasize the common structure between each point, and the fact that it is a three way connection. We now move onto making this relationship more precise. We only discuss the details of the correspondence of type theory and logic, and type theory and category theory. The other correspondence between logic and category theory follows similarly. Furthermore, we do not go into complete detail of each of these correspondences, but we give plenty of references for the curious reader.

2.1 Logic

Intuitionism began with Luitzen Brouwer. Implicit in his work was an interpretation of the formulas of propositional and predicate intuitionistic logic as compu-

tational objects. Brouwer's student Arend Heyting made this interpretation explicit for intuitionistic predicate logic against the advice of Brouwer. Brouwer believed that intuitionistic logic should never be written down, but only exist in the mind of the mathematician. Additionally, Andrey Kolmogorov defined this interpretation for intuitionistic propositional logic. This interpretation has become known as the Brouwer-Heyting-Kolmogorov-interpretation or the BHK-interpretation of intuitionistic logic. Let's consider this interpretation for intuitionistic propositional logic with conjunction, disjunction, and implication. We denote arbitrary computational constructions as c which can be built up from pairs of proof terms (t_0, t_1), unary functions denoted by λ-abstractions, and injections for proof terms for sums $inl(t)$ for inject left and $inr(t)$ for inject right. The BHK-interpretation defined in Def. 2.1.0.1 defines the assignment of proof terms using these constructs to formulas of intuitionistic propositional logic.

Definition 2.1.0.1.

The BHK-interpretation:

$$
\begin{array}{lcl}
c\, r\, (A_1 \wedge A_2) & \iff & c = (t_0, t_1) \textit{ such that } t_0\, r\, A_1 \textit{ and } t_1\, r\, A_2. \\
c\, r\, (A_1 \vee A_2) & \iff & (c = inl(t) \textit{ and } t\, r\, A_1) \textit{ or } (c = inr(t) \textit{ and } t\, r\, A_2). \\
c\, r\, (A_1 \to A_2) & \iff & c \textit{ is a function, } \lambda x.t, \textit{ such that for any } d\, r\, A_1 \\
 & & (\lambda x.t) d\, r\, A_2.
\end{array}
$$

We say a construction c realizes $A \iff c\, r\, A$.

This was the first step towards the correspondence between type theory and logic. The second was due to Curry. We mentioned in Section 1 that Curry noticed that the types of the combinatory logic correspond to the formulas of intuitionistic propositional logic. This suggested that combinatory logic can be seen as a proof

assignment to propositional logic. This was Curry's main contribution to this line of work. The third step was due to Howard. In [145] Howard revealed the correspondence between STLC and intuitionistic propositional logic in natural deduction style. He essentially uses the BHK-interpretation to assign proof terms to natural deduction and then shows that this really is STLC. It is a beautiful result. More on this can be found in [64, 145, 92, 93, 126, 136]. Since these early steps the correspondence between logic and type theory has been developed quite extensively. Reynolds' and Girard extended this correspondence to second order predicate logic using system F, and to higher order logic using system F^{ω} by Girard [142, 59]. We will see other advances to this correspondence with logic in Section 4 where we discuss dependent types.

There is one requirement a type theory must meet in order for it to correspond to a consistent logic. Computational constructs such as objects of type theory must be total (terminating). That is they must always produce a result. One part of the correspondence between type theory and logic is that the reduction rules of the type theory amount to the cut-elimination algorithm for the logic. That is, reducing terms amounts to normalizing proofs. The validity of the cut theorem – states that any non-cut-free proof can always be reduced to a cut-free one – implies consistency of the logic. The cut theorem in type theory amounts to being able to prove that all terms in the type theory are terminating. Speaking of cut elimination one might think that this correspondence only holds for sequent calculi, but one can normalize natural deduction proofs as well [110]. It is widely known that showing a type theory to be

consistent – through the remainder of the thesis we will use the words consistent and normalizing interchangeably – can be a very difficult task, and often requires advanced mathematical tools. In fact a lot of the work going into defining new type theories goes into showing it consistent. The type theories we have seen up till now are all consistent. We will discuss in detail how to show type theories to be normalizing in Section 6.

2.2 Category Theory

The year 1980 was a wonderful year for type theory. Not only did Howard show that there exists a correspondence between natural deduction style propositional logic and type theory, but Joachim Lambek also showed that there is a correspondence between type theory and cartesian closed categories [1]. In this section we briefly outline how this is the case and give an interpretation of STLC in a cartesian closed category. Before we can interpret STLC we first summarize some basic definitions of category theory. We begin with the definition of a category.

Definition 2.2.0.1.

A category *denoted* $\mathcal{C}, \mathcal{D}, \cdots$ *is an abstract mathematical structure consisting of a set of objects* Obj *denoted* $A, B, C, \cdots$ *and a set of morphisms* Mor *denoted* $f, g, h, \cdots$. *Two functions assigning objects to morphisms called* src *and* tar. *The function* src *assigns a morphism its source object (domain object) while* tar *assigns its target object (range object). We denote this assignment as* $f : A \to B$, *where* $src(f) = A$ *and* $tar(f) = B$. *Now for each object* $A \in Obj$ *there exists a unique family of morphisms called identities denoted* $id_A : A \to A$. *For any two morphisms* $f : A \to B$ *and*

$g : B \to C$ the composition of f and g must be a morphism $g \circ f : A \to C$.

Morphisms must obey the following rules:

$$\frac{f : A \to B \qquad id : B \to B}{id \circ f = f} \qquad \frac{f : A \to B \qquad id : A \to A}{f \circ id = f}$$

$$\frac{c : C \to D \qquad b : B \to C \qquad a : A \to B}{(c \circ b) \circ a = c \circ (b \circ a)}$$

In order to interpret STLC we will need a category with some special features. The first of these is the final object.

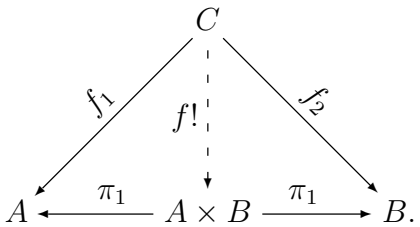
Definition 2.2.0.2.

An object 1 *of a category* $\mathcal{C}$ *is the* final *object if and only if there exists exactly one morphism* $\Diamond_A : A \to 1$ *for every object* A.

We will use the final object and finite products to interpret typing contexts. Finite products are a generalization of the cartesian product in set theory.

Definition 2.2.0.3.

An object of a category $\mathcal{C}$ *denoted* $A \times B$ *is called a binary product of the objects* A *and* B *iff there exists morphisms* $\pi_1 : A \times B \to A$ *and* $\pi_1 : A \times B \to B$ *such that for any object* C *and morphisms* $f_1 : C \to A$ *and* $f_2 : C \to B$ *there exists a unique morphism* $f : C \to A \times B$ *such that the following diagram commutes (we denote the fact that* f *is unique by* $!f$*):*

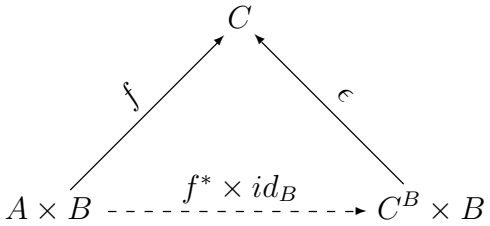


The notion of a binary product can be extended in the straightforward way to finite products of objects denoted $A_1 \times \cdots \times A_n$ for some natural number n. We will

use finite products to interpret typing contexts in the category. We need one more categorical structure to interpret STLC in a category. We need a special object that can be used to model implication or the arrow type.

Definition 2.2.0.4.

An exponential *of two objects A and B in a category $\mathcal{C}$ is an object B^A and an arrow $\epsilon : B^A \times A \to B$ called the evaluator. The evaluator must satisfy the universal property: for any object A and arrow $f : A \times B \to C$, there is a unique arrow, $f^* : A \to C^B$ such that the following diagram commutes:*



We call f^* the currying of f. By universality of ϵ every binary morphism can be curried uniquely. In the above definition we are using $\_ \times \_$ as an endofunctor. That is for any morphisms $f : A \to C$ and $g : B \to D$ we obtain the morphism $f \times g : A \times B \to C \times D$. We say a category $\mathcal{C}$ has all products and all exponentials if and only if for any two objects in $\mathcal{C}$ the product of those two objects exists in $\mathcal{C}$ and similarly for exponentials.

Definition 2.2.0.5.

A category $\mathcal{C}$ is cartesian closed *if and only if it has a terminal object* 1*, all products, and all exponentials.*

This is all the category theory we introduce in this thesis. The interested reader should see [42, 65, 82, 103] for excellent introductions to the subject.

We now have everything needed to interpret STLC as a category. Our interpretation follows that of [65]. The idea behind the interpretation is to interpret types as objects and terms as morphisms. Now a term alone does not make up a morphism, because they lack a source and a target object. So instead we interpret only typeable terms in a typing context. That is we interpret triples $\langle \Gamma, t, T \rangle$ where $\Gamma \vdash t : T$.

Definition 2.2.0.6.

Suppose $\mathcal{C}$ is a cartesian closed category. Then we interpret STLC in the category $\mathcal{C}$ by first interpreting types as follows:

$$\begin{array}{lcl} [\![X]\!] & = & \hat{X} \\ [\![T_1 \to T_2]\!] & = & [\![T_2]\!]^{[\![T_1]\!]} \end{array}$$

Then typing contexts are interpreted in the following way:

$$\begin{array}{lcl} [\![\cdot]\!] & = & 1 \\ [\![\Gamma, x : T]\!] & = & [\![\Gamma]\!] \times [\![T]\!] \end{array}$$

Finally, we interpret terms as follows:

Variables:

$$[\![\langle \Gamma, x_i : T_i, x_i, T_i \rangle]\!] = ([\![\Gamma]\!]) \times [\![T]\!] \xrightarrow{\pi_i} [\![T]\!]$$

λ-Abstractions:

$$[\![\langle \Gamma, \lambda x : T_1 . t, T_1 \to T_2 \rangle]\!] = [\![\Gamma]\!] \xrightarrow{[\![\langle \Gamma, x : T_1, t, T_2 \rangle]\!]^*} [\![T_2]\!]^{[\![T_1]\!]}$$

Applications:

$$[\![\langle \Gamma, t_1\ t_2, T_2 \rangle]\!] = [\![\Gamma]\!] \xrightarrow{\langle [\![\langle \Gamma, t_1, T_1 \to T_2 \rangle]\!], [\![\langle \Gamma, t_2, T_1 \rangle]\!] \rangle} [\![T_2]\!]^{[\![T_1]\!]} \times [\![T_1]\!] \xrightarrow{\epsilon} [\![T_2]\!]$$

In the previous definition $\hat{X}$ is just an additional object of the category. It does not matter what we call it. It does however need to be unique. This is how we interpret STLC as a cartesian closed category. Modeling other type theories with more advanced features follows quite naturally. It is not until we hit dependent types where things change drastically.

2.3 Impact

The reader may now be wondering what the benefits are of the computational trinity if there are any at all. The three perspectives of computation are all just that. They provide a unique angle on computation. To paraphrase [151] a good idea in one can be moved over to the others and it can be very "fruitful" to to look at the idea at each angle[1] .

Type theory can be seen as a foundation of typed functional programming languages. After all they are typed λ-calculi. Thus, the correspondence between type theory and logic results in programming becoming proving. Programs are proofs and their types are the propositions they are proving. This correspondence tells us exactly how to add verification to our programming languages. We isolate in some way a consistent fragment of our typed functional programming language. This fragment becomes the logic in which we prove properties of the programs definable within our programming language. So the benefit of the correspondence between logic and type theory is that it allows one language for programming and stating and proving properties of these programs.

The first use of the correspondence between logic and type theory for programming and mathematics – that is proving theorems – was Automath. Automath was a formal language much like a type theory devised by Nicolaas de Bruijn in the late sixties. A large body of ideas in modern type theory came from Automath. It al-

[1] Actually, Zenger was talking about the connection between type theory and programming, but we think it applies very nicely here.

lowed for the specification of complete mathematical theories and was equipped with a automated proof checker which was used to check the correctness of the formalized theories. In fact Automath can be thought of as the grandfather to dependent type theory. It was a wonderful line of work that resulted in a large number of great ideas. One important thing was that de Bruijn independently from Howard stated the correspondence between intuitionistic propositional logic and type theory [126].

The correspondence between type theory and category theory has many benefits. The biggest benefit is that category theory is a very abstract theory. It allows one to interpret type theories in such a way that one can see the basic structure of the theory. It has also been extensively researched so when moving over to category theory all the tools of the theory come along with it. This makes complex properties about type theories more tractable. It can also be very enlightening to take an idea and encode it in category theory. Develop the idea there and then move it over to type theory. Often the complexities of syntax get in the way when working directly in type theory, but these problems do not exist in category theory.

CHAPTER 3

CLASSICAL TYPE THEORY

Note that every type theory we have seen up till now has been intuitionistic. That is they correspond to intuitionistic logic. We clearly state that all the work Curry, Howard, de Bruijn, Girard, and others did was with respect to intuitionistic logic. So a natural question is what about classical logic?

3.1 The $\lambda\mu$-Calculus

The reason intuitionistic logic was the focus is that it lends itself very nicely to being interpreted as a system of computation. That's the entire point behind the BHK-interpretation and the work of Brouwer. This, it seemed, was not the case for classical logic, until Timothy Griffin's seminal paper titled "A Formulae-as-Types Notion of Control" [64]. This offered a typing to the control operator call/cc, and to everyone surprise connected control operators to classical proofs. Later, Michel Parigot constructed the $\lambda\mu$-calculus in 1992 [101]. Parigot was able to define a classical sequent calculus called free deduction which had a cut-elimination procedure validating the cut-theorem for classical logic [100]. This allowed for Parigot to define a computational perspective of free deduction which he called the $\lambda\mu$-calculus. This type theory can be considered the first type-safe strongly normalizing classical type theory. We now briefly introduce the $\lambda\mu$-calculus. The syntax and reduction rules are in Figure 15.

We can think of the language of the $\lambda\mu$-calculus as an extension of the λ-

Syntax:

$$\begin{array}{lcl} T, A, B, C & ::= & X \mid \bot \mid A \to B \\ t & ::= & x \mid \lambda x.t \mid \mu\alpha.s \mid t_1\ t_2 \\ s & ::= & [\alpha]t \end{array}$$

Full β-reduction:

$$\frac{}{(\lambda x.t)\ t' \rightsquigarrow [t'/x]t}\ \text{R\_Beta} \qquad \frac{}{(\mu\alpha.s)\ t' \rightsquigarrow [t'/{*}\alpha]s}\ \text{R\_Struct}$$

$$\frac{}{[\alpha](\mu\beta.s) \rightsquigarrow [\alpha/\beta]s}\ \text{R\_Renaming} \qquad \frac{t \rightsquigarrow t'}{\lambda x.t \rightsquigarrow \lambda x.t'}\ \text{R\_Lam}$$

$$\frac{s \rightsquigarrow s'}{\mu\alpha.s \rightsquigarrow \mu\alpha.s'}\ \text{R\_Mu} \qquad \frac{t \rightsquigarrow t'}{[\alpha]t \rightsquigarrow [\alpha]t'}\ \text{R\_Naming}$$

$$\frac{t' \rightsquigarrow t''}{t'\ t \rightsquigarrow t''\ t}\ \text{R\_App1} \qquad \frac{t' \rightsquigarrow t''}{t\ t' \rightsquigarrow t\ t''}\ \text{R\_App2}$$

Figure 15. Syntax and reduction rules for the $\lambda\mu$-calculus

calculus. We extend it with two new operators. The first is the μ-abstraction $\mu\alpha.s$ where α is called a co-variable, an output port, or an output variable. We call the μ-abstraction a control operator. This name conveys the fact that the μ-abstraction has the ability to control whether a value is returned or placed into its bound output port. The body of the μ-abstraction must be a term called a statement denoted by the metavariable s. Statements have the form $[\alpha]t$. We can think of this as assigning (or naming) an output port to a term. Now we extend the reduction rules with two new reduction rules and two new congruence rules for the μ-abstraction and naming operator. The R_Struct rule is called the structural reduction rule. This allows one to target reduction to a named subterm of the body of the μ-abstraction. This rule

$$\frac{}{x : \Gamma, A^x \vdash A, \Delta}\ \text{Var} \qquad \frac{t : \Gamma, A^x \vdash B, \Delta}{\lambda x.t : \Gamma \vdash A \to B, \Delta}\ \text{Lam} \qquad \frac{s : \Gamma \vdash A^\alpha, \Delta}{\mu\alpha.s : \Gamma \vdash A, \Delta}\ \text{Mu}$$

$$\frac{\begin{array}{c} t_2 : \Gamma' \vdash A, \Delta' \\ t_1 : \Gamma \vdash A \to B, \Delta \end{array}}{t_1\ t_2 : \Gamma, \Gamma' \vdash B, \Delta, \Delta'}\ \text{App} \qquad \frac{t : \Gamma \vdash A, \Delta}{[\alpha]t : \Gamma \vdash A^\alpha, \Delta}\ \text{NameApp}$$

Figure 16. Type-checking algorithm for the $\lambda\mu$-calculus

uses a special substitution operation $[t/^*\alpha]s$ which says to replace every subterm of s matching the pattern $[\alpha]t'$ with $[\alpha](t'\ t)$. We may also write $[t/^*\alpha]t'$ for the similar operation on terms. This is called structural substitution.

As we said above the language of the $\lambda\mu$-calculus is an extension of the λ-calculus, but its type assignment is very different than STLC. The type assignment rules are defined in Figure 16. Right away we can see a difference in the form of judgment. We now have $e : \Gamma \vdash \Delta$ rather than $\Gamma \vdash t : \tau$. The former is in sequent form. This is the orignal presentation used by Parigot. The feature of this is that it make it easy to see when the set of assumptions and conclusions are modified [1] Think of $e : \Gamma \vdash \Delta$ as e being a witness[2] of the sequent $\Gamma \vdash \Delta$. Just as in the other type theories we have seen, Γ is the typing context or the set of assumptions (input ports). Keeping to the style of Parigot we denote elements of Γ by A^x instead of $x : A$. The

[1]This is not the only formalization we could have used. See [44] for another example which is closer to the style we have been using for the earlier type theories.

[2]Actually, "the witness", because typing in unique.

environment Δ is either empty $\cdot$, a formula A, one or more co-assumptions or output ports, or a formula A followed by one or more output ports. Negation is defined in the same way as it is in intuitionistic logic. That is $\neg A =^{def} A \rightarrow \bot$. Note that in Δ we always have ${\bot}^{\alpha}$ (false) and in Γ we always have ${\top}^{x}$ (true) where $\top \equiv \bot \rightarrow \bot$ for any Δ and Γ trivially. We often leave these left implicit to make the presentation clean unless absolutely necessary. These two facts hold because a sequent ${A_1}^{x_1}, \cdots, {A_i}^{x_i} \vdash B, {B_1}^{\alpha_1}, \cdots, {B_i}^{\alpha_i}$ can be interpreted as $({A_1}^{x_1} \wedge \cdots \wedge {A_i}^{x_i}) \implies (B \vee {B_1}^{\alpha_1} \vee \cdots \vee {B_i}^{\alpha_i})$ where $\implies$ is implication. Using this interpretation we can see that adding true to the left and/or false to the right does not impact the logical truth of the statement. This implies the following lemma.

Lemma 3.1.0.1. *The following rules are admissible w.r.t. the $\lambda\mu$-calculus:*

$$\frac{\alpha \textit{ fresh in } \Delta \qquad s : \Gamma \vdash \Delta}{\mu\alpha.s : \Gamma \vdash \bot, \Delta}\ \text{BtmInt} \qquad \frac{\alpha \textit{ fresh in } \Delta \qquad t : \Gamma \vdash \bot, \Delta}{[\alpha]t : \Gamma \vdash \Delta}\ \text{BtmElim}$$

The $\lambda\mu$-calculus is a classical type theory so it should be the case that the law of excluded middle (LEM), $A \vee \neg A$, holds, or equivalently the law of double negation (LDN) $\neg\neg A \rightarrow A$. Since we do not have disjunction as a primitive we show LDN. Before showing the derivation of the LDN we first define some derived rules for handling negation and sequent manipulation rules. The following definition defines all derivable rules. We will take these as primitive to make things cleaner. We do not show the derivations here, because they are rather straightforward.

Lemma 3.1.0.2. *The following rules are derivable using the typing rules and the rules of Lemma 3.1.0.1:*

$$\frac{t : \Gamma, A^x \vdash \bot, \Delta}{\lambda x.t : \Gamma \vdash \neg A, \Delta}\ \text{NegInt1}$$

$$\frac{t_1 : \Gamma \vdash \neg A, \Delta \qquad t_2 : \Gamma \vdash A, \Delta}{t_1\ t_2 : \Gamma \vdash \bot, \Delta}\ \text{NegElim1}$$

$$\frac{\alpha \textit{ fresh in } \Delta \qquad s : \Gamma, A^x \vdash \Delta}{\lambda x.\mu\alpha.s : \Gamma \vdash \neg A, \Delta}\ \text{NegInt2}$$

$$\frac{t_1 : \Gamma \vdash \neg A, \Delta \qquad t_2 : \Gamma \vdash A, \Delta}{t_1\ t_2 : \Gamma \vdash \Delta}\ \text{NegElim2}$$

We are now in the state where we can prove $\neg\neg A \to A$ in the $\lambda\mu$-calculus.

Example 3.1.0.3.

In this example we prove $\neg\neg A \to A$. Suppose D is the following derivation:

$$\frac{\dfrac{}{y : \neg\neg A^y \vdash \neg\neg A, A^\alpha}\ \text{Var} \qquad \dfrac{\dfrac{\dfrac{}{x : \neg\neg A^y, A^x \vdash A}\ \text{Var}}{[\alpha]x : \neg\neg A^y, A^x \vdash A^\alpha}\ \text{NameApp}}{\lambda x.\mu\beta.[\alpha]x : \neg\neg A^y \vdash \neg A, A^\alpha}\ \text{NegInt2}}{y\ (\lambda x.\mu\beta.[\alpha]x) : \neg\neg A^y \vdash \bot, A^\alpha}\ \text{NegElim1}$$

Then the final proof is as follows:

$$\frac{\dfrac{\dfrac{D}{[\beta'](y\ (\lambda x.\mu\beta.[\alpha]x)) : \neg\neg A^y \vdash A^\alpha}\ \text{BtmElim}}{\mu\alpha.[\beta'](y\ (\lambda x.\mu\beta.[\alpha]x)) : \neg\neg A^y \vdash A}\ \text{Mu}}{\lambda y.\mu\alpha.[\beta'](y\ (\lambda x.\mu\beta.[\alpha]x)) : \cdot \vdash \neg\neg A \to A}\ \text{Lam}$$

In the above example we leave out freshness constraints to make the presentation cleaner. This example shows that the $\lambda\mu$-calculus really is classical. So from the logical perspective of computation we gain classical reasoning, but do we gain anything programmatically? It turns out that we do. We can think of the μ-abstraction and naming application as continuations which allow us to define exceptions. In fact a great way of thinking about the μ-abstraction $\mu\alpha.[\beta]t$ is due to Geuvers et al.:

From a computational point of view one should think of $\mu\alpha.[\beta]t$ as a combined operation that catches exceptions labeled α in t and throws the results of t to β. [57]

Using this point of view we can define $\mathsf{catch}_\alpha\, t$ and $\mathsf{throw}_\alpha\, t$.

Definition 3.1.0.4.

The following defines exceptions within the $\lambda\mu$-calculus:

$$\mathsf{catch}_\alpha\, t \;:=\; \mu\alpha.[\alpha]t$$

$$\mathsf{throw}_\alpha\, t \;:=\; \mu\beta.[\alpha]t, \text{ where } \beta \text{ is fresh}$$

Using our reduction rules with the addition of $\mu\alpha.[\alpha]t \rightsquigarrow t$ provided that α is fresh in t[3], we can easily define some nice reduction rules for these definitions.

Definition 3.1.0.5.

Reduction rules for exceptions:

$$\mathsf{catch}_\alpha\,(\mathsf{throw}_\alpha\, t) \rightsquigarrow \mathsf{catch}_\alpha\, t$$

$$\mathsf{throw}_\alpha\,(\mathsf{catch}_\beta\, t) \rightsquigarrow \mathsf{throw}_\alpha\,([\alpha/\beta]t)$$

There are other reductions one might want. For the others and an extension of the $\lambda\mu$-calculus see [57].

3.2 The $\lambda\Delta$-Calculus

In the previous section we introduced classical type theories and defined the $\lambda\mu$-calculus. We saw that it was a sequent style logic. In this section we define the natural deduction equivalent of $\lambda\mu$-calculus called the $\lambda\Delta$-calculus. After we define the type theory we give a brief explanation of its equivalence to the $\lambda\mu$-calculus, but we do not prove its equivalence. The $\lambda\Delta$-calculus was defined by Jakob Rehof and

[3]This is sometimes called η-reduction for control operators.

Syntax:

$$\begin{array}{lcl} T, A, B, C & ::= & X \mid \bot \mid A \to B \\ t & ::= & x \mid \lambda x : T.t \mid \Delta x : T.t \mid t_1\ t_2 \end{array}$$

Full β-reduction:

$$\frac{}{(\lambda x : T.t)\ t' \rightsquigarrow [t'/x]t}\ \text{Beta}$$

$$\frac{\begin{array}{l} y \text{ fresh in } t \text{ and } t' \\ z \text{ fresh in } t \text{ and } t' \end{array}}{(\Delta x : \neg(T_1 \to T_2).t)\ t' \rightsquigarrow \Delta y : \neg T_2.[\lambda z : T_1 \to T_2.(y\ (z\ t'))/x]t}\ \text{StructRed}$$

Figure 17. Syntax and reduction rules for the $\lambda\Delta$-calculus

Morten Sørensen in Rehof's thesis [113]. Their work on the $\lambda\Delta$-calculus was done independently of the $\lambda\mu$-calculus and they were not aware of their equivalence until Parigot pointed it out. To our knowledge no actual proof was ever published, but the proof is rather straightforward. The $\lambda\Delta$-calculus is an extension of STLC with the LDN in the form of a control operator called Δ. Unlike the other type theories we have seen we are going to first present the language and then the typing rules. Lastly, we will define the reduction rules. It is our belief that the reduction rules may be more clear after the reader sees the typing rules.

The language is defined in Figure 17[4] and the typing rules in Figure 18. We give the formulation a la Church, but the formulation a la Curry does exist. We can see that the syntax really is just the extension of STLC with the $\Delta x : T.t$ control

[4]We only include a subset of the reduction rules given by in Rehof and Sørensen. Just as before negation is defined just as it is in intuitionistic logic. For the others see [113].

$$\frac{}{\Gamma, x : A, \Gamma' \vdash x : A}\ \text{Var} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A.t : A \to B}\ \text{Lam} \qquad \frac{\Gamma, x : \neg A \vdash t : \bot}{\Gamma \vdash \Delta x : \neg A.t : A}\ \text{Delta}$$

$$\frac{\Gamma \vdash t_2 : A \quad \Gamma \vdash t_1 : A \to B}{\Gamma \vdash t_1\ t_2 : B}\ \text{App}$$

Figure 18. Type-checking algorithm for the $\lambda\Delta$-calculus

operator. This operator is the elimination form for absurdity $\bot$. We can see this connection by looking at its typing rule Delta. Here we assume $\neg A$ and show $\bot$, and obtain A. We can use this rule to prove LDN:

Example 3.2.0.1.

Suppose $\Gamma \stackrel{\mathsf{def}}{\equiv} x : \neg\neg A, y : \neg A$. *Then the proof of* $\neg\neg A \to A$ *in the* $\lambda\Delta$*-calculus:*

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash x : \neg\neg A}\ \text{Var} \qquad \dfrac{}{\Gamma \vdash y : \neg A}\ \text{Var}}{\Gamma \vdash x\ y : \bot}\ \text{App}}{x : \neg\neg A \vdash \Delta y : \neg A.(x\ y) : A}\ \text{Delta}}{\cdot \vdash \lambda x : \neg\neg A.(\Delta y : \neg A.(x\ y)) : \neg\neg A \to A}\ \text{Lam}$$

The $\lambda\Delta$-calculus is equivalent to the $\lambda\mu$-calculus. The following definition gives an embedding from the $\lambda\mu$-calculus to the $\lambda\Delta$-calculus.

Definition 3.2.0.2.

The following embeds the Church style formulation of the $\lambda\mu$*-calculus into the* $\lambda\Delta$*-calculus:*

Context:

$$|\Gamma, A^x| := |\Gamma|, x : A$$

$$|\Delta, A^\alpha| := |\Delta|, x : \neg A, \textit{where } x \textit{ is fresh in } |\Delta|$$

Terms and Statements:

$$|x| := x$$

$$|\alpha| := y, \textit{ for some fresh variable } y$$

$$|\lambda x : A.t| := \lambda x : A.|t|$$

$$|t_1\ t_2| := |t_1|\,|t_2|$$

$$|\mu\alpha : A.s| := \Delta x : \neg A.|s|^\alpha_x$$

$$|[\alpha]t|^\alpha_x := x\,|t|$$

$$|[\alpha]t|^\beta_x := z\,|t|, \textit{ where } \alpha \textit{ is distinct from } \beta \textit{ and } z \textit{ is fresh in } t$$

Using the previous definition we can now prove that if a term is typeable in the $\lambda\mu$-calculus then we can construct a corresponding term of the same type in the $\lambda\Delta$-calculus.

Lemma 3.2.0.3.

i. If $t : \Gamma \vdash A, \Delta$ *then* $\Gamma, |\Delta| \vdash |t| : A$.

ii. If $t : \Gamma \vdash ., \Delta$ *then* $\Gamma, |\Delta| \vdash |t| : \bot$.

The previous lemma establishes that the $\lambda\Delta$-calculus is at least as expressive – in terms of typeability – than the $\lambda\mu$-calculus. It so happens that we can prove that the $\lambda\mu$-calculus is at least as strong as the $\lambda\Delta$-calculus which implies that both type theories are equivalent with respect to typeability. We assume without loss of generality that the typing context Γ is sorted so that all negative types come after all positive types. A negative type is of the form $\neg A$.

Definition 3.2.0.4.

The following embeds the Church style formulation of the $\lambda\Delta$-calculus into the $\lambda\mu$-calculus:

Contexts:
If $\Gamma \equiv x_1 : A_1, \ldots, x_i : A_i, y_1 : \neg B_1, \ldots, y_j : \neg B_j$, *then*

$$|x_1 : A_1, \ldots, x_i : A_i| := {A_1}^{x_1}, \ldots, {A_i}^{x_i} \equiv \Gamma_\mu$$

$$|y_1 : \neg B_1, \ldots, y_j : \neg B_j| := {B_1}^{\alpha_1}, \ldots, {B_j}^{\alpha_j} \equiv \Delta$$

Terms:

$$|x| := x$$

$$|\lambda x : A.t| := \lambda x : A.|t|$$

$$|t_1\ t_2| := |t_1|\,|t_2|$$

$$|\Delta x : \neg A.t| := \mu\alpha.[\beta]|t|, \text{ where } \alpha \not\equiv \beta$$

Similar to the previous embedding we can now prove that all inhabited types of the $\lambda\Delta$-calculus are inhabited in the $\lambda\mu$-calculus.

Lemma 3.2.0.5. *If* $\Gamma \vdash t : A$ *then* $|t| : \Gamma_\mu \vdash A, \Delta$.

Both of the above lemmas can be proven by induction on the form of the assumed typing derivations. This equivalence extends to the reduction rules as well, but the reduction rules are not step by step equivalent. Terms of the $\lambda\Delta$-calculus will have to do more reduction than the corresponding terms of the $\lambda\mu$-calculus. We do not show this here.

3.3 Beautiful Dualities

There are some beautiful dualities present in classical logic. We say a mathematical or logical construct is dual to another if there exists an involution translating each construct to each other. An involution is a self invertible one-to-one correspon-

dence. That is if i is an involution then $i(i(x)) = x$. Now in classical logic negation is self dual, by De Morgan's laws conjunction is dual to disjunction and vice versa, and existential quantification is dual to universal quantification and vice versa. These dualities lead to wonderful symmetries in Gentzen's sequent calculus. One can see these symmetries in the rules for conjunction and disjunction. They are mirror images of each other. These beautiful dualities are not only found in classical logic, but even exist in intuitionistic logic. However, the dualities in intuitionistic logic are not well understood from a type theoretic perspective.

3.3.1 The Duality of Computation

The $\bar{\lambda}\mu\tilde{\mu}$-calculus. Pierre-Louis Curien and Hugo Herbelin put these dualities to work in a very computational way. They used these dualities to show that the call-by-value reduction strategy (CBV) is dual to the call-by-name reduction strategy (CBN). To do this they crafted an extension of the $\lambda\mu$-calculus formalized in such a way that the symmetries are explicit [44]. They are not the first to attempt this. Andrzej Filinski to our knowledge was the first to investigate dualities with respect to programming languages in his masters thesis [55]. It is there he investigates the dualities in a categorical setting. Advancing on this early work Peter Selinger gave a categorical semantics to the $\lambda\mu$-calculus and then used these semantics to show that CBV is dual to CBN [120]. However, Selinger's work did not provide an involution of duality. In [120] Selinger defines a new class of categories called control categories. These provide a model for control operators. He takes the usual cartesian closed category and enriches it with a new functor modeling classical disjunction. While this

is beautiful work we do not go into the details here.

Following Filinski and Selinger are the work of Curien and Herbelin, while following them is the work of Philip Wadler. Below we discuss Curien and Herbelin's work then Wadler's. Before going into their work we first define call-by-value and call-by-name reduction.

The call-by-value reduction strategy is a restriction of full β-reduction. It is defined for the $\lambda\mu$-calculus as follows. We first extend the language of the $\lambda\mu$-calculus by defining two new syntactic categories called values and evaluation contexts.

$$v ::= x \mid \lambda x.t \mid \mu\alpha.t$$
$$E ::= \Box \mid E\,t \mid v\,E \mid [\alpha]E$$

Values are the well-formed results of computations. In the $\lambda\mu$-calculus we only consider variables, λ-abstractions, and μ-abstractions as values. The evaluation contexts are defined by E. They give the locations of reduction and reduction order. They tell us that one may reduce the head of an application at any moment, but only reduce the tail of an application if and only if the head has been reduced to a value. This is called left-to-right CBV and is defined next.

Definition 3.3.1.1.

CBV is defined by the following rules:

$$\frac{}{(\lambda x.t)\,v \rightsquigarrow [v/x]t}\ \text{Beta} \qquad \frac{}{(\mu\alpha.s)\,v \rightsquigarrow [v/^*\alpha]s}\ \text{Struct} \qquad \frac{}{[\alpha](\mu\beta.s) \rightsquigarrow [\alpha/\beta]s}\ \text{Naming}$$

$$\frac{t \rightsquigarrow t'}{E[t] \rightsquigarrow E[t']}\ \text{Context}$$

A similar definition can be given for right-to-left CBV, but we do not give it here.

CBN can now be defined. We use the same definition of values as for CBV, but we redefine the evaluation contexts.

$$E ::= \Box \mid E\,t \mid [\alpha]E \mid \mu\alpha.E$$

Definition 3.3.1.2.

CBN is defined by the following rules:

$$\frac{}{(\lambda x.t)\,t' \rightsquigarrow [t/x]t}\ \text{Beta} \qquad \frac{}{(\mu\alpha.s)\,t \rightsquigarrow [t/^*\alpha]s}\ \text{Struct} \qquad \frac{}{[\alpha](\mu\beta.s) \rightsquigarrow [\alpha/\beta]s}\ \text{Naming}$$

$$\frac{t \rightsquigarrow t'}{E[t] \rightsquigarrow E[t']}\ \text{Context}$$

The difference between CBN and CBV is that in CBN no reduction takes place within the argument to a function. Instead we wait and reduce the argument if it is needed within a function. If the argument is never used it is never reduced. CBN in general is less efficient than CBV, but it can terminate more often than CBV. If the argument to a function is divergent then CBV will never terminate, because it must reduce the argument to a value, but CBN may terminate if the argument is never used, because arguments are not reduced.

At this point we would like to give some intuition of why CBV is dual to CBN. We reformulate an explanation due to Curien and Herbelin in [44]. To understand the relationship between CBN and CBV we encode CBV on top of CBN using a new term construct and reduction rule. It is well-known how to encode CBN on top of CBV, but encoding CBV on top of CBN illustrates their relationship between each other. Suppose we extend the language of the CBN $\lambda\mu$-calculus with the following term:

$$t ::= \ldots \mid \mathsf{let}\, x = E \,\mathsf{in}\, t$$

This extends the language to allow for terms to contain their evaluation contexts. Then we add the following reduction rule:

$$\frac{}{v\,(\mathsf{let}\, x = \Box \,\mathsf{in}\, t) \rightsquigarrow [v/x]t}\ \text{LetCtx}$$

Using this new term and reduction rule we can now encode CBV on top of CBN. That is a CBV redex is defined in the following way:

$$(\lambda x.t)_{CBV}\, t' := t'\,(\mathsf{let}\, y = \Box \,\mathsf{in}\, (\lambda x.t)y)$$

Now consider the following redex:

$$(\mu x.s)\,(\mathsf{let}\, x = \Box \,\mathsf{in}\, t)$$

We can reduce the previous term by first reducing the μ-redex, but we can also start by reducing the let-redex, because μ-abstractions are values. However, the two reducts obtained from doing these reductions are not always joinable. This forms a critical pair and shows an overlap between the LetCtx rule and the μ-reduction rule. This can be overcome by giving priority to one or the other redex. Now if we give priority to μ-redexes over all other redexes then it turns out that the reduction strategy will be all CBN, but if we choose to give priority to the let-redexes over all other redexes then all terms containing let-redexes will be reduced using CBV, because the let-expression forces the term we are binding to x to be a value. What does this have to do with duality? Well the let-expression we added to the $\lambda\mu$-calculus is actually the dual to the μ-abstraction. To paraphrase Curien and Herbelin [44]:

> *The CBV discipline manipulates input in the same way as the $\lambda\mu$-calculus manipulates output. That is computing $t_1\, t_2$ can be viewed as filling the hole of the context $t_1\, \Box$ with the result of t_2 – its value – hence this value of t_2 is an input. This seems dual to passing output values to output ports in the $\lambda\mu$-calculus.*

This tells us that to switch from CBN to CBV we take the dual of the μ-abstraction suggesting that CBV is dual to CBN and vice versa. This was the starting point of Curien and Herbelin's work. They make this relationship more precise by defining an extension of the $\lambda\mu$-calculus with duals of λ-abstractions and μ-abstractions. This requires the dual to implication. Let's define Curien and Herbelin's extension of the $\lambda\mu$-calculus and then discuss how they used it to show that CBV is dual to CBN. Curien and Herbelin called their extension of the $\lambda\mu$-calculus the $\bar{\lambda}\mu\tilde{\mu}$-calculus. Despite the ugly name it is a beautiful type theory. Its syntax and reduction rules are in Figure 19 and its typing rules are in Figure 20.

The new type $A - B$ is the dual to implication called subtraction. It is logically equivalent to $A \wedge \neg B$ which is the dual to $\neg A \vee B$ which is logically equivalent to $A \to B$. The syntactic category c are called commands. They have the form of $\langle v \mid e \rangle$ where v is a computation and e is its environment. Commands essentially encode an abstract stack machine directly in the type theory. We can think of e as the stack of terms to which v will be applied to. It also turns out that logically commands denote cuts using the cut-rule of the underlying sequent calculus. Values defined by the syntactic category v come in three flavors: variables, λ-abstractions, μ-abstractions,

Syntax:

$$\begin{array}{rcl} T, A, B, C & ::= & \bot \mid X \mid A \to B \mid A - B \\ c & ::= & \langle v \mid e \rangle \\ v & ::= & x \mid \lambda x.v \mid \mu\alpha.c \mid e \cdot v \\ e & ::= & \alpha \mid \tilde{\mu}x.c \mid v \cdot e \mid \beta\lambda.e \end{array}$$

CBV reduction:

$$\frac{}{\langle \lambda x.v_1 \mid v_2 \cdot e \rangle \rightsquigarrow \langle v_2 \mid \tilde{\mu}x.\langle v_1 \mid e \rangle \rangle} \text{ R\_Beta} \qquad \frac{}{\langle \mu\beta.c \mid e \rangle \rightsquigarrow [e/\beta]c} \text{ R\_Mu}$$

$$\frac{}{\langle v \mid \tilde{\mu}x.c \rangle \rightsquigarrow [v/x]c} \text{ R\_MuT} \qquad \frac{}{\langle e_2 \cdot v \mid \beta\lambda.e_1 \rangle \rightsquigarrow \langle \mu\beta.\langle v \mid e_1 \rangle \mid e_2 \rangle} \text{ R\_CoBeta}$$

$$\frac{c \rightsquigarrow c'}{E[c] \rightsquigarrow E[c']} \text{ E\_Ctx}$$

Figure 19. The Syntax and Reduction Rules for the $\bar{\lambda}\mu\tilde{\mu}$-Calculus

Terms:

$$\frac{}{\Gamma, x : A \vdash x : A \mid \Delta} \text{ Var} \qquad \frac{\Gamma, x : A \vdash v : B \mid \Delta}{\Gamma \vdash \lambda x.v : A \to B \mid \Delta} \text{ Lam}$$

$$\frac{c : (\Gamma \vdash \beta : B, \Delta)}{\Gamma \vdash \mu\beta.c : B \mid \Delta} \text{ Mu} \qquad \frac{\begin{array}{c}\Gamma \mid e : A \vdash \Delta \\ \Gamma \vdash v : B \mid \Delta\end{array}}{\Gamma \vdash e \cdot v : B - A \mid \Delta} \text{ CoCtx}$$

Contexts:

$$\frac{}{\Gamma \mid \alpha : A \vdash \alpha : A, \Delta} \text{ Covar} \qquad \frac{c : (\Gamma, x : A \vdash \Delta)}{\Gamma \mid \tilde{\mu}x.c : A \vdash \Delta} \text{ Comu}$$

$$\frac{\begin{array}{c}\Gamma \vdash v : A \mid \Delta \\ \Gamma \mid e : B \vdash \Delta\end{array}}{\Gamma \mid v \cdot e : A \to B \vdash \Delta} \text{ Ctx} \qquad \frac{\Gamma \mid e : B \vdash \beta : A, \Delta}{\Gamma \mid \beta\lambda.e : B - A \vdash \Delta} \text{ Colam}$$

Commands:

$$\frac{\begin{array}{c}\Gamma \vdash v : A \mid \Delta \\ \Gamma \mid e : A \vdash \Delta\end{array}}{\langle v \mid e \rangle : (\Gamma \vdash \Delta)} \text{ Cut}$$

Figure 20. The Typing Rules for the $\bar{\lambda}\mu\tilde{\mu}$-Calculus

and co-contexts denoted by $e \cdot v$. These can be thought of as the computations to give to the co-λ-abstraction and their output routed to an output port bound by the co-λ-abstraction. Finally, we have expressions or co-terms which come in four flavors: co-variables (output ports), $\tilde{\mu}$-abstractions, contexts, and co-λ-abstractions. The $\tilde{\mu}$-abstraction is the encoding of the let-expression we defined above. We write $\mathsf{let}\, x = \Box \,\mathsf{in}\, v$ as $\tilde{\mu}x.\langle v \mid e \rangle$ where e is the evaluation context for v. Thus, the $\tilde{\mu}$-abstraction is the dual to the μ-abstraction. Now contexts are commands. These provide a way of feeding input to programs. Co-λ-abstractions denoted $\beta\lambda.e$ are the dual to λ-abstractions. In stead of taking input arguments they return outputs assigned to the output port bound by the abstraction. We can see that this is a rather large reformulation/extension of the $\lambda\mu$-calculus. Just to summarize: Curien and Herbelin extended the $\lambda\mu$-calculus with all the duals of the constructs of the $\lambda\mu$-calculus.

Now reduction amounts to cuts logically, and computationally as running these abstract machine states we are building. Programming and proving amounts to the construction of these abstract machines. Other then this the reduction rules are straightforward. The typing algorithm consists of three types of judgments:

$$\begin{array}{ll} \text{Commands:} & c : (\Gamma \vdash \Delta) \\ \text{Terms:} & \Gamma \vdash v : A \mid \Delta \\ \text{Contexts:} & \Gamma \mid e : A \vdash \Delta \end{array}$$

As we said early the command typing rule is cut while the judgment for terms and contexts consist of the left rules and the right rules respectively. The bar $\mid$ separates input from output or left from right. Finally, using this type theory Curien and Herbelin define a duality of the $\bar{\lambda}\mu\tilde{\mu}$-calculus into itself. Then using this duality they

$$
\begin{array}{rcl}
T, A, B, C & ::= & X \mid A \wedge B \mid A \vee B \mid \neg A \\
t, a, b, c & ::= & x \mid \langle a, b \rangle \mid \mathsf{inl}\, t \mid \mathsf{inr}\, t \mid [k]\mathrm{not} \mid (s).\alpha \\
k, l & ::= & \alpha \mid [k, l] \mid \mathrm{fst}\, k \mid \mathrm{snd}\, k \mid \mathrm{not}[t] \mid x.(s) \\
s & ::= & t \cdot k
\end{array}
$$

Figure 21. Syntax of the Dual Calculus

show that starting with the CBN $\bar{\lambda}\mu\tilde{\mu}$-calculus and taking the dual one obtains the CBV $\bar{\lambda}\mu\tilde{\mu}$-calculus.

3.3.2 The Dual Calculus

Philip Wadler invented a type theory equivalent to Curien and Herbelin's $\bar{\lambda}\mu\tilde{\mu}$-calculus called the dual calculus [140]. What we mean by equivalent here is that both correspond to Gentzen's classical sequent calculus LK, but both type theories are definitionally inequivalent. The difference between the two type theories is that the $\bar{\lambda}\mu\tilde{\mu}$-calculus is defined with only negation, implication, and subtraction. Then using De Morgan's laws we can define conjunction and disjunction. However, the dual calculus is defined with only negation, conjunction, and disjunction. Then we define implication, which implies we may define λ-abstractions. This is a truly remarkable feature of classical logic.

The syntax of the dual calculus is defined in Figure 21. It is similar to the $\bar{\lambda}\mu\tilde{\mu}$-calculus, consisting of types, terms, coterms (continuations), and statements. As types we have propositional variables, conjunction, disjunction, and negation. Note that negation must be a primitive in the dual calculus rather than being defined.

Terms in the dual calculus are variables, the introduction form for conjunction called pairs denoted $\langle a, b \rangle$, the introduction forms for disjunction denoted $\mathsf{inl}\, t$ and $\mathsf{inr}\, t$ which can be read as inject left and inject right respectively. The next term is the introduction form of negation denoted $[k]\mathrm{not}$. The final term is a binder for coterms and is the computational correspondent to the left-to-right rule. It is denoted $(s).\alpha$. This can be thought of as running the statement s and then routing its output to the output port α. The continuations or coterms are the duals to terms and consist of covariables denoted α, copairs denoted $[k, l]$, the duals of inject-left and inject-right called first and second denoted $\mathrm{fst}\, k$ and $\mathrm{snd}\, k$ respectively. The next coterm is the elimination form of negation denoted $\mathrm{not}[t]$ which can be thought of as the continuation which takes as input a term of a negative formula and routes its output to some output port. Finally, the dual to binding an output port is binding an input port. This is denoted $x.(s)$. Now statements are the introduction of a cut and are denoted $t \cdot k$. Computationally, we can think of this as a command which runs the term t and routes its output to the continuation k which continues the computation.

The reduction rules for the dual calculus are in Figure 22 and the typing rules are in Figure 23. The reduction rules correspond to cut-elimination and can be thought of a simplification process on proofs. Computationally they can be thought of as running programs with their continuations. We derive three judgments from the typing rules for terms, coterms, and statements. They have the following forms:

$$\begin{array}{ll} \text{Terms}: & \Gamma \vdash \Delta \leftarrow t \rightarrow A \\ \text{Coterms}: & \Gamma \vdash \Delta \leftarrow k \leftarrow A \\ \text{Statements}: & \Gamma \vdash \Delta \leftarrow s \end{array}$$

The syntax of judgments are different from Wadler's original syntax. Here we use

$$\frac{}{(a \cdot \alpha).\alpha \rightsquigarrow a}\ \text{EtaR} \qquad \frac{}{x.(x \cdot k) \rightsquigarrow k}\ \text{EtaL} \qquad \frac{}{(s).\alpha \cdot k \rightsquigarrow [k/\alpha]s}\ \text{BetaR}$$

$$\frac{}{a \cdot x.(s) \rightsquigarrow [a/x]s}\ \text{BetaL} \qquad \frac{}{[k]\text{not} \cdot \ \text{not}[a] \rightsquigarrow a \cdot k}\ \text{BetaNeg}$$

$$\frac{}{\mathsf{inl}\ a \cdot [k, l] \rightsquigarrow a \cdot k}\ \text{BetaCoProd1} \qquad \frac{}{\mathsf{inr}\ a \cdot [k, l] \rightsquigarrow a \cdot l}\ \text{BetaCoProd2}$$

$$\frac{}{\langle a, b\rangle \cdot \ \text{fst}\, k \rightsquigarrow a \cdot k}\ \text{BetaProd1} \qquad \frac{}{\langle a, b\rangle \cdot \ \text{snd}\, k \rightsquigarrow b \cdot k}\ \text{BetaProd2}$$

Figure 22. Reduction Rules for the Dual Calculus

the arrows to indicate data flow. One meaning for the judgment $\Gamma \vdash \Delta \leftarrow t \rightarrow A$ is that when all the variables in Γ have an input in t then computing t either returns a value of type A or routes its output to a covariable in Δ. One meaning for the judgment $\Gamma \vdash \Delta \leftarrow k \leftarrow A$ is when the continuation gets input for all the variables in Γ and gets an input of A it computes a value which is stored in an output port in Δ. Finally, the meaning of $\Gamma \vdash \Delta \leftarrow s$ is that after the s is done computing it stores its output in an output port in Δ. Each judgment has a logical meaning. The typing rules for terms correspond to the right rules of LK, and the typing rules correspond to the left rules of LK, while the judgment for statements correspond to the cut rule of LK.

It has been said that Wadler invented the dual calculus when reading Curien and Herbelin's paper and found the subtraction operator confusing. This was his reason for going with conjunction and disjunction instead of implication. He knew

Terms:

$$\frac{}{\Gamma, x : A \vdash \Delta \leftarrow x \rightarrow A}\ \text{t\_AxR} \qquad \frac{\Gamma \vdash \Delta \leftarrow a \rightarrow A \quad \Gamma \vdash \Delta \leftarrow b \rightarrow B}{\Gamma \vdash \Delta \leftarrow \langle a, b \rangle \rightarrow A \wedge B}\ \text{t\_Prod}$$

$$\frac{\Gamma \vdash \Delta \leftarrow a \rightarrow A}{\Gamma \vdash \Delta \leftarrow \mathsf{inl}\ a \rightarrow A \vee B}\ \text{t\_CoProdl} \qquad \frac{\Gamma \vdash \Delta \leftarrow b \rightarrow B}{\Gamma \vdash \Delta \leftarrow \mathsf{inr}\ b \rightarrow A \vee B}\ \text{t\_CoProdr}$$

$$\frac{\Gamma \vdash \Delta \leftarrow k \leftarrow A}{\Gamma \vdash \Delta \leftarrow [k]\text{not} \rightarrow \neg A}\ \text{t\_NegR} \qquad \frac{\Gamma \vdash \Delta, \alpha : A \leftarrow s}{\Gamma \vdash \Delta \leftarrow (s).\alpha \rightarrow A}\ \text{t\_IR}$$

Coterms:

$$\frac{}{\Gamma \vdash \Delta, \alpha : A \leftarrow \alpha \leftarrow A}\ \text{ct\_AxL} \qquad \frac{\Gamma \vdash \Delta \leftarrow k \leftarrow A \quad \Gamma \vdash \Delta \leftarrow l \leftarrow B}{\Gamma \vdash \Delta \leftarrow [k, l] \leftarrow A \vee B}\ \text{ct\_CoProd}$$

$$\frac{\Gamma \vdash \Delta \leftarrow k \leftarrow A}{\Gamma \vdash \Delta \leftarrow \text{fst}\ k \leftarrow A \wedge B}\ \text{ct\_ProdFst} \qquad \frac{\Gamma \vdash \Delta \leftarrow k \leftarrow B}{\Gamma \vdash \Delta \leftarrow \text{snd}\ k \leftarrow A \wedge B}\ \text{ct\_ProdSnd}$$

$$\frac{\Gamma \vdash \Delta \leftarrow t \rightarrow A}{\Gamma \vdash \Delta \leftarrow \text{not}[t] \leftarrow \neg A}\ \text{ct\_NegL} \qquad \frac{\Gamma, x : A \vdash \Delta \leftarrow s}{\Gamma \vdash \Delta \leftarrow x.(s) \leftarrow A}\ \text{ct\_IR}$$

Statements:

$$\frac{\Gamma \vdash \Delta \leftarrow t \rightarrow A \quad \Gamma \vdash \Delta \leftarrow k \leftarrow A}{\Gamma \vdash \Delta \leftarrow t \cdot k}\ \text{st\_Cut}$$

Figure 23. Typing Rules for the Dual Calculus

that conjunction and disjunction are duals in a well-known way unlike implication and subtraction. Then using negation, conjunction, and disjunction he defined implication, λ-abstractions, and application. Now the definition of these differs depending on which reduction strategy is used.

Definition 3.3.2.3.

Under CBN Implication, λ-abstractions, and application are defined in the following way:

$$\begin{array}{lcl} A \to B & := & (\neg A) \vee B \\ \lambda x.t & := & (\mathsf{inl}\,(\,[x.((\mathsf{inr}\,t)\cdot\alpha)]not)\cdot\alpha).\alpha \\ t\,k & := & [\,not[t], k] \end{array}$$

Under CBV Implication, λ-abstractions, and application are defined in the following way:

$$\begin{array}{lcl} A \to B & := & \neg(A \wedge \neg B) \\ \lambda x.t & := & [z.(z \cdot fst\,(x.(z \cdot snd(\,not[t])))))]not \\ t\,k & := & not[\langle t, [k]not\rangle] \end{array}$$

Notice that the two ways of defining implication in the previous definition are duals. Wadler used the dual calculus to show that CBV is dual to CBN in [140] just like Curien and Herbelin did in [44]. However, in a follow up paper Wadler showed that his duality of the dual calculus into itself is an involution [141]. This was a step further than Selinger. While Curien and Herbelin's duality was an involution they did not prove it. In his follow up paper Wadler also showed that the CBV $\lambda\mu$-calculus is dual to the CBN $\lambda\mu$-calculus by translating it into the dual calculus and taking the dual of the translation, and then translating back to the $\lambda\mu$-calculus.

CHAPTER 4

DEPENDENT TYPE THEORY

All the type theories we have seen thus far consist of what are called "non-dependent types". These are types which do not depend on terms. System F^{ω} is an advance where there is a copy of STLC at the type level, but this is not a dependency, hence, system F^{ω} is still simply typed. So it is natural to wonder if it is beneficial to allow types to depend on terms. The answer it turns out is yes. Much like the history of System F, dependent types came out of two fields: programming language research and mathematical logic. As we mentioned above, the first practical application of the computational trinity was a system called Automath which was pioneered by de Bruijn in the 1970's [27]. It also turns out that Automath's core type theory employed dependent types, and many claim it to be the beginning of the research area under the umbrella term "dependent type theory". Since the work of de Bruijn a large body of research on dependent type theory has been conducted. We start with the work of Per Martin-Löf.

4.1 Martin-Löf's Type Theory

Martin-Löf is a Swedish mathematical logician and philosopher who was interested in defining a constructive foundations of mathematics. The foundation he defined he called Type Theory, but what is now referred to as Martin-Löf's Type Theory [53, 89]. It is considered the first full dependent type theory. Type Theory is defined by giving a syntax and deriving three judgments. Martin-Löf placed

$$
\begin{array}{lcl}
S & ::= & \mathsf{Type} \mid \mathsf{True} \\
T, A, B, C & ::= & X \mid \top \mid \ \bot \ \mid A + B \mid \Pi x : A.B \mid \Sigma x : A.B \\
t, s, a, b, c & ::= & x \mid \mathsf{tt} \mid \lambda x : A.t \mid t_1\ t_2 \mid (t_1, t_2) \mid \mathsf{case}\ s\ \mathsf{of}\ x, y.t \mid \mathsf{case}\ s\ \mathsf{of}\ x.t_1, y.t_2 \mid \\
 & & \mathsf{abort}
\end{array}
$$

Figure 24. The syntax of Martin-Löf's Type Theory

particular attention to judgments. In Type Theory types can be considered as specifications of programs, propositions, and sets. Martin-Löf then stresses that one cannot know the meaning of a type without first knowing what its canonical members are, knowing how to construct larger members from the canonical members, and being able to tell when two types are equal. To describe this meaning he used judgments. The judgments are derived using inference rules just as we have seen, and they tell us exactly which elements are canonical and which can be constructed from smaller members. There is also an equality judgment which describes how to tell when two terms are equal. Martin-Löf's Type Theory came in two flavors: intensional type theory and extensional type theory. The difference amounts to equality types and whether the equality judgment is distinct from the propositional equality or not. The impact of intensional vs extensional is quite profound. The latter can be given a straightforward categorical model, while the former cannot. We first define a basic core of Martin-Löf's Type Theory and then we describe how to make it intensional and then extensional.

The syntax of Martin-Löf's Type Theory is defined in Figure 24. The language

consists of sorts S denoted Type and True. The sort Type is a type universe and has as inhabitants types. It is used to classify which things are valid types. The sort True will be used when treating types as propositions to classify which formulas are true. The second part of the language are types T. Types consist of propositional variables X, true or top $\top$, false or bottom $\bot$, sum types $A + B$ which correspond to constructive disjunction, dependent products $\Pi x : A.B$ which correspond to function types, universal quantification, and implication, and disjoint union $\Sigma x : A.B$ which correspond to pairs, constructive conjunction and existential quantification. We can see that dependent products and disjoint union bind terms in types, hence, types do depend on terms. The third and final part of the language are terms. We only comment on the term constructs we have not seen before. The term tt is the inhabitant of $\top$ and is called unit. We have a term which corresponds to a contradiction called abort. Finally, we have two case constructs: case s of $x, y.t$ and case s of $x.t_1, y.t_2$. The former is the elimination form for disjoint union and says if s is a pair then substitute the first projection for x in t and the second projection for y in t. Having the ability to project out both pieces of a pair results in the disjoint union also called Σ-types being strong. A weak disjoint union type is one in which only the first projection of a pair is allowed . The second case construct case s of $x.t_1, y.t_2$ is the elimination form for the sum type. This says that if s is a term of type $A + B$, but is an inhabitant of the type A then substitute a for x in t_1, or if s is an inhabitant of B substitute it for y in t_2. This we will see is the elimination form for constructive disjunction.

In dependent type theory we replace arrow types $A \to B$ with dependent

product types $\Pi x : A.B$, where B is allowed to depend on x. It turns out that we can define arrow types as $\Pi x : A.B$ when x is free in B; that is, B does not depend on x. We will often abbreviate this by $A \to B$. Recall that the arrow type corresponds to implication. The dependent product type also corresponds to universal quantification, because it asserts for all terms of type A we have B, or for all proofs of the proposition A we have B. Additionally, in dependent type theory we replace cartesian product $A \times B$ by disjoint unions $\Sigma x : A.B$ where B may depend on x. The inhabitants of this type are pairs (a, b) where b may depend on a. Now simple pairs can be defined just like arrow types are defined using product types. The type $A \times B$ is defined by $\Sigma x : A.B$ where B does not depend on x. Then b in the pair (a, b) does not depend on a. We can define projections for simple pairs as follows:

$$\begin{array}{lcl} \pi_1\, t & := & \mathsf{case}\, t\, \mathsf{of}\, x, y.x \\ \pi_2\, t & := & \mathsf{case}\, t\, \mathsf{of}\, x, y.y. \end{array}$$

The kinding rules are defined in Figure 25. These rules derive the judgment $\Gamma \vdash T : \mathsf{Type}$ which describes all well-formed types – inhabitants of Type. Now types are also propositions of intuitionistic logic. The judgment $\Gamma \vdash T : \mathsf{True}$ describes which propositions are true constructively. The rules deriving this judgment are in Figure 26. Note that while $\bot$ is a type, it is not a true proposition. This judgment validates the correspondence between types and propositions. In fact we could have denoted $\Pi x : A.B$ as $\forall x : A.B$ and $\Sigma x : A.B$ as $\exists x : A.B$. The typing rules are defined in Figure 27. We include the typing rules for the derived forms for arrow types and cartesian products. These can be derived as well. The rules here are straightforward, so we only comment on the elimination rule for sum types. The rule is defined as

$$\frac{}{\Gamma \vdash \bot : \mathsf{Type}}\ \text{K\_Bottom} \qquad \frac{}{\Gamma \vdash \top : \mathsf{Type}}\ \text{K\_Unit} \qquad \frac{\Gamma, x : A \vdash B : \mathsf{Type}}{\Gamma \vdash \Sigma x : A.B : \mathsf{Type}}\ \text{K\_Ext}$$

$$\frac{\Gamma \vdash A : \mathsf{Type} \quad \Gamma \vdash B : \mathsf{Type}}{\Gamma \vdash A \times B : \mathsf{Type}}\ \text{K\_Prod} \qquad \frac{\Gamma, x : A \vdash B : \mathsf{Type}}{\Gamma \vdash \Pi x : A.B : \mathsf{Type}}\ \text{K\_Pi}$$

$$\frac{\Gamma \vdash A : \mathsf{Type} \quad \Gamma \vdash B : \mathsf{Type}}{\Gamma \vdash A \to B : \mathsf{Type}}\ \text{K\_Arrow} \qquad \frac{\Gamma \vdash A : \mathsf{Type} \quad \Gamma \vdash B : \mathsf{Type}}{\Gamma \vdash A + B : \mathsf{Type}}\ \text{K\_Coprod}$$

Figure 25. Kinding for Martin-Löf's Type Theory

$$\frac{}{\Gamma \vdash \top : \mathsf{True}}\ \text{LTrue} \qquad \frac{\Gamma \vdash A : \mathsf{True} \quad \Gamma \vdash B : \mathsf{True}}{\Gamma \vdash A \times B : \mathsf{True}}\ \text{LProd} \qquad \frac{\Gamma, x : A \vdash B : \mathsf{True}}{\Gamma \vdash \Pi x : A.B : \mathsf{True}}\ \text{LForalli}$$

$$\frac{\Gamma \vdash t : A \quad \Gamma \vdash \Pi x : A.B : \mathsf{True}}{\Gamma \vdash [t/x]B : \mathsf{True}}\ \text{LForalle} \qquad \frac{\Gamma, X : \mathsf{True} \vdash A : \mathsf{True}}{\Gamma \vdash X \to A : \mathsf{True}}\ \text{LImpi}$$

$$\frac{\Gamma \vdash X : \mathsf{True} \quad \Gamma \vdash X \to A : \mathsf{True}}{\Gamma \vdash A : \mathsf{True}}\ \text{LImpe} \qquad \frac{\Gamma \vdash A : \mathsf{True}}{\Gamma \vdash A + B : \mathsf{True}}\ \text{LOri1} \qquad \frac{\Gamma \vdash B : \mathsf{True}}{\Gamma \vdash A + B : \mathsf{True}}\ \text{LOri2}$$

$$\frac{\Gamma \vdash A + B : \mathsf{True} \quad \Gamma, A : \mathsf{True} \vdash C : \mathsf{True} \quad \Gamma, B : \mathsf{True} \vdash C : \mathsf{True}}{\Gamma \vdash C : \mathsf{True}}\ \text{LOre} \qquad \frac{\Gamma \vdash t : A \quad \Gamma \vdash [t/x]B : \mathsf{True}}{\Gamma \vdash \Sigma x : A.B : \mathsf{True}}\ \text{LExti}$$

$$\frac{x \text{ fresh in } C \quad \Gamma \vdash \Sigma x : A.B : \mathsf{True} \quad \Gamma, x : A, B : \mathsf{True} \vdash C : \mathsf{True}}{\Gamma \vdash C : \mathsf{True}}\ \text{LExte}$$

Figure 26. Validity for Martin-Löf's Type Theory

$$\frac{}{\Gamma \vdash \mathsf{tt} : \top}\ \text{Unit} \qquad \frac{\Gamma \vdash A : \mathsf{Type}}{\Gamma, x : A \vdash x : A}\ \text{Var} \qquad \frac{\Gamma \vdash t : A \quad \Gamma \vdash b : [t/x]B}{\Gamma \vdash (t, b) : \Sigma x : A.B}\ \text{Sum}$$

$$\frac{\begin{array}{l} x, y \text{ fresh in } C \\ \Gamma \vdash s : \Sigma x : A.B \\ \Gamma, x : A, y : B \vdash c : C \end{array}}{\Gamma \vdash \mathsf{case}\ s\ \mathsf{of}\ x, y.c : C}\ \text{Case1} \qquad \frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B}{\Gamma \vdash (a, b) : A \times B}\ \text{Prod}$$

$$\frac{\Gamma \vdash c : A \times B}{\Gamma \vdash \pi_1\ c : A}\ \text{Prod1} \qquad \frac{\Gamma \vdash c : A \times B}{\Gamma \vdash \pi_2\ c : B}\ \text{Prod2} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A.t : \Pi x : A.B}\ \text{Pi}$$

$$\frac{\begin{array}{l} \Gamma \vdash t' : A \\ \Gamma \vdash t : \Pi x : A.B \end{array}}{\Gamma \vdash t\ t' : [t'/x]B}\ \text{App1} \qquad \frac{\begin{array}{l} x \text{ fresh in } B \\ \Gamma, x : A \vdash t : B \end{array}}{\Gamma \vdash \lambda x : A.t : A \to B}\ \text{Arrow}$$

$$\frac{\begin{array}{l} \Gamma \vdash t' : A \\ \Gamma \vdash t : A \to B \end{array}}{\Gamma \vdash t\ t' : B}\ \text{App2} \qquad \frac{\Gamma \vdash B : \mathsf{Type} \quad \Gamma \vdash a : A}{\Gamma \vdash a : A + B}\ \text{CoProd1} \qquad \frac{\Gamma \vdash A : \mathsf{Type} \quad \Gamma \vdash b : B}{\Gamma \vdash b : A + B}\ \text{CoProd2}$$

$$\frac{\begin{array}{l} \Gamma \vdash s : A + B \\ \Gamma, x : A \vdash c : C \\ \Gamma, y : B \vdash c' : C \end{array}}{\Gamma \vdash \mathsf{case}\ s\ \mathsf{of}\ x.c, y.c' : C}\ \text{Case2} \qquad \frac{\Gamma \vdash A : \mathsf{Type}}{\Gamma, x :\bot \vdash \mathsf{abort} : A}\ \text{Abort}$$

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash a = b : A}{\Gamma \vdash b : A}\ \text{Conv}$$

Figure 27. Typing Rules for Martin Löf's Type Theory

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash a = a : A}\ \text{Eq\_Refl} \qquad \frac{\Gamma \vdash a = b : A}{\Gamma \vdash b = a : A}\ \text{Eq\_Sym} \qquad \frac{\Gamma \vdash a = b : A \quad \Gamma \vdash b = c : A}{\Gamma \vdash a = c : A}\ \text{Eq\_Trans}$$

$$\frac{\Gamma \vdash a : \top}{\Gamma \vdash a = \mathsf{tt} : \top}\ \text{Eq\_Unit} \qquad \frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B}{\Gamma \vdash a = \pi_1\,(a, b) : A}\ \text{Eq\_Fst}$$

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B}{\Gamma \vdash b = \pi_2\,(a, b) : B}\ \text{Eq\_Snd} \qquad \frac{\Gamma \vdash t : A \quad \Gamma, x : A \vdash b : B}{\Gamma \vdash (\lambda x : A.b)\,t = [t/x]b : [t/x]B}\ \text{Eq\_Beta}$$

$$\frac{\Gamma \vdash t_1 : \Pi x : A.B}{\Gamma \vdash t_1 = \lambda x : A.(t_1\,x) : \Pi x : A.B}\ \text{Eq\_Eta}$$

$$\frac{\Gamma \vdash a : A \quad \Gamma, x : A \vdash c : C \quad \Gamma, y : B \vdash c' : C}{\Gamma \vdash \mathsf{case}\ a\ \mathsf{of}\ x.c, y.c' = [a/x]c : [a/x]C}\ \text{Eq\_Case1}$$

$$\frac{\Gamma \vdash b : B \quad \Gamma, x : A \vdash c : C \quad \Gamma, y : B \vdash c' : C}{\Gamma \vdash \mathsf{case}\ b\ \mathsf{of}\ x.c, y.c' = [b/x]c' : [b/x]C}\ \text{Eq\_Case2}$$

$$\frac{\Gamma \vdash t : T \quad \Gamma \vdash a : [t/x]A \quad \Gamma, x : T, y : A \vdash b : B}{\Gamma \vdash \mathsf{case}\,(t, a)\ \mathsf{of}\ x, y.b = [t/x][a/y]b : [t/x][a/y]B}\ \text{Eq\_Case3}$$

Figure 28. Equality for Martin-Löf's Type Theory

$$\frac{\begin{array}{l}\Gamma \vdash s : A + B \\ \Gamma, x : A \vdash c : C \\ \Gamma, y : B \vdash c' : C\end{array}}{\Gamma \vdash \mathsf{case}\ s\ \mathsf{of}\ x.c, y.c' : C}\ \text{Case2}.$$

We mentioned above that this rule corresponds to the elimination form for constructive disjunction. This rule tells us that to eliminate $A \vee B$ we must assume A and prove C and then assume B and prove C, but this is exactly what the above rule tells us. The computational correspondence is that the case construct gives us away to case split over terms of two types.

As it stands Martin-Löf's Type Theory is a very powerful logic. The axiom of choice must be an axiom of set theory, because it cannot be proven from the other axioms. The axiom of choice states that the cartesian product of a family of non-empty sets is non-empty. Martin-Löf showed in [89] that the axiom of choice can be proven with just the theory we have defined thus far in this section. Thus, one could also prove the well-ordering theorem. This is good, because it shows that Type Theory is powerful enough to be a candidate for a foundation of mathematics. This is also good for dependent type based verification, because we can formulate expressive specifications of programs.

The final judgment of Martin-Löf's Type Theory is the definitional equality judgment. It has the form $\Gamma \vdash a = b : A$. The rules deriving this judgment tell us when we can consider two terms as being equal. Then two types whose elements are equal based on this judgment are equal. The equality rules are defined in Figure 28 where we leave the congruence rules implicit for presentation purposes.

These rules look very much like full β-reduction, but these are equalities. They are symmetric, transitive, and reflexive unlike reduction which is not symmetric. This is a definitional equality and it can be used during type checking implicitly at will using the following rule:

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash a = b : A}{\Gamma \vdash b : A}\ \text{Conv}$$

We now describe when Martin-Löf's Type Theory is extensional or intensional.

Extensional Type Theory. In extensional type theory our equality judgment is not distinct from propositional equality. To make Martin-Löf's type theory extensional we add the following rules:

$$\begin{array}{ll} \textit{Kinding} & \textit{Typing} \\ \dfrac{\Gamma \vdash A : \mathsf{Type} \quad \Gamma \vdash a : A \quad \Gamma \vdash b : A}{\Gamma \vdash \mathsf{Id}\ A\ a\ b : \mathsf{Type}} & \dfrac{\Gamma \vdash a = b : A}{\Gamma \vdash \mathsf{tt} : \mathsf{Id}\ A\ a\ b} \quad \dfrac{\Gamma \vdash t : \mathsf{Id}\ A\ a\ b}{\Gamma \vdash a = b : A} \end{array}$$

Using these rules we can prove all of the usual axioms of identity: reflexivity, transitivity, and symmetry [89]. Notice that these rules collapse definitional equality into propositional equality. The right most typing rule is where extensional type theory gets its power. This rule states that propositional equations can be used interchangeably anywhere. This power comes with sacrifice, some meta-theoretic properties one may wish to have like termination of equality and decidability of type checking no longer hold [128, 129].

Intensional Type Theory. Now to make Martin-Löf's Type Theory intensional we add the following rules:

Kinding:

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash \mathsf{r}(a) : \mathsf{Id}\ A\ a\ a}$$

Typing:

$$\frac{\Gamma \vdash c : \mathsf{Id}\ A\ a\ b \qquad \Gamma, x : A \vdash d : B(x, x, \mathsf{r}(x)) \qquad \Gamma, x : A, y : A, z : \mathsf{Id}\ A\ x\ y \vdash B(x, y, z) : \mathsf{Type}}{\Gamma \vdash \mathsf{J}(d, a, b, c) : B(a, b, c)}$$

Equality:

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash \mathsf{J}(d, a, a, \mathsf{r}(a)) = d\ a : B(a, a, \mathsf{r}(a))}$$

As we can see here propositional equality is distinct from the definitional equality judgment. In the above rules $\mathsf{r}(a)$ is the constant denoting reflexivity, and $\mathsf{J}(a, b, c, d)$ is just an annotation on d with all the elements of the equality. Using these we can prove reflexivity, transitivity, and symmetry. We do not go into any more detail here between intensional and extensional type theory, but a lot of research has gone into understanding intensional type theory. Models of intensional type theory are more complex than extensional type theory. Recently, there has been an upsurge of interest in intensional type theory due to a new model for type theory where types are interpreted as homotopies [16]. See [128, 129, 68, 70] for more information.

We said at the beginning of this section that we would only define a basic core of Martin-Löf's type theory. We have done both for intensional Type Theory and extensional Type Theory, but Martin-Löf included a lot more than this in his classic paper [89]. He included ways of defining finite types as well as arbitrary infinite types called universes much like Type. He also included rules for defining inductive types which in the design of programming languages are very useful [48].

The universe Type contains all well-formed types. It is quite natural to think of Type as a type itself. This is called the Type : Type axiom. In fact Martin-Löf did that in his original theory, but Girard was able to prove that such an axiom destroys

the consistency of the theory. Girard was able to define the Burali-Forti paradox in Type Theory with Type : Type [35, 36]. Now Type : Type is inconsistent when the type theory needs to correspond to logic, but if it is used purely for programming it is a very nice feature. It can be used in generic programming [28]. In Martin-Löf's Type Theory without the Type : Type axiom types are program specifications, hence, the theory can be seen as a terminating functional programming language [97].

4.2 The Calculus of Constructions

An entire class of type theories called Pure Type Systems may be expressed by a very simple core type theory, a set of type universes called sorts, a set of axioms, and a set of rules. The rules specify how the sorts are to be used, and govern what dependencies are allowed in the type theory. There is a special class of eight pure type systems with only two sorts called $\Box$ and $*$[1] called the λ-cube [17]. The following expresses the language of this class of types theories.

Definition 4.2.0.1.

The language of the λ-cube:

$$t, a, b \quad ::= \quad \Box \mid * \mid c \mid x \mid t_1\ t_2 \mid \lambda x : t_1.t_2 \mid \Pi x : t_1.t_2$$

Notice in the previous definition that terms and types are members of the same language. They are not separated into two syntactic categories. This is one of the beauties of pure type systems. They have a really clean syntax, but this beauty comes with a cost. Some collapsed type theories are very hard to reason about.

[1] It is also standard to called these Type and *Prop* respectively. Type is the same as we have seen above and Prop classifies logical propositions.

A pure type system is defined as a triple (S, A, R), where S is a set of sorts and is a subset of the constants of the language, A is a set of axioms, and R is a set of rules. In the λ-cube $\{*, \Box\} \subseteq S$, $A = \{(*, \Box)\}$, and R varies depending on the system. The axioms stipulate which sorts the constants of the language have. In the λ-cube there are at least two constants $\Box$ and $*$. The set of rules are subsets of the set $\{(*,*), (*,\Box), (\Box,*), (\Box,\Box)\}$. These rules represent four forms of dependencies:

i. terms depend on terms: $(*,*)$
ii. terms depend on types: $(\Box,*)$
iii. types depend on terms: $(*,\Box)$
iv. types depend on types: $(\Box,\Box)$

In the λ-cube terms always depend on terms, hence $(*,*) \in R$ for any system. For example, $(\lambda x : t.a)\, b$ is a term depending on a term and $\lambda x : \mathsf{Type}.b$ where b is a type is a type depending on a type. An example of a term depending on a type is the Λ-abstraction of system F. Finally, an example of a type depending on a term is the product type of Martin-Löf's Type Theory. Now using the notion of dependency we define the core set of inference rules in the next definition.

Definition 4.2.0.2.

Given a system of the λ-cube (S, A, R) the inference rules are defined as follows:

$$\frac{}{(\lambda x : t.b)\, a \rightsquigarrow [a/x]b}\ \text{Beta} \qquad \frac{(c,s) \in A}{\cdot \vdash c : s}\ \text{Axioms}$$

$$\frac{\Gamma \vdash a : s}{\Gamma, x : a \vdash x : a}\ \text{Var} \qquad \frac{\Gamma \vdash a : b \qquad \Gamma \vdash a' : s \qquad (x : a') \notin \Gamma}{\Gamma, x : a' \vdash a : b}\ \text{Weakening}$$

$$\frac{\Gamma \vdash a' : b \qquad \Gamma \vdash a : \Pi x : b.b'}{\Gamma \vdash a\, a' : [a'/x]b'}\ \text{App}$$

$$\frac{\Gamma \vdash a : b \qquad \Gamma \vdash b' : s \qquad b \equiv_\beta b'}{\Gamma \vdash a : b'}\ \text{Conv} \qquad \frac{\Gamma \vdash a : s_1 \qquad \Gamma, x : a \vdash b : s_2 \qquad (s_1, s_2) \in R}{\Gamma \vdash \Pi x : a.b : s_2}\ \text{Pi}$$

$$\frac{\Gamma, x : a \vdash b : b' \qquad \Gamma \vdash \Pi x : a.b' : s}{\Gamma \vdash \lambda x : a.b : \Pi x : a.b'}\ \text{Lam}$$

In the previous definition s ranges over S, and we left out the congruence rules for reduction to make the definition more compact. However, either they need to be added or evaluation contexts do, for a full treatment of reduction. It turns out that this is all we need to define every intuitionistic type theory we have defined in this part of the thesis including a few others we have not defined. However, Martin-Löf's Type Theory is not definable as a pure type system. Taking the set R to be $\{(*,*)\}$ results in STLC. System F results from taking the set $R = \{(*,*),(\Box,*)\}$. System F^ω is definable by the set $R = \{(*,*),(\Box,*),(\Box,\Box)\}$.

A good question now to ask is what type theory results from adding all possible rules to R? That is what type theory is defined by $R = \{(*,*),(*,\Box),(\Box,*),(\Box,\Box)\}$? This type theory is clearly a dependent type theory and is called the Calculus of Constructions (CoC). It was first defined by Thierry Coquand in [38]. It is the most powerful of all the eight pure type system in the λ-cube. We have seen one formulation of CoC as a pure type system, but we give one more.

It turns out that CoC is really just an extension of system F^ω. We do not have to define it using a collapsed syntax – even though it is prettier. We call the extension of system F^ω to CoC separated CoC to distinguish it from the collapsed versions. The syntax for separated CoC is in Figure 29[2]. This formulation simply extends system

[2]We do not have a citation for where this formulation can be found. It was learned

$$
\begin{array}{lcl}
S & ::= & * \\
K & ::= & \mathsf{Type} \mid \Pi X : K.K' \mid \Pi x : T.K \\
T & ::= & X \mid \lambda X : K.T \mid \lambda x : T_1.T_2 \mid T_1\ T_2 \mid T\ t \mid \Pi X : K.T \mid \Pi x : T.T' \\
t & ::= & x \mid \lambda x : T.t \mid \lambda X : K.t \mid t_1\ t_2 \mid t\ T
\end{array}
$$

Figure 29. Syntax for the Separated Calculus of Constructions

$$
\frac{}{\Gamma \vdash \mathsf{Type} : *}\ \text{S\_Type} \qquad \frac{\Gamma, X : K \vdash K' : *}{\Gamma \vdash \Pi X : K.K' : *}\ \text{S\_Prod1} \qquad \frac{\Gamma, x : T \vdash K : *}{\Gamma \vdash \Pi x : T.K : *}\ \text{S\_Prod2}
$$

Figure 30. Sorting Rules for the Separated Calculus of Constructions

F^ω with dependency. Due to the separation of the language we increase the number of judgments. We now have four judgments: sorting, kinding, typing, and equality. They are defined as one would expect. We do not go into much detail here. The sorting rules are defined in Figure 30, the kinding rules are defined in Figure 31, the typing rules in Figure 32, and finally the equality rules in Figure 33. We can see that this formulation makes sense from the PTS perspective, because system F^ω has the following set of rules $R = \{(*, *), (\Box, *), (\Box, \Box)\}$ and we need to make it dependent. That is we need types to depend on terms. So we add $(*, \Box)$ to R and we obtain CoC.

We have now introduced every type theory we need for the remainder of this

by the author from Hugo Herbelin at the 2011 Oregon Programming Language Summer School.

$$\frac{}{\Gamma, X : \mathsf{Type}, \Gamma' \vdash X : \mathsf{Type}} \text{ K\_Var} \qquad \frac{\Gamma, x : T_1 \vdash T_2 : \mathsf{Type}}{\Gamma \vdash \Pi x : T_1 . T_2 : \mathsf{Type}} \text{ K\_Prod1}$$

$$\frac{\Gamma, X : K \vdash T : \mathsf{Type}}{\Gamma \vdash \Pi X : K . T : \mathsf{Type}} \text{ K\_Prod2} \qquad \frac{\Gamma \vdash T_1 : \mathsf{Type} \quad \Gamma, x : T_1 \vdash T_2 : K}{\Gamma \vdash \lambda x : T_1 . T_2 : \Pi x : T_1 . K} \text{ K\_Lam1}$$

$$\frac{\Gamma \vdash K_1 : * \quad \Gamma, X : K_1 \vdash T : K_2}{\Gamma \vdash \lambda X : K_1 . T : \Pi X : K_1 . K_2} \text{ K\_Lam2} \qquad \frac{\Gamma \vdash T_1 : \Pi X : K_1 . K_2 \quad \Gamma \vdash T_2 : K_1}{\Gamma \vdash T_1\ T_2 : K_2} \text{ K\_App1}$$

$$\frac{\Gamma \vdash T : \Pi x : T . K \quad \Gamma \vdash t : T}{\Gamma \vdash T\ t : [t/x]K} \text{ K\_App2}$$

Figure 31. Kinding Rules for the Separated Calculus of Constructions

$$\frac{\Gamma \vdash T : \mathsf{Type}}{\Gamma, x : T, \Gamma' \vdash x : T} \text{ Var} \qquad \frac{\Gamma \vdash T_1 : \mathsf{Type} \quad \Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x : T_1 . t : \Pi x : T_1 . T_2} \text{ Lam}$$

$$\frac{\Gamma \vdash t_1 : \Pi x : T_1 . T_2 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\ t_2 : [t_2/x]T_2} \text{ App} \qquad \frac{\Gamma \vdash K : * \quad \Gamma, X : K \vdash t : T}{\Gamma \vdash \lambda X : K . t : \Pi X : K . T} \text{ TypeAbs}$$

$$\frac{\Gamma \vdash T : K \quad \Gamma \vdash t : \Pi X : K . T'}{\Gamma \vdash t\ T : [T/X]T'} \text{ TypeApp} \qquad \frac{T_1 \approx T_2 \quad \Gamma \vdash t : T_1}{\Gamma \vdash t : T_2} \text{ Conv}$$

Figure 32. Typing Rules for the Separated Calculus of Constructions

$$\frac{}{(\lambda x : T.t)\, t' \approx [t'/x]t}\ \text{R\_Beta1} \qquad \frac{}{(\lambda X : K.t)\, T \approx [T/X]t}\ \text{R\_Beta2}$$

$$\frac{}{(\lambda x : T.T')\, t \approx [t/x]T}\ \text{R\_Beta3} \qquad \frac{}{(\lambda X : K.T)\, T' \approx [T'/X]T}\ \text{R\_Beta4}$$

$$\frac{t \approx t'}{\lambda x : T.t \approx \lambda x : T.t'}\ \text{R\_Lam1} \qquad \frac{T \approx T'}{\lambda X : K.T \approx \lambda X : K.T'}\ \text{R\_Lam2}$$

$$\frac{T \approx T'}{\lambda x : A.T \approx \lambda x : A.T'}\ \text{R\_Lam3} \qquad \frac{t_1 \approx t_1'}{t_1\ t_2 \approx t_1'\ t_2}\ \text{R\_App1} \qquad \frac{t_2 \approx t_2'}{t_1\ t_2 \approx t_1\ t_2'}\ \text{R\_App2}$$

$$\frac{t_1 \approx t_2}{T\ t_1 \approx T\ t_2}\ \text{R\_App3} \qquad \frac{T_1 \approx T_2}{T_1\ t \approx T_2\ t}\ \text{R\_App4} \qquad \frac{t \approx t'}{t\ T \approx t'\ T}\ \text{R\_TypeApp1}$$

$$\frac{T_1 \approx T_2}{T\ T_1 \approx T\ T_2}\ \text{R\_TypeApp2} \qquad \frac{T_1 \approx T_2}{T_1\ T \approx T_2\ T}\ \text{R\_TypeApp3}$$

Figure 33. The Equality for the Separated Calculus of Constructions

part of thesis. So far we have taken a trip down the rabbit hole of type theory, from the early days of type theory all the way to modern type theory. It is now time to see what we can use these for in programming language research.

CHAPTER 5

DEPENDENT TYPES IN PRACTICE

Type theories are wonderful core languages for programming languages. Many programming languages have been created based on type theories. Some examples are Haskell, OCaml, ML, ACL2, Isabelle, Coq, Agda, Epigram, Guru and Idris. In fact there has been an entire book written on using type theory for programming by Benjamin Pierce [104]. Programming languages are defined with a goal in mind. Some programming languages are general purpose languages and others are domain specific. For example, ML, Haskell, Epigram, Idris, Guru and OCaml are examples of general purpose programming languages. ACL2, Isabelle, Coq, and Agda are more domain specific, because they are proof assistants. New programming languages are designed usually to provide new advancements in the field. These advancements usually arise from programming language research or research on type theory. In this section we discuss the current applications of dependent type theories in both proof assistants and as cores to general purpose functional programming languages. We also discuss and give some motivation for using dependent types in the design of general purpose functional programming languages.

The latest big advancement that has resulted in a surge of new language designs is using dependent types to verify properties of programming languages. To cite just a few references [139, 26, 90, 9, 98, 130, 87]. Now dependent types are very powerful and provide a rich environment for verification, but they are very hard to

reason about. So it is natural to wonder if we can obtain some of the features of dependent type theories without having to adopt full dependent types. This is the chosen path the inventors of Haskell took. Tim Sheard showed that an extension of system F^{ω} is a strong enough type theory to obtain some features that dependent types yield [123]. He defined a language called Ω which is based off of system F^{ω}. It has also been shown that system F^{ω} extended with the natural numbers can be used to state some nice properties of programs. One example is checking array out of bounds violations during type checking versus during run-time. The kind of types which allow the encoding of these types of features are called indexed types and are investigated in [56, 151]. Indexed types are just types which depend on some data. However, this data is in no way connected to the language of terms. This directly implies some indexed types are indeed definable in system F^{ω}, while other indexed types may require an extension of the type language with other typing features, e.g. existential types, natural numbers, etc. It has been conjectured that indexed types may be computationally as powerful as dependent types. However, to make indexed types as strong as dependent types the resulting type system would be very cluttered. One would have to add a lot of new operators and duplications at the type level. Another approach that provides dependent like features to a simple type theory are Generalized Algebraic Data Types (GADT) [147]. These have been added to Haskell [72]. These provide a way of guarding recursive data types. The main feature of GADTs are enforcement of local constraints.

There are alternatives to type-based verification. A large body of work has

been done using model checking and testing to verify correctness of programs we cite only a few [11, 15, 39, 78, 150]. However, these are external tools while dependent types are part of the programming language itself. There has been some work on automated theorem proving using dependent types. Alasdair Armstrong shows in [14] that automated theorem provers can work in harmony with dependent type theory. One thing this accomplishes is that repetitive trivial proofs can be done automatically. This work also shows that the research on dependent type theory benefits from the work on automated theorem proving. We believe that dependent type theories are, however, the answer. They are more or just as powerful as the alternatives in a concise and elegant fashion. They can be used as the core of proof assistants, general purpose programming languages, domain specific languages, and an entire arsenal of features can be encoded in them.

There are several well-known proof assistants based on dependent type theory. One of the first is called NuPrl which is based on Martin-Löf's Type Theory [34]. The proof assistant Coq is another proof assistant and is based on an extension of Coquand's CoC called the Calculus of Inductive Constructions [134]. Coq has been used to verify the correctness of very large scale mathematics and programs. The proof of the four colour theorem has been fully checked in it [62]. A C compiler has been formally verified with in Coq [85, 84]. This project is called CompCert. Agda is the second proof assistant based on Martin-Löf's Type Theory. However, we are not aware of any large scale mathematics in Agda. Finally, Twelf is a proof assistant based on a restricted version of Martin-Löf's Type Theory called LF [102].

More information on proof assistants can be found in [58]. These projects show that dependent types are powerful enough to do real-world large scale mathematics, but what about general purpose programming languages?

The number one application of dependent types in general purpose programming languages is type based verification of programs. Hongewi Xi has done a large amount of work on this topic. He has shown that array bounds checks can be eliminated when using dependent types [148]. They can be eliminated by defining the type of arrays to include their size. Then all programs which manipulate arrays must respect the arrays size constraints which are encoded in the type. Xi shows in [146] that dependent types can be used to eliminate dead code in the form of unreachable branches of case-expressions. He derives constraints based on the patterns and the type of the function being defined. Then through type checking branches can actually be eliminated. All of this and more can be found in Xi's thesis [149].

One promising idea is to take a very expressive dependent type theory and add general recursion and $\mathsf{Type} : \mathsf{Type}$. Then, either identify through a judgment or syntactically identify a sublanguage of the dependent type theory which is consistent. This consistent sublanguage will correspond to a logic by computational trinity, and is called the proof fragment. Garrin Kimmel et al. show in [74] that crafting such a type theory where the proof fragment is syntactically separated from the general purpose programming language can be done and provides interesting features. The language they use is called Sep^3 which stands for Separation of Proof and Program. Kimmel uses Sep^3 to verify the correctness of a call-by-value λ-calculus interpreter.

The unique feature of Sep^3 is that it allows for constraints to be verified about non-terminating programs. All the proof assistants we have seen are all terminating. That is, all programs one writes in them are terminating. This makes it very difficult to formalize and verify properties of non-terminating programs. However, Sep^3 is a language which allows for non-terminating programs to be defined in the programming language and even be mentioned in propositions. This is called freedom of speech. It turns out that the proof fragment can be completely erased after type checking. This means that proofs are really just specificational. This erasing is done by defining a meta-level function called the eraser [88]. The erasure was investigated in a similar setting as Sep^3 in [124]. It is then applied to a program after being type checked. This will make running programs more efficient. Sep^3 also contains $\mathsf{Type} : \mathsf{Type}$ this axiom while inconsistent is wonderful for programming. It is shown in [28] that this axiom can be used to encode lots of extra programming features. $\mathsf{Type} : \mathsf{Type}$ is also very useful for generic programming. This axiom, which allows for large eliminations, that is types defined by recursion, allow for the definition of very generic programs. An example is a completely generic zipwith function. This function would take a function of arbitrary arity, two list of equal length, and returns a list of the same lengths as the input, where the operator is applied to the two lists pairwise. This has actually been done in Sep^3 although it was not published.

All this work shows that dependent types need to be in main stream programming. They provide ways to fully verify the correctness of programs thus eliminating bugs. One unique feature of dependent types are that they are first class citizens of

the programming language. This allows for programmers to prove properties of their programs in the same language they wrote them in, thus eliminating the need to learn and use external tools. Dependent type theories correspond to logics by the computational trinity, and can be used to proof check large scale mathematics. Dependent types are the future of programming languages.

CHAPTER 6

METATHEORY OF TYPE THEORIES

In this section we discuss how to reason about type theories at the meta-level. There are many properties that one might wish to prove about a type theory, but the property we will concentrate on is consistency of type theories. The mathematical tools we discuss in this section have many applications not just consistency. However, proving consistency gives a clear view of how to use these mathematical tools.

We have said several times that if a type theory is to correspond to a logic then it must be consistent. Consistency tells us that if a theorem can be proven, then it is true with respect to some semantics. To show a type theory consistent it is enough to show that it is weakly normalizing [126].

Definition 6.0.0.3.

A type theory is weakly normalizing *if and only for all terms t there exists a term t' such that $t \rightsquigarrow^* t'$ and there does not exists any term t'' such that $t' \rightsquigarrow t''$. We call t' a* normal form*.*

Loosely put, based on the computational trinity terms correspond to proofs and reduction corresponds to cut-elimination. We know that if all proofs can be normalized using cut then we know that the logic is consistent. Now Gentzen actually showed that, if all proofs can be normalized using cut elimination no matter what order the cuts are done, then the logic is consistent, but weak normalization still leaves open the possibility that a proof might have an infinite reduction sequence. Based on this

fact some require their type theories to be strongly normalizing.

Definition 6.0.0.4.

A type theory is strongly normalizing *or* terminating *if and only for all terms* t *there are no infinite descending chains beginning with* t*. That is, it is never the case that* $t \rightsquigarrow t_1 \rightsquigarrow t_2 \rightsquigarrow \cdots$.

Strong normalization gives a tighter correspondence with cut elimination then weak normalization, because there are no chances of an infinite cut-elimination process [126]. However, weak normalization is enough. We just need to know that a term can be normalized.

It turns out that for all simply typed type theories weak normalization actually implies strong normalization [125]. This turns out to be quite a profound result, because it is harder to prove strong normalization than it is weak normalization. If weak implies strong then we never have to do the harder proof. There is a long standing conjecture about weak normalization implying strong normalization called the Barendregt-Geuvers-Klop conjecture [126]. They conjectured that for any PTS weak normalization implies strong normalization. Now we already know that weak normalization implies strong normalization for simply typed theories. These are the class of PTS' where their set of rules are a subset of $\{(*,*),(\Box,\Box),(\Box,*),(\Box,\Box)\}$. However, it is unknown whether weak implies strong normalization for the class of dependent PTS'.

Gödel's famous theorems tell us that to prove consistency of a theory one must use a more powerful theory than the one that is being proven consistent. Thus, to

reason about a type theory we translate the theory into a more powerful theory. We call this more powerful theory the semantics of the type theory and it can be thought of as giving meaning to the type theory. The most difficult task is choosing what semantics to give to the type theory under consideration. Throughout the remainder of this section we summarize several possible semantics to give to type theories.

6.1 Hereditary Substitution

In [110] Prawitz shows that using a lexicographic combination of the structural ordering on intuitionistic propositional formulas and the structural ordering on proofs, propositional intuitionistic logic can be proven consistent. This implies that STLC can be proven consistent using the same ordering. Indeed it can be [60, 10, 86]. These proofs have a particular structure and are completely constructive. Kevin Watkins was the first to make their constructive content explicit [143]. He examined these proofs and defined a function called the hereditary substitution function, which captures the constructive content of these proofs. Following Watkins, Robin Adams did the same for dependent types [6].

Intuitively, the hereditary substitution function is just like ordinary capture avoiding substitution except that if as a result of substitution a new redex is introduced, that redex is then recursively reduced. We write $[t/x]^T t'$ for hereditarily substituting t for x of type T into t'. Let's consider an example.

Example 6.1.0.1.

Consider the terms $t \equiv \lambda x : X.x$ *and* $t' \equiv (yz)$. *Then ordinary capture avoiding*

substitution would have the following result:

$$[t/y]t' = (\lambda x : X.x)z.$$

However, hereditary substitution has the following result:

$$[t/y]^{X \to X} t' = z,$$

because hereditary substitution first capture avoidingly substitutes t for y in t' and examines the result. It then sees that a new redex $(\lambda x : T.x)z$ has been created. Then it recursively reduces this redex as follows: $[z/x]^X x$.

Hereditary substitution is important for a number of reasons. It was first used as a means to conduct the metatheory of the type theory LF which is the core of the proof assistant Twelf. LF is based on canonical forms. That is, the language itself does not allow any non-normal forms to be defined. That is $(\lambda x : T.t)\, t'$ is not a valid term in LF. Thus, their operational semantics cannot use ordinary capture avoiding substitution, because as we saw in the above example, we can substitute normal forms into a normal form and end up with a non-normal form. So Watkins used hereditary substitution instead of capture avoiding substitution in their operational semantics [143]. Adams extended this work to dependent types in his thesis [6]. We will show how to use hereditary substitution to show weak normalization. Let's consider how to define hereditary substitution for STLC.

6.2 Hereditary Substitution for STLC

The definition of the hereditary substitution depends on a partial function called *ctype*. This function is equivalent to the *treduce* function used in [143]. It is

defined by the following definition.

Definition 6.2.0.1.

The partial function is defined with respect to a fixed type T and has two arguments, a free variable x, and a term t, where x may be free in t. We define ctype by induction on the form of t.

$$ctype_T(x, x) = T$$

$$ctype_T(x, t_1\ t_2) = T''$$

Where $ctype_T(x, t_1) = T' \to T''$.

The *ctype* function simply computes the type of a term in weak-head normal form. The following lemma states two very important properties of *ctype*. We do not include any proofs here, but they can be found in [51].

Lemma 6.2.0.2.

i. If $ctype_T(x, t) = T'$ *then* $head(t) = x$ *and* T' *is a subexpression of* T.

ii. If $\Gamma, x : T, \Gamma' \vdash t : T'$ *and* $ctype_T(x, t) = T''$ *then* $T' \equiv T''$.

The purpose of *ctype* is to detect when a new redex will be created in the definition of the hereditary substitution function. We define the hereditary substitution function next.

Definition 6.2.0.3.

The following defines the hereditary substitution function for STLC. It is defined by recursion on the form of the term being substituted into and the cut type T.

$$[t/x]^T x = t$$

$[t/x]^T y = y$

Where y is a variable distinct from x.

$[t/x]^T(\lambda y : T'.t') = \lambda y : T'.([t/x]^T t')$

$[t/x]^T(t_1\ t_2) = ([t/x]^T t_1)\ ([t/x]^T t_2)$

Where $([t/x]^T t_1)$ is not a λ-abstraction, or both $([t/x]^T t_1)$ and t_1 are λ-abstractions.

$[t/x]^T(t_1\ t_2) = [([t/x]^T t_2)/y]^{T''} s_1'$

Where $([t/x]^T t_1) \equiv \lambda y : T''.s_1'$ for some y, s_1', and T'' and $ctype_T(x, t_1) = T'' \to T'$.

We can see that every case of the previous definition except the application cases are identical to the definition of capture-avoiding substitution. This is intentional, because the hereditary substitution function should only differ when a new redex is created as a result of a capture-avoiding substitution. The creation of a new redex as a result of a capture-avoiding substitution can only occur when substituting into an application with respect to STLC.

One thing to note about our definition of the hereditary substitution function defined above is that we define it in terms of all terms not just normal forms. This was first done by Harley Eades and Aaron Stump in [50] in their work on using the hereditary substitution function to show normalization of Stratified System F. Secondly, the definition of the hereditary substitution function is nearly total by definition. In fact it is only the second case of application that prevents totality from

being trivial. Now if this case was used we know that $ctype_T(x, t_1) = T'' \to T'$, and by Lemma 6.2.0.2, $T'' \to T'$ is a subexpression of T. This implies that T'' is a strict subexpression on T. So in this case the type decreases by the strict subexpression ordering. In fact we prove totality of the hereditary substitution function for STLC using the lexicographic combination (T, t) of the strict subexpression ordering. This shows that *ctype* reveals information about the types of the input terms to the hereditary substitution function, which allows us to use the well-founded ordering to prove properties of the hereditary substitution function.

We do not want to underplay the importance of the ordering on types. In order to be able to even define the hereditary substitution function and prove that it is indeed a total function one must have an ordering on types. This is very important. Now in the case of STLC the ordering is just the subexpression ordering, while for other systems the ordering can be much more complex. For some type theories no ordering exists on just the types. Whatever ordering we use for the types *ctype* brings this ordering into the definition of the hereditary substitution function.

How do we know when a new redex was created as a result of a capture-avoiding substitution? A new redex was created when the hereditary substitution function is being applied to an application, and if the the hereditary substitution function is applied to the head of the application and the head was not a λ-abstraction to begin with, but the result of the hereditary substitution function was a λ-abstraction. If this is not the case then no redex was created. The first case for applications in the definition of the hereditary substitution function takes care of this situation. Now

the final case for applications handles when a new redex was created. In this case we know applying the hereditary substitution function to the head of the application results in a λ-abstraction and we know *ctype* is defined. So by Lemma 6.2.0.2 we know the head of t_1 is x so t_1 cannot be a λ-abstraction. Thus, we have created a new redex so we reduce this redex by hereditarily substituting $[t/x]^T t_2'$ for for y of type T'' into the body of the λ-abstraction t_1'. We use hereditary substitution here because we may create more redexes as a result of reducing the previously created redex.

In STLC the only way to create redexes is through hereditarily substituting into the head of an application. This is because according to our operational semantics for STLC (full β-reduction) the only redex is the one contracted by the β-rule. If our operational semantics included more redexes we would have more ways to create redexes and the definition of the hereditary substitution function would need to account for this. Hence, the definition of the hereditary substitution function is guided by the chosen operational semantics.

The hereditary substitution function has several properties. First it is a total and type preserving function.

Lemma 6.2.0.4. *Suppose* $\Gamma \vdash t : T$ *and* $\Gamma, x : T, \Gamma' \vdash t' : T'$. *Then there exists a term* t'' *such that* $[t/x]^T t' = t''$ *and* $\Gamma, \Gamma' \vdash t'' : T'$.

The next property is normality preserving, which states that when the hereditary substitution function is applied to normal forms then the result of the hereditary substitution function is a normal form. We state this formally as follows:

Lemma 6.2.0.5. *If* $\Gamma \vdash n : T$ *and* $\Gamma, x : T \vdash n' : T'$ *then there exists a normal term* n'' *such that* $[n/x]^T n' = n''$.

The final property is soundness with respect to reduction.

Lemma 6.2.0.6. *If* $\Gamma \vdash t : T$ *and* $\Gamma, x : T, \Gamma' \vdash t' : T'$ *then* $[t/x]t' \rightsquigarrow^* [t/x]^T t'$.

Soundness with respect to reduction shows that the hereditary substitution function does nothing more than what we can do with the operational semantics and ordinary capture avoiding substitution. All of these properties should hold for any hereditary substitution function, not just for STLC. They are correctness properties that must hold in order to use the hereditary substitution function to show normalization.

We can now prove normalization of STLC using the hereditary substitution function. We first define a semantics for the types of STLC.

Definition 6.2.0.7.

First we define when a normal form is a member of the interpretation of type T *in context* Γ

$$n \in [\![T]\!]_\Gamma \iff \Gamma \vdash n : T,$$

and this definition is extended to non-normal forms in the following way

$$t \in [\![T]\!]_\Gamma \iff t \rightsquigarrow^! n \in [\![T]\!]_\Gamma,$$

where $t \rightsquigarrow^! t'$ *is syntactic sugar for* $t \rightsquigarrow^* t' \not\rightsquigarrow$.

The interpretation of types was inspired by the work of Prawitz in [111] although we use open terms here where he used closed terms. Next we show that the definition of the interpretation of types is closed under hereditary substitutions.

Lemma 6.2.0.8. *If* $n' \in [\![T']\!]_{\Gamma,x:T,\Gamma'}$, $n \in [\![T]\!]_{\Gamma}$, *then* $[n/x]^T n' \in [\![T']\!]_{\Gamma,\Gamma'}$.

Proof. By Lemma 6.2.0.4 we know there exists a term $\hat{n}$ such that $[n/x]^T n' = \hat{n}$ and $\Gamma, \Gamma' \vdash \hat{n} : T'$ and by Lemma 6.2.0.5 $\hat{n}$ is normal. Therefore, $[n/x]^T n' = \hat{n} \in [\![T']\!]_{\Gamma,\Gamma'}$. □

Finally, by the definition of the interpretation of types the following result implies that STLC is normalizing.

Theorem 6.2.0.9. *If* $\Gamma \vdash t : T$ *then* $t \in [\![T]\!]_{\Gamma}$.

Proof. This holds by a straightforward proof by induction on the structure of the assumed typing derivation where the application case uses the previous lemma. □

Corollary 6.2.0.10. *If* $\Gamma \vdash t : T$, *then there exists a normal form* n, *such that* $t \rightsquigarrow^! n$.

This proof method has been applied to a number of different type theories. Eades and Stump show that SSF is weakly normalizing using this proof technique in [50]. The advantage of hereditary substitution is that it shows promise of being less complex than other normalization techniques. For example, when proving normalization using the Tait-Girard reducibility method – discussed in the next section – the main soundness theorem must universally quantify over the domain of well-formed substitutions, but this is not needed in the proof using hereditary substitution. In addition, the interpretation of types do not require advanced mathematical machinerary like the reducibility method where they require quantification over types as well as

recursion over types (large eliminations). This implies that it would be easier to formalize in hereditary substitution proofs in proof assistants. However, it is currently unknown which type theories can be proven normalizing using hereditary substitution.

The type theories currently known to be proven normalizing using the hereditary substitution proof technique are relatively simple. For example, the simply-typed λ-calculus, predicative polymorphic versions of system F, and the LF family of dependent types have all been shown to be weakly-normalizing using hereditary substitution. However, it is currently unknown how to prove normalization of system T using hereditary substitution, because the usual ordering on types is not proof theoretically strong enough. The termination ordinal of the ordering on types would have to be at least ϵ_0, but such an ordering is non-trivial to construct. Furthermore, impredicative theories like system F pose a similar problem. It is currently unknown how to construct the ordering for second order types. The hereditary substitution technique heavily depends on the ordering on types and the lack of one prevent its use. As the type theory becomes more expressive through extensions the harder the ordering on types becomes to construct.

Hereditary substitution can be used to maintain canonical forms and even prove weak normalization of predicative simple type theories. It can also be used as a normalization function. A normalization function is a function that when given a term it returns the normal form of the input term. Andreas Abel and Dulma Rodriguez used hereditary substitution in this manner in [4]. They used it to normalize types

in a type theory with type-level computation much like system F^{ω}. In that paper the authors were investigating subtyping in the presence of type level computation. They found that hereditary substitution could be used to normalize types and then do subtyping. This allowed them to only define subtyping on normal types. Similar to their work Chantal Keller and Thorsten Altenkirch use hereditary substitution to define a normalizer for STLC and formalize their work in Agda [73]. As we mentioned above the drawback of hereditary substitution is that it does not scale to richer type theories. Thus, to prove consistency of more advanced type theories we need another technique that does scale.

6.3 Tait-Girard Reducibility

The Tait-Girard reducibility method is a technique for showing weak and strong normalization of type theories. It originated from the work of William Tait. He showed strong normalization of system Tusing an interpretation of types based on set theory with comprehension. He called this interpretation saturated sets. Later, John Yves Girard, against popular belief[1], extended Tait's method to be able to prove system F strongly normalizing. He called his method reducibility candidates. The reducibility candidates method is based on second order set theory with comprehension. It turns out that the genius work of Girard extends to a large class of type theories. The standard reference on all the topics of this section is Girard's wonderful book [60]. We will summarize how to show strong normalization of STLC using Tait's

[1]It has been said that while Girard was working on extending Tait's method other researchers, notably Stephen Kleene, criticized him for trying. They thought it was an impossible endeavor.

method and then show how this is extended to system F. We leave all proofs to the interested reader, but they can be found in [60].

The first step in proving strong normalization of STLC using Tait's method is to define the interpretation of types. An interpretation of a type T is a set of closed terms closed under eliminations. We denote the set of strongly normalizing terms as SN. Defining the interpretation of types depends on an extension by Girard which constrains the number of lemmas down to a minimal amount. A term is *neutral* if it is of the form $t_1\, t_2$ for some terms t_1 and t_2. Neutrality characterizes the terms which cannot be easily seen to be normalizing.

Definition 6.3.0.1.

The interpretation of types are defined as follows:

$$\begin{array}{lcl} [\![X]\!] & = & \{t \mid t \in \mathsf{SN}\} \\ [\![T_1 \to T_2]\!] & = & \{t \mid \forall t' \in [\![T_1]\!].t\, t' \in [\![T_2]\!]\} \end{array}$$

The interpretation of types are known as reducibility sets. We say a term is reducible if it is a member of one of these sets. Next we have some constraints the previous definition must satisfy. Girard called these the **CR 1-4** properties. Their proofs can be found in [60].

Lemma 6.3.0.2. *If $t \in [\![T]\!]$, then $t \in \mathsf{SN}$.*

Lemma 6.3.0.3. *If $t \in [\![T]\!]$ and $t \rightsquigarrow t'$ then $t' \in [\![T]\!]$.*

Lemma 6.3.0.4. *If t is neutral, $t' \in [\![T]\!]$ and $t \rightsquigarrow t'$ then $t \in [\![T]\!]$.*

Lemma 6.3.0.5. *If t is neutral and normal then $t \in [\![T]\!]$.*

The proof that $[\![T]\!]$ defined in Def. 6.3.0.10 satisfies these four properties can

be done by induction on the structure of T. We need two additional lemmas to show that all typeable terms of STLC are reducible.

Lemma 6.3.0.6. *If for all $t_2 \in [\![T_1]\!]$ and $[t_2/x]t_1 \in [\![T_2]\!]$ then $\lambda x : T_1.t_1 \in [\![T_1 \to T_2]\!]$.*

The proof is by case analysis on the possible reductions of $(\lambda x : T_1.t_1)\, t_2$. To prove that all terms are reducible we must first define sets of well-formed substitutions. We denote the empty substitution as $\emptyset$.

Definition 6.3.0.7.

Well-formed substitutions are defined as follows:

$$\frac{}{\vdash \emptyset} \qquad \frac{t \in [\![T]\!] \qquad \vdash \sigma}{\vdash \sigma \cup (x, t)}$$

We say a substitution is well-formed with respect to a context if the substitution is well-formed, the domain of the substitution consists of all the variables of the context, and the range of the substitution consists of terms with the same type as the variable they are replacing. We denote this by $\Gamma \vdash \sigma$. Thus, if $\Gamma \vdash \sigma$ then the domain of σ is the domain of Γ and the range of σ are reducible typeable terms with the same type as the variable they are replacing. We define the interpretation of a context as $\Gamma = \{\sigma \,|\, \Gamma \vdash \sigma\}$. We now have everything we need to show that all typeable terms are reducible, hence, strongly normalizing.

Theorem 6.3.0.8. *If $\sigma \in [\![\Gamma]\!]$ and $\Gamma \vdash t : T$ then $\sigma\, t \in [\![T]\!]$.*

Corollary 6.3.0.9. *If $\cdot \vdash t : T$ then $t \in \mathsf{SN}$.*

Girard extended this method into a more powerful one called reducibility candidates to be able to prove strong normalization for system F. We first extend the

definition of a neutral term to include $t[T]$. The definition of the interpretation of types are defined next.

Definition 6.3.0.10.

The interpretation of types are defined as follows:

$$\begin{array}{lcl} [\![X]\!]_\rho & = & \rho(X) \\ [\![T_1 \to T_2]\!]_\rho & = & \{t \,|\, \forall t' \in [\![T_1]\!]_\rho . t\ t' \in [\![T_2]\!]_\rho\} \\ [\![\forall X.T]\!]_\rho & = & \{t \,|\, \forall T'.t[T'] \in [\![T]\!]_{\rho\{X \mapsto [\![T']\!]\}}\} \end{array}$$

The sets defined in the previous definition are called parameterized reducibility sets. Recall that system F has type variables so when we interpret types we must interpret type variables. The naive extension would just replace type variables with types, but Girard quickly realized this would not work, because the final case of the above definition of the interpretation of types would then be $[\![\forall X.T]\!] = \{t \,|\, \forall T'.t[T'] \in [\![[T'/X]\,T]\!]\}$ and we can no longer consider this a well-defined definition, because it is not structurally recursive over the type. So instead we replace type variables with reducibility candidates. A reducibility candidate is just a reducibility set which satisfies the four **CR** properties above. We denote the set of all reducibility candidates as Red. Then in the definition of $[\![\forall X.T]\!]_\rho$ we quantify over all reducibility sets. This is where set comprehension is coming in; also note that this is a second order quantification. We need two forms of substitutions: substitutions mapping term variables to terms and type variables to types, but also substitutions mapping type variables to reducibility candidates called reducibility candidate substitutions. We denote the former by σ and the latter as ρ. The following two definitions derive when both of these types of substitutions are well-formed.

Definition 6.3.0.11.

Well-formed substitutions:

$$\frac{}{\vdash \emptyset} \qquad \frac{t \in [\![T]\!]_\sigma \qquad \vdash \sigma}{\vdash \sigma \cup \{(x, t)\}} \qquad \frac{\vdash \sigma}{\vdash \sigma \cup \{(X, T)\}}$$

Definition 6.3.0.12.

Well-formed reducibility candidate substitutions:

$$\frac{}{\vdash \emptyset} \qquad \frac{[\![\rho'\ T]\!] \in \mathsf{Red} \qquad \vdash \rho}{\vdash \rho \cup \{(X, [\![T]\!]_{\rho'})\}}$$

The following lemmas depend on the following definition of well-formed substitutions with respect to reducibility candidate substitutions.

Definition 6.3.0.13.

Well-formed substitution with respect to a well-formed reducibility candidate substitution:

$$\frac{}{\emptyset \vdash \emptyset} \qquad \frac{[\![\rho'\ T]\!] \in \mathsf{Red} \qquad \rho \vdash \sigma}{\rho \cup \{(X, [\![T]\!]_{\rho'})\} \vdash \sigma \cup \{(X, T)\}}$$

We are now set to prove some new lemmas. The following proofs depend on the lemmas we have proven above about STLC. We do not repeat them here. We say a parameterized reducibility set $[\![T]\!]_\rho$ is a reducibility candidate of type $\sigma\ T$ if and only if $[\![\sigma\ T]\!] \in \mathsf{Red}$ and $\rho \vdash \sigma$. First, we prove that parameterized reducibility sets are members of Red.

Lemma 6.3.0.14. *$[\![T]\!]_\rho$ is a reducibility candidate of type $\sigma\ T$ where $\rho \vdash \sigma$.*

Our second result shows that substitution can be pushed into the parameter of the reducibility set. Set comprehension is hiding in the statement of the lemma. In order to push the substitution down into the parameter we must first know that $[\![T]\!]_\rho$ really

is a set.

Lemma 6.3.0.15. *If* $\vdash \rho$ *then* $[\![[T/X]\,T']\!]_\rho = [\![T']\!]_{\rho\{X \mapsto [\![T]\!]_\rho\}}$.

These final two lemmas are straightforward. They are similar to the lemmas above for λ-abstraction and application.

Lemma 6.3.0.16. *If for every type* T *and reducibility candidate* R, $[T/X]t \in [\![T']\!]_{\rho\{X \mapsto R\}}$, *then* $\Lambda X.t \in [\![\forall X.T']\!]_\rho$.

Lemma 6.3.0.17. *If* $t \in [\![\forall X.T]\!]_\rho$, *then* $t[T'] \in [\![[T'/X]\,T]\!]_\rho$ *for every type* T'.

Finally, we can prove type soundness and obtain strong normalization.

Theorem 6.3.0.18. *If* $\Gamma \vdash t : T$, $\Gamma \vdash \sigma$ *and* $\Gamma \vdash \rho$ *then,* $\sigma t \in [\![T]\!]_\rho$.

Corollary 6.3.0.19. *If* $\cdot \vdash t : T$ *then* $t \in SN$.

The proof of the corollary follows from Theorem 6.3.0.18 by using a certain ρ. The ρ one must use is the one were every type variable is mapped to a subset of SN. This extension is a giant leap forward and is the foundation of what we now call logical relations. The interpretation of types we defined above are actually unary predicates defined by recursion on the type. The form we defined them in here are in logical relation form rather than the syntax of Girard in [60].

6.4 Logical Relations

Logical relations are straightforward extensions of reducibility sets. They were first proposed as generalizations of Tait's reducibility candidates by Gordon Plotkin in 1973 [109]. They were further generalized by Richard Statman in 1985 [127]. Logical relations can be thought of as predicates defined by recursion on their parameter. Usually, this is the type. They are always closed under eliminations and a usually

defined in the same way as we defined the interpretation of types for STLC and system F. They are called "logical", because of the fact that they are closed under eliminations. This allows us to prove properties that are not preserved by elimination. Termination is an example of this. Logical relations are not required to be unary. However, we have not seen any applications of n-arity logical relations where $n > 2$. An example where Binary logical relations have been used is the study program equivalence. Logical relations have been used in a wide range of applications in fact, from consistency proofs all the way to encryption.

Andrew Pitts used logical relations to show when two inhabitants of the disjoint union type are equivalent in [107]. Karl Crary gives a nice introduction to logical relations in [105] where he shows how to solve the equivalence problem for terms. The equivalence problem is being able to decide operational equivalence of terms. Eijiro Sumii and Benjamin Pierce use logical relations to prove properties of a type theory used for encryption in [132]. They prove behavior equivalences between terms of this calculus which depend on encryption. One such property is to show that a particular piece of data a program is keeping secret from attacks is never recovered by some attacker. This property can be formalized as a behavior equivalence. They then use logical relations to prove such equivalences.

6.4.1 Step-Indexed Logical Relations

There is one last extension to logical relations. So far we have introduced the reducibility method and its extension to reducibility candidates. We briefly summarized the fact that reducibility candidates gives rise to logical relations. However, we

have used logical relations only to prove properties about terminating theories. Can logical relations be used to reason about non-terminating type theories? It turns out that we can, but it requires yet another extension of logical relations.

Adding the ability to define general recursive types to a type theory results in the theory being non-terminating. That is, one can define a diverging term. In the field of programming languages recursive types are a very powerful feature. One property one may wish to prove about a type theory with recursive types is contextual equivalence of terms. Logical relations are usually used to prove such a property, but they turn out not to work in the presence of recursive types. This was an outstanding open problem until Andrew Appel and David McAllester were able to find an extension of logical relations called step-indexed logical relations.

Step-indexed models were first introduced by Andrew Appel and David McAllester in [13] as the semantics of recursive types. At the time it was not known how to model recursive types without using complex machinery like domain theory. Later, Amal Ahmed extended the earlier work by Appel and McAllester and was able to prove contextual equivalence of terms of system F with recursive types [7]. Since this earlier work a number of applications of step-indexed logical relations have been conducted, e.g. [5, 8, 94, 138]. One drawback of using step-indexed logical relations is that the proofs usually involve tedious computations of step indices. In [47] Derek Dreyer et al. introduce a way of encapsulating the step index in such away that the index no longer needs to be present in the model.

The major application of step-indexed logical relations have so far been meta-

theoretic results such as type safety, contextual equivalence or other safety results. It was not until 2012 when they were actually used to prove normalization of typed λ-calculi. Chris Casinghino et al. in [30] developed a type theory with general recursion and recursive types with a collapsed syntax. This is a very interesting development, because they use modal operators to separate a logical world (only terminating programs) from a programmatic world within the same language. They then prove normalization of the logical world using step-indexed logical relations.

We briefly describe what step-indexed logical relations are through an example. We extend the CBV STLC with iso-recursive types and then try to prove type safety of this extension using logical relations. We will run into trouble and will be forced to use step-indexed logical relations instead.

Type safety is a property of a programming language which guarantees that computation never gets stuck. We can always either complete the computation (hit a value) or continue computing (take another step). Type safety is defined by the following theorem:

Theorem 6.4.1.1. $\cdot \vdash t : T$ *and* $t \rightsquigarrow^* t'$ *then* $val(t')$ *or* $\exists t''.t' \rightsquigarrow t''$.

Where val(t) is a predicate on terms which is true iff t is a syntactic value. Recall values are either a variable or a λ-abstraction. Usually, type safety is shown by proving type preservation and progress theorems. These theorems however are corollaries of our type-safety theorem. The reason it is usually done this way is because it is thought to be easier than giving a direct proof. We will see here that it is actually

pretty simple to give a direct proof using the logical relations[2]. We do not show any proofs here, but they can all be found in the extended version of [7] which can be accessed through Ahmed's web page[3].

We begin with the proof of type safety of the call-by-value STLC, and then extend this to include iso-recursive types. To get the proof of the type safety theorem to go through we need to first define the logical relations.

Definition 6.4.1.2.

We define logical relations for values and then we extend this definition to terms.

Logical relations for values:

$$\mathcal{V}[\![X]\!] = \{v \mid \cdot \vdash v : X\}$$

$$\mathcal{V}[\![T_1 \to T_2]\!] = \{\lambda x : T.t \mid \cdot \vdash \lambda x : T_1 : T_1 \to T_2 \wedge \forall v.v \in \mathcal{V}[\![T_1]\!] \implies [v/x]t \in \mathcal{E}[\![T_2]\!]\}$$

Logical relations extended to terms:

$$\mathcal{E}[\![T]\!] = \{t \mid \cdot \vdash t : T \wedge \exists v.t \rightsquigarrow^* v \wedge v \in \mathcal{V}[\![T]\!]\}$$

Well-formed substitutions:

$$\mathcal{G}[\![\Gamma, x : T]\!] = \{\gamma[x \mapsto v] \mid \gamma \in \mathcal{G}[\![\Gamma]\!] \wedge v \in \mathcal{V}[\![T]\!]\}$$

To express when a particular open term t has meaning with respect to our chosen semantics we define a new judgment which has the form $\Gamma \models e : T$. This judgment can be read as t models type T in context Γ.

$$\Gamma \models t : T =^{df} \forall \gamma \in \mathcal{G}[\![\Gamma]\!] \implies \gamma(t) \in \mathcal{E}[\![T]\!].$$

We now turn to the fundamental property of logical relations. We state this property

[2]This section is based off of a lecture given by Amal Ahmed at the 2011 Oregon Programming Language Summer School.

[3]http://www.ccs.neu.edu/home/amal/papers/lr-recquant-techrpt.pdf

as follows:

Lemma 6.4.1.3. *If* $\Gamma \vdash t : T$ *then* $\Gamma \models e : T$.

Proof. By induction on the structure of the assumed typing derivation. □

The fundamental property then allows us to prove our main theorem. To make expressing this result cleaner we define the following predicate:

$$\text{safe}(t) =^{def} \forall t'.t \rightsquigarrow^* t' \implies (\text{val}(t') \vee \exists t''.t' \rightsquigarrow t'').$$

Theorem 6.4.1.4. *If* $\Gamma \vdash t : T$ *then safe*(t).

Proof. By induction on the assumed typing derivation. □

To summarize we have shown how to prove type safety using logical relations of CBV STLC. Next we extend CBV STLC with iso-recursive types. To the types we add $\mu\alpha.T$ and type variables α. Do not confuse this operator with that of the operator of the $\lambda\mu$-calculus. It is unfortunate, but this operator is used to capture many different notions throughout the literature. The terms are extended to include fold t and unfold t, and values are extended to include fold v. Finally, we add fold E and unfold E to the syntax for evaluation contexts. To deal with free type variables we either can add them to contexts Γ or add a new context specifically for keeping track of type variables. We will do the latter and add the following to our syntax:

$$\Delta \;:=\; \cdot \;\mid\; \Delta, \alpha$$

We need one additional rule to complete the operational semantics which is unfold(fold v) $\rightsquigarrow v$. We complete the extension by adding two new type checking rules. They are defined as follows:

$$\frac{\Gamma, \Delta \vdash t : [\mu\alpha.T/\alpha]T}{\Gamma, \Delta \vdash \text{fold } t : \mu\alpha.T}\ \text{fold} \qquad \frac{\Gamma, \Delta \vdash t : \mu\alpha.T}{\Gamma, \Delta \vdash \text{unfold } t : [\mu\alpha.T/\alpha]T}\ \text{unfold}$$

Let's try and apply the same techniques we used in the previous section to prove type safety of our extended language.

We first have to extend the definition of the logical relations to deal with recursive types.

Definition 6.4.1.5.

We define logical relations for values and then we extend this definition to expressions (terms t).

Logical relations for values:

$\mathcal{V}[\![\alpha]\!]_\rho = \rho(\alpha)$

$\mathcal{V}[\![X]\!]_\rho = \{v \mid \cdot \vdash v : X\}$

$\mathcal{V}[\![T_1 \to T_2]\!]_\rho =$
$\quad \{\lambda x : T.t \mid (\cdot \vdash \lambda x : T_1 : T_1 \to T_2 \wedge \forall v.v \in \mathcal{V}[\![T_1]\!]_\rho) \implies [v/x]t \in \mathcal{E}[\![T_2]\!]_\rho\}$

$\mathcal{V}[\![\mu\alpha.T]\!]_\rho = \{\mathit{fold}\ v \mid \forall v.\mathit{unfold}\ (\mathit{fold}\ v) \in \mathcal{V}[\![[\mu\alpha.T/\alpha]T]\!]_\rho.$

Logical relations extended to expressions:

$\mathcal{E}[\![T]\!]_\rho \ = \ \{t \mid \cdot \vdash t : T \wedge \exists v.t \rightsquigarrow^* v \wedge v \in \mathcal{V}[\![T]\!]_\rho\}$

Well-formed substitutions:

$\mathcal{G}[\![\Gamma, x : T]\!]_\rho \ = \ \{\gamma[x \mapsto v] \mid \gamma \in \mathcal{G}[\![\Gamma]\!]_\rho \wedge v \in \mathcal{V}[\![T]\!]_\rho\}$

This definition is slightly different from the previous. Since we have type variables we need to use Girard's trick to handle reducibility-candidates substitutions. Then we added the case for recursive types. Here we took the usual idea of using the elimination form for μ-types. Now is this definition well-founded? Recall that

one of the main ideas pertaining to logical relations is that the definitions are done by induction on the structure of the type. Now it is easy to see that the definition above is clearly well-founded in all the previous cases, but it would seem not to be for the case of the μ-type. The type $[\mu\alpha.T/\alpha]T$ increased in size rather than decreasing. So how can we fix this? First we notice that by the definition of our operational semantics unfold (fold v)) $\rightsquigarrow v$, so we can replace unfold (fold v) with just v in the definition. So that simplifies matters a bit, although this does not help us with respect to well-foundedness. One more attempt would be to take the substitution and push it into ρ. Let's see what happens when we try this. Take the following for our new definition of the logical relation for μ-types:

$$\mathcal{V}[\![\mu\alpha.T]\!]_\rho = \{\text{fold } v \mid \forall v.v \in \mathcal{V}[\![T]\!]_{\rho[\alpha \mapsto \mathcal{V}[\![\mu\alpha.T]\!]_\rho]}\}.$$

Now we can really see the problem. This new definition is defined in terms of itself! This is a result of the recursive type being recursive. So how can we fix this? To define a well-founded definition for recursive types we need something a little more powerful then just ordinary logical relations. This is where step indices come to the rescue.

We need to not only consider the structure of the type as the measure of well-foundedness for our definition for recursive types, but also the operational behavior defined by our operational semantics. Let's just dive right in and define a new definition of our logical relations. All of our logical relations are interpretations.

Definition 6.4.1.6.

We define an interpretation as $\mathcal{I} \in \mathcal{P}(\mathbb{N} \times Term)$.

We say an interpretation is well-formed if its elements are all atoms (members of the set Atom). An atom is a set of tuples of natural numbers and closed terms. Additionally, we require an interpretation to be an element of the set Type. Atom and Type are defined by the following definition.

Definition 6.4.1.7.

$$\begin{aligned} \mathit{Atom} &= \{(k,t) \mid k \in \mathbb{N} \wedge t \in \mathit{Closed}^{\mathit{Term}}\} \\ \mathit{Atom}^{\mathit{value}} &= \mathit{Atom\ restricted\ to\ values} \\ \mathit{Type} &= \{\mathcal{I} \subseteq \mathit{Atom}^{\mathit{value}} \mid \forall (k,v) \in \mathcal{I}. \forall j \leq k. (j,v) \in \mathcal{I}\} \end{aligned}$$

One of the key concepts of step-indexed logical relations is the notion of approximation. Hence, we need to be able to take approximations of interpretations. This will be more clear below.

Definition 6.4.1.8.

The n-approximation function on interpretations is defined as follows:

$$\lfloor \mathcal{I} \rfloor_n = \{(k,v) \in \mathcal{I} \mid k < n\}$$

We are now in a position to start defining the interpretations of types (logical relations).

Definition 6.4.1.9.

Logical relations for values:

$$\mathcal{V}[\![\alpha]\!]_\rho = \rho(\alpha)$$

$$\mathcal{V}[\![X]\!]_\rho = \{(k,v) \in \mathit{Atom}^{\mathit{value}} \mid \cdot \vdash v : X\}$$

$$\begin{aligned} \mathcal{V}[\![T_1 \to T_2]\!]_\rho = \{(k, \lambda x : T.t) \in \mathit{Atom}^{\mathit{value}} \mid \cdot \vdash \lambda x : T_1.t : T_1 \to T_2 \wedge \\ \forall j \leq k. \forall v. (j,v) \in \mathcal{V}[\![T_1]\!]_\rho \implies (j, [v/x]t) \in \mathcal{T}[\![T_2]\!]_\rho\} \end{aligned}$$

$$\bar{\mathcal{V}}_n[\![\mu\alpha.T]\!]_\rho = \{(k, \mathit{fold}\ v) \in \mathit{Atom}^{\mathit{value}} \mid k < n \wedge \forall j < k. (j,v) \in \mathcal{V}[\![T]\!]_{\rho[\alpha \mapsto \bar{\mathcal{V}}_k[\![\mu\alpha.T]\!]_\rho]}\}$$

$$\mathcal{V}[\![\mu\alpha.T]\!]_\rho = \bigcup_{n \geq 0} \bar{\mathcal{V}}_n[\![\mu\alpha.T]\!]_\rho$$

The next definition extends the previous to terms.

Definition 6.4.1.10.

Logical relations extended to terms:

$$\mathcal{T}[\![T]\!] = \{(k,t) \in Atom \mid \cdot \vdash t : T \wedge \\ \forall j \leq k. \forall t'. t \rightsquigarrow^{j} t' \wedge \cdot \vdash t' : T \wedge (irred(t') \implies (j,t') \in \mathcal{V}[\![T]\!]_{\rho})\}$$

We will need two types of substitutions one for term variables and one for type variables. The following definitions tell us when they are well-formed.

Definition 6.4.1.11.

Well-formed term-variable substitutions:

$$\mathcal{G}[\![\cdot]\!] = \{(k,\emptyset)\}$$

$$\mathcal{G}[\![\Gamma, x:T]\!] = \{(k, \gamma[x \mapsto v]) \mid k \in \mathbb{N} \wedge (k,\gamma) \in \mathcal{G}[\![\Gamma]\!] \wedge (k,v) \in \mathcal{V}[\![T]\!]_{\emptyset}\}$$

Definition 6.4.1.12.

Well-formed type-variable contexts:

$$\mathcal{D}[\![\cdot]\!] = \{\emptyset\}$$

$$\mathcal{D}[\![\Delta, \alpha]\!] = \{\rho[\alpha \mapsto \mathcal{I}] \mid \rho \in \mathcal{D}[\![\Delta]\!] \wedge \mathcal{I} \in Type\}$$

Finally, we define when term t is in the interpretation of type T as follows:

$$\Gamma \models t : T =^{def} \forall k \geq 0. \forall \gamma. (k,\gamma) \in \mathcal{G}[\![\Gamma]\!] \implies (k, \gamma(t)) \in \mathcal{T}[\![T]\!]_{\emptyset}.$$

Let's take a step back and consider our new definition and use it to define exactly what a we mean by step-indexed logical relations. Instead of our logical relations being sets of closed terms they are now tuples of natural numbers and closed terms. This natural number is called the step index. This is the number of steps necessary for the closed term to reach a value. By steps we mean the number of rule applications of our operational semantics. For example, $t\ t' \rightsquigarrow^{1} [t'/x]t =^{1} t''$, where t'' is the actual result of the substitution. Thus, applications actually consumes two steps!

Now all of our definitions of the logical relations are well-defined using an ordering consisting of only the type except for the definition of the logical relation for μ-types. This is the case as we saw earlier where we need the step index. The main point of this definition is that we take larger and larger approximations of the runtime behavior of the elements of the μ-type logical relation. So we define the logical relation for μ-types in terms of an auxiliary logical relation, where the number of steps the members of the relation are allowed to take is bound by some natural number n. This corresponds to $\bar{\mathcal{V}}_n[\![\mu\alpha.T]\!]_\rho$. Then we define the logical relation for μ-types as the union of all the approximations, i.e. $\mathcal{V}[\![\mu\alpha.T]\!]_\rho$.

We can now conclude type safety for STLC with recursive types. We will need the following two lemmas in the proof of the fundamental property of the logical relation. We write $\Delta \vdash T$ to mean that all the type variables in Δ are free in T. The first lemma is known as downward closure of the step-index logical relation. The second is simple substitution commuting just as we saw for system F above.

Lemma 6.4.1.13. *If* $\Delta \vdash T$, $\rho \in \mathcal{D}[\![\Delta]\!]$, $(k, v) \in \mathcal{V}[\![T]\!]_\rho$, *and* $j \leq k$ *then* $(j, v) \in \mathcal{V}[\![T]\!]_\rho$.

Lemma 6.4.1.14. *If* $\Delta, \alpha \vdash T$, $\rho \in \mathcal{D}[\![\Delta]\!]$ *and* $\mathcal{I} = \lfloor \mathcal{V}[\![\mu\alpha.T]\!]_\rho \rfloor_k$ *then* $\lfloor \mathcal{V}[\![[\mu\alpha.T/\alpha]T]\!]_\rho \rfloor_n = \lfloor \mathcal{V}[\![T]\!]_{\rho[\alpha \mapsto \mathcal{I}]} \rfloor_n$.

Finally, we conclude with the fundamental property of logical relations.

Theorem 6.4.1.15. *If* $\Gamma \vdash t : T$ *then* $\Gamma \models t : T$.

From the fundamental property of the logical relations we can prove type safety in a similar way as for standard STLC above.

PART B

DESIGN

CHAPTER 7

FREEDOM OF SPEECH

The design of any programming language must facilitate reasoning about the programming language itself. This facilitation comes in the form of a rigorous definition which makes mathematically precise all of the structure of the programming language. This provides a means for researchers and implementors to fully understand the limits of the language. Contrary to what programming language designers have done in the past this rigorous definition does not only include the syntax, but also includes mathematical definitions of the type system as well as an interpreter. This is a significant benefit of basing programming languages on type theories. As we have seen a type theory must be rigorously defined.

Considering the motivation we gave in the introduction, a mathematically rigorous design provides the necessary structure to allow programmers to mathematically reason about the programs they write in the programming language. In addition, this makes it possible for designers to reason about the correctness of the language itself, and thus allow them to make strong guarantees of the correctness of the language to their programmers. To prevent major bugs programming languages must be rigorously defined. If we do not know what the programming language allows, then we cannot be sure what is safe.

Recall from Chapter 2 that the computational trinity states that type theories are simultaneously a logic as well as a programming language, and this double

perspective provides the means of verifying properties of the programs we write in the theory, but this feature comes with a strong invariant, every program must terminate. Now suppose we are working on a large development in a programming language based on type theory like Martin-Löf's Type Theory or CoC which has come to a difficult problem that can be solved by the definition of some terminating function. Furthermore, suppose we have found the solution in some research paper, but the termination proof is very complex and uses an advanced semantic proof technique, and lastly suppose that this termination proof is not formalized in a type theory, but is an informal proof that we trust. Since we are conducting our development in a terminating type theory there is only one course of action. We have to formalize this complex termination argument. This can be devastating to the development, because doing such a proof may take months, even years to complete! Another hypothetical is supposing we do not care if a program terminates. There are correctness properties that hold regardless of termination. For example, consider associativity of list append – using a pseudo syntax:

$$\forall(A\,B\,C : \mathsf{List}\,\mathsf{Nat}).\mathsf{append}\,A\,(\mathsf{append}\,B\,C) = \mathsf{append}\,(\mathsf{append}\,A\,B)\,C$$

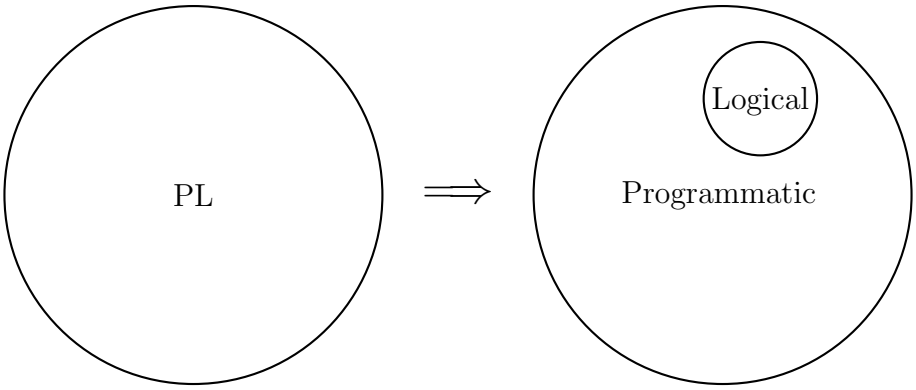
This equation holds regardless of termination of A, B, or C, because if anyone of them happen to diverge, then both sides of the equation diverge.

So the natural question is can we design a programming language based on type theory that allows for non-terminating programs, but also be able to verify properties of the programs we write in the language? It turns out that we can, and in the next two chapters we introduce the designs of two new dependently-typed

functional programming languages called Freedom of Speech and Separation of Proof from Program that amount to solutions to this question.

Directly supporting non-termination is one solution, but there is another possibility. Non-terminating functions can be simulated using the notion of coinduction which is dual to induction. Coinduction provides a means to define and observe infinite streams of data. Furthermore, coinductive types push the notion of non-termination into the types. There are a large number of applications of coinduction, and even more when we allow the mixture of induction and coinduction, for example the definition of an infinite stream of trees requires their mixture. However, currently there is no known type theory that supports both induction and coinduction, and supports their (unrestricted) mixture while maintaining type safety. In the third chapter of Design we introduce a new logic and corresponding type theory called Dualized Intuitionistic Logic and Dualized Type Theory respectively that is based on a logic rich in duality that shows promise of being a logical framework for induction and coinduction that is type safe and supports their mixture.

The requirement that every program must terminate can be relaxed by first designing a very powerful programming language (PL), and then carving out two fragments of programs. The first consists of all the terminating programs called the logical fragment, and the second consists of all programs, and this is called the programmatic fragment. That is, we have a picture that looks something like:

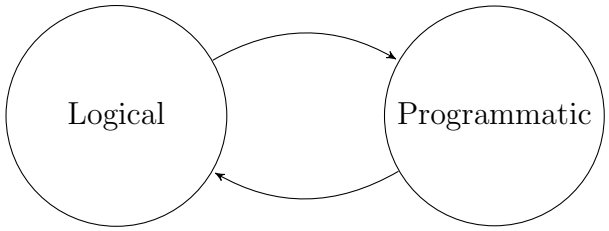


The logical fragment will then be considered the "logical framework" of the language. This is where the programs are proofs and the types are propositions; that is, the logical fragment makes use of the computational trinity (Chapter 2). Now the programmatic fragment is where all the usual programming will take place. One can then use the logical fragment to verify properties of the programs written in the programmatic fragment. Alternately, one can understand the fragments as worlds in the sense of possible world semantics of modal logics. Below we will see how these two worlds are connected. In fact, Chris Casinghino et al. have constructed a programming language very much like freedom of speech, but they add modal operators to the type system to distinguish the logical and programmatic fragments in [30, 31]. Then they characterize which programs can be moved from one fragment to another, they call the types of these programs mobile types.

Once the logical fragment has been identified three additional features will need to be added. The first feature is that types in the logical fragment will need to be able to depend on programs from the programmatic fragment, but this feature has to be designed so as to prevent these programs from being applied to any arguments

or this would prevent the logical fragment from being logically consistent. For an overview of logical consistency and how it can be proven see Chapter 6. We call this feature freedom of speech, because it intuitively states that logical types and programs can talk about potentially non-terminating programs, but they are never allowed to actually run them.

The second and third features are usability features. The logical fragment is purely specificational. Its primary use is for the verification of programs written in the programmatic fragment. Furthermore, carrying around non-computationally interesting proofs is expensive. Thus, it is important to allow some specificational data to be stripped away at compile time. In addition, it is important that the programmer be the one to decide which data is removed. Now logical programs are proofs, but they are also terminating programs, and thus can be considered both logical programs and programmatic programs. The third and final feature is the ability to write programs in the logical fragment and then move them into the programmatic fragment. This facilitates code reuse and provides a means to write verified terminating programs. We mentioned that the logical and programmatic fragments can be consider worlds. Now the freedom of speech property relates the programmatic fragment to the logical fragment by allowing the objects of the logical fragment to express properties about the objects of the programmatic fragment. Additionally, the feature allowing the programs of the logical fragment to be moved into the programmatic fragment relates the logical fragment to the programmatic fragment. Thus, we have a situation best captured by the following picture:



*

In this chapter we introduce the design of a programming language that contains all of these features, and some additional ones. It is called Freedom of Speech, because it is the first core dependently-typed functional programming language with the freedom of speech property.

7.1 Syntax and Reduction Relation

We begin by first defining the syntax and reduction relation, and then move on to the type system. The features discussed in the introduction to this chapter will be made explicit when we introduce the type system, but we will see hints of them in the syntax.

The syntax and the CBV reduction relation is defined in Figure 34. The syntax is collapsed similarly to the Calculus of Constructions – see Chapter 4.2 for more about the Calculus of Constructions. So we distinguish between types and terms (programs) judgmentally. Expressions and the typing judgment will depend on two annotations. The first is called the consistency classifier and is denoted θ which can be one of L or $\mathcal{C}$ where an expression tagged with the former is interpreted to mean "belonging to the logical fragment" and the latter to mean "belonging to the programmatic fragment." The second annotation is denoted ε which is called the stage annotation and can be either $+$ or $-$ where the former means "run time" and

Syntax:

$$
\begin{array}{rrl}
\text{(Classifiers)} & \theta & ::= L \mid P \\
\text{(Stages)} & \varepsilon & ::= + \mid - \\
\text{(Expressions)} & e, t & ::= \mathsf{Type} \mid \mathbb{N} \mid x \mid (x \ :^{\theta} \ e_1)^{\varepsilon} \to e_2 \mid e_1 = e_2 \mid \mathsf{S} \mid \mathsf{Z} \mid \\
 & & \quad \lambda x \,.\, e \mid \mathsf{rec}\, f\ x\ e \mid \mathsf{rec}^{-}\ f\ \ e \mid e_1\ e_2 \mid \mathsf{join} \mid \mathsf{injdom} \mid \mathsf{injran} \mid \\
 & & \quad \mathsf{contra} \mid \mathsf{abort} \\
\text{(Values)} & v & ::= x \mid \mathsf{Type} \mid \mathbb{N} \mid (x \ :^{\theta} \ v_1)^{\varepsilon} \to v_2 \mid e_1 = e_2 \mid \lambda x \,.\, e \mid \\
 & & \quad \mathsf{join} \mid \mathsf{injdom} \mid \mathsf{injran} \mid \mathsf{rec}\, f\ x\ v \mid \mathsf{rec}^{-}\ f\ \ v \\
\text{(Evaluation Contexts)} & \mathcal{C} & ::= \square \mid (x \ :^{\theta} \ \mathcal{C})^{\varepsilon} \to e_2 \mid (x \ :^{\theta} \ e_1)^{\varepsilon} \to \mathcal{C} \mid \mathsf{rec}\, f\ x\, \mathcal{C} \mid \\
 & & \quad \mathsf{rec}^{-}\ f\ \mathcal{C} \mid v\, \mathcal{C} \mid \mathcal{C}\ e \\
\text{(Typing Contexts)} & \Gamma & ::= \cdot \mid x :^{\theta} e \mid \Gamma_1, \Gamma_2
\end{array}
$$

CBV reduction:

$$\frac{}{(\lambda x \,.\, e)\ v \rightsquigarrow_{CBV} [v/x]e}\ \text{Cbv\_App} \qquad \frac{}{(\mathsf{rec}\, f\ x\ v')\ v \rightsquigarrow_{CBV} [\mathsf{rec}\, f\ x\ v'/f][v/x]v'}\ \text{Cbv\_Rec}$$

$$\frac{e \rightsquigarrow_{CBV} e'}{\mathcal{C}[e] \rightsquigarrow \mathcal{C}[e']}\ \text{Red\_Ctxt} \qquad \frac{}{\mathcal{C}[\mathsf{abort}] \rightsquigarrow \mathsf{abort}}\ \text{Red\_Abort} \qquad \frac{e_1 \rightsquigarrow^{*} e \quad e_2 \rightsquigarrow^{*} e}{e_1 \downarrow e_2}\ \text{ComputeJoin}$$

Figure 34. Syntax and reduction rules for freedom of speech

the latter means "compile time." The consistency classifier essentially is how we carve out the logical fragment from the programmatic fragment, and the stage annotation implements the second feature above where all programs marked as compile time will be erased before run time. We now give a brief overview of the syntax of Freedom of Speech.

Expressions are denoted t,$t_1, \ldots, t_i$ and e,$e_1, \ldots, e_i$, and consist of Type which is the type of all types, $\mathbb{N}$ the type of natural numbers, variables denoted x,y,z, ..., dependent function types denoted $(x :^{\theta} e_1)^{\varepsilon} \to e_2$ – note that the programmer gets to decide whether an argument is "logical" or "programmatic" and "run time" or "compile time" – next we have equations denoted $e_1 = e_2$, the successor function and the natural number zero denoted S and Z respectively, λ-abstractions denoted $\lambda x \,.\, e$, two recursors $\mathsf{rec}\, f\ x\ e$ and $\mathsf{rec}^{-}\ f\ \ e$ where x is considered bound in e in the former, application denoted $e_1\ e_2$, three forms of proofs of equations denoted join, injdom, and injran, where the latter two are injectivity proofs and are discussed further below, and finally we have two forms of contradictions one logical and one programmatic denoted contra and abort respectively.

The CBV reduction relation is broken up into three different judgments. The first is denoted $e_1 \rightsquigarrow_{CBV} e_2$ and defines standard CBV β-reduction and consists of two rules CBV_App and CBV_Rec where values are defined in Figure 34. We will see that the latter rule can be used for both terminating and non-terminating recursion depending on which fragment the recursor is typed in. Note that there are no congruence rules in the definition of the first judgment. The second judgment

denoted $e_1 \rightsquigarrow e_2$ extends the first with congruence rules, and a rule for aborting a contradictory computation. This judgment consists of two rules Red_Ctxt and Red_Abort. These two are defined in terms of evaluation contexts which can be thought of as a means of specifying where with in an expression computation can take place. The syntax of evaluation contexts can be found in Figure 34. This defines a fragment of the syntax of expressions where exactly one well formed subexpression has been replaced with a hole. Holes are denoted $\Box$. Let's consider a few examples:

Example 7.1.0.1.

The following are all well-formed contexts:

$$((\lambda x\,.\,\mathsf{S}\,x)\,\Box),\ (\Box\,\mathsf{Z}),\ \textit{and}\ \mathsf{rec}\,f\,x\,(\mathsf{rec}\,g\,y\,\Box).$$

Now $(\lambda x.\mathsf{S}\,\Box)\,\mathsf{Z}$ *and* $\Box\,\Box$ *are not well-formed contexts.*

In addition to the syntax of evaluation contexts we also define an operation that takes an evaluation context and an expression and "plugs" the expression into the hole of the context.

Definition 7.1.0.2.

Plugging an expression e *into an evaluation context* $\mathcal{C}$ *is denoted* $\mathcal{C}[e]$ *and is defined by recursion on the form of* $\mathcal{C}$ *as follows:*

$$\begin{array}{rcl}
\Box[e] & = & e \\
((x\ :^{\theta}\ \mathcal{C})^{\varepsilon} \to e_2)[e] & = & (x\ :^{\theta}\ \mathcal{C}[e])^{\varepsilon} \to e_2 \\
((x\ :^{\theta}\ e_1)^{\varepsilon} \to \mathcal{C})[e] & = & (x\ :^{\theta}\ e_1)^{\varepsilon} \to (\mathcal{C}[e]) \\
(\mathsf{rec}\,f\,x\,\mathcal{C})[e] & = & \mathsf{rec}\,f\,x\,(\mathcal{C}[e]) \\
(\mathsf{rec}^{-}\ f\ \mathcal{C})[e] & = & \mathsf{rec}^{-}\ f\ (\mathcal{C}[e]) \\
(v\,\mathcal{C})[e] & = & v\,(\mathcal{C}[e]) \\
(\mathcal{C}\ e')[e] & = & (\mathcal{C}[e])\ e'
\end{array}$$

At this point one can easily see that the Red_Ctxt and Red_Abort are actually a family of congruence rules parametric in the evaluation context $\mathcal{C}$. Now the rule

Red_Ctxt simply extended the CBV β-reduction to evaluation contexts, while the rule Red_Abort says that if abort appears anywhere in an evaluation context, then the computation is aborted and concludes just abort. Let's consider an example using what we have introduced thus far before moving onto the type system.

Example 7.1.0.3.

First, we define the abstracted booleans and abstracted-boolean case similarly to how we defined booleans in system F (see Example 1.2.0.4 in Chapter 1):

$$\begin{aligned}
\mathsf{true} &\stackrel{\mathsf{def}}{\equiv} \lambda x\,.\,\lambda f_1\,.\,\lambda f_2\,.\,\lambda y\,.\,\lambda z\,.\,f_1\;y\\
\mathsf{false} &\stackrel{\mathsf{def}}{\equiv} \lambda x\,.\,\lambda f_1\,.\,\lambda f_2\,.\,\lambda y\,.\,\lambda z\,.\,f_2\;z\\
\mathsf{case} &\stackrel{\mathsf{def}}{\equiv} \lambda x\,.\,\lambda u\,.\,\lambda f_1\,.\,\lambda f_2\,.\,\lambda y\,.\,\lambda z\,.\,(u\;x\;f_1\;f_2\;y\;z)
\end{aligned}$$

We can define the usual Church-encoded booleans in terms of abstracted booleans as follows:

$$\begin{aligned}
\textit{CH-true} &\stackrel{\mathsf{def}}{\equiv} \lambda x.\lambda y.\lambda z.\mathsf{true}\;x\;(\lambda x\,.\,x)\;(\lambda x\,.\,x)\;y\;z\\
\textit{CH-false} &\stackrel{\mathsf{def}}{\equiv} \lambda x.\lambda y.\lambda z.\mathsf{false}\;x\;(\lambda x\,.\,x)\;(\lambda x\,.\,x)\;y\;z\\
\textit{CH-case} &\stackrel{\mathsf{def}}{\equiv} \lambda x.\lambda b.\lambda y.\lambda z.\mathsf{case}\;x\;b\;(\lambda x\,.\,x)\;(\lambda x\,.\,x)\;y\;z
\end{aligned}$$

Now suppose we have the division div *and the* isZero *functions. Then using these we can define a function that takes in two natural numbers as input, and then divides the first plus two by the second. However, there is a catch, the division function is undefined when the second argument is zero. So instead of leaving the exception handling to the division function let's craft our function to throw an error when the second argument is zero. In what way can we throw an error? This is exactly when we use the* abort *term. We can define our function as follows:*

$$\textit{div-plus-2} \stackrel{\mathsf{def}}{\equiv} \lambda x.\lambda y.\mathsf{case}\;\mathbb{N}\;(\mathsf{isZero}\;y)\;(\lambda x\,.\,\mathsf{abort})\;(\lambda x.(\mathsf{div}\;(\mathsf{S}\;(\mathsf{S}\;x))\;y))\;y\;x$$

In the definition of the function div-plus-2 *we had to bundle up the abort program inside a constant function to prevent our function from always triggering abort. This*

ensures us that our function will abort if and only if it calls the function $\lambda x\,.\,\mathsf{abort}$. *Similarly, we had to bundle up the division function inside a* λ*-abstraction so as to ensure that it does not throw an exception before our function can. Let's consider some computations using* **div-plus-2:**

$$
\begin{array}{ll}
 & (\lambda x.\lambda y.\mathsf{case}\,\mathbb{N}\,(\mathsf{isZero}\,y)\,(\lambda x\,.\,\mathsf{abort})\,(\lambda x.(\mathsf{div}\,(\mathsf{S}\,(\mathsf{S}\,x))\,y))\,y\,x)\,(\mathsf{S}\,(\mathsf{S}\,\mathsf{Z}))\,\mathsf{Z} \\
\rightsquigarrow^2 & \mathsf{case}\,\mathbb{N}\,(\mathsf{isZero}\,\mathsf{Z})\,(\lambda x\,.\,\mathsf{abort})\,(\lambda x.(\mathsf{div}\,(\mathsf{S}\,(\mathsf{S}\,x))\,\mathsf{Z}))\,\mathsf{Z}\,(\mathsf{S}\,(\mathsf{S}\,\mathsf{Z})) \\
\rightsquigarrow^7 & (\mathsf{isZero}\,\mathsf{Z})\,\mathbb{N}\,(\lambda x\,.\,\mathsf{abort})\,(\lambda x.(\mathsf{div}\,(\mathsf{S}\,(\mathsf{S}\,x))\,\mathsf{Z}))\,\mathsf{Z}\,(\mathsf{S}\,(\mathsf{S}\,\mathsf{Z})) \\
\rightsquigarrow^* & \mathsf{true}\,\mathbb{N}\,(\lambda x\,.\,\mathsf{abort})\,(\lambda x.(\mathsf{div}\,(\mathsf{S}\,(\mathsf{S}\,x))\,\mathsf{Z}))\,\mathsf{Z}\,(\mathsf{S}\,(\mathsf{S}\,\mathsf{Z})) \\
\rightsquigarrow^6 & (\lambda x\,.\,\mathsf{abort})\,\mathsf{Z} \\
\rightsquigarrow & \mathsf{abort}
\end{array}
$$

$$
\begin{array}{ll}
 & (\lambda x.\lambda y.\mathsf{case}\,\mathbb{N}\,(\mathsf{isZero}\,y)\,(\lambda x\,.\,\mathsf{abort})\,(\lambda x.(\mathsf{div}\,(\mathsf{S}\,(\mathsf{S}\,x))\,y))\,y\,x)\,(\mathsf{S}\,(\mathsf{S}\,\mathsf{Z}))\,(\mathsf{S}\,(\mathsf{S}\,\mathsf{Z})) \\
\rightsquigarrow^2 & \mathsf{case}\,\mathbb{N}\,(\mathsf{isZero}\,(\mathsf{S}\,(\mathsf{S}\,\mathsf{Z})))\,(\lambda x\,.\,\mathsf{abort})\,(\lambda x.(\mathsf{div}\,(\mathsf{S}\,(\mathsf{S}\,x))\,(\mathsf{S}\,(\mathsf{S}\,\mathsf{Z}))))\,(\mathsf{S}\,(\mathsf{S}\,\mathsf{Z}))\,(\mathsf{S}\,(\mathsf{S}\,\mathsf{Z})) \\
\rightsquigarrow^7 & (\mathsf{isZero}\,(\mathsf{S}\,(\mathsf{S}\,\mathsf{Z})))\,\mathbb{N}\,(\lambda x\,.\,\mathsf{abort})\,(\lambda x.(\mathsf{div}\,(\mathsf{S}\,(\mathsf{S}\,x))\,(\mathsf{S}\,(\mathsf{S}\,\mathsf{Z}))))\,(\mathsf{S}\,(\mathsf{S}\,\mathsf{Z}))\,(\mathsf{S}\,(\mathsf{S}\,\mathsf{Z})) \\
\rightsquigarrow^* & \mathsf{false}\,\mathbb{N}\,(\lambda x\,.\,\mathsf{abort})\,(\lambda x.(\mathsf{div}\,(\mathsf{S}\,(\mathsf{S}\,x))\,\mathsf{S}\,(\mathsf{S}\,\mathsf{Z})))\,(\mathsf{S}\,(\mathsf{S}\,\mathsf{Z}))\,(\mathsf{S}\,(\mathsf{S}\,\mathsf{Z})) \\
\rightsquigarrow^6 & (\lambda x.(\mathsf{div}\,(\mathsf{S}\,(\mathsf{S}\,x))\,\mathsf{S}\,(\mathsf{S}\,\mathsf{Z})))\,(\mathsf{S}\,(\mathsf{S}\,\mathsf{Z})) \\
\rightsquigarrow & \mathsf{div}\,(\mathsf{S}\,(\mathsf{S}\,(\mathsf{S}\,(\mathsf{S}\,\mathsf{Z}))))\,\mathsf{S}\,(\mathsf{S}\,\mathsf{Z}) \\
\rightsquigarrow^* & \mathsf{S}\,(\mathsf{S}\,\mathsf{Z})
\end{array}
$$

The previous example illustrates the syntax and the reduction relation of the freedom of speech language, but it also illustrates how the CBV reduction order can be manipulated using λ-abstractions. Notice that by wrapping the division function up in a λ-abstraction we prevent the reduction relation from reducing it until it is actually needed. If the isZero function returns true then the division function is never needed and thus is never ran. This shows how call-by-name reduction can be simulated by CBV reduction.

There is one final judgment defined with the reduction relation in Figure 34, and it is denoted $e_1 \downarrow e_2$. We call this the joinability judgment and is defined by only one rule ComputeJoin. This rule states that whenever there exists an expression e such that the expressions e_1 and e_2 both reduce to e in any number of steps, then e_1 and e_2 are joinable. If $e_1 \downarrow e_2$ holds then we consider the expressions e_1 and e_2 to

be equivalent. This judgment will be used to give the join expression its type in the typing rule join.

7.2 Type System

Throughout this thesis we have seen that the reduction rules paired with the type system is really where the heart of the programming language or type theory lives. The reduction rules tells us how to compute while the typing rules tell us which programs are valid and which programs we can consider as proofs. In this section we present the typing rules of Freedom of Speech, and we will make explicit how it contains all of the features we introduced in the introduction to this chapter.

The type system contains many rules, and so we break up the system and introduce it in chunks. To see the complete definition in one place see the appendix in [49]. Recall from the previous section that the syntax is collapsed. Types and programs are described by a single syntactic category called expressions. This then implies that we only have a single judgment defining the typing relation. This judgment is denoted $\Gamma \vdash^{\theta} e : e'$, and we read this judgment as the expression e has type e' in fragment θ in environment Γ. We can think of this judgment as being parametric in θ, and thus is two judgments in one. If θ is L, then e is a proof or logical program, while if θ is P, then e is a potentially diverging program. So it is the typing judgment that does the carving out of the two fragments.

We begin introducing the type system with the kinding rules:

$$\frac{}{\Gamma \vdash^{P} \mathsf{Type} : \mathsf{Type}}\ \text{K\_Type} \qquad \frac{}{\Gamma \vdash^{L} \mathbb{N} : \mathsf{Type}}\ \text{K\_Nat}$$

$$\frac{\Gamma \vdash^{\theta'} e_1 : \mathsf{Type} \quad \Gamma, x :^{\theta'} e_1 \vdash^{\theta} e_2 : \mathsf{Type}}{\Gamma \vdash^{\theta} (x :^{\theta'} e_1)^{\varepsilon} \to e_2 : \mathsf{Type}}\ \text{K\_Pi} \qquad \frac{\Gamma \vdash^{\theta_1} e : e_1 \quad \Gamma \vdash^{\theta_2} e' : e_2}{\Gamma \vdash^{L} e = e' : \mathsf{Type}}\ \text{K\_Eq}$$

The expression Type is a universe containing all of the expressions that can be considered well-defined types with respect to an environment. Throughout the sequel we sometimes denote expressions of type Type as A instead of e for readability. We call an expression e a type if and only if $\Gamma \vdash^{\theta} e : \mathsf{Type}$ for some environment Γ. If θ is L, then e is can be considered a formula otherwise e is simply a type. The rules above tell us that there are four types: the type Type, which is only a programmatic type (for the explanation why see Section 4.1), the type of natural numbers, dependent product types, and equations between well typed expressions. These are the only expressions that will be considered types.

A few remarks about the types. Notice that the rule K_Pi does not require the consistency classifier of the arguments to match the consistency classifier of the range type. This allows functions to take in arguments from one fragment and then produce a program of the opposite fragment. For example, the type $(x :^{P} \mathbb{N})^{+} \to x = x$ takes a programmatic argument, but produces a formula which is the statement that x is equivalent to itself. Thus, formulas can depend on potentially diverging programs. This is exactly the freedom of speech property. Finally, note that the expressions of an equation do not have to have the same type. This is called heterogenous equality. In addition, note that the consistency classifiers for the two equations can be different. For example, $\mathsf{S}\,\mathsf{Z}$ is logical and programmatic, thus $\mathsf{S}\,\mathsf{Z} = \mathsf{S}\,\mathsf{Z}$, where the first is logical

and the second is programmatic, is a valid expression.

Now that we have characterized the types we can introduce how both logical and programmatic programs are typed. First, we introduce the axioms for natural numbers and assumptions:

$$\frac{}{\Gamma \vdash^L \mathsf{S} : (x :^L \mathbb{N})^+ \to \mathbb{N}}\ \text{Succ} \qquad \frac{}{\Gamma \vdash^L \mathsf{Z} : \mathbb{N}}\ \text{Zero} \qquad \frac{\Gamma \vdash^\theta e : \mathsf{Type} \quad x :^\theta e \in \Gamma}{\Gamma \vdash^\theta x : e}\ \text{Var}$$

The astute reader will have noticed that natural numbers are typed only in the logical fragment, but one might think that these are purely programmatic, because they do not make for very interesting formulas. As we mentioned above the logical fragment can be considered as a terminating functional programming language so if one wishes to ensure that the program they are constructing is terminating then they can construct it in the logical fragment. To use the natural numbers in the programmatic fragment we add an additional typing rule:

$$\frac{\Gamma \vdash^L e : e_1}{\Gamma \vdash^P e : e_1}\ \text{Coerce}$$

This rule allows any well-typed logical expression to be coerced into the programmatic fragment. This corresponds to the edge relating the logical fragment to the programmatic fragment in the diagram * from the introduction of this chapter. Thus, natural numbers can be used in either fragment. The third rule we introduce above is the variable rule Var which depends on the premise $\Gamma \vdash^\theta e : \mathsf{Type}$. This premise – we will see these all throughout the definition of the typing relation – ensures that the expression e is indeed a type. This is necessary because we only have one syntactic

category for expressions, so the only way to tell the difference between a type and an expression is by using the typing judgment.

Programmatic and logical functions are introduced using two styles of λ-abstractions:

$$\frac{\Gamma \vdash^{\theta} e_1 : \mathsf{Type} \quad \Gamma, x :^{\theta} e_1 \vdash^{\theta'} e : e_2}{\Gamma \vdash^{\theta'} \lambda x \,.\, e : (x :^{\theta} e_1)^{+} \to e_2}\ \text{Lam} \qquad \frac{\Gamma \vdash^{\theta'} e_1 : \mathsf{Type} \quad \Gamma, x :^{\theta'} e_1 \vdash^{\theta} v : e_2 \quad x \notin \mathsf{FV}(v)}{\Gamma \vdash^{\theta} v : (x :^{\theta'} e_1)^{-} \to e_2}\ \text{ILam}$$

The former introduces λ-abstractions with runtime relevant arguments, while the latter introduces λ-abstractions whose argument is compile time relavent only. We call the latter implicit λ-abstractions, because the argument is left implicit. The Lam rule is intuitive, but notice that again the arguments consistency classifier does not have to match the bodies, thus allowing freedom of speech.

The typing rule ILam has a restriction on the body of the implicit λ-abstraction. In fact, there are a number of value restrictions in the definition of the typing relation. This restriction is necessary in order to maintain the meta-theoretic property called progress. This property states that every well-typed program can either take a computational step or is already a value. Suppose the restriction on ILam is lifted. Then the judgment $\cdot \vdash^{L} \mathsf{Z}\,\mathsf{Z} : (x :^{L} \mathbb{N} = (y :^{L} \mathbb{N})^{+} \to \mathbb{N})^{-} \to \mathbb{N}$ holds, but $\mathsf{Z}\,\mathsf{Z}$ is neither a value nor a redex. It is a stuck term! Thus, this value restriction is necessary to rule out stuck terms of this form.

Now we have two distinct λ-abstractions, and thus we will need two types of applications:

$$\frac{\Gamma \vdash^{\theta'} [v/x]e_2 : \mathsf{Type} \quad \Gamma \vdash^{\theta'} e : (x :^{\theta} e_1)^{+} \to e_2 \quad \Gamma \vdash^{\theta} v : e_1}{\Gamma \vdash^{\theta'} e\ v : [v/x]e_2}\ \text{AppPiTerm} \qquad \frac{\Gamma \vdash^{\theta'} [v/x]e_2 : \mathsf{Type} \quad \Gamma \vdash^{\theta'} e : (x :^{\theta} e_1)^{-} \to e_2 \quad \Gamma \vdash^{\theta} v : e_1}{\Gamma \vdash^{\theta'} e : [v/x]e_2}\ \text{AppAllTerm}$$

The former eliminates function types with runtime arguments, while the latter eliminates function types with compile-time arguments. These rules are straightforward, but do contain value restrictions similar to ILam. These restrictions prevent the application of a logical function to a diverging non-value computation. For example, if we lift the value restriction then we could type $(e\ \mathsf{loop})$ in the logical fragment, where loop is a program that simply calls itself infinitely often, but then the term $(e\ \mathsf{loop})$ diverges, and hence is no longer a proof. Remember, logical programs cannot run programmatic programs, however, the term $(e\ \mathsf{loop})$ is a prefect example of a programmatic expression.

So far we have seen the typing rules for types, natural numbers, λ-abstractions, and applications. Now we introduce the typing rules for introducing and eliminating – or using – equations. The typing rules are as follows:

$$\frac{e \downarrow e' \quad \Gamma \vdash^{\theta_1} e : e_1 \quad \Gamma \vdash^{\theta_2} e' : e_2}{\Gamma \vdash^{L} \mathsf{join} : e = e'}\ \text{join} \qquad \frac{\Gamma \vdash^{\theta} [e_1'/x]e_2 : \mathsf{Type} \quad \Gamma \vdash^{\theta} e : [e_1/x]e_2 \quad \Gamma \vdash^{L} e' : e_1 = e_1'}{\Gamma \vdash^{\theta} e : [e_1'/x]e_2}\ \text{Conv}$$

The former introduces an equation by first judging that the expressions e and e' are joinable as defined in Figure 34, then requiring that e and e' be well typed. Note that the expressions e and e' can have different types and consistency classifiers. Thus, this rule introduces heterogenous equality. Now equations are completely logical hence we

can consider join as the proof that e and e' are equivalent. Furthermore, e_1 and e_2 can be Type, thus e and e' can be types making join a proof of an equation between types. We can now introduce equations, but how are they used? If we have a proof that the expressions e_1 and e_1' are equivalent, that is we know $\Gamma \vdash^L e' : e_1 = e_1'$, and if we also have an expression e whose type depends on e_1, then the rule Conv allows us to replace e_1 in the type e with its equivalent e_1'. Thus, the rule Conv allows one to substitute equals for equals in types.

The rules Join and Conv have one particular disadvantage. Suppose

$\Gamma \overset{\mathsf{def}}{\equiv} z :^L \mathsf{Type}, y :^L \mathsf{Type}, x :^L z, u :^L ((x_1 :^L \mathbb{N})^+ \to z) = ((x_1 :^L \mathbb{N})^+ \to y)$,
$\Gamma \vdash^L \lambda x_1 \,.\, x : (x_1 :^L \mathbb{N})^+ \to z$, and
$\Gamma \vdash^L \mathsf{Z} : \mathbb{N}$.

Then $\Gamma \vdash^L (\lambda x_1 \,.\, x)\,\mathsf{Z} : z$. By applying Conv and using the assumption u, we may conclude $\Gamma \vdash^L (\lambda x_1 \,.\, x)\,\mathsf{Z} : y$, but $(\lambda x_1 \,.\, x)\,\mathsf{Z} \rightsquigarrow x$ and $\Gamma \vdash^L x : z$. Therefore, we have obtained a counterexample to type preservation! Notice that if we knew $z = y$ then this would no longer be a counterexample, but it is impossible to prove that $z = y$ from knowing $((x_1 :^L \mathbb{N})^+ \to z) = ((x_1 :^L \mathbb{N})^+ \to y)$ using only Join and Conv. So to prevent counterexamples such as these we add the following rules:

$$\frac{\begin{array}{l}\Gamma \vdash^L e_1 = e_1' : \mathsf{Type} \\ \Gamma \vdash^L e' : ((x :^\theta e_1)^\varepsilon \to e_2) = ((x :^\theta e_1')^\varepsilon \to e_2')\end{array}}{\Gamma \vdash^L \mathsf{injdom} : e_1 = e_1'}\ \text{InjDom}$$

$$\frac{\begin{array}{l}\Gamma \vdash^L [v/x]e_2 = [v/x]e_2' : \mathsf{Type} \\ \Gamma \vdash^L e' : ((x :^\theta e_1)^\varepsilon \to e_2) = ((x :^\theta e_1')^\varepsilon \to e_2') \\ \Gamma \vdash^\theta v : e_1\end{array}}{\Gamma \vdash^L \mathsf{injran} : [v/x]e_2 = [v/x]e_2'}\ \text{InjRan}$$

These rules are known as injectivity rules for dependent products. The second rule may seem a bit strange, especially the second premise, because we are substituting

a value v for two free variables of possibly different types. This is, however, not a problem, because from the premises of InjRan, InjDom, and Conv we may conclude $\Gamma \vdash^{\theta} v : e_1'$. Then by InjDom we know $\Gamma \vdash^{L} \mathsf{injdom} : e_1 = e_1'$, and clearly $\Gamma \vdash^{\theta} v : e_1$ is equivalent to $\Gamma \vdash^{\theta} v : [e_1/x]x$. Thus by Conv, $\Gamma \vdash^{\theta} v : [e_1'/x]x$, which is equivalent to $\Gamma \vdash^{\theta} v : e_1'$. So the rule InjRan is sound.

Equational reasoning is based on the runtime behavior of programs, but there are a number of different programs of a different type so what happens when it is possible to obtain a equation between contradicting values? The reader may object to the very question, because the logical fragment – as we have said above – must be consistent, but it is possible to start with inconsistent assumptions, and in these cases the logical fragment needs to be able to trigger a contradiction. We introduce the following rules to eliminate contradictory equations:

$$\frac{\Gamma \vdash^{\theta} e_1 : \mathsf{Type}}{\Gamma \vdash^{P} \mathsf{abort} : e_1}\ \text{Abort} \qquad \frac{\Gamma \vdash^{\theta} e_1 : \mathsf{Type} \quad \Gamma \vdash^{L} e : \mathsf{Z} = \mathsf{S}\, e'}{\Gamma \vdash^{L} \mathsf{contra} : e_1}\ \text{Contra}$$

$$\frac{\Gamma \vdash^{\theta} e_1 : \mathsf{Type} \quad \Gamma \vdash^{L} e : v = \mathsf{abort}}{\Gamma \vdash^{L} \mathsf{contra} : e_1}\ \text{ContraAbort}$$

The first rule introduces abort and states that abort can have any type. As we have seen above abort is used to indicate error states. The second rule, ContraAbort, introduces the proof contra with any well-defined type as long as there is a proof of some value being equivalent to abort. This means, if a program enters an error state, then a contradiction has occurred. An example might be trying to verify the correctness of div for all natural numbers. Then when Z is given as the second

argument it will reduce to abort. Thus, we would be able to obtain our result by using contra. The third rule, Contra, is similar to the previous rule, except it requires a proof that Z is equivalent to a natural number greater than Z.

There is one additional type of inconsistent equation that might be introduced that can cause major problems. The fact that in an inconsistent context one can prove two dependent products equivalent when they have differing stage annotations or consistency classifiers can break freedom of speech. For example, it would be possible to equate programmatic functions taking programmatic arguments to programmatic functions taking logical arguments, which would be the opposite of the freedom of speech property. Even worse we could equate logical functions taking programmatic arguments to logical functions taking logical arguments which breaks the freedom of speech property. To observe inconsistencies arising from equations between dependent products having different compile time or runtime arguments, or different consistency classifiers we add the following two rules:

$$\frac{\begin{array}{l}\Gamma \vdash^{\theta''} e : \mathsf{Type} \\ \Gamma \vdash^{L} e' : ((x :^{\theta} e_1)^{\varepsilon} \to e_2) = ((x :^{\theta'} e_1')^{\varepsilon'} \to e_2') \\ \theta \neq \theta'\end{array}}{\Gamma \vdash^{L} \mathsf{contra} : e} \text{ ContraPiTh}$$

$$\frac{\begin{array}{l}\Gamma \vdash^{\theta''} e : \mathsf{Type} \\ \Gamma \vdash^{L} e' : ((x :^{\theta} e_1)^{\varepsilon} \to e_2) = ((x :^{\theta'} e_1')^{\varepsilon'} \to e_2') \\ \varepsilon \neq \varepsilon'\end{array}}{\Gamma \vdash^{L} \mathsf{contra} : e} \text{ ContraPiEp}$$

It is possible to obtain other inconsistent equations, for example, an equation between a dependent product and the type of natural numbers, but we do not add rules to explicitly observe these inconsistencies, because they are not needed to conduct the

meta-theoretic analysis of Freedom of Speech.

The final three rules of the freedom of speech design handle natural number – terminating – and general recursion. Terminating recursion is restricted to the logical fragment, but can be moved over to the programmatic fragment using the Coerce typing rule. Terminating recursion is introduced using the following rules:

$$\frac{\begin{array}{l}\Gamma, x :^L \mathbb{N} \vdash^L (y :^L \mathbb{N})^+ \to (p :^L x = \mathsf{S}\, y)^- \to [y/x]e_2 : \mathsf{Type} \\ \Gamma, x :^L \mathbb{N}, f :^L (y :^L \mathbb{N})^+ \to (p :^L x = \mathsf{S}\, y)^- \to [y/x]e_2 \vdash^L v : e_2 \\ f, p \notin \mathsf{FV}\,(e_2)\end{array}}{\Gamma \vdash^L \mathsf{rec}\, f\, x\, v : (x :^L \mathbb{N})^+ \to e_2}\ \text{RecNat}$$

$$\frac{\begin{array}{l}\Gamma, x :^L \mathbb{N} \vdash^L (y :^L \mathbb{N})^- \to (u :^L x = \mathsf{S}\, y)^- \to [y/x]e_2 : \mathsf{Type} \\ \Gamma, x :^L \mathbb{N}, f :^L (y :^L \mathbb{N})^- \to (u :^L x = \mathsf{S}\, y)^- \to [y/x]e_2 \vdash^L v : e_2 \\ f, p \notin \mathsf{FV}\,(e_2)\end{array}}{\Gamma \vdash^L \mathsf{rec}^-\ f\ v : (x :^L \mathbb{N})^- \to e_2}\ \text{RecNatComp}$$

A recursive definition is denoted $\mathsf{rec}\, f\, x\, v$ and should be thought of as a function from a natural number to some type e_2. So there are two rules: one with a runtime argument, and one with a compile time argument. The function we are defining we call f; f is bound in v. Now x is the recursive argument that decreases across recursive calls, which is also bound in v, finally v is the definition of the recursive function. Note that the invariant that the recursive argument decreases across recursive calls is captured by the type of f. We can see that the programmer has to provide a proof that x is one larger than the argument supplied to f at the sight of the recursive call. Thus, termination is clearly captured by the type of f.

Lastly, we have the rule for general recursion:

$$\frac{\begin{array}{l}\Gamma \vdash^{\theta'} e_1 : \mathsf{Type} \\ \Gamma, f :^{\theta} (x :^{\theta'} e_1)^+ \to e_2, x :^{\theta'} e_1 \vdash^{\theta} e : e_2\end{array}}{\Gamma \vdash^P \mathsf{rec}\, f\, x\, e : (x :^{\theta'} e_1)^+ \to e_2}\ \text{Rec}$$

This rule is – expectedly – restricted to the programmatic fragment. The things to note about this rule are that f has the exact same type as $\mathsf{rec}\, f\, x\, e$, and that there are no proofs requiring any arguments to decrease. Lastly, notice that the recursive function we are defining, that is f, has a general type. Furthermore, notice that e_2 can itself be Type, thus we can define types by recursion. These are known as large eliminations and facilitate generic programming.

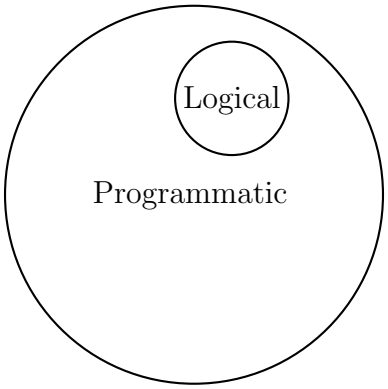
Throughout this chapter we have introduced the complete design of a new dependently typed functional programming language where proofs and general recursive programs can live in harmony with the freedom of speech property. As we discussed the design we paid careful attention to our design choices. Many of these choices were made to enforce particular meta-theoretic properties such as type safety and consistency. It turns out that type preservation and consistency do indeed hold, and their proofs are detailed in Chapter 10. However, we do not prove progress. There are two interesting insights we gained from the design and analysis of freedom of speech. The first one is that there are a number of value restrictions required throughout the typing rules. These make programming very difficult, and it is imperative to find ways to lift those restrictions. In fact, we conjecture that the value restrictions would no longer be needed if call-by-name reduction was adopted instead of CBV, but this is an open problem. The second interesting insight is meta-theoretic and is discussed in Chapter 10. In the following chapter we introduce the design of a dependently-typed functional programming language inspired by freedom of speech, but lifts all of the value restrictions, and is extended to include data types and pattern matching.

CHAPTER 8

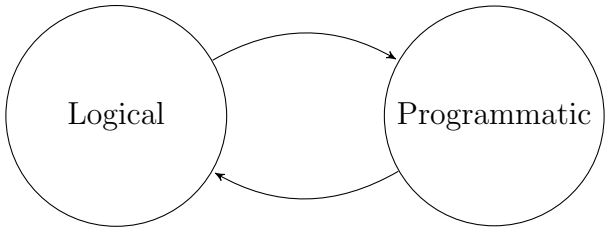
SEPARATION OF PROOF FROM PROGRAM

The previous chapter introduced the freedom of speech dependently-typed functional programming language. This language contains two fragments, the logical fragment and the programmatic fragment. However, these fragments are judgmentally kept separate. That is, the syntax was the same for both fragments, even more so, the syntax for programs and types are collapsed into a single syntactic category. This particular design is very appealing because it has an elegant definition, but this elegance comes at a cost. The type system suffers from many value restrictions, and as we have said above, this makes programming very hard.

Consider a design where instead of a judgmental separation of the two fragments – logical and programmatic – we insist on a syntactic one. This means that the logical and programmatic fragments have completely distinct languages for types and programs – but their semantics are not – and then the two fragments are related using freedom of speech by typing rules. So we go from a picture that looks like this:



To the following picture:



We have seen this diagram before in the previous chapter, but there it was used to give an intuitive understanding of the free speech property, but the two fragments are not strictly separate. Now we insist that they are in fact strictly separate, and the first diagram above is no longer applicable; unlike the freedom of speech language. We will see that this separation allows for the lifting of most of the value restrictions from the typing rules.

The freedom of speech language is not a very expressive programming language. For example, we did not show how to construct full programs, nor did freedom of speech contain any notion of abstract data types or pattern matching. Thus, it is rather difficult to consider freedom of speech as a real-world programming language. In this chapter we introduce the design of a new programming language called Separation of Proof from Program (Sep^3) that contains all of these new features. The logical and programmatic fragments are strictly separate, and Sep^3 is a full real-world programming language complete with abstract datatypes and pattern matching. There are also additional features that we will introduce below. The definition of Sep^3 is very large with over a hundred typing rules and a large amount of syntax. Therefore, we do not have the space to introduce the complete definition in the same style as we did for freedom of speech, but instead concentrate on the most important aspects of

the design. If the reader wishes to wade through the entire definition, then please see the appendix in [49] but the reader may wish to read this chapter first to understand the need for two definitions.

Sep^3 consists of two distinct phases: compile time and runtime, and Sep^3 separates each phase into its own language. The compile time phase uses the annotated Sep^3 language, and the runtime phase uses the unannotated Sep^3 language. The former is then translated into the second using an meta-level function called the eraser. This function removes the annotations from objects of the annotated language yielding objects of the unannotated language, furthermore, the eraser function removes all objects marked compile time, which includes the entire proof language. Thus to reiterate, the primary goal of the annotated language is type checking and proving, while the primary goal of the unannotated language is computation.

The reader may suspect that the unannotated language most likely consists primarily of the programmatic fragment, because if the proof language is removed and has no computational content, then why keep it around? This is precisely correct, and an implementation of Sep^3 would indeed do this, but we define an additional eraser function to essentially translate the proof language of Sep^3 into the Curry-style version, that is a language with no annotations. We conjecture that the metatheory of Sep^3 would be easier to carry out in the unannotated language rather than the annotated one. We do not pursue this strand of thought further, but wanted to simply comment on this notion. We do not define the eraser functions here, but the interested reader can find their complete definitions in the appendix. Throughout the

remainder of this chapter we will concentrate solely on the annotated Sep^3 language which we will just call Sep^3.

The overall picture. Sep^3 consists of four main judgments:

$\Delta, \Gamma \vdash LK : \mathsf{Logical}_i$	Super kinding.	Logical Fragment
$\Delta, \Gamma \vdash P : LK$	Kinding.	
$\Delta, \Gamma \vdash p : P$	Proof typing.	
$\Delta, \Gamma \vdash t : t'$	Program typing.	Programmatic Fragment

As the previous table indicates the logical fragment is broken up into three judgments. This is because the language of the logical fragment is no longer collapsed. There are distinct languages for proofs, predicates (types), kinds, and a new category we call super kinds. Each of these will be discussed further below. Now the programmatic fragment consists of only a single judgment, because it has a collapsed syntax very much like the freedom of speech language. In fact, the programmatic fragment is basically an extension of the freedom of speech language, and so is not really that different. For this reason we do not discuss the programmatic fragment in detail, but only discuss a few important aspects of the fragment.

In the freedom of speech language the connection between the two fragments is very explicit. The connection from the logical fragment to the programmatic fragment is modeled by the Coerce rule, and the connection from the programmatic fragment was modeled by the dependent product types, which allowed arguments to functions to be programs from the programmatic fragment. However, Sep^3 does not have a coercion rule from the logical fragment to the programmatic fragment. Instead both

fragments are related via the dependent product types. Logical programs are allowed to have programmatic inputs, and programmatic programs are allowed to have logical inputs.

The programmatic fragment. The programmatic fragment can be thought of as an extension of the programmatic fragment of the freedom of speech language. The extensions include datatypes, and pattern matching. There is one significant improvement to the programmatic fragment over the freedom of speech language, and that is the lifting of the value restrictions. For example, the following are the λ-abstraction and function application rules from the programmatic fragment of Sep[3]:

$$\frac{\Delta, \Gamma, x :^{\mathbf{val}} t_1 \vdash t : t_2}{\Delta, \Gamma \vdash \lambda x_+ : t_1.t : \Pi x_+ : t_1.t_2} \text{ TRM\_LamPL}$$

$$\frac{\begin{array}{l}\Delta, \Gamma, x : A \vdash t : t' \\ x \notin \mathsf{FV}(|t|)\end{array}}{\Delta, \Gamma \vdash \lambda x_- : A.t : \Pi x_- : A.t'} \text{ TRM\_LamMI} \qquad \frac{\begin{array}{l}\Delta, \Gamma \vdash t : \Pi x_\varepsilon : A.t' \\ \Delta, \Gamma \vdash a : A\end{array}}{\Delta, \Gamma \vdash t\ a_\varepsilon : [a/x]t'} \text{ TRM\_App}$$

We can see that these rules look very much like the rules of freedom of speech, but there are some small differences. First, we annotate bound variables by their stage annotations. This facilitates reasoning as well as the definitions of the eraser functions. Secondly, in the TRM_LamPL we annotate the free variable x in the context of the premise with the val annotation. This indicates that x ranges over programmatic values, and this is used to enforce call-by-value applications. Compare this to the rule TRM_LamMI where x does not only range over values, but also over diverging terms.

These rules include the freedom of speech property. The type A is an alias for one of t, P, or LK, which means compile-time functions – using TRM_LamMI – can

take in programmatic arguments, proof arguments, or even predicates.

Finally, the application rule TRM_App does not contain any value restriction. The argument a ranges over general terms, proofs, and predicates. Thus, we have successfully lifted the value restriction in the programmatic fragment.

The logical fragment. We would now like to introduce the logical fragment in some detail so the reader can get an idea of how different it is from the logical fragment of the freedom of speech language. The syntax of the logical fragment of Sep[3] is defined as follows:

(Super Kinds) $L ::= \mathsf{Logical}_i$

(Logical Kind) $LK ::= x$
$\mid \mathsf{Formula}_i$
$\mid \forall x : A.LK$

(Predicates) $P ::= x$
$\mid \Lambda x : A \,.\, P$
$\mid P\ a$
$\mid \forall x : A.P$
$\mid \mathsf{let}\ x = p\ in\ P$
$\mid \mathsf{let}\ x = P\ in\ P'$
$\mid \mathsf{let}\ x = t\ [p]\ in\ P$
$\mid t_1 = t_2$
$\mid t\,!$
$\mid P_1 + P_2$
$\mid \exists x : A.P$
$\mid \bot_i$
$\mid t < t'$

(Proofs) $p ::= x$
$\mid \mathsf{inl}\ p\ \mathsf{with}\ P$
$\mid \mathsf{inr}\ p\ \mathsf{with}\ P$
$\mid \mathbf{case}\ p\ \mathbf{of}\ x.p', y.p''$
$\mid \Lambda x : A \,.\, p$
$\mid p\ a$
$\mid (a, p)\ \mathsf{as}\ P$
$\mid \mathbf{case}\ p_1\ of\ (x, y).p_2$
$\mid \mathsf{let}\ x = p'\ in\ p$
$\mid \mathsf{let}\ x = P\ in\ p$
$\mid \mathsf{let}\ x = t\ [y]\ in\ p$
$\mid \mathsf{join}\ t_1\ t_2$
$\mid \mathbf{conv}\ p\ \mathbf{by}\ q_1 \ldots q_n\ \mathbf{at}\ x_1 \ldots x_m.P$
$\mid \mathbf{predconv}\ p\ P$
$\mid \mathbf{valax}\ t$
$\mid \mathbf{ord}\ t\ t'$
$\mid \mathbf{case}\ t[x]p\ \mathbf{of}\ R$
$\mid \mathbf{tcase}\ t[x]\ \mathbf{of}\ \mathsf{abort}\ \rightarrow p_1 |! \rightarrow p_2$
$\mid \mathbf{ind}\ f\ x : t, p_1.p_2$
$\mid \mathsf{contr}\ p_1$
$\mid \mathsf{contraval}\ p_1\ p_2$

Super kinds are the types of logical kinds, which are themselves the types of predicates. Finally, predicates are the types of proofs. The definition of the logical fragment is

very reminiscent of the definition of system F^{ω} – see Section 1.2 for an introduction of F^{ω}. In fact, predicates can be computed using x, $\Lambda x : A \,.\, P$, $P\ a$, $\mathsf{let}\, x = p\, in\, P$, $\mathsf{let}\, x = P\, in\, P'$, and $\mathsf{let}\, x = t\,[p]\, in\, P$; we have three different let expressions, because of the strict separation of the various languages. This implies that the logical fragment is indeed a higher order language, but a predicative one. We have stratified the super kinds and logical kinds into an infinite hierarchy of kinds. We denote this by $\mathsf{Logical}_i$ and $\mathsf{Formula}_i$. This stratification is similar to levels in SSF – see Section 1.2 for more information on SSF.

Now in comparison with the freedom of speech language we can see that Sep^3 has three new predicates: $t\,!$, $P_1 + P_2$, $\exists x : A.P$, and $t < t'$. The first is called the termination predicate and we will discuss this further below. The second is the type of sum types or, logically, the type for disjunction, and the third is the type for existential quantification. Note that we can existentially quantify over logical kinds, predicates, proofs, and terms. Thus, the existential type makes use of the freedom of speech property similar to the dependent product type. The final new predicate is the structural ordering type $t < t'$ which says that t is structurally smaller than t'. The structural ordering type will be discussed further below.

The proof language is fairly straightforward, and is very similar to the freedom of speech language with the addition of the necessary proofs of the new predicates just discussed. Thus, we do not go over each proof construct. The typing rules for logical kinds, predicates, and proofs can be found in Figure 35, Figure 36, Figure 37, and Figure 38 respectively. The reader may wish to look over the typing rules for

$$\frac{\Gamma\ \mathsf{Ok} \quad \Delta\ \mathsf{Ok}}{\Delta, \Gamma \vdash \mathsf{Formula}_i : \mathsf{Logical}_{(i+1)}}\ \text{LK\_Formula} \qquad \frac{\Delta, \Gamma \vdash A : \mathcal{A}_i \quad \Delta, \Gamma, x : A \vdash LK : \mathsf{Logical}_j}{\Delta, \Gamma \vdash \forall x\ :\ A.LK : \mathsf{Logical}_{\mathbf{max}\,(i+1,j)}}\ \text{LK\_Predicate}$$

Figure 35. Type-checking Rules for Logical Kinds

$$\frac{\Delta, \Gamma \vdash LK : \mathsf{Logical}_i \quad x :^{\gamma} LK \in \Gamma}{\Delta, \Gamma \vdash x : LK}\ \text{PRD\_Var} \qquad \frac{\Gamma\ \mathsf{Ok} \quad \Delta\ \mathsf{Ok} \quad x = (P, LK)\ \in\ \Delta}{\Delta, \Gamma \vdash x : LK}\ \text{PRD\_GD} \qquad \frac{\Gamma\ \mathsf{Ok} \quad \Delta\ \mathsf{Ok}}{\Delta, \Gamma \vdash \bot_i : \mathsf{Formula}_i}\ \text{PRD\_Btm}$$

$$\frac{\Delta, \Gamma \vdash P_1 : \mathsf{Formula}_i \quad \Delta, \Gamma \vdash P_2 : \mathsf{Formula}_j}{\Delta, \Gamma \vdash P_1 + P_2 : \mathsf{Formula}_{\mathbf{max}\,(i,j)}}\ \text{PRD\_Disj} \qquad \frac{\Delta, \Gamma \vdash P' : LK \quad \Delta, \Gamma \vdash LK : \mathsf{Logical}_i \quad \Delta, \Gamma, x : P' \vdash P : \mathsf{Formula}_j}{\Delta, \Gamma \vdash \forall x\ :\ P'.P : \mathsf{Formula}_{\mathbf{max}\,(i,j)}}\ \text{PRD\_Forall1}$$

$$\frac{\Delta, \Gamma \vdash t : \mathsf{Type}_0 \quad \Delta, \Gamma, x : t \vdash P : \mathsf{Formula}_i}{\Delta, \Gamma \vdash \forall x\ :\ t.P : \mathsf{Formula}_{\mathbf{max}\,(1,i)}}\ \text{PRD\_Forall2} \qquad \frac{\Delta, \Gamma \vdash t : \mathsf{Type}_i \quad \Delta, \Gamma, x : t \vdash P : \mathsf{Formula}_j}{\Delta, \Gamma \vdash \forall x\ :\ t.P : \mathsf{Formula}_{\mathbf{max}\,(i+1,j)}}\ \text{PRD\_Forall3}$$

$$\frac{\Delta, \Gamma \vdash LK : \mathsf{Logical}_i \quad \Delta, \Gamma, x : LK \vdash P : \mathsf{Formula}_j}{\Delta, \Gamma \vdash \forall x\ :\ LK.P : \mathsf{Formula}_{\mathbf{max}\,(i,j)}}\ \text{PRD\_Forall4} \qquad \frac{\Delta, \Gamma \vdash P' : LK \quad \Delta, \Gamma \vdash LK : \mathsf{Logical}_i \quad \Delta, \Gamma, x : P' \vdash P : \mathsf{Formula}_j}{\Delta, \Gamma \vdash \exists x : P'.P : \mathsf{Formula}_{\mathbf{max}\,(i,j)}}\ \text{PRD\_Ext1}$$

$$\frac{\Delta, \Gamma \vdash t : \mathsf{Type}_0 \quad \Delta, \Gamma, x : t \vdash P : \mathsf{Formula}_i}{\Delta, \Gamma \vdash \exists x : t.P : \mathsf{Formula}_{\mathbf{max}\,(1,i)}}\ \text{PRD\_Ext2} \qquad \frac{\Delta, \Gamma \vdash t : \mathsf{Type}_i \quad \Delta, \Gamma, x : t \vdash P : \mathsf{Formula}_j}{\Delta, \Gamma \vdash \exists x : t.P : \mathsf{Formula}_{\mathbf{max}\,(i+1,j)}}\ \text{PRD\_Ext3}$$

$$\frac{\Delta, \Gamma \vdash LK : \mathsf{Logical}_i \quad \Delta, \Gamma, x : LK \vdash P : \mathsf{Formula}_j}{\Delta, \Gamma \vdash \exists x : LK.P : \mathsf{Formula}_{\mathbf{max}\,(i,j)}}\ \text{PRD\_Ext4} \qquad \frac{\Delta, \Gamma \vdash p : P' \quad \Delta, \Gamma, x : P' \vdash P : LK \quad x\ \notin\ \mathsf{FV}\,(p)}{\Delta, \Gamma \vdash \mathsf{let}\, x = p\, in\, P : LK}\ \text{PRD\_LetPF}$$

$$\frac{\Delta, \Gamma \vdash P : LK' \quad \Delta, \Gamma, x : LK \vdash P' : LK \quad x\ \notin\ \mathsf{FV}\,(P)}{\Delta, \Gamma \vdash \mathsf{let}\, x = P\, in\, P' : LK}\ \text{PRD\_LetPRD} \qquad \frac{\Delta, \Gamma \vdash t : t' \quad \Delta, \Gamma, x : t', x' : x = t \vdash P : LK \quad x\ \notin\ \mathsf{FV}\,(t)}{\Delta, \Gamma \vdash \mathsf{let}\, x = t\,[x']\, in\, P : LK}\ \text{PRD\_Let}$$

$$\frac{\Delta, \Gamma \vdash t : t_1 \quad \Delta, \Gamma \vdash t' : t_2}{\Delta, \Gamma \vdash t = t' : \mathsf{Formula}_i}\ \text{PRD\_K\_Eq} \qquad \frac{\Delta, \Gamma \vdash t : t'}{\Delta, \Gamma \vdash t\,! : \mathsf{Formula}_i}\ \text{PRD\_TRM}$$

$$\frac{\Delta, \Gamma \vdash A : W \quad \Delta, \Gamma, x : A \vdash P : LK}{\Delta, \Gamma \vdash \Lambda x : A\,.\,P : \forall x\ :\ A.LK}\ \text{PRD\_Lam} \qquad \frac{\Delta, \Gamma \vdash P : \forall x\ :\ A.LK \quad \Delta, \Gamma \vdash a : A}{\Delta, \Gamma \vdash P\ a : [a/x]LK}\ \text{PRD\_App}$$

Figure 36. Type-checking Rules for Predicates

$$\frac{\Delta, \Gamma \vdash P : LK \quad x :^{\gamma} P \in \Gamma}{\Delta, \Gamma \vdash x : P}\ \text{PRF\_Var} \qquad \frac{\Gamma\ \mathsf{Ok} \quad \Delta\ \mathsf{Ok} \quad x = (p, P)\ \in\ \Delta}{\Delta, \Gamma \vdash x : P}\ \text{PRF\_GD}$$

$$\frac{\Delta, \Gamma \vdash p : [a/x]P \quad \Delta, \Gamma \vdash a : A}{\Delta, \Gamma \vdash (a, p)\ \mathsf{as}\ (\exists x : A.P) : \exists x : A.P}\ \text{PRF\_Exti} \qquad \frac{\Delta, \Gamma \vdash p_1 : \exists x : A.P \quad \Delta, \Gamma, x : A, y : P \vdash p_2 : P' \quad x\ \notin\ \mathsf{FV}\,(P')}{\Delta, \Gamma \vdash \mathbf{case}\ p_1\ of\ (x, y).p_2 : P'}\ \text{PRF\_ExtE}$$

$$\frac{\Delta, \Gamma \vdash p : P_1 \quad \Delta, \Gamma \vdash P_2 : \mathsf{Formula}_i}{\Delta, \Gamma \vdash \mathsf{inl}\ p\ \mathsf{with}\ P_2 : P_1 + P_2}\ \text{PRF\_Inl} \qquad \frac{\Delta, \Gamma \vdash p : P_2 \quad \Delta, \Gamma \vdash P_1 : \mathsf{Formula}_i}{\Delta, \Gamma \vdash \mathsf{inr}\ p\ \mathsf{with}\ P_1 : P_1 + P_2}\ \text{PRF\_Inr}$$

$$\frac{\Delta, \Gamma \vdash p : P_1 + P_2 \quad \Delta, \Gamma, x : P_1 \vdash p' : P \quad \Delta, \Gamma, x : P_2 \vdash p'' : P}{\Delta, \Gamma \vdash \mathbf{case}\ p\ \mathbf{of}\ x.p', x.p'' : P}\ \text{PRF\_OrElim} \qquad \frac{\Delta, \Gamma \vdash t : \mathsf{Type}_i \quad \Delta, \Gamma, x : t \vdash p : P}{\Delta, \Gamma \vdash \Lambda x : t\,.\,p : \forall x\ :\ t.P}\ \text{PRF\_FT}$$

$$\frac{\Delta, \Gamma \vdash P : LK \quad \Delta, \Gamma \vdash LK : \mathsf{Logical}_i \quad \Delta, \Gamma, x :^{\mathbf{val}} P \vdash p : P'}{\Delta, \Gamma \vdash \Lambda x : P\,.\,p : \forall x\ :\ P.P'}\ \text{PRF\_FPRD} \qquad \frac{\Delta, \Gamma \vdash LK : \mathsf{Logical}_i \quad \Delta, \Gamma, x :^{\mathbf{val}} LK \vdash p : P}{\Delta, \Gamma \vdash \Lambda x : LK\,.\,p : \forall x\ :\ LK.P}\ \text{PRF\_FLK}$$

$$\frac{\Delta, \Gamma \vdash p : \forall x\ :\ A.P \quad \Delta, \Gamma \vdash a : A}{\Delta, \Gamma \vdash p\ a : [a/x]P}\ \text{PRF\_App} \qquad \frac{\Delta, \Gamma \vdash p' : P' \quad \Delta, \Gamma, x : P' \vdash p : P \quad x\ \notin\ \mathsf{FV}\,(p')}{\Delta, \Gamma \vdash \mathsf{let}\ x = p'\ in\ p : P}\ \text{PRF\_LetPRF}$$

$$\frac{\Delta, \Gamma \vdash P : LK \quad \Delta, \Gamma, x : LK \vdash p : P \quad x\ \notin\ \mathsf{FV}\,(P)}{\Delta, \Gamma \vdash \mathsf{let}\ x = P\ in\ p : P}\ \text{PRF\_LetPRD}$$

Figure 37. Type-checking Rules for Proofs

$$\frac{\Delta, \Gamma \vdash t : t' \quad \Delta, \Gamma, x : t', x' : x = t \vdash p : P \quad x \notin \mathsf{FV}(t)}{\Delta, \Gamma \vdash \mathsf{let}\, x = t\,[x']\, in\, p : P}\ \text{PRF\_Let} \qquad \frac{|t| \curlyvee |t'| \quad \Delta, \Gamma \vdash t : t_1 \quad \Delta, \Gamma \vdash t' : t_2}{\Delta, \Gamma \vdash \mathsf{join}\, t\, t' : t = t'}\ \text{PRF\_Join}$$

$$\frac{x_1, \ldots, x_n \notin \mathsf{FV}(|P|) \quad \Delta, \Gamma \vdash p : [t_1/x_1] \ldots [t_n/x_n]P \quad \Delta, \Gamma \vdash [t'_1/x_1] \ldots [t'_n/x_n]P : \mathsf{Formula}_i \quad \Delta, \Gamma \vdash\ eqpf\ q_1\, \varepsilon_1 : t_1 = t'_1 \ldots \Delta, \Gamma \vdash\ eqpf\ q_n\, \varepsilon_n : t_n = t'_n}{\Delta, \Gamma \vdash \mathbf{conv}\, p\, \mathbf{by}\, q_1 \ldots q_n\, \mathbf{at}\, x_1 \ldots x_n.P : [t'_1/x_1] \ldots [t'_n/x_n]P}\ \text{PRF\_Conv}$$

$$\frac{\Delta, \Gamma \vdash p : P \quad P =_\beta P'}{\Delta, \Gamma \vdash \mathbf{predconv}\, p\, P' : P'}\ \text{PRF\_PRDConv} \qquad \frac{\Delta, \Gamma \vdash \mathbf{val}\ t}{\Delta, \Gamma \vdash \mathbf{valax}\, t : t\,!}\ \text{PRF\_Val}$$

$$\frac{\Delta, \Gamma \vdash t : t_1 \quad \Delta, \Gamma \vdash t' : t_2 \quad \Delta, \Gamma \vdash \mathbf{val}\ t \quad \Delta, \Gamma \vdash \mathbf{val}\ t' \quad t \sqsubseteq^+ t'}{\Delta, \Gamma \vdash \mathbf{ord}\, t\, t' : t < t'}\ \text{PRF\_Ord} \qquad \frac{\Delta, \Gamma, x : t_1, u :^{\mathbf{val}} x\,!, f :^{\mathbf{val}} \forall y : t_2.\forall u : y < x.[y/x]P \vdash p : P}{\Delta, \Gamma \vdash \mathbf{ind}\, f\, x : t_1, u.p : \forall x : t_1.\forall u : x\,!.P}\ \text{PRF\_Ind}$$

$$\frac{\Delta, \Gamma \vdash P : \mathsf{Formula}_i \quad \Delta, \Gamma \vdash p' : \mathsf{C}\, a_{1\varepsilon_1} \ldots a_{r\varepsilon_r} = \mathsf{C}\, a'_{1\varepsilon'_1} \ldots a'_{s\varepsilon'_s} \quad \mathsf{C} \neq \mathsf{K} \quad \forall a \in \{a_1, \ldots, a_r\}.((\exists \mathbf{t}.\Delta, \Gamma \vdash a : t) \implies (\exists t', p.(a \equiv t' \wedge \Delta, \Gamma \vdash p : t'\,!))) \quad \forall a \in \{a'_1, \ldots, a'_s\}.((\exists \mathbf{t}.\Delta, \Gamma \vdash a : t) \implies (\exists t', p.(a \equiv t' \wedge \Delta, \Gamma \vdash p : t'\,!)))}{\Delta, \Gamma \vdash \mathsf{contr}\, p' : P}\ \text{PRF\_CTR1}$$

$$\frac{\Delta, \Gamma \vdash p : \bot_i \quad \Delta, \Gamma \vdash P : \mathsf{Formula}_i}{\Delta, \Gamma \vdash \mathsf{contr}\, p : P}\ \text{PRF\_CTR2} \qquad \frac{\Delta, \Gamma \vdash P : \mathsf{Formula}_i \quad \Delta, \Gamma \vdash p_1 : t = \mathsf{abort}\, t' \quad \Delta, \Gamma \vdash p_2 : t\,!}{\Delta, \Gamma \vdash \mathsf{contraval}\, p_1\, p_2 : P}\ \text{PRF\_CTRV}$$

$$\frac{\Delta, \Gamma \vdash t : t' \quad \Delta, \Gamma \vdash p : t\,! \quad \Delta, \Gamma \vdash^{PB} R\, t\, t'\, y\, \Delta_3(\mathsf{getHC}(t')) : P}{\Delta, \Gamma \vdash \mathbf{case}\, t[y]p\, \mathbf{of}\, R : P}\ \text{PRF\_Case} \qquad \frac{\Delta, \Gamma \vdash t : t' \quad \Delta, \Gamma, u :^{\mathbf{val}} t\,! \vdash p_2 : P \quad \Delta, \Gamma, u :^{\mathbf{val}} \mathsf{abort}\, t' = t \vdash p_1 : P}{\Delta, \Gamma \vdash \mathbf{tcase}\, t[u]\, \mathbf{of}\, \mathsf{abort}\ \to p_1|!\to p_2 : P}\ \text{PRF\_TCase}$$

Figure 38. Type-checking Rules for Proofs Continued

proofs to get an idea of which proof construct introduces each of the new predicates.

The termination prediate. In the freedom of speech language if one wanted to prove that a programmatic program is terminating one simply had to define it in the logical fragment, but this is not possible in Sep^3, because of the strict separation of the logical and programmatic fragments. The termination predicate adds the ability to prove programmatic programs terminating. The termination predicate is denoted $t\,!$, and can be read as the program t terminates at a value. We prove $t\,!$ using the rule PRF_Val:

$$\frac{\Delta, \Gamma \vdash \mathbf{val}\ t}{\Delta, \Gamma \vdash \mathbf{valax}\, t : t\,!}\ \text{PRF\_Val}$$

This rule says, that if we can show $\Delta, \Gamma \vdash \mathbf{val}\ t$, then we can prove $t\,!$. We call the judgment $\Delta, \Gamma \vdash \mathbf{val}\ t$ the semantic value judgment. If it holds, then t is either an actual syntactic value, a free variable annotated in the context as ranging over values, or a termination cast. See Figure 39 for the definition of the semantic value judgment. One may feel that the rule PRF_Val is a bit restrictive. This rule is supposed to allow the proof that a terminating program is indeed terminating, but on the nose it seems that the only things we can prove are terminating are semantic values, which are trivially terminating. However, this is not the case. If one can prove that t is terminating for some non-value t, then one must have constructed the value of t, say v. Now it is trivially the case that t and v are joinable, thus using PRF_Join we can prove $t = v$. In addition, since we know v is a value, then we can prove that it is a semantic value trivially. Thus, using these two proofs we can then prove $\Delta, \Gamma \vdash \mathbf{valax}\, v : t\,!$ using the rules PRF_Val and PRF_Conv.

$$\frac{\begin{array}{l}\Gamma\ \mathsf{Ok}\\ \Delta\ \mathsf{Ok}\\ x :^{\mathbf{val}} t \in \Gamma\end{array}}{\Delta, \Gamma \vdash \mathbf{val}\ x}\ \text{V\_Var} \qquad \frac{\begin{array}{c}\Gamma\ \mathsf{Ok}\\ \Delta\ \mathsf{Ok}\end{array}}{\Delta, \Gamma \vdash \mathbf{val}\ \mathsf{Type}_i}\ \text{V\_Type} \qquad \frac{\begin{array}{c}\Gamma\ \mathsf{Ok}\\ \Delta\ \mathsf{Ok}\end{array}}{\Delta, \Gamma \vdash \mathbf{val}\ \Pi x_\varepsilon : A.t}\ \text{V\_Pi}$$

$$\frac{\begin{array}{c}\Gamma\ \mathsf{Ok}\\ \Delta\ \mathsf{Ok}\end{array}}{\Delta, \Gamma \vdash \mathbf{val}\ \lambda x_+ : A.t}\ \text{V\_LamPlus} \qquad \frac{\Delta, \Gamma \vdash \mathbf{val}\ t}{\Delta, \Gamma \vdash \mathbf{val}\ \lambda x_- : A.t}\ \text{V\_LamMinus}$$

$$\frac{\begin{array}{c}\Gamma\ \mathsf{Ok}\\ \Delta\ \mathsf{Ok}\end{array}}{\Delta, \Gamma \vdash \mathbf{val}\ \mathsf{rec}\, f\ x : t_1.t_2}\ \text{V\_Rec} \qquad \frac{\Delta, \Gamma \vdash \text{v-or-p}\ a_1 \ldots \Delta, \Gamma \vdash \text{v-or-p}\ a_n}{\Delta, \Gamma \vdash \mathbf{val}\ \mathsf{C}\ a_{1_{\varepsilon_1}} \ldots a_{n_{\varepsilon_n}}}\ \text{V\_Ctor}$$

$$\frac{\begin{array}{c}\Gamma\ \mathsf{Ok}\\ \Delta\ \mathsf{Ok}\end{array}}{\Delta, \Gamma \vdash \mathbf{val}\ \mathbf{tcast}\ t\ \mathbf{by}\ p}\ \text{V\_tCast}$$

Figure 39. Semantic Values

The termination cast is a means of casting programs to values if they can be proven to be terminating. That is, any terminating program may be treated as a value. Lets consider the introduction rule for the termination cast:

$$\frac{\begin{array}{c}\Delta, \Gamma \vdash t : t'\\ \Delta, \Gamma \vdash p : t\,!\end{array}}{\Delta, \Gamma \vdash \mathbf{tcast}\ t\ \mathbf{by}\ p : t'}\ \text{TRM\_tCast}$$

This rules states that if we have a well-typed t of type t', and a proof p that t terminates, that is of type $t\,!$, then we may cast t to a value denoted $\mathbf{tcast}\ t\ \mathbf{by}\ p$ which is also of type t'.

There is one final component of the termination predicate called the termination case expression. This is an expression that amounts to computationally case

splitting over the termination behavior of a program. Now do not be alarmed! This does not imply that this is a proof of the halting problem. Consider the typing rule for the termination case expression:

$$\frac{\begin{array}{l}\Delta, \Gamma \vdash t : t' \\ \Delta, \Gamma, u :^{\mathbf{val}} t\,! \vdash p_2 : P \\ \Delta, \Gamma, u :^{\mathbf{val}} \mathsf{abort}\, t' = t \vdash p_1 : P\end{array}}{\Delta, \Gamma \vdash \mathbf{tcase}\, t[u]\, \mathbf{of}\, \mathsf{abort} \rightarrow p_1 | ! \rightarrow p_2 : P}\ \text{PRF\_TCase}$$

This rule status that if t terminates then execute the second branch p_2, which is allowed to use the proof showing that t terminates, otherwise if t is diverges – here we model divergence using $\mathsf{abort}\, t'$ – then execute the first branch p_1 which is allowed to use the proof showing t is equivalent to $\mathsf{abort}\, t'$. Now computationally speaking this rule is undecidable. It is impossible to take an arbitrary program and decide whether it diverges or not, but adding this axiom is not inconsistent, because it simply captures the notion that either a program terminates or it diverges, and this is no more inconsistent then adding the law-of-excluded middle to our logic. In fact, one might expect that adding termination case could imply the law of excluded middle making our logic classical, but this is an open problem.

Structural ordering. There is one final interesting feature that sets Sep^3 apart from the freedom of speech language, and many others. Recall that in the freedom of speech language the logical fragment contained a terminating recursor, but the recursive argument had to be a natural number. Sep^3 relaxes this requirement to an arbitrary datatype by introducing a new predicate called the structural order denoted $t < t'$. Intuitively, if t is a well-defined strict subexpression of t', then we may conclude $t < t'$. The rule for introducing the structural order predicate is as

follows:

$$\frac{\begin{array}{l}\Delta, \Gamma \vdash t : t_1 \\ \Delta, \Gamma \vdash t' : t_2 \\ \Delta, \Gamma \vdash \mathbf{val}\ t \\ \Delta, \Gamma \vdash \mathbf{val}\ t' \\ t \sqsubseteq^{+}\ t'\end{array}}{\Delta, \Gamma \vdash \mathbf{ord}\, t\, t' : t < t'}\ \text{PRF\_Ord}$$

This rule is very much like the rule for introducing the termination predicate in that it is defined for values only, however, when mixed with equality it can be very powerful. The judgment $t \sqsubseteq^{+}\ t'$ is the subexpression ordering on datatype constructors. See the appendix for the complete definition.

The structural order predicate is then used in the definition of induction in the logical fragment:

$$\frac{\Delta, \Gamma, x : t_1, u :^{\mathbf{val}} x\,!, f :^{\mathbf{val}} \forall y\ :\ t_2.\forall u\ :\ y < x.[y/x]P \vdash p : P}{\Delta, \Gamma \vdash \mathbf{ind}\, f\ x : t_1, u.p : \forall x\ :\ t_1.\forall u\ :\ x\,!.P}\ \text{PRF\_Ind}$$

As we can see the inductive call must be given a proof that the argument we are inducting over has structurally decreased. This means that we can now do induction on many different types of data, for example, lists, trees, natural numbers, and any other inductively defined data.

Throughout this chapter we have seen only the positive perspective of the design of Sep^3. So now we briefly discuss one major lesson learned from this design. Relaxing the value restrictions was achieved by a strict separation of the logical and programmatic fragments. In addition, the proof language has been separated into several different languages. This strict separation prevents code reuse. That is, there are functions that can be written in both the programmatic fragment and the logical fragment, but these have to be written twice. Thus, through the strict separation we

have gained no value restrictions, but lost code reuse. This is the major drawback of the Sep^3 design.

In this chapter we introduced a new dependently-typed functional programming language called Separation of Proof from Program. Sep^3 is a large advancement over the freedom of speech language. It has several new predicates for disjunction, existential quantification, termination, and structural ordering. In addition Sep^3 contains the freedom of speech property, but is designed so as to prevent the need for the value restrictions we have in the freedom of speech language. Thus, programming in Sep^3 is more enjoyable. Lastly, Sep^3 contains inductive datatypes with pattern matching. Therefore, real-world programming examples can be carried out in Sep^3; in fact, several examples can be found in the paper [74].

CHAPTER 9

DUALIZED LOGIC AND TYPE THEORY

Freedom of Speech and Sep^3 are two very powerful programming languages, but there is one feature they do not have, the ability to reason about infinite data structures in an elegant way. Infinite data can be defined and reasoned about using corecursion and coinduction respectively. Now both of the previous languages contain induction, but only Sep^3 contained inductive data types. This chapter is about the design of a new simply typed theory called Dualized Type Theory (DTT).

DTT is based on a new bi-intuitionistic logic (BINT) called Dualized Intuitionistic Logic (DIL). Bi-intuitionisim is a conservative extension of intuitionistic logic with prefect duality. That is, for every logical connective of the logic its dual connective is also a logical connective of the logic. Due to the rich duality in BINT we believe it shows promise of being a logical foundation for induction and coinduction, because induction is dual to coinduction. Our working hypothesis is that a logical foundation based on intuitionistic duality will allow the semantic duality between induction and coinduction to be expressed in type theory, yielding a solution to the problems with these important features in existing systems. For example, Agda restricts how inductive and coinductive types can be nested (see the discussion in [3]), while Coq supports general mixed inductive and coinductive data, but in doing so, sacrifices type preservation.

One particular difference in the design of DTT from Freedom of Speech and

Sep[3] is that we started from a logic, and then derived a corresponding type theory from it. Thus, we make explicit use of the computational trinity (Chapter 2) by starting on the logical side, and then moving to the type theoretic side.

Classical logic is rich with duality. Using the De Morgan dualities it is straightforward to prove that conjunction is dual to disjunction and negation is self dual. In addition, it is also possible to prove that $\neg A \wedge B$ is dual to implication. In intuitionistic logic these dualities are no longer provable, but in [112] Rauszer gives a conservative extension of the Kripke semantics for intuitionistic logic that not only models conjunction, disjunction, negation, and implication, but also the dual to implication, by introducing a new logical connective. The usual interpretation of implication in a Kripke model is as follows:

$$[\![A \to B]\!]_w = \forall w'.w \leq w' \to [\![A]\!]_{w'} \to [\![B]\!]_{w'}$$

Now Rauszer took the dual of the previous interpretation to obtain the following:

$$[\![A - B]\!]_w = \exists w'.w' \leq w \wedge \neg[\![A]\!]_{w'} \wedge [\![B]\!]_{w'}$$

This is called subtraction or exclusion. Propositional BINT logic is a conservative extension of propositional intuitionistic logic with perfect duality. That is, it contains the logical connectives for disjunction, conjunction, implication, and subtraction, and it is sound and complete with respect to the Rauszer's extended Kripke semantics.

BINT logic is fairly unknown in computer science. Filinski studied a fragment of BINT logic in his investigation into first class continuations in [55]. Crolard introduced a logic and corresponding type theory called subtractive logic, and showed

it can be used to study constructive coroutines in [40, 41]. He initially defined subtractive logic in sequent style with the Dragalin restriction, and then defined the corresponding type theory in natural deduction style by imposing a restriction on Parigot's $\lambda\mu$-calculus in the form of complex dependency tracking. Just as linear logicians have found – for example in [119] – Pinto and Uustalu were able to show that imposing the Dragalin restriction in subtractive logic results in a failure of cut elimination [106]. They recover cut elimination by proposing a new BINT logic called L that lifts the Dragalin restriction by labeling formulas and sequents with nodes and graphs respectively; this labeling corresponds to placing constraints on the sequents where the graphs can be seen as abstract Kripke models. Goré et. al. also proposed a new BINT logic that enjoys cut elimination using nested sequents; however it is currently unclear how to define a type theory with nested sequents [63]. Bilinear logic in its intuitionistic form is a linear version of BINT and has been studied by Lambek in [80, 81]. Biasi and Aschieri propose a term assignment to polarized bi-intuitionistic logic in [24]. One can view the polarities of their logic as an internalization of the polarities of the logic we propose in this article. Bellin has studied BINT similar to that of Biasi and Aschieri from a philosophical perspective in [20, 21, 23], and he defined a linear version of Crolard's subtractive logic for which he was able to construct a categorical model using linear categories in [22].

Throughout the remainder of this chapter we introduce the design of DIL and its corresponding type theory DTT. DIL is a single-sided polarized formulation of Pinto and Uustalu's labeled sequent calculus L. DIL builds on L by removing the

following rules (see Section 9.1 for a complete definition of L):

$$\frac{\Gamma \vdash_{G\cup\{(n,n)\}} \Delta}{\Gamma \vdash_G \Delta}\ \text{refl} \qquad \frac{\begin{array}{l} n_1 G n_2 \\ n_2 G n_3 \\ \Gamma \vdash_{G\cup\{(n_1,n_3)\}} \Delta \end{array}}{\Gamma \vdash_G \Delta}\ \text{trans}$$

$$\frac{\begin{array}{l} nGn' \\ \Gamma, n : T, n' : T \vdash_G \Delta \end{array}}{\Gamma, n : T \vdash_G \Delta}\ \text{monL} \qquad \frac{\begin{array}{l} n'Gn \\ \Gamma \vdash_G n' : T, n : T, \Delta \end{array}}{\Gamma \vdash_G n : T, \Delta}\ \text{monR}$$

We show in Chapter 11 that in the absence of the previous rules DIL still maintains consistency (Section 11.1) and completeness (Section 11.2). Furthermore, DIL is defined using a dualized syntax which reduces the number of inference rules needed to define the logic. Again, DIL is a single-sided sequent calculus with multiple conclusions and thus must provide a means of moving conclusions from left to right. This is done in DIL using cuts on hypotheses. We call these types of cuts "axiom cuts."

Now we consider BINT logic to be the closest extension of intuitionistic logic to classical logic while maintaining constructivity. BINT has two forms of negation, one defined as usual, $\neg A \stackrel{\mathsf{def}}{=} A \to \bot$, and a second defined interms of subtraction, $\sim A \stackrel{\mathsf{def}}{=} \top - A$. The latter we call "non-A". Now in BINT it is possible to prove $A \vee \sim A$ for any A [40]. Furthermore, when the latter is treated as a type in DTT, the inhabitant is a continuation without a canonical form, because the inhabitant contains as a subexpression an axiom cut. Thus, the presence of these continuations prevents the canonicity result for a type theory – like DTT – from holding. Thus, if general cut elimination was a theorem of DIL, then $A \vee \sim A$ would not be provable. So DIL must contain cuts that cannot be eliminated. This implies that DIL does not enjoy general cut elimination, but all cuts other than axiom cuts can be eliminated. Throughout

$$\begin{array}{rrcl} \text{(formulas)} & A, B, C & ::= & \top \mid \bot \mid A \supset B \mid A \prec B \mid A \wedge B \mid A \vee B \\ \text{(graphs)} & G & ::= & \cdot \mid (n, n') \mid G, G' \\ \text{(contexts)} & \Gamma & ::= & \cdot \mid n : A \mid \Gamma, \Gamma' \end{array}$$

Figure 40. Syntax of L.

the sequel we define "cut elimination" as the elimination of all cuts other than axiom cuts, and we call DIL "cut free" with respect to this definition of cut elimination. The latter point is similar to Wadler's dual calculus [141].

9.1 Pinto and Uustalu's L

In this section we briefly introduce Pinto and Uustalu's L from [106]. The syntax for formulas, graphs, and contexts of L are defined in Figure 40, while the inference rules are defined in Figure 41. The formulas include true and false denoted $\top$ and $\bot$ respectively, implication and subtraction denoted $A \supset B$ and $A \prec B$ respectively, and finally, conjunction and disjunction denoted $A \wedge B$ and $A \vee B$ respectively. So we can see that for every logical connective its dual is a logical connective of the logic. This is what we meant by BINT containing perfect intuitionistic duality in the introduction. Sequents have the form $\Gamma \vdash_G n : A, \Delta$, where Γ and Δ are multisets of formulas labeled by a node, G is the abstract Kripke model or sometimes referred to as simply the graph of the sequent, and n is a node in G.

Graphs are treated as sets of edges and we denote $(n_1, n_2) \in G$ by $n_1 G n_2$. Furthermore, we denote the union of two graphs G and G' as $G \cup G'$. Now each formula present in a sequent is labelled with a node in the graph. This labeling is

$$\frac{\Gamma \vdash_{G \cup \{(n,n)\}} \Delta}{\Gamma \vdash_G \Delta}\ \text{refl} \qquad \frac{\begin{array}{l} n_1 G n_2 \\ n_2 G n_3 \\ \Gamma \vdash_{G \cup \{(n_1,n_3)\}} \Delta \end{array}}{\Gamma \vdash_G \Delta}\ \text{trans} \qquad \frac{}{\Gamma, n : T \vdash_G n : T, \Delta}\ \text{hyp}$$

$$\frac{\begin{array}{l} n G n' \\ \Gamma, n : T, n' : T \vdash_G \Delta \end{array}}{\Gamma, n : T \vdash_G \Delta}\ \text{monL} \qquad \frac{\begin{array}{l} n' G n \\ \Gamma \vdash_G n' : T, n : T, \Delta \end{array}}{\Gamma \vdash_G n : T, \Delta}\ \text{monR}$$

$$\frac{\Gamma \vdash_G \Delta}{\Gamma, n : \top \vdash_G \Delta}\ \text{trueL} \qquad \frac{}{\Gamma \vdash_G n : \top, \Delta}\ \text{trueR} \qquad \frac{}{\Gamma, n : \bot \vdash_G \Delta}\ \text{falseL}$$

$$\frac{\Gamma \vdash_G \Delta}{\Gamma \vdash_G n : \bot, \Delta}\ \text{falseR} \qquad \frac{\Gamma, n : T_1, n : T_2 \vdash_G \Delta}{\Gamma, n : T_1 \wedge T_2 \vdash_G \Delta}\ \text{andL} \qquad \frac{\begin{array}{l} \Gamma \vdash_G n : T_1, \Delta \\ \Gamma \vdash_G n : T_2, \Delta \end{array}}{\Gamma \vdash_G n : T_1 \wedge T_2, \Delta}\ \text{andR}$$

$$\frac{\begin{array}{l} \Gamma, n : T_1 \vdash_G \Delta \\ \Gamma, n : T_2 \vdash_G \Delta \end{array}}{\Gamma, n : T_1 \vee T_2 \vdash_G \Delta}\ \text{disjL} \qquad \frac{\Gamma \vdash_G n : T_1, n : T_2, \Delta}{\Gamma \vdash_G n : T_1 \vee T_2, \Delta}\ \text{disjR}$$

$$\frac{\begin{array}{l} n G n' \\ \Gamma \vdash_G n' : T_1, \Delta \\ \Gamma, n' : T_2 \vdash_G \Delta \end{array}}{\Gamma, n : T_1 \supset T_2 \vdash_G \Delta}\ \text{impL} \qquad \frac{\begin{array}{l} n' \notin |G|, |\Gamma|, |\Delta| \\ \Gamma, n' : T_1 \vdash_{G \cup \{(n,n')\}} n' : T_2, \Delta \end{array}}{\Gamma \vdash_G n : T_1 \supset T_2, \Delta}\ \text{impR}$$

$$\frac{\begin{array}{l} n' \notin |G|, |\Gamma|, |\Delta| \\ \Gamma, n' : T_1 \vdash_{G \cup \{(n,n')\}} n' : T_2, \Delta \end{array}}{\Gamma, n' : T_1 \prec T_2 \vdash_G \Delta}\ \text{subL} \qquad \frac{\begin{array}{l} n' G n \\ \Gamma \vdash_G n' : T_1, \Delta \\ \Gamma, n' : T_2 \vdash_G \Delta \end{array}}{\Gamma \vdash_G n : T_1 \prec T_2, \Delta}\ \text{subR}$$

Figure 41. Inference Rules for L.

denoted $n : A$ and should be read as the formula A is true at the node n. We denote the operation of constructing the list of nodes in a graph or context by $|G|$ and $|\Gamma|$ respectively. The reader should note that it is possible for some nodes in the sequent to not appear in the graph. For example, the sequent $n : A \vdash. \ n : A, \cdot$ is a derivable sequent. Now the complete graph can always be recovered if needed by using the graph structural rules refl, trans, monL, and monR.

The labeling on formulas essentially adds constraints to the set of Kripke models. This is evident in the proof of consistency for DIL in Section 11.1; see the definition of validity. Consistency of L is stated in [106] without a detailed proof, but is proven complete with respect to Rauszer's Kripke semantics using a counter model construction. In Section 9.2 we give a translation of the formulas of L into the formulas of DIL, and in Section 11.2 we will give a translation of the rest of L into DIL which will be used to conclude completeness of DIL.

9.2 Dualized Intuitionistic Logic

The syntax for polarities, formulas, and graphs of DIL is defined in Figure 42, where **a** ranges over atomic formulas. The following definition shows that DIL's formulas are simply polarized versions of L's formulas.

Definition 9.2.0.1.

The following defines a translation of formulas of L to formulas of DIL:

$$\begin{array}{lcl} \ulcorner \top \urcorner & = & \langle + \rangle \\ \ulcorner \bot \urcorner & = & \langle - \rangle \end{array} \qquad \begin{array}{lcl} \ulcorner A \wedge B \urcorner & = & \ulcorner A \urcorner \wedge_+ \ulcorner B \urcorner \\ \ulcorner A \vee B \urcorner & = & \ulcorner A \urcorner \wedge_- \ulcorner B \urcorner \end{array} \qquad \begin{array}{lcl} \ulcorner A \supset B \urcorner & = & \ulcorner A \urcorner \rightarrow_+ \ulcorner B \urcorner \\ \ulcorner B \prec A \urcorner & = & \ulcorner A \urcorner \rightarrow_- \ulcorner B \urcorner \end{array}$$

We represent graphs as lists of edges denoted $n_1 \preccurlyeq_p n_2$, where we consider the edge $n_1 \preccurlyeq_+ n_2$ to mean that there is a path from n_1 to n_2, and the edge $n_1 \preccurlyeq_- n_2$ to

$$
\begin{array}{rrcl}
\text{(polarities)} & p & ::= & + \mid - \\
\text{(formulas)} & A, B, C & ::= & \mathbf{a} \mid \langle p \rangle \mid A \to_p B \mid A \wedge_p B \\
\text{(graphs)} & G & ::= & \cdot \mid n \preccurlyeq_p n' \mid G, G' \\
\text{(contexts)} & \Gamma & ::= & \cdot \mid p\, A \mathbin{@} n \mid \Gamma, \Gamma'
\end{array}
$$

Figure 42. Syntax for DIL.

mean that there is a path from n_2 to n_1. Lastly, contexts denoted Γ are represented as lists of formulas. Throughout the sequel we denote the opposite of a polarity p by $\bar{p}$. This is defined by $\bar{+} = -$ and $\bar{-} = +$. The inference rules for DIL are in Figure 43.

The sequent has the form $G; \Gamma \vdash p\, A \mathbin{@} n$ which when p is positive (resp. negative) can be read as the formula A is true (resp. false) at node n in the context Γ with respect to the graph G. The inference rules depend on a reachability judgment that provides a means of proving when a node is reachable from another within some graph G. This judgment is defined in Figure 44. In addition, the imp rule depends on the operations $|G|$ and $|\Gamma|$ which simply compute the list of all the nodes in G and Γ respectively. The condition $n' \notin |G|, |\Gamma|$ in the imp rule is required for consistency.

The most interesting inference rules of DIL are the rules for implication and coimplication from Figure 43. Let us consider these two rules in detail. These rules mimic the definitions of the interpretation of implication and coimplication in a Kripke model. The imp rule states that the formula $p\,(A \to_p B)$ holds at node n if assuming $p\, A \mathbin{@} n'$ for an arbitrary node n' reachable from n, then $p\, B \mathbin{@} n'$ holds. Notice that

$$\frac{G \vdash n \preccurlyeq_p^* n'}{G; \Gamma, p\, A \mathbin{@} n, \Gamma' \vdash p\, A \mathbin{@} n'}\ \text{ax} \qquad \frac{}{G; \Gamma \vdash p\, \langle p \rangle \mathbin{@} n}\ \text{unit}$$

$$\frac{G; \Gamma \vdash p\, A \mathbin{@} n \qquad G; \Gamma \vdash p\, B \mathbin{@} n}{G; \Gamma \vdash p\, (A \wedge_p B) \mathbin{@} n}\ \text{and} \qquad \frac{G; \Gamma \vdash p\, A_d \mathbin{@} n}{G; \Gamma \vdash p\, (A_1 \wedge_{\bar{p}} A_2) \mathbin{@} n}\ \text{andBar}$$

$$\frac{\begin{array}{l} n' \notin |G|, |\Gamma| \\ (G, n \preccurlyeq_p n'); \Gamma, p\, A \mathbin{@} n' \vdash p\, B \mathbin{@} n' \end{array}}{G; \Gamma \vdash p\, (A \to_p B) \mathbin{@} n}\ \text{imp}$$

$$\frac{\begin{array}{l} G \vdash n \preccurlyeq_{\bar{p}}^* n' \\ G; \Gamma \vdash \bar{p}\, A \mathbin{@} n' \qquad G; \Gamma \vdash p\, B \mathbin{@} n' \end{array}}{G; \Gamma \vdash p\, (A \to_{\bar{p}} B) \mathbin{@} n}\ \text{impBar}$$

$$\frac{G; \Gamma, \bar{p}\, A \mathbin{@} n \vdash +\, B \mathbin{@} n' \qquad G; \Gamma, \bar{p}\, A \mathbin{@} n \vdash -\, B \mathbin{@} n'}{G; \Gamma \vdash p\, A \mathbin{@} n}\ \text{cut}$$

Figure 43. Inference Rules for DIL.

when p is positive n' will be a future node, but when p is negative n' will be a past node. Thus, universally quantifying over past and future worlds is modeled here by adding edges to the graph. Now the impBar rule states the formula $p\,(A \to_{\bar{p}} B)$ is derivable if there exists a node n' that is provably reachable from n, $\bar{p}\, A$ is derivable at node n', and $p\, B \mathbin{@} n'$ is derivable at node n'. When p is positive n' will be a past node, but when p is negative n' will be a future node. This is exactly dual to implication. Thus, existence of past and future worlds is modeled by the reachability judgment.

Before moving on to proving consistency and completeness of DIL we first show that the formula $A \wedge_- \sim A$ has a proof in DIL that contains a cut that cannot

$$\frac{}{G, n \preccurlyeq_p n', G' \vdash n \preccurlyeq_p^* n'}\ \text{rel\_ax} \qquad \frac{}{G \vdash n \preccurlyeq_p^* n}\ \text{rel\_refl}$$

$$\frac{G \vdash n \preccurlyeq_p^* n' \qquad G \vdash n' \preccurlyeq_p^* n''}{G \vdash n \preccurlyeq_p^* n''}\ \text{rel\_trans} \qquad \frac{G \vdash n' \preccurlyeq_{\bar{p}}^* n}{G \vdash n \preccurlyeq_p^* n'}\ \text{rel\_flip}$$

Figure 44. Reachability Judgment for DIL.

be eliminated. This also serves as an example of a derivation in DIL. Consider the following where we leave off the reachability derivations for clarity and $\Gamma' \equiv \Gamma, -(A \wedge_- \sim A) @ n, -A @ n$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{G;\Gamma' \vdash -A @ n}\,\text{ax} \quad \dfrac{}{G;\Gamma' \vdash +\langle+\rangle @ n}\,\text{unit}}{G;\Gamma' \vdash + \sim A @ n}\,\text{impBar}}{G;\Gamma' \vdash +(A \wedge_- \sim A) @ n}\,\text{andBar} \quad \dfrac{}{G;\Gamma' \vdash -(A \wedge_- \sim A) @ n}\,\text{ax}}{G;\Gamma, -(A \wedge_- \sim A) @ n \vdash +A @ n}\,\text{cut}}{G;\Gamma, -(A \wedge_- \sim A) @ n \vdash +(A \wedge_- \sim A) @ n}\,\text{andBar}$$

Now using only an axiom cut we may conclude the following derivation:

$$\frac{G;\Gamma, -(A \wedge_- \sim A) @ n \vdash +(A \wedge_- \sim A) @ n \qquad \dfrac{}{G;\Gamma, -(A \wedge_- \sim A) @ n \vdash -(A \wedge_- \sim A) @ n}\,\text{ax}}{G;\Gamma \vdash +(A \wedge_- \sim A) @ n}\,\text{cut}$$

The reader should take notice to the fact that all cuts within the previous two derivations are axiom cuts, where the inner most cut uses the hypothesis of the outer cut. Therefore, neither can be eliminated.

9.3 Dualized Type Theory

In this section we give DIL a term assignment yielding Dualized Type Theory (DTT). First, we introduce DTT, and give several examples illustrating how to program in DTT.

$$
\begin{array}{lrcl}
\text{(indices)} & d & ::= & 1 \mid 2 \\
\text{(polarities)} & p & ::= & + \mid - \\
\text{(types)} & A, B, C & ::= & \langle p \rangle \mid A \to_p B \mid A \wedge_p B \\
\text{(terms)} & t & ::= & x \mid \mathbf{triv} \mid (t, t') \mid \mathbf{in}_d\, t \mid \lambda x.t \mid \langle t, t' \rangle \mid \nu\, x.t \bullet t' \\
\text{(canonical terms)} & c & ::= & x \mid \mathbf{triv} \mid (t, t') \mid \mathbf{in}_d\, t \mid \lambda x.t \mid \langle t, t' \rangle \\
\text{(graphs)} & G & ::= & \cdot \mid n \preccurlyeq_p n' \mid G, G' \\
\text{(contexts)} & \Gamma & ::= & \cdot \mid x : p\, A \mathbin{@} n \mid \Gamma, \Gamma'
\end{array}
$$

Figure 45. Syntax for DTT.

The syntax for DIL is defined in Figure 45. Polarities, types, and graphs are all the same as they were in DIL. Contexts differ only by the addition of labeling each hypothesis with a variable. Terms, denoted t, consist of introduction forms, together with cut terms $\nu\, x.t \bullet t'$[1]. We denote variables as x, y, z, ... The term **triv** is the introduction form for units, (t, t') is the introduction form for pairs, similarly the terms $\mathbf{in}_1\, t$ and $\mathbf{in}_2\, t$ introduce disjunctions, $\lambda x.t$ introduces implication, and $\langle t, t' \rangle$ introduces coimplication. The type-assignment rules are defined in Figure 46, and result from a simple term assignment to the rules for DIL. Finally, the reduction rules for DTT are defined in Figure 47. The reduction rules should be considered rewrite rules that can be applied anywhere within a term. (The congruence rules for this are omitted.)

Programming in DTT is not functional programming as usual, so we now give several illustrative examples. The reader familiar with type theories based on sequent

[1]In classical type theories the symbol μ usually denotes cut, but we have reserved that symbol – indexed by a polarity – to be used with inductive (positive polarity) and coinductive (negative polarity) types in future work.

$$\frac{G \vdash n \preccurlyeq_p^* n'}{G; \Gamma, x : p\, A \mathbin{@} n, \Gamma' \vdash x : p\, A \mathbin{@} n'}\ \text{Ax} \qquad \frac{}{G; \Gamma \vdash \mathbf{triv} : p\, \langle p \rangle \mathbin{@} n}\ \text{Unit}$$

$$\frac{G; \Gamma \vdash t_1 : p\, A \mathbin{@} n \qquad G; \Gamma \vdash t_2 : p\, B \mathbin{@} n}{G; \Gamma \vdash (t_1, t_2) : p\, (A \wedge_p B) \mathbin{@} n}\ \text{And}$$

$$\frac{G; \Gamma \vdash t : p\, A_d \mathbin{@} n}{G; \Gamma \vdash \mathbf{in}_d\, t : p\, (A_1 \wedge_{\bar{p}} A_2) \mathbin{@} n}\ \text{AndBar}$$

$$\frac{\begin{array}{l} n' \notin |G|, |\Gamma| \\ (G, n \preccurlyeq_p n'); \Gamma, x : p\, A \mathbin{@} n' \vdash t : p\, B \mathbin{@} n' \end{array}}{G; \Gamma \vdash \lambda x.t : p\, (A \to_p B) \mathbin{@} n}\ \text{Imp}$$

$$\frac{\begin{array}{l} G \vdash n \preccurlyeq_{\bar{p}}^* n' \\ G; \Gamma \vdash t_1 : \bar{p}\, A \mathbin{@} n' \qquad G; \Gamma \vdash t_2 : p\, B \mathbin{@} n' \end{array}}{G; \Gamma \vdash \langle t_1, t_2 \rangle : p\, (A \to_{\bar{p}} B) \mathbin{@} n}\ \text{ImpBar}$$

$$\frac{\begin{array}{c} G; \Gamma, x : \bar{p}\, A \mathbin{@} n \vdash t_1 : + B \mathbin{@} n' \\ G; \Gamma, x : \bar{p}\, A \mathbin{@} n \vdash t_2 : - B \mathbin{@} n' \end{array}}{G; \Gamma \vdash \nu\, x.t_1 \bullet t_2 : p\, A \mathbin{@} n}\ \text{Cut}$$

Figure 46. Type-Assignment Rules for DTT.

calculi will find the following very familiar. The encodings are very similar to that of Curien and Herbelin's $\bar{\lambda}\mu\tilde{\mu}$-calculus [44]. The locus of computation is the cut term, so naturally, function application is modeled using cuts. Suppose

$$\begin{array}{lll} D_1 & =^{\text{def}} & G; \Gamma \vdash \lambda x.t : + (A \to_+ B) \mathbin{@} n \\ D_2 & =^{\text{def}} & G; \Gamma \vdash t' : + A \mathbin{@} n \\ \Gamma' & =^{\text{def}} & \Gamma, y : - B \mathbin{@} n \end{array}$$

Then we can construct the following typing derivation:

$$\frac{D_1 \qquad \dfrac{D_2 \qquad \dfrac{}{G; \Gamma' \vdash y : - B \mathbin{@} n}\ \text{ax}}{G; \Gamma' \vdash \langle t', y \rangle : - (A \to_+ B) \mathbin{@} n}\ \text{impBar}}{G; \Gamma \vdash \nu\, y.\lambda x.t \bullet \langle t', y \rangle : + B \mathbin{@} n}\ \text{cut}$$

$$\frac{}{\nu\, z.\lambda x.t \bullet \langle t_1, t_2\rangle \rightsquigarrow \nu\, z.[t_1/x]t \bullet t_2}\ \text{RImp}$$

$$\frac{}{\nu\, z.\langle t_1, t_2\rangle \bullet \lambda x.t \rightsquigarrow \nu\, z.t_2 \bullet [t_1/x]t}\ \text{RImpBar} \qquad \frac{}{\nu\, z.(t_1, t_2) \bullet \mathbf{in}_1\, t \rightsquigarrow \nu\, z.t_1 \bullet t}\ \text{RAnd1}$$

$$\frac{}{\nu\, z.(t_1, t_2) \bullet \mathbf{in}_2\, t \rightsquigarrow \nu\, z.t_2 \bullet t}\ \text{RAnd2} \qquad \frac{}{\nu\, z.\mathbf{in}_1\, t \bullet (t_1, t_2) \rightsquigarrow \nu\, z.t \bullet t_1}\ \text{RAndBar1}$$

$$\frac{}{\nu\, z.\mathbf{in}_2\, t \bullet (t_1, t_2) \rightsquigarrow \nu\, z.t \bullet t_2}\ \text{RAndBar2} \qquad \frac{x \notin \mathsf{FV}(t)}{\nu\, x.t \bullet x \rightsquigarrow t}\ \text{RRet}$$

$$\frac{}{\nu\, z.(\nu\, x.t_1 \bullet t_2) \bullet t \rightsquigarrow \nu\, z.[t/x]t_1 \bullet [t/x]t_2}\ \text{RBetaL}$$

$$\frac{}{\nu\, z.c \bullet (\nu\, x.t_1 \bullet t_2) \rightsquigarrow \nu\, z.[c/x]t_1 \bullet [c/x]t_2}\ \text{RBetaR}$$

Figure 47. Reduction Rules for DTT.

Implication was indeed eliminated, yielding the conclusion.

There is some intution one can use while thinking of this style of programming. In [75] Kimura and Tatsuta explain how we can think of positive variables as input ports, and negative variables as output ports. Clearly, these notions are dual. Then a cut of the form $\nu\, z.t \bullet t'$ can be intuitively understood as a device capable of routing information. We think of this term as first running the term t, and then plugging its value into the continuation t'. Thus, negative terms are continuations. Now consider the instance of the previous term $\nu\, z.t \bullet y$ where t is a positive term and y is a negative variable (an output port). This can be intuitively understood as after running t, route its value through the output port y. Now consider the instance $\nu\, z.t \bullet z$. This term can be understood as after running the term t, route its value through the output

part z, but then capture this value as the return value. Thus, the cut term reroutes output ports into the actual return value of the cut.

There is one additional bit of intuition we can use when thinking about programming in DTT. We can think of cuts of the form

$$\nu z.(\lambda x_1 \cdots \lambda x_i.t) \bullet \langle t_1, \langle t_2, \cdots \langle t_i, z\rangle \cdots \rangle$$

as an abstract machine, where $\lambda x_1 \cdots \lambda x_i.t$ is the functional part of the machine, and $\langle t_1, \langle t_2, \cdots \langle t_i, z\rangle \cdots \rangle$ is the stack of inputs the abstract machine will apply the function to ultimately routing the final result of the application through z, but rerouting this into the return value. This intuition is not new, but was first observed by Curien and Herbelin in [44]; see also [43].

Similarly to the eliminator for implication we can define the eliminator for disjunction in the form of the usual case analysis. Suppose $G;\Gamma \vdash t : +\,(A \wedge_{-} B)\,@\,n$, $G;\Gamma, x : +\,A\,@\,n \vdash t_1 : +\,C\,@\,n$, and $G;\Gamma, x : +\,B\,@\,n \vdash t_2 : +\,C\,@\,n$ are all admissible. Then we can derive the usual eliminator for disjunction. Define

$$\mathbf{case}\ t\ \mathbf{of}\ x.t_1, x.t_2 \stackrel{\mathsf{def}}{=} \nu\, z_0.(\nu\, z_1.(\nu\, z_2.t \bullet (z_1, z_2)) \bullet (\nu\, x.t_2 \bullet z_0)) \bullet (\nu\, x.t_1 \bullet z_0).$$

Then we have the following result.

Lemma 9.3.0.1. *The following rule is admissible:*

$$\frac{\begin{array}{ll} G;\Gamma, x : p\ A\ @\ n \vdash t_1 : p\ C\ @\ n & \\ G;\Gamma, x : p\ B\ @\ n \vdash t_2 : p\ C\ @\ n & \qquad G;\Gamma \vdash t : p\,(A \wedge_{\bar{p}} B)\,@\,n \end{array}}{G;\Gamma \vdash \mathbf{case}\ t\ \mathbf{of}\ x.t_1, x.t_2 : p\ C\ @\ n}\ \text{case}$$

Proof. Due to the size of the derivation in question we give several derivations which compose together to form the typing derivation of $G;\Gamma \vdash \mathbf{case}\ t\ \mathbf{of}\ x.t_1, x.t_2 : p\ C @ n$.

The typing derivation begins using cut as follows:

$$\dfrac{\begin{array}{cc} D_0 & D_1 \end{array}}{G; \Gamma \vdash \nu\, z_0.(\nu\, z_1.(\nu\, z_2.t \bullet (z_1, z_2)) \bullet (\nu\, x.t_2 \bullet z_0)) \bullet (\nu\, x.t_1 \bullet z_0) : + C \mathbin{@} n}\ \text{cut}$$

Then the remainder of the derivation depends on the following sub-derivations:

D_0 :

$$\dfrac{\begin{array}{cc} D_3 & D_4 \end{array}}{G; \Gamma, z_0 : - C \mathbin{@} n \vdash \nu\, z_1.(\nu\, z_2.t \bullet (z_1, z_2)) \bullet (\nu\, x.t_2 \bullet z_0) : + A \mathbin{@} n}\ \text{cut}$$

D_1 :

$$\dfrac{\begin{array}{cc} D_2 & \dfrac{}{G; \Gamma, z_0 : - C \mathbin{@} n, x : + A \mathbin{@} n \vdash z_0 : - C \mathbin{@} n}\ \text{ax} \end{array}}{G; \Gamma, z_0 : - C \mathbin{@} n \vdash \nu\, x.t_1 \bullet z_0 : - A \mathbin{@} n}\ \text{cut}$$

D_2 :

$$\dfrac{G; \Gamma, x : + A \mathbin{@} n \vdash t_1 : + C \mathbin{@} n}{G; \Gamma, z_0 : - C \mathbin{@} n, x : + A \mathbin{@} n \vdash t_1 : + C \mathbin{@} n}\ \text{Weakening}$$

D_4 :

$$\dfrac{\begin{array}{cc} D_5 & G; \Gamma, z_0 : - C \mathbin{@} n, z_1 : - A \mathbin{@} n, x : + B \mathbin{@} n \vdash z_0 : - C \mathbin{@} n \end{array}}{G; \Gamma, z_0 : - C \mathbin{@} n, z_1 : - A \mathbin{@} n \vdash \nu\, x.t_2 \bullet z_0 : - B \mathbin{@} n}\ \text{cut}$$

D_3 :

$$\dfrac{\begin{array}{cc} D_6 & D_7 \end{array}}{G; \Gamma, z_0 : - C \mathbin{@} n, z_1 : - A \mathbin{@} n \vdash \nu\, z_2.t \bullet (z_1, z_2) : + B \mathbin{@} n}\ \text{cut}$$

D_5 :

$$\dfrac{G; \Gamma, x : + B \mathbin{@} n \vdash t_2 : + C \mathbin{@} n}{G; \Gamma, z_0 : - C \mathbin{@} n, z_1 : - A \mathbin{@} n, x : + B \mathbin{@} n \vdash t_2 : + C \mathbin{@} n}\ \text{Weakening}$$

D_6 :

$$\dfrac{G; \Gamma \vdash t : + (A \wedge_{-} B) \mathbin{@} n}{G; \Gamma, z_0 : - C \mathbin{@} n, z_1 : - A \mathbin{@} n, z_2 : - B \mathbin{@} n \vdash t : + (A \wedge_{-} B) \mathbin{@} n}\ \text{Weakening}$$

D_7 :

$$\dfrac{\begin{array}{cc} D_8 & D_9 \end{array}}{G; \Gamma, z_0 : - C \mathbin{@} n, z_1 : - A \mathbin{@} n, z_2 : - B \mathbin{@} n \vdash (z_1, z_2) : - (A \wedge_{-} B) \mathbin{@} n}\ \text{and}$$

D_8 :

$$\dfrac{}{G; \Gamma, z_0 : - C \mathbin{@} n, z_1 : - A \mathbin{@} n, z_2 : - B \mathbin{@} n \vdash z_1 : - A \mathbin{@} n}\ \text{ax}$$

D_9 :

$$\dfrac{}{G; \Gamma, z_0 : - C \mathbin{@} n, z_1 : - A \mathbin{@} n, z_2 : - B \mathbin{@} n \vdash z_2 : - B \mathbin{@} n}\ \text{ax}$$

□

Now consider the term $\nu\, x.\mathbf{in}_1\, (\nu\, y.\mathbf{in}_2\, \langle y, \mathbf{triv} \rangle \bullet x) \bullet x$. This term is the inhabitant of the type $A \wedge_{-} \sim A$, and its typing derivation follows from the derivation given in Section 9.2. We can see by looking at the syntax that the cuts involved are indeed on the axiom x, thus this term has no canonical form. In [41] Crolard shows that inhabitants such as these amount to a constructive coroutine. That is, it is a

restricted form of a continuation.

We now consider several example reductions in DTT. In the following examples we underline non-top-level redexes. The first example simply α-converts the function $\lambda x.x$ into $\lambda z.z$ as follows:

$$\begin{array}{lll} \lambda z.\underline{\nu\, y.\lambda x.x \bullet \langle z, y\rangle} & \overset{(\text{RImp})}{\rightsquigarrow} & \lambda z.\underline{\nu\, y.z \bullet y} \\ & \overset{(\text{RRet})}{\rightsquigarrow} & \lambda z.z \end{array}$$

A more involved example is the application of the function $\lambda x.(\lambda y.y)$ to the arguments **triv** and **triv**.

$$\begin{array}{lll} \nu\, z.\lambda x.(\lambda y.y) \bullet \langle \mathbf{triv}, \langle \mathbf{triv}, z\rangle\rangle & \overset{(\text{RImp})}{\rightsquigarrow} & \nu\, z.\lambda y.y \bullet \langle \mathbf{triv}, z\rangle \\ & \overset{(\text{RImp})}{\rightsquigarrow} & \nu\, z.\mathbf{triv} \bullet z \\ & \overset{(\text{RRet})}{\rightsquigarrow} & \mathbf{triv} \end{array}$$

PART C

BASIC SYNTACTIC ANALYSIS

CHAPTER 10

FREEDOM OF SPEECH

The following chapters conduct basic meta-theoretic analysis of the freedom of speech language, and dualized logic and its corresponding type theory dualized type theory. We consider these analysis to be basic because they are straightforward applications of existing techniques, and consist of the most basic properties that should hold for any programming language.

At least a basic meta-theoretic analysis of a programming language is very important. Every programming language should at least have been proven type safe, and if the language contains a logical fragment, then it should be proven consistent. These give strong guarantees to the programmer. Type safety ensures that if a program has a type, then it can either compute something, or is the answer itself. Furthermore, it ensures that if a program has a type, then after running it, the result has the same type. Lastly, consistency – as we have mentioned before – ensures that the objects of the language we call proofs really are proof. These sound like common-sense properties, and they are, but many programming language implementors fail to establish them. A programming language with such guarantees higers one confidence in the correctness and consistency of the language. Therefore, a basic meta-theoretic analysis is necessary to be able to trust the programs we write, and the verification of programs carried out within the languages themselves.

In Chapter 7 we defined the Freedom of Speech language in its entirety. We

now introduce its analysis. The main results consist of logical consistency of the logical fragment and type preservation. The former is shown by proving weak normalization of the logical fragment – the reader may wish to recall the definition of weak normalization; see Definition 6.0.0.3 of Chapter 6. The remainder of this chapter proceeds by first introducing some basic lemmas to which the main results will depend. Then we prove type preservation, and following this we will prove weak normalization.

10.1 Basic Results

The main results depend on several auxiliary results, and we present each of them in this section. The first two basic results are weakening and substitution for typing. The latter states that if we know a term e is well typed in an environment with a free variable x, and given an expression a with the same type as the free variable, then $[a/x]e$ has the same type as e. This is an important result for type preservation, because we want to ensure that for any $(\lambda x\,.\,e)\,v$ that has type e', its contractum $[v/x]e$ has type e'.

Lemma 10.1.0.1. *[Weakening] If* $\Gamma \vdash^{\theta} e : e'$*, then* $\Gamma, \Gamma' \vdash^{\theta} e : e'$.

Proof. This holds by straightforward induction on the form of the assumed typing derivation. □

Lemma 10.1.0.2. *[Substitution for Typing] If* $\Gamma, x :^{\theta} e_1, \Gamma' \vdash^{\theta'} e : e_2$ *and* $\Gamma \vdash^{\theta} v : e_1$*, then* $\Gamma, [v/x]\Gamma' \vdash^{\theta'} [v/x]e : [v/x]e_2$.

Proof. This is a proof by induction on the form of the assumed typing derivation. We

only give the non-trivial cases. All other cases are either similar to the cases given here or are trivial.

Case.

$$\frac{\begin{array}{l}\Gamma, x :^{\theta} e_1, \Gamma' \vdash^{\theta''} e_1' : \mathsf{Type} \\ \Gamma, x :^{\theta} e_1, \Gamma', y :^{\theta''} e_1' \vdash^{\theta'} e_2' : \mathsf{Type}\end{array}}{\Gamma, x :^{\theta} e_1, \Gamma' \vdash^{\theta'} (y :^{\theta''} e_1')^{\epsilon} \to e_2' : \mathsf{Type}}\ \text{K\_Pi}$$

By the induction hypothesis, $\Gamma, [v/x]\Gamma' \vdash^{\theta''} [v/x]e_1' : \mathsf{Type}$ and $\Gamma, [v/x]\Gamma', y :^{\theta''} [v/x]e_1' \vdash^{\theta'} [v/x]e_2' : \mathsf{Type}$. Now we can apply K_Pi to obtain, $\Gamma, [v/x]\Gamma' \vdash^{\theta'} (y :^{\theta''} [v/x]e_1')^{\epsilon} \to [v/x]e_2' : \mathsf{Type}$, which is equivalent to $\Gamma, [v/x]\Gamma' \vdash^{\theta'} [v/x]((y :^{\theta''} e_1')^{\epsilon} \to e_2') : \mathsf{Type}$.

Case.

$$\frac{\begin{array}{l}\Gamma, x :^{\theta} e_1, \Gamma' \vdash^{\theta_1} e' : e_1' \\ \Gamma, x :^{\theta} e_1, \Gamma' \vdash^{\theta_2} e'' : e_2'\end{array}}{\Gamma, x :^{\theta} e_1, \Gamma' \vdash^{L} e' = e'' : \mathsf{Type}}\ \text{K\_Eq}$$

By the induction hypothesis we know $\Gamma, [v/x]\Gamma' \vdash^{\theta_1} [v/x]e' : [v/x]e_1'$ and $\Gamma, [v/x]\Gamma' \vdash^{\theta_2} [v/x]e'' : [v/x]e_2'$. We can now apply K_Eq to obtain $\Gamma, [v/x]\Gamma' \vdash^{L} [v/x]e' = [v/x]e'' : \mathsf{Type}$, which is equivalent to $\Gamma, [v/x]\Gamma' \vdash^{L} [v/x](e' = e'') : \mathsf{Type}$.

Case.

$$\frac{\Gamma, x :^{\theta} e_1, \Gamma' \vdash^{\theta'} e_1' : \mathsf{Type} \quad y :^{\theta'} e_1' \in (\Gamma, x :^{\theta} e_1, \Gamma')}{\Gamma, x :^{\theta} e_1, \Gamma' \vdash^{\theta'} y :^{\theta'} e_1'} \text{ Var}$$

If x is distinct from y, then this case follows by first applying the induction hypothesis to $\Gamma, x :^{\theta} e_1, \Gamma' \vdash^{\theta'} e_1' : \mathsf{Type}$ and then reapplying the Var rule. Now suppose $x \equiv y$. Then the previous typing assumption is equivalent to the following:

$$\frac{\Gamma, x :^{\theta} e_1, \Gamma' \vdash^{\theta} e_1 : \mathsf{Type} \quad x :^{\theta} e_1 \in (\Gamma, x :^{\theta} e_1, \Gamma')}{\Gamma, x :^{\theta} e_1, \Gamma' \vdash^{\theta} x :^{\theta} e_1} \text{ Var}$$

It suffices to show that $\Gamma, [v/x]\Gamma' \vdash^{\theta} v : e_1$ is derivable. We know by assumption that $\Gamma \vdash^{\theta} v : e_1$ is derivable, but by weakening (Lemma 10.1.0.1) we know $\Gamma, [v/x]\Gamma' \vdash^{\theta} v : e_1$.

The remainder of the cases follow similarly to the cases presented above.

□

The third result is regularity, which ensures that if an expression has a type, then its type is well typed. Recall that in the logical fragment $\mathsf{Type} : \mathsf{Type}$ does not hold, thus if e is joinable with Type, then $\Gamma \vdash^{L} e' : e$ does not imply that $\Gamma \vdash^{L} e : \mathsf{Type}$. However, this does hold in the programmatic fragment. Therefore, regularity has the following statement and proof.

Lemma 10.1.0.3. *[Regularity]*

i. If $\Gamma \vdash^{L} e' : e$ *and it is not the case that* $e \equiv \mathsf{Type}$*, then* $\Gamma \vdash^{L} e : \mathsf{Type}$.

ii. If $\Gamma \vdash^{P} e' : e$ *then* $\Gamma \vdash^{P} e : \mathsf{Type}$.

Proof. This is a proof by induction on the form of the assumed typing derivation. The most interesting cases are presented below; all other cases are either similar to the cases given here or are trivial. If it is not the case that $e \equiv \mathsf{Type}$, then both proofs can be given simultaneously. In fact, the proof of part ii when $e \equiv \mathsf{Type}$ holds is trivial, thus we do not present it here.

Case.

$$\frac{\begin{array}{l}\Gamma \vdash^{\theta} e_2 : \mathsf{Type} \\ \Gamma, x :^{\theta} e_2 \vdash^{\theta'} e_1 : e_3\end{array}}{\Gamma \vdash^{\theta'} \lambda x \,.\, e_1 : (x :^{\theta} e_2)^{+} \to e_3}\ \text{Lam}$$

By assumption we know $\Gamma \vdash^{\theta} e_2 : \mathsf{Type}$ and by the induction hypothesis, $\Gamma, x :^{\theta} e_2 \vdash^{\theta'} e_3 : \mathsf{Type}$. Finally by applying K_Pi, $\Gamma \vdash^{\theta'} (x :^{\theta} e_2)^{+} \to e_3 : \mathsf{Type}$.

Case.

$$\frac{\begin{array}{l}\Gamma, x :^{L} \mathbb{N} \vdash^{L} (y :^{L} \mathbb{N})^{+} \to (p :^{L} x = \mathsf{S}\, y)^{-} \to [y/x]e_2 : \mathsf{Type} \\ \Gamma, x :^{L} \mathbb{N}, f :^{L} (y :^{L} \mathbb{N})^{+} \to (p :^{L} x = \mathsf{S}\, y)^{-} \to [y/x]e_2 \vdash^{L} v : e_2 \\ f, p \notin \mathsf{FV}\,(e_2)\end{array}}{\Gamma \vdash^{L} \mathsf{rec}\, f\, x\, v : (x :^{L} \mathbb{N})^{+} \to e_2}\ \text{RecNat}$$

By the induction hypothesis, $\Gamma, x :^{L} \mathbb{N}, f :^{L} (y :^{L} \mathbb{N})^{+} \to (p :^{L} x = S\ y)^{-} \to [y/x]e_2 \vdash^{L} e_2 : \mathsf{Type}$. Since f is not free in e_2 we know $\Gamma, x :^{L} \mathbb{N} \vdash^{L} e_2 : \mathsf{Type}$. Finally, by K_Pi using $\Gamma, x :^{L} \mathbb{N} \vdash^{L} e_2 : \mathsf{Type}$ and K_Nat, $\Gamma \vdash^{L} (x :^{L} \mathbb{N})^{+} \to e_2 : \mathsf{Type}$.

Case.

$$\frac{\Gamma \vdash^{\theta'} e_1 : \mathsf{Type} \quad \Gamma, f :^{\theta} (x :^{\theta'} e_1)^{+} \to e_2, x :^{\theta'} e_1 \vdash^{\theta} e_3 : e_2}{\Gamma \vdash^{P} \mathsf{rec}\, f\ x\ e_3 : (x :^{\theta'} e_1)^{+} \to e_2}\ \text{Rec}$$

By assumption $\Gamma \vdash^{\theta'} e_1 : \mathsf{Type}$ and by the induction hypothesis $\Gamma, f :^{\theta} (x :^{\theta'} e_1)^{+} \to e_2, x :^{\theta'} e_1 \vdash^{\theta} e_2 : \mathsf{Type}$. Now by K_Pi, $\Gamma, f :^{\theta} (x :^{\theta'} e_1)^{+} \to e_2 \vdash^{\theta} (x :^{\theta'} e_1)^{+} \to e_2 : \mathsf{Type}$. Clearly, f is not free in $(x :^{\theta'} e_1)^{+} \to e_2$, thus, $\Gamma \vdash^{\theta} (x :^{\theta'} e_1)^{+} \to e_2 : \mathsf{Type}$.

□

The next result is an inversion principle, and this is needed in the prove of type preservation. This principle states that for certain term constructors if we know the conclusion of their typing rule is derivable, then we know the premises are derivable as well. Inversion in general is very hard to prove for advanced type systems like Freedom of Speech, so we only prove the inversion principles that are actually needed.

Lemma 10.1.0.4. *[Inversion]*

i. If $\Gamma \vdash^{\theta'} \lambda x.e : (x :^{\theta} e_1)^{+} \to e_2$, *then* $\Gamma \vdash^{L} p : (x :^{\theta} e_1)^{+} \to e_2 = (x :^{\theta} a)^{+} \to b$ *and* $\Gamma, x :^{\theta} a \vdash^{\theta'} e : b$.

ii. If $\Gamma \vdash^{L} \mathsf{rec}\, f\ x\ v : (x :^{L} \mathbb{N})^{+} \to e$, *then* $\Gamma \vdash^{L} p : (x :^{L} \mathbb{N})^{+} \to e = (x :^{L} a)^{+} \to b$ *and* $\Gamma, x :^{L} a, f :^{L} (y :^{L} a)^{+} \to (p :^{L} x = \mathsf{S}\, y)^{-} \to [y/x]b \vdash^{L} v : b$.

ii. If $\Gamma \vdash^{P} \mathsf{rec}\, f\ x\ e : (x :^{\theta'} e_1)^{+} \to e_2$, *then* $\Gamma \vdash^{L} p : (x :^{\theta'} e_1)^{+} \to e_2 = (x :^{\theta'} a)^{+} \to b$ *and* $\Gamma, x :^{\theta'} a, f :^{\theta} (x :^{\theta'} a)^{+} \to b \vdash^{\theta} e : b$.

Proof. We proceed by induction on the form of the assumed typing derivation. In

each part there are exactly two cases. The first is when the assumed typing derivation ends with the introduction typing rule for the respective term constructor, or when the derivation ends with the conversion rule. The former is trivial, so we only give the cases for the latter.

Case.

$$\frac{\Gamma \vdash^{\theta'} e' : [e_1'/y]e_2' \quad \Gamma \vdash^{L} e'' : e_1' = e_1''}{\Gamma \vdash^{\theta'} e' : [e_1''/y]e_2'} \text{ Conv}$$

i. Here $e' \equiv \lambda x.e$ and $[e_1''/y]e_2' \equiv (x :^{\theta} e_1)^{+} \to e_2$. Now we have two cases to consider, either e_2' is y and e_1' is a dependent product, or e_2' is a dependent product. If the former is true, then we know e_1'' is a dependent product, and this part follows by first applying the induction hypothesis to $\Gamma \vdash^{\theta'} e' : [e_1'/y]e_2'$, followed by applying the Conv typing rule.

   Suppose the e_2' is a dependent product. Then $[e_1''/y]e_2' \equiv (x :^{\theta} [e_1''/y]r)^{+} \to [e_1''/y]s$ for some expressions r and s. Clearly, $[e_1'/y]e_2' \equiv (x :^{\theta} [e_1'/y]r)^{+} \to [e_1'/y]s$. By the induction hypothesis, $\Gamma \vdash^{\theta'} e' : [e_1'/y]e_2'$ implies $\Gamma, x :^{\theta} a \vdash^{\theta'} e : b$ and $\Gamma \vdash^{L} p : ((x :^{\theta} [e_1'/y]r)^{+} \to [e_1'/y]s) = ((x :^{\theta} a)^{+} \to b)$. Finally, since $\Gamma \vdash^{L} e'' : e_1' = e_1''$ we can apply Conv to obtain $\Gamma \vdash^{\theta'} p : ((x :^{\theta} [e_1''/y]r)^{+} \to [e_1''/y]s) = ((x :^{\theta} a)^{+} \to b)$.

ii. Here $e' \equiv \mathsf{rec}\, f\, x\, v$ and $[e_1''/y]e_2' \equiv (x :^{L} Nat)^{+} \to e$. Just as we saw in the previous part we have two cases to consider, either e_2' is y and e_1'

is a dependent product, or e_2' is a dependent product. If the former is true, then we know e_1'' is a dependent product, and this part follows by first applying the induction hypothesis to $\Gamma \vdash^{\theta'} e' : [e_1'/y]e_2'$, followed by applying the Conv typing rule.

Suppose the e_2' is a dependent product. Then $[e_1''/y]e_2' \equiv (x :^L [e_1''/y]r)^+ \to [e_1''/y]s$ for some expressions r and s. Clearly, $[e_1'/y]e_2' \equiv (x :^L [e_1'/y]r)^+ \to [e_1'/y]s$. By the induction hypothesis, $\Gamma \vdash^{\theta'} e' : [e_1'/y]e_2'$ implies $\Gamma, x :^{\theta} a, f :^L (y :^L a)^+ \to (p :^L x = \mathsf{S}\, y)^- \to [y/x]b \vdash^{\theta'} e : b$ and $\Gamma \vdash^L p : ((x :^{\theta} [e_1'/y]r)^+ \to [e_1'/y]s) = ((x :^{\theta} a)^+ \to b)$. Finally, we know $\Gamma \vdash^L e'' : e_1' = e_1''$ by assumption so we can apply Conv to obtain $\Gamma \vdash^L p : ((x :^{\theta} [e_1''/y]r)^+ \to [e_1''/y]s) = ((x :^{\theta} a)^+ \to b)$.

iii. This part is similar to the previous part.

□

10.2 Type Preservation

We are now in a position to prove type preservation. The proof is straightforward, and holds by induction on the typing derivation. Type preservation is definately a result that should be proven for any type theory or programming language being designed, implemented, or studied. This result ensures that reduction does not wonder outside the bounds of the type system. If type preservation did not hold, then one could start with a proof e of some type, and then use reduction to obtain that e is a proof of some other type, or that e is not a proof at all, or if one starts with the application of some program to a series of arguments that is supposed to return a

natural number, but then after reduction ends up with some other type, would make reasoning about the correctness of the original program impossible!

Unusually so, the proof of logical consistency of freedom of speech actually depends on type preservation. It is usually the case that they are important, but distinct results. In the next section we will define an interpretation of types to prove weak normalization. Due to the collapsed nature of the freedom of speech language these interpretations depend heavily on the typing relation. It is this dependency that results in the proofs of the critical properties of the interpretation of types to depend on type preservation. We conclude this section with the statement and proof of type preservation.

Theorem 10.2.0.1. *[Preservation] If* $\Gamma \vdash^{\theta} a : c$ *and* $a \rightsquigarrow b$ *then* $\Gamma \vdash^{\theta} b : c$.

Proof. We proceed by induction on the assumed typing derivation and only consider non-vacuous cases. We will implicitly assume $a \rightsquigarrow b$ in each case.

Case.

$$\frac{\begin{array}{l}\Gamma \vdash^{\theta'} [v/x]e_2 : \mathsf{Type} \\ \Gamma \vdash^{\theta'} e : (x :^{\theta} e_1)^{+} \to e_2 \\ \Gamma \vdash^{\theta} v : e_1\end{array}}{\Gamma \vdash^{\theta'} e\ v : [v/x]e_2}\ \text{AppPiTerm}$$

We have three cases to consider either $e \rightsquigarrow e'$, $e \equiv \lambda x.e'$ and $e\ v \rightsquigarrow [v/x]e'$, or $e \equiv \mathsf{rec}\, x\, f\, v'$ and $e\, v \rightsquigarrow [\mathsf{rec}\, f\, x\, v'/f][v/x]v'$. Consider the former, by the induction hypothesis, if $e \rightsquigarrow e'$ then $\Gamma \vdash^{L} e' : (x :^{\theta} e_1)^{+} \to e_2$. Now by applying the same rule, $\Gamma \vdash^{L} e'\ v : [v/x]e_2$.

Now suppose the second case, by assumption we know $\Gamma \vdash^{\theta} v : e_1$ and $\Gamma \vdash^{L} \lambda x.e' : (x :^{\theta} e_1)^{+} \to e_2$. By the Inversion Lemma (Lemma 10.1.0.4), $\Gamma, x :^{\theta} e_1' \vdash^{L} e' : e_2'$, and $\Gamma \vdash^{L} p : (x :^{\theta} e_1)^{\varepsilon} \to e_2 = (x :^{\theta} e_1')^{\varepsilon} \to e_2'$. Now we may conclude $\Gamma, x :^{\theta} a \vdash^{L} e' : e_2$ and $\Gamma \vdash^{\theta} v : a$ using the assumption $\Gamma \vdash^{\theta} v : e_1$, the injection rules InjDom and InjRan, and Conv. Thus, by substitution for typing (Lemma 10.1.0.2), $\Gamma \vdash^{L} [v/x]e' : [v/x]e_2$.

The third case follows similarly to the previous case. First, apply inversion (Lemma 10.1.0.4) to the typing assumption for e followed by conversion using the injection rules and the conversion rules, and then finally apply substitution for typing (Lemma 10.1.0.2).

Case.

$$\frac{\begin{array}{l}\Gamma \vdash^{\theta'} [v/x]e_2 : \mathsf{Type} \\ \Gamma \vdash^{\theta'} e : (x :^{\theta} e_1)^{-} \to e_2 \\ \Gamma \vdash^{\theta} v : e_1\end{array}}{\Gamma \vdash^{\theta'} e : [v/x]e_2}\ \text{AppAllTerm}$$

By the induction hypothesis, if $e \rightsquigarrow e'$ then $\Gamma \vdash^{L} e' : (x :^{\theta} e_1)^{-} \to e_2$. Thus, by applying AppAllTerm, $\Gamma \vdash^{L} e' : [v/x]e_2$.

Case.

$$\frac{\begin{array}{l}\Gamma \vdash^{\theta} [e_1'/x]e_2 : \mathsf{Type} \\ \Gamma \vdash^{\theta} e : [e_1/x]e_2 \\ \Gamma \vdash^{L} e' : e_1 = e_1'\end{array}}{\Gamma \vdash^{\theta} e : [e_1'/x]e_2}\ \text{Conv}$$

By the induction hypothesis, if $e \rightsquigarrow e''$ then $\Gamma \vdash^{\theta} e'' : [e_1/x]e_2$. By applying Conv, $\Gamma \vdash^{\theta} e'' : [e_1'/x]e_2$.

Case.

$$\frac{\Gamma \vdash^{L} e : e_1}{\Gamma \vdash^{P} e : e_1} \text{ Coerce}$$

By the induction hypothesis, if $e \rightsquigarrow e'$ then $\Gamma \vdash^{L} e' : e_1$ and by applying Coerce, $\Gamma \vdash^{P} e' : e_1$.

Case.

$$\frac{\Gamma \vdash^{\theta'} e_1 : \mathsf{Type} \quad \Gamma, f :^{\theta} (x :^{\theta'} e_1)^{+} \to e_2, x :^{\theta'} e_1 \vdash^{\theta} e : e_2}{\Gamma \vdash^{P} \mathsf{rec}\, f\ x\ e : (x :^{\theta'} e_1)^{+} \to e_2} \text{ Rec}$$

By the induction hypothesis, if $e \rightsquigarrow e'$ then $\Gamma, f :^{\theta} (x :^{\theta'} e_1)^{+} \to e_2, x :^{\theta'} e_1 \vdash^{\theta} e' : e_2$ and by applying Rec, $\Gamma \vdash^{P} rec\ f\ x\ e' : (x :^{\theta'} e_1)^{+} \to e_2$.

□

10.3 Logical consistency

One recurring statement throughout this thesis is that if we are to consider a type theory or programming language, or even a fragment of one as a logic, then that logic must be proven consistent. We have to believe what it is telling us is true! If consistency has not been shown, then neither the designer nor the programmer has the right to consider any part of the language as containing proofs or formulas. We have claimed that there is a logical fragment of freedom of speech, thus we must show that this fragment is consistent. The proof of logical consistency is the topic of this section.

We prove logical consistency using reducibility candidates first proposed by Girard. For an overview of the reducibility method see Section 6.3. We begin with the definition of the interpretation of types.

Definition 10.3.0.1.

Let $J = \{e \mid e \downarrow join \vee e \downarrow injdom \vee e \downarrow injran\}$ *be the set of all the proofs of equations and* V *be the set of terms such that* $t \rightsquigarrow^! v$*, where* v *is a value. We define the interpretation of types as follows:*

$e \in [\![\mathbb{N}]\!]^L_\rho$ *if and only if*
- $\cdot \vdash^L e : \mathbb{N}$
- $e \in N = \{e' \mid e' \rightsquigarrow^! \mathsf{S}^n\ \mathsf{Z}$ *where* $n \in \mathbb{N}\}$

$e \in [\![(x :^\theta e_1)^+ \to e_2]\!]^L_\rho$ *if and only if*
- $\cdot \vdash^L e : (x :^\theta \rho\, e_1)^+ \to \rho\, e_2$
- $\cdot \vdash^\theta \rho\,((x :^\theta e_1)^+ \to e_2) : \mathsf{Type}$
- $e \in V$
- $\forall e' \in [\![e_1]\!]^\theta_\rho . e\ e' \in [\![e_2]\!]^L_{\rho[x \mapsto e']}$

$e \in [\![(x :^\theta e_1)^- \to e_2]\!]^L_\rho$ *if and only if*
- $\cdot \vdash^L e : (x :^\theta \rho\, e_1)^- \to \rho\, e_2$
- $\cdot \vdash^\theta \rho\,((x :^\theta e_1)^- \to e_2) : \mathsf{Type}$
- $e \in V$
- $\forall e' \in [\![e_1]\!]^\theta_\rho . e \in [\![e_2]\!]^L_{\rho[x \mapsto e']}$

$e \in [\![e_1 = e_2]\!]^L_\rho$ *if and only if*
- $\cdot \vdash^L e : \rho\ e_1 = \rho\ e_2$
- $\cdot \vdash^\theta \rho\ (e_1 = e_2) : \mathsf{Type}$
- $e \in J$
- $\rho\ e_1 \downarrow \rho\ e_2$

$e \in [\![e']\!]^P_\rho$ *if and only if*
- $\cdot \vdash^P e : \rho\ e'$
- $\cdot \vdash^\theta \rho\ e' : \mathsf{Type}$
- $e \in V$

The interpretation of types consists of two parts, a deep interpretation and a shallow one. We give a deep interpretation of the logical fragment where we characterize the termination behavior of the programs, while we give a shallow interpretation of the programs of the programmatic fragment. The only property we can guarantee of programmatic programs is that the terminating programs are well typed. Note that the interpretation of types depends heavily on the typing judgment, this is the dependency mentioned in the previous section on type preservation. Thus, the proofs

$$\frac{}{\emptyset \in [\![\cdot]\!]} \qquad \frac{\Gamma \vdash^L e' : \mathsf{Type} \qquad \rho \in [\![\Gamma]\!] \qquad e \in [\![e']\!]^L_\rho}{\rho \cup \{(x, e)\} \in [\![\Gamma, x :^L e']\!]}$$

$$\frac{\Gamma \vdash^P e' : \mathsf{Type} \qquad \rho \in [\![\Gamma]\!] \qquad v \in [\![e']\!]^P_\rho}{\rho \cup \{(x, v)\} \in [\![\Gamma, x :^P e']\!]}$$

Figure 48. Well-formed substitutions

of the critical properties, especially CR-Pres, depend on type preservation. Finally, the interpretation of types depends on the notion of well-formed substitutions which are defined in Figure 48.

Before we can move on we must justify why the interpretation of types covers all cases, especially, for the logical fragment. We must show that there is no type-level computation, and that the $\mathsf{Type} : \mathsf{Type}$ axiom is purely programmatic. The following two lemmata establish this fact:

Lemma 10.3.0.2. *It is not the case that* $\cdot \vdash^L \mathsf{Type} : \mathsf{Type}$.

Proof. Suppose it is the case that $\cdot \vdash^L \mathsf{Type} : \mathsf{Type}$ is derivable. Then it must be the case that its derivation ends with the one of AppAllTerm or Conv, but both of which would have $\cdot \vdash^L \mathsf{Type} : \mathsf{Type}$ as a premise. No other rules are applicabile. Thus, $\cdot \vdash^L \mathsf{Type} : \mathsf{Type}$ is a non-terminating derivation, a contradiction. □

Lemma 10.3.0.3. *[Characterization of Logical Types] If* $\cdot \vdash^L e : \mathsf{Type}$*, then* $e \equiv \mathbb{N}$*,* $e \equiv e_1 = e_2$*, or* $e \equiv (x :^\theta e_1)^\varepsilon \to e_2$ *for some expressions* e_1*,* e_2*, staging classifier* θ

and consistency classifier ε.

Proof. This is a proof by induction on the assumed typing derivation. The previous lemma imples that e cannot be Type. Thus, we only have the following non-trivial cases:

Case.

$$\frac{}{\cdot \vdash^{L} \mathbb{N} : \mathsf{Type}}\ \text{K\_Nat}$$

In this case $e \equiv \mathbb{N}$, thus we obtain our result.

Case.

$$\frac{\begin{array}{c}\cdot \vdash^{\theta'} e_1 : \mathsf{Type} \\ \cdot, x :^{\theta'} e_1 \vdash^{L} e_2 : \mathsf{Type}\end{array}}{\cdot \vdash^{L} (x :^{\theta'} e_1)^{\varepsilon} \to e_2 : \mathsf{Type}}\ \text{K\_Pi}$$

Similar to the previous case.

Case.

$$\frac{\begin{array}{c}\cdot \vdash^{\theta_1} e : e_1 \\ \cdot \vdash^{\theta_2} e' : e_2\end{array}}{\cdot \vdash^{L} e = e' : \mathsf{Type}}\ \text{K\_Eq}$$

Similar to the previous case.

Case.

$$\frac{\begin{array}{l}\cdot \vdash^L [v/x]\mathsf{Type} : \mathsf{Type} \\ \cdot \vdash^L e : (x :^\theta e_1)^+ \to \mathsf{Type} \\ \cdot \vdash^\theta v : e_1\end{array}}{\cdot \vdash^L e\ v : [v/x]\mathsf{Type}}\ \text{AppPiTerm}$$

This case is impossible, because it requires $\cdot \vdash^L [v/x]\mathsf{Type} : \mathsf{Type}$ which is equivalent to $\cdot \vdash^L \mathsf{Type} : \mathsf{Type}$ to be derivable, but by the previous lemma this is impossible.

Case.

$$\frac{\begin{array}{l}\cdot \vdash^L [v/x]\mathsf{Type} : \mathsf{Type} \\ \cdot \vdash^L e : (x :^\theta e_1)^- \to e_2 \\ \cdot \vdash^\theta v : e_1\end{array}}{\cdot \vdash^L e : [v/x]\mathsf{Type}}\ \text{AppAllTerm}$$

Similar to the previous case.

Case.

$$\frac{\begin{array}{l}\cdot \vdash^L [e_1'/x]\mathsf{Type} : \mathsf{Type} \\ \cdot \vdash^L e : [e_1/x]\mathsf{Type} \\ \cdot \vdash^L e' : e_1 = e_1'\end{array}}{\cdot \vdash^L e : [e_1'/x]\mathsf{Type}}\ \text{Conv}$$

Similar to the previous case.

□

We now state and prove the critical properties of the interpretation of types. These are standard, and so we simply list them with their proofs.

Lemma 10.3.0.4. *[CR-Norm] If $t \in [\![e]\!]^\theta_\rho$ then t is closed and $t \in V$.*

Proof. This holds by definition of the interpretation of types. □

Lemma 10.3.0.5. *[CR-Pres] If $t \in [\![e]\!]^\theta_\rho$ and $t \rightsquigarrow t'$ then $t' \in [\![e]\!]^\theta_\rho$*

Proof. This is a proof by structural induction on e. By assumption we know $\cdot \vdash^\theta \rho\ e :$ Type, for each case below. Hence, we assume this for the remainder of the proof. In each of the function-type cases below we assume $t \rightsquigarrow t'$, and $t' \in V$, because in each case $t \in V$. We have two cases to consider when $\theta = L$ and when $\theta = P$. The latter is trivial so we only consider the former.

Case. Let $e \equiv \mathbb{N}$. Suppose $t \in [\![\mathbb{N}]\!]^L_\rho$. Then by the definition of the interpretation of types, $t \in N$ and t is closed. Since CBV is deterministic, if $t \rightsquigarrow t'$ then $t' \in N$, and t being closed implies t' is closed. At this point we still need show that $\cdot \vdash^L t' : \mathbb{N}$. This easily follows from type preservation (Theorem 13.1.0.4). Thus, $t' \in [\![\mathbb{N}]\!]^L_\rho$.

Case. Let $e \equiv (x :^\theta e_1)^+ \to e_2$. Suppose $t \in [\![e]\!]^L_\rho$. By the definition of the interpretation of types we know for all $e' \in [\![e_1]\!]^\theta_\rho$, we have $t\ e' \in [\![e_2]\!]^L_{\rho[x \mapsto e']}$. Let $e'' \in [\![e_1]\!]^\theta_\rho$. By the definition of left-to-right CBV, $t\ e'' \rightsquigarrow t'\ e''$. By the induction hypothesis, $t'\ e'' \in [\![e_2]\!]^L_{\rho[x \mapsto e'']}$. Now we need to show $\cdot \vdash^L t' : (x :^\theta \rho\ e_1)^+ \to \rho\ e_2$. This easly follows from type preservation. Thus, by the definition of the interpretation of types, $t' \in [\![e]\!]^L_\rho$, because e'' is arbitrary.

Case. Let $e \equiv (x :^\theta e_1)^- \to e_2$. Suppose $t \in [\![e]\!]^L_\rho$. By type preservation we know $\cdot \vdash^L t' : (x :^\theta \rho\ e_1)^- \to \rho\ e_2$. Let $u \in [\![e_1]\!]^\theta_\rho$. Then $t \in [\![e_2]\!]^L_{\rho[x \mapsto u]}$.

By the induction hypothesis, $t' \in [\![e_2]\!]^L_{\rho[x \mapsto u]}$. Thus, by the definition of the interpretation of types, $t' \in [\![e]\!]^L_\rho$.

Case. Let $e \equiv e_1 = e_2$ and $t \in [\![e]\!]^L_\rho$. By type preservation we know $\cdot \vdash^L t' : \rho\ e_1 = \rho\ e_2$. By the definition of the interpretation of types, $t \in V$, $\rho\ e_1 \downarrow \rho\ e_2$, hence, $t' \in V$, because $t \rightsquigarrow t'$. Thus, again by the definition of the interpretation of types, $t' \in [\![e]\!]^L_\rho$.

□

Lemma 10.3.0.6. *[CR-Prog] If $t \rightsquigarrow t'$, $\cdot \vdash^L t : \rho\ e$, and $t' \in [\![e]\!]^L_\rho$ then $t \in [\![e]\!]^L_\rho$.*

Proof. This is a proof by structural induction on e. By assumption we know $\cdot \vdash^\theta \rho\ e : \mathsf{Type}$, for each case below. Hence, we assume this for the remainder of the proof. In each of the cases below we assume $t \rightsquigarrow t'$, t is closed, and $t' \in [\![e]\!]^L_\rho$ and in the function-type cases we assume $t \in V$, because in each case $t' \in V$. By assumption t has the required type, thus we omit this assumption from the remainder of the proof.

Case. Let $e \equiv \mathbb{N}$. By the definition of the interpretation of types we know $t' \in N$ and t' is closed. Now since CBV is deterministic and $t \rightsquigarrow t'$ we know $t \in N$. Thus, $t \in [\![\mathbb{N}]\!]^L_\rho$.

Case. Let $e \equiv (x :^\theta e_1)^+ \to e_2$. By the definition of the interpretation of types, for all $e' \in [\![e_1]\!]^\theta_\rho$, we have $t'\ e' \in [\![e_2]\!]^L_{\rho[x \mapsto e']}$. Let $e'' \in [\![e_1]\!]^\theta_\rho$. By the definition of left-to-right CBV, $t\ e'' \rightsquigarrow t'\ e''$ and by the induction hypothesis, $t\ e'' \in [\![e_2]\!]^L_{\rho[x \mapsto e'']}$. By the definition of the interpretation of types, $t \in [\![e]\!]^L_\rho$.

Case. Let $e \equiv (x :^{\theta} e_1)^{-} \to e_2$ and $u \in [\![e_1]\!]^{\theta}_{\rho}$. Then $t' \in [\![e_2]\!]^{L}_{\rho[x \mapsto u]}$. By the induction hypothesis, $t \in [\![e_2]\!]^{L}_{\rho[x \mapsto u]}$, thus, by the definition of the interpretation of types, $t \in [\![e]\!]^{L}_{\rho}$.

Case. Let $e \equiv e_1 = e_2$. By the definition of the interpretation of types, $t' \in V$, $\rho\ e_1 \downarrow \rho\ e_2$, hence, $t \in V$, because $t \rightsquigarrow t'$. Therefore, by the definition of the interpretation of types, $t \in [\![e]\!]^{L}_{\rho}$.

□

The following results about substitution are also standard.

Lemma 10.3.0.7. *[Substitution Distribution]* $[\![e]\!]^{\theta}_{\rho[x \mapsto e']} = [\![[e'/x]e]\!]^{\theta}_{\rho}$

Proof. This is a proof by induction on the form of e. We first consider when $\theta = L$ and then when $\theta = P$. Note that we do not show both directions of the equality. We only prove left to right the other direction is similar.

Case. Let $e \equiv \mathbb{N}$. Since $\rho\ \mathbb{N} = \mathbb{N}$ for any substitution ρ, clearly $[\![e]\!]^{L}_{\rho[x \mapsto e']} = [\![[e'/x]e]\!]^{L}_{\rho}$.

Case. Let $e \equiv (y :^{\theta} e_1)^{+} \to e_2$. Suppose $a \in [\![e]\!]^{\theta}_{\rho[x \mapsto e']}$. Then by the interpretations of types, $\cdot \vdash^{\theta} a : (y :^{\theta} \rho[x \mapsto e']\ e_1)^{+} \to (\rho[x \mapsto e']\ e_2)$, $a \in V$, and for any $a' \in [\![e_1]\!]^{L}_{\rho[x \mapsto e']}$ we have $a\ a' \in [\![e_2]\!]^{L}_{\rho[x \mapsto e'][y \mapsto a']}$. Clearly, $\rho[x \mapsto e'][y \mapsto a']$ is equivalent to $\rho[y \mapsto [e'/x]a'][x \mapsto e']$, thus, $a\ a' \in [\![e_2]\!]^{L}_{\rho[y \mapsto [e'/x]a'][x \mapsto e']}$. Now by the induction hypothesis, $[\![e_1]\!]^{L}_{\rho[x \mapsto e']} = [\![[e'/x]e_1]\!]^{L}_{\rho}$ and $[\![e_2]\!]^{L}_{\rho[y \mapsto [e'/x]a'][x \mapsto e']} =$

$[\![[e'/x]e_2]\!]^L_{\rho[y\mapsto[e'/x]a']}$. Therefore, by the definition of the interpretation of types, $[\![e]\!]^L_{\rho[x\mapsto e']} \subseteq [\![[e'/x]e]\!]^L_\rho$.

Case. Let $e \equiv (y :^\theta e_1)^- \to e_2$. Similar to the previous case.

Case. Let $e \equiv e_1 = e_2$. Suppose $a \in [\![e]\!]^\theta_{\rho[x\mapsto e']}$. By the definition of the interpretation of types, $\cdot \vdash^L a : \rho[x \mapsto e']\ e_1 = \rho[x \mapsto e']\ e_2$, $\cdot \vdash^L \rho[x \mapsto e']\ e_1 = \rho[x \mapsto e']\ e_2 : \mathsf{Type}$, $a \in V$, and $\rho[x \mapsto e']\ e_1 \downarrow \rho[x \mapsto e']\ e_2$. Clearly, if $\cdot \vdash^L a : \rho[x \mapsto e']\ e_1 = \rho[x \mapsto e']\ e_2$ then $\cdot \vdash^L a : \rho\ [e'/x]e_1 = \rho\ [e'/x]e_2$, the same applies to the kinding judgment and if $\rho[x \mapsto e']\ e_1 \downarrow \rho[x \mapsto e']\ e_2$ then $\rho\ [e'/x]e_1 \downarrow \rho\ [e'/x]e_2$. Therefore, by the definition of the interpretation of types, $[\![e]\!]^L_{\rho[x\mapsto e']} \subseteq [\![[e'/x]e]\!]^L_\rho$.

Now assume $\theta = P$.

Case. Assume $a \in [\![e]\!]^P_{\rho[x\mapsto e']}$. By the definition of the interpretation of types, $\cdot \vdash^P a : \rho[x \mapsto e']\ e$, $\cdot \vdash^P \rho[x \mapsto e']\ e : \mathsf{Type}$ and a is a value. Just as in the previous case if $\cdot \vdash^P a : \rho[x \mapsto e']\ e$ and $\cdot \vdash^P \rho[x \mapsto e']\ e : \mathsf{Type}$ then $\cdot \vdash^P a : \rho\ [e'/x]e$ and $\cdot \vdash^P \rho\ [e'/x]e : \mathsf{Type}$. Therefore, by the definition of the interpretation of types, $[\![e]\!]^L_{\rho[x\mapsto e']} \subseteq [\![[e'/x]e]\!]^L_\rho$.

□

Lemma 10.3.0.8. *[Well-formed substitution for typing] If* $\rho \in [\![\Gamma]\!]$ *and* $\Gamma, \Gamma' \vdash^\theta e : e'$ *then* $\Gamma' \vdash^\theta \rho\ e : \rho\ e'$.

Proof. This is a proof by induction on the form of the assumped typing judgment. We only show a few cases all the others are either trivial, or similar to the cases we

given below.

Case.

$$\frac{\begin{array}{l}\Gamma, \Gamma' \vdash^{\theta'} e_1 : \mathsf{Type} \\ \Gamma, \Gamma', x :^{\theta'} e_1 \vdash^{\theta} e_2 : \mathsf{Type}\end{array}}{\Gamma, \Gamma' \vdash^{\theta} (x :^{\theta'} e_1)^{\epsilon} \to e_2 : \mathsf{Type}}\ \text{K\_Pi}$$

By the induction hypothesis, $\Gamma' \vdash^{\theta'} \rho\ e_1 : \mathsf{Type}$ and $\Gamma', x :^{\theta'} \rho\ e_1 \vdash^{\theta} \rho\ e_2 : \mathsf{Type}$. Thus, by applying K_Pi $\Gamma' \vdash^{\theta} (x :^{\theta'} \rho\ e_1)^{\epsilon} \to \rho\ e_2 : \mathsf{Type}$, which is equivalent to $\Gamma' \vdash^{\theta} \rho\ ((x :^{\theta'} e_1)^{\epsilon} \to e_2) : \mathsf{Type}$.

Case.

$$\frac{\begin{array}{l}\Gamma, \Gamma' \vdash^{\theta} e : \mathsf{Type} \\ x :^{\theta} e \in \Gamma, \Gamma'\end{array}}{\Gamma, \Gamma' \vdash^{\theta} x :^{\theta} e}\ \text{Var}$$

We have two cases to consider, either $x :^{\theta} e \in \Gamma$ or $x :^{\theta} e \in \Gamma'$. Suppose the former. By the induction hypothesis, $\Gamma' \vdash^{\theta} \rho\ e : \mathsf{Type}$. Since $x :^{\theta} e \in \Gamma$ there exists a mapping $(x \mapsto e') \in \rho$ such that $e' \in [\![e]\!]^{\theta}_{\rho}$. By the definition of the interpretation of types $\cdot \vdash^{\theta} e' : \rho\ e$. Now by weakening (Lemma 10.1.0.1) we know $\Gamma' \vdash^{\theta} e' : \rho\ e$ which is equivalent to $\Gamma' \vdash^{\theta} \rho\ x :^{P} \rho\ e$.

Now suppose the that $x :^{\theta} e \in \Gamma'$, then $\rho\, x = x$. So by the induction hypothesis we know $\Gamma' \vdash^{\theta} \rho\ e : \mathsf{Type}$, and by reapplying the Var rule we obtain $\Gamma' \vdash^{\theta} x :^{\theta} \rho\, e$ which is equivalent to $\Gamma' \vdash^{\theta} \rho\, x :^{\theta} \rho\, e$.

□

In order to prove that the Join and Conv are logically consistent, we need to know that if a type is equivalent to another, then their interpretations are equivalent. It turns out that instead of proving the equality directly we can prove a much simpler result. It suffices to show that if a type is equal to another, then the interpretation of the first is subset of the interpretation of the second. Then using symmetry of the equality proof we can obtain that they are in fact equal. The following two lemmas prove this for both the programmatic fragment and the logical fragment respectfully:

Lemma 10.3.0.9. *[Computational Semantic Conversion] If* $e \downarrow e'$ *and* $\Gamma \vdash^{\theta'} e : B$ *and* $\Gamma \vdash^{\theta''} e' : C$ *then* $[\![[e/x]A]\!]^P_\rho \subseteq [\![[e'/x]A]\!]^P_\rho$.

Proof. We know that $a \in [\![[e/x]A]\!]^P_\rho$ iff $\cdot \vdash^P a : \rho\ [e/x]A$ and a *is a value* and $\cdot \vdash^P \rho\ [e/x]A : \mathsf{Type}$ by the definition of the interpretation of types. By Join, $\Gamma \vdash^L join : e = e'$. Finally, by Conv, $\cdot \vdash^P a : \rho\ [e'/x]A$. Now by regularity $\cdot \vdash^P \rho\ [e'/x]A : \mathsf{Type}$. Thus, $a \in [\![[e'/x]A]\!]^P_\rho$ and $[\![[e/x]A]\!]^P_\rho \subseteq [\![[e'/x]A]\!]^P_\rho$. □

Lemma 10.3.0.10. *[Logical Semantic Conversion] If* $e \downarrow e'$ *and* $\Gamma \vdash^{\theta'} e : B$ *and* $\Gamma \vdash^{\theta''} e' : C$ *then* $[\![[e/x]A]\!]^L_\rho \subseteq [\![[e'/x]A]\!]^L_\rho$.

Proof. We proceed by induction on the form of A. In each case we must show $\cdot \vdash^L a : \rho[e'/x]A$ and $\cdot \vdash^L \rho[e'/x]A : \mathsf{Type}$. The former is easly accomplished by first applying Join to obtain $\cdot \vdash^L join : e = e'$ and then applying Conv to obtain the desired result. The latter is obtained by regularity.

Case. Let $A \equiv \mathbb{N}$. Obvious.

Case. Let $A \equiv (x :^{\theta} a_1)^{+} \to a_2$, ρ be an arbitrary substitution and $a \in [\![[e/x]A]\!]^L_\rho$. By the definition of the interpretations of types, $a \in [\![[e/x]A]\!]^L_\rho$ iff for any $a' \in [\![[e/x]a_1]\!]^\theta_\rho$ we have $a\ a' \in [\![[e/x]a_2]\!]^L_{\rho[x\mapsto a']}$. Let a'' be an arbitrary element of $[\![[e/x]a_1]\!]^\theta_\rho$. We have two cases to consider: when $\theta = L$ and $\theta = P$. If $\theta = L$ then by the induction hypothesis, $[\![[e/x]a_1]\!]^L_\rho = [\![[e'/x]a_1]\!]^L_\rho$. If $\theta = P$ then we apply Lemma 10.3.0.9 to obtain $[\![[e/x]a_1]\!]^P_\rho = [\![[e'/x]a_1]\!]^P_\rho$. Also by the induction hypothesis, $[\![[e/x]a_2]\!]^L_{\rho[x\mapsto a'']} = [\![[e'/x]a_2]\!]^L_{\rho[x\mapsto a'']}$. Thus, $a'' \in [\![[e'/x]a_1]\!]^\theta_\rho$ and $a\ a'' \in [\![[e'/x]a_2]\!]^L_{\rho[x\mapsto a'']}$ for arbitrary a'', hence, by the definition of the interpretations of types, $a \in [\![[e'/x]A]\!]^L_\rho$. Therefore, $[\![[e/x]A]\!]^L_\rho \subseteq [\![[e'/x]A]\!]^L_\rho$.

Case. Let $A \equiv (x :^{\theta} e_1)^{-} \to e_2$. Similar to the previous case.

Case. Let $A \equiv a_1 = a_2$, ρ be an arbitrary substitution and $a \in [\![[e/x]A]\!]^L_\rho$. By the definition of the interpretations of types, $a \in [\![[e/x]A]\!]^L_\rho$ iff $a \in V$ and $\rho[e/x]a_1 \downarrow \rho[e/x]a_2$. Clearly, if $e \downarrow e'$ and $\rho[e/x]a_1 \downarrow \rho[e/x]a_2$ then $\rho[e'/x]a_1 \downarrow \rho[e'/x]a_2$. Thus, $a \in [\![[e'/x]A]\!]^L_\rho$. Therefore, $[\![[e/x]A]\!]^L_\rho \subseteq [\![[e'/x]A]\!]^L_\rho$.

□

Finally, we have arrived at the main result, logical consistency. The following soundness result implies that any well-typed expression can be closed using a series of well-typed expressions, and this closed expression is joinable with a well-typed value. Thus, all expressions of the logical fragment terminate with a value, and we are free to call these expressions and values proofs, and their types formulas.

Theorem 10.3.0.11. *[Type soundness] If* $\Gamma \vdash^L e : e'$ *then for all* $\rho \in [\![\Gamma]\!], \rho\ e \in$

$[\![e']\!]^L_\rho$.

Proof. Throughout this proof we implicitly assume an arbitrary $\rho \in [\![\Gamma]\!]$ and that $\cdot \vdash^L \rho\ e' : \mathsf{Type}$. The latter holds by first applying regularity to obtain $\Gamma \vdash^L e : \mathsf{Type}$ and then applying Lemma 10.3.0.8 to obtain $\cdot \vdash^L \rho\ e : \mathsf{Type}$.

Case.

$$\frac{\Gamma \vdash^L e : \mathsf{Type} \quad x :^L e \in \Gamma}{\Gamma \vdash^L x :^L e}\ \text{Var}$$

By definition of $[\![\Gamma]\!]$ we know there exists some e' such that $(x, e') \in \rho$ and $e' \in [\![e]\!]^L_\rho$. Now $\rho\ x \equiv e' \in [\![e]\!]^L_\rho$.

Case.

$$\frac{\Gamma \vdash^\theta e_1 : \mathsf{Type} \quad \Gamma, x :^\theta e_1 \vdash^L e : e_2}{\Gamma \vdash^L \lambda x.e : (x :^\theta e_1)^+ \to e_2}\ \text{Lam}$$

We need to show $\rho\ \lambda x.e \equiv \lambda x.(\rho\ e) \in [\![(x :^\theta e_1)^+ \to e_2]\!]^L_\rho$. So by the definition of the interpretation of types, we must show for any $e' \in [\![e_1]\!]^\theta_\rho$, $(\lambda x.\rho\ e)\ e' \in [\![e_2]\!]^L_{\rho[x \mapsto e']}$. Let $e' \in [\![e_1]\!]^\theta_\rho$. By CR-Norm, e' is closed and $e' \in V$, hence, $e' \rightsquigarrow^!\ v$ and by CR-Pres, $v \in [\![e_1]\!]^\theta_\rho$. By the definition of the left-to-right CBV, $(\lambda x.(\rho\ e))\ e' \rightsquigarrow^*\ (\lambda x.(\rho\ e))\ v \rightsquigarrow \rho[v/x]\ e$. We know $v \in [\![e_1]\!]^\theta_\rho$ and $\rho \in [\![\Gamma]\!]$, hence, by the definition of well-formed substitutions, $\rho[v/x] \in [\![\Gamma, x :^\theta e_1]\!]$.

We can now apply the induction hypothesis to obtain, $\rho[v/x]e \in [\![e_2]\!]^L_{\rho[x\mapsto v]}$. It is easy to see that $(\lambda x.(\rho\ e))\ e'$ is closed so we can apply CR-Pres and obtain $(\lambda x.(\rho\ e))\ e' \in [\![e_2]\!]^L_{\rho[x\mapsto v]}$. By Lemma 10.3.0.10 and Lemma 10.3.0.7, $[\![e_2]\!]^L_{\rho[x\mapsto v]} = [\![e_2]\!]^L_{\rho[x\mapsto e']}$. Thus, by the definition of the interpretation of types, $\lambda x.(\rho\ e) \in [\![(x :^\theta e_1)^+ \to e_2]\!]^L_\rho$.

Case.

$$\frac{\begin{array}{l}\Gamma \vdash^\theta e_1 : \mathsf{Type} \\ \Gamma, x :^\theta e_1 \vdash^L v : e_2 \\ x \notin fvs(v)\end{array}}{\Gamma \vdash^L v : (x :^\theta e_1)^- \to e_2}\ \text{ILam}$$

We need to show $\rho\ v \in [\![(x :^\theta e_1)^- \to e_2]\!]^L_\rho$. So by the definition of the interpretation of types, we must show for any $e' \in [\![e_1]\!]^\theta_\rho$, $\rho\ v \in [\![e_2]\!]^L_{\rho[x\mapsto e']}$. Let $e' \in [\![e_1]\!]^\theta_\rho$. By CR-Norm, e' is closed and $e' \in V$, hence, $e' \rightsquigarrow^! v'$ and by CR-Pres $v' \in [\![e_1]\!]^\theta_\rho$. By the definition of the left-to-right CBV, $(\rho\ v)\ e' \rightsquigarrow^* (\rho\ v)\ v' \rightsquigarrow \rho[v'/x]\ v \equiv \rho\ v$. We know $v' \in [\![e_1]\!]^\theta_\rho$ and $\rho \in [\![\Gamma]\!]$, hence, by the definition of well-formed substitutions, $\rho[v'/x] \in [\![\Gamma, x :^\theta e_1]\!]$. Thus, we can now apply the induction hypothesis to obtain, $\rho\ v \in [\![e_2]\!]^L_{\rho[x\mapsto v']}$. By Lemma 10.3.0.10 and Lemma 10.3.0.7 $[\![e_2]\!]^L_{\rho[x\mapsto v']} = [\![e_2]\!]^L_{\rho[x\mapsto e']}$. Thus, by the definition of the interpretation of types, $\rho\ v \in [\![(x :^\theta e_1)^- \to e_2]\!]^L_\rho$.

Case.

$$\frac{\Gamma \vdash^L e : (x :^\theta e_1)^+ \to e_2 \quad \Gamma \vdash^\theta v : e_1}{\Gamma \vdash^L e\ v : [v/x]e_2}\ \text{AppPiTerm}$$

By the induction hypothesis, $\rho\ e \in [\![(x :^\theta e_1)^+ \to e_2]\!]^L_\rho$. If $\theta = L$ then by the induction hypothesis, $\rho\ v \in [\![e_1]\!]^\theta_\rho$. If $\theta = P$ then by Lemma 10.3.0.8, $\cdot \vdash^P \rho\ v : e_1$, and by the definition of the interpretation of types, $\rho\ v \in [\![e_1]\!]^\theta_\rho$. Now we know by the definition of the interpretation of types that for any $v' \in [\![e_1]\!]^\theta_\rho$, $(\rho\ e)\ v' \in [\![e_2]\!]^L_{\rho[x \mapsto v']}$. Instantiate v' with $\rho\ v$. Then $(\rho\ e)\ \rho\ v \equiv \rho(e\ v) \in [\![e_2]\!]^L_{\rho[x \mapsto \rho\ v]}$.

Case.

$$\frac{\Gamma \vdash^L e : (x :^\theta e_1)^- \to e_2 \quad \Gamma \vdash^\theta v : e_1}{\Gamma \vdash^L e : [v/y]e_2}\ \text{AppAllTerm}$$

By the induction hypothesis, $\rho\ e \in [\![(x :^\theta e_1)^- \to e_2]\!]^L_\rho$. If $\theta = L$ then by the induction hypothesis, $\rho\ v \in [\![e_1]\!]^\theta_\rho$. If $\theta = P$ then by Lemma 10.3.0.8, $\cdot \vdash^P \rho\ v : \rho\ e_1$, thus, by the definition of the interpretation of types, $\rho\ v \in [\![e_1]\!]^\theta_\rho$. We know by the definition of the interpretation of types that for any $v' \in [\![e_1]\!]^\theta_\rho$, $\rho\ e \in [\![e_2]\!]^L_{\rho[x \mapsto v']}$. Instantiate v' with $\rho\ v$. Then $\rho\ e \in [\![e_2]\!]^L_{\rho[x \mapsto \rho\ v]}$.

Case.

$$\frac{e \downarrow e' \quad \Gamma \vdash^{\theta_1} e : e_1 \quad \Gamma \vdash^{\theta_2} e' : e_2}{\Gamma \vdash^L \mathsf{join} : e = e'}\ \text{join}$$

It is a property of left-to-right CBV that if $e \downarrow e'$ then $\rho\ e \downarrow \rho\ e'$ for any substitution ρ. By Lemma 10.3.0.8, $\cdot \vdash^L join : \rho\ e = \rho\ e'$. Hence by the definition of the interpretation of types, $join \in [\![e = e']\!]^L_\rho$.

Case.

$$\frac{\Gamma \vdash^L e' : ((x :^\theta e_1)^+ \to e_2) = ((x :^\theta e_1')^+ \to e_2')}{\Gamma \vdash^L injdom : e_1 = e_1'}\ \text{InjDom}$$

By the induction hypothesis, $e' \in [\![((x :^\theta e_1)^+ \to e_2) = ((x :^\theta e_1')^+ \to e_2')]\!]^L_\rho$, which implies that $\rho\ e_1 \downarrow \rho\ e_1'$. By Lemma 10.3.0.8, $\cdot \vdash^L injdom : \rho\ e_1 = \rho\ e_1'$. Therefore by the definition of the interpretation of types, $injdom \in [\![e_1 = e_1']\!]^L_\rho$.

Case.

$$\frac{\begin{array}{l}\Gamma \vdash^L e' : ((x :^\theta e_1)^+ \to e_2) = ((x :^\theta e_1')^+ \to e_2') \\ \Gamma \vdash^\theta v : e_1\end{array}}{\Gamma \vdash^L injran : [v/x]e_2 = [v/x]e_2'}\ \text{InjRan}$$

By the induction hypothesis, $e' \in [\![((x :^\theta e_1)^+ \to e_2) = ((x :^\theta e_1')^+ \to e_2')]\!]^L_\rho$, which implies that $\rho\ e_2 \downarrow \rho\ e_2'$. Since compatibility joinablity is closed under substitution, $\rho\ [v/x]e_2 \downarrow \rho\ [v/x]e_2'$, we can move ρ to the outside, because x is not a member of the domain of ρ. By Lemma 10.3.0.8, $\cdot \vdash^L injran : \rho\ [v/x]e_2 = \rho\ [v/x]e_2'$. Therefore by the definition of theinterpretation of types, $injran \in [\![[v/x]e_2 = [v/x]e_2']\!]^L_\rho$.

Case.

$$\frac{\Gamma \vdash^L e : [e_1/x]e_2 \quad \Gamma \vdash^L e' : e_1 = e_1'}{\Gamma \vdash^L e : [e_1'/x]e_2}\ \text{Conv}$$

Applying the induction hypothesis to $\Gamma \vdash^L e' : e_1 = e_1'$ implies $\rho\, e_1 \downarrow \rho\, e_1'$, and $\cdot \vdash^L \rho\, e_1 = \rho\, e_1' : \mathsf{Type}$. The latter implies that $\cdot \vdash^{\theta'} e_1 : A$ an $\cdot \vdash^{\theta'} e_1' : A'$. Now by the induction hypothesis, $e \in [\![[e_1/x]e_2]\!]^L_\rho$, and by Lemma 10.3.0.10, $[\![[e_1/x]e_2]\!]^L_\rho = [\![[e_1'/x]e_2]\!]^L_\rho$, thus, $e \in [\![[e_1'/x]e_2]\!]^L_\rho$.

Case.

$$\frac{}{\Gamma \vdash^L \mathsf{S} : (x :^L \mathbb{N})^+ \to \mathbb{N}}\ \text{Succ}$$

Suppose $e \in [\![\mathbb{N}]\!]^L_\rho$ then by CR-Norm e is closed and $e \in N$, hence, $e \rightsquigarrow^* n$, where n is a numeral. Clearly, $S\ n$ is closed and $S\ n \in N$, thus, $S\ n \in [\![\mathbb{N}]\!]^L_{\rho[x \mapsto n]} = [\![\mathbb{N}]\!]^L_\rho$. Before we can apply CR-Prog we must show that $\cdot \vdash^L S\ e : \mathbb{N}$. By the definition of the interpretation of types, $\cdot \vdash^L e : \mathbb{N}$, and by assumption $\cdot \vdash^L S : (x :^L \mathbb{N})^+ \to e_2$. We can now apply AppPiTerm to obtain $\cdot \vdash^L S\ e : \mathbb{N}$. Therefore, by CR-Prog, $S\ e \in [\![\mathbb{N}]\!]^L_\rho$.

Case.

$$\frac{}{\Gamma \vdash^L \mathsf{Z} : \mathbb{N}}\ \text{Zero}$$

Clearly, Z is closed and $Z \in N$. Thus, $Z \in [\![\mathbb{N}]\!]^L_\rho$.

Case.

$$\frac{\Gamma \vdash^{\theta} e_1 : \mathsf{Type} \quad \Gamma \vdash^{L} e : \mathsf{Z} = \mathsf{S}\ e'}{\Gamma \vdash^{L} \mathsf{contra} : e_1}\ \text{Contra}$$

By the induction hypothesis, $\rho\ e \in [\![Z = S\ e']\!]^L_\rho$. By the definition of the interpretation of types, Z and $S\ e'$ are compatibly joinable, which is not the case, hence a contradiction. Thus, we obtain that $contra \in [\![e'']\!]^L_\rho$ for any e''.

Case.

$$\frac{\Gamma \vdash^{\theta} e_1 : \mathsf{Type} \quad \Gamma \vdash^{L} e : v = \mathsf{abort}}{\Gamma \vdash^{L} \mathsf{contra} : e_1}\ \text{ContraAbort}$$

This case is similar to the previous case.

Case.

$$\frac{\Gamma \vdash^{\theta''} e : \mathsf{Type} \quad \Gamma \vdash^{L} e' : ((x :^{\theta} e_1)^{\epsilon} \to e_2) = ((x :^{\theta'} e_1')^{\epsilon'} \to e_2') \quad \theta \neq \theta'}{\Gamma \vdash^{L} contra : e}\ \text{ContraPiTh}$$

The induction hypothesis allows us to conclude that e' is a member of the interpretation of its type which tells us that $\rho\ ((x :^{\theta} e_1)^{\epsilon} \to e_2) \downarrow \rho\ ((x :^{\theta'} e_1')^{\epsilon'} \to e_2'$, but this is a contradiction, because $\theta \neq \theta'$. Therefore, $contra \in [\![e]\!]^L_\rho$.

Case.

$$\frac{\Gamma \vdash^{\theta''} e : \mathsf{Type} \quad \Gamma \vdash^{L} e' : ((x :^{\theta} e_1)^{\epsilon} \to e_2) = ((x :^{\theta'} e_1')^{\epsilon'} \to e_2') \quad \epsilon \neq \epsilon'}{\Gamma \vdash^{L} \mathit{contra} : e}\ \text{ContraPiEp}$$

Similar to the previous case.

Case.

$$\frac{\Gamma, x :^{L} \mathbb{N} \vdash^{L} (y :^{L} \mathbb{N})^{+} \to (p :^{L} x = \mathsf{S}\, y)^{-} \to [y/x]e_2 : \mathsf{Type} \quad \Gamma, x :^{L} \mathbb{N}, f :^{L} (y :^{L} \mathbb{N})^{+} \to (p :^{L} x = \mathsf{S}\, y)^{-} \to [y/x]e_2 \vdash^{L} v : e_2 \quad f, p \notin \mathsf{FV}(e_2)}{\Gamma \vdash^{L} \mathsf{rec}\, f\, x\, v : (x :^{L} \mathbb{N})^{+} \to e_2}\ \text{RecNat}$$

We need to show that $\rho\ rec\ f\ x\ v \equiv rec\ f\ x\ \rho\ v \in [\![(x :^{L} \mathbb{N})^{+} \to e_2]\!]^{L}_{\rho}$. By the definition of the interpretation of types, $rec\ f\ x\ \rho\ v \in [\![(x :^{L} \mathbb{N})^{+} \to e_2]\!]^{L}_{\rho}$ iff for any $e \in [\![\mathbb{N}]\!]^{L}_{\rho}$, we have $(rec\ f\ x\ \rho\ v)\ e \in [\![e_2]\!]^{L}_{\rho[x \mapsto e]}$, because $f \notin FV(e_2)$. Let e' be an arbitrary element of $[\![\mathbb{N}]\!]^{L}_{\rho}$. By the definition of the interpretation of types, $e' \rightsquigarrow^{!} n \in N$, hence, $n \in V$ and by CR-Pres, $n \in [\![\mathbb{N}]\!]^{L}_{\rho}$. Thus, $(rec\ f\ x\ \rho\ v)\ e' \rightsquigarrow^{*} (rec\ f\ x\ \rho\ v)\ n \rightsquigarrow [n/x][rec\ f\ x\ \rho\ v/f]\rho\ v$. By the induction hypothesis, for any $\rho' \in [\![\Gamma, x :^{L} \mathbb{N}, f :^{L} (y :^{L} \mathbb{N})^{+} \to (p :^{L} x = S\ y)^{-} \to [y/x]e_2]\!]$, we have $\rho'\ v \in [\![e_2]\!]^{L}_{\rho'}$.

At this point we need to show that $[n/x][rec\ f\ x\ \rho\ v/f]\rho \in [\![\Gamma, x :^{L} \mathbb{N}, f :^{L} (y :^{L} \mathbb{N})^{+} \to (p :^{L} x = S\ y)^{-} \to [y/x]e_2]\!]$, but to conclude this we must have

$rec\ f\ x\ \rho\ v \in [\![(y :^L \mathbb{N})^+ \to (p :^L x = S\ y)^- \to [y/x]e_2]\!]^L_{\rho[x\mapsto n]}$, which means we have to prove the following proposition.

Proposition 10.3.0.12. *For any $n \in \mathbb{N}$, $rec\ f\ x\ \rho\ v \in [\![(y :^L \mathbb{N})^+ \to (p :^L x = S\ y)^- \to [y/x]e_2]\!]^L_{\rho[x\mapsto n]}$.*

We proceed by induction on n. For the base case let $n \equiv 0$. Then $rec\ f\ x\ \rho\ v \in [\![(y :^L \mathbb{N})^+ \to (p :^L x = S\ y)^- \to [y/x]e_2]\!]^L_{\rho[x\mapsto 0]}$ iff for any $e' \in [\![\mathbb{N}]\!]^L_{\rho[x\mapsto 0]}$, we have $(rec\ f\ x\ \rho\ v)\ e' \in [\![(p :^L x = S\ y)^- \to [y/x]e_2]\!]^L_{\rho[x\mapsto 0][y\mapsto e']}$. Let e'' be an arbitrary element of $[\![\mathbb{N}]\!]^L_{\rho[x\mapsto 0]}$. Then we know $e'' \in [\![\mathbb{N}]\!]^L_{\rho[x\mapsto 0]}$ and $e'' \rightsquigarrow^!\ n \in N$, hence, $n \in V$. Now we have to show that $(rec\ f\ x\ \rho\ v)\ e'' \rightsquigarrow^*\ (rec\ f\ x\ \rho\ v)\ n \in [\![(p :^L x = S\ y)^- \to [y/x]e_2]\!]^L_{\rho[x\mapsto 0][y\mapsto n]}$. By the definition of the interpretation types we must show that for any $e''' \in [\![x = S\ y]\!]^L_{\rho[x\mapsto 0][y\mapsto n]}$, we have, $(rec\ f\ x\ \rho\ v)\ n \in [\![[y/x]e_2]\!]_{\rho[x\mapsto 0][y\mapsto n][p\mapsto e''']}$. Let $a \in [\![x = S\ y]\!]^L_{\rho[x\mapsto 0][y\mapsto n]}$. By the definition of the interpretation, $a \downarrow join$ and $0 \downarrow S\ n$, but this is a contradiction.

Now let $n \equiv S\ n'$. By the definition of the interpretation of types, $rec\ f\ x\ \rho\ v \in [\![(y :^L \mathbb{N})^+ \to (p :^L x = S\ y)^- \to [y/x]e_2]\!]^L_{\rho[x\mapsto n]}$ iff for any $a \in [\![\mathbb{N}]\!]^L_{\rho[x\mapsto n]}$, we have $(rec\ f\ x\ \rho\ v)\ a \in [\![(p :^L x = S\ y)^- \to [y/x]e_2]\!]^L_{\rho[x\mapsto n][y\mapsto a]}$. Let a' be an arbitrary a. By CR-Norm and the definition of the interpretation of types, $a' \in N$, $a' \in V$, hence, $a' \rightsquigarrow^!\ v' \in \mathbb{N}$. By CR-Pres and the definition of the interpretation of types, $(rec\ f\ x\ \rho\ v)\ a' \rightsquigarrow^*\ (rec\ f\ x\ \rho\ v)\ v' \in [\![(p :^L x = S\ y)^- \to [y/x]e_2]\!]^L_{\rho[x\mapsto n][y\mapsto v']}$ iff for any $a'' \in [\![x = S\ y]\!]^L_{\rho[x\mapsto n][y\mapsto v']}$, we have $(rec\ f\ x\ \rho\ v)\ v' \in [\![[y/x]e_2]\!]^L_{\rho[x\mapsto n][y\mapsto v'][p\mapsto a'']}$. Let $u \in [\![x = S\ y]\!]^L_{\rho[x\mapsto n][y\mapsto v']}$. By the definition of the interpretation of types, $u \downarrow join$ and $S\ n' \downarrow S\ v'$, which

implies $n' \downarrow v'$. Now $(rec\ f\ x\ \rho\ v)\ v' \rightsquigarrow [v'/x][rec\ f\ x\ \rho\ v/f]\rho\ v$, so by CR-Pres it suffices to show, $[v'/x][rec\ f\ x\ \rho\ v/f]\rho\ v \in [\![[y/x]e_2]\!]^L_{\rho[x\mapsto n][y\mapsto v'][p\mapsto u]}$. It is easy to see that x is not free in $[y/x]e_2$ and we know by assumption p is also not free in $[y/x]e_2$, hence, $[\![[y/x]e_2]\!]^L_{\rho[x\mapsto n][y\mapsto v'][p\mapsto u]} = [\![[y/x]e_2]\!]^L_{\rho[y\mapsto v']}$, and by a simple renaming of variables, $[\![[y/x]e_2]\!]^L_{\rho[y\mapsto v']} = [\![e_2]\!]^L_{\rho[x\mapsto v']}$. Thus, it suffices to show, $[v'/x][rec\ f\ x\ \rho\ v/f]\rho\ v \in [\![e_2]\!]^L_{\rho[x\mapsto v']}$. Finally, by Lemma 10.3.0.7, it suffices to show, $[n'/x][rec\ f\ x\ \rho\ v/f]\rho\ v \in [\![e_2]\!]^L_{\rho[x\mapsto n']}$, because $v' \downarrow n'$.

By the inner induction hypothesis, $rec\ f\ x\ \rho\ v \in [\![(y :^L \mathbb{N})^+ \to (p :^L x = S\ y)^- \to [y/x]e_2]\!]^L_{\rho[x\mapsto n']}$. Thus, $[n'/x][rec\ f\ x\ \rho\ v/f]\rho \in [\![\Gamma, x :^L \mathbb{N}, f :^L (y :^L \mathbb{N})^+ \to (p :^L x = S\ y)^- \to [y/x]e_2]\!]$. We can now apply the outer induction hypothesis, where we substitute v for e, e_2 for e', $\Gamma, x :^L \mathbb{N}, f :^L (y :^L \mathbb{N})^+ \to (p :^L x = S\ y)^- \to [y/x]e_2$ for Γ, and $[n'/x][rec\ f\ x\ \rho\ v/f]\rho$ for ρ, to obtain, $[n'/x][rec\ f\ x\ \rho\ v/f]\rho\ v \in [\![e_2]\!]^L_{\rho[x\mapsto n'][f\mapsto rec\ f\ x\ \rho\ v/f]} = [\![e_2]\!]^L_{\rho[x\mapsto n']}$, because $f \notin FV(e_2)$. This concludes the proof of the proposition.

By Proposition 10.3.0.12, $rec\ f\ x\ \rho\ v \in [\![(y :^L \mathbb{N})^+ \to (p :^L x = S\ y)^- \to [y/x]e_2]\!]^L_{\rho[x\mapsto n]}$. Thus, $[n/x][rec\ f\ x\ \rho\ v/f]\rho \in [\![\Gamma, x :^L \mathbb{N}, f :^L (y :^L \mathbb{N})^+ \to (p :^L x = S\ y)^- \to [y/x]e_2]\!]$ and we can finally conclude, $[n/x][rec\ f\ x\ \rho\ v/f]\rho\ v \in [\![e_2]\!]^L_{\rho[x\mapsto n]}$. By CR-Pres, $(rec\ f\ x\ \rho\ v)\ e \in [\![e_2]\!]^L_{\rho[x\mapsto n]}$. Therefore, $rec\ f\ x\ \rho\ v \in [\![(x :^L \mathbb{N})^+ \to e_2]\!]^L_\rho$.

Case.

$$\frac{\begin{array}{l}\Gamma, x :^L \mathbb{N} \vdash^L (y :^L \mathbb{N})^- \to (u :^L x = \mathsf{S}\, y)^- \to [y/x]e_2 : \mathsf{Type} \\ \Gamma, x :^L \mathbb{N}, f :^L (y :^L \mathbb{N})^- \to (u :^L x = \mathsf{S}\, y)^- \to [y/x]e_2 \vdash^L v : e_2 \\ f, p \notin \mathsf{FV}\,(e_2)\end{array}}{\Gamma \vdash^L \mathsf{rec}^-\ f\ v : (x :^L \mathbb{N})^- \to e_2}\ \text{RecNatComp}$$

We must show, for any $a \in [\![\mathbb{N}]\!]^L_\rho$, we have $rec^-\ f\ \rho\ v \in [\![e_2]\!]^L_{\rho[x\mapsto a]}$. Let a' be an arbitrary a. Then by CR-Norm and the interpretation of types, $a' \in N$, $a' \in V$ and $a \rightsquigarrow^! n \in \mathbb{N}$. So $(rec^-\ f\ \rho\ v)\ a \rightsquigarrow^* (rec^-\ f\ \rho\ v)\ n \rightsquigarrow [n/x][rec^-\ f\ \rho\ v/f]\rho\ v \equiv [rec^-\ f\ \rho\ v/f]\rho\ v$. By CR-Pres it suffices to show, $[rec^-\ f\ \rho\ v/f]\rho\ v \in [\![e_2]\!]^L_{\rho[x\mapsto n]}$. To conclude this we have to show $[rec^-\ f\ \rho\ v/f]\rho \in [\![\Gamma, x :^L \mathbb{N}, f :^L (y :^L \mathbb{N})^- \to (p :^L x = S\ y)^- \to [y/x]e_2]\!]$. This requires us to prove the following proposition.

Proposition 10.3.0.13. *For any $n \in \mathbb{N}$, $rec^-\ f\ \rho\ v \in [\![(y :^L \mathbb{N})^- \to (p :^L x = S\ y)^- \to [y/x]e_2]\!]^L_{\rho[x\mapsto n]}$.*

We proceed by induction on n. For the base case let $n \equiv 0$. Then $rec^-\ f\ \rho\ v \in [\![(y :^L \mathbb{N})^- \to (p :^L x = S\ y)^- \to [y/x]e_2]\!]^L_{\rho[x\mapsto 0]}$ iff for any $e' \in [\![\mathbb{N}]\!]^L_{\rho[x\mapsto 0]}$, we have $rec^-\ f\ \rho\ v \in [\![(p :^L x = S\ y)^- \to [y/x]e_2]\!]^L_{\rho[x\mapsto 0][y\mapsto e']}$. Let $e'' \in [\![\mathbb{N}]\!]^L_{\rho[x\mapsto 0]}$. Then, we know, $e'' \rightsquigarrow^! n' \in N$, hence, $n' \in V$. Now we have to show that $(rec^-\ f\ \rho\ v)\ e'' \rightsquigarrow^* (rec^-\ f\ \rho\ v)\ n' \equiv rec^-\ f\ \rho\ v \in [\![(p :^L x = S\ y)^- \to [y/x]e_2]\!]^L_{\rho[x\mapsto 0][y\mapsto n']}$. By the definition of the interpretation types we must show that for any $e''' \in [\![x = S\ y]\!]^L_{\rho[x\mapsto 0][y\mapsto n']}$ we have $rec^-\ f\ \rho\ v \in [\![[y/x]e_2]\!]_{\rho[x\mapsto 0][y\mapsto n'][p\mapsto e''']}$. Let $a \in [\![x = S\ y]\!]^L_{\rho[x\mapsto 0][y\mapsto n']}$. Then by the definition of the interpretation, $a \downarrow join$ and $0 \downarrow S\ n$, but this is a contradiction.

Now let $n \equiv S\ n'$. By the definition of the interpretation of types, $rec^-\ f\ \rho\ v \in [\![(y :^L \mathbb{N})^- \to (p :^L x = S\ y)^- \to [y/x]e_2]\!]^L_{\rho[x\mapsto n]}$ iff for any $a \in [\![\mathbb{N}]\!]^L_{\rho[x\mapsto n]}$, we

have $rec^-\ f\ \rho\ v \in [\![(p :^L x = S\ y)^- \to [y/x]e_2]\!]^L_{\rho[x\mapsto n][y\mapsto a]}$. Let a' be an arbitrary a. Then by CR-Norm and the definition of the interpretation of types, $a' \in N$, $a' \in V$, hence, $a' \rightsquigarrow^! v' \in \mathbb{N}$. By CR-Pres and the definition of the interpretation of types, $(rec^-\ f\ \rho\ v)\ a' \rightsquigarrow^* (rec^-\ f\ \rho\ v)\ v' \equiv rec^-\ f\ \rho\ v \in [\![(p :^L x = S\ y)^- \to [y/x]e_2]\!]^L_{\rho[x\mapsto n][y\mapsto v']}$ iff for any $a'' \in [\![x = S\ y]\!]^L_{\rho[x\mapsto n][y\mapsto v']}$, we have $rec^-\ f\ \rho\ v \in [\![[y/x]e_2]\!]^L_{\rho[x\mapsto n][y\mapsto v'][p\mapsto a'']}$. Suppose $u \in [\![x = S\ y]\!]^L_{\rho[x\mapsto n][y\mapsto v']}$. By the definition of the interpretation of types, $u \downarrow join$ and $S\ n' \downarrow S\ v'$, which implies, $n' \downarrow v'$. Now $(rec^-\ f\ \rho\ v)\ v' \rightsquigarrow [v'/x][rec^-\ f\ \rho\ v/f]\rho\ v \equiv [rec^-\ f\ \rho\ v/f]\rho\ v$, so by CR-Pres it suffices to show, $[rec^-\ f\ \rho\ v/f]\rho\ v \in [\![[y/x]e_2]\!]^L_{\rho[x\mapsto n][y\mapsto v'][p\mapsto u]}$. It is easy to see that x is not free in $[y/x]e_2$ and we know p is also not free in $[y/x]e_2$, hence, $[\![[y/x]e_2]\!]^L_{\rho[x\mapsto n][y\mapsto v'][p\mapsto u]} = [\![[y/x]e_2]\!]^L_{\rho[y\mapsto v']}$, and by a simple renaming of variables, $[\![[y/x]e_2]\!]^L_{\rho[y\mapsto v']} = [\![e_2]\!]^L_{\rho[x\mapsto v']}$. Thus, it suffices to show, $[rec^-\ f\ \rho\ v/f]\rho\ v \in [\![e_2]\!]^L_{\rho[x\mapsto v']}$. Finally, by Lemma 10.3.0.7, it suffices to show, $[rec^-\ f\ \rho\ v/f]\rho\ v \in [\![e_2]\!]^L_{\rho[x\mapsto n']}$, because $v' \downarrow n'$.

By the inner induction hypothesis, $rec^-\ f\ \rho\ v \in [\![(y :^L \mathbb{N})^- \to (p :^L x = S\ y)^- \to [y/x]e_2]\!]^L_{\rho[x\mapsto n']}$. Thus, $[rec^-\ f\ \rho\ v/f]\rho \in [\![\Gamma, x :^L \mathbb{N}, f :^L (y :^L \mathbb{N})^- \to (p :^L x = S\ y)^- \to [y/x]e_2]\!]$. We can now apply the outer induction hypothesis, where we substitute v for e, e_2 for e', $\Gamma, x :^L \mathbb{N}, f :^L (y :^L \mathbb{N})^- \to (p :^L x = S\ y)^- \to [y/x]e_2$ for Γ, and $[rec^-\ f\ \rho\ v/f]\rho$ for ρ, to obtain, $[rec^-\ f\ \rho\ v/f]\rho\ v \in [\![e_2]\!]^L_{\rho[x\mapsto n'][f\mapsto rec^-\ f\ \rho\ v/f]} = [\![e_2]\!]^L_{\rho[x\mapsto n']}$, because $f \notin FV(e_2)$. This concludes the proof of the proposition.

By Proposition 10.3.0.12, $rec^-\ f\ \rho\ v \in [\![(y :^L \mathbb{N})^- \to (p :^L x = S\ y)^- \to$

$[y/x]e_2]\!]^L_{\rho[x\mapsto n]}$. Thus, $[rec^-\ f\ \rho\ v/f]\rho \in [\![\Gamma, x :^L \mathbb{N}, f :^L (y :^L \mathbb{N})^- \to (p :^L x = S\ y)^- \to [y/x]e_2]\!]$ and we can finally conclude, $[rec^-\ f\ \rho\ v/f]\rho\ v \in [\![e_2]\!]^L_{\rho[x\mapsto n]}$. By CR-Pres, $rec^-\ f\ \rho\ v \in [\![e_2]\!]^L_{\rho[x\mapsto n]}$. Therefore, $rec^-\ f\ \rho\ v \in [\![(x :^L \mathbb{N})^- \to e_2]\!]^L_\rho$.

□

CHAPTER 11

DUALIZED LOGIC AND TYPE THEORY

In Section 9 we introduced Dualized Intuitionistic Logic (DIL) and its corresponding type theory Dualized Type Theory (DTT). Now we present the basic metatheory of both DIL and DTT. We start with proving consistency of DIL, and then prove completeness by reduction to Pinto and Uustalu's L. Then we move onto show type preservation and strong normalization for DTT. We show the latter using a version of Krivine's classical realizability by translating DIL into a classical logic.

11.1 Consistency of DIL

In this section we prove consistency of DIL with respect to Rauszer's Kripke semantics for BINT logic. All of the results in this section have been formalized in the Agda proof assistant[1]. We begin by first defining a Kripke frame.

Definition 11.1.0.1.

*A **Kripke frame** is a pair (W, R) of a set of worlds W, and a preorder R on W.*

Then we extend the notion of a Kripke frame to include an evaluation for atomic formulas resulting in a Kripke model.

Definition 11.1.0.2.

*A **Kripke model** is a tuple (W, R, V), such that, (W, R) is a Kripke frame, and V is a binary monotone relation on W and the set of atomic formulas of DIL.*

Now we can interpret formulas in a Kripke model as follows:

[1]Agda source code is available at https://github.com/heades/DIL-consistency

Definition 11.1.0.3.

The interpretation of the formulas of DIL in a Kripke model (W, R, V) *is defined by recursion on the structure of the formula as follows:*

$$\begin{array}{rcl} [\![\langle + \rangle]\!]_w & = & \top \\ [\![\langle - \rangle]\!]_w & = & \bot \\ [\![\mathbf{a}]\!]_w & = & V\, w\, \mathbf{a} \end{array} \qquad \begin{array}{rcl} [\![A \wedge_+ B]\!]_w & = & [\![A]\!]_w \wedge [\![B]\!]_w \\ [\![A \wedge_- B]\!]_w & = & [\![A]\!]_w \vee [\![B]\!]_w \\ [\![A \to_+ B]\!]_w & = & \forall w' \in W. R\, w\, w' \to [\![A]\!]_{w'} \to [\![B]\!]_{w'} \\ [\![A \to_- B]\!]_w & = & \exists w' \in W. R\, w'\, w \wedge \neg [\![A]\!]_{w'} \wedge [\![B]\!]_{w'} \end{array}$$

The interpretation of formulas really highlights the fact that implication is dual to coimplication. Monotonicity holds for this interpretation.

Lemma 11.1.0.4. *[Monotonicity] Suppose* (W, R, V) *is a Kripke model,* A *is some DIL formula, and* $w, w' \in W$*. Then* $R\, w\, w'$ *and* $[\![A]\!]_w$ *imply* $[\![A]\!]_{w'}$*.*

At this point we must set up the mathematical machinery which allows for the interpretation of sequents in a Kripke model. This will require the interpretation of graphs, and hence, nodes. We interpret nodes as worlds in the model using a function we call a node interpreter.

Definition 11.1.0.5.

Suppose (W, R, V) *is a Kripke model and* S *is a set of nodes of an abstract Kripke model* G*. Then a* ***node interpreter*** *on* S *is a function from* S *to* W*.*

Now using the node interpreter we can interpret edges as statements about the reachability relation in the model. Thus, the interpretation of a graph is just the conjunction of the interpretation of its edges.

Definition 11.1.0.6.

Suppose (W, R, V) *is a Kripke model,* G *is an abstract Kripke model, and* N *is a node interpreter on the set of nodes of* G*. Then the interpretation of* G *in the Kripke*

model is defined by recursion on the structure of the graph as follows:

$$\begin{array}{lcl} [\![\cdot]\!]_N & = & \top \\ [\![n_1 \preccurlyeq_+ n_2, G]\!]_N & = & R\,(N\,n_1)\,(N\,n_2) \wedge [\![G]\!]_N \\ [\![n_1 \preccurlyeq_- n_2, G]\!]_N & = & R\,(N\,n_2)\,(N\,n_1) \wedge [\![G]\!]_N \end{array}$$

Now we can prove that if a particular reachability judgment holds, then the interpretation of the nodes are reachable in the model.

Lemma 11.1.0.7. *[Reachability Interpretation] Suppose* (W, R, V) *is a Kripke model, and* $[\![G]\!]_N$ *for some abstract Kripke graph* G*. Then*

i. if $G \vdash n_1 \preccurlyeq_+ n_2$*, then* $R\,(N\,n_1)\,(N\,n_2)$*, and*

ii. if $G \vdash n_1 \preccurlyeq_- n_2$*, then* $R\,(N\,n_2)\,(N\,n_1)$*.*

We now have everything we need to interpret abstract Kripke models. The final ingredient to the interpretation of sequents is the interpretation of contexts.

Definition 11.1.0.8.

If F *is some meta-logical formula, we define* $p\,F$ *as follows:*

$$+\,F = F \quad \textit{and} \quad -\,F = \neg F.$$

Definition 11.1.0.9.

Suppose (W, R, V) *is a Kripke model,* Γ *is a context, and* N *is a node interpreter on the set of nodes in* Γ*. The interpretation of* Γ *in the Kripke model is defined by recursion on the structure of the context as follows:*

$$\begin{array}{lcl} [\![\cdot]\!]_N & = & \top \\ [\![p\,A \mathbin{@} n, \Gamma]\!]_N & = & p[\![A]\!]_{(N\,n)} \wedge [\![\Gamma]\!]_N \end{array}$$

Combining these interpretations results in the following definition of validity.

Definition 11.1.0.10.

Suppose (W, R, V) *is a Kripke model,* Γ *is a context, and* N *is a node interpreter on the set of nodes in* Γ*. The interpretation of sequents is defined as follows:*

$$[\![G; \Gamma \vdash p\ A\ @\ n]\!]_N \quad = \quad \textit{if } [\![G]\!]_N \textit{ and } [\![\Gamma]\!]_N, \textit{ then } p[\![A]\!]_{(N\,n)}.$$

Notice that in the definition of validity the graph G is interpreted as a set of constraints imposed on the set of Kripke models, thus reinforcing the fact that the graphs on sequents really are abstract Kripke models. Finally, using the previous definition of validity we can prove soundness.

Theorem 11.1.0.11. *[Soundness] Suppose* $G; \Gamma \vdash p\ A\ @\ n$*. Then for any Kripke model* (W, R, V) *and node interpreter* N *on* $|G|$*,* $[\![G; \Gamma \vdash p\ A\ @\ n]\!]_N$*.*

11.2 Completeness of DIL

In this section we prove that every derivable sequent in L can be translated to a derivable sequent of DIL. We will call a sequent in L a L-sequent and a sequent in DIL a DIL-sequent. Throughout this section we assume without loss of generality that all L-sequents have non-empty right-hand sides. That is, for every L-sequent, $\Gamma \vdash_G \Delta$, we assume that $\Delta \neq \cdot$. We do not loose generality because it is possible to prove that $\Gamma \vdash_G \cdot$ holds if and only if $\Gamma \vdash_G n : \bot$ for any node n (proof omitted).

Along the way, we will see admissibility of the analogues of the rules we mentioned in Section 9.2. The proof of consistency was with respect to DIL including the cut rule, but we prove completeness with respect to DIL where the general cut rule has been replaced with the following two inference rules, which can be seen as restricted instances of the cut rule:

$$\frac{p\ B\ @\ n' \in (\Gamma, \bar{p}\ A\ @\ n) \qquad G; \Gamma, \bar{p}\ A\ @\ n \vdash \bar{p}\ B\ @\ n'}{G; \Gamma \vdash p\ A\ @\ n}\ \text{axCut}$$

$$\frac{\bar{p}\ B\ @\ n' \in (\Gamma, \bar{p}\ A\ @\ n) \qquad G; \Gamma, \bar{p}\ A\ @\ n \vdash p\ B\ @\ n'}{G; \Gamma \vdash p\ A\ @\ n}\ \text{axCutBar}$$

These two rules are required for the crucial left-to-right lemma. This lemma depends on the following admissible rule:

Lemma 11.2.0.1. *[Weakening] If $G; \Gamma \vdash p\ B\ @\ n$ is derivable, then $G; \Gamma, p_1\ A\ @\ n_1 \vdash p_2\ B\ @\ n_1$ is derivable.*

Proof. This holds by straightforward induction on the assumed typing derivation. □

Note that we will use admissible rules as if they are inference rules of the logic throughout the sequel.

Lemma 11.2.0.2. *[Left-to-Right] If $G; \Gamma_1, \bar{p}\ A\ @\ n, \Gamma_2 \vdash \bar{p'}\ B\ @\ n'$ is derivable, then so is $G; \Gamma_1, \Gamma_2, p'\ B\ @\ n' \vdash p\ A\ @\ n$.*

Proof. Suppose $G; \Gamma_1, \bar{p}\ A\ @\ n, \Gamma_2 \vdash \bar{p'}\ B\ @\ n'$ is derivable and $\Gamma_3 =^{\text{def}} \Gamma_1, \bar{p}\ A\ @\ n, \Gamma_2$. Then we derive $G; \Gamma_1, \Gamma_2, p'\ B\ @\ n' \vdash p\ A\ @\ n$ as follows:

$$\frac{p'\ B\ @\ n' \in (\Gamma_3, p'\ B\ @\ n') \qquad \dfrac{G; \Gamma_3 \vdash \bar{p'}\ B\ @\ n'}{G; \Gamma_3, p'\ B\ @\ n' \vdash \bar{p'}\ B\ @\ n'}\ \text{Weakening}}{G; \Gamma_1, \Gamma_2, p'\ B\ @\ n' \vdash p\ A\ @\ n}\ \text{axCut}$$

Thus, we obtain our result. □

We mentioned DIL avoids analogs of a number of rules from L. To be able to translate every derivable sequent of L to DIL, we must show admissibility of those rules in DIL.

Lemma 11.2.0.3. *[Reflexivity] If* $G, m \preccurlyeq_{p'} m; \Gamma \vdash p\ A \mathbin{@} n$ *is derivable, then so is* $G; \Gamma \vdash p\ A \mathbin{@} n$.

Proof. This holds by a straightforward induction on the form of the assumed derivation. □

Lemma 11.2.0.4. *[Transitivity] If* $G, n_1 \preccurlyeq_{p'} n_3; \Gamma \vdash p\ A \mathbin{@} n$ *is derivable,* $n_1 \preccurlyeq_{p'} n_2 \in G$ *and* $n_2 \preccurlyeq_{p'} n_3 \in G$*, then* $G; \Gamma \vdash p\ A \mathbin{@} n$ *is derivable.*

Proof. This holds by a straightforward induction on the form of the assumed derivation. □

Lemma 11.2.0.5. *[AndL] If* $G; \Gamma, \bar{p}\ A \mathbin{@} n \vdash p\ B \mathbin{@} n$ *is derivable, then* $G; \Gamma \vdash p\ (A \wedge_{\bar{p}} B) \mathbin{@} n$ *is derivable.*

Proof. Suppose $G; \Gamma, \bar{p}\ A \mathbin{@} n \vdash p\ B \mathbin{@} n$ is derivable. By weakening we know

$$G; \Gamma, \bar{p}\ (A \wedge_{\bar{p}} B) \mathbin{@} n, \bar{p}\ B \mathbin{@} n, \bar{p}\ A \mathbin{@} n \vdash p\ B \mathbin{@} n.$$

Then $G; \Gamma \vdash p\ (A \wedge_{\bar{p}} B) \mathbin{@} n$ is derivable as follows:

$$\dfrac{\dfrac{\dfrac{D_1 \qquad D_2}{G; \Gamma, \bar{p}\ (A \wedge_{\bar{p}} B) \mathbin{@} n \vdash p\ B \mathbin{@} n}\ \text{Cut}}{G; \Gamma, \bar{p}\ (A \wedge_{\bar{p}} B) \mathbin{@} n \vdash p\ (A \wedge_{\bar{p}} B) \mathbin{@} n}\ \text{AndBar} \qquad \dfrac{\dfrac{}{G; \Gamma, \bar{p}\ (A \wedge_{\bar{p}} B) \mathbin{@} n \vdash \bar{p}\ (A \wedge_{\bar{p}} B) \mathbin{@} n}\ \text{ax}}{G; \Gamma \vdash p\ (A \wedge_{\bar{p}} B) \mathbin{@} n}\ \text{Cut}}$$

where we have the following subderivations:

D_0 :

$$\dfrac{\dfrac{}{G; \Gamma, \bar{p}\ (A \wedge_{\bar{p}} B) \mathbin{@} n, \bar{p}\ B \mathbin{@} n, p\ A \mathbin{@} n \vdash p\ A \mathbin{@} n}\ \text{ax}}{G; \Gamma, \bar{p}\ (A \wedge_{\bar{p}} B) \mathbin{@} n, \bar{p}\ B \mathbin{@} n, p\ A \mathbin{@} n \vdash p\ (A \wedge_{\bar{p}} B) \mathbin{@} n}\ \text{AndBar}$$

D_1 :

$$G;\Gamma, \bar{p}\,(A \wedge_{\bar{p}} B) \mathbin{@} n, \bar{p}\,B \mathbin{@} n, \bar{p}\,A \mathbin{@} n \vdash p\,B \mathbin{@} n$$

$$\dfrac{\dfrac{}{G;\Gamma, \bar{p}\,(A \wedge_{\bar{p}} B) \mathbin{@} n, \bar{p}\,B \mathbin{@} n, \bar{p}\,A \mathbin{@} n \vdash \bar{p}\,B \mathbin{@} n}\,\text{ax}}{G;\Gamma, \bar{p}\,(A \wedge_{\bar{p}} B) \mathbin{@} n, \bar{p}\,B \mathbin{@} n \vdash p\,A \mathbin{@} n}\,\text{Cut}$$

D_2 :

$$\dfrac{\dfrac{}{G;\Gamma, \bar{p}\,(A \wedge_{\bar{p}} B) \mathbin{@} n, \bar{p}\,B \mathbin{@} n, p\,A \mathbin{@} n \vdash \bar{p}\,(A \wedge_{\bar{p}} B) \mathbin{@} n}\,\text{ax} \qquad D_0}{G;\Gamma, \bar{p}\,(A \wedge_{\bar{p}} B) \mathbin{@} n, \bar{p}\,B \mathbin{@} n \vdash \bar{p}\,A \mathbin{@} n}\,\text{Cut}$$

□

Lemma 11.2.0.6. *[MonoL] If $G;\Gamma, p\,A \mathbin{@} n_1, p\,A \mathbin{@} n_2, \Gamma' \vdash p'\,B \mathbin{@} n'$ is derivable and $n_1 \preccurlyeq_p n_2 \in G$, then $G;\Gamma, p\,A \mathbin{@} n_1, \Gamma' \vdash p'\,B \mathbin{@} n'$ is derivable.*

Proof. This result easily follows by part one of Corollary 11.2.0.17, and contraction (Lemma 11.2.0.9). □

Lemma 11.2.0.7. *[MonoR] If $G;\Gamma, \bar{p}\,A \mathbin{@} n_1, \Gamma' \vdash p\,A \mathbin{@} n_2$ and $n_1 \preccurlyeq_p n_2 \in G$, then $G;\Gamma, \Gamma' \vdash p\,A \mathbin{@} n_2$ is derivable.*

Proof. Suppose $G;\Gamma, \bar{p}\,A \mathbin{@} n_1, \Gamma' \vdash p\,A \mathbin{@} n_2$ and $n_1 \preccurlyeq_p n_2 \in G$. Then by part one of monotonicity (Corollary 11.2.0.17) we know $G;\Gamma, \bar{p}\,A \mathbin{@} n_2, \Gamma' \vdash p\,A \mathbin{@} n_2$. Finally, we know by the axiom cut rule that $G;\Gamma, \Gamma' \vdash p\,A \mathbin{@} n_2$. □

Lemma 11.2.0.8. *[Exchange] If $G;\Gamma \vdash p\,A \mathbin{@} n$ is derivable and π is a permutation of Γ, then $G;\pi\,\Gamma \vdash p\,A \mathbin{@} n$ is derivable.*

Proof. This holds by a straightforward induction on the form of the assumed derivation. □

Note that we often leave the application of exchange implicit for readablity. Finally, we have the admissible for contraction.

Lemma 11.2.0.9. *[Contraction] If* $G; \Gamma, p\ A\ @\ n, p\ A\ @\ n, \Gamma' \vdash p'\ B\ @\ n'$, *then* $G; \Gamma, p\ A\ @\ n, \Gamma' \vdash p'\ B\ @\ n'$.

Proof. This holds by a straightforward induction on the form of the assumed derivation. □

The proofs of the previous admissible rules depend on a general monotonicity result (Lemma 11.2.0.16) for DIL. The proof of this will require some auxiliary mathematical machinery. First, we show that arbitrary edges can be weakened into the reachability judgment.

Lemma 11.2.0.10. *[Graph Weakening] If* $G \vdash n_1 \preccurlyeq_p^* n_2$, *then* $G, n_3 \preccurlyeq_{p'} n_4 \vdash n_1 \preccurlyeq_p^* n_2$.

Proof. This holds by a straightforward induction on the form of the assumed derivation. □

Now monotonicity is the notion of truths holding forever into the future, and falsehood holding forever into the past. That is, if a formula is true in some current world, then it must remain true, and if a forumla is false in the current world, then it remains false forever into the past. Thus, a false formula may eventually become true in the future, but a true formula cannot be true now and then become false in the future. To capture this notion syntactically we require the definition of a function which will push edges forward in the abstract Kripke model – the graph on the sequent. This function is ultimately used to construct the reachability constraint on implication in the general monotonicity lemma.

Definition 11.2.0.11.

We define the function raise *on abstract graphs as follows:*

$$\begin{array}{lcl}
\mathsf{raise}\,(n_1, n_2, \cdot) & = & \cdot \\
\mathsf{raise}\,(n_1, n_2, (n_1 \preccurlyeq_p m, G)) & = & n_2 \preccurlyeq_p m, \mathsf{raise}\,(n_1, n_2, G) \\
\mathsf{raise}\,(n_1, n_2, (m \preccurlyeq_p n_1, G)) & = & m \preccurlyeq_p n_2, \mathsf{raise}\,(n_1, n_2, G) \\
\mathsf{raise}\,(n_1, n_2, (m \preccurlyeq_p m', G)) & = & m \preccurlyeq_p m', \mathsf{raise}\,(n_1, n_2, G), \\
\quad \textit{where } m \not\equiv n_1 \textit{ and } m' \not\equiv n_1. & & \\
\mathsf{raise}\,(n_1, n_2, (m \preccurlyeq_{\bar{p}} m', G)) & = & m \preccurlyeq_{\bar{p}} m', \mathsf{raise}\,(n_1, n_2, G), \\
\quad \textit{where } m \not\equiv n_1 \textit{ and } m' \not\equiv n_1. & &
\end{array}$$

The following asserts that the orientation of an edge can be flipped, as long as the polarity is flipped.

Lemma 11.2.0.12. *[RelAssumFlip] If* $G_1, n_1 \preccurlyeq_p n_2, G_2 \vdash m \preccurlyeq_{p'} m'$*, then* $G_1, n_2 \preccurlyeq_{\bar{p}} n_1, G_2 \vdash m \preccurlyeq_{p'} m'$.

Proof. This is a proof by induction on the form of the assumed derivation. We only consider the case of the ax rule, because the remainder of the cases hold either trivially or by simple applicaitons of the induction hypothesis followed by the rule in the corresponding case.

Case.

$$\frac{}{G, n \preccurlyeq_{p''} n', G' \vdash n \preccurlyeq^*_{p''} n'}\ \text{ax}$$

We only consider the non-trivial case when $G, n \preccurlyeq_{p''} n', G' \equiv G_1, n_1 \preccurlyeq_p n_2, G_2$. This implies that $n \equiv m \equiv n_1$, $n' \equiv m' \equiv n_2$, and $p'' \equiv p' \equiv p$. It suffices to show $G_1, n_2 \preccurlyeq_{\bar{p}} n_1, G_2 \vdash n_1 \preccurlyeq^*_p n_2$. Clearly, we know by the rel_ax rule, $G_1, n_2 \preccurlyeq_{\bar{p}} n_1, G_2 \vdash n_2 \preccurlyeq^*_{\bar{p}} n_1$, and then by the rel_flip rule we know $G_1, n_2 \preccurlyeq_{\bar{p}} n_1, G_2 \vdash n_1 \preccurlyeq^*_p n_2$.

□

Using the raise function we can show that all nodes related to some node n_1 which is related to a node n_2 in some subgraph G_1, are also related to n_2. Now using the following result we will be able to raise the lowerbound on edges in a DIL-sequent, which will then be used to construct the reachability requirements for implication in the general monotonicity lemma (Lemma 11.2.0.16).

Lemma 11.2.0.13. *[Raising the Lower Bound] If* $G \vdash n_1 \preccurlyeq^*_p n_2$ *and* $G, G_1 \vdash m \preccurlyeq^*_{p'} m'$*, then* $G, \mathsf{raise}\,(n_1, n_2, G_1) \vdash m \preccurlyeq^*_{p'} m'$.

Proof. This is a proof by induction on the form of $G, G_1 \vdash m \preccurlyeq^*_{p'} m'$.

Case.

$$\frac{}{G', m \preccurlyeq_{p'} m', G'' \vdash m \preccurlyeq^*_{p'} m'}\ \text{ax}$$

Note that it is the case that $G', m \preccurlyeq_{p'} m', G'' \equiv G, G_1$. If $m \preccurlyeq_{p'} m' \in G$, then we obtain our result, so suppose $m \preccurlyeq_{p'} m' \in G_1$. Suppose $p \equiv p'$. Now if $m \not\equiv n_1$, then clearly, we obtain our result. Consider the case where $m \equiv n_1$. Then it suffices to show $G, \mathsf{raise}\,(n_1, n_2, G'_1), n_2 \preccurlyeq_p m', \mathsf{raise}\,(n_1, n_2, G''_1) \vdash n_1 \preccurlyeq^*_p m'$ where $G_1 \equiv G'_1, n_1 \preccurlyeq_p m', G''_1$. This holds by the following derivation:

$$\dfrac{G \vdash n_1 \preccurlyeq^*_p n_2 \qquad \dfrac{}{G, \mathsf{raise}\,(n_1, n_2, G'_1), n_2 \preccurlyeq_p m', \mathsf{raise}\,(n_1, n_2, G''_1) \vdash n_2 \preccurlyeq^*_p m'}\ \text{rel\_ax}}{G, \mathsf{raise}\,(n_1, n_2, G'_1), n_2 \preccurlyeq_p m', \mathsf{raise}\,(n_1, n_2, G''_1) \vdash n_1 \preccurlyeq^*_p m'}\ \text{rel\_trans}$$

Now suppose $p' \equiv \bar{p}$. if $m' \not\equiv n_1$, then clearly, we obtain our result. Consider the case where $m' \equiv n_1$. Then it suffices to show $G, \mathsf{raise}\,(n_1, n_2, G_1'), m \preccurlyeq_{\bar{p}} n_2, \mathsf{raise}\,(n_1, n_2, G_1'') \vdash m \preccurlyeq^*_{\bar{p}} n_1$ where $G_1 \equiv G_1', m \preccurlyeq_{\bar{p}} n_1, G_1''$. Finally, this case follows by applying the rel_trans rule.

Case.

$$\frac{}{G, G_1 \vdash m \preccurlyeq^*_{p'} m}\ \text{refl}$$

Note that in this case $m' \equiv m$. Our result follows from simply an application of the rel_refl rule.

Case.

$$\frac{G, G_1 \vdash m \preccurlyeq^*_{p'} m'' \qquad G, G_1 \vdash m'' \preccurlyeq^*_{p'} m'}{G, G_1 \vdash m \preccurlyeq^*_{p'} m'}\ \text{rel\_trans}$$

This case holds by two applications of the induction hypothesis followed by applying the rel_trans rule.

Case.

$$\frac{G, G_1 \vdash m' \preccurlyeq^*_{\bar{p}'} m}{G, G_1 \vdash m \preccurlyeq^*_{p'} m'}\ \text{flip}$$

It suffices to show $G, \mathsf{raise}\,(n_1, n_2, G) \vdash m \preccurlyeq^*_{p'} m'$. We know $G \vdash n_1 \preccurlyeq^*_{p} n_2$, so by the induction hypothesis we know $G, \mathsf{raise}\,(n_2, n_1, G) \vdash m' \preccurlyeq^*_{\bar{p}} m$. So it suffices to show that $G, \mathsf{raise}\,(n_2, n_1, G) \vdash m' \preccurlyeq^*_{\bar{p}'} m$ implies

$G, \mathsf{raise}\,(n_1, n_2, G) \vdash m \preccurlyeq^*_{p'} m'$, but this easily follows by repeated applications of Lemma 11.2.0.12.

□

We can now use the previous result to raise the lower bound on sequents.

Lemma 11.2.0.14. *[Graph Node Containment] If $G \vdash n_1 \preccurlyeq^*_p n_2$ and n_1 and n_2 are unique, then $n_1, n_2 \in |G|$.*

Proof. This holds by straightforward induction on the form of $G \vdash n_1 \preccurlyeq^*_p n_2$. □

Lemma 11.2.0.15. *[Raising the Lower Bound Logically] If $G, G_1, G'; \Gamma \vdash p\ A\ @\ n$ and $G, G' \vdash n_1 \preccurlyeq^*_p n_2$, then $G, \mathsf{raise}\,(n_1, n_2, G_1), G'; \Gamma \vdash p\ A\ @\ n$.*

Proof. This is a proof by induction on the form of $G, G_1, G'; \Gamma \vdash p\ A\ @\ n$. We assume with out loss of generality that $n_1 \in |G_1|$, and that $n_1 \not\equiv n_2$. If this is not the case then $\mathsf{raise}\,(n_1, n_2, G_1) = G_1$, and the result holds trivially.

Case.

$$\frac{G, G_1, G' \vdash n' \preccurlyeq^*_p n}{G, G_1, G'; \Gamma, p\ A\ @\ n' \vdash p\ A\ @\ n}\ \text{ax}$$

Clearly, if $G, G_1, G' \vdash n' \preccurlyeq^*_p n$, then $G, G', G_1 \vdash n' \preccurlyeq^*_p n$. Thus, this case follows by raising the lower bound (Lemma 11.2.0.13), and applying the ax rule.

Case.

$$\frac{}{G, G_1, G'; \Gamma \vdash p\ \langle p \rangle\ @\ n}\ \text{unit}$$

Trivial.

Case.

$$\frac{G, G_1, G'; \Gamma \vdash p\ A_1\ @\ n \qquad G, G_1, G'; \Gamma \vdash p\ A_2\ @\ n}{G, G_1, G'; \Gamma \vdash p\ (A_1 \wedge_p A_2)\ @\ n}\ \text{and}$$

This case holds by two applications of the induction hypothesis, and then applying the and rule.

Case.

$$\frac{G, G_1, G'; \Gamma \vdash p\ A_d\ @\ n}{G, G_1, G'; \Gamma \vdash p\ (A_1 \wedge_{\bar{p}} A_2)\ @\ n}\ \text{andBar}$$

Similar to the previous case.

Case.

$$\frac{\begin{array}{l} n' \notin |G, G_1, G'|, |\Gamma| \\ (G, G_1, G', n \preccurlyeq_p n'); \Gamma, p\ A_1\ @\ n' \vdash p\ A_2\ @\ n' \end{array}}{G, G_1, G'; \Gamma \vdash p\ (A_1 \rightarrow_p A_2)\ @\ n}\ \text{imp}$$

Since we know $n_1 \not\equiv n_2$, then by Lemma 11.2.0.14 we know $n_1, n_2 \in |G, G'|$. Thus, $n' \not\equiv n_1 \not\equiv n_2$. Now by the induction hypothesis we know $(G, \mathsf{raise}\,(n_1, n_2, G_1), G', n \preccurlyeq_p n'); \Gamma, p\ A_1\ @\ n' \vdash p\ A_2\ @\ n'$. This case then follows by the application of the imp rule to the former.

Case.

$$\frac{\begin{array}{l} G, G_1, G' \vdash n \preccurlyeq^*_{\bar{p}} n' \\ G, G_1, G'; \Gamma \vdash \bar{p}\ A_1 \mathbin{@} n' \qquad G, G_1, G'; \Gamma \vdash p\ A_2 \mathbin{@} n' \end{array}}{G, G_1, G'; \Gamma \vdash p\,(A_1 \to_{\bar{p}} A_2) \mathbin{@} n}\ \text{impBar}$$

Clearly, $G, G_1, G' \vdash n \preccurlyeq^*_{\bar{p}} n'$ implies $G, G', G_1 \vdash n \preccurlyeq^*_{\bar{p}} n'$, and by raising the lower bound (Lemma 11.2.0.13) we know $G, G', \mathsf{raise}\,(n_1, n_2, G_1) \vdash n \preccurlyeq^*_{\bar{p}} n'$ which implies $G, \mathsf{raise}\,(n_1, n_2, G_1), G' \vdash n \preccurlyeq^*_{\bar{p}} n'$.

Case.

$$\frac{p\ T' \mathbin{@} n' \in \Gamma \qquad G, G_1, G'; \Gamma, \bar{p}\ T \mathbin{@} n \vdash \bar{p}\ T' \mathbin{@} n'}{G, G_1, G'; \Gamma \vdash p\ T \mathbin{@} n}\ \text{axCut}$$

This case follows by a simple application of the induction hypothesis, and then reapplying the rule.

Case.

$$\frac{\bar{p}\ T' \mathbin{@} n' \in \Gamma \qquad G, G_1, G'; \Gamma, \bar{p}\ T \mathbin{@} n \vdash p\ T' \mathbin{@} n'}{G, G_1, G'; \Gamma \vdash p\ T \mathbin{@} n}\ \text{axCutBar}$$

Similar to the previous case.

□

Finally, we have everything we need to prove that general monotonicity is admissible in DIL. This implies the usual admissible monotonicity rule as a corollary.

Lemma 11.2.0.16. *[General Monotonicity] If* $G \vdash n_1 \preccurlyeq^*_{p_1} n'_1$, ..., $G \vdash n_i \preccurlyeq^*_{p_i} n'_i$, $G \vdash m \preccurlyeq^*_p m'$, *and* $G; \bar{p_1}\, A_1 \mathbin{@} n_1, \ldots, \bar{p_i}\, A_i \mathbin{@} n_i \vdash p\, B \mathbin{@} m$, *then* $G; \bar{p_1} A_1 @ n'_1, \ldots, \bar{p_i} A_i @ n'_i \vdash p\, B \mathbin{@} m'$.

Proof. This is a proof by induction on the form of $G; \bar{p_1}\, A_1 \mathbin{@} n_1, \ldots, \bar{p_i}\, A_i \mathbin{@} n_i \vdash p\, B \mathbin{@} m$. We assume without loss of generality that all of $n_1, n'_1 \ldots, n_i, n'_i$ are unique. Thus, they are all member of $|G|$.

Case.

$$\frac{G \vdash n_i \preccurlyeq^*_{\bar{p_i}} n'}{G; \bar{p_1}\, A_1 \mathbin{@} n_1, \ldots, \bar{p_i}\, A_i \mathbin{@} n_i \vdash \bar{p_i}\, A_i \mathbin{@} n'}\ \text{ax}$$

It must be the case that $p\, B \mathbin{@} m \equiv \bar{p_i}\, A \mathbin{@} n'$. In addition we know $G \vdash n_i \preccurlyeq^*_{p_i} n'_i$ and $G \vdash n' \preccurlyeq^*_{\bar{p_i}} m'$. It suffices to show $G; \bar{p_1} A_1 @ n'_1, \ldots, \bar{p_i} A_i @ n'_i \vdash \bar{p_i}\, A_i \mathbin{@} m'$. This is derivable as follows:

$$\dfrac{\dfrac{\dfrac{G \vdash n_i \preccurlyeq^*_{p_i} n'_i}{G \vdash n'_i \preccurlyeq^*_{\bar{p_i}} n_i}\ \text{rel\_flip} \qquad \dfrac{G \vdash n_i \preccurlyeq^*_{\bar{p_i}} n' \quad G \vdash n' \preccurlyeq^*_{\bar{p_i}} m'}{G \vdash n_i \preccurlyeq^*_{\bar{p_i}} m'}\ \text{rel\_trans}}{G \vdash n'_i \preccurlyeq^*_{\bar{p_i}} m'}\ \text{rel\_trans}}{G; \bar{p_1} A_1 @ n'_1, \ldots, \bar{p_i} A_i @ n'_i \vdash \bar{p_i}\, A_i \mathbin{@} m'}\ \text{ax}$$

Case.

$$\frac{}{G; \bar{p_1}\, A_1 \mathbin{@} n_1, \ldots, \bar{p_i}\, A_i \mathbin{@} n_i \vdash p\, \langle p \rangle \mathbin{@} m_1}\ \text{unit}$$

Trivial.

Case.

$$\frac{G; \bar{p_1}\ A_1 \mathbin{@} n_1,\ \ldots,\ \bar{p_i}\ A_i \mathbin{@} n_i \vdash p\ B_1 \mathbin{@} m \quad G; \bar{p_1}\ A_1 \mathbin{@} n_1,\ \ldots,\ \bar{p_i}\ A_i \mathbin{@} n_i \vdash p\ B_2 \mathbin{@} m}{G; \bar{p_1}\ A_1 \mathbin{@} n_1,\ \ldots,\ \bar{p_i}\ A_i \mathbin{@} n_i \vdash p\ (B_1 \wedge_p B_2) \mathbin{@} m}\ \text{and}$$

This case follows easily by applying the induction hypothesis to each premise and then applying the and rule.

Case.

$$\frac{G; \bar{p_1}\ A_1 \mathbin{@} n_1,\ \ldots,\ \bar{p_i}\ A_i \mathbin{@} n_i \vdash p\ B_d \mathbin{@} m}{G; \bar{p_1}\ A_1 \mathbin{@} n_1,\ \ldots,\ \bar{p_i}\ A_i \mathbin{@} n_i \vdash p\ (B_1 \wedge_{\bar{p}} B_2) \mathbin{@} m}\ \text{andBar}$$

This case follows easily by the induction hypothesis and then applying andBar.

Case.

$$\frac{n' \notin |G|, |\bar{p_1}\ A_1 \mathbin{@} n_1,\ \ldots,\ \bar{p_i}\ A_i \mathbin{@} n_i| \quad (G, m_1 \preccurlyeq_p n'); \bar{p_1}\ A_1 \mathbin{@} n_1,\ \ldots,\ \bar{p_i}\ A_i \mathbin{@} n_i, p\ B_1 \mathbin{@} n' \vdash p\ B_2 \mathbin{@} n'}{G; \bar{p_1}\ A_1 \mathbin{@} n_1,\ \ldots,\ \bar{p_i}\ A_i \mathbin{@} n_i \vdash p\ (B_1 \rightarrow_p B_2) \mathbin{@} m}\ \text{imp}$$

We know by assumption $G \vdash n_1 \preccurlyeq^*_{p_1} n'_1, \ldots, G \vdash n_i \preccurlyeq^*_{p_i} n'_i$, and by graph weakening (Lemma 11.2.0.10) $G, m \preccurlyeq_p n' \vdash n_1 \preccurlyeq^*_{p_1} n'_1, \ldots, G, m \preccurlyeq_p n' \vdash n_i \preccurlyeq^*_{p_i} n'_i$. We also know by applying the rel_refl rule that $G, m \preccurlyeq_p n' \vdash n' \preccurlyeq^*_{\bar{p}} n'$ and $G, m \preccurlyeq_p n' \vdash n' \preccurlyeq^*_p n'$. Thus, by the induction hypothesis we know $(G, m \preccurlyeq_p n'); \bar{p_1} A_1 @ n'_1, \ldots, \bar{p_i} A_i @ n'_i, p\ B_1 \mathbin{@} n' \vdash p\ B_2 \mathbin{@} n'$. Now we can raise the lower bound logically (Lemma 11.2.0.15) with $G_1 \equiv m \preccurlyeq_p n'$ and the assumption $G \vdash m \preccurlyeq^*_p m'$ to obtain

$(G, \mathsf{raise}\,(m, m', m \preccurlyeq_p n')); \bar{p_1} A_1 @ n'_1, \ldots, \bar{p_i} A_i @ n'_i, p\ B_1 @ n' \vdash p\ B_2 @ n'$, but this is equivalent to $(G, m \preccurlyeq_p n'); \bar{p_1} A_1 @ n'_1, \ldots, \bar{p_i} A_i @ n'_i, p\ B_1 @ n' \vdash p\ B_2 @ n'$. Finally, using the former, we obtain our result by applying the imp rule.

Case.

$$\frac{\begin{array}{l} G \vdash m \preccurlyeq^*_{\bar{p}} n' \\ G; \bar{p_1}\ A_1 @ n_1, \ldots, \bar{p_i}\ A_i @ n_i \vdash \bar{p}\ B_1 @ n' \\ G; \bar{p_1}\ A_1 @ n_1, \ldots, \bar{p_i}\ A_i @ n_i \vdash p\ B_2 @ n' \end{array}}{G; \bar{p_1}\ A_1 @ n_1, \ldots, \bar{p_i}\ A_i @ n_i \vdash p\ (B_1 \rightarrow_{\bar{p}} B_2) @ m}\ \text{impBar}$$

We can easily derive $G \vdash m' \preccurlyeq^*_{\bar{p}} n'$ as follows:

$$\dfrac{\dfrac{\dfrac{G \vdash m \preccurlyeq^*_{\bar{p}} n'}{G \vdash n' \preccurlyeq^*_p m}\ \text{rel\_flip} \qquad G \vdash m \preccurlyeq^*_p m'}{G \vdash n' \preccurlyeq^*_p m'}\ \text{rel\_trans}}{G \vdash m' \preccurlyeq^*_{\bar{p}} n'}\ \text{rel\_flip}$$

This case then follows by applying the induction hypothesis twice to both $G; \bar{p_1}\ A_1 @ n_1, \ldots, \bar{p_i}\ A_i @ n_i, \bar{p}\ B_{\bar{d}} @ m \vdash \bar{p}\ B_1 @ n'$ and $G; \bar{p_1}\ A_1 @ n_1, \ldots, \bar{p_i}\ A_i @ n_i, \bar{p}\ B_{\bar{d}} @ m \vdash p\ B_2 @ n'$ using the assumptions $G \vdash n_1 \preccurlyeq^*_{p_1} n'_1, \ldots, G \vdash n_i \preccurlyeq^*_{p_i} n'_i$, $G \vdash m \preccurlyeq^*_p m'$, and the fact that we know $G \vdash n' \preccurlyeq^*_p n'$ and $G \vdash n' \preccurlyeq^*_{\bar{p}} n'$.

Case.

$$\frac{\begin{array}{l} \bar{p_j}\ A_j @ n_j \in (\bar{p_1}\ A_1 @ n_1, \ldots, \bar{p_i}\ A_i @ n_i) \\ G; \bar{p_1}\ A_1 @ n_1, \ldots, \bar{p_i}\ A_i @ n_i, \bar{p}\ B @ m \vdash p_j\ A_j @ n_j \end{array}}{G; \bar{p_1}\ A_1 @ n_1, \ldots, \bar{p_i}\ A_i @ n_i \vdash p\ B @ m}\ \text{axCut}$$

We know by assumption that $G \vdash n_1 \preccurlyeq^*_{p_1} n'_1, \ldots, G \vdash n_i \preccurlyeq^*_{p_i} n'_i$, and $G \vdash m \preccurlyeq^*_p m'$. In particular, we know $G \vdash n_j \preccurlyeq^*_{p_j} n'_j$. It is also the

case that if $\bar{p_j}\, A_j \mathbin{@} n_j \in (\bar{p_1}\, A_1 \mathbin{@} n_1, \ldots, \bar{p_i}\, A_i \mathbin{@} n_i)$, then $\bar{p_j}\, A_j \mathbin{@} n_j' \in (\bar{p_1} A_1 @ n', \ldots, \bar{p_i} A_i @ n_i')$. This case then follows by applying the induction hypothesis to $G; \bar{p_1}\, A_1 \mathbin{@} n_1, \ldots, \bar{p_i}\, A_i \mathbin{@} n_i, \bar{p}\, B \mathbin{@} m \vdash p_j\, A_j \mathbin{@} n_j$, to obtain, $G; \bar{p_1} A_1 @ n_1', \ldots, \bar{p_i} A_i @ n_i', \bar{p}\, B \mathbin{@} m_1' \vdash p_j\, A_j \mathbin{@} n_j'$, followed by applying the axCut rule.

Case.

$$\frac{\begin{array}{l}\bar{p_j}\, A_j \mathbin{@} n_j \in (\bar{p_1}\, A_1 \mathbin{@} n_1, \ldots, \bar{p_i}\, A_i \mathbin{@} n_i) \\ G; \bar{p_1}\, A_1 \mathbin{@} n_1, \ldots, \bar{p_i}\, A_i \mathbin{@} n_i, \bar{p}\, B \mathbin{@} m \vdash p_j\, A_j \mathbin{@} n_j\end{array}}{G; \bar{p_1}\, A_1 \mathbin{@} n_1, \ldots, \bar{p_i}\, A_i \mathbin{@} n_i \vdash p\, B \mathbin{@} m}\ \text{axCutBar}$$

Similar to the previous case.

□

The proofs of the admissible rules MonoL and MonoR depend on the following monotonicity result.

Corollary 11.2.0.17. *[Monotonicity] Suppose* $G \vdash n_1 \preccurlyeq_p^* n_2$. *Then*

i. if $G; \Gamma, \bar{p}\, A \mathbin{@} n_1, \Gamma' \vdash p'\, B \mathbin{@} n'$, *then* $G; \Gamma, \bar{p}\, A \mathbin{@} n_2, \Gamma' \vdash p'\, B \mathbin{@} n'$, *and*

ii. if $G; \Gamma \vdash p\, A \mathbin{@} n_1$, *then* $G; \Gamma \vdash p\, A \mathbin{@} n_2$.

Proof. This result follows easily from Lemma 11.2.0.16. □

We now have everything we need to prove that every derivable sequent of L can be translated to a derivable sequent in DIL. The proof technique we use here does not provide an algorithm taking a sequent of L and yielding a derivable sequent in

DIL. Such an algorithm would have to choose a particular conclusion in the L-sequent to be the active formula of the DIL-sequent, but this is quite difficult. Instead we show that all possible conclusions in the L-sequent can be chosen to be active and yield a derivable DIL-sequent.

Using the translation of formulas given in Definition 9.2.0.1 we can easily translate contexts. Right contexts Γ in L are translated to positive hypotheses, while left contexts, not including the formula chosen as the active formula, are translated into negative hypotheses. The following definition defines the translation of both types of contexts.

Definition 11.2.0.18.

We extend the translation of formulas to contexts Γ *and* Δ *with respect to a polarity* p *as follows:*

$$\begin{aligned} \ulcorner . \urcorner^p &= . \\ \ulcorner n : A, \Gamma \urcorner^p &= p \ulcorner A \urcorner @ n, \ulcorner \Gamma \urcorner^p \end{aligned}$$

Abstract Kripke models are straightforward to translate.

Definition 11.2.0.19.

We define the translation of graphs G *in L to graphs in DIL as follows:*

$$\begin{aligned} \ulcorner . \urcorner &= . \\ \ulcorner (n_1, n_2), G \urcorner &= n_1 \preccurlyeq_+ n_2, \ulcorner G \urcorner \end{aligned}$$

The previous definition implies the following result:

Lemma 11.2.0.20. *[Reachability] If* $n_1 G n_2$*, then* $\ulcorner G \urcorner \vdash n_1 \preccurlyeq_+^* n_2$*.*

The translation of a derivable L-sequent is a DIL-sequent which requires a particular formula as the active formula. We define such a translation in the following

definition.

Definition 11.2.0.21.

An activation of a derivable L-sequent $\Gamma \vdash_G \Delta$ *is a DIL-sequent* $\ulcorner G \urcorner; \ulcorner \Gamma \urcorner^+, \ulcorner \Delta_1, \Delta_2 \urcorner^- \vdash + \ulcorner A \urcorner @ n$, *where* $\Delta = \Delta_1, n : A, \Delta_2$.

Finally, the following theorem is the main result showing that any activation of a derivable L-sequent is derivable in DIL.

Theorem 11.2.0.22. *[Containment of L in DIL] If* $\ulcorner G \urcorner; \Gamma' \vdash + A @ n$ *is an activation of the derivable L-sequent* $\Gamma \vdash_G \Delta$, *then* $\ulcorner G \urcorner; \Gamma' \vdash + A @ n$ *is derivable.*

Proof. This is a proof by induction on the form of the sequent $\Gamma \vdash_G \Delta$.

Case.

$$\frac{\Gamma \vdash_{G,(n,n)} \Delta}{\Gamma \vdash_G \Delta} \text{ refl}$$

We know by the induction hypothesis that every activation of $\Gamma \vdash_{G,(n,n)} \Delta$ is derivable. Suppose that $\ulcorner G, (n, n) \urcorner; \Gamma' \vdash + A @ n$ is an arbitrary activation, where $\ulcorner \Delta \urcorner^- \equiv \ulcorner \Delta_1 \urcorner^-, - A @ n, \ulcorner \Delta_2 \urcorner^-$ and $\Gamma' \equiv \ulcorner \Delta_1 \urcorner^-, \ulcorner \Delta_2 \urcorner^-$. This is equivalent to $\ulcorner G \urcorner, n \preccurlyeq_+ n; \Gamma' \vdash + A @ n$, and by the admissible rule for reflexivity (Lemma 11.2.0.3) we have $\ulcorner G \urcorner; \Gamma' \vdash + A @ n$.

Case.

$$\frac{\begin{array}{l} n_1 G n_2 \\ n_2 G n_3 \\ \Gamma \vdash_{G,(n_1,n_3)} \Delta \end{array}}{\Gamma \vdash_G \Delta} \text{ Trans}$$

We know by the induction hypothesis that every activation of $\Gamma \vdash_{G,(n_1,n_3)} \Delta$ is derivable. Suppose that $\ulcorner G, (n_1, n_3)\urcorner; \Gamma' \vdash + A \mathbin{@} n$ is an arbitrary activation, where $\ulcorner \Delta \urcorner^- \equiv \ulcorner \Delta_1 \urcorner^-, - A \mathbin{@} n, \ulcorner \Delta_2 \urcorner^-$ and $\Gamma' \equiv \ulcorner \Delta_1 \urcorner^-, \ulcorner \Delta_2 \urcorner^-$. This sequent is equivalent to $\ulcorner G \urcorner, n_1 \preccurlyeq_+ n_3; \Gamma' \vdash + A \mathbin{@} n$. Furthermore, it is clear by definition that if $n_1 G n_2$ and $n_2 G n_3$, then $n_1 \preccurlyeq_+ n_2 \in \ulcorner G \urcorner$ and $n_2 \preccurlyeq_+ n_3 \in \ulcorner G \urcorner$. Thus, by the admissible rule for transitivity (Lemma 11.2.0.4) we have $\ulcorner G \urcorner; \Gamma' \vdash + A \mathbin{@} n$, and we obtain our result.

Case.

$$\frac{}{\Gamma, n : A \vdash_G n : A, \Delta}\ \text{hyp}$$

It suffices to show that every activation of $\Gamma, n : A \vdash_G n : A, \Delta$ is derivable. Clearly, $\ulcorner G \urcorner; \ulcorner \Gamma \urcorner^+, + \ulcorner A \urcorner \mathbin{@} n, \ulcorner \Delta \urcorner^- \vdash + \ulcorner A \urcorner \mathbin{@} n$ is a activation of $\Gamma, n : A \vdash_G n : A, \Delta$. In addition, it is derivable:

$$\dfrac{\dfrac{\dfrac{}{\ulcorner G \urcorner \vdash n \preccurlyeq_+^* n}\ \text{Refl}}{\ulcorner G \urcorner; \ulcorner \Gamma \urcorner^+, \ulcorner \Delta \urcorner^-, + \ulcorner A \urcorner \mathbin{@} n \vdash + \ulcorner A \urcorner \mathbin{@} n}\ \text{ax}}{\ulcorner G \urcorner; \ulcorner \Gamma \urcorner^+, + \ulcorner A \urcorner \mathbin{@} n, \ulcorner \Delta \urcorner^- \vdash + \ulcorner A \urcorner \mathbin{@} n}\ \text{Exchange}$$

In the previous derivation we make use of the exchange rule which is admissible by Lemma 11.2.0.8.

Now consider any other activation $\ulcorner G \urcorner; \Gamma' \vdash + B \mathbin{@} n'$. It must be the case that $\Gamma' = \ulcorner \Gamma \urcorner^+, + A \mathbin{@} n, \ulcorner \Delta_1 \urcorner^-, - \ulcorner A \urcorner \mathbin{@} n, \ulcorner \Delta_2 \urcorner^-$ for some Δ_1 and Δ_2. This sequent is then derivable as follows:

$$\dfrac{\dfrac{\dfrac{}{\ulcorner G\urcorner \vdash n \preccurlyeq_+^* n}\,\text{Refl}}{\ulcorner G\urcorner;\ulcorner\Gamma\urcorner^+,\ulcorner\Delta_1\urcorner^-,\ulcorner\Delta_2\urcorner^-,-B\,@\,n',+A\,@\,n\vdash+\ulcorner A\urcorner\,@\,n}\,\text{ax}}{\ulcorner G\urcorner;\ulcorner\Gamma\urcorner^+,+A\,@\,n,\ulcorner\Delta_1\urcorner^-,\ulcorner\Delta_2\urcorner^-,-B\,@\,n'\vdash+\ulcorner A\urcorner\,@\,n}\,\text{Exchange}}{\ulcorner G\urcorner;\ulcorner\Gamma\urcorner^+,+A\,@\,n,\ulcorner\Delta_1\urcorner^-,-\ulcorner A\urcorner\,@\,n,\ulcorner\Delta_2\urcorner^-\vdash+B\,@\,n'}\,\text{L-to-R}$$

Thus, we obtain our result.

Case.

$$\dfrac{\begin{array}{l}n_1Gn_2\\ \Gamma,n_1:A,n_2:A\vdash_G\Delta\end{array}}{\Gamma,n_1:A\vdash_G\Delta}\,\text{monL}$$

Certainly, if n_1Gn_2, then $n_1 \preccurlyeq_+ n_2 \in \ulcorner G\urcorner$. We know by the induction hypothesis that all activations of $\Gamma, n_1 : A, n_2 : A \vdash_G \Delta$ are derivable. Suppose $\ulcorner G\urcorner;\Gamma' \vdash +B\,@\,n$ is an arbitrary activation. Then it must be the case that $\Gamma' \equiv \ulcorner\Gamma\urcorner^+,+A\,@\,n_1,+A\,@\,n_2,\ulcorner\Delta_1\urcorner^-,\ulcorner\Delta_2\urcorner^-$, where $\ulcorner\Delta\urcorner^- \equiv \ulcorner\Delta_1\urcorner^-,-B\,@\,n,\ulcorner\Delta_2\urcorner^-$. Now we apply the monoL admissible rule (Lemma 11.2.0.6) to obtain $\ulcorner G\urcorner;\ulcorner\Gamma\urcorner^+,+A\,@\,n_1,\ulcorner\Delta_1\urcorner^-,\ulcorner\Delta_2\urcorner^-\vdash+B\,@\,n$, which is an arbitrary activation of $\Gamma, n_1 : A \vdash_G \Delta$.

Case.

$$\dfrac{\begin{array}{l}n_1Gn_2\\ \Gamma\vdash_G n_1:A,n_2:A,\Delta\end{array}}{\Gamma\vdash_G n_2:A,\Delta}\,\text{monR}$$

If n_1Gn_2, then $n_1 \preccurlyeq_+ n_2 \in \ulcorner G\urcorner$. We know by the induction hypothesis that all activations of $\Gamma \vdash_G n_1 : A, n_2 : A, \Delta$ are derivable. In particular, the activation (modulo exchange (Lemma 11.2.0.8)) $\ulcorner G\urcorner;\ulcorner\Gamma\urcorner^+,\ulcorner\Delta\urcorner^-,-A\,@\,n_1\vdash$

$+A \mathbin{@} n_2$ is derivable. It suffices to show that $\ulcorner G \urcorner; \ulcorner \Gamma \urcorner^{+}, \ulcorner \Delta \urcorner^{-} \vdash +A \mathbin{@} n_2$. This follows from the monoR admissible rule (Lemma 11.2.0.7). Finally, any other activation of $\Gamma \vdash_G n_2 : A, \Delta$ can be activated into $\ulcorner G \urcorner; \ulcorner \Gamma \urcorner^{+}, \ulcorner \Delta \urcorner^{-} \vdash +A \mathbin{@} n_2$ (Lemma 11.2.0.2). Thus, we obtain our result.

Case.

$$\frac{\Gamma \vdash_G \Delta}{\Gamma, n' : \top \vdash_G \Delta}\ \text{trueL}$$

We know by the induction hypothesis that all activations of $\Gamma \vdash_G \Delta$ are derivable. Suppose $\ulcorner G \urcorner; \Gamma' \vdash +A @ n$ is an arbitrary activation of $\Gamma \vdash_G \Delta$. Then it must be the case that $\Gamma' = \ulcorner \Gamma \urcorner^{+}, \ulcorner \Delta_1 \urcorner^{-}, \ulcorner \Delta_2 \urcorner^{-}$, where $\ulcorner \Delta \urcorner^{-} \equiv \ulcorner \Delta_1 \urcorner^{-}, -A @ n, \ulcorner \Delta_2 \urcorner^{-}$. Now by weakening (Lemma 11.2.0.1) we know $\ulcorner G \urcorner; \Gamma', + \langle + \rangle @ n' \vdash +A \mathbin{@} n$, and by exchange (Lemma 11.2.0.8) $\ulcorner G \urcorner; + \langle + \rangle \mathbin{@} n', \Gamma' \vdash +A \mathbin{@} n$, which is exactly an arbitrary activation of $\Gamma, n' : \top \vdash_G \Delta$.

Case.

$$\frac{}{\Gamma \vdash_G n : \top, \Delta}\ \text{trueR}$$

It suffices to show that every activation of $\Gamma \vdash_G n : \top, \Delta$ is derivable. Consider the activation $\ulcorner G \urcorner; \ulcorner \Gamma \urcorner^{+}, \ulcorner \Delta \urcorner^{-} \vdash +\ulcorner \top \urcorner \mathbin{@} n$. This is easily derivable by applying the unit rule. Now any other activation of $\Gamma \vdash_G n : \top, \Delta$ implies $\ulcorner G \urcorner; \ulcorner \Gamma \urcorner^{+}, \ulcorner \Delta \urcorner^{-} \vdash +\ulcorner \top \urcorner @ n$ is derivable by Lemma 11.2.0.2, and hence, are derivable.

Case.

$$\frac{}{\Gamma, n :\bot \vdash_G \Delta}\ \text{falseL}$$

Suppose $\ulcorner G \urcorner; \ulcorner \Gamma \urcorner^+, + \ulcorner \bot \urcorner \mathbin{@} n, \ulcorner \Delta_1 \urcorner^-, \ulcorner \Delta_2 \urcorner^- \vdash + A \mathbin{@} n'$ is an arbitrary activation of $\Gamma, n :\bot \vdash_G \Delta$, where $\ulcorner \Delta \urcorner^- \equiv \ulcorner \Delta_1 \urcorner^-, - A \mathbin{@} n', \ulcorner \Delta_2 \urcorner^-$. We can easily see that by definition $\ulcorner G \urcorner; \ulcorner \Gamma \urcorner^+, + \ulcorner \bot \urcorner \mathbin{@} n, \ulcorner \Delta_1 \urcorner^-, \ulcorner \Delta_2 \urcorner^- \vdash + A \mathbin{@} n'$ is equivalent to $\ulcorner G \urcorner; \ulcorner \Gamma \urcorner^+, + \langle - \rangle \mathbin{@} n, \ulcorner \Delta_1 \urcorner^-, \ulcorner \Delta_2 \urcorner^- \vdash + A \mathbin{@} n'$. We can derive the latter as follows:

$$\frac{+ \langle - \rangle \mathbin{@} n \in \Gamma', - A \mathbin{@} n' \qquad \dfrac{}{\ulcorner G \urcorner; \Gamma', - A \mathbin{@} n' \vdash - \langle - \rangle \mathbin{@} n}\ \text{unit}}{\ulcorner G \urcorner; \ulcorner \Gamma \urcorner^+, + \langle - \rangle \mathbin{@} n, \ulcorner \Delta_1 \urcorner^-, \ulcorner \Delta_2 \urcorner^- \vdash + A \mathbin{@} n'}\ \text{axCutBar}$$

In the previous derivation $\Gamma' \equiv \ulcorner \Gamma \urcorner^+, + \langle - \rangle \mathbin{@} n, \ulcorner \Delta_1 \urcorner^-, \ulcorner \Delta_2 \urcorner^-$. Thus, any activation of $\Gamma, n :\bot \vdash_G \Delta$ is derivable.

Case.

$$\frac{\Gamma \vdash_G \Delta}{\Gamma \vdash_G n' :\bot, \Delta}\ \text{falseR}$$

We know by the induction hypothesis that all activations of $\Gamma \vdash_G \Delta$ are derivable. Suppose $\ulcorner G \urcorner; \Gamma' \vdash + A \mathbin{@} n$ is an arbitrary activation of $\Gamma \vdash_G \Delta$. Then it must be the case that $\Gamma' = \ulcorner \Gamma \urcorner^+, \ulcorner \Delta \urcorner^-$. Now by weakening (Lemma 11.2.0.1) we know $\ulcorner G \urcorner; \Gamma', - \langle - \rangle \mathbin{@} n' \vdash + A \mathbin{@} n$, and by the left-to-right lemma (Lemma 11.2.0.2) $\ulcorner G \urcorner; \Gamma', - A \mathbin{@} n \vdash + \langle - \rangle \mathbin{@} n'$, which – modulo exchange – is equivalent to $\ulcorner G \urcorner; \ulcorner \Gamma \urcorner^+, \ulcorner \Delta \urcorner^- \vdash + \ulcorner \bot \urcorner \mathbin{@} n'$. Thus, we obtain

our result.

Case.

$$\frac{\Gamma, n : T_1, n : T_2 \vdash_G \Delta}{\Gamma, n : T_1 \wedge T_2 \vdash_G \Delta}\ \text{andL}$$

We know by the induction hypothesis that all activations of $\Gamma, n : T_1, n : T_2 \vdash_G \Delta$ are derivable. In particular, we know $\ulcorner G \urcorner; \ulcorner \Gamma \urcorner^+, + T_1 @ n, + T_2 @ n, \ulcorner \Delta_1 \urcorner^-, \ulcorner \Delta_2 \urcorner^- \vdash + A @ n'$ where $\ulcorner \Delta \urcorner^- = \ulcorner \Delta_1 \urcorner^-, -A@n', \ulcorner \Delta_2 \urcorner^-$. Using exchange we know $\ulcorner G \urcorner; \ulcorner \Gamma \urcorner^+, \ulcorner \Delta_1 \urcorner^-, \ulcorner \Delta_2 \urcorner^-, + T_1 @ n, + T_2 @ n \vdash + A @ n'$, and by the left-to-right lemma $\ulcorner G \urcorner; \ulcorner \Gamma \urcorner^+, \ulcorner \Delta_1 \urcorner^-, \ulcorner \Delta_2 \urcorner^-, + T_1 @ n, - A @ n' \vdash - T_2 @ n$, and finally by one more application of exchange $\ulcorner G \urcorner; \ulcorner \Gamma \urcorner^+, \ulcorner \Delta_1 \urcorner^-, \ulcorner \Delta_2 \urcorner^-, - A @ n', + T_1 @ n \vdash - T_2 @ n$. At this point we know $\ulcorner G \urcorner; \ulcorner \Gamma \urcorner^+, \ulcorner \Delta_1 \urcorner^-, \ulcorner \Delta_2 \urcorner^-, - A @ n' \vdash - T_1 \wedge_+ T_2 @ n$ by the using the admissible andL rule (Lemma 11.2.0.5). Now using left-to-right $\ulcorner G \urcorner; \ulcorner \Gamma \urcorner^+, \ulcorner \Delta_1 \urcorner^-, \ulcorner \Delta_2 \urcorner^-, + T_1 \wedge_+ T_2 @ n \vdash + A @ n'$ is derivable. Lastly, by exchange $\ulcorner G \urcorner; \ulcorner \Gamma \urcorner^+, + T_1 \wedge_+ T_2 @ n, \ulcorner \Delta_1 \urcorner^-, \ulcorner \Delta_2 \urcorner^- \vdash + A @ n'$ is derivable, which is clearly and an arbitrary activation of $\Gamma, n : T_1 \wedge T_2 \vdash_G \Delta$.

Case.

$$\frac{\Gamma \vdash_G n : A, \Delta \quad \Gamma \vdash_G n : B, \Delta}{\Gamma \vdash_G n : A \wedge B, \Delta}\ \text{andR}$$

We know by the induction hypothesis that all activations of $\Gamma \vdash_G n : A, \Delta$ and $\Gamma \vdash_G n : B, \Delta$ are derivable. In particular, $\ulcorner G \urcorner; \ulcorner \Gamma \urcorner^+, \ulcorner \Delta \urcorner^- \vdash + A @ n$

and $\ulcorner G \urcorner; \ulcorner \Gamma \urcorner^{+}, \ulcorner \Delta \urcorner^{-} \vdash +B \,@\, n$ are derivable. Now by applying the and rule we obtain $\ulcorner G \urcorner; \ulcorner \Gamma \urcorner^{+}, \ulcorner \Delta \urcorner^{-} \vdash +A \wedge_{+} B @ n$, which is a particular activation of $\Gamma \vdash_G n : A \wedge B, \Delta$. Finally, consider any other activation, then that sequent implies $\ulcorner G \urcorner; \ulcorner \Gamma \urcorner^{+}, \ulcorner \Delta \urcorner^{-} \vdash +A \wedge_{+} B \,@\, n$ is derivable using Lemma 11.2.0.2. Thus, we obtain our result.

Case.

$$\frac{\begin{array}{c}\Gamma, n : A \vdash_G \Delta \\ \Gamma, n : A \vdash_G \Delta\end{array}}{\Gamma, n : A \vee B \vdash_G \Delta}\ \text{disjL}$$

We know by the induction hypothesis that all activations of $\Gamma, n : A \vdash_G \Delta$ and $\Gamma, n : B \vdash_G \Delta$ are derivable. So suppose $\ulcorner G \urcorner; \ulcorner \Gamma \urcorner^{+}, + \ulcorner A \urcorner \,@\, n, \ulcorner \Delta' \urcorner^{-} \vdash +C \,@\, n'$ and $\ulcorner G \urcorner; \ulcorner \Gamma \urcorner^{+}, + \ulcorner B \urcorner \,@\, n, \ulcorner \Delta' \urcorner^{-} \vdash +E \,@\, n''$ are particular activations, where $\ulcorner \Delta \urcorner^{-} \equiv \ulcorner \Delta_1 \urcorner^{-}, -C \,@\, n', \ulcorner \Delta_2 \urcorner^{-}, -E \,@\, n'', \ulcorner \Delta_3 \urcorner^{-}$, and $\ulcorner \Delta' \urcorner^{-} \equiv \ulcorner \Delta_1 \urcorner^{-}, \ulcorner \Delta_2 \urcorner^{-}, \ulcorner \Delta_3 \urcorner^{-}$. By exchange (Lemma 11.2.0.8) we know $\ulcorner G \urcorner; \ulcorner \Gamma \urcorner^{+}, \ulcorner \Delta' \urcorner^{-}, + \ulcorner A \urcorner \,@\, n \vdash +C \,@\, n'$ and $\ulcorner G \urcorner; \ulcorner \Gamma \urcorner^{+}, \ulcorner \Delta' \urcorner^{-}, + \ulcorner B \urcorner \,@\, n \vdash +E \,@\, n''$. Now by the left-to-right lemma (Lemma 11.2.0.2) we know $\ulcorner G \urcorner; \ulcorner \Gamma \urcorner^{+}, \ulcorner \Delta' \urcorner^{-}, -C \,@\, n' \vdash - \ulcorner A \urcorner \,@\, n$ and $\ulcorner G \urcorner; \ulcorner \Gamma \urcorner^{+}, \ulcorner \Delta' \urcorner^{-}, -E \,@\, n'' \vdash - \ulcorner B \urcorner \,@\, n$, and by applying weakening (and exchange) we know $\ulcorner G \urcorner; \ulcorner \Gamma \urcorner^{+}, \ulcorner \Delta' \urcorner^{-}, -C @ n', -E @ n'' \vdash - \ulcorner A \urcorner @ n$ and $\ulcorner G \urcorner; \ulcorner \Gamma \urcorner^{+}, \ulcorner \Delta' \urcorner^{-}, -C @ n', -E \,@\, n'' \vdash - \ulcorner B \urcorner \,@\, n$. At this point we can apply the and rule to obtain $\ulcorner G \urcorner; \ulcorner \Gamma \urcorner^{+}, \ulcorner \Delta' \urcorner^{-}, -C \,@\, n', -E \,@\, n'' \vdash - \ulcorner A \urcorner \wedge_{-} \ulcorner B \urcorner \,@\, n$ to which we can apply the left-to-right lemma to and obtain

$\ulcorner G \urcorner; \ulcorner \Gamma \urcorner^{+}, \ulcorner \Delta' \urcorner^{-}, -E \mathbin{@} n'', +\ulcorner A \urcorner \wedge_{-} \ulcorner B \urcorner \mathbin{@} n \vdash +C \mathbin{@} n'$. Finally, we can apply exchange again to obtain $\ulcorner G \urcorner; \ulcorner \Gamma \urcorner^{+}, +\ulcorner A \urcorner \wedge_{-} \ulcorner B \urcorner \mathbin{@} n, \ulcorner \Delta' \urcorner^{-}, -E \mathbin{@} n'' \vdash +C \mathbin{@} n'$, which – modulo exchange – is an arbitrary activation of $\Gamma, n : A \vee B \vdash_G \Delta$. Thus, we obtain our result.

Case.

$$\frac{\Gamma \vdash_G x : T_1, x : T_2, \Delta}{\Gamma \vdash_G x : T_1 \vee T_2, \Delta}\ \text{disjR}$$

This case is similar to the case of andR case, except, it makes use of the andBar rule.

Case.

$$\frac{\begin{array}{l} n_1 G n_2 \\ \Gamma \vdash_G n_2 : T_1, \Delta \\ \Gamma, n_2 : T_2 \vdash_G \Delta \end{array}}{\Gamma, n_1 : T_1 \supset T_2 \vdash_G \Delta}\ \text{impL}$$

We know by the induction hypothesis that all activations of $\Gamma \vdash_G y : T_1, \Delta$ and $\Gamma, y : T_2 \vdash_G \Delta$ are derivable. In particular, we know $\ulcorner G \urcorner; \ulcorner \Gamma \urcorner^{+}, \ulcorner \Delta \urcorner^{-} \vdash +\ulcorner T_1 \urcorner @ n_2$ is derivable, and so is $\ulcorner G \urcorner; \ulcorner \Gamma \urcorner^{+}, \ulcorner \Delta \urcorner^{-} \vdash -\ulcorner T_2 \urcorner @ n_2$. The latter being derivable by applying the induction hypothesis followed by exchange (Lemma 11.2.0.8) and the left-to-right lemma (Lemma 11.2.0.2). We know $n_1 G n_2$ by assumption and so by Lemma 11.2.0.20 $\ulcorner G \urcorner \vdash n_1 \preccurlyeq^{*}_{+} n_2$. Thus, by applying the impBar rule we obtain $\ulcorner G \urcorner; \ulcorner \Gamma \urcorner^{+}, \ulcorner \Delta \urcorner^{-} \vdash -\ulcorner T_1 \urcorner \rightarrow_{+} \ulcorner T_2 \urcorner @ n_1$. At this point we can apply left-to-right to the previous sequent and obtain

and activation of $\Gamma, n_1 : T_1 \supset T_2 \vdash_G \Delta$, thus we obtain our result.

Case.

$$\frac{\begin{array}{l} n_2 \notin |G|, |\Gamma|, |\Delta| \\ \Gamma, n_2 : T_1 \vdash_{G \cup \{(n_1, n_2)\}} n_2 : T_2, \Delta \end{array}}{\Gamma \vdash_G n_1 : T_1 \supset T_2, \Delta} \text{ impR}$$

This case follows the same pattern as the previous cases. We know by the induction hypothesis that all activations of $\Gamma, y : T_1 \vdash_{G \cup \{(x,y)\}} y : T_2, \Delta$ are derivable. In particular, $\ulcorner G \urcorner, n_1 \preccurlyeq_+ n_2; \ulcorner \Gamma \urcorner^+, + \ulcorner T_1 \urcorner @ n_2, \ulcorner \Delta \urcorner^- \vdash + \ulcorner T_2 \urcorner @ n_2$ is derivable. By exchange (Lemma 11.2.0.8)

$\ulcorner G \urcorner, n_1 \preccurlyeq_+ n_2; \ulcorner \Gamma \urcorner^+, \ulcorner \Delta \urcorner^-, + \ulcorner T_1 \urcorner @ n_2 \vdash + \ulcorner T_2 \urcorner @ n_2$ is derivable, and by applying the imp rule we obtain $\ulcorner G \urcorner; \ulcorner \Gamma \urcorner^+, \ulcorner \Delta \urcorner^- \vdash + \ulcorner T_1 \urcorner \rightarrow_+ \ulcorner T_2 \urcorner @ n_1$, which is a particular activation of $\Gamma \vdash_G n_1 : T_1 \supset T_2, \Delta$. Note that in the previous application of imp we use the fact that if $n_2 \notin |G|, |\Gamma|, |\Delta|$, then $n_2 \notin |\ulcorner G \urcorner|, |\ulcorner \Gamma \urcorner^+, \ulcorner \Delta \urcorner^-|$. Lastly, any other activation of $\Gamma \vdash_G n_1 : T_1 \supset T_2, \Delta$ implies $\ulcorner G \urcorner; \ulcorner \Gamma \urcorner^+, \ulcorner \Delta \urcorner^- \vdash + \ulcorner T_1 \urcorner \rightarrow_+ \ulcorner T_2 \urcorner @ n_1$ is derivable by the left-to-right lemma, and hence is derivable.

Case.

$$\frac{\begin{array}{l} n_1 \notin |G|, |\Gamma|, |\Delta| \\ \Gamma, n_1 : T_1 \vdash_{G \cup \{(n_1, n_2)\}} n_1 : T_2, \Delta \end{array}}{\Gamma, n_2 : T_1 \prec T_2 \vdash_G \Delta} \text{ subL}$$

We know by the induction hypothesis that all activation of

$\Gamma, n_1 : T_1 \vdash_{G \cup \{(n_1, n_2)\}} n_1 : T_2, \Delta$ are derivable. In particular, $\ulcorner G \urcorner, n_1 \preccurlyeq_+$

$n_2; \ulcorner \Gamma \urcorner^+, + \ulcorner T_1 \urcorner @ n_1, \ulcorner \Delta \urcorner^- \vdash + \ulcorner T_2 \urcorner @ n_1$ is derivable. By exchange (Lemma 11.2.0.8) $\ulcorner G \urcorner, n_1 \preccurlyeq_+ n_2; \ulcorner \Gamma \urcorner^+, \ulcorner \Delta \urcorner^-, + \ulcorner T_1 \urcorner @ n_1 \vdash + \ulcorner T_2 \urcorner @ n_1$ is derivable. Now by the left-to-right lemma we know $\ulcorner G \urcorner, n_1 \preccurlyeq_+ n_2; \ulcorner \Gamma \urcorner^+, \ulcorner \Delta \urcorner^-, - \ulcorner T_2 \urcorner @ n_1 \vdash - \ulcorner T_1 \urcorner @ n_1$, and by assumption we know $y \notin |G|, |\Gamma|, |\Delta|$ which implies $n_1 \notin |\ulcorner G \urcorner|, |\ulcorner \Gamma \urcorner^+, \ulcorner \Delta \urcorner^-|$ is derivable. Thus, by applying the imp rule we know $\ulcorner G \urcorner, n_1 \preccurlyeq_+ n_2; \ulcorner \Gamma \urcorner^+, \ulcorner \Delta \urcorner^- \vdash - \ulcorner T_2 \urcorner \rightarrow_- \ulcorner T_1 \urcorner @ n_2$ is derivable. Clearly, this is a particular activation of $\Gamma, n_2 : T_1 \prec T_2 \vdash_G \Delta$, and any other activation implies $\ulcorner G \urcorner, n_1 \preccurlyeq_+ n_2; \ulcorner \Gamma \urcorner^+, \ulcorner \Delta \urcorner^- \vdash - \ulcorner T_2 \urcorner \rightarrow_- \ulcorner T_1 \urcorner @ n_2$ is derivable by the left-to-right lemma, and hence are derivable.

Case.

$$\frac{\begin{array}{l} yGx \\ \Gamma \vdash_G y : T_1, \Delta \\ \Gamma, y : T_2 \vdash_G \Delta \end{array}}{\Gamma \vdash_G x : T_1 \prec T_2, \Delta} \text{ subR}$$

This case follows in the same way as the case for impL, except the particular activation of $\Gamma, y : T_2 \vdash_G \Delta$ has to have the active formulas such that the rule impBar can be applied.

□

Corollary 11.2.0.23. *[Completeness] DIL is complete.*

Proof. Completeness of L is proved in [106], and by Theorem 11.2.0.22 we know that every derivable sequent of L is derivable in DIL. □

11.3 Metatheory of DTT

We now present the basic metatheory of DTT, starting with type preservations. We begin with the inversion lemma which is necessary for proving type preservation.

Lemma 11.3.0.1. *[Inverstion]*

i. *If* $G; \Gamma, x : p\ A\ @\ n, \Gamma' \vdash x : p\ A\ @\ n'$, *then* $G \vdash n\ \preccurlyeq_p^*\ n'$.

ii. *If* $G; \Gamma \vdash (t_1, t_2) : p\ (A \wedge_p B)@n$, *then* $G; \Gamma \vdash t_1 : p\ A@n$ *and* $G; \Gamma \vdash t_2 : p\ B@n$.

iii. *If* $G; \Gamma \vdash \mathbf{in}_d\ t : p\ (A_1 \wedge_{\bar{p}} A_2)\ @\ n$, *then* $G; \Gamma \vdash t : p\ A_d\ @\ n$.

iv. *If* $G; \Gamma \vdash \lambda x.t : p\ (A \rightarrow_p B)@n$, *then* $(G, n\ \preccurlyeq_p\ n'); \Gamma, x : p\ A@n' \vdash t : p\ B@n'$ *for any* $n' \notin |G|, |\Gamma|$.

v. *If* $G; \Gamma \vdash \langle t_1, t_2 \rangle : p\ (A \rightarrow_{\bar{p}} B)\ @\ n$, *then* $G \vdash n\ \preccurlyeq_{\bar{p}}^*\ n'$, $G; \Gamma \vdash t_1 : \bar{p}\ A\ @\ n'$, *and* $G; \Gamma \vdash t_2 : p\ B\ @\ n'$ *for some node* n'.

Proof. Each case of the above lemma holds by a trivial proof by induction on the assumed typing derivation. □

The lemmas node substitution for typing and substitution for typing are essential for the cases of type preservation that reduce a top-level redex. Node substitution, denoted $[n_1/n_2]n$, is defined as follows:

$$\begin{array}{lcl} [n_1/n_2]n_2 & = & n_1 \\ [n_1/n_2]n & = & n \text{ where } n \text{ is distinct from } n_2 \end{array}$$

The following lemmas are necessary in the proof of node substitution for typing.

Lemma 11.3.0.2. *[Node Renaming] If* $G_1, G_2 \vdash n_1\ \preccurlyeq_p^*\ n_3$, *then for any nodes* n_4 *and* n_5, *where* n_5 *is distinct from* n_1 *and* n_3, *we have* $[n_4/n_5]G_1, [n_4/n_5]G_2 \vdash n_1\ \preccurlyeq_p^*\ n_3$.

Proof. This is a proof by induction on the assumed reachability derivation. Throughout each case suppose we have nodes n_4 and n_5.

Case.

$$\frac{}{G, n_1 \preccurlyeq_p n_3, G' \vdash n_1 \preccurlyeq_p^* n_3}\ \text{ax}$$

Trivial.

Case.

$$\frac{}{G_1, G_2 \vdash n \preccurlyeq_p^* n}\ \text{refl}$$

Trivial.

Case.

$$\frac{G_1, G_2 \vdash n_1 \preccurlyeq_p^* n' \qquad G_1, G_2 \vdash n' \preccurlyeq_p^* n_3}{G_1, G_2 \vdash n_1 \preccurlyeq_p^* n_3}\ \text{trans}$$

By the induction hypothesis we know that for any nodes n_4' and n_5' that $[n_4'/n_5']G_1, [n_4'/n_5']G_2 \vdash n_1 \preccurlyeq_p^* n'$, and for any nodes n_4'' and n_5'' that $[n_4''/n_5'']G_1, [n_4''/n_5'']G_2 \vdash n' \preccurlyeq_p^* n_3$. Choose n_4 for n_4' and n_4'' and n_5 for n_5' and n_5'' to obtain $[n_4/n_5]G_1, [n_4/n_5]G_2 \vdash n_1 \preccurlyeq_p^* n'$ and $[n_4/n_5]G_1, [n_4/n_5]G_2 \vdash n' \preccurlyeq_p^* n_3$. Finally, this case follows by reapplying the rule to the previous two facts.

Case.

$$\frac{G \vdash n' \preccurlyeq^*_{\bar{p}} n}{G \vdash n \preccurlyeq^*_p n'} \text{ flip}$$

Similar to the previous case.

□

Lemma 11.3.0.3. *[Node Substitution for Reachability] If* $G, n_1 \preccurlyeq_{p_1} n_2, G' \vdash n_4 \preccurlyeq^*_p n_5$ *and* $G, G' \vdash n_1 \preccurlyeq^*_{p_1} n_3$*, then* $[n_3/n_2]G, [n_3/n_2]G' \vdash [n_3/n_2]n_4 \preccurlyeq^*_p [n_3/n_2]n_5$*.*

Proof. This is a proof by induction on the form of the assumed reachability derivation. Throughout the following cases we assume $G, G' \vdash n_1 \preccurlyeq^*_{p_1} n_3$ holds.

Case.

$$\frac{}{G_1, n_4 \preccurlyeq_p n_5, G_2 \vdash n_4 \preccurlyeq^*_p n_5} \text{ ax}$$

Suppose $G_1, n_4 \preccurlyeq_p n_5, G_2 = G, n_1 \preccurlyeq_{p_1} n_2, G'$. Then either $n_1 \preccurlyeq_{p_1} n_2 \in G_1$, $n_1 \preccurlyeq_{p_1} n_2 \in G_2$, or $n_1 \preccurlyeq_{p_1} n_2 \equiv n_4 \preccurlyeq_p n_5$. Suppose $n_1 \preccurlyeq_{p_1} n_2 \in G_1$, then $G_1 = G'_1, n_1 \preccurlyeq_p n_2, G''_1$. Then it is easy to see that $[n_3/n_2](G'_1, G''_1, n_4 \preccurlyeq_p n_5), [n_3/n_2]G_2 \vdash [n_3/n_2]n_4 \preccurlyeq^*_p [n_3/n_2]n_5$ is derivable by applying ax. The case where $n_1 \preccurlyeq_{p_1} n_2 \in G_2$ is similar.

Now suppose $n_1 \preccurlyeq_{p_1} n_2 \equiv n_4 \preccurlyeq_p n_5$. Then we know by assumption that

$$\frac{}{G_1, n_1 \preccurlyeq_p n_2, G_2 \vdash n_1 \preccurlyeq^*_p n_2} \text{ ax}$$

Then it suffices to show $[n_3/n_2]G_1, [n_3/n_2]G_2 \vdash [n_3/n_2]n_1 \preccurlyeq_p^* [n_3/n_2]n_2$, which is equivalent to $[n_3/n_2]G_1, [n_3/n_2]G_2 \vdash [n_3/n_2]n_1 \preccurlyeq_p^* n_3$. Now if n_1 is equivalent to n_2, then $[n_3/n_2]G_1, [n_3/n_2]G_2 \vdash [n_3/n_2]n_1 \preccurlyeq_p^* n_3$ holds by reflexivity, and if n_1 is distinct from n_2, then $[n_3/n_2]G_1, [n_3/n_2]G_2 \vdash [n_3/n_2]n_1 \preccurlyeq_p^* n_3$ is equivalent to $[n_3/n_2]G_1, [n_3/n_2]G_2 \vdash n_1 \preccurlyeq_p^* n_3$. We know by assumption that $G, G' \vdash n_1 \preccurlyeq_{p_1}^* n_3$ holds, which is equivalent to $G_1, G_2 \vdash n_1 \preccurlyeq_p^* n_3$. Now if n_3 is equal to n_2, then $[n_3/n_2]G_1, [n_3/n_2]G_2 \vdash n_1 \preccurlyeq_p^* n_3$ is equivalent to $G_1, G_2 \vdash n_1 \preccurlyeq_p^* n_3$. So suppose n_3 is distinct from n_2, then by Lemma 11.3.0.2 we know $[n_3/n_2]G_1, [n_3/n_2]G_2 \vdash n_1 \preccurlyeq_p^* n_3$.

Case.

$$\frac{}{G, n_1 \preccurlyeq_{p_1} n_2, G' \vdash n \preccurlyeq_p^* n}\ \text{refl}$$

Trivial.

Case.

$$\frac{G, n_1 \preccurlyeq_{p_1} n_2, G' \vdash n_4 \preccurlyeq_p^* n_6 \qquad G \vdash n_6 \preccurlyeq_p^* n_5}{G, n_1 \preccurlyeq_{p_1} n_2, G' \vdash n_4 \preccurlyeq_p^* n_5}\ \text{trans}$$

This case by applying the induction to each premise, and then reapplying the rule.

Case.

$$\frac{G, n_1 \preccurlyeq_{p_1} n_2, G' \vdash n_5 \preccurlyeq_{\bar{p}}^* n_4}{G, n_1 \preccurlyeq_{p_1} n_2, G' \vdash n_4 \preccurlyeq_p^* n_5}\ \text{flip}$$

This case holds by applying the induction hypothesis to the premise, and then reapplying the rule.

□

Lemma 11.3.0.4. *[Node Substitution for Typing] If* $G, n_1 \preccurlyeq_{p_1} n_2, G'; \Gamma \vdash t : p_2\ A \mathbin{@} n_3$ *and* $G, G' \vdash n_1 \preccurlyeq^*_{p_1} n_4$*, then* $[n_4/n_2]G, [n_4/n_2]G'; [n_4/n_2]\Gamma \vdash t : p_2\ A \mathbin{@} [n_4/n_2]n_3$.

Proof. This is a proof by induction on the form of the assumed typing derivation. Throughout each of the following cases we assume $G, G' \vdash n_1 \preccurlyeq^*_{p_1} n_4$ holds.

Case.

$$\frac{G, n_1 \preccurlyeq_{p_1} n_2, G' \vdash n \preccurlyeq^*_p n_3}{G, n_1 \preccurlyeq_{p_1} n_2, G'; \Gamma_1, y : p_2\ A \mathbin{@} n, \Gamma_2 \vdash y : p_2\ A \mathbin{@} n_3}\ \text{ax}$$

First, by node substitution for reachability (Lemma 11.3.0.3) we know $[n_4/n_2]G, [n_4/n_2]G' \vdash [n_4/n_2]n \preccurlyeq^*_p [n_4/n_2]n_3$. Thus, by applying the ax rule we may derive $[n_4/n_2]G, [n_4/n_2]G'; [n_4/n_2]\Gamma_1, y : p_2\ A \mathbin{@} [n_4/n_2]n, [n_4/n_2]\Gamma_2 \vdash y : p_2\ A \mathbin{@} [n_4/n_2]n_3$.

Case.

$$\frac{}{G, n_1 \preccurlyeq_{p_1} n_2, G'; \Gamma \vdash \mathbf{triv} : p_2\ \langle p_2 \rangle \mathbin{@} n_3}\ \text{Unit}$$

Trivial.

Case.

$$\frac{G, n_1 \preccurlyeq_{p_1} n_2; \Gamma \vdash t_1 : p_2\, A_1 \mathbin{@} n_3 \qquad G, n_1 \preccurlyeq_{p_1} n_2; \Gamma \vdash t_2 : p_2\, A_2 \mathbin{@} n_3}{G, n_1 \preccurlyeq_{p_1} n_2; \Gamma \vdash (t_1, t_2) : p_2\, (A_1 \wedge_{p_2} A_2) \mathbin{@} n_3}\ \text{And}$$

This case holds by applying the induction hypothesis to each premise, and then reapplying the rule.

Case.

$$\frac{G, n_1 \preccurlyeq_{p_1} n_2; \Gamma \vdash t' : p_2\, A_d \mathbin{@} n_3}{G, n_1 \preccurlyeq_{p_1} n_2; \Gamma \vdash \mathbf{in}_d\, t' : p_2\, (A_1 \wedge_{\bar{p_2}} A_2) \mathbin{@} n_3}\ \text{AndBar}$$

This case holds by applying the induction hypothesis to the premise, and then reapplying the rule.

Case.

$$\frac{\begin{array}{l} n' \notin |G, n_1 \preccurlyeq_{p_1} n_2, G'|, |\Gamma| \\ (G, n_1 \preccurlyeq_{p_1} n_2, G', n_3 \preccurlyeq_{p} n'); \Gamma, x : p_2\, A_1 \mathbin{@} n' \vdash t' : p_2\, A_2 \mathbin{@} n' \end{array}}{G, n_1 \preccurlyeq_{p_1} n_2, G'; \Gamma \vdash \lambda x.t' : p_2\, (A_1 \to_{p_2} A_2) \mathbin{@} n_3}\ \text{Imp}$$

First, if $n' \notin |G, n_1 \preccurlyeq_{p_1} n_2, G'|, |\Gamma|$, then $n' \notin |G, G'|, |\Gamma|$. Furthermore, we know that $[n_4/n_2]n' \notin |[n_4/n_2]G, [n_4/n_2]G'|, |[n_4/n_2]\Gamma|$, because we know n' is distinct from n_2 by assumption, and if n' is equal to n_4, then $n' \notin |G, n_1 \preccurlyeq_{p_1} n_2, G'|, |\Gamma|$ implies that n_1 must also be n_4, because we know by assumption that $G, G' \vdash n_1 \preccurlyeq^*_{p_1} n_4$ which could only be derivide by reflexivity since $n' \notin |G, G'|, |\Gamma|$, but we know by assumption that $n' \notin |G, n_1 \preccurlyeq_{p_1} n_2, G'|, |\Gamma|$ which implies that n' must be distinct from n_1, and hence a contradiction, thus n' cannot be n_4. Therefore, we know $n' \notin |[n_4/n_2]G, [n_4/n_2]G'|, |[n_4/n_2]\Gamma|$.

By the induction hypothesis we know

$[n_4/n_2](G, G', n_3 \preccurlyeq_p n'); [n_4/n_2]\Gamma, x : p_2\, A_1 \mathbin{@} [n_4/n_2]n' \vdash t' : p_2\, A_2 \mathbin{@} [n_4/n_2]n'$

which is equivalent to

$([n_4/n_2]G, [n_4/n_2]G', [n_4/n_2]n_3 \preccurlyeq_p n'); [n_4/n_2]\Gamma, x : p_2\, A_1 @ n' \vdash t' : p_2\, A_2 @ n'.$

Finally, this case follows by applying the Imp rule using

$n' \notin |[n_4/n_2]G, [n_4/n_2]G'|, |[n_4/n_2]\Gamma|$ and the previous fact.

Case.

$$\frac{\begin{array}{c} G, n_1 \preccurlyeq_{p_1} n_2, G' \vdash n_3 \preccurlyeq^*_{\bar{p}_2} n' \\ G, n_1 \preccurlyeq_{p_1} n_2, G'; \Gamma \vdash t_1 : \bar{p}_2\, A_1 \mathbin{@} n' \\ G, n_1 \preccurlyeq_{p_1} n_2, G'; \Gamma \vdash t_2 : p_2\, A_2 \mathbin{@} n' \end{array}}{G, n_1 \preccurlyeq_{p_1} n_2, G'; \Gamma \vdash \langle t_1, t_2 \rangle : p_2\, (A_1 \to_{\bar{p}_2} A_2) \mathbin{@} n_3}\ \text{ImpBar}$$

We now by assumption that $G, G' \vdash n_1 \preccurlyeq^*_{p_1} n_4$ holds. So by node substitution for reachability (Lemma 11.3.0.3) we know $[n_4/n_2]G, [n_4/n_2]G' \vdash [n_4/n_2]n_3 \preccurlyeq^*_{\bar{p}_2} [n_4/n_2]n'$. Now by the induction hypothesis we know

$[n_4/n_2]G, [n_4/n_2]G'; [n_4/n_2]\Gamma \vdash t_1 : \bar{p}_2\, A_1 \mathbin{@} [n_4/n_2]n'$ and

$[n_4/n_2]G, [n_4/n_2]G'; [n_4/n_2]\Gamma \vdash t_2 : p_2\, A_2 \mathbin{@} [n_4/n_2]n'$. This case then follows by applying the rule ImBar to the previous three facts.

Case.

$$\frac{\begin{array}{c} G, n_1 \preccurlyeq_{p_1} n_2, G'; \Gamma, y : \bar{p}_2\, A \mathbin{@} n_3 \vdash t_1 : + C \mathbin{@} n \\ G, n_1 \preccurlyeq_{p_1} n_2, G'; \Gamma, y : \bar{p}_2\, A \mathbin{@} n_3 \vdash t_2 : - C \mathbin{@} n \end{array}}{G, n_1 \preccurlyeq_{p_1} n_2, G'; \Gamma \vdash \nu\, x.t_1 \bullet t_2 : p_2\, A \mathbin{@} n_3}\ \text{Cut}$$

This case follows by applying the induction hypothesis to each premise, and then reapplying the rule.

□

Lemma 11.3.0.5. *[Substitution for Typing] If* $G; \Gamma \vdash t_1 : p_1\, A \mathbin{@} n_1$ *and* $G; \Gamma, x : p_1\, A \mathbin{@} n_1 \vdash t_2 : p_2\, B \mathbin{@} n_2$*, then* $G; \Gamma \vdash [t_1/x]t_2 : p_2\, B \mathbin{@} n_2$*.*

Proof. This proof holds by a straightforward induction on the second assumed typing relation.

Case.

$$\frac{G \vdash n \preccurlyeq_p^* n'}{G; \Gamma_1, y : p\, C \mathbin{@} n, \Gamma_2 \vdash y : p\, C \mathbin{@} n'}\ \text{ax}$$

Trivial.

Case.

$$\frac{}{G; \Gamma' \vdash \mathbf{triv} : p\, \langle p \rangle \mathbin{@} n}\ \text{Unit}$$

Trivial.

Case.

$$\frac{G; \Gamma' \vdash t_1' : p\, A \mathbin{@} n \qquad G; \Gamma' \vdash t_2' : p\, B \mathbin{@} n}{G; \Gamma' \vdash (t_1', t_2') : p\, (C_1 \wedge_p C_2) \mathbin{@} n}\ \text{And}$$

Suppose $\Gamma' \equiv \Gamma, x : p_1\, B \mathbin{@} n_1$. Then this case follows from applying the induction hypothesis to each premise and then reapplying the rule.

Case.

$$\frac{G; \Gamma' \vdash t : p\ C_d\ @\ n}{G; \Gamma' \vdash \mathbf{in}_d\ t : p\ (C_1 \wedge_{\bar{p}}\ C_2)\ @\ n}\ \text{AndBar}$$

Suppose $\Gamma' \equiv \Gamma, x : p_1\ B\ @\ n_1$. Then this case follows from applying the induction hypothesis to the premise and then reapplying the rule.

Case.

$$\frac{\begin{array}{l} n' \notin |G|, |\Gamma| \\ (G, n \preccurlyeq_p n'); \Gamma', x : p\ C_1\ @\ n' \vdash t : p\ C_2\ @\ n' \end{array}}{G; \Gamma' \vdash \lambda x.t : p\ (C_1 \rightarrow_p\ C_2)\ @\ n}\ \text{Imp}$$

Similarly to the previous case.

Case.

$$\frac{\begin{array}{l} G \vdash n\ \preccurlyeq^*_{\bar{p}}\ n' \\ G; \Gamma' \vdash t'_1 : \bar{p}\ C_1\ @\ n' \qquad G; \Gamma' \vdash t'_2 : p\ C_2\ @\ n' \end{array}}{G; \Gamma' \vdash \langle t'_1, t'_2 \rangle : p\ (C_1 \rightarrow_{\bar{p}}\ C_2)\ @\ n}\ \text{ImpBar}$$

Suppose $\Gamma' \equiv \Gamma, x : p_1\ B\ @\ n_1$. Then this case follows from applying the induction hypothesis to each premise and then reapplying the rule.

Case.

$$\frac{\begin{array}{l} G; \Gamma', y : \bar{p}\ C\ @\ n \vdash t'_1 : +\ C'\ @\ n' \\ G; \Gamma', y : \bar{p}\ C\ @\ n \vdash t'_2 : -\ C'\ @\ n' \end{array}}{G; \Gamma' \vdash \nu\, x.t'_1 \bullet t'_2 : p\ C\ @\ n}\ \text{Cut}$$

Similarly to the previous case.

□

Finally, we prove type preservation.

Lemma 11.3.0.6. *[Type Preservation] If* $G; \Gamma \vdash t : p\, A \mathbin{@} n$, *and* $t \rightsquigarrow t'$, *then* $G; \Gamma \vdash t' : p\, A \mathbin{@} n$.

Proof. This is a proof by induction on the form of the assumed typing derivation. We only consider non-trivial cases. All the other cases either follow directly from assumptions or are similar to the cases we provide below.

Case.

$$\frac{G; \Gamma, x : \bar{p}\, A \mathbin{@} n \vdash t_1 : + B \mathbin{@} n' \quad G; \Gamma, x : \bar{p}\, A \mathbin{@} n \vdash t_2 : - B \mathbin{@} n'}{G; \Gamma \vdash \nu\, x.t_1 \bullet t_2 : p\, A \mathbin{@} n}\ \text{Cut}$$

The interesting cases are the ones where the assumed cut is a redex itself, otherwise this case holds by the induction hypothesis. Thus, we case split on the form of this redex.

Case. Suppose $\nu\, x.t_1 \bullet t_2 \equiv \nu\, x.\lambda y.t_1' \bullet \langle t_2', t_2'' \rangle$, thus, $t_1 \equiv \lambda y.t_1'$ and $t_2 \equiv \langle t_2', t_2'' \rangle$.

This then implies that $B \equiv B_1 \to_+ B_2$ for some B_1 and B_2. Then

$$t \equiv \nu\, x.t_1 \bullet t_2 \equiv \nu\, x.\lambda y.t_1' \bullet \langle t_2', t_2'' \rangle \rightsquigarrow \nu\, x.[t_2'/y]t_1' \bullet t_2'' \equiv t'.$$

Now by inversion we know the following:

(1) $G, (n' \preccurlyeq_+ n''); \Gamma, x : \bar{p}\, A \mathbin{@} n, y : + B_1 \mathbin{@} n'' \vdash t_1' : + B_2 \mathbin{@} n''$ for some $n'' \notin |G|, |\Gamma, x : \bar{p}\, A \mathbin{@} n|$

(2) $G; \Gamma, x : \bar{p}\, A \mathbin{@} n \vdash t_2' : + B_1 \mathbin{@} n'''$

(3) $G; \Gamma, x : \bar{p}\, A \mathbin{@} n \vdash t_2'' : - B_2 \mathbin{@} n'''$

(4) $G \vdash n' \preccurlyeq_+^* n'''$

$$\frac{}{\Gamma, x : p\ A, \Gamma' \vdash_c x : p\ A}\ \text{ClassAx} \qquad \frac{}{\Gamma \vdash_c \mathbf{triv} : p\ \langle p \rangle}\ \text{ClassUnit}$$

$$\frac{\Gamma \vdash_c t_1 : p\ A \qquad \Gamma \vdash_c t_2 : p\ B}{\Gamma \vdash_c (t_1, t_2) : p\ (A \wedge_p B)}\ \text{ClassAnd} \qquad \frac{\Gamma \vdash_c t : p\ A_d}{\Gamma \vdash_c \mathbf{in}_d\ t : p\ (A_1 \wedge_{\bar{p}} A_2)}\ \text{ClassAndBar}$$

$$\frac{\Gamma, x : p\ A \vdash_c t : p\ B}{\Gamma \vdash_c \lambda x.t : p\ (A \to_p B)}\ \text{ClassImp} \qquad \frac{\Gamma \vdash_c t_1 : \bar{p}\ A \qquad \Gamma \vdash_c t_2 : p\ B}{\Gamma \vdash_c \langle t_1, t_2 \rangle : p\ (A \to_{\bar{p}} B)}\ \text{ClassImpBar}$$

$$\frac{\Gamma, x : \bar{p}\ A \vdash_c t_1 : +\ B \quad \Gamma, x : \bar{p}\ A \vdash_c t_2 : -\ B}{\Gamma \vdash_c \nu\, x.t_1 \bullet t_2 : p\ A}\ \text{ClassCut}$$

Figure 49. Classical typing of DTT terms

Using (1) and (4) we may apply node substitution for typing (Lemma 11.3.0.4) to obtain (5) $[n'''/n'']G; [n'''/n'']\Gamma, x : \bar{p}\ A\ @\ n, y : +\ B_1\ @\ n''' \vdash t_1' : +\ B_2\ @\ n'''$.

Finally, by applying substitution for typing using (2) and (5) we obtain

$$(6)\ [n'''/n'']G; [n'''/n'']\Gamma, x : \bar{p}\ A\ @\ n \vdash [t_2'/y]t_1' : +\ B_2\ @\ n''',$$

and since n'' is a fresh in G and Γ we know (6) is equivalent to

$$(7)\ G; \Gamma, x : \bar{p}\ A\ @\ n \vdash [t_2'/y]t_1' : +\ B_2\ @\ n'''.$$

Finally, by applying the Cut rule using (7) and (3) we obtain

$$G; \Gamma \vdash \nu\, x.[t_2'/y]t_1' \bullet t_2'' : p\ A\ @\ n.$$

□

A more substantial property is strong normalization of reduction for typed terms. To prove this result, we will prove a stronger property, namely strong normalization for reduction of terms which are typable using the system of classical typing rules in Figure 49 [40]. This is justified by the following easy result (proof omitted), where $\ulcorner\Gamma\urcorner$ just drops the world annotations from assumptions in Γ:

Theorem 11.3.0.7. *If* $G; \Gamma \vdash t : p\ A\ @\ n$, *then* $\ulcorner\Gamma\urcorner \vdash_c t : p\ A$

Let **SN** be the set of terms which are strongly normalizing with respect to the reduction relation. Let *Var* be the set of term variables, and let us use x and y as metavariables for variables. We will prove strong normalization for classically typed terms using a version of Krivine's classical realizability [79]. We define three interpretations of types in Figure 50. The definition is by mutual induction, and can easily be seen to well-founded, as the definition of $[\![A]\!]^+$ invokes the definition of $[\![A]\!]^-$ with the same type, which in turn invokes the definition of $[\![A]\!]^{+c}$ with the same type; and the definition of $[\![A]\!]^{+c}$ may invoke either of the other definitions at a strictly smaller type. The reader familiar with such proofs will also recognize the debt owed to Girard [61]. That is we can see that this style of proof is very similar to proofs by the Tait-Girard reducibility methods – See Section 6.3.

Lemma 11.3.0.8. *[Step interpretations] If* $t \in [\![A]\!]^+$ *and* $t \rightsquigarrow t'$, *then* $t' \in [\![A]\!]^+$; *and similiarly if* $t \in [\![A]\!]^-$ *or* $t \in [\![A]\!]^{+c}$.

Proof. The proof is by a mutual well-founded induction. Assume $t \in [\![A]\!]^+$ and $t \rightsquigarrow t'$. We must show $t' \in [\![A]\!]^+$. For this, it suffices to assume $y \in \mathit{Var}$ and $t'' \in [\![A]\!]^-$, and

$$
\begin{array}{lcl}
t \in [\![A]\!]^{+} & \Leftrightarrow & \forall x \in \mathit{Var}.\ \forall t' \in [\![A]\!]^{-}.\ \nu\, x.t \bullet t' \in \mathbf{SN} \\
t \in [\![A]\!]^{-} & \Leftrightarrow & \forall x \in \mathit{Var}.\ \forall t' \in [\![A]\!]^{+c}.\ \nu\, x.t' \bullet t \in \mathbf{SN} \\
t \in [\![\langle + \rangle]\!]^{+c} & \Leftrightarrow & t \in \mathit{Var} \ \vee\ t \equiv \mathbf{triv} \\
t \in [\![\langle - \rangle]\!]^{+c} & \Leftrightarrow & t \in \mathit{Var} \\
t \in [\![A \to_{+} B]\!]^{+c} & \Leftrightarrow & t \in \mathit{Var} \ \vee\ \exists x, t'.t \equiv \lambda x.\, t' \ \wedge\ \forall t'' \in [\![A]\!]^{+}.\ [t''/x]t' \in [\![B]\!]^{+} \\
t \in [\![A \to_{-} B]\!]^{+c} & \Leftrightarrow & t \in \mathit{Var} \ \vee\ \exists t_1 \in [\![A]\!]^{-}, t_2 \in [\![B]\!]^{+}.\ t \equiv \langle t_1, t_2 \rangle \\
t \in [\![A \wedge_{+} B]\!]^{+c} & \Leftrightarrow & t \in \mathit{Var} \ \vee\ \exists t_1 \in [\![A]\!]^{+}, t_2 \in [\![B]\!]^{+}.\ t \equiv (t_1, t_2) \\
t \in [\![A_1 \wedge_{-} A_2]\!]^{+c} & \Leftrightarrow & t \in \mathit{Var} \ \vee\ \exists d. \exists t' \in [\![A_d]\!]^{+}.\ t \equiv \mathbf{in}_d\, t'
\end{array}
$$

Figure 50. Interpretations of types

show $\nu\, y.t' \bullet t'' \in \mathbf{SN}$. From the assumption that $t \in [\![A]\!]^{+}$, we have

$$\nu\, y.t \bullet t'' \in \mathbf{SN}$$

which indeed implies that

$$\nu\, y.t' \bullet t'' \in \mathbf{SN}$$

A similar argument applies if $t \in [\![A]\!]^{-}$.

For the last part of the lemma, assume $t \in [\![A]\!]^{+c}$ with $t \rightsquigarrow t'$, and show $t' \in [\![A]\!]^{+c}$. The only possible cases are the following, where $t \notin \mathit{Vars}$.

If $A \equiv A_1 \to_{+} A_2$, then t is of the form $\lambda x.t_a$ for some x and t_a, where for all $t_b \in [\![A_1]\!]^{+}$, we have $[t_b/x]t_a \in [\![A_2]\!]^{+}$. Since $t \rightsquigarrow t'$, t' must be $\lambda x.t'_a$ for some t'_a with $t_a \rightsquigarrow t'_a$. It suffices now to assume an arbitrary $t_b \in [\![A_1]\!]^{+}$, and show $[t_b/x]t'_a \in [\![A_2]\!]^{+}$. But $[t_b/x]t_a \rightsquigarrow [t_b/x]t'_a$ follows from $t_a \rightsquigarrow t'_a$, so by our IH, we have $[t_b/x]t'_a \in [\![A_2]\!]^{+}$, as required.

If $A \equiv A_1 \to_{-} A_2$, then t is of the form $\langle t_1, t_2 \rangle$ for some $t_1 \in [\![A_1]\!]^{-}$ and $t_2 \in [\![A_2]\!]^{+}$; and $t' \equiv \langle t'_1, t'_2 \rangle$ where either $t'_1 \equiv t_1$ and $t_2 \rightsquigarrow t'_2$ or else $t_1 \rightsquigarrow t'_1$ and

$t_2' \equiv t_2$. Either way, we have $t_1' \in [\![A_1]\!]^-$ and $t_2' \in [\![A_2]\!]^+$ by our IH, so we have $\langle t_1', t_2' \rangle \in [\![A_1 \to_- A_2]\!]^{+c}$ as required.

The other cases for $A \equiv A_1 \wedge_p A_2$ are similar to the previous one. □

Lemma 11.3.0.9. *[SN interpretations]*

1. $[\![A]\!]^+ \subseteq \mathbf{SN}$
2. $\mathit{Vars} \subseteq [\![A]\!]^-$
3. $[\![A]\!]^- \subseteq \mathbf{SN}$
4. $[\![A]\!]^{+c} \subseteq \mathbf{SN}$

Proof. For purposes of this proof and subsequent ones, define $\delta(t)$ to be the length of the longest reduction sequence from t to a normal form, for $t \in \mathbf{SN}$.

The proof of the lemma is by mutual well-founded induction on the pair (A, n), where n is the number of the proposition in the statement of the lemma; the well-founded ordering in question is the lexicographic combination of the structural ordering on types (for A) and the ordering $1 > 2 > 4 > 3$ (for n).

For proposition (1): assume $t \in [\![A]\!]^+$, and show $t \in \mathbf{SN}$. Let x be a variable. By IH(2), $x \in [\![A]\!]^-$, so by the definition of $[\![A]\!]^+$, we have

$$\nu\, x.t \bullet x \in \mathbf{SN}$$

This implies $t \in \mathbf{SN}$.

For proposition (2): assume $x \in \mathit{Vars}$, and show $x \in [\![A]\!]^-$. For the latter, it suffices to assume arbitrary $y \in \mathit{Vars}$ and $t' \in [\![A]\!]^{+c}$, and show $\nu\, y.t' \bullet x \in \mathbf{SN}$. We will prove this by inner induction on $\delta(t')$, which is defined by IH(4). By the definition of $[\![A]\!]^{+c}$ for the various cases of A, we see that $\nu\, y.t' \bullet x$ cannot be a redex itself, as t' cannot be a cut. If t' is a normal form we are done. If $t \rightsquigarrow t''$, then we have $t'' \in [\![A]\!]^{+c}$ by Lemma 11.3.0.8, and we may apply the inner induction hypothesis.

For proposition (3): assume $t \in [\![A]\!]^-$, and show $t \in \mathbf{SN}$. By the definition of $[\![A]\!]^-$ and the fact that $\mathit{Vars} \subseteq [\![A]\!]^{+c}$ by definition of $[\![A]\!]^{+c}$, we have

$$\nu\, y.y \bullet t \in \mathbf{SN}$$

This implies $t \in \mathbf{SN}$ as required.

For proposition (4): assume $t \in [\![A]\!]^{+c}$, and consider the following cases. If $t \in \mathit{Vars}$ or $A \equiv \langle + \rangle$, then t is normal and the result is immediate. So suppose $A \equiv A_1 \to_+ A_2$. Then $t \equiv \lambda x.t'$ for some x and t' where for all $t'' \in [\![A_1]\!]^+$, $[t''/x]t' \in [\![A_2]\!]^+$. By IH(2), the variable x itself is in $[\![A_1]\!]^+$, so we know that $t' \equiv [x/x]t' \in [\![A_2]\!]^+$. Then by IH(1) we have $t' \in \mathbf{SN}$, which implies $\lambda x.t' \in \mathbf{SN}$. If $A \equiv A_1 \to_- A_2$, then $t \equiv \langle t_1, t_2 \rangle$ for some $t_1 \in [\![A_1]\!]^-$ and $t_2 \in [\![A_2]\!]^+$. By IH(3) and IH(1), $t_1 \in \mathbf{SN}$ and $t_2 \in \mathbf{SN}$, which implies $\langle t_1, t_2 \rangle \in \mathbf{SN}$. The cases for $A \equiv A_1 \wedge_p A_2$ are similar to this one. □

Definition 11.3.0.10.

[Interpretation of contexts] $[\![\Gamma]\!]$ *is the set of substitutions* σ *such that for all* $x : p\, A \in \Gamma$, $\sigma(x) \in [\![A]\!]^p$.

Lemma 11.3.0.11. *[Canonical positive is positive]* $[\![A]\!]^{+c} \subseteq [\![A]\!]^+$

Proof. Assume $t \in [\![A]\!]^{+c}$ and show $t \in [\![A]\!]^+$. For the latter, assume arbitrary $x \in \mathit{Vars}$ and $t' \in [\![A]\!]^-$, and show $\nu\, x.t \bullet t' \in \mathbf{SN}$. This follows immediately from the assumption that $t' \in [\![A]\!]^-$. □

Finally, we have everything we need to conclude strong normalization of DTT.

Theorem 11.3.0.12. *[Soundness] If* $\Gamma \vdash_c t : p\ A$ *then for all* $\sigma \in [\![\Gamma]\!]$, $\sigma t \in [\![A]\!]^p$.

Proof. The proof is by induction on the derivation of $\Gamma \vdash_c t : p\ A$. We consider the two possible polarities for the conclusion of the typing judgment separately.

Case.

$$\frac{}{\Gamma, x : p\ A, \Gamma' \vdash_c x : p\ A}\ \text{ClassAx}$$

Since $\sigma \in [\![\Gamma, x : p\ A, \Gamma']\!]$, $\sigma(x) \in [\![A]\!]^p$ as required.

Case.

$$\frac{}{\Gamma \vdash_c \mathbf{triv} : +\ \langle + \rangle}\ \text{ClassUnit}$$

We have $\mathbf{triv} \in [\![\langle + \rangle]\!]^{+c}$ by definition.

Case.

$$\frac{}{\Gamma \vdash_c \mathbf{triv} : -\ \langle - \rangle}\ \text{ClassUnit}$$

To prove $\mathbf{triv} \in [\![\langle - \rangle]\!]^-$, it suffices to assume arbitrary $y \in \mathit{Vars}$ and $t \in [\![\langle - \rangle]\!]^{+c}$, and show $\nu\, y.t \bullet \mathbf{triv} \in \mathbf{SN}$. By definition of $[\![\langle - \rangle]\!]^{+c}$, $t \in \mathit{Vars}$, and then $\nu\, y.t \bullet \mathbf{triv}$ is in normal form.

Case.

$$\frac{\Gamma \vdash_c t_1 : +\ A \qquad \Gamma \vdash_c t_2 : +\ B}{\Gamma \vdash_c (t_1, t_2) : +\ A \wedge_+ B}\ \text{ClassAnd}$$

By Lemma 11.3.0.11, it suffices to show $(\sigma t_1, \sigma t_2) \in [\![A \wedge_+ B]\!]^{+c}$. This follows directly from the definition of $[\![A \wedge_+ B]\!]^{+c}$, since the IH gives us $\sigma t_1 \in [\![A]\!]^+$ and $\sigma t_2 \in [\![B]\!]^+$.

Case.

$$\frac{\Gamma \vdash_c t_1 : - A_1 \qquad \Gamma \vdash_c t_2 : - A_2}{\Gamma \vdash_c (t_1, t_2) : - A_1 \wedge_- A_2} \text{ ClassAnd}$$

It suffices to assume arbitrary $y \in \mathit{Vars}$ and $t' \in [\![A_1 \wedge_- A_2]\!]^{+c}$, and show $\nu\, y.t' \bullet (\sigma\, t_1, \sigma\, t_2) \in \mathbf{SN}$. If $t' \in \mathit{Vars}$, then this follows by Lemma 11.3.0.9 from the facts that $\sigma t_1 \in [\![A_1]\!]^+$ and $\sigma t_2 \in [\![A2]\!]^+$, which we have by the IH. So suppose t' is of the form $\mathbf{in}_d\, t''$ for some d and some $t'' \in [\![A_d]\!]^+$. By the definition of **SN**, it suffices to show that all one-step successors t_a of the term in question are **SN**. The proof of this is by inner induction on $\delta(t'') + \delta(\sigma t_1) + \delta(\sigma t_2)$, which exists by Lemma 11.3.0.9, using also Lemma 11.3.0.8. Suppose that we step to t_a by stepping t'', σt_1, or σt_2. Then the result holds by the inner IH. So consider the step

$$\nu\, y.\mathbf{in}_d\, t'' \bullet (\sigma\, t_1, \sigma\, t_2) \rightsquigarrow \nu\, y.t'' \bullet \sigma\, t_d$$

We then have $\nu\, y.t'' \bullet \sigma\, t_d \in \mathbf{SN}$ from the facts that $t'' \in [\![A_d]\!]^+$ and $\sigma t_d \in [\![A_d]\!]^-$, by the definition of $[\![A_d]\!]^+$.

Case.

$$\frac{\Gamma \vdash_c t : + A_d}{\Gamma \vdash_c \mathbf{in}_d\, t : + A_1 \wedge_- A_2} \text{ ClassAndBar}$$

By Lemma 11.3.0.11, it suffices to prove $\mathbf{in}_d\, \sigma\, t \in [\![A_1 \wedge_- A_2]\!]^+$, but by the definition of $[\![A_1 \wedge_- A_2]\!]^+$, this follows directly from $\sigma t \in [\![A_d]\!]^+$, which we have by the IH.

Case.

$$\frac{\Gamma \vdash_c t : - A_d}{\Gamma \vdash_c \mathbf{in}_d\, t : - A_1 \wedge_+ A_2}\ \text{ClassAndBar}$$

To prove $\mathbf{in}_d\, \sigma\, t \in [\![A_1 \wedge_+ A_2]\!]^-$, it suffices to assume arbitrary $y \in \mathit{Vars}$ and $t' \in [\![A_1 \wedge_+ A_2]\!]^{+c}$, and show $\nu\, y.t' \bullet \mathbf{in}_d\, \sigma\, t \in \mathbf{SN}$. If $t' \in \mathit{Vars}$, then this follows from the fact that $\sigma t \in \mathbf{SN}$, which we have by Lemma 11.3.0.9 from $\sigma t \in [\![A_d]\!]^-$ (which the IH gives us). So suppose t' is of the form (s_1, s_2) for some $s_1 \in [\![A_1]\!]^+$ and $s_2 \in [\![A_2]\!]^+$. It suffices to prove that all one-step successors of the term in question are in $\mathbf{SN}$, as we did in a previous case above. Lemma 11.3.0.9 lets us proceed by inner induction on $\delta(\sigma t) + \delta(s_1) + \delta(s_2)$, using also Lemma 11.3.0.8. If we step σt, s_1 or s_2, then the result holds by inner IH. Otherwise, we have the step

$$\nu\, y.(s_1, s_2) \bullet \mathbf{in}_d\, \sigma\, t \rightsquigarrow \nu\, y.s_d \bullet \sigma\, t$$

And this successor is in $\mathbf{SN}$ by the facts that $s_d \in [\![A_d]\!]^+$ and $\sigma t \in [\![A_d]\!]^-$, from the definition of $[\![A_d]\!]^+$.

Case.

$$\frac{\Gamma, x : + A \vdash_c t : + B}{\Gamma \vdash_c \lambda x.t : + A \to_+ B}\ \text{ClassImp}$$

By Lemma 11.3.0.11, it suffices to assume arbitrary $y \in \mathit{Vars}$ and $t' \in [\![A]\!]^+$, and prove $[t'/x](\sigma t) \in [\![B]\!]^+$. But this follows immediately from the IH, since $[t'/x](\sigma t) \equiv (\sigma[x \mapsto t'])t$ and $\sigma[x \mapsto t] \in [\![\Gamma, x : + A]\!]$.

Case.

$$\frac{\Gamma, x : - A \vdash_c t : - B}{\Gamma \vdash_c \lambda x.t : - A \to_- B}\ \text{ClassImp}$$

It suffices to assume arbitrary $y \in \mathit{Vars}$ and $t' \in [\![A \to_- B]\!]^{+c}$, and show $\nu\, y.t' \bullet \lambda x.\sigma\ t \in \mathbf{SN}$. Let us first observe that $\sigma\ t \in \mathbf{SN}$, because by the IH, for all $\sigma' \in [\![\Gamma, x :- A]\!]$, we have $\sigma' t \in [\![B]\!]^-$, and $[\![B]\!]^- \subseteq \mathbf{SN}$ by Lemma 11.3.0.9. We may instantiate this with $\sigma[x \mapsto x]$, since by Lemma 11.3.0.9, $x \in [\![A]\!]^-$. Since $\sigma\ t \in \mathbf{SN}$, we also have $\lambda x.\sigma\ t \in \mathbf{SN}$. Now let us consider cases for the assumption $t' \in [\![A \to_- B]\!]^{+c}$. If $t' \in \mathit{Vars}$ then we directly have $\nu\, y.t' \bullet \lambda x.\sigma\ t \in \mathbf{SN}$ from $\lambda x.\sigma\ t \in \mathbf{SN}$. So assume $t' \equiv \langle t_1, t_2 \rangle$ for some $t_1 \in [\![A]\!]^-$ and $t_2 \in [\![B]\!]^+$. By Lemma 11.3.0.9 again, we may reason by inner induction on $\delta(t_1)+\delta(t_2)+\delta(\sigma t)$ to show that all one-step successors of $\nu\, y.\langle t_1, t_2 \rangle \bullet \lambda x.\sigma\ t$ are in $\mathbf{SN}$, using also Lemma 11.3.0.8. If t_1, t_2, or σt steps, then the result follows by the inner IH. So suppose we have the step

$$\nu\, y.\langle t_1, t_2 \rangle \bullet \lambda x.\sigma\ t \rightsquigarrow \nu\, y.t_2 \bullet [t_1/x](\sigma\ t)$$

Since $t_1 \in [\![A]\!]^-$, the substitution $\sigma[x \mapsto t_1]$ is in $[\![\Gamma, x :- A]\!]$. So we may apply the IH to obtain $[t_1/x](\sigma t) \equiv \sigma[x \mapsto t_1] \in [\![B]\!]^-$. Then since $t_2 \in [\![B]\!]^+$, we have $\nu\, y.t_2 \bullet [t_1/x](\sigma\ t)$ by definition of $[\![B]\!]^+$.

Case.

$$\frac{\Gamma \vdash_c t_1 :- A \qquad \Gamma \vdash_c t_2 :+ B}{\Gamma \vdash_c \langle t_1, t_2 \rangle :+ (A \to_- B)}\ \text{ClassImpBar}$$

By Lemma 11.3.0.11, as in previous cases of positive typing, it suffices to prove $\langle \sigma\, t_1, \sigma\, t_2 \rangle \in [\![A \to_- B]\!]^{+c}$. By the definition of $[\![A \to_- B]\!]^{+c}$, this follows directly from $\sigma t_1 \in [\![A]\!]^-$ and $\sigma t_2 \in [\![B]\!]^+$, which we have by the IH.

Case.

$$\frac{\Gamma \vdash_c t_1 : +A \qquad \Gamma \vdash_c t_2 : -B}{\Gamma \vdash_c \langle t_1, t_2 \rangle : -(A \to_+ B)}\ \text{ClassImpBar}$$

It suffices to assume arbitrary $y \in \mathit{Vars}$ and $t' \in [\![A \to_+ B]\!]^{+c}$, and show $\nu\, y.t' \bullet \langle \sigma\, t_1, \sigma\, t_2 \rangle \in \mathbf{SN}$. By the IH, we have $\sigma t_1 \in [\![A]\!]^+$ and $\sigma t_2 \in [\![B]\!]^-$, and hence $\sigma t_1 \in \mathbf{SN}$ and $\sigma t_2 \in \mathbf{SN}$ by Lemma 11.3.0.9. If $t' \in \mathit{Vars}$, then these facts are sufficient to show the term in question is in $\mathbf{SN}$. So suppose $t' \equiv \lambda x.t_3$, for some $x \in \mathit{Vars}$ and t'' such that for all $t_4 \in [\![A]\!]^+$, $[t_4/x]t_3 \in [\![B]\!]^+$. By similar reasoning as in a previous case, we have $t_3 \in \mathbf{SN}$. So we may proceed by inner induction on $\delta(t_1) + \delta(t_2) + \delta(t_3)$ to show that all one-step successors of $\nu\, y.\lambda x.t_3 \bullet \langle \sigma\, t_1, \sigma\, t_2 \rangle$ are in $\mathbf{SN}$, using also Lemma 11.3.0.8. If it is t_3, σt_1, or σt_2 which steps, then the result follows by the inner IH. So consider this step:

$$\nu\, y.\lambda x.t_3 \bullet \langle \sigma\, t_1, \sigma\, t_2 \rangle \rightsquigarrow \nu\, y.[\sigma\, t_1/x]t_3 \bullet \sigma\, t_2$$

Since we have that $\sigma t_1 \in [\![A]\!]^+$, the assumption about substitution instances of t_3 gives us that $[\sigma t_1/x]t_3 \in [\![B]\!]^+$, which is then sufficient to conclude $\nu\, y.[\sigma\, t_1/x]t_3 \bullet \sigma\, t_2 \in \mathbf{SN}$ by the definition of $[\![B]\!]^+$.

Case.

$$\frac{\Gamma, x : -A \vdash_c t_1 : +B \qquad \Gamma, x : -A \vdash_c t_2 : -B}{\Gamma \vdash_c \nu\, x.t_1 \bullet t_2 : +A}\ \text{ClassCut}$$

It suffices to assume arbitrary $y \in \mathit{Vars}$ and $t' \in [\![A]\!]^-$, and show $\nu\, y.(\nu\, x.\sigma\, t_1 \bullet \sigma\, t_2) \bullet t' \in \mathbf{SN}$. By the IH and part 2 of Lemma 11.3.0.9, we know that $\sigma t_1 \in [\![B]\!]^+$ and $\sigma t_2 \in [\![B]\!]^-$. By Lemma 11.3.0.9 again, we have $t' \in \mathbf{SN}$, $\sigma t_1 \in \mathbf{SN}$,

and $\sigma t_2 \in \mathbf{SN}$. So we may reason by induction on $\delta(t') + \delta(\sigma t_1) + \delta(\sigma t_2)$ to show that all one-step successors of $\nu\, y.(\nu\, x.\sigma\, t_1 \bullet \sigma\, t_2) \bullet t'$ are in $\mathbf{SN}$, using also Lemma 11.3.0.8. If it is t', σt_1, or σt_2 which steps, then the result follows by the inner IH. The only possible other reduction is by the RBetaL reduction rule (Figure 47). And then, since $t' \in [\![A]\!]^-$, we may apply the IH to conclude that $[t'/x](\sigma t_1) \in [\![B]\!]^+$ and $[t'/x](\sigma t_2) \in [\![B]\!]^-$. By the definition of $\in [\![B]\!]^+$, this suffices to prove $\nu\, y.[t'/x]\sigma\, t_1 \bullet [t'/x]\sigma\, t_2 \in \mathbf{SN}$, as required.

Case.

$$\frac{\Gamma, x :- A \vdash_c t_1 :+ B \qquad \Gamma, x :- A \vdash_c t_2 :- B}{\Gamma \vdash_c \nu\, x.t_1 \bullet t_2 :- A}\ \text{ClassCut}$$

It suffices to consider arbitrary $y \in \mathit{Vars}$ and $t' \in [\![A]\!]^{+c}$, and show $\nu\, y.t' \bullet (\nu\, x.\sigma\, t_1 \bullet \sigma\, t_2) \in \mathbf{SN}$. By the IH and part 2 of Lemma 11.3.0.9, we have $\sigma t_1 \in [\![B]\!]^+$ and $\sigma t_2 \in [\![B]\!]^-$, which implies $\sigma t_1 \in \mathbf{SN}$ and $\sigma t_2 \in \mathbf{SN}$ by Lemma 11.3.0.9 again. We proceed by inner induction on $\delta(t') + \delta(\sigma t_1) + \delta(\sigma t_2)$, using Lemma 11.3.0.8, to show that all one-step successors of $\nu\, y.t' \bullet (\nu\, x.\sigma\, t_1 \bullet \sigma\, t_2)$ are in $\mathbf{SN}$. If it is t', σt_1, or σt_2 which steps, then the result holds by inner IH. The only other reduction possible is by RBetaR, since t' cannot be a cut term by the definition of $[\![A]\!]^{+c}$. In this case, the IH gives us $[t'/x]\sigma t_1 \in [\![B]\!]^+$ and $[t'/x]\sigma t_2 \in [\![B]\!]^-$, and we then have $\nu\, y.[t'/x]\sigma\, t_1 \bullet [t'/x]\sigma\, t_2 \in \mathbf{SN}$ by the definition of $[\![B]\!]^+$.

□

Corollary 11.3.0.13. *[Strong Normalization] If $G; \Gamma \vdash t : p\ A\ @\ n$, then $t \in \mathbf{SN}$.*

Proof. This follows easily by putting together Theorems 11.3.0.7 and 11.3.0.12, with Lemma 11.3.0.9. □

Corollary 11.3.0.14. *[Cut Elimination] If* $G;\Gamma \vdash t : p\ A\ @\ n$*, then there is normal* t' *with* $t \rightsquigarrow^* t'$ *and* t' *containing only cut terms of the form* $\nu\, x.y \bullet t$ *or* $\nu\, x.t \bullet y$*, for* y *a variable.*

Using the previous results we can see that the canonicity restrictions placed on the reduction relation enforces confluence of the reduction relation.

Lemma 11.3.0.15. *[Local Confluence] The reduction relation of Figure 47 is locally confluent.*

Proof. We may view the reduction rules as higher-order pattern rewrite rules. It is easy to confirm that all critical pairs (e.g., between RBetaR and the rules RImp, RImpBar, RAnd1, RAndBar1, RAnd2, and RAndBar2) are joinable. Local confluence then follows by the higher-order critical pair lemma [95]. □

Theorem 11.3.0.16. *[Confluence for Typable Terms] The reduction relation restricted to terms typable in DTT is confluent.*

Proof. Suppose $G;\Gamma \vdash t : p\ A\ @\ n$ for some G, Γ, p, and A. By Lemma 11.3.0.6, any reductions in the unrestricted reduction relation from t are also in the reduction relation restricted to typable terms. The result now follows from Newman's Lemma, using Lemma 11.3.0.15 and Theorem 11.3.0.13. □

PART D

NORMALIZATION BY HEREDITARY SUBSTITUTION

CHAPTER 12

STRATIFIED SYSTEM F AND BEYOND

One motivation for the work in this thesis is that programming languages must contain the means of verifying the software written in them. This verification would catch major bugs during development as opposed to after the software is released to the public. For example, the bug found in the breaking system of the 2010 Prius would have never been released, and thus no accidents would have occurred. A programming language that contains the ability to verify programs would have to contain some notion of a logic, and this logic must be trusted. That is, the proofs written in this logic must be true; this corresponds to logical consistency.

In Section 6.1 we introduce the hereditay substitution proof technique for showing normalization and metioned that has one caveat, it is not known which type theories it can be applied to – see Section 6.1 for a list of type theories that are known to be proven normalizing using hereditary substitution, and a list of some that are not. The contribution of each of the following chapters is to widen the set of type theories hereditay substitution can be applied to. We first give several proofs of normalization using hereditary substitution for several extensions of stratified system F (SSF), and then give the first proof of normalization by hereditary substitution for a classical type theory called the $\lambda\Delta$-calculus. Throughout all of the following chapters we assume the reader is familiar with hereditary substitution. If the reader is not, then they should first read Section 6.1.

12.1 Stratified System F

In [50] Harley Eades and Aaron Stump show that Stratified System F (SSF) is normalizing using a proof method which uses the hereditary substitution function implicitly. We find it apparent that the hereditary substitution technique we have introduced in Section 6.1 is easier to understand, use, and is more informative – with respect to the hereditary substitution function and its interaction with the type theory – than the implicit version of the proof method. So in this section we re-prove normalization of SSF using the hereditary function explicitly. This will set the stage for the later chapters which extend SSF with various features and then reprove normalization using hereditary substitution. For a brief introduction to SSF and its history see Section 1.2.

Stratified System F, consists of types which are stratified into levels (or ranks) based on type-quantification. The types that belong to level zero have no type-quantification, the types at level one only quantify over types of level zero, and the types at level n quantify over the types of level $n-1$. Stratifying System F into levels prevents impredicativity. That is a type $\forall X.T'$ is no longer allowed quantify over itself. This restriction is similar to Russell's simple theory of types.

First, we briefly reintroduce SSF for the readers convenience. The syntax for Stratified System F can be found in the next definition followed by the definition of the reduction rules stated as rewrite rules where we omit the congruence rules.

Definition 12.1.0.1.

The syntax for terms, types, and kinds:

$$\begin{array}{lcl} K & := & *_0 \mid *_1 \mid \ldots \\ T & := & X \mid T \to T \mid \forall X : K.T \\ t & := & x \mid \lambda x : T.t \mid t\ t \mid \Lambda X : K.t \mid t[T] \end{array}$$

Definition 12.1.0.2.

Full β-reduction for SSF:

$$\begin{array}{rcl} (\Lambda X : *_p.t)[T] & \rightsquigarrow & [T/X]t \\ (\lambda x : T.t)t' & \rightsquigarrow & [t'/x]t \end{array}$$

Both the kinding and typing relations depend on well-formed contexts which is defined next.

Definition 12.1.0.3.

Context well-formedness rules:

$$\frac{}{\cdot\ Ok} \qquad \frac{\Gamma\ Ok}{\Gamma, X : *_p\ Ok} \qquad \frac{\Gamma \vdash T : *_p \qquad \Gamma\ Ok}{\Gamma, x : T\ Ok}$$

As stated before we use kinds to denote the level of a type. The following defines kinding relation:

Definition 12.1.0.4.

The kind assignment rules for SSF are defined as follows:

$$\frac{\Gamma \vdash T_1 : *_p \qquad \Gamma \vdash T_2 : *_q}{\Gamma \vdash T_1 \to T_2 : *_{max(p,q)}} \qquad \frac{\Gamma, X : *_q \vdash T : *_p}{\Gamma \vdash \forall X : *_q.T : *_{max(p,q)+1}}$$

$$\frac{\Gamma(X) = *_p \qquad p \leq q \qquad \Gamma\ Ok}{\Gamma \vdash X : *_q}$$

These kind assignement rules are slightly different then the level assignment rules defined by Leivant. The following rule:

$$\frac{\Gamma, X : *_q \vdash T : *_p}{\Gamma \vdash \forall X : *_q.T : *_{max(p,q)+1}}$$

was orignially defined to be the following by Leivant:

$$\frac{\Gamma, X : *_q \vdash T : *_p}{\Gamma \vdash \forall X : *_q . T : *_{max(p+1,q)}}$$

We conjecture that this modification does not hinder the expressive power of the type theory, meaning, any typeable term of Leivant's system is typeable in ours, but at potentially higher level. However, we do not prove this here. The following lemma shows that all kindable types are kindable with respect to a well-formed context.

Lemma 12.1.0.5. *If* $\Gamma \vdash T : *_p$ *then* $\Gamma\ Ok$.

Proof. This is a proof by structural induction on the kinding derivation of $\Gamma \vdash T : *_p$.

Case.

$$\frac{\Gamma(X) = *_p \qquad p \leq q \qquad \Gamma\ Ok}{\Gamma \vdash X : *_q}$$

By inversion of the kind-checking rule $\Gamma\ Ok$.

Case.

$$\frac{\Gamma \vdash T_1 : *_p \qquad \Gamma \vdash T_2 : *_q}{\Gamma \vdash T_1 \to T_2 : *_{max(p,q)}}$$

By the induction hypothesis, $\Gamma \vdash T_1 : *_p$ and $\Gamma \vdash T_2 : *_q$ both imply $\Gamma\ Ok$. Since the arrow-type kind-checking rule does not modify Γ in anyway Γ will remain Ok.

Case.

$$\frac{\Gamma, X : *_q \vdash T : *_p}{\Gamma \vdash \forall X : *_q.T : *_{max(p,q)+1}}$$

By the induction hypothesis $\Gamma, X : *_p \; Ok$, and by inversion of the type-variable well-formed contexts rule $\Gamma \; Ok$.

□

The previous lemma ensures that if a type is kindable then we could not have used a "bogus" context where we assume something we are not allowed to assume. Now we define the typing relation:

Definition 12.1.0.6.

Type assignment rules for SSF:

$$\frac{\Gamma(x) = T \qquad \Gamma \; Ok}{\Gamma \vdash x : T} \qquad \frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x : T_1.t : T_1 \to T_2} \qquad \frac{\Gamma \vdash t_1 : T_1 \to T_2 \qquad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1 \; t_2 : T_2}$$

$$\frac{\Gamma, X : *_p \vdash t : T}{\Gamma \vdash \Lambda X : *_p.t : \forall X : *_p.T} \qquad \frac{\Gamma \vdash t : \forall X : *_l.T_1 \qquad \Gamma \vdash T_2 : *_l}{\Gamma \vdash t[T_2] : [T_2/X]T_1}$$

The type assignment rules depend on the kinding relation defined above. Note that the level of the type T and the level of the type variable X in the type application rule must be the same.

12.1.1 Basic Syntactic Lemmas

We now state several results about the kinding relation. All of these are just basic results needed by the proofs of the key results later. The reader may wish to just quickly read through them. We simply list them with their proofs. Briefly,

Lemma 12.1.1.8 is used in the proof of Substitution for Typing (Lemma 12.1.5.27), Lemma 12.1.1.9 is used in the main substitution lemma (Lemma 12.1.5.25), and Lemma 12.1.1.10 is used in the proof of Context Weakening for the Interpretation of Types (Lemma 12.1.5.26).

Lemma 12.1.1.7. *[Level Weakening for Kinding] If* $\Gamma \vdash T : *_r$ *and* $r < s$ *then* $\Gamma \vdash T : *_s$.

Proof. We show level weakening for kinding by structural induction on the kinding derivation of $T : *_r$.

Case.

$$\frac{\Gamma(X) = *_p \qquad p \leq q \qquad \Gamma \; Ok}{\Gamma \vdash X : *_q}$$

By assumption we know $q < s$, hence by reapplying the rule and transitivity we obtain $\Gamma \vdash X : *_s$.

Case.

$$\frac{\Gamma \vdash T_1 : *_p \qquad \Gamma \vdash T_2 : *_q}{\Gamma \vdash T_1 \to T_2 : *_{max(p,q)}}$$

By the induction hypothesis $\Gamma \vdash T_1 : *_s$ and $\Gamma \vdash T_2 : *_s$ for some arbitrary $s > max(p, q)$. Therefore, by reapplying the rule we obtain $\Gamma \vdash T_1 \to T_2 : *_s$.

Case.

$$\frac{\Gamma, X : *_q \vdash T' : *_p}{\Gamma \vdash \forall X : *_q . T' : *_{max(p,q)+1}}$$

We know by assumption that $max(p, q)+1 < s$ which implies that $max(p, q) < s-1$. Now by the induction hypothesis $\Gamma, X : *_q \vdash T' : *_{s-1}$. Lastly, we reapply the rule and obtain $\Gamma \vdash \forall X : *_q . T' : *_s$.

□

Lemma 12.1.1.8. *[Substitution for Kinding, Context-Ok]*

Suppose $\Gamma \vdash T' : *_p$. *If* $\Gamma, X : *_p, \Gamma' \vdash T : *_q$ *with a derivation of depth* d, *then* $\Gamma, [T'/X]\Gamma' \vdash [T'/X]T : *_q$ *with a derivation of depth* d. *Also, if* $\Gamma, X : *_p, \Gamma'\ Ok$ *with a derivation of depth* d, *then* $\Gamma, [T'/X]\Gamma'\ Ok$ *with a derivation of depth* d.

Proof. This is a prove by induction on d. We prove the first implication first, and then the second, doing a case analysis for each implication on the form of the derivation whose depth is being considered.

Case.

$$\frac{(\Gamma, X : *_p, \Gamma')(Y) = *_r \qquad r \leq s \qquad \Gamma, X : *_p, \Gamma'\ Ok}{\Gamma, X : *_p, \Gamma' \vdash Y : *_s}$$

By assumption $\Gamma \vdash T' : *_p$. We must consider whether or not $X \equiv Y$. If $X \equiv Y$ then $[T'/X]Y \equiv T'$, $r = p$, and $q = s$; this conclusion is equivalent to $\Gamma, [T'/X]\Gamma' \vdash T' : *_q$ and by the induction hypothesis $\Gamma, [T'/X]\Gamma'\ Ok$. If $X \not\equiv Y$ then $[T'/X]Y \equiv Y$ and by the induction hypothesis $\Gamma, [T'/X]\Gamma'\ Ok$,

hence, $\Gamma, [T'/X]\Gamma' \vdash Y : *_q$.

Case.

$$\frac{\Gamma, X : *_p, \Gamma' \vdash T_1 : *_r \qquad \Gamma, X : *_p, \Gamma' \vdash T_2 : *_s}{\Gamma, X : *_p, \Gamma' \vdash T_1 \to T_2 : *_{max(r,s)}}$$

Here $q = max(r, s)$ and by the induction hypothesis $\Gamma, [T'/X]\Gamma' \vdash [T'/X]T_1 : *_r$ and $\Gamma, [T'/X]\Gamma' \vdash [T'/X]T_2 : *_s$. We can now reapply the rule to get $\Gamma, [T'/X]\Gamma' \vdash [T'/X](T_1 \to T_2) : *_q$.

Case.

$$\frac{\Gamma, X : *_q, \Gamma', Y : *_r \vdash T : *_s}{\Gamma, X : *_p, \Gamma' \vdash \forall Y : *_r.T : *_{max(r,s)+1}}$$

Here $q = max(r, s) + 1$ and by the induction hypothesis $\Gamma, [T'/X]\Gamma', Y : *_r \vdash [T'/X]T : *_s$. We can reapply this rule to get $\Gamma, [T'/X]\Gamma' \vdash [T'/X]\forall Y : *_r.T : *_q$.

We now show the second implication. The case were $d = 0$ cannot arise, since it requires the context to be empty. Suppose $d = n + 1$. We do a case analysis on the last rule applied in the derivation of $\Gamma, X : *_p, \Gamma'$.

Case. Suppose $\Gamma' = \Gamma'', Y : *_q$.

$$\frac{\Gamma, X : *_p, \Gamma'' \; Ok}{\Gamma, X : *_p, \Gamma'', Y : *_q \; Ok}$$

By the induction hypothesis, $\Gamma, [T'/X]\Gamma'' \; Ok$. Now, by reapplying the rule above $\Gamma, [T'/X]\Gamma'', Y : *_q \; Ok$, hence $\Gamma, [T'/X]\Gamma' \; Ok$, since $X \not\equiv Y$.

Case. Suppose $\Gamma' = \Gamma'', y : T$.

$$\frac{\Gamma, X : *_p, \Gamma'' \vdash T : *_q \qquad \Gamma, X : *_p, \Gamma'' \; Ok}{\Gamma, X : *_p, \Gamma'', y : T \; Ok}$$

By the induction hypothesis, $\Gamma', [T'/X]\Gamma'' \vdash [T'/X]T : *_q$ and $\Gamma', [T'/X]\Gamma'' \; Ok$. Thus, by reapplying the rule above $\Gamma, [T'/X]\Gamma'', x : [T'/X]T \; Ok$, therefore, $\Gamma, [T'/X]\Gamma' \; Ok$.

□

Lemma 12.1.1.9. *[Context Strengthening for Kinding, Context-Ok]*

If $\Gamma, x : T', \Gamma' \vdash T : *_p$ *with a derivation of depth* d*, then* $\Gamma, \Gamma' \vdash T : *_p$ *with a derivation of depth* d*. Also, if* $\Gamma, x : T, \Gamma' \; Ok$ *with a derivation of depth* d*, then* $\Gamma, \Gamma' \; Ok$ *with a derivation of depth* d*.*

Proof. This is a prove by induction on d. We prove the first implication first, and then the second, doing a case analysis for each implication on the form of the derivation whose depth is being considered.

Case.

$$\frac{(\Gamma, x : T', \Gamma')(X) = *_p \qquad p \leq q \qquad \Gamma, x : T', \Gamma' \; Ok}{\Gamma, x : T', \Gamma' \vdash X : *_q}$$

By the second implication of the induction hypothesis, $\Gamma, \Gamma'\ Ok$. Also, $(\Gamma, \Gamma')(X) = *_p$. Now by reapplying the rule above, $\Gamma, \Gamma' \vdash X : *_q$.

Case.

$$\frac{\Gamma, x : T', \Gamma' \vdash T_1 : *_p \qquad \Gamma, x : T', \Gamma' \vdash T_2 : *_q}{\Gamma, x : T', \Gamma' \vdash T_1 \to T_2 : *_{max(p,q)}}$$

By the first implication of the induction hypothesis, $\Gamma, \Gamma' \vdash T_1 : *_p$ and $\Gamma, \Gamma' \vdash T_2 : *_q$. By reapplying the rule above we get, $\Gamma, \Gamma' \vdash T_1 \to T_2 : *_{max(p,q)}$.

Case.

$$\frac{\Gamma, x : T, \Gamma', Y : *_q \vdash T : *_p}{\Gamma, x : T', \Gamma' \vdash \forall Y : *_q . T : *_{max(p,q)+1}}$$

By the first implication of the induction hypothesis, $\Gamma, \Gamma', Y : *_q \vdash T : *_p$. By reapplying the rule we get, $\Gamma, \Gamma' \vdash \forall Y : *_q . T : *_{max(p,q)+1}$.

We now prove the second implication. The case where $d = 0$ cannot arise, since it requires the context to be empty. Suppose $d = n + 1$. We do a case analysis on the last rule applied in the derivation of $\Gamma, x : T, \Gamma'\ Ok$.

Case. Suppose $\Gamma' = \Gamma'', Y : *_l$. Then the last rule of the derivation of $\Gamma, x : T, \Gamma'\ Ok$ is as follows.

$$\frac{\Gamma, x : T, \Gamma''\ Ok}{\Gamma, x : T, \Gamma'', Y : *_l\ Ok}$$

By the second implication of the induction hypothesis, $\Gamma, \Gamma''\ Ok$. Now reapplying the rule we get, $\Gamma, \Gamma'', Y : *_l\ Ok$, which is equivalent to $\Gamma, \Gamma'\ Ok$.

Case. Suppose $\Gamma' = \Gamma'', y : T'$. Then the last rule of the derivation of $\Gamma, x : T, \Gamma'\ Ok$ is as follows.

$$\frac{\Gamma, x : T, \Gamma'' \vdash T' : *_p \qquad \Gamma, x : T, \Gamma''\ Ok}{\Gamma, x : T, \Gamma'', y : T'\ Ok}$$

By the first implication of the induction hypothesis, $\Gamma, \Gamma'' \vdash T' : *_p$ and by the second, $\Gamma, \Gamma''\ Ok$. Therefore, by reapplying the rule above, $\Gamma, \Gamma'', y : T'\ Ok$, which is equivalent to $\Gamma, \Gamma'\ Ok$.

□

Lemma 12.1.1.10. *[Context Weakening for Kinding] If* $\Gamma, \Gamma'', \Gamma'\ Ok$, *and* $\Gamma, \Gamma' \vdash T : *_p$, *then* $\Gamma, \Gamma'', \Gamma' \vdash T : *_p$.

Proof. This is a proof by structural induction on the kinding derivation of $\Gamma, \Gamma' \vdash T : *_p$.

Case.

$$\frac{(\Gamma, \Gamma')(X) = *_p \qquad p \leq q \qquad \Gamma, \Gamma'\ Ok}{\Gamma, \Gamma' \vdash X : *_q}$$

If $(\Gamma, \Gamma')(X) = *_p$ then $(\Gamma, \Gamma'', \Gamma')(X) = *_p$, hence, by reapplying the type-variable kind-checking rule, $\Gamma, \Gamma'', \Gamma' \vdash T : *_p$.

Case.

$$\frac{\Gamma, \Gamma' \vdash T_1 : *_p \qquad \Gamma, \Gamma' \vdash T_2 : *_q}{\Gamma, \Gamma' \vdash T_1 \to T_2 : *_{max(p,q)}}$$

By the induction hypothesis $\Gamma, \Gamma'', \Gamma' \vdash T_1 : *_p$ and $\Gamma, \Gamma'', \Gamma' \vdash T_2 : *_q$, hence, by reapplying the arrow-type kind-checking rule $\Gamma, \Gamma'', \Gamma'' \vdash T_1 \rightarrow T_2 : *_{max(p,q)}$.

Case.

$$\frac{\Gamma, \Gamma', X : *_q \vdash T' : *_p}{\Gamma, \Gamma' \vdash \forall X : *_q . T' : *_{max(p,q)+1}}$$

By the induction hypothesis $\Gamma, \Gamma'', \Gamma', X : *_p \vdash T : *_q$, hence, by reapplying the forall-type kind-checking rule $\Gamma, \Gamma'', \Gamma' \vdash \forall X : *_p . T : *_{max(p,q)+1}$.

□

Lemma 12.1.1.11. *[Regularity] If* $\Gamma \vdash t : T$ *then* $\Gamma \vdash T : *_p$ *for some* p.

Proof. This proof is by structural induction on the derivation of $\Gamma \vdash t : T$.

Case.

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

By the definition of well-formedness contexts $\Gamma \vdash T : *_p$ for some p.

Case.

$$\frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x : T_1 . t : T_1 \rightarrow T_2}$$

By the induction hypothesis $\Gamma \vdash T_1 : *_p$, $\Gamma, x : T_1 \vdash T_2 : *_q$ and by Lemma 12.1.1.9, $\Gamma \vdash T_2 : *_q$. By applying the arrow-type kind-checking rule we get $\Gamma \vdash T_1 \to T_2 : *_{max(p,q)}$.

Case.

$$\frac{\Gamma \vdash t_1 : T_1 \to T_2 \qquad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\ t_2 : T_2}$$

By the induction hypothesis $\Gamma \vdash T_1 \to T_2 : *r$ and $\Gamma \vdash T_1 : *_p$. By inversion of the arrow-type kind-checking rule $r = max(p, q)$, for some q, which implies $\Gamma \vdash T_2 : *_q$.

Case.

$$\frac{\Gamma, X : *_p \vdash t : T}{\Gamma \vdash \Lambda X : *_p.t : \forall X : *_q.T}$$

By the induction hypothesis $\Gamma, X : *_q \vdash T : *_p$. By applying the forall-type kind-checking rule $\Gamma \vdash \forall X.T : *_{max(p,q)+1}$.

Case.

$$\frac{\Gamma \vdash t : \forall X : *_p.T_1 \qquad \Gamma \vdash T_2 : *_p}{\Gamma \vdash t[T_2] : [T_2/X]T_1}$$

By assumption $\Gamma \vdash T_2 : *_r$. By the induction hypothesis $\Gamma \vdash \forall X : *_p.T_1 : *_s$ and by inversion of the forall-type kind-checking rule $r = max(p, q) + 1$, for some q, which implies $\Gamma, X : *_p \vdash T_1 : *_q$. Now, by Lemma 12.1.1.8, $\Gamma \vdash [T_2/X]T_1 : *_q$.

□

We have stated all the basic lemmas we will need. We now proceed to the proof of normalization for SSF using hereditary substitution.

12.1.2 Ordering on Types

The following definition defines a well-founded ordering on the types of SSF. It consists of essentially the strict-subexpression ordering with an additional case for universal types. For the case of universal types the ordering states that a they are always larger than their instantiation. Now this seems odd, because syntactically the level of the instantiation could have increased, but it turns out that the level of the type actually decreases. That is, we know the level of the universal type is larger than the level of the instantiation. This is exactly the point of stratification!

Definition 12.1.2.12.

The ordering $>_\Gamma$ is defined as the least relation satisfying the universal closures of the following formulas:

$$\begin{array}{lll} T_1 \to T_2 & >_\Gamma & T_1 \\ T_1 \to T_2 & >_\Gamma & T_2 \\ \forall X : *_l . T & >_\Gamma & [T'/X]T \text{ where } \Gamma \vdash T' : *_l . \end{array}$$

We need transitivity in a number of places so we state that next.

Lemma 12.1.2.13. *[Transitivity of $>_\Gamma$] Let T, T', and T'' be kindable types. If $T >_\Gamma T'$ and $T' >_\Gamma T''$ then $T >_\Gamma T''$.*

Proof. Suppose $T >_\Gamma T'$ and $T' >_\Gamma T''$. If $T \equiv T_1 \to T_2$ then, T' must be a subexpression of T. Now if $T' \equiv T'_1 \to T'_2$ then, T'' must be a subexpression of T', which implies that T'' is a subexpression of T. Thus, $T >_\Gamma T''$. If $T' \equiv \forall X : *_l . T'_1$

then, there exists a type T_2' where, $\Gamma \vdash T_2' : *_l$, such that, $T'' \equiv [T_2'/X]T_1'$. The level of T' is $max(l, l')+1$, where l' is the level of T_1', the level of T'' is $max(l, l')$, and the level of T is $max(max(l, l') + 1, p)$, where p is the level of the type, which is, the second subexpression of T. Clearly, $max(max(l, l') + 1, p) \geq max(l, l')$, thus, $T >_\Gamma T''$.

If $T \equiv \forall X : *_l.T_1$, then $T' \equiv [T_2/X]T_1$ for some type T_2, where $\Gamma \vdash T_2 : *_l$. If $[T_2/X]T_1 \equiv T_1' \to T_2'$ then the level of T' is $max(p, q)$, where p is the level of T_1' and q is the level of T_2'. Now T'' must be a subexpression of T', hence the level of T'' is either p or q. Now, since the level of T is greater than the level of T' and we know, $max(p, q)$ is greater than both p and q then $T >_\Gamma T''$. If $[T_2/X]T_1 \equiv \forall Y : *_{l'}.T_1'$, then $T'' \equiv [T_2'/X]T_1'$. Now if p is the level of T_1, then the level of T is $max(l, p)+1$ and the level of T' must be $max(l, p)$ since we know the level of T' is greater than the level of T'' then clearly, the level of T is greater than the level of T''. Thus, $T >_\Gamma T''$. $\square$

To prove that the ordering on types ($>_\Gamma$) is well founded we need a function which computes the depth of a type. We will use this in a lexicographic ordering in the proof of Lemma 12.1.2.15 and is vital to showing that our ordering on types is well founded.

Definition 12.1.2.14.

The depth of a type T is defined as follows:

$$\begin{array}{lcl} depth(X) & = & 1 \\ depth(T \to T') & = & depth(T) + depth(T') \\ depth(\forall X : *_l.T) & = & depth(T) + 1 \end{array}$$

We define the following metric (l, d) in lexicographic combination, where l is the level of a type T and d is the depth of T. The following lemma shows that if

$T >_\Gamma T'$ then $(l, d) > (l', d')$. We will use this lemma to show well-foundedness of the ordering on types $>_\Gamma$.

Lemma 12.1.2.15. *[Well-Founded Measure] If $T >_\Gamma T'$ then $(l, d) > (l', d')$, where $\Gamma \vdash T : *_l$, $depth(T) = d$, $\Gamma \vdash T : *_{l'}$, and $depth(T') = d'$.*

Proof. Assume $T >_\Gamma T'$ for some types T and T'. We case split on the form of T. Clearly, T is not a type variable.

Case. Suppose $T \equiv T_1 \to T_2$. Then T' must be of the form T_1 or T_2. In both cases we have two cases to consider; either T and T' have the same level or they do not. Consider the first form and suppose they have the same level. Then it is clear that $depth(T) > depth(T')$. Now consider the latter form and suppose T and T' have the same level. Then, clearly, $depth(T) > depth(T')$. In either form if the level of T and T' are different, then the level of T is larger than the level of T'. In all cases $(l, d) > (l', d')$.

Case. Suppose $T \equiv \forall X : *_l.T_1$. Then T' must be of the form $[T_2/X]T_1$ for some type $\Gamma \vdash T_2 : *_l$. It is obvious that the level of T is always larger than the level of T'. Hence, $(l, d) > (l', d')$.

□

We now have the desired results to prove that the ordering $>_\Gamma$ is well-founded.

Theorem 12.1.2.16. *[Well-Founded Ordering] The ordering $>_\Gamma$ is well-founded on types T such that $\Gamma \vdash T : *_l$ for some l.*

Proof. If there exists a infinite decreasing sequence using our ordering on types, then there is an infinite decreasing sequence using our measure by Lemma 12.1.2.15, but that is impossible. □

12.1.3 Hereditary Substitution

The definition of the hereditary substitution function is a basic extension of hereditary substitution function for STLC. Before defining the hereditary substitution function we first define the construct type function for SSF. This function is now defined for three different types of input: term variables, term applications, and type applications.

Definition 12.1.3.17.

The construct type function for SSF is defined as follows:

$ctype_T(x, x) = T$

$ctype_T(x, t_1\ t_2) = T''$

Where $ctype_T(x, t_1) = T' \to T''$.

$ctype_T(x, t[T']) = [T'/X]T''$

Where $ctype_T(x, t) = \forall X : *_l.T''$.

Finally, we can define the hereditary substitution function for SSF.

Definition 12.1.3.18.

We define the hereditary substitution function for SSF as follows:

$[t/x]^T x = t$

$[t/x]^T y = y$

Where y is a variable distinct from x.

$[t/x]^T(\lambda y : T'.t') = \lambda y : T'.([t/x]^T t')$

$[t/x]^T(\Lambda X : *_l.t') = \Lambda X : *_l.([t/x]^T t')$

$[t/x]^T(t_1\ t_2) = ([t/x]^T t_1)\ ([t/x]^T t_2)$

Where $([t/x]^T t_1)$ is not a λ-abstraction, or both $([t/x]^T t_1)$ and t_1 are λ-abstractions.

$[t/x]^T(t_1\ t_2) = [([t/x]^T t_2)/y]^{T''} s_1'$

Where $([t/x]^T t_1) \equiv \lambda y : T''.s_1'$ for some y, s_1', and T'' and $ctype_T(x, t_1) = T'' \to T'$.

$[t/x]^T(t'[T']) = ([t/x]^T t')[T']$

Where $[t/x]^T t'$ is not a type abstraction or t' and $[t/x]^T t'$ are type abstractions.

$[t/x]^T(t'[T']) = [T'/X]s_1'$

Where $[t/x]^T t' \equiv \Lambda X : *_l.s_1'$, for some X, s_1' and $\Gamma \vdash T' : *_q$, such that, $q \leq l$ and $ctype_T(x, t') = \forall X : *_l.T''$.

The next lemma states the familiar properties of the construct type function. The first property is slightly different then the one defined for STLC. The difference arises from the fact that the ordering on types is not just the subexpression ordering, but relies on the level of the type in the ordering on types. So instead of T being a subexpression of the output of $ctype_T$ it will be greater than or equal to the output of $ctype_T$. The remainder of the properties are as usual.

Lemma 12.1.3.19. *[Properties of $ctype_T$]*

i. If $\Gamma, x : T, \Gamma' \vdash t : T'$ and $ctype_T(x, t) = T''$, then $head(t) = x$, $T' \equiv T''$, and $T' \leq_{\Gamma,\Gamma'} T$.

ii. If $\Gamma, x : T, \Gamma' \vdash t_1\ t_2 : T'$, $\Gamma \vdash t : T$, $[t/x]^T t_1 = \lambda y : T_1.q$, and t_1 is not, then

there exists a type ψ *such that* $ctype_T(x, t_1) = \psi$.

iii. If $\Gamma, x : T, \Gamma' \vdash t'[T''] : T'$, $\Gamma \vdash t : T$, $[t/x]^T t' = \Lambda X : *_l.t''$, *and* t' *is not, then there exists a type* ψ *such that* $ctype_T(x, t') = \psi$.

Proof. We prove part one first. This is a proof by induction on the structure of t.

Case. Suppose $t \equiv x$. Then $ctype_T(x, x) = T$. Clearly, $head(x) = x$ and $T \equiv T$.

Case. Suppose $t \equiv t_1\ t_2$. Then $ctype_T(x, t_1\ t_2) = T''$ when $ctype_T(x, t_1) = T' \to T''$.

Now $t > t_1$ so by the induciton hypothesis $head(t_1) = x$ and $T' \to T'' \leq_{\Gamma,\Gamma'} T$.

Therefore, $head(t_1\ t_2) = x$, and certainly, $T'' \leq_{\Gamma,\Gamma'} T$.

Next we prove part two. This is a proof by induction on the structure of $t_1\ t_2$.

The only possiblities for the form of t_1 is x, $\hat{t}_1\ \hat{t}_2$, or $\hat{t}[T'']$. All other forms would not result in $[t/x]^T t_1$ being a λ-abstraction and t_1 not. If $t_1 \equiv x$ then there exist a type T'' such that $T \equiv T'' \to T'$ and $ctype_T(x, x\ t_2) = T'$ when $ctype_T(x, x) = T \equiv T'' \to T'$ in this case. We know T'' to exist by inversion on $\Gamma, x : T, \Gamma' \vdash t_1\ t_2 : T'$.

Now suppose $t_1 \equiv (\hat{t}_1\ \hat{t}_2)$. Now knowing t_1' to not be a λ-abstraction implies that $\hat{t}_1$ is also not a λ-abstraction or $[t/x]^T t_1$ would be an application instead of a λ-abstraction. So it must be the case that $[t/x]^T \hat{t}_1$ is a λ-abstraction and $\hat{t}_1$ is not. Since $t_1\ t_2 > t_1$ we can apply the induction hypothesis to obtain there exists a type ψ such that $ctype_T(x, \hat{t}_1) = \psi$. Now by inversion on $\Gamma, x : T, \Gamma' \vdash t_1\ t_2 : T'$ we know there exists a type T'' such that $\Gamma, x : T, \Gamma' \vdash t_1 : T'' \to T'$. We know $t_1 \equiv (\hat{t}_1\ \hat{t}_2)$ so by inversion on $\Gamma, x : T, \Gamma' \vdash t_1 : T'' \to T'$ we know there exists a type ψ'' such that $\Gamma, x : T, \Gamma' \vdash \hat{t}_1 : \psi'' \to (T'' \to T')$. By part two of Lemma 12.1.3.19 we

know $\psi \equiv \psi'' \to (T'' \to T')$ and $ctype_T(x, t_1) = ctype_T(x, \hat{t}_1\ \hat{t}_2) = T'' \to T'$ when $ctype_T(x, \hat{t}_1) = \psi'' \to (T'' \to T')$, because we know $ctype_T(x, \hat{t}_1) = \psi$.

The case where t_1 is a type application is similar to the previous case.

The remaining parts of the lemma are similar to part two. □

12.1.4 Main Properties

We now define *rset* as an extension of the same function for STLC by adding type application redexes to the set of overall redexes of a term. It is defined in the following definition.

Definition 12.1.4.20.

The following function constructs the set of redexes within a term:

$$rset(x) = \emptyset$$

$$rset(\lambda x : T.t) = rset(t)$$

$$rset(\Lambda X : *_l.t) = rset(t)$$

$$rset(t_1\ t_2)$$
$$\begin{array}{lll} = & rset(t_1, t_2) & \text{if } t_1 \text{ is not a } \lambda\text{-abstraction.} \\ = & \{t_1\ t_2\} \cup rset(t_1', t_2) & \text{if } t_1 \equiv \lambda x : T.t_1'. \end{array}$$

$$rset(t''[T''])$$
$$\begin{array}{lll} = & rset(t'') & \text{if } t'' \text{ is not a type abstraction.} \\ = & \{t''[T'']\} \cup rset(t''') & \text{if } t'' \equiv \Lambda X : *_l.t'''. \end{array}$$

The extension of rset to multiple arguments is defined as follows:

$$rset(t_1, \ldots, t_n) =^{def} rset(t_1) \cup \cdots \cup rset(t_n).$$

Next we state all the properties of the hereditary substitution function. They are equivalent to the properties stated in Section 13.2.3 the only difference are their proofs.

Lemma 12.1.4.21. *[Total and Type Preserving] Suppose* $\Gamma \vdash t : T$ *and* $\Gamma, x : T, \Gamma' \vdash t' : T'$. *Then there exists a term* t'', *such that,* $[t/x]^T t' = t''$ *and* $\Gamma, \Gamma' \vdash t'' : T'$.

Proof. This is a proof by induction on the lexicorgraphic combination (T, t') of $>_{\Gamma,\Gamma'}$ and the strict subexpression ordering. We case split on t'.

Case. Suppose t' is either x or a variable y distinct from x. Trivial in both cases.

Case. Suppose $t' \equiv \lambda y : T_1.t'_1$. By inversion on the typing judgement we know $\Gamma, x : T, \Gamma', y : T_1 \vdash t'_1 : T_2$. We also know $t' > t'_1$, hence we can apply the induction hypothesis to obtain $[t/x]^T t'_1 = \hat{t}'_1$ and $\Gamma, \Gamma', y : T_1 \vdash \hat{t} : T_2$ for some term $\hat{t}'_1$. By the definition of the hereditary substitution function $[t/x]^T t' = \lambda y : T_1.[t/x]^T t'_1 = \lambda y : T_1.\hat{t}'_1$. It suffices to show that $\Gamma, \Gamma' \vdash \lambda y : T_1.\hat{t}'_1 : T_1 \to T_2$. By simply applying the λ-abstraction typing rule using $\Gamma, \Gamma', y : T_1 \vdash \hat{t} : T_2$ we obtain $\Gamma, \Gamma' \vdash \lambda y : T_1.\hat{t}'_1 : T_1 \to T_2$.

Case. Suppose $t' \equiv \Lambda X : *_l.t'_1$. Similar to the previous case.

Case. Suppose $t' \equiv t'_1\ t'_2$. By inversion we know $\Gamma, x : T, \Gamma' \vdash t'_1 : T'' \to T'$ and $\Gamma, x : T, \Gamma' \vdash t'_2 : T''$ for some types T' and T''. Clearly, $t' > t'_i$ for $i \in \{1, 2\}$. Thus, by the induction hypothesis there exists terms m_1 and m_2 such that $[t/x]^T t'_i = m_i$, $\Gamma, \Gamma' \vdash m_1 : T'' \to T'$ and $\Gamma, \Gamma' \vdash m_2 : T''$ for $i \in \{1, 2\}$. We case split on whether or not m_1 is a λ-abstraction and t'_1 is

not, or $ctype_T(x, t'_1)$ is undefined. We only consider the non-trivial cases when $m_1 \equiv \lambda y : T''.m'_1$, t'_1 is not a λ-abstraction, and $ctype_T(x, t'_1) = \psi'' \to \psi'$. Suppose the former. Now by Lemma 12.1.3.19 it is the case that there exists a ψ such that $ctype_T(x, t'_1) = \psi$, $\psi \equiv T'' \to T'$, and $\psi \leq_{\Gamma,\Gamma'} T$, hence $T >_{\Gamma,\Gamma'} T''$. Then $[t/x]^T(t'_1\ t'_2) = [m_2/y]^{\psi''} m'_1$. Therefore, by the induction hypothesis there exists a term m such that $[m_2/y]^{T''} m'_1 = m$ and $\Gamma, \Gamma' \vdash m : T''$.

Case. Suppose $t' \equiv t'_1[T'']$. Similar to the previous case.

□

Lemma 12.1.4.22. *[Redex Preserving] If* $\Gamma \vdash t : T$, $\Gamma, x : T, \Gamma' \vdash t' : T'$, *then* $|rset(t', t)| \geq |rset([t/x]^T t')|$.

Proof. This is a proof by induction on the lexicorgraphic combination (T, t') of $>_{\Gamma,\Gamma'}$ and the strict subexpression ordering. We case split on the structure of t'.

Case. Let $t' \equiv x$ or $t' \equiv y$ where y is distinct from x. Trivial.

Case. Let $t' \equiv \lambda x : T_1.t''$. Then $[t/x]^T t' \equiv \lambda x : T_1.[t/x]^T t''$. Now

$$\begin{aligned} rset(\lambda x : T_1.t'', t) &= rset(\lambda x : T_1.t'') \cup rset(t) \\ &= rset(t'') \cup rset(t) \\ &= rset(t'', t). \end{aligned}$$

We know that $t' > t''$ by the strict subexpression ordering, hence by the induction hypothesis $|rset(t'', t)| \geq_{\Gamma,\Gamma'} |rset([t/x]^T t'')|$ which implies $|rset(t', t)| \geq |rset([t/x]^T t')|$.

Case. Let $t' \equiv \Lambda X : *_l.t''$. Similar to the previous case.

Case. Let $t' \equiv inl(t'')$. We know $rset(t', t) = rset(t'', t)$. Since $t' > t''$ we can apply the induction hypothesis to obtain $|rset(t'', t)| \geq |rset([t/x]^T t'')|$. This implies $|rset(t', t)| \geq_{\Gamma,\Gamma'} |rset([t/x]^T t')|$.

Case. Let $t' \equiv inr(t'')$. Similar to the previous case.

Case. Let $t' \equiv t'_1\ t'_2$. First consider when t'_1 is not a λ-abstraction. Then

$$rset(t'_1\ t'_2, t) = rset(t'_1, t'_2, t)$$

Clearly, $t' > t'_i$ for $i \in \{1, 2\}$, hence, by the induction hypothesis $|rset(t'_i, t)| \geq |rset([t/x]^T t'_i)|$. We have two cases to consider. That is whether or not $[t/x]^T t'_1$ is a λ-abstraction or not. Suppose so. Then by Lemma 12.1.3.19 $ctype_T(x.t'_1) = \psi$ and by inversion on $\Gamma, x : T, \Gamma' \vdash t'_1\ t'_2 : T'$ there exists a type T'' such that $\Gamma, x : T, \Gamma' \vdash t_1 : T'' \to T'$. Again, by Lemma 12.1.3.19 $\psi \equiv T'' \to T'$. Thus, $ctype_T(x, t'_1) = T'' \to T'$ and $T'' \to T'$ is a subexpression of T. So by the definition of the hereditary substitution function $[t/x]^T t'_1\ t'_2 = [([t/x]^T t'_2)/y]^{T''} t''_1$, where $[t/x]^T t'_1 = \lambda y : T''.t''_1$. Hence,

$$|rset([t/x]^T t'_1\ t'_2)| = |rset([([t/x]^T t'_2)/y]^{T''} t''_1)|.$$

Now $T >_{\Gamma,\Gamma'} T''$ so by the induction hypothesis

$$\begin{array}{rcl} |rset([([t/x]^T t'_2)/y]^{T''} t''_1)| & \leq & |rset([t/x]^T t'_2, t''_1)| \\ & \leq & |rset(t'_2, t''_1, t)| \\ & = & |rset(t'_2, [t/x]^T t'_1, t)| \\ & \leq & |rset(t'_2, t'_1, t)| \\ & = & |rset(t'_1, t'_2, t)|. \end{array}$$

Suppose $[t/x]^T t'_1$ is not a λ-abstractions or $ctype_T(x, t'_1)$ is undefined.

Then

$$\begin{aligned} rset([t/x]^T(t_1'\ t_2')) &= rset([t/x]^T t_1'\ [t/x]^T t_2') \\ &= rset([t/x]^T t_1', [t/x]^T t_2'). \\ &\geq rset(t_1', t_2', t). \end{aligned}$$

Next suppose $t_1' \equiv \lambda y : T_1.t_1''$. Then

$$rset((\lambda y : T_1.t_1'')\ t_2', t) = \{(\lambda y : T_1.t_1'')\ t_2'\} \cup rset(t_1'', t_2', t).$$

By the definition of the hereditary substitution function,

$$\begin{aligned} rset([t/x]^T(\lambda y : T_1.t_1'')\ t_2') &= rset([t/x]^T(\lambda y : T_1.t_1'')\ [t/x]^T t_2') \\ &= rset((\lambda y : T_1.[t/x]^T t_1'')\ [t/x]^T t_2') \\ &= \{(\lambda y : T_1.[t/x]^T t_1'')\ [t/x]^T t_2'\} \\ &\quad \cup rset([t/x]^T t_1'') \\ &\quad \cup rset([t/x]^T t_2'). \end{aligned}$$

Since $t' > t_1''$ and $t' > t_2'$ we can apply the induction hypothesis to obtain, $|rset(t_1'', t)| \geq |rset([t/x]^T t_1'')|$ and $|rset(t_2', t)| \geq |rset([t/x]^T t_2')|$. Therefore, $|\{(\lambda y : T_1.t_1'')\ t_2'\} \cup rset(t_1'', t) \cup rset(t_2', t)| \geq |\{(\lambda y : T_1.[t/x]^T t_1'')\ [t/x]^T t_2'\} \cup rset([t/x]^T t_1'') \cup rset([t/x]^T t_2')|$.

Case. Suppose $t' \equiv t_1'[T'']$. It suffices to show that $|rset(t, t')| \geq |rset([t/x]^T t')|$.

Now

$$\begin{aligned} |rset(t, t')| &= |rset(t, t_1'[T''])| \\ &= |rset(t) \cup rset(t_1'[T''])| \\ &= |rset(t) \cup rset(t_1')| \\ &= |rset(t, t_1')|. \end{aligned}$$

and

$$|rset[t/x]^T t')| = |rset([t/x]^T(t_1'[T'']))|.$$

We have several cases to consider. Suppose t_1' and $[t/x]^T t_1'$ are not type abstractions. Then

$$\begin{aligned} |rset([t/x]^T(t_1'[T'']))| &= |rset(([t/x]^T t_1')[T''])| \\ &= |rset([t/x]^T t_1')|. \end{aligned}$$

We can see that $t' > t'_1$ so by the induction hypothesis

$$\begin{array}{rcl} |rset([t/x]^T t'_1)| & \leq & |rset(t, t'_1)| \\ & = & |rset(t, t')|. \end{array}$$

Suppose $t'_1 \equiv \Lambda X : *_l.t''_1$. Then

$$\begin{array}{rcl} |rset(t, t')| & = & |rset(t, t'_1[T''])| \\ & = & |\{t'_1[T'']\} \cup rset(t, t''_1)| \end{array}$$

and

$$\begin{array}{rcl} |rset([t/x]^T t')| & = & |rset([t/x]^T (t'_1[T'']))| \\ & = & |rset((\Lambda X : *_l.[t/x]^T t''_1)[T''])| \\ & = & |\{(\Lambda X : *_l.[t/x]^T t''_1)[T'']\} \cup rset([t/x]^T t''_1)|. \end{array}$$

Again, $t' > t'_1$ so by the induciton hypothesis $|rset([t/x]^T t''_1)| \leq |rset(t, t''_1)$.

Thus, $|rset(t, t')| \geq |rset([t/x]^T t')|$.

Suppose t'_1 is not a type abstraction, but $[t/x]^T t'_1 \equiv \lambda X : *_l.t''_1$. Then

$$\begin{array}{rcl} |rset([t/x]^T t')| & = & |rset([T''/X]t''_1)| \\ & = & |rset(t''_1)| \end{array}$$

and

$$\begin{array}{rcl} |rset(t', t)| & = & |rset(t'_1[T''], t)| \\ & = & |rset(t'_1, t)|. \end{array}$$

Since $t' > t'_1$ we can apply the induction hypthothesis to obtain

$$\begin{array}{l} |rset([t/x]^T t'_1)| = |rset(t''_1) \\ \leq |rset(t'_1, t)|. \end{array}$$

Therefore, $|rset([t/x]^T t')| \leq |rset(t', t)|$.

Case. Let $t' \equiv t'_1\ t'_2$. First consider when t'_1 is not a λ-abstraction. Then

$$rset(t'_1\ t'_2, t) = rset(t'_1, t'_2, t)$$

Clearly, $t' > t'_i$ for $i \in \{1, 2\}$, hence, by the induction hypothesis $|rset(t'_i, t)| \geq |rset([t/x]^T t'_i)|$. We have three cases to consider. That is whether or not $[t/x]^T t'_1$ is a λ-abstraction and t'_1 is not, or $ctype_T(x, t'_1)$ is undefined. Suppose t'_1 is a λ-abstraction. Then by Lemma 12.1.3.19 $ctype_T(x.t'_1) = \psi$ and by inversion on $\Gamma, x : T, \Gamma' \vdash t'_1\ t'_2 : T'$ there exists a type T'' such that $\Gamma, x : T, \Gamma' \vdash t_1 : T'' \to T'$. Again, by Lemma 12.1.3.19 $\psi \equiv T'' \to T'$. Thus, $ctype_T(x, t'_1) = T'' \to T'$ and $T'' \to T'$ is a subexpression of T. So by the definition of the hereditary substitution function $[t/x]^T t'_1\ t'_2 = [([t/x]^T t'_2)/y]^{T''} t''_1$, where $[t/x]^T t'_1 = \lambda y : T''.t''_1$. Hence,

$$|rset([t/x]^T t'_1\ t'_2)| = |rset([([t/x]^T t'_2)/y]^{T''} t''_1)|.$$

Now $T >_{\Gamma,\Gamma'} T''$ so by the induction hypothesis

$$\begin{array}{rcl} |rset([([t/x]^T t'_2)/y]^{T''} t''_1)| & \leq & |rset([t/x]^T t'_2, t''_1)| \\ & \leq & |rset(t'_2, t''_1, t)| \\ & = & |rset(t'_2, [t/x]^T t'_1, t)| \\ & \leq & |rset(t'_2, t'_1, t)| \\ & = & |rset(t'_1, t'_2, t)|. \end{array}$$

Suppose $[t/x]^T t'_1$ is not a λ-abstractions or $ctype_T(x, t'_1)$ is undefined. Then

$$\begin{array}{rcl} rset([t/x]^T(t'_1\ t'_2)) & = & rset([t/x]^T t'_1\ [t/x]^T t'_2) \\ & = & rset([t/x]^T t'_1, [t/x]^T t'_2). \\ & \leq & rset(t'_1, t'_2, t). \end{array}$$

Next suppose $t'_1 \equiv \lambda y : T_1.t''_1$. Then

$$rset((\lambda y : T_1.t''_1)\ t'_2, t) \quad = \quad \{(\lambda y : T_1.t''_1)\ t'_2\} \cup rset(t''_1, t'_2, t).$$

By the definition of the hereditary substitution function,

$$\begin{array}{rcl} rset([t/x]^T(\lambda y : T_1.t_1'')\ t_2') & = & rset([t/x]^T(\lambda y : T_1.t_1'')\ [t/x]^T t_2') \\ & = & rset((\lambda y : T_1.[t/x]^T t_1'')\ [t/x]^T t_2') \\ & = & \{(\lambda y : T_1.[t/x]^T t_1'')\ [t/x]^T t_2'\} \\ & & \cup rset([t/x]^T t_1'') \\ & & \cup rset([t/x]^T t_2'). \end{array}$$

Since $t' > t_1''$ and $t' > t_2'$ we can apply the induction hypothesis to obtain, $|rset(t_1'', t)| \geq |rset([t/x]^T t_1'')|$ and $|rset(t_2', t)| \geq |rset([t/x]^T t_2')|$. Therefore, $|\{(\lambda y : T_1.t_1'')\ t_2'\} \cup rset(t_1'', t) \cup rset(t_2', t)| \geq |\{(\lambda y : T_1.[t/x]^T t_1'')\ [t/x]^T t_2'\} \cup rset([t/x]^T t_1'') \cup rset([t/x]^T t_2')|$.

□

Lemma 12.1.4.23. *[Normality Preserving] If $\Gamma \vdash n : T$ and $\Gamma, x : T \vdash n' : T'$ then there exists a normal term n'' such that $[n/x]^T n' = n''$.*

Proof. By Lemma 12.1.4.21 we know there exists a term n'' such that $[n/x]^T n' = n''$ and by Lemma 12.1.4.22 $|rset(n', n)| \geq |rset([n/x]^T n')|$. Hence, $|rset(n', n)| \geq |rset(n'')|$, but $|rset(n', n)| = 0$. Therefore, $|rset(n'')| = 0$ which implies n'' has no redexes. □

Lemma 12.1.4.24. *[Soundness with Respect to Reduction] If $\Gamma \vdash t : T$ and $\Gamma, x : T, \Gamma' \vdash t' : T'$ then $[t/x]t' \rightsquigarrow^* [t/x]^T t'$.*

Proof. This is a proof by induction on the lexicorgraphic combination (T, t') of $>_{\Gamma,\Gamma'}$ and the strict subexpression ordering. We case split on the structure of t'. When applying the induction hypothesis we must show that the input terms to the substitution and the hereditary substitution functions are typeable. We do not explicitly state typing results that are simple conseqences of inversion.

Case. Suppose t' is a variable x or y distinct from x. Trivial in both cases.

Case. Suppose $t' \equiv \lambda y : T'.\hat{t}$. Then $[t/x]^T(\lambda y : T'.\hat{t}) = \lambda y : T'.([t/x]^T\hat{t})$. Now $t' > \hat{t}$ so we can apply the induction hypothesis to obtain $[t/x]\hat{t} \rightsquigarrow^* [t/x]^T\hat{t}$. At this point we can see that since $\lambda y : T'.[t/x]\hat{t} \equiv [t/x](\lambda y : T'.\hat{t})$ and we may conclude that $\lambda y : T'.[t/x]\hat{t} \rightsquigarrow^* \lambda y : T'.[t/x]^T\hat{t}$.

Case. Suppose $t' \equiv \Lambda X : *_l.\hat{t}$. Similar to the previous case.

Case. Suppose $t' \equiv t'_1\ t'_2$. By Lemma 12.1.4.21 there exists terms $\hat{t}'_1$ and $\hat{t}'_2$ such that $[t/x]^T t'_1 = \hat{t}'_1$ and $[t/x]^T t'_2 = \hat{t}'_2$. Since $t' > t'_1$ and $t' > t'_2$ we can apply the induction hypothesis to obtain $[t/x]t'_1 \rightsquigarrow^* \hat{t}'_1$ and $[t/x]t'_2 \rightsquigarrow^* \hat{t}'_2$. Now we case split on whether or not $\hat{t}'_1$ is a λ-abstraction and t'_1 is not, or $ctype_T(x, t'_1)$ is undefined. If $ctype_T(x, t'_1)$ is undefined or $\hat{t}'_1$ is not a λ-abstraction then $[t/x]^T t' = ([t/x]^T t'_1)\ ([t/x]^T t'_2) \equiv \hat{t}'_1\ \hat{t}'_2$. Thus, $[t/x]t' \rightsquigarrow^* [t/x]^T t'$, because $[t/x]t' = ([t/x]t'_1)\ ([t/x]t'_2)$. So suppose $\hat{t}'_1 \equiv \lambda y : T'.\hat{t}''_1$ and t'_1 is not a λ-abstraction. By Lemma 12.1.3.19 there exists a type ψ such that $ctype_T(x, t'_1) = \psi$, $\psi \equiv T'' \to T'$, and ψ is a subexpression of T, where by inversion on $\Gamma, x : T, \Gamma' \vdash t' : T'$ there exists a type T'' such that $\Gamma, x : T, \Gamma' \vdash t'_1 : T'' \to T'$. Then by the definiton of the hereditary substitution function $[t/x]^T(t'_1\ t'_2) = [\hat{t}'_2/y]^{T'}\hat{t}''_1$. Now we know $T >_{\Gamma,\Gamma'} T'$ so we can apply the induction hypothesis to obtain $[\hat{t}'_2/y]\hat{t}''_1 \rightsquigarrow^* [\hat{t}'_2/y]^{T'}\hat{t}''_1$. Now by knowing that $(\lambda y : T'.\hat{t}''_1)\ t'_2 \rightsquigarrow [\hat{t}'_2/y]\hat{t}''_1$ and by the previous fact we know $(\lambda y : T'.\hat{t}''_1)\ t'_2 \rightsquigarrow^* [\hat{t}'_2/y]^{T'}\hat{t}''_1$. We now make use of the well known result of full β-reduction. The result is stated as

$$\frac{\begin{array}{l} a \rightsquigarrow^* a' \\ b \rightsquigarrow^* b' \end{array} \qquad a'\ b' \rightsquigarrow^* c}{a\ b \rightsquigarrow^* c}$$

where a, a', b, b', and c are all terms. We apply this result by instantiating a, a', b, b', and c with $[t/x]t'_1$, $\hat{t}'_1$, $[t/x]t'_2$, $\hat{t}'_2$, and $[\hat{t}'_2/y]^{T'}\hat{t}''_1$ respectively. Therefore, $[t/x](t'_1\ t'_2) \rightsquigarrow^* [\hat{t}'_2/y]^{T'}\hat{t}''_1$.

Case. Suppose $t' \equiv t'_1[T'']$. Since $t' > t'_1$ we can apply the induction hypothesis to obtain $[t/x]t'_1 \rightsquigarrow^* [t'/x]^T t'_1$. We case split on whether or not $[t'/x]^T t'_1$ is a type abstraction and t'_1 is not. The case where it is not is trivial so we only consider the case where $[t'/x]^T t'_1 \equiv \Lambda X : *_l.s'$. Then $[t'/x]^T t' = [T'/X]s'$. Now we have $[t/x]t'_1 \rightsquigarrow^* [t'/x]^T t'_1$ and $[t/x](t'_1[T]) \equiv ([t/x]t'_1)[T] \rightsquigarrow^* ([t'/x]^T t'_1)[T] \rightsquigarrow [T/X]s'$. Thus, $[t/x]t' \rightsquigarrow^* [t'/x]^T t'$.

□

12.1.5 The Main Substitution Lemma

The definition of the interpretation of types is identical to the definition for STLC (Definition 6.2.0.7), so we do not repeat it here. Before concluding normalization we state the main substitution lemma for the interpretation of types.

Lemma 12.1.5.25. *[Substitution for the Interpretation of Types] If* $n' \in [\![T']\!]_{\Gamma,x:T,\Gamma'}$, $n \in [\![T]\!]_\Gamma$, *then* $[n/x]^T n' \in [\![T']\!]_{\Gamma,\Gamma'}$.

Proof. By Lemma 12.1.4.21 we know there exists a term $\hat{n}$ such that $[n/x]^T n' = \hat{n}$ and $\Gamma, \Gamma' \vdash \hat{n} : T'$ and by Lemma 12.1.4.23 $\hat{n}$ is normal. Therefore, $[n/x]^T n' = \hat{n} \in [\![T']\!]_{\Gamma,\Gamma'}$. □

Before moving on to proving soundness of typing and concluding normalization we need a couple of results about the interpretation of types: context weakening and type substitution. They both are used in the proof of the type soundness theorem (Theorem 12.1.6.28).

Lemma 12.1.5.26. *[Context Weakening for Interpretations of Types] If* $\Gamma, \Gamma', \Gamma'' \; Ok$ *and* $n \in [\![T]\!]_{\Gamma,\Gamma''}$*, then* $n \in [\![T]\!]_{\Gamma,\Gamma',\Gamma''}$.

Proof. This proof is by structural induction on n.

Case. Let $n \equiv x$. By the definition of the interpretation of types $\Gamma(x) = T$. Clearly, $(\Gamma, \Gamma')(x) = T$, and Lemma 12.1.1.10 gives us $\Gamma, \Gamma' \vdash T : *_p$ hence, $x \in [\![T]\!]_{\Gamma,\Gamma'}$.

Case. Let $n \equiv \lambda x : T_1.n'$. By the definition of the interpretation of types, there exists a type T_2, such that $T = T_1 \to T_2$, and $n' \in [\![T_2]\!]_{\Gamma,x:T_1}$. By the induction hypothesis, $n' \in [\![T_2]\!]_{\Gamma,\Gamma',x:T_1,}$, and by the definition of the interpretation of types $\lambda x : T_1.n' \in [\![T_1 \to T_2]\!]_{\Gamma,\Gamma'}$.

Case. Let $n \equiv n_1 \; n_2$. By the definition of the interpretation of types, there exists a type T_1, such that $n_1 \in [\![T_1 \to T_2]\!]_{\Gamma}$, and $n_2 \in [\![T_2]\!]_{\Gamma}$. By the induction hypothesis, $n_1 \in [\![T_1 \to T_2]\!]_{\Gamma,\Gamma'}$, and $n_2 \in [\![T_2]\!]_{\Gamma,\Gamma'}$. Thus, by the definition of the interpretation of types $n_1 n_2 \in [\![T_2]\!]_{\Gamma,\Gamma'}$.

Case. Let $n \equiv \Lambda X : *_p.n'$. By the definition of the interpretation of types, there exists a type T', such that $n' \in [\![T']\!]_{\Gamma,X:*_p}$, and by the induction hypothesis $n' \in [\![T']\!]_{\Gamma,X:*_p,\Gamma'}$. By the definition of the interpretation of types $\Lambda X : *_p.n' \in [\![\forall X : *_p.T']\!]_{\Gamma,\Gamma'}$.

Case. Let $n \equiv n'[T']$. By the definition of the interpretation of types, there exists a type T'' and l, such that $T = [T'/X]T''$, $\Gamma \vdash T' : *_l$, and $n' \in [\![\forall X : *_l.T'']\!]_\Gamma$. By the induction hypothesis $n' \in [\![\forall X : *_l.T'']\!]_{\Gamma,\Gamma'}$. We know, $\Gamma \vdash T' : *_k$, for some $k \leq l$, so by Lemma 12.1.1.10, $\Gamma, \Gamma' \vdash T' : *_k$, hence $\Gamma \vdash T' : *_l$. Thus, $n[T'] \in [\![[T'/X]T'']\!]_{\Gamma,\Gamma'}$.

□

Lemma 12.1.5.27. *[Type Substitution for the Interpretation of Types]*
If $n \in [\![T']\!]_{\Gamma,X:*_l,\Gamma'}$ *and* $\Gamma \vdash T : *_l$, *then* $[T/X]n \in [\![[T/X]T']\!]_{\Gamma,[T/X]\Gamma'}$.

Proof. This proof is by structural induction on n.

Case. n is a variable y. Clearly, $[T/X]n \equiv [T/X]y = y \in [\![T']\!]_{\Gamma,X:*_l,\Gamma'}$, and $(\Gamma, [T/X]\Gamma')(y) = [T/X]T'$. Also, we have $\Gamma, [T/X]\Gamma' \vdash [T/X]T' : *_p$ for some p, by Lemma 12.1.1.8. Hence, by the definition of the interpretation of types, $y \in [\![[T/X]T']\!]_{\Gamma,[T/X]\Gamma'}$.

Case. Let $n \equiv \lambda y : \psi.n'$. By the definition of the interpretation of types $T' \equiv \psi \to \psi'$. By the induction hypothesis $[T/X]n' \in [\![[T/X]\psi']\!]_{\Gamma,\Gamma',y:[T/X]\psi}$. Again by the definition of the interpretation of types $\lambda y : [T/X]\psi.[T/X]n' \equiv [T/X](\lambda y : \psi.n') \in [\![[T/X]T']\!]_{\Gamma,[T/X]\Gamma'}$.

Case. Let $n \equiv n_1\ n_2$. By the definition of the interpretation of types $T' \equiv \psi$, $n_1 \in [\![\psi' \to \psi]\!]_{\Gamma,X:*_q,\Gamma'}$, and $n_2 \in [\![\psi']\!]_{\Gamma,X:*_q,\Gamma'}$. By the induction hypothesis $[T/X]n_1 \in [\![[T/X](\psi' \to \psi)]\!]_{\Gamma,[T/X]\Gamma'}$ and $[T/X]n_2 \in [\![[T/X]\psi']\!]_{\Gamma,[T/X]\Gamma'}$.

Now by the definition of the interpretation of types $([T/X]n_1)([T/X]n_2) \in [\![[T/X]\psi]\!]_{\Gamma,[T/X]\Gamma'}$, since $[T/X]n_1$, cannot be a λ-abstraction.

Case. Let $n \equiv \Lambda Y : *_q.n'$. By the definition of the interpretation of types $T' = \forall Y : *_q.\psi$ and $n' \in [\![\psi]\!]_{\Gamma,X:*_l,\Gamma',Y:*_q}$. By the induction hypothesis $[T/X]n' \in [\![[T/X]\psi]\!]_{\Gamma,[T/X]\Gamma',Y:*_q}$ and by the definition of the interpretation of types $\Lambda Y : *_q.[T/X]n' \in [\![\forall Y : *_q.[T/X]\psi]\!]_{\Gamma,[T/X]\Gamma'}$ which is equivalent to $[T/X](\Lambda Y : *_q.n') \in [\![[T/X](\forall Y : *_q.\psi)]\!]_{\Gamma,[T/X]\Gamma'}$.

Case. Let $n \equiv n'[\psi]$. By the definition of the interpretation of types $T' = [\psi/Y]\psi'$, for some Y, ψ, and there exists a q such that $\Gamma, X : *_l, \Gamma' \vdash \psi : *_q$, and $n' \in [\![\forall Y : *_q.\psi']\!]_{\Gamma,X:*_l,\Gamma'}$. By the induction hypothesis $[T/X]n' \in [\![[T/X](\forall Y : *_q.\psi')]\!]_{\Gamma,[T/X]\Gamma'}$. Therefore, by the definition of the interpretation of types $([T/X]n')[\psi] \in [\![[\psi/Y]([T/X]\psi')]\!]_{\Gamma,[T/X]\Gamma'}$, which is equivalent to $[T/X](n'[\psi]) \in [\![[T/X]([\psi/Y]\psi')]\!]_{\Gamma,[T/X]\Gamma'}$.

□

12.1.6 Concluding Normalization

We are now ready to present our main result. The next theorem shows that the type assignment rules are sound with respect to the interpretation of types.

Theorem 12.1.6.28. *[Type Soundness] If* $\Gamma \vdash t : T$ *then* $t \in [\![T]\!]_\Gamma$.

Proof. This is a proof by induction on the structure of the typing derivation of t.

Case.

$$\frac{\Gamma(x) = T \qquad \Gamma \; Ok}{\Gamma \vdash x : T}$$

By regularity $\Gamma \vdash T : *_l$ for some l, hence $[\![T]\!]_\Gamma$ is nonempty. Clearly, $x \in [\![T]\!]_\Gamma$ by the definition of the interpretation of types.

Case.

$$\frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x : T_1.t : T_1 \to T_2}$$

By the induction hypothesis $t \in [\![T_2]\!]_{\Gamma, x:T_1}$ and by the definition of the interpretation of types $t \rightsquigarrow^! n \in [\![T_2]\!]_{\Gamma, x:T_1}$ and $\Gamma, x : T_1 \vdash n : T_2$. Thus, by applying the λ-abstraction type-checking rule, $\Gamma \vdash \lambda x : T_1.n : \Pi x : T_1.T_2$ so by the definition of the interpretation of types $\lambda x : T_1.n \in [\![T_1 \to T_2]\!]_\Gamma$. Thus, according to the definition of the interpretation of types $\lambda x : T_1.t \rightsquigarrow^! \lambda x : T_1.n \in [\![T_1 \to T_2]\!]_\Gamma$.

Case.

$$\frac{\Gamma \vdash t_1 : T_2 \to T_1 \qquad \Gamma \vdash t_2 : T_2}{\Gamma \vdash t_1 \; t_2 : T_1}$$

By the induction hypothesis $t_1 \rightsquigarrow^! n_1 \in [\![T_2 \to T_1]\!]_\Gamma$, $t_2 \rightsquigarrow^! n_2 \in [\![T_2]\!]_\Gamma$, $\Gamma \vdash T_2 \to T_1 : *_p$, and $\Gamma \vdash T_2 : *_q$. Inversion on the arrow-type kind-checking rule yields, $\Gamma \vdash T_1 : *_r$, and by Lemma 12.1.1.10, $\Gamma, x : T_2, \Gamma' \vdash T_1 : *_r$.

Now we know from above that $n_1 \in [\![T_2 \to T_1]\!]_\Gamma$ and $n_2 \in [\![T_2]\!]_\Gamma$, hence

$\Gamma \vdash n_1 : T_2 \to T_1$ and $\Gamma \vdash n_2 : T_2$. It suffices to show that $n_1\ n_2 \in [\![T_2]\!]_\Gamma$. Clearly, $n_1\ n_2 = [n_1/z](z\ n_2)$ for some variable $z \notin FV(n_1, n_2)$. Lemma 12.1.4.21, Lemma 12.1.4.24, and Lemma 12.1.4.23 allow us to conclude that $[n_1/z](z\ n_2) \rightsquigarrow^* [n_1/z]^{T_2 \to T_1}(z\ n_2)$, $\Gamma \vdash [n_1/z]^{T_2 \to T_1}(z\ n_2) : T_2$, and $[n_1/z]^{T_2 \to T_1}(z\ n_2)$ is normal. Thus, $t_1\ t_2 \rightsquigarrow^* n_1\ n_2 = [n_1/z](z\ n_2) \rightsquigarrow^! [n_1/z]^{T_2 \to T_1}(z\ n_2) \in [\![T_2]\!]_\Gamma$.

Case.

$$\frac{\Gamma, X : *_p \vdash t : T}{\Gamma \vdash \Lambda X : *_p.t : \forall X : *_p.T}$$

By the induction hypothesis and definition of the interpretation of types $t \in [\![T]\!]_{\Gamma, X:*_p}$, $t \rightsquigarrow^! n \in [\![T]\!]_{\Gamma, X:*_p}$ and $\Lambda X : *_p.n \in [\![T]\!]_\Gamma$. Again, by definition of the interpretation of types $\Lambda X : *_p.t \rightsquigarrow^! \Lambda X : *_p.n \in [\![T]\!]_\Gamma$.

Case.

$$\frac{\Gamma \vdash t : \forall X : *_l.T_1 \qquad \Gamma \vdash T_2 : *_l}{\Gamma \vdash t[T_2] : [T_2/X]T_1}$$

By the induction hypothesis $t \in [\![\forall X : *_l.T_1]\!]_\Gamma$ and by the definition of the interpretation of types we know $t \rightsquigarrow^! n \in [\![\forall X : *_l.T_1]\!]_\Gamma$. We case split on whether or not n is a type abstraction. If not then again, by the definition of the interpretation of types $n[T_2] \in [\![[T_2/X]T_1]\!]_\Gamma$, therefore $t \in [\![[T_2/X]T_1]\!]_\Gamma$. Suppose $n \equiv \Lambda X : *_l.n'$. Then $t[T_2] \rightsquigarrow^* (\Lambda X : *_l.n')[T_2] \rightsquigarrow [T_2/X]n'$. By the definition of the interpretation of types $n' \in [\![T_1]\!]_{\Gamma, X:*_l}$. Therefore, by

Lemma 12.1.5.27 $[T_2/X]n' \in [\![[T_2/X]T_1]\!]_\Gamma$.

□

Therefore, we conclude normalization of SSF.

Corollary 12.1.6.29. *[Normalization] If $\Gamma \vdash t : T$, then there exists a normalform n, such that $t \rightsquigarrow^! n$.*

12.2 Stratified System F^+

Stratified System F^+ (SSF^+) is an extension of SSF with sum types denoted $T_1 + T_2$, whose elimination form *case* t *of* $x_1.t_1, x_2.t_2$ is used to case split on a whether or not term t with a sum type is truly x_1 of type T_1, or else x_2 of type T_2. We consider sum types with so-called commuting conversions, which allow independent cases to be permuted past each other (see Fig 51 below). Commuting conversions are well-known to pose technical difficulties for normalization proofs based on reducibility (see [133] and Chapter 10 of [60]). We will see that they can be handled straightforwardly with hereditary substitution.

The syntax, reduction rules, and commuting conversions for SSF^+ can be found in Figure 51. The kind-assignment rules are defined in Figure 53 and the type-assignment rules in defined in Figure 54. The kinding/typing relations depend on well-formed contexts which are defined in Figure 52. To ensure substitutions over contexts behave in an expected manner, we rename variables as necessary to ensure contexts have at most one declaration per variable. Lastly, the basic meta-theoretic results are used throughout this chapter (we omit their proofs, because they are similar to the proofs for SSF):

Syntax:

$$\begin{array}{lcl} K & := & *_0 \mid *_1 \mid \ldots \\ T & := & X \mid T \to T \mid \forall X : K.T \mid T + T \\ t & := & x \mid \lambda x : T.t \mid t\ t \mid \Lambda X : K.t \mid t[T] \mid inl(t) \mid inr(t) \mid \text{case } t \text{ of } x.t,x.t \end{array}$$

Reduction Rules:

$$\begin{array}{llll} (\Lambda X : *_p.t)[T] & \rightsquigarrow & [T/X]t & \text{case } inl(t) \text{ of } x.t_1,x.t_2 \rightsquigarrow [t/x]t_1 \\ (\lambda x : T.t)t' & \rightsquigarrow & [t'/x]t & \text{case } inr(t) \text{ of } x.t_1,x.t_2 \rightsquigarrow [t/x]t_2 \end{array}$$

Commuting Conversions:

$$\begin{array}{lcl} (\text{case } t \text{ of } x.t_1,x.t_2)\ t' & \rightsquigarrow & \text{case } t \text{ of } x.(t_1\ t'),x.(t_2\ t') \\ \text{case } (\text{case } t \text{ of } x.t_1,x.t_2) \text{ of } y.s_1,y.s_2 & \rightsquigarrow & \text{case } t \text{ of} \\ & & \quad x.(\text{case } t_1 \text{ of } y.s_1,y.s_2), \\ & & \quad x.(\text{case } t_2 \text{ of } y.s_1,y.s_2) \end{array}$$

Figure 51. Syntax, Reduction Rules, and Commuting Conversions for SSF^+

$$\frac{}{\cdot\ Ok} \qquad \frac{\Gamma\ Ok}{\Gamma, X : *_p\ Ok} \qquad \frac{\Gamma \vdash T : *_p \qquad \Gamma\ Ok}{\Gamma, x : T\ Ok}$$

Figure 52. Well-formedness of Contexts for SSF^+

Lemma 12.2.0.1. *If* $\Gamma \vdash T : *_p$ *then* $\Gamma\ Ok$.

Proof. This holds by straightforward induction on the form of the assumed kinding derivation. □

Lemma 12.2.0.2. *[Level Weakening for Kinding] If* $\Gamma \vdash T : *_r$ *and* $r < s$ *then* $\Gamma \vdash T : *_s$.

$$\frac{\Gamma \vdash T_1 : *_p \qquad \Gamma \vdash T_2 : *_q}{\Gamma \vdash T_1 \to T_2 : *_{max(p,q)}} \qquad \frac{\Gamma, X : *_q \vdash T : *_p}{\Gamma \vdash \forall X : *_q . T : *_{max(p,q)+1}}$$

$$\frac{\Gamma \vdash T_1 : *_p \qquad \Gamma \vdash T_2 : *_q}{\Gamma \vdash T_1 + T_2 : *_{max(p,q)}} \qquad \frac{\begin{array}{c}\Gamma(X) = *_p \\ \Gamma\ Ok \qquad p \leq q\end{array}}{\Gamma \vdash X : *_q}$$

Figure 53. SSF$^+$ Kinding Rules

$$\frac{\begin{array}{c}\Gamma(x) = T \\ \Gamma\ Ok\end{array}}{\Gamma \vdash x : T} \qquad \frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x : T_1 . t : T_1 \to T_2} \qquad \frac{\Gamma \vdash t_1 : T_1 \to T_2 \qquad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\, t_2 : T_2}$$

$$\frac{\Gamma, X : *_l \vdash t : T}{\Gamma \vdash \Lambda X : *_l . t : \forall X : *_l . T} \qquad \frac{\begin{array}{c}\Gamma \vdash t : \forall X : *_l . T_1 \\ \Gamma \vdash T_2 : *_l\end{array}}{\Gamma \vdash t[T_2] : [T_2/X]T_1} \qquad \frac{\begin{array}{c}\Gamma \vdash t : T_1 \\ \Gamma \vdash T_2 : *_p\end{array}}{\Gamma \vdash inl(t) : T_1 + T_2}$$

$$\frac{\begin{array}{c}\Gamma \vdash t : T_2 \\ \Gamma \vdash T_1 : *_p\end{array}}{\Gamma \vdash inr(t) : T_1 + T_2} \qquad \frac{\begin{array}{c}\Gamma \vdash t : T_1 + T_2 \\ \Gamma, x : T_1 \vdash t_1 : T \\ \Gamma, x : T_2 \vdash t_2 : T\end{array}}{\Gamma \vdash \text{case } t \text{ of } x.t_1, x.t_2 : T}$$

Figure 54. SSF$^+$ Type-Assignment Rules

Proof. This holds by straightforward induction on the form of the assumed kinding derivation. □

Lemma 12.2.0.3. *[Substitution for Kinding, Context-Ok] Suppose* $\Gamma \vdash T' : *_p$. *If* $\Gamma, X : *_p, \Gamma' \vdash T : *_q$ *with a derivation of depth* d, *then* $\Gamma, [T'/X]\Gamma' \vdash [T'/X]T : *_q$ *with a derivation of depth* d. *Also, if* $\Gamma, X : *_p, \Gamma'$ *Ok with a derivation of depth* d, *then* $\Gamma, [T'/X]\Gamma'$ *Ok with a derivation of depth* d.

Proof. This holds by straightforward induction on the d. □

Lemma 12.2.0.4. *[Regularity] If* $\Gamma \vdash t : T$ *then* $\Gamma \vdash T : *_p$ *for some* p.

Proof. This holds by straightforward induction on the form of the assumed typing derivation. □

12.2.1 Ordering on Types

In this section we define the ordering on types. It is a straightforward extension of the ordering for SSF.

Definition 12.2.1.5.

Suppose $op \in \{+, \rightarrow\}$. *Then the ordering* $>_\Gamma$ *is defined as the least relation satisfying the universal closures of the following formulas:*

$$\begin{array}{lll} T_1 \ op \ T_2 & >_\Gamma & T_1 \\ T_1 \ op \ T_2 & >_\Gamma & T_2 \\ \forall X : *_l . T & >_\Gamma & [T'/X]T \ \textit{where} \ \Gamma \vdash T' : *_l . \end{array}$$

Theorem 12.2.1.6. *[Well-Founded Ordering] The ordering* $>_\Gamma$ *is well-founded on types* T *such that* $\Gamma \vdash T : *_l$ *for some* l.

Proof. The depth function, defined in the following definition, is used in the following proof.

Definition 12.2.1.7.

The depth of a type T is defined as follows:

$$\begin{array}{lcl} depth(X) & = & 1 \\ depth(T \to T') & = & depth(T) + depth(T') \\ depth(T + T') & = & depth(T) + depth(T') \\ depth(\forall X : *_l.T) & = & depth(T) + 1 \end{array}$$

We define the metric (l, d) in lexicographic combination, where l is the level of a type T and d is the depth of T.

Lemma 12.2.1.8. *[Well-Founded Measure] If $T >_\Gamma T'$ then $(l, d) > (l', d')$, where $\Gamma \vdash T : *_l$, $depth(T) = d$, $\Gamma \vdash T : *_{l'}$, and $depth(T') = d'$.*

Proof. This holds by straightforward induction on the structure of T. □

Finally, the proof of well-foundedness of $>_\Gamma$. If there exists an infinite decreasing sequence using our ordering on types, then there is an infinite decreasing sequence using our measure by Lemma 12.2.1.8, but that is impossible. □

We need transitivity in a number of places in the proof of the main substitution lemma.

Lemma 12.2.1.9. *[Transitivity of $>_\Gamma$] Let T, T', and T'' be kindable types.*

If $T >_\Gamma T'$ and $T' >_\Gamma T''$ then $T >_\Gamma T''$.

Proof. This proof is similar to the proof for the ordering used in the proof of normalization of SSF (Lemma 12.1.2.13). □

$app_T\ t_1\ t_2 = t_1\ t_2$

Where t_1 is not a λ-abstraction or a case construct.

$app_T\ (\lambda x : T'.t_1)\ t_2 = [t_2/x]^{T'} t_1$

$app_T\ (\text{case}\ t_0\ \text{of}\ x.t_1,x.t_2)\ t = \text{case}\ t_0\ \text{of}\ x.(app_T\ t_1\ t),x.(app_T\ t_2\ t)$

$rcase_T\ t_0\ y\ t_1\ t_2 = \text{case}\ t_0\ \text{of}\ y.t_1,y.t_2$

Where t_0 is not an inject-left or an inject-right term or a case construct.

$rcase_T\ inl(t')\ y\ t_1\ t_2 = [t'/y]^{T_1}\ t_1$

$rcase_T\ inr(t')\ y\ t_1\ t_2 = [t'/y]^{T_2}\ t_2$

$rcase_T\,(\text{case}\ t'_0\ \text{of}\ x.t'_1,x.t'_2)\,y\,t_1\,t_2 =$
$\text{case}\ t'_0\ \text{of}\ x.(rcase_T\,t'_1\,y\,t_1\,t_2),x.(rcase_T\,t'_2\,y\,t_1\,t_2)$

Figure 55. Hereditary Substitution Function for Stratified System F^+

12.2.2 Hereditary Substitution

The definition of the hereditary substitution function for SSF^+ is in Figure 55 and Figure 56. First, one should read this definition as a mutually recursive function in terms of the hereditary substitution function $[t/x]^T t'$, the application reduction function $app_T\ t_1\ t_2$, and case construct reduction function $rcase_T\ t_0\ x\ t_1\ t_2$. The definitions of all these functions depend the same definition of $ctype_T(x,t)$ as the proof of normalization of SSF (Definition 12.1.3.17). So we do not repeat it here.

The hereditary substitution function is an extension of the hereditary substitution function for SSF. The most interesting case of the definition is when a commuting conversion is created as a result of substitution. In this case we know by the $ctype_T$ function that the head of the application is in the form $x\,t_1 \cdots t_2$. Furthermore, we

$[t/x]^T x = t$

$[t/x]^T y = y$

Where y is a variable distinct from x.

$[t/x]^T(\lambda y : T'.t') = \lambda y : T'.([t/x]^T t')$

$[t/x]^T(\Lambda X : *_l.t') = \Lambda X : *_l.([t/x]^T t')$

$[t/x]^T inr(t') = inr([t/x]^T t')$

$[t/x]^T inl(t') = inl([t/x]^T t')$

$[t/x]^T(t_1\ t_2) = ([t/x]^T t_1)\ ([t/x]^T t_2)$

Where $([t/x]^T t_1)$ is not a λ-abstraction or a case construct, or both $([t/x]^T t_1)$ and t_1 are λ-abstractions or case constructs, or $ctype_T(x, t_1)$ is undefined.

$[t/x]^T(t_1\ t_2) = [([t/x]^T t_2)/y]^{T''} s_1'$

Where $([t/x]^T t_1) = \lambda y : T''.s_1'$ for some y, s_1', and T'' and $ctype_T(x, t_1) = T'' \to T'$.

$[t/x]^T(t_1\ t_2) = \text{case } w \text{ of } y.(app_T\ r\ ([t/x]^T t_2)), y.(app_T\ s\ ([t/x]^T t_2))$

Where $[t/x]^T t_1 = \text{case } w \text{ of } y.r, y.s$ for some terms w, r, s and variable y, and $ctype_T(x, t_1) = T'' \to T'$.

$[t/x]^T(t'[T']) = ([t/x]^T t')[T']$

Where $[t/x]^T t'$ is not a type abstraction or t' and $[t/x]^T t'$ are type abstractions.

$[t/x]^T(t'[T']) = [T'/X]s_1'$

Where $[t/x]^T t' = \Lambda X : *_l.s_1'$, for some X, s_1' and $\Gamma \vdash T' : *_q$, such that, $q \leq l$ and
t' is not a type abstraction.

$[t/x]^T(\text{case } t_0 \text{ of } y.t_1, y.t_2) = \text{case } ([t/x]^T t_0) \text{ of } y.([t/x]^T t_1), y.([t/x]^T t_2)$

Where $([t/x]^T t_0)$ is not an inject-left or an inject-right term or a case construct, or $([t/x]^T t_0)$ and t_0 are both inject-left or inject-right terms or case constructs, or $ctype_T(x, t_0)$ is undefined.

$[t/x]^T(\text{case } t_0 \text{ of } y.t_1, y.t_2) = rcase_T\ ([t/x]^T t_0)\ y\ ([t/x]^T t_1)\ ([t/x]^T t_2)$

Where $([t/x]^T t_0)$ is an inject-left or an inject-right term or a case construct and $ctype_T(x, t_0) = T_1 + T_2$.

Figure 56. Hereditary Substitution Function for Stratified System F^+ Continued

know that applying the hereditary substitution function to the head of the application results in a case construct. So we recursively reduce the created redex in the same way the reduction rules do, but when we push the argument into the branches of the resulting case construct more redexes may be created. So to handle recursively reducing all of the newly created redexes in the branches we call the application reduction function app_T. This function reduces redexes by recursively calling itself and the hereditary substitution function. The remaining cases where new redexes are potentially created are similar to these cases. The function $rcase_T$ handles reducing case constructs.

12.2.3 Main Properties

We are now to the point where we can prove the properties of the hereditary substitution function. Just as we did in the proof for SSF we first must show the properties of $ctype_T$ hold.

Lemma 12.2.3.10. *[Properties of $ctype_T$]*

i. *If $ctype_T(x,t) = T'$ then $head(t) = x$ and T' is a subexpression of T.*

ii. *If $\Gamma, x : T, \Gamma' \vdash t : T'$ and $ctype_T(x,t) = T''$ then $T' \equiv T''$.*

iii. *If $\Gamma, x : T, \Gamma' \vdash t_1\ t_2 : T'$, $\Gamma \vdash t : T$, $[t/x]^T t_1 = \lambda y : T_1.q$, and t_1 is not then there exists a type T such that $ctype_T(x,t_1) = T$.*

iv. *If $\Gamma, x : T, \Gamma' \vdash t_1\ t_2 : T'$, $\Gamma \vdash t : T$, $[t/x]^T t_1 = case\ t'_0\ of\ y.t'_1,y.t'_2$, and t_1 is not then there exists a type T such that $ctype_T(x,t_1) = T$.*

v. *If $\Gamma, x : T, \Gamma' \vdash case\ t_0\ of\ y.t_1,y.t_2 : T'$, $\Gamma \vdash t : T$, $[t/x]^T t_0 = case\ t'_0\ of\ z.t'_1,z.t'_2$,*

and t_0 is not then there exists a type T such that $ctype_T(x, t_0) = T$.

vi. If $\Gamma, x : T, \Gamma' \vdash$ case t_0 of $y.t_1$,$y.t_2 : T'$, $\Gamma \vdash t : T$, $[t/x]^T t_0 = inl(t')$, and t_0 is not then there exists a type T such that $ctype_T(x, t_0) = T$.

vii. If $\Gamma, x : T, \Gamma' \vdash$ case t_0 of $y.t_1$,$y.t_2 : T'$, $\Gamma \vdash t : T$, $[t/x]^T t_0 = inr(t')$, and t_0 is not then there exists a type T such that $ctype_T(x, t_0) = T$.

Proof. The first two cases are equivalent to the proof of the properties for SSF (Lemma 12.1.3.19). Cases three through five are similar, so we only give the proof of part three. This is a proof by induction on the structure of $t_1\ t_2$.

The only possiblities for the form of t_1 is x or $\hat{t}_1\ \hat{t}_2$. All other forms would not result in $[t/x]^T t_1$ being a λ-abstraction and t_1 not. If $t_1 \equiv x$ then there exist a type T'' such that $T \equiv T'' \to T'$ and $ctype_T(x, x\ t_2) = T'$ when $ctype_T(x, x) = T \equiv T'' \to T'$ in this case. We know T'' to exist by inversion on $\Gamma, x : T, \Gamma' \vdash t_1\ t_2 : T'$.

Now suppose $t_1 \equiv (\hat{t}_1\ \hat{t}_2)$. Now knowing t_1' to not be a λ-abstraction implies that $\hat{t}_1$ is also not a λ-abstraction or $[t/x]^T t_1$ would be an application instead of a λ-abstraction. So it must be the case that $[t/x]^T \hat{t}_1$ is a λ-abstraction and $\hat{t}_1$ is not. Since $t_1\ t_2 > t_1$ we can apply the induction hypothesis to obtain there exists a type T such that $ctype_T(x, \hat{t}_1) = T$. Now by inversion on $\Gamma, x : T, \Gamma' \vdash t_1\ t_2 : T'$ we know there exists a type T'' such that $\Gamma, x : T, \Gamma' \vdash t_1 : T'' \to T'$. We know $t_1 \equiv (\hat{t}_1\ \hat{t}_2)$ so by inversion on $\Gamma, x : T, \Gamma' \vdash t_1 : T'' \to T'$ we know there exists a type T'' such that $\Gamma, x : T, \Gamma' \vdash \hat{t}_1 : T'' \to (T'' \to T')$. By part two of Lemma 12.2.3.10 we know $T \equiv T'' \to (T'' \to T')$ and $ctype_T(x, t_1) = ctype_T(x, \hat{t}_1\ \hat{t}_2) = T'' \to T'$ when

$ctype_T(x, \hat{t}_1) = T'' \to (T'' \to T')$, because we know $ctype_T(x, \hat{t}_1) = T$. □

We now move on to proving the main properties of the hereditary substitution function. First, we show that for typeable terms it is a total function and the output maintains the same type as the principle term of substitution.

Lemma 12.2.3.11. *[Total and Type Preserving] Suppose* $\Gamma \vdash t : T$ *and* $\Gamma, x : T, \Gamma' \vdash t' : T'$. *Then there exists a term* t'' *such that* $[t/x]^T t' = t''$ *and* $\Gamma, \Gamma' \vdash t'' : T'$.

Proof. This is a proof by induction on the lexicorgraphic combination (T, t') of $>_{\Gamma,\Gamma'}$ and the strict subexpression ordering. We case split on t'.

Case. Suppose t' is either x or a variable y distinct from x. Trivial in both cases.

Case. Suppose $t' \equiv \lambda y : T_1.t'_1$. By inversion on the typing judgement we know $\Gamma, x : T, \Gamma', y : T_1 \vdash t'_1 : T_2$. We also know $t' > t'_1$, hence we can apply the induction hypothesis to obtain $[t/x]^T t'_1 = \hat{t}'_1$ and $\Gamma, \Gamma', y : T_1 \vdash \hat{t} : T_2$ for some term $\hat{t}'_1$. By the definition of the hereditary substitution function $[t/x]^T t' = \lambda y : T_1.[t/x]^T t'_1 = \lambda y : T_1.\hat{t}'_1$. It suffices to show that $\Gamma, \Gamma' \vdash \lambda y : T_1.\hat{t}'_1 : T_1 \to T_2$. By simply applying the λ-abstraction typing rule using $\Gamma, \Gamma', y : T_1 \vdash \hat{t} : T_2$ we obtain $\Gamma, \Gamma' \vdash \lambda y : T_1.\hat{t}'_1 : T_1 \to T_2$.

Case. Suppose $t' \equiv \Lambda X : *_l.t'_1$. Similar to the previous case.

Case. Suppose $t' \equiv t'_1\ t'_2$. By inversion we know $\Gamma, x : T, \Gamma' \vdash t'_1 : T'' \to T'$ and $\Gamma, x : T, \Gamma' \vdash t'_2 : T''$ for some types T' and T''. Clearly, $t' > t'_i$ for $i \in \{1, 2\}$. Thus, by the induction hypothesis there exists terms m_1 and m_2 such that $[t/x]^T t'_i = m_i$, $\Gamma, \Gamma' \vdash m_1 : T'' \to T'$ and $\Gamma, \Gamma' \vdash m_2 : T''$ for

$i \in \{1, 2\}$. We case split on whether or not m_1 is a λ-abstraction, a case construct and t_1' is not, or $ctype_T(x, t_1')$ is undefined. We only consider the non-trivial cases when $m_1 \equiv \lambda y : T''.m_1'$ and t_1' is not a λ-abstraction or $m_1 \equiv$ case m_0' of $y.m_1'$,$y.m_2'$, $ctype_T(x, t_1') = T'' \to T'$, and t_1' is not a case construct. Suppose the former. Now by Lemma 12.2.3.10 it is the case that there exists a T such that $ctype_T(x, t_1') = T$, $T \equiv T'' \to T'$, and T is a subexpression of T, hence $T >_{\Gamma,\Gamma'} T''$. Then $[t/x]^T(t_1'\ t_2') = [m_2/y]^{T''}m_1'$. Therefore, by the induction hypothesis there exists a term m such that $[m_2/y]^{T''}m_1' = m$ and $\Gamma, \Gamma' \vdash m : T''$.

Suppose $m_1 \equiv$ case m_0' of $y.m_1'$,$y.m_2'$ and t_1' is not a case construct. Now $[t/x]^T t' =$ case m_0' of $y.app_T\ m_1'\ m_2$,$y.app_T\ m_2'\ m_2$. By inversion on $\Gamma, \Gamma' \vdash m_1 : T'' \to T'$ we know there exists terms T_1 and T_2 such that $\Gamma, \Gamma' \vdash m_0' : T_1 + T_2$ and $\Gamma, \Gamma', y : T_i \vdash m_i' : T'' \to T'$ for $i \in \{1, 2\}$. It suffcies to show that there exists terms q and q' such that $app_T\ m_1'\ m_2 = q$ and $app_T\ m_2'\ m_2 = q'$. To obtain this result we prove the following proposition. Note that $ctype_T(x, t_1') = T'' \to T'$ which by Lemma 12.2.3.10 is equivalent to $T'' \to T'$ and is a subexpression of T, hence $T >_{\Gamma,\Gamma'} T''$ and $T >_{\Gamma,\Gamma'} T'$.

Proposition. For all $\Gamma \vdash m_2 : T''$ and $\Gamma \vdash m_1' : T'' \to T'$ there exists a term q such that $app_T\ m_1'\ m_2 = q$ and $\Gamma \vdash q : T'$.

We prove this by nested induction on the ordering (T, t', m_1') and case splitting on the structure of m_1'.

Case. Suppose m_1' is neither a λ-abstraction or a case construct. Then

$app_T\ m_1'\ m_2 = m_1'\ m_2$. Take $m_1'\ m_2$ for q and by applying the application typing rule we know $\Gamma \vdash m_1'\ m_2 : T'$.

Case. Suppose $m_1' \equiv \lambda z : T''.m_1''$. Then $app_T\ m_1'\ m_2 = [m_2/z]^{T''}m_1''$. By inversion on the assumption $\Gamma \vdash m_1' : T'' \to T'$ we know $\Gamma, z : T'' \vdash m_1'' : T'$. Since $T >_\Gamma T''$ we can apply the outter induction hypothesis to obtain there there exists a q such that $[m_2/z]^{T''}m_1'' = q$ and $\Gamma \vdash q : T'$. Therefore, $app_T\ m_1'\ m_2 = q$.

Case. Suppose $m_1' \equiv$ case m_0'' of $z.m_1''$,$z.m_2''$. Then

$app_T\ m_1'\ m_2 =$ case m_0'' of $z.app_T\ m_1''\ m_2$,$z.app_T\ m_2''\ m_2$. By inversion on the assumption $\Gamma \vdash m_1' : T'' \to T'$ we know there exists types T_1 and T_2 such that $\Gamma \vdash m_0'' : T_1 + T_2$, $\Gamma, z : T_1 \vdash m_1'' : T'' \to T'$ and $\Gamma, z : T_2 \vdash m_2'' : T'' \to T'$. Since $m_1' > m_1''$ and $m_1' > m_2''$ we can apply the inner induction hypothesis to obtain there exists terms q' and q'' such that $app_T\ m_1''\ m_2 = q'$, $\Gamma, z : T_1 \vdash q' : T'$, $app_T\ m_1''\ m_2 = q''$ and $\Gamma, z : T_2 \vdash q'' : T'$. Hence, $app_T\ m_1'\ m_2 =$ case m_0'' of $z.app_T\ m_1''\ m_2$,$z.app_T\ m_2''\ m_2 =$ case m_0'' of $z.q'$,$z.q''$. It suffices to to show that $\Gamma \vdash$ case m_0'' of $z.q'$,$z.q'' : T'$. This is a simple consequence of applying the case-construct typing rule using $\Gamma \vdash m_0'' : T_1 + T_2$, $\Gamma, z : T_1 \vdash q' : T'$, and $\Gamma, z : T_2 \vdash q'' : T'$.

By the previous proposition there exists terms q and q' such that $[t/x]^T t' =$ case m_0' of $y.app_T\ m_1'\ m_2$,$y.app_T\ m_2'\ m_2 =$ case m_0' of $y.q$,$y.q'$, where $app_T\ m_1'\ m_2 = q$, $\Gamma, \Gamma', y : T_1 \vdash q : T'$, $app_T\ m_1'\ m_2 = q'$ and $\Gamma, \Gamma', y : T_2 \vdash q' : T'$. It suffices to show that $\Gamma, \Gamma' \vdash$ case m_0' of $y.q$,$y.q' : T'$. From

above we know that $\Gamma, \Gamma' \vdash m_0' : T_1 + T_2$, $\Gamma, \Gamma', y : T_1 \vdash q : T'$ and $\Gamma, \Gamma', y : T_2 \vdash q' : T'$. Thus, by applying the case-construct typing rule we obtain $\Gamma, \Gamma' \vdash$ case m_0' of $y.q$,$y.q' : T'$.

Case. Suppose $t' \equiv t_1'[T'']$. Similar to the previous case.

Case. Suppose $t' \equiv inl(t)$. Trivial.

Case. Suppose $t' \equiv inr(t)$. Trivial.

Case. Suppose $t' \equiv$ case m_0 of $y.m_1$,$y.m_2$. By inversion on the assumption $\Gamma, x : T, \Gamma' \vdash t' : T'$ we know the following:

$\Gamma, x : T, \Gamma' \vdash m_0 : T_1 + T_2$, for some types T_1 and T_2,
$\Gamma, x : T, \Gamma', y : T_1 \vdash m_1 : T, and$
$\Gamma, x : T, \Gamma', y : T_2 \vdash m_2 : T.$

It is easy to see that $t' > m_i$ for all $i \in \{0, 1, 2\}$. Hence, by the induction hypothesis there exists terms m_0', m_1', and m_2' such that $[t/x]^T m_i = m_i'$ for all $i \in \{0, 1, 2\}$,

(i) $\Gamma, \Gamma \vdash m_0' : T_1 + T_2$,
(ii) $\Gamma, \Gamma', y : T_1 \vdash m_1' : T'$, and
(iii) $\Gamma, \Gamma', y : T_2 \vdash m_2' : T'$.

We have two cases to consider.

Case. Suppose m_0 and m_0' are inject-left terms, inject-right terms, or case constructs, or m_0 is an inject-left term, inject-right term, or a case-construct and m_0' is not, or m_0 and m_0' are neither inject-left terms, inject-right terms, or case constructs, or $ctype_T(x, m_0)$ is undefined. Then $[t/x]^T$(case m_0 of $y.m_1$,$y.m_2$) = case m_0' of $y.m_1'$,$y.m_2'$ and by applying the case-construct typing rule to i - iii above we obtain $\Gamma, \Gamma' \vdash$ case m_0' of $y.m_1'$,$y.m_2' : T'$.

Case. Suppose m_0' is an inject-left term, inject-right term, or case construct and $ctype_T(x, m_0) = T_1 + T_2$. Then

$[t/x]^T(\text{case } m_0 \text{ of } y.m_1, y.m_2) = rcase_T\ m_0'\ y\ m_1'\ m_2'$ and by

Lemma 12.2.3.10 we know $T_1 + T_2 \equiv T_1 + T_2$ and is a subexpression of T. It suffices to show that there exists some term q such that

$rcase_T\ m_0'\ y\ m_1'\ m_2' = q$ and $\Gamma, \Gamma' \vdash q : T'$. We obtain this result by the following proposition.

Proposition. For all $\Gamma \vdash q_0 : T$, $\Gamma, y : T_1 \vdash q_1 : T'$, and $\Gamma, y : T_2 \vdash q_2 : T'$ there exists a term $\hat{q}$ such that $rcase_T\ q_0\ y\ q_1\ q_2 = \hat{q}$ and $\Gamma \vdash \hat{q} : T'$. We prove this by induction on the the ordering (T, t', q_0) and case split on the structure of q_0.

Case. Suppose q_0 is not an inject-left term, inject-right term, or a case construct. Then

$rcase_T\ q_0\ y\ q_1\ q_2 = \text{case } q_0 \text{ of } y.q_1, y.q_2$ and by applying the case-construct typing rule using the assumptions $\Gamma \vdash q_0 : T$, $\Gamma, y : T_1 \vdash q_1 : T'$, and $\Gamma, y : T_2 \vdash q_2 : T'$ we obtain $\Gamma, \Gamma' \vdash \text{case } q_0 \text{ of } y.q_1, y.q_2 : T'$.

Case. Suppose $q_0 \equiv inl(q_0')$. Then $rcase_T\ q_0\ y\ q_1\ q_2 = [q_0'/y]^{T_1} q_1$ and by inversion on $\Gamma \vdash q_0 : T$ we know $\Gamma \vdash q_0' : T_1$. It suffices to show that there exists a term $\hat{q}$ such that $[q_0'/y]^{T_1} q_1 = \hat{q}$ and $\Gamma \vdash \hat{q} : T'$. Since $T >_\Gamma T'$ we can apply the the outer induction hypothesis to obtain that there exists such a term $\hat{q}$.

Case. Suppose $q_0 \equiv inl(q_0')$. Similar to the previous case.

Case. Suppose $q_0 \equiv \text{case } q_0' \text{ of } z.q_1', z.q_2'$. Then

$rcase_T\ q_0\ y\ q_1\ q_2 = \text{case } q_0' \text{ of } z.(rcase_T\ q_1'\ y\ q_1\ q_2), z.(rcase_T\ q_2'\ y\ q_1\ q_2)$.

We know by assumption that $\Gamma \vdash q_0 : T$, $\Gamma \vdash q_0 : T$, and $\Gamma, y : T_1 \vdash q_1 : T'$ so by inversion we know the following:

(i) $\Gamma \vdash q_0' : T_1' + T_2'$, for some types T_1' and T_2',
(ii) $\Gamma, z : T_1' \vdash q_1' : T$, and
(iii) $\Gamma, z : T_2' \vdash q_2' : T$.

Now $q_0 > q_1'$ and $q_0 > q_1'$ so we can apply the induction hypothesis twice to obtain terms $\hat{q}_1$ and $\hat{q}_2$ such that $rcase_T\ q_1'\ y\ q_1\ q_2 = \hat{q}_1$, $rcase_T\ q_1'\ y\ q_1\ q_2 = \hat{q}_1$, $\Gamma, z : T_1' \vdash \hat{q}_1 : T'$ and $\Gamma, z : T_2' \vdash \hat{q}_2 : T'$. So case q_0' of $z.(rcase_T\ q_1'\ y\ q_1\ q_2), z.(rcase_T\ q_2'\ y\ q_1\ q_2) =$ case q_0' of $z.\hat{q}_1, z.\hat{q}_2$. It suffices to show that case q_0' of $z.\hat{q}_1, z.\hat{q}_2 = \hat{q}$ and $\Gamma \vdash \hat{q} : T$ for some term $\hat{q}$. Now $q_0 > q_0'$ so we can apply the induction hypothesis to obtain our result, but before we can we must show that $\Gamma \vdash$ case q_0' of $z.\hat{q}_1, z.\hat{q}_2 : T'$. This is a direct consequence of applying the case-construct typing rule using i, $\Gamma, z : T_1' \vdash \hat{q}_1 : T'$ and $\Gamma, z : T_2' \vdash \hat{q}_2 : T'$. Therefore, by the induction hypothesis there exists a term $\hat{q}$ such that case q_0' of $z.\hat{q}_1, z.\hat{q}_2 = \hat{q}$ and $\Gamma \vdash \hat{q} : T$.

□

The next result we show is that the hereditary substitution function cannot create new redexes, but this requires we first show that redexes are either preserved or destroyed. Just as we have seen before this requires the following definition.

Definition 12.2.3.12.

The following function constructs the set of redexes within a term:

$rset(x) = \emptyset$

$rset(\lambda x : T.t) = rset(t)$

$rset(\Lambda X : *_l.t) = rset(t)$

$rset(t_1\ t_2) = rset(t_1, t_2)$
Where t_1 *is not a* λ*-abstraction.*

$rset(t_1\ t_2) = \{t_1\ t_2\} \cup rset(t_1', t_2)$
Where $t_1 \equiv \lambda x : T.t_1'$.

$rset(t''[T'']) = rset(t'')$
Where t'' *is not a type absraction.*

$rset(t''[T'']) = \{t''[T'']\} \cup rset(t''')$
Where $t'' \equiv \Lambda X : *_l.t'''$.

$rset(inl(t)) = rset(t)$

$rset(inr(t)) = rset(t)$
$rset(case\ t_0\ of\ x.t_1,x.t_2) = rset(t_0) \cup rset(t_1, t_2)$
Where t_1 *is not an inject-left term or an inject-right term.*

$rset(case\ t_0\ of\ x.t_1,x.t_2) = \{case\ t_0\ of\ x.t_1,x.t_2\} \cup rset(t_0) \cup rset(t_1, t_2)$
Where t_1 *is an inject-left term or an inject-right term.*

The extention of $rset$ *to multiple arguments is defined as follows:*

$$rset(t_1, \ldots, t_n) =^{def} rset(t_1) \cup \cdots \cup rset(t_n).$$

Lemma 12.2.3.13. *[Redex Preserving] If* $\Gamma \vdash t : T$, $\Gamma, x : T, \Gamma' \vdash t' : T'$, *and* t' *then* $|rset(t', t)| \geq |rset([t/x]^T t')|$.

Proof. This is a proof by induction on the lexicorgraphic combination (T, t') of $>_{\Gamma,\Gamma'}$ and the strict subexpression ordering. We case split on the structure of t', and we only show the cases that differ from the proof of the same lemma for SSF

(Lemma 12.1.4.22).

Case. Let $t' \equiv inl(t'')$. We know $rset(t', t) = rset(t'', t)$. Since $t' > t''$ we can apply the induction hypothesis to obtain $|rset(t'', t)| \geq |rset([t/x]^T t'')|$. This implies $|rset(t', t)| \geq |rset([t/x]^T t')|$.

Case. Let $t' \equiv inr(t'')$. Similar to the previous case.

Case. Let $t' \equiv t'_1\ t'_2$. First consider when t'_1 is not a λ-abstraction or a case construct. Then

$$rset(t'_1\ t'_2, t) = rset(t'_1, t'_2, t)$$

Clearly, $t' > t'_i$ for $i \in \{1, 2\}$, hence, by the induction hypothesis $|rset(t'_i, t)| \geq |rset([t/x]^T t'_i)|$. We have three cases to consider. That is whether or not $[t/x]^T t'_1$ is a λ-abstraction, a case construct, or neither, or $ctype_T(x, t'_1)$ is undefined. However, we only show the new cases. Suppose $[t/x]^T t'_1 =$ case t''_0 of $y.t''_1$,$y.t''_2$. Then

$$\begin{array}{rcl} |rset([t/x]^T(t'_1\ t'_2))| & = & |rset(\text{case } t''_0 \text{ of } y.(app_T\ t''_1\ [t/x]^T t'_2), y.(app_T\ t''_2\ [t/x]^T t'_2))| \\ & = & |rset(t''_0, (app_T\ t''_1\ [t/x]^T t'_2), (app_T\ t''_2\ [t/x]^T t'_2))|. \end{array}$$

We know $t' > t'_1$ and $t' > t'_2$ so by the induction hypothesis

$$\begin{array}{rcl} |rset([t/x]^T t'_1)| & = & |rset(t''_0, t''_1, t''_2)| \\ & \leq & |rset(t'_1, t)| \end{array}$$

and

$$|rset([t/x]^T t'_2)| \leq |rset(t'_2, t)|.$$

By inversion on $\Gamma, x : T, \Gamma' \vdash t_1\ t_2 : T'$ there exists a type T'' such that $\Gamma, x : T, \Gamma' \vdash t'_1 : T'' \to T'$. So by Lemma 12.2.3.10, $ctype_T(x, t'_1) = T$,

$T \equiv T'' \to T'$, and T is a subexpression of T. Hence, $T >_{\Gamma,\Gamma'} T''$ and $T >_{\Gamma,\Gamma'} T'$. At this point we must show the following proposition.

Proposition. For all $\Gamma \vdash t_1 : T'' \to T'$ and $\Gamma \vdash t_2 : T''$ we have $|rset(app_T\ t_1\ t_2)| \leq |rset(t_1, t_2)|$.

We prove this by nested induction on the ordering (T, t', t_1) and case split on the structure of t_1.

Case. Suppose t_1 is not a λ-abstraction or a case construct. Then $app_T\ t_1\ t_2 = t_1\ t_2$ and $|rset(t_1\ t_2)| = |rset(t_1, t_2)|$. Thus, $|rset(app_T\ t_1\ t_2)| \leq |rset(t_1, t_2)|$.

Case. Suppose $t_1 \equiv \lambda y : T''.t_1''$. Then $app_T\ t_1\ t_2 = [t_2/y]^{T''}t_1''$. By inversion on the assumption $\Gamma \vdash t_1 : T'' \to T'$ we know $\Gamma, y : T'' \vdash t_1'' : T'$. We know $T >_{\Gamma,\Gamma'} T''$ so by the outter induction hypothesis

$$\begin{array}{rcl} |rset([t_2/y]^{T''}t_1'')| & \leq & |rset(t_1'', t_2) \\ & = & |rset(t_1, t_2). \end{array}$$

Thus, $|rset(app_T\ t_1\ t_2) \leq |rset(t_1, t_2)|$.

Case. Suppose $t_1 \equiv$ case t_0'' of $y.t_1''$,$y.t_2''$. Then

$$app_T\ t_1\ t_2 = \text{case } t_0'' \text{ of } y.app_T\ t_1''\ t_2, y.app_T\ t_2''\ t_2.$$

By inversion on the assumption $\Gamma \vdash t_1 : T'' \to T'$ we know $\Gamma, y : T_1'' \vdash t_1'' : T'' \to T'$ and $\Gamma, y : T_2'' \vdash t_2'' : T'' \to T'$. Since $t_1 > t_1''$ and $t_1 > t_2''$ we can apply the induction hypothesis to obtain

$$|rset(app_T\ t_1''\ t_2)| \leq |rset(t_1'', t_2)|$$

and

$$|rset(app_T\ t_2''\ t_2)| \leq |rset(t_2'', t_2)|.$$

Suppose t_0'' is not an inject-left or an inject-right term. Then

$$\begin{aligned} |rset(t_1, t_2)| &= |rset(\text{case } t_0'' \text{ of } y.t_1'', y.t_2'', t_2)| \\ &= |rset(t_0'', t_1'', t_2'', t_2)| \end{aligned}$$

and

$$\begin{aligned} |rset(app_T\ t_1\ t_2)| &= |rset(\text{case } t_0'' \text{ of } y.app_T\ t_1''\ t_2, y.app_T\ t_2''\ t_2)| \\ &= |rset(t_0'') \cup rset(app_T\ t_1''\ t_2) \cup rset(app_T\ t_2''\ t_2)| \\ &\leq |rset(t_0'') \cup rset(t_1'', t_2) \cup rset(t_2'', t_2)| \\ &= |rset(t_0'') \cup rset(t_1'', t_2'', t_2)| \\ &= |rset(t_0'', t_1'', t_2'', t_2)|. \end{aligned}$$

Therefore, $|rset(app_T\ t_1\ t_2)| \leq |rset(t_1, t_2)|$.

Now suppose $t_0'' \equiv inl(\hat{t}_0)$. We only show the case when t_0'' is an inject-left term, because the case when it is an inject-right term is similar. By definition we know

$$\begin{aligned} |rset(t_1, t_2)| &= |rset(\text{case } t_0'' \text{ of } y.t_1'', y.t_2'', t_2)| \\ &= |\{\text{case } t_0'' \text{ of } y.t_1'', y.t_2''\} \cup rset(t_0'', t_1'', t_2'', t_2)| \end{aligned}$$

and

$$\begin{aligned} &|rset(app_T\ t_1\ t_2)| \\ &= |rset(\text{case } t_0'' \text{ of } y.app_T\ t_1''\ t_2, y.app_T\ t_2''\ t_2)| \\ &\leq |\{\text{case } t_0'' \text{ of } y.app_T\ t_1''\ t_2, y.app_T\ t_2''\ t_2\} \cup rset(t_0'') \cup rset(t_1'', t_2'', t_2)| \\ &= |\{\text{case } t_0'' \text{ of } y.app_T\ t_1''\ t_2, y.app_T\ t_2''\ t_2\} \cup rset(t_0'', t_1'', t_2'', t_2)|. \end{aligned}$$

Therefore, $|rset(app_T\ t_1\ t_2)| \leq |rset(t_1, t_2)|$.

Finally, suppose $t_1' \equiv \text{case } t_0'' \text{ of } y.t_1'', y.t_2''$. Then

$$\begin{aligned} &|rset([t/x]^T(t_1'\ t_2'))| \\ &= |rset((\text{case } [t/x]^T t_0'' \text{ of } y.[t/x]^T t_1'', y.[t/x]^T t_2'')\ [t/x]^T t_2')| \\ &= |\{[t/x]^T(t_1'\ t_2')\} \cup rset([t/x]^T t_0'', [t/x]^T t_1'', [t/x]^T t_2'', [t/x]^T t_2')|. \end{aligned}$$

Now $t' > t_0''$, $t' > t_1''$, $t' > t_2''$, and $t' > t_2'$ so by the induction hypothesis

$$\begin{aligned} &|rset([t/x]^T t_0'')| \leq |rset(t_0'', t)|, \\ &|rset([t/x]^T t_1'')| \leq |rset(t_1'', t)|, \\ &|rset([t/x]^T t_2'')| \leq |rset(t_2'', t)|, \text{ and} \\ &|rset([t/x]^T t_2')| \leq |rset(t_2', t)|. \end{aligned}$$

Hence,

$$|rset([t/x]^T t_0'', [t/x]^T t_1'', [t/x]^T t_2'', [t/x]^T t_2')| \leq |rset(t_0'', t_1'', t_2'', t_2', t)|.$$

Now

$$\begin{aligned} |rset(t'_1\ t'_2, t)| &= |rset((\text{case } t''_0 \text{ of } y.t''_1, y.t''_2)\ t'_2, t)| \\ &= |\{t'_1\ t'_2\} \cup rset(t''_0, t''_1, t''_2, t'_2, t)|. \end{aligned}$$

Therefore, $|rset([t/x]^T(t'_1\ t'_2))| \leq |rset(t'_1\ t'_2, t)|$.

Case. Let $t' \equiv$ case t'_0 of $y.t'_1$,$y.t'_2$. Suppose t'_0 is not an inject-left term, and inject-right term, or a case-construct. First we know

$$|rset(t', t)| = |rset(t'_0, t'_1, t'_2, t)|.$$

Now we have several cases to consider, when $[t/x]^T t'_0$ is an inject-left term, an inject-right term, a case construct, something else entirely, or $ctype_T(x, t'_0)$ is undefined. Suppose it is something else entirely or $ctype_T(x, t'_0)$ is undefined. Then

$$\begin{aligned} |rset([t/x]^T t')| &= |rset(\text{case } [t/x]^T t'_0 \text{ of } y.([t/x]^T t'_1), y.([t/x]^T t'_2))| \\ &= |rset([t/x]^T t'_0, ([t/x]^T t'_1), ([t/x]^T t'_2))|. \end{aligned}$$

We can see that $t' > t'_0$, $t' > t'_1$, $t' > t'_2$ so by the induction hyothesis

$$\begin{aligned} &|rset([t/x]^T t'_0| \leq |rset(t'_0, t)|, \\ &|rset([t/x]^T t'_1| \leq |rset(t'_1, t)|, \text{ and} \\ &|rset([t/x]^T t'_2| \leq |rset(t'_2, t)|. \end{aligned}$$

This implies that $|rset([t/x]^T t'_0, ([t/x]^T t'_1), ([t/x]^T t'_2))| \leq |rset(t'_0, t'_1, t'_2, t)|$.

Therefore, $|rset(t', t)| \geq |rset([t/x]^T t')$.

Now suppose $[t/x]^T t'_0 \equiv inl(t''_0)$. We only show the case for when $[t/x]^T t'_0$ is an inject-left term, because the case for when it is an inject-right term is similar. We can see that

$$\begin{aligned} |rset(t', t)| &= |rset(\text{case } t'_0 \text{ of } y.t'_1, y.t'_2, t)| \\ &= |rset(t'_0, t'_1, t'_2, t)| \end{aligned}$$

and $|rset([t/x]^T t')| = |rset([t_0''/y]^{T_1}([t/x]^T t_1'))|$. By inversion on $\Gamma, x : T, \Gamma' \vdash t' : T'$ we know there exists types T_1 and T_2 such that $\Gamma, x : T, \Gamma' \vdash t_0 : T_1 + T_2$. So by Lemma 12.2.3.10 there exists a type T such that $ctype_T(x, t_0') = T$, $T \equiv T_1 + T_2$, and T is a subexpression of T. Thus, $T >_{\Gamma,\Gamma'} T_1$, $T >_{\Gamma,\Gamma'} T_2$, $t > t_0'$, and $t > t_1'$ so we can apply the induction hypothesis to obtain

$$\begin{array}{rcl} |rset([t_0''/y]^{T_1}([t/x]^T t_1'))| & \leq & |rset(t_0'', [t/x]^T t_1')| \\ & = & |rset([t/x]^T t_0', [t/x]^T t_1')| \\ & \leq & |rset(t_0', t_1', t)| \\ & \leq & |rset(t_0', t_1', t_2', t)|. \end{array}$$

Next suppose $[t/x]^T t_0' \equiv$ case t_0'' of $y.t_1''$,$y.t_2''$. Then

$$|rset([t/x]^T t')| = |rset(rcase_T\ [t/x]^T t_0'\ y\ [t/x]^T t_1'\ [t/x]^T t_2')|$$

and

$$\begin{array}{rcl} |rset(t', t)| & = & |rset(\text{case } t_0' \text{ of } y.t_1'\text{,}y.t_2', t)| \\ & = & |rset(t_0', t_1, t_2', t)|. \end{array}$$

Note that by inverision on $\Gamma, x : T, \Gamma' \vdash t' : T'$ we know there exists types T_1 and T_2 such that $\Gamma, x : T, \Gamma' \vdash t_0' : T_1 + T_2$. So by Lemma 12.2.3.10 there exists a type T such that $ctype_T(x, t_0') = T$, $T \equiv T_1 + T_2$, and T is a subexpression of T. Thus, $T >_{\Gamma,\Gamma'} T_1$ and $T >_{\Gamma,\Gamma'} T_2$. It suffices to show that $|rset(rcase_T\ [t/x]^T t_0'\ [t/x]^T t_1'\ [t/x]^T t_2')| \leq |rset([t/x]^T t_0', [t/x]^T t_1', [t/x]^T t_2')|$ which is a consequence of the following proposition.

Proposition. For all $\Gamma \vdash t : T_1 + T_2$, $\Gamma, y : T_1 \vdash t_1' : T'$, and $\Gamma, y : T_2 \vdash t_2' : T'$ we have $|rset(rcase_T\ t\ y\ t_1'\ t_2')| \leq |rset(t, t_1', t_2')|$.

We prove this proposition by nested induction on (T, t', t) and we case split on t.

Case. Suppose $t \equiv inl(t')$. Then

$$rset(rcase_T\ t\ y\ t'_1\ t'_2) = rset([t'/y]^{T_1}t'_1).$$

By inversion on $\Gamma \vdash t : T_1 + T_2$ we know $\Gamma \vdash t' : T_1$. So by the outer induction hypothesis

$$\begin{array}{rcl} |rset([t'/y]^{T_1}t'_1)| & \leq & |rset(t'_1, t')| \\ & = & |rset(t, t'_1)| \\ & \leq & |rset(t, t'_1, t'_2)|. \end{array}$$

Therefore, $|rset(rcase_T\ t\ y\ t'_1\ t'_2)| \leq |rset(t, t'_1, t'_2)|$.

Case. Suppose $t \equiv inr(t')$. This case is similar to the previous case.

Case. Suppose $t \equiv$ case t_0 of $z.t_1$,$z.t_2$. Then

$$\text{rcase}_T\ t\ y\ t'_1\ t'_2 = \text{case } t_0 \text{ of } z.(\text{rcase}_T\ t_1\ y\ t'_1\ t'_2),z.(\text{rcase}_T\ t_2\ y\ t'_1\ t'_2).$$

Now $t > t'_i$ for $i \in \{0, 1, 2\}$. Before we can apply the inductive hypothesis we must show that t_1 and t_2 are typeable. By inversion on the assumption $\Gamma \vdash t : T_1 + T_2$ we know $\Gamma, z : T'_1 \vdash t_1 : T_1 + T_2$ and $\Gamma, z : T'_2 \vdash t_2 : T_1 + T_2$. So by the inner induction hypothesis $|rset(\text{rcase}_T\ t_i\ y\ t'_1\ t'_2)| \leq |rset(t_i, t'_1, t'_2)|$.

We have two cases to consider either t_0 is not an inject-left term or an inject-right term, or it is. If not then

$$rset(\text{rcase}_T\ t\ y\ t'_1\ t'_2) = rset(t_0, \text{rcase}_T\ t_1\ y\ t'_2\ t'_2, \text{rcase}_T\ t_2\ y\ t'_2\ t'_2)$$

otherwise

$$\begin{array}{l} rset(\text{rcase}_T\ t\ y\ t'_1\ t'_2) \\ \quad = \{\text{rcase}_T\ t\ y\ t'_1\ t'_2\} \cup rset(t_0, \text{rcase}_T\ t_1\ y\ t'_1\ t'_2, \text{rcase}_T\ t_2\ y\ t'_1\ t'_2). \end{array}$$

Suppose t_0 is not an inject-left or an inject-right term. Then

$$|rset(t, t'_1, t'_2)| = |rset(t_0, t_1, t_2, t'_1, t'_2)|.$$

Now we know

$$\begin{aligned}
&|rset(t_0, \text{rcase}_T\ t_1\ y\ t_2'\ t_2', \text{rcase}_T\ t_2\ y\ t_2'\ t_2')| \\
&\quad = |rset(t_0) \cup rset(\text{rcase}_T\ t_1\ y\ t_2'\ t_2') \cup rset(\text{rcase}_T\ t_2\ y\ t_2'\ t_2')| \\
&\quad \leq |rset(t_0) \cup rset(t_1, t_2', t_2') \cup rset(t_1, t_2', t_2')| \\
&\quad = |rset(t_0, t_1, t_2, t_2', t_2')|.
\end{aligned}$$

Therefore, $|rset(\text{rcase}_T\ t\ y\ t_1'\ t_2')| \leq |rset(t, t_1', t_2')|$.

Now suppose $t_0 \equiv inl(t_0')$. Then

$$|rset(t, t_1', t_2')| = |\{t\} \cup rset(t_0, t_1, t_2, t_1', t_2')|.$$

It suffices to show that

$$\begin{aligned}
&|\{\text{rcase}_T\ t\ y\ t_1'\ t_2'\} \cup rset(t_0, \text{rcase}_T\ t_1\ y\ t_1'\ t_2', \text{rcase}_T\ t_2\ y\ t_1'\ t_2')| \\
&\quad \leq |\{t\} \cup rset(t_0, t_1, t_2, t_1', t_2')|.
\end{aligned}$$

Let $A = \text{rcase}_T\ t_1\ y\ t_1'\ t_2'$ and $B = \text{rcase}_T\ t_2\ y\ t_1'\ t_2'$. Since

$|rset(\text{rcase}_T\ t_i\ y\ t_1'\ t_2')| \leq |rset(t_i, t_1', t_2')|$ we obtain the following:

$$\begin{aligned}
&|\{\text{rcase}_T\ t\ y\ t_1'\ t_2'\} \cup rset(t_0, A, B)| \\
&\quad = |\{\text{rcase}_T\ t\ y\ t_1'\ t_2'\}| + |rset(t_0)| + |rset(A)| + |rset(B)| \\
&\quad \leq |\{\text{rcase}_T\ t\ y\ t_1'\ t_2'\}| + |rset(t_0)| + |rset(t_1, t_1', t_2')| + |rset(t_2, t_1', t_2')| \\
&\quad = |\{\text{rcase}_T\ t\ y\ t_1'\ t_2'\}| + |rset(t_0, t_1, t_2, t_1', t_2')| \\
&\quad = |\{t\} \cup rset(t_0, t_1, t_2, t_1', t_2')|.
\end{aligned}$$

The case when t_0 is an inject-right term is similar to the case when it is an inject-left term.

Now by the previous proposition we know

$$|rset(rcase_T\ [t/x]^T t_0'\ [t/x]^T t_1'\ [t/x]^T t_2')| \leq |rset([t/x]^T t_0', [t/x]^T t_1', [t/x]^T t_2')|,$$

becuase by Lemma 12.2.3.11 t_0', t_1', and t_2' have the same types as $[t/x]^T t_0'$, $[t/x]^T t_1'$, and $[t/x]^T t_2'$. Now $t' > t_0'$, $t' > t_1'$, and $t' > t_2'$, so

$$\begin{aligned}
&|rset([t/x]^T t_0')| \leq |rset(t_0', t), \\
&|rset([t/x]^T t_1')| \leq |rset(t_1', t),\ \text{and} \\
&|rset([t/x]^T t_2')| \leq |rset(t_2', t).
\end{aligned}$$

Thus,

$$\begin{array}{rcl} |rset([t/x]^T t'_0, [t/x]^T t'_1, [t/x]^T t'_2)| & \leq & |rset(t'_0, t'_1, t'_2, t)| \\ & = & |rset(t', t)|. \end{array}$$

Suppose $t'_0 \equiv inl(t''_0)$. Again, we only show the case for when t'_0 is an inject-left term. We know

$$\begin{array}{rcl} |rset([t/x]^T t')| & = & |rset(\text{case } [t/x]^T t'_0 \text{ of } y.[t/x]^T t'_1, y.[t/x]^T t'_2)| \\ & = & |rset(\text{case } inl([t/x]^T t''_0) \text{ of } y.[t/x]^T t'_1, y.[t/x]^T t'_2)| \\ & = & |\{[t/x]^T t'\} \cup rset([t/x]^T t''_0, [t/x]^T t'_1, [t/x]^T t'_2)| \end{array}$$

and

$$\begin{array}{rcl} |rset(t', t)| & = & |rset(\text{case } t'_0 \text{ of } y.t'_1, y.t'_2, t)| \\ & = & |rset(\text{case } inl(t''_0) \text{ of } y.t'_1, y.t'_2, t)| \\ & = & |\{t'\} \cup rset(t''_0, t'_1, t'_2)|. \end{array}$$

Now $t' > t''_0$, $t' > t'_1$, and $t' > t'_2$ so by the induction hypothesis

$$\begin{array}{rcl} |rset([t/x]^T t''_0)| & \leq & |rset(t''_0, t)| \\ |rset([t/x]^T t'_1)| & \leq & |rset(t'_1, t)| \\ |rset([t/x]^T t'_2)| & \leq & |rset(t'_2, t)|. \end{array}$$

Therefore, $|rset([t/x]^T t''_0, [t/x]^T t'_1, [t/x]^T t'_2)| \leq |rset(t''_0, t'_1, t'_2, t)|$ which implies that $|rset([t/x]^T t')| \leq |rset(t', t)|$.

Finally, suppose $t'_0 \equiv \text{case } t''_0 \text{ of } z.t''_1, z.t''_2$. Then

$$\begin{array}{l} |rset([t/x]^T t')| \\ = |rset(\text{case } [t/x]^T t'_0 \text{ of } y.[t/x]^T t'_1, y.[t/x]^T t'_2)| \\ = |rset(\text{case case } [t/x]^T t''_0 \text{ of } z.[t/x]^T t''_0 t''_1, z.[t/x]^T t''_0 t''_2 \text{ of } y.[t/x]^T t'_1, y.[t/x]^T t'_2)| \\ = |\{[t/x]^T t'\} \cup rset([t/x]^T t'_0, [t/x]^T t'_1, [t/x]^T t'_2)| \end{array}$$

and

$$\begin{array}{rcl} |rset(t', t)| & = & |rset(\text{case } t'_0 \text{ of } y.t'_1, y.t'_2, t)| \\ & = & |rset(\text{case case } t''_0 \text{ of } z.t''_1, z.t''_2 \text{ of } y.t'_1, y.t'_2, t)| \\ & = & |\{t'\} \cup rset(t'_0, t'_1, t'_2)|. \end{array}$$

Now $t' > t''_0$, $t' > t'_1$, and $t' > t'_2$ so by the induction hypothesis

$$\begin{array}{rcl} |rset([t/x]^T t'_0)| & \leq & |rset(t'_0, t)|, \\ |rset([t/x]^T t'_1)| & \leq & |rset(t'_1, t)|, \text{ and} \\ |rset([t/x]^T t'_2)| & \leq & |rset(t'_2, t)|. \end{array}$$

Therefore, $|rset([t/x]^T t''_0, [t/x]^T t'_1, [t/x]^T t'_2)| \leq |rset(t''_0, t'_1, t'_2, t)|$ which implies that $|rset([t/x]^T t')| \leq |rset(t', t)|$.

□

We now use the previous result to show the hereditary substitution function to be normality preserving.

Lemma 12.2.3.14. *[Normality Preserving] If* $\Gamma \vdash n : T$ *and* $\Gamma, x : T' \vdash n' : T'$ *then there exists a normal term* n'' *such that* $[n/x]^T n' = n''$.

Proof. By Lemma 12.2.3.11 we know there exists a term n'' such that $[n/x]^T n' = t$ and by Lemma 12.2.3.13 $|rset(n', n)| \geq |rset([n/x]^T n')|$. Hence, $|rset(n', n)| \geq |rset(t)|$, but $|rset(n', n)| = 0$. Therefore, $|rset(t)| = 0$ which implies n'' has no redexes. It suffices to show that n'' has no structural redexes. We prove this by induction on the lexicographic ordering (T, n'). We case split on the structure of n'.

Case. Suppose n' is a variable x or y distinct from x. Trivial in both cases.

Case. Suppose $n' \equiv \lambda y : T''.\hat{n'}$. Then $[n/x]^T n' = \lambda y : T''.[t/x]^T \hat{n'}$. By inversion on the assumption $\Gamma, x : T' \vdash n' : T'$ we know $\Gamma, x : T', \Gamma', y : T'' \vdash \hat{n'} : T'$. Since $n' > \hat{n}$ we can apply the induction hypothesis to obtain there exists a term t' such that $[t/x]^T \hat{n} = t'$ and t' has no structural redexes. Therefore, neither does $\lambda y : T''.[t/x]^T \hat{n'}$.

Case. Suppose $n' \equiv \Lambda X : *_l.\hat{n}$. Similar to the previous case.

Case. Suppose $n' \equiv inl(n'_0)$. Similar to the λ-abstraction case.

Case. Suppose $n' \equiv n'_1\ n'_2$. By inversion we know $\Gamma, x : T, \Gamma' \vdash n'_1 : T'' \to T'$ and $\Gamma, x : T, \Gamma' \vdash n'_2 : T''$ for some types T' and T''. Clearly, $n' > n'_i$ for $i \in \{1, 2\}$. Thus, by the induction hypothesis there exists normal terms m_1 and m_2 such

that $[n/x]^T n'_i = m_i$ such that m_i have no structural redexes. We case split on whether or not m_1 is a λ-abstraction or a case construct and n'_1 is not, or $ctype_T(x, n'_1)$ is undefined. We only consider the non-trivial cases when $m_1 \equiv \lambda y : T''.m'_1$ or $m_1 \equiv$ case m'_0 of $y.m'_1$,$y.m'_2$ and n'_1 is not a λ-abstraction or a case construct. Suppose the former. Now by Lemma 12.2.3.10 there exists a type T such that $ctype_T(x, n'_1) = T$, $T \equiv T'' \to T'$, and T is a subexpression of T, hence $T >_{\Gamma,\Gamma'} T''$. So $[n/x]^T(n'_1\ n'_2) = [m_2/y]^{T''} m'_1$ and by the induction hypothesis there exists a term m such that $[m_2/y]^{T''} m'_1 = m$ and m has no structural redexes..

Suppose $m_1 \equiv$ case m'_0 of $y.m'_1$,$y.m'_2$. By inversion on $\Gamma, \Gamma' \vdash m_1 : T'' \to T'$ we know there exists terms T_1 and T_2 such that $\Gamma, \Gamma' \vdash m'_0 : T_1 + T_2$ and $\Gamma, \Gamma', y : T_i \vdash m'_i : T'' \to T'$ for $i \in \{1, 2\}$. Note that by Lemma 12.2.3.10 there exists a type T such that $ctype_T(x, n'_1) = T$, $T \equiv T'' \to T'$, and T is a subexpression of T, hence $T >_{\Gamma,\Gamma'} T'$ and $T >_{\Gamma,\Gamma'} T''$. Now $[t/x]^T t' =$ case m'_0 of $y.app_T\ m'_1\ m_2$,$y.app_T\ m'_2\ m_2$. It suffcies to show that there exists terms q and q' such that $app_T\ m'_1\ m_2 = q$, $app_T\ m'_2\ m_2 = q'$ and q and q' have no structural redexes. To obtain this result we prove the following proposition.

Proposition. For all normal terms m_2 and m'_1 such that $\Gamma \vdash m_2 : T''$ and $\Gamma \vdash m'_1 : T'' \to T'$ there exists a term q such that $app_T\ m'_1\ m_2 = q$ and q has no structural redexes.

We prove this by nested induction on the ordering (T, n', m'_1) and case split-

ting on the structure of m_1'.

Case. Suppose m_1' is neither a λ-abstraction or a case construct. Then $app_T\ m_1'\ m_2 = m_1'\ m_2$. Take $m_1'\ m_2$ for q and we know q has no structural redexes, because m_1' and m_2 are normal.

Case. Suppose $m_1' \equiv \lambda z : T''.m_1''$. Then $app_T\ m_1'\ m_2 = [m_2/z]^{T''}m_1''$. By inversion on the assumption $\Gamma \vdash m_1' : T'' \to T'$ we know $\Gamma, z : T'' \vdash m_1'' : T'$. Since $T >_\Gamma T''$ we can apply the outter induction hypothesis to obtain there there exists a q such that $[m_2/z]^{T''}m_1'' = q$ and q has no structural redexes.

Case. Suppose $m_1' \equiv$ case m_0'' of $z.m_1''$,$z.m_2''$. Then $app_T\ m_1'\ m_2 =$ case m_0'' of $z.app_T\ m_1''\ m_2$,$z.app_T\ m_2''\ m_2$. By inversion on the assumption $\Gamma \vdash m_1' : T'' \to T'$ we know there exists types T_1 and T_2 such that $\Gamma \vdash m_0'' : T_1 + T_2$, $\Gamma, z : T_1 \vdash m_1'' : T'' \to T'$ and $\Gamma, z : T_2 \vdash m_2'' : T'' \to T'$. Since $m_1' > m_1''$ and $m_1' > m_2''$ we can apply the inner induction hypothesis to obtain there exists terms q' and q'' such that $app_T\ m_1''\ m_2 = q'$, q' has no structural redexes, $app_T\ m_1''\ m_2 = q''$ and q'' has no structural redexes. Hence, $app_T\ m_1'\ m_2 =$ case m_0'' of $z.app_T\ m_1''\ m_2$,$z.app_T\ m_2''\ m_2 =$ case m_0'' of $z.q'$,$z.q''$ and case m_0'' of $z.q'$,$z.q''$ has no structural redexes. Note that m_0'' is normal, because m_1' is normal.

By the previous proposition there exists terms q and q' such that $[n/x]^T n' =$ case m_0' of $y.app_T\ m_1'\ m_2$,$y.app_T\ m_2'\ m_2 =$ case m_0' of $y.q$,$y.q'$, where $app_T\ m_1'\ m_2 = q$, $app_T\ m_1'\ m_2 = q'$, and q and q' have no structural

redexes. Thus, case m_0' of $y.q$,$y.q'$ has no structural redexes.

Case. Suppose $n' \equiv$ case m_0 of $y.m_1$,$y.m_2$. By inversion on the assumption $\Gamma, x : T, \Gamma' \vdash n' : T'$ we know the following:

$\Gamma, x : T, \Gamma' \vdash m_0 : T_1 + T_2$, for some types T_1 and T_2,
$\Gamma, x : T, \Gamma', y : T_1 \vdash m_1 : T, and$
$\Gamma, x : T, \Gamma', y : T_2 \vdash m_2 : T.$

It is easy to see that $n' > m_i$ for all $i \in \{0, 1, 2\}$. Hence, by the induction hypothesis there exists terms m_0', m_1', and m_2' such that $[t/x]^T m_i = m_i'$ and m_i' have no structural redexes for all $i \in \{0, 1, 2\}$. We have two cases to consider.

Case. Suppose m_0' is not an inject-left term, inject-right term, or case construct, or m_0 is an inject-left term, an inject-right term, or a case construct, or $ctype_T(x, m_0)$ is undefined. Then

$[n/x]^T$(case m_0 of $y.m_1$,$y.m_2$) = case m_0' of $y.m_1'$,$y.m_2'$ which has no structural redexes.

Case. Suppose m_0' is an inject-left term, inject-right term, or case construct and m_0 is not an inject-left term, an inject-right term, or a case construct. Then $[n/x]^T$(case m_0 of $y.m_1$,$y.m_2$) $= rcase_T\ m_0'\ y\ m_1'\ m_2'$, where by Lemma 12.2.3.10 there exists a type T such that $ctype_T(x, m_0) = T$, $T \equiv T_1 + T_2$, and T is a subexpression of T, hence $T >_{\Gamma,\Gamma'} T_1$ and $T >_{\Gamma,\Gamma'} T_2$. Consider the case when $m_0' \equiv inl(m_0'')$. Then we know by the definition of $rcase$ that $rcase_T\ m_0'\ y\ m_1'\ m_2' = [m_0''/y]^{T_1} m_1'$. Clearly, $T >_{\Gamma,\Gamma'} T_1$ hence by the the induction hypothesis there exists a term r such that

$[m_0''/y]^{T_1} m_1' = r$ and r has no structural redexes. Similarly for when $m_0' \equiv inr(m_0'')$. So suppose $m_0' \equiv$ case m_0'' of $z.m_1''$,$z.m_2''$ then it suffices to show that there exists some term q such that $rcase_T\ m_0'\ y\ m_1'\ m_2' = q$ and q has no structural redexes. We obtain this result by the following proposition.

Proposition. For all normal terms q and q_1 such that $\Gamma \vdash q_0 : T$, $\Gamma, y : T_1 \vdash q_1 : T'$, and $\Gamma, y : T_2 \vdash q_2 : T'$ there exists a term $\hat{q}$ such that $rcase_T\ q_0\ y\ q_1\ q_2 = \hat{q}$ and $\hat{q}$ has no structural redexes. We prove this by induction on the the ordering (T, n', q_0) and case split on the structure of q_0.

Case. Suppose q_0 is not an inject-left term, inject-right term, or a case construct. Then

$rcase_T\ q_0\ y\ q_1\ q_2 =$ case q_0 of $y.q_1$,$y.q_2$ which has no structural redexes.

Case. Suppose $q_0 \equiv inl(q_0')$. Then $rcase_T\ q_0\ y\ q_1\ q_2 = [q_0'/y]^{T_1} q_1$ and by inversion on $\Gamma \vdash q_0 : T$ we know $\Gamma \vdash q_0' : T_1$. It suffices to show that there exists a term $\hat{q}$ such that $[q_0'/y]^{T_1} q_1 = \hat{q}$ and $\hat{q}$ has no structural redexes. Clearly, $T >_\Gamma T'$ so by the outer induction hypothesis there exists such a term $\hat{q}$.

Case. Suppose $q_0 \equiv inl(q_0')$. Similar to the previous case.

Case. Suppose $q_0 \equiv$ case q_0' of $z.q_1'$,$z.q_2'$. Then

$rcase_T\ q_0\ y\ q_1\ q_2 =$ case q_0' of $z.(rcase_T\ q_1'\ y\ q_1\ q_2)$,$z.(rcase_T\ q_2'\ y\ q_1\ q_2)$.

We know by assumption that $\Gamma \vdash q_0 : T$, $\Gamma \vdash q_0 : T$, and $\Gamma, y : T_1 \vdash$

$q_1 : T'$ so by inversion we know the following:

(i) $\Gamma \vdash q_0' : T_1' + T_2'$, for some types T_1' and T_2',
(ii) $\Gamma, z : T_1' \vdash q_1' : T$, and
(iii) $\Gamma, z : T_2' \vdash q_2' : T$.

Now $q_0 > q_1'$ and $q_0 > q_1'$ so we can apply the inner induction hypothesis twice to obtain terms $\hat{q}_1$ and $\hat{q}_2$ such that $rcase_T\ q_1'\ y\ q_1\ q_2 = \hat{q}_1$, $rcase_T\ q_1'\ y\ q_1\ q_2 = \hat{q}_1$ where $\hat{q}_1$ and $\hat{q}_2$ have no structural redexes. So case q_0' of $z.(rcase_T\ q_1'\ y\ q_1\ q_2)$,$z.(rcase_T\ q_2'\ y\ q_1\ q_2)$ = case q_0' of $z.\hat{q}_1$,$z.\hat{q}_2$. It suffices to show that case q_0' of $z.\hat{q}_1$,$z.\hat{q}_2 = \hat{q}$ for some normal term $\hat{q}$. Now $q_0 > q_0'$ so we can apply the induction hypothesis to obtain our result, but before we can we must show that $\Gamma \vdash$ case q_0' of $z.\hat{q}_1$,$z.\hat{q}_2 : T'$. This is a direct consequence of applying the case-construct typing rule using i, $\Gamma, z : T_1' \vdash \hat{q}_1 : T'$ and $\Gamma, z : T_2' \vdash \hat{q}_2 : T'$. Therefore, by the inner induction hypothesis there exists a term $\hat{q}$ such that case q_0' of $z.\hat{q}_1$,$z.\hat{q}_2 = \hat{q}$ and $\hat{q}$ is has no structural redexes.

□

Finally, we show that the hereditary substitution function is consistent with respect to reduction.

Lemma 12.2.3.15. *[Soundness with Respect to Reduction] If* $\Gamma \vdash t : T$ *and* $\Gamma, x : T, \Gamma' \vdash t' : T'$ *then* $[t/x]t' \rightsquigarrow^* [t/x]^T t'$.

Proof. This is a proof by induction on the lexicorgraphic combination (T, t') of $>_\Gamma$ and the strict subexpression ordering. We case split on the structure of t'. When applying

the induction hypothesis we must show that the input terms to the substitution and the hereditary substitution functions are typeable. We do not explicitly state typing results that are simple conseqences of inversion. Furthermore, we only give the cases that differ from the proof of the same result for SSF (Lemma 12.1.4.24).

Case. Suppose $t' \equiv inl(t'_0)$. Then $[t/x]^T t' = inl([t/x]^T t'_0)$. We can see that $t' > t'_0$ so by the induction hypothesis $[t/x]t'_0 \rightsquigarrow^* [t/x]^T t'_0$. Hence, $inl([t/x]t'_0) \rightsquigarrow^* inl([t/x]^T t'_0)$ which implies that $[t/x](inl(t'_0)) \rightsquigarrow^* [t/x]^T(inl(t'_0))$.

Case. Suppose $t' \equiv inr(t'_0)$. Similar to the previous case.

Case. Suppose $t' \equiv$ case t'_0 of $y.t'_1$,$y.t'_2$. Clearly, $t' > t'_0$, $t' > t'_1$, and $t' > t'_2$, so we can apply the induction hypothesis to conclude $[t/x]t'_0 \rightsquigarrow^* [t/x]^T t'_0$, $[t/x]t'_1 \downarrow [t/x]^T t'_1$, and $[t/x]t'_2 \rightsquigarrow^* [t/x]^T t'_2$. We have several cases to consider, either when $[t/x]^T t'_0$ is an inject-left term or an inject-right term and t'_0 is not, when $[t/x]^T t'_0$ is a case construct and t'_0 is not, or $[t/x]^T t'_0$ is not an inject-left term, an inject-right term, or a case construct, or $ctype_T(x, t'_0)$ is undefined. The cases when $[t/x]^T t'_0$ is not an inject-left term, an inject-right term, or a case construct, or $ctype_T(x, t'_0)$ is undefined are trivial.

Let's consider the case when $[t/x]^T t'_0$ is an inject-left term or an inject-right term and t'_0 is not. Since the case when $[t/x]^T t'_0$ is an inject-left term is similar to the case when it is an inject-right term we only consider the former. Suppose $[t/x]^T t'_0 = inl(t''_0)$ and t'_0 is not an inject-left term. By Lemma 12.2.3.10 there exists a type T such that $ctype_T(x, t'_0) = T$, $T \equiv T_1 + T_2$, and T is a subexpression of T, where by inversion on $\Gamma, x :$

$T, \Gamma' \vdash t' : T'$ there exists types T_1 and T_2 such that $\Gamma, x : T, \Gamma' \vdash t'_0 : T_1 + T_2$. Thus, $T >_{\Gamma,\Gamma'} T_1$ and $T >_{\Gamma,\Gamma'} T_2$. So $[t/x]^T t' = [t''_0/y]^{T_1}([t/x]^T t'_1)$ and we know from above that $[t/x]t'_1 \rightsquigarrow^* [t/x]^T t'_1$. Now $T >_{\Gamma,\Gamma'} T_1$, so by the induction hypothesis, $[t''_0/y]([t/x]^T t'_1) \rightsquigarrow^* [t''_0/y]^{T_1}([t/x]^T t'_1)$. Thus, $[t''_0/y]([t/x]t'_1) \rightsquigarrow^* [t''_0/y]^{T_1}([t/x]^T t'_1)$. It suffices to show $[t/x]t' \rightsquigarrow^* [t''_0/y]([t/x]t'_1)$. We can see that

$$\begin{array}{rcl} [t/x]t' & = & [t/x](\text{case } x \text{ of } y.t'_1, y.t'_2) \\ & \equiv & \text{case } [t/x]x \text{ of } y.[t/x]t'_1, y.[t/x]t'_2 \\ & \equiv & \text{case } inl(t''_0) \text{ of } y.[t/x]t'_1, y.[t/x]t'_2 \\ & \rightsquigarrow & [t''_0/y]([t/x]t'_1). \end{array}$$

Suppose $[t/x]t'_0 = \text{case } t''_0 \text{ of } z.t''_1, z.t''_2$ and t'_0 is not. It suffices to show that $[t/x]t \rightsquigarrow^* [t/x]^T t'$, which is equivalent to showing $[t/x](\text{case } t'_0 \text{ of } y.t'_1, y.t'_2) \rightsquigarrow^* [t/x]^T(\text{case } t'_0 \text{ of } y.t'_1, y.t'_2)$. Now

$$\begin{array}{l} [t/x]^T(\text{case } t'_0 \text{ of } y.t'_1, y.t'_2) \\ \quad = \text{case } t''_0 \text{ of } z.(\text{rcase}_T\ t''_1\ y\ t'_1\ t'_2), z.(\text{rcase}_T\ t''_1\ y\ t'_1\ t'_2) \end{array}$$

and

$$\begin{array}{l} [t/x](\text{case } t'_0 \text{ of } y.t'_1, y.t'_2) \\ \quad = \text{case } [t/x]t'_0 \text{ of } y.[t/x]t'_1, y.[t/x]t'_2 \\ \quad \rightsquigarrow^* \text{case } (\text{case } t''_0 \text{ of } z.t''_1, z.t''_2) \text{ of } y.[t/x]t'_1, y.[t/x]t'_2 \\ \quad \rightsquigarrow \text{case } t''_0 \text{ of } z.(\text{case } t''_1 \text{ of } y.t'_1, y.t'_2), z.(\text{case } t''_2 \text{ of } y.t'_1, y.t'_2), \end{array}$$

because we know from above that $[t/x]t'_0 \rightsquigarrow^* [t/x]^T t'_0$. So it suffices to show that $(\text{case } t''_1 \text{ of } y.t'_1, y.t'_2) \rightsquigarrow^* (\text{rcase}_T\ t''_1\ y\ t'_1\ t'_2)$ and $(\text{case } t''_2 \text{ of } y.t'_1, y.t'_2) \rightsquigarrow^* (\text{rcase}_T\ t''_2\ y\ t'_1\ t'_2)$, because we know from above that $[t/x]t_i \rightsquigarrow^* [t/x]^T t'_i$. This is a consequence of the following proposition. First note that again by Lemma 12.2.3.10 there exists a type T such that $ctype_T(x, t'_0) = T$, $T \equiv T_1 + T_2$, and T is a subexpression of T, where by inversion on the assumption $\Gamma, x : T, \Gamma' \vdash t' : T'$ there exists types T_1 and T_2 such that $\Gamma, x : T, \Gamma' \vdash t'_0 :$

$T_1 + T_2$. Hence, $T >_{\Gamma,\Gamma'} T_1$ and $T >_{\Gamma,\Gamma'} T_2$.

Proposition. For all $\Gamma \vdash t_0 : T_1 + T_2$, $\Gamma, y : T_1 \vdash t_1 : T''$ and $\Gamma, y : T_2 \vdash t_2 : T''$ we have (case t_0 of $y.t_1$,$y.t_2$) $\rightsquigarrow^*$ (rcase$_T$ t_0 y t_1 t_2).

We prove this by nested induction on the ordering (T, t', t_0) and case splitting on the structure of t_0.

Case. Suppose t_0 is not an inject-left term, an inject-right term, or a case construct. Then

$$\text{rcase}_T\ t_0\ y\ t_1\ t_2 = \text{case } t_0 \text{ of } y.t_1,y.t_2.$$

Case. Suppose $t_0 \equiv inl(t_0')$. Then

$$\text{rcase}_T\ t_0\ y\ t_1\ t_2 = [t_0'/y]^{T_1} t_1$$

and

$$\begin{array}{rcl} \text{case } t_0 \text{ of } y.t_1,y.t_2 & \equiv & \text{case } inl(t_0') \text{ of } y.t_1,y.t_2 \\ & \rightsquigarrow & [t_0'/y]t_1. \end{array}$$

Now $T >_\Gamma T_1$ so by the outer-induction hypothesis $[t_0'/y]t_1 \rightsquigarrow^* [t_0'/y]^{T_1} t_1$.

Therefore, (case t_0 of $y.t_1$,$y.t_2$) $\rightsquigarrow^*$ (rcase$_T$ t_0 y t_1 t_2).

Case. Suppose $t_0 \equiv inl(t_0')$. Similar to the previous case.

Case. Suppose $t_0 \equiv$ case t_0' of $z.t_1'$,$z.t_2'$. Then

$$\text{rcase}_T\ t_0\ y\ t_1\ t_2 = \text{case } t_0' \text{ of } z.(\text{rcase}_T\ t_1'\ y\ t_1\ t_2),z.(\text{rcase}_T\ t_2'\ y\ t_1\ t_2)$$

and

$$\begin{array}{l} \text{case } t_0 \text{ of } y.t_1,y.t_2 \\ \quad \equiv \text{case (case } t_0' \text{ of } z.t_1',z.t_2') \text{ of } y.t_1,y.t_2 \\ \quad \rightsquigarrow \text{case } t_0' \text{ of } z.(\text{case } t_1' \text{ of } y.t_1,y.t_2),z.(\text{case } t_2' \text{ of } y.t_1,y.t_2). \end{array}$$

Trivially, $t_0 > t_1'$ and $t_0 > t_2'$ so by the inner-induction hypothesis

$$(\text{case } t_1' \text{ of } y.t_1,y.t_2) \rightsquigarrow^* (\text{rcase}_T\ t_1'\ y\ t_1\ t_2)$$

and

$$(\text{case } t_2' \text{ of } y.t_1,y.t_2) \rightsquigarrow^* (\text{rcase}_T\ t_2'\ y\ t_1\ t_2).$$

Therefore, $(\text{case } t_0 \text{ of } y.t_1,y.t_2) \rightsquigarrow^* (\text{rcase}_T\ t_0\ y\ t_1\ t_2)$.

Case. Suppose $t' \equiv t_1'\ t_2'$. By Lemma 12.2.3.11 there exists terms $\hat{t}_1'$ and $\hat{t}_2'$ such that $[t/x]^T t_1' = \hat{t}_1'$ and $[t/x]^T t_2' = \hat{t}_2'$. Since $t' > t_1'$ and $t' > t_2'$ we can apply the induction hypothesis to obtain $[t/x]t_1' \rightsquigarrow^* \hat{t}_1'$ and $[t/x]t_2' \rightsquigarrow^* \hat{t}_2'$. Now we case split on whether or not $\hat{t}_1'$ is a λ-abstraction and t_1' is not, $\hat{t}_1'$ is a case construct and t_1' is not, $ctype_T(x, t_1')$ is undefined, or $\hat{t}_1'$ is neither a λ-abstraction or a case construct. We only show the case where $\hat{t}_1'$ is a case construct.

Finally, suppose $\hat{t}_1' \equiv \text{case } t_0 \text{ of } y.t_1,y.t_2$ and t_0' is not a case construct. By Lemma 12.2.3.10 there exists a type T such that $ctype_T(x, t_1') = T$, $T \equiv T'' \to T'$ and T is a subexpression of T, where by inversion on the assumption $\Gamma, x : T, \Gamma' \vdash t' : T'$ there exists a type T'' such that $\Gamma, x : T, \Gamma' \vdash t_1' : T'' \to T'$. Now

$$[t/x]^T(t_1'\ t_2') = \text{case } t_0 \text{ of } y.(app_T\ t_1\ ([t/x]^T t_2')),y.(app_T\ t_1\ ([t/x]^T t_2'))$$

and

$$[t/x](t_1'\ t_2') = ([t/x]t_1')([t/x]t_2').$$

Clearly, $t' > t_1'$ and $t' > t_2'$, so by the induction hypothesis $[t/x]t_1' \rightsquigarrow^* [t/x]^T t_1'$ and $[t/x]t_2' \rightsquigarrow^* [t/x]^T t_2'$. Thus,

$$\begin{array}{rcl}(([t/x]t_1')\ ([t/x]t_2') & \rightsquigarrow^* & (\text{case } t_0 \text{ of } y.t_1,y.t_2)\ ([t/x]t_2') \\ & \rightsquigarrow & \text{case } t_0 \text{ of } y.(app_T\ t_1([t/x]t_2')),y.(app_T\ t_2([t/x]t_2'))\end{array}$$

and

$$((\text{case } t_0 \text{ of } y.(app_T\ t_1\ ([t/x]t_2')),y.(app_T\ t_1\ ([t/x]t_2')))) \rightsquigarrow^*$$

$$(\text{case } t_0 \text{ of } y.(app_T\ t_1\ ([t/x]^T t_2')),y.(app_T\ t_1\ ([t/x]^T t_2'))).$$

It suffices to show that $(t_1\ ([t/x]t_2')) \rightsquigarrow^* (app_T\ t_1\ ([t/x]t_2'))$ and $(t_2\ ([t/x]t_2')) \rightsquigarrow^* (app_T\ t_2\ ([t/x]t_2'))$. This is a consequence of the following proposition:

Proposition. For all $\Gamma \vdash t_1 : T_1 \to T_2$ and $\Gamma \vdash t_2 : T_1$ we have $(t_1\ t_2) \rightsquigarrow^* (app_T\ t_1\ t_2)$.

We prove this by nested induction on the ordering (T, t', t_1) and case split on the structure of t_1.

Case. Suppose t_1 is not a λ-abstraction or a case construct. Then $app_T\ t_1\ t_2 = t_1\ t_2$.

Case. Suppose $t_1 \equiv \lambda y : T_1.t_1''$. Then $app_T\ t_1\ t_2 = [t_2/y]^{T_1} t_1''$. Clearly, $T >_\Gamma T_1$ so by the outer-induction hypothesis $[t_2/y]t_1'' \rightsquigarrow^* [t_2/y]^{T_1} t_1''$. Therefore, $(t_1\ t_2) \rightsquigarrow^* (app_T\ t_1\ t_2)$.

Case. Suppose $t_1 \equiv \text{case } t_0' \text{ of } y.t_1',y.t_2'$. Then

$$app_T\ t_1\ t_2 = \text{case } t_0' \text{ of } y.(app_T\ t_1'\ t_2),y.(app_T\ t_2'\ t_2)$$

and

$$\begin{array}{rcl}(t_1\ t_2) & = & (\text{case } t_0' \text{ of } y.t_1',y.t_2')\ t_2 \\ & \rightsquigarrow & \text{case } t_0' \text{ of } y.(t_1'\ t_2),y.(t_2'\ t_2).\end{array}$$

We can see that $t_1 > t_1'$ and $t_1 > t_2'$ so by the inner-induction hypothesis, $(t_1'\ t_2) \rightsquigarrow^* (app_T\ t_1'\ t_2)$ and $(t_2'\ t_2) \rightsquigarrow^* (app_T\ t_2'\ t_2)$. Therefore,

(case t'_0 of $y.(t'_1\ t_2)$,$y.(t'_2\ t_2)$) $\rightsquigarrow^*$ (case t'_0 of $y.(app_T\ t'_1\ t_2)$,$y.(app_T\ t'_2\ t_2)$),

which implies $(t_1\ t_2) \rightsquigarrow^* (app_T\ t_1\ t_2)$.

□

12.2.4 Concluding Normalization

Similarly to SSF, the definition of the interpretation of types is identical to the definition for STLC (Definition 6.2.0.7), so we do not repeat it here. We define $t \rightsquigarrow^! t'$ to be $t \rightsquigarrow^* t'$ and t' is normal. Before moving on to proving soundness of typing and concluding normalization we need a basic result about the interpretation of types: type substitution. It is used in the proof of the type soundness theorem (Theorem 12.2.4.18).

Lemma 12.2.4.16. *[Type Substitution for the Interpretation of Types]*

If $n \in [\![T']\!]_{\Gamma,X:*_l,\Gamma'}$ *and* $\Gamma \vdash T : *_l$ *then* $[T/X]n \in [\![[T/X]T']\!]_{\Gamma,[T/X]\Gamma'}$.

Proof. This proof is similar to the proof of the same lemma for SSF (Lemma 12.1.5.27).

□

Next we now show substitution for the interpretation of types.

Lemma 12.2.4.17. *[Hereditary Substitution for the Interpretation of Types] If* $n' \in [\![T']\!]_{\Gamma,x:T,\Gamma'}$, $n \in [\![T]\!]_\Gamma$, *then* $[n/x]^T n' \in [\![T']\!]_{\Gamma,\Gamma'}$.

Proof. By Lemma 12.2.3.11 we know there exists a term $\hat{n}$ such that $[n/x]^T n' = \hat{n}$ and $\Gamma, \Gamma' \vdash \hat{n} : T'$ and by Lemma 12.2.3.14 $\hat{n}$ is normal. Therefore, $[n/x]^T n' = \hat{n} \in [\![T']\!]_{\Gamma,\Gamma'}$. □

Finally, we are ready to present our main result, which implies normalization of SSF^{+}.

Theorem 12.2.4.18. *[Type Soundness] If* $\Gamma \vdash t : T$ *then* $t \in [\![T]\!]_{\Gamma}$.

Proof. This is a proof by induction on the assumed typing derivation. We only show the cases that differ from the proof of type soundness for SSF (Theorem 12.1.6.28).

Case.

$$\frac{\Gamma \vdash t : T_1 \qquad \Gamma \vdash T_2 : *_p}{\Gamma \vdash inl(t) : T_1 + T_2}$$

By the induction hypothesis, $t \in [\![T_1]\!]_{\Gamma}$ and by the definition of the interpretation of types, $t \rightsquigarrow^{!} n \in [\![T_1]\!]_{\Gamma}$, and $inl(n) \in [\![T_1 + T_2]\!]_{\Gamma}$. Again, by the definition of the interpretation of types $inl(t) \rightsquigarrow^{!} inl(n) \in [\![T_1 + T_2]\!]_{\Gamma}$.

Case.

$$\frac{\Gamma \vdash t : T_2 \qquad \Gamma \vdash T_1 : *_p}{\Gamma \vdash inr(t) : T_1 + T_2}$$

Similar the inject-left case above.

Case.

$$\frac{\Gamma \vdash t_0 : T_1 + T_2 \qquad \Gamma, x : T_1 \vdash t_1 : T \qquad \Gamma, x : T_2 \vdash t_2 : T}{\Gamma \vdash \text{case } t_0 \text{ of } x.t_1, x.t_2 : T}$$

By the induction hypothesis and the definition of the interpretation of types $t_0 \rightsquigarrow^{!} n_0 \in [\![T_1 + T_2]\!]_{\Gamma}$ and $\Gamma \vdash n_0 : T_1 + T_2$, $t_1 \rightsquigarrow^{!} n_1 \in [\![T]\!]_{\Gamma, x:T_1}$ and $\Gamma, x : T_1 \vdash n_1 : T$, and $t_2 \rightsquigarrow^{!} n_2 \in [\![T]\!]_{\Gamma, x:T_2}$ and $\Gamma, x : T_2 \vdash n_2 : T$. Clearly,

$$\begin{array}{lll} \text{case } t_0 \text{ of } x.t_1,x.t_2 & \rightsquigarrow^* & \text{case } n_0 \text{ of } x.n_1,x.n_2 \\ & = & [n_0/z](\text{case } z \text{ of } x.n_1,x.n_2), \end{array}$$

for some variable $z \notin FV(n_0, n_1, n_2) \cup \{x\}$. Lemma 12.2.3.11, Lemma 12.2.3.15, and Lemma 12.2.3.14 allow us to conclude that $[n_0/z](\text{case } z \text{ of } x.n_1,x.n_2) \rightsquigarrow^* [n_0/z]^{T_1+T_2}(\text{case } z \text{ of } x.n_1,x.n_2)$, $\Gamma \vdash [n_0/z]^{T_1+T_2}(\text{case } z \text{ of } x.n_1,x.n_2) : T$, and $[n_0/z]^{T_1+T_2}(\text{case } z \text{ of } x.n_1,x.n_2)$ is normal. Thus, $[n_0/z]^{T_1+T_2}(\text{case } z \text{ of } x.n_1,x.n_2) \in [\![T]\!]_\Gamma$ and we obtain case t_0 of $x.t_1,x.t_2 \in [\![T]\!]_\Gamma$.

□

Corollary 12.2.4.19. *[Normalization] If $\Gamma \vdash t : T$, then there exists a normal form n, such that $t \rightsquigarrow^! n$.*

12.3 Dependent Stratified System F$^=$

Dependent Stratified System F$^=$ (DSSF$^=$) is SSF extended with dependent types and equations between terms. Equations between terms are an important concept in Martin-Löf type theory [69, 96], and play a central role also in dependently typed programming languages, such as the Guru language [131], Sep^3 (Section 8), the freedom of speech language (Section 7), Coq [134], and many more. The syntax and reduction rules are defined in Figure 57. The kind-assignment rules are defined in Figure 58. One thing to note regarding the kind-assignment rules is that the level of an equation is the same level as the type of the terms in the equation. The terms used in an equation must have the same type – also known as homogenous equality. Finally, the type-assignment rules are defined in Figure 59. Note that $t_1 \downarrow t_2$ denotes

$$
\begin{array}{lcl}
t & := & x \mid \lambda x : T.t \mid t\ t \mid \Lambda X : K.t \mid t[T] \mid join \\
T & := & X \mid \Pi x : T.T \mid \forall X : K.T \mid t = t \\
K & := & *_0 \mid *_1 \mid \ldots
\end{array}
$$

$$
\begin{array}{lcl}
(\Lambda X : *_p.t)[T] & \rightsquigarrow & [T/X]t \\
(\lambda x : T.t)t' & \rightsquigarrow & [t'/x]t
\end{array}
$$

Figure 57. Syntax of Terms, Types, and Kinds and Reduction Rules for DSSF$^=$

$$
\frac{\Gamma(X) = *_p \qquad \Gamma\ Ok \qquad p \leq q}{\Gamma \vdash X : *_q}
\qquad
\frac{\Gamma \vdash T_1 : *_p \qquad \Gamma, x : T_1 \vdash T_2 : *_q}{\Gamma \vdash \Pi x : T_1.T_2 : *_{max(p,q)}}
$$

$$
\frac{\Gamma, X : *_q \vdash T : *_p}{\Gamma \vdash \forall X : *_q.T : *_{max(p,q)+1}}
\qquad
\frac{\Gamma \vdash t_1 : T \qquad \Gamma \vdash t_2 : T \qquad \Gamma \vdash T : *_p}{\Gamma \vdash t_1 = t_2 : *_p}
$$

Figure 58. DSSF$^=$ Kinding Rules

there exists a term t such that $t_1 \rightsquigarrow^* t$ and $t_2 \rightsquigarrow^* t$.

12.3.1 Basic Syntactic Results

In this section we give a number of basic results. The reader may wish to skip this section, and only refer to the results while reading the rest of the section. The most interesting results covered in this section are the proof that type-syntactic equality is an equivalence relation, and syntactic inversion of the typing relation. The latter depends on a judgment called type-syntactic equality. It is defined in Figure 60. We show that syntactic conversion holds for syntactic-type equality in

$$\frac{\Gamma\ Ok \quad \Gamma(x) = T}{\Gamma \vdash x : T} \qquad \frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x : T_1.t : \Pi x : T_1.T_2} \qquad \frac{\Gamma \vdash t_2 : T_1 \quad \Gamma \vdash t_1 : \Pi x : T_1.T_2}{\Gamma \vdash t_1\ t_2 : [t_2/x]T_2}$$

$$\frac{\Gamma, X : *_l \vdash t : T}{\Gamma \vdash \Lambda X : *_l.t : \forall X : *_l.T} \qquad \frac{\Gamma \vdash t : \forall X : *_l.T_1 \quad \Gamma \vdash T_2 : *_l}{\Gamma \vdash t[T_2] : [T_2/X]T_1} \qquad \frac{t_1 \downarrow t_2 \quad \Gamma \vdash t_1 : T \quad \Gamma\ Ok \quad \Gamma \vdash t_2 : T}{\Gamma \vdash join : t_1 = t_2}$$

$$\frac{\Gamma \vdash t_0 : t_1 = t_2 \quad \Gamma \vdash t : [t_1/x]T}{\Gamma \vdash t : [t_2/x]T}$$

Figure 59. DSSF$^{=}$ Type-Assignment Rules

$$\frac{\Gamma \vdash p : t_1 = t_2}{\Gamma \vdash [t_1/x]T \approx [t_2/x]T}\ \text{TEQ}_1 \qquad \frac{\Gamma \vdash [t_1/x]T \approx [t_1/x]T' \quad \Gamma \vdash p : t_1 = t_2}{\Gamma \vdash [t_1/x]T \approx [t_2/x]T'}\ \text{TEQ}_2$$

Figure 60. DSSF$^{=}$ Type Syntactic Equality

Lemma 12.3.1.11. We only state the syntactic inversion lemma for the typing relation, because syntactic inversion for the kinding relation is trivial. Note that we use syntactic inversion for kinding throughout this section without indication. We now list several basic lemmata:

Lemma 12.3.1.1. *If* $\Gamma \vdash T : *_p$ *then* $\Gamma\ Ok$.

Proof. Similar to the proof of the same lemma for SSF^{+}. □

Lemma 12.3.1.2. *[Type Substitution for Kinding, Typing, and Context-Ok]*

Suppose $\Gamma \vdash T' : *_p$. *Then*

i. if $\Gamma, X : *_p, \Gamma' \vdash T : *_q$ *with a derivation of depth* d, *then* $\Gamma, [T'/X]\Gamma' \vdash [T'/X]T : *_q$ *with a derivation of depth* d,

ii. if $\Gamma, X : *_l, \Gamma' \vdash t : T$ *with a derivation of depth* d, *then* $\Gamma, [T'/X]\Gamma' \vdash [T'/X]t : [T'/X]T$ *with a derivation of depth* d, *and*

iii. if $\Gamma, X : *_p, \Gamma'$ *Ok with a derivation of depth* d, *then* $\Gamma, [T'/X]\Gamma'$ *Ok with a derivation of depth* d.

Proof. Similar to the proof of the same lemma for SSF^+. □

Lemma 12.3.1.3. *[Term Substitution for Kinding, Typing, and Context-Ok] Suppose* $\Gamma \vdash t' : T'$. *Then*

i. if $\Gamma, x : T', \Gamma' \vdash T : *_l$ *with a derivation of depth* d, *then* $\Gamma, [t'/x]\Gamma' \vdash [t'/x]T : *_l$ *with a derivation of depth* d,

ii. if $\Gamma, x : T', \Gamma' \vdash t : T$ *with a derivation of depth* d, *then* $\Gamma, [t'/x]\Gamma' \vdash [t'/x]t : [t'/x]T$ *with a derivation of depth* d, *and*

iii. if $\Gamma, x : T', \Gamma'$ *Ok with a derivation of depth* d, *then* $\Gamma, [t'/x]\Gamma'$ *Ok with a derivation of depth* d.

Proof. All cases hold by straightforward induction on the d. □

Lemma 12.3.1.4. *[Context Weakening for Kinding and Typing]*

Assume $\Gamma, \Gamma'', \Gamma'$ *Ok,* $\Gamma, \Gamma' \vdash T : *_p$ *and* $\Gamma, \Gamma' \vdash t : T$. *Then i.* $\Gamma, \Gamma'', \Gamma' \vdash T : *_p$ *and ii.* $\Gamma, \Gamma'', \Gamma' \vdash t : T$.

Proof. Similar to the proof of the same lemma for SSF^{+}. □

Lemma 12.3.1.5. *[Regularity] If* $\Gamma \vdash t : T$ *then* $\Gamma \vdash T : *_p$ *for some* p.

Proof. This holds by straightforward induction on the form of the assumed typing derivation. □

At this point we show that type-syntactic equality is indeed an equivalence relation, and that it respects substitution. Each of these may be used throughout the remainder of the section without explicit mention.

Lemma 12.3.1.6. *[Transitivity of Type Equality] If* $\Gamma \vdash T_1 \approx T_2$ *and* $\Gamma \vdash T_2 \approx T_3$ *then* $\Gamma \vdash T_1 \approx T_3$.

Proof. This holds by straightforward induction on the form of the assumed type equality derivation. □

Lemma 12.3.1.7. *[Symmetry of Type Equality] If* $\Gamma \vdash T \approx T'$ *then* $\Gamma \vdash T' \approx T$.

Proof. This holds by straightforward induction on the form of the assumed type equality derivation. □

Lemma 12.3.1.8. *[Substitution for Type Equality] If* $\Gamma, x : T, \Gamma' \vdash T' \approx T''$ *and* $\Gamma \vdash t : T$ *then* $\Gamma, [t/x]\Gamma' \vdash [t/x]T' \approx [t/x]T''$.

Proof. This holds by straightforward induction on the form of the assumed type equality derivation. □

Lemma 12.3.1.9. *If* $\Gamma \vdash T \approx \Pi j : T_1.T_2$ *then there exists a term* h *and types* T_1' *and* T_2' *such that* $T \equiv \Pi h : T_1'.T_2'$.

Proof. This is a proof by induction on the form of the assume type-equality derivation.

Case.

$$\frac{\Gamma \vdash p : t_1 = t_2}{\Gamma \vdash [t_1/x](\Pi j : T_1'.T_2') \approx [t_2/x](\Pi j : T_1'.T_2')} \text{ TEQ}_1$$

Trivial, because T must also be a Π-type.

Case.

$$\frac{\Gamma \vdash [t_1/x]T' \approx [t_1/x](\Pi j : T_1'.T_2') \qquad \Gamma \vdash p : t_1 = t_2}{\Gamma \vdash [t_1/x]T' \approx [t_2/x](\Pi j : T_1'.T_2')} \text{ TEQ}_2$$

By the induction hypothesis $T \equiv [t_1/x]T' \equiv \Pi h : \psi_1.\psi_2$ for some term h and types ψ_1 and ψ_2.

□

Lemma 12.3.1.10. *[Type Equality Context Conversion] If* $\Gamma, x : [T_1/X]T, \Gamma' \vdash t : T'$ *and* $\Gamma \vdash p : T_1 = T_2$ *then* $\Gamma, x : [T_2/X]T, \Gamma' \vdash t : T'$.

Proof. This hold by straightforward induction on the assumed typing derivation. □

The next lemma is type syntactic conversion which states that if a term t inhabits a type T, then it inhabits all types equivalent to T. Following this is injectivity of Π-types which is needed in the proof of syntactic inversion.

Lemma 12.3.1.11. *[Type Syntactic Conversion] If* $\Gamma \vdash t : T$ *and* $\Gamma \vdash T \approx T'$ *then* $\Gamma \vdash t : T'$.

Proof. If $\Gamma \vdash t : T$ and $\Gamma \vdash T \approx T'$ then we know several things: $T \equiv [\bar{t}/\bar{x}]T''$, $T' \equiv [\bar{t}'/\bar{x}']T''$, $\Gamma \vdash \bar{p} : \bar{t} = \bar{t}'$, and $\Gamma \vdash t : [\bar{t}/\bar{x}]T''$ for some type T''. Suppose each vector has i elements. Then by applying the conversion type-checking rule i times with the appropriate proof from our vector of proofs we will obtain $\Gamma \vdash t : [\bar{t}'/\bar{x}]T''$. This last result is exactly, $\Gamma \vdash t : T'$. □

Lemma 12.3.1.12. *[Injectivity of* Π*-Types for Type Equality] If* $\Gamma \vdash \Pi y : T_1.T_2 \approx \Pi y : T_1'.T_2'$ *then* $\Gamma \vdash T_1 \approx T_1'$ *and* $\Gamma, y : T_1 \vdash T_2 \approx T_2'$.

Proof. This is a proof by induction on the form of the assumed typing derivation.

Case.

$$\frac{\Gamma \vdash p : t_1 = t_2}{\Gamma \vdash [t_1/x]T' \approx [t_2/x]T'} \ \mathrm{TEQ}_1$$

Trivial, becasue T' and T'' only differ by terms, which do not affect the ordering of types.

Case.

$$\frac{\Gamma \vdash [t_1/x]T' \approx [t_1/x]T'' \qquad \Gamma \vdash p : t_1 = t_2}{\Gamma \vdash [t_1/x]T' \approx [t_2/x]T''} \ \mathrm{TEQ}_2$$

By the induction hypothesis $T >_\Gamma [t_1/x]T''$, which implies that $T >_\Gamma [t_2/x]T''$.

□

Finally, we prove syntactic inversion, but first we will need some convenient syntax that is used in the statement of the following lemma. We write $\exists(a_1, a_2, \ldots, a_n)$ for $\exists a_1.\exists a_2 \ldots \exists a_n$.

Lemma 12.3.1.13. *[Syntactic Inversion]*

i. *If* $\Gamma \vdash \lambda x : T_1.t : T$ *then* $\exists T_2.\ \Gamma, x : T_1 \vdash t : T_2 \wedge \Gamma \vdash \Pi x : T_1.T_2 \approx T$.

ii. *If* $\Gamma \vdash t_1\ t_2 : T$ *then* $\exists(x, T_1, T_2)$.

$\Gamma \vdash t_1 : \Pi x : T_1.T_2 \wedge \Gamma \vdash t_2 : T_1 \wedge \Gamma \vdash T \approx [t_2/x]T_2$.

iii. *If* $\Gamma \vdash \Lambda X : *_l.t : T$ *then* $\exists T'.\ \Gamma, X : *_l \vdash t : T' \wedge \Gamma \vdash T \approx \forall X : *_l.T'$.

iv. *If* $\Gamma \vdash t[T_2] : T$ *then* $\exists(T_1, T_2)$.

$\Gamma \vdash t : \forall X : *_l.T_1 \wedge \Gamma \vdash T_2 : *_l \wedge \Gamma \vdash T \approx [T_2/X]T_1$.

v. *If* $\Gamma \vdash join : T$ *then* $\exists(t_1, t_2, T')$.

$t_1 \downarrow t_2 \wedge \Gamma \vdash t_1 : T' \wedge \Gamma \vdash t_2 : T' \wedge \Gamma \vdash T \approx t_1 = t_2 \wedge \Gamma\ Ok$.

Proof. We prove all cases by induction on the form of the typing relation.

Case. Part i.

Case.

$$\frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x : T_1.t : \Pi x : T_1.T_2}$$

Trivial.

Case.

$$\frac{\Gamma \vdash p : t_1 = t_2 \qquad \Gamma \vdash \lambda x : T_1.t : [t_1/y]T'}{\Gamma \vdash \lambda x : T_1.t : [t_2/y]T'}$$

Here $T \equiv [t_2/y]T'$. By the induction hypothesis, where T is $[t_1/y]T'$, there exists a type T_2, such that $\Gamma, x : T_1 \vdash t : T_2$ and $\Gamma \vdash \Pi x : T_1.T_2 \approx [t_1/y]T'$, which implies that $\Gamma \vdash [t_1'/y](\Pi x : T_1'.T_2') \approx [t_1/y]T'$ and $\Gamma \vdash p' : t_1' = t_1$ for some terms t_1' and p'. Hence, by TEq_2 $\Gamma \vdash [t_1/y](\Pi x : T_1'.T_2') \approx [t_1/y]T'$ and by applying the same rule a second time, except using the proof $\Gamma \vdash p : t_1 = t_2$ we obtain $\Gamma \vdash [t_1/y](\Pi x : T_1'.T_2') \approx [t_2/y]T'$. Finally, using TEq_2 a third time using $\Gamma \vdash p' : t_1' = t_1$ we obtain $\Gamma \vdash [t_1'/y](\Pi x : T_1'.T_2') \approx [t_2/y]T'$, which is equivalent to $\Gamma \vdash \Pi x : T_1.T_2 \approx [t_2/y]T'$.

Case. Part ii.

Case.

$$\frac{\Gamma \vdash t_1 : \Pi x : T_1.T_2 \qquad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\ t_2 : [t_2/x]T_2}$$

Trivial.

Case.

$$\frac{\Gamma \vdash p : t_1 = t_2 \qquad \Gamma \vdash t_1\ t_2 : [t_1/y]T}{\Gamma \vdash t_1\ t_2 : [t_2/y]T}$$

Similar to the previous case.

Case. Part iii.

Case.

$$\frac{\Gamma, X : *_l \vdash t : T}{\Gamma \vdash \Lambda X : *_l . t : \forall X : *_l . T}$$

Trivial.

Case.

$$\frac{\Gamma \vdash p : t_1 = t_2 \qquad \Gamma \vdash \Lambda X : *_l . t : [t_1/y]T''}{\Gamma \vdash \Lambda X : *_l . t : [t_2/y]T''}$$

Similar to the previous case.

Case. Part iv.

Case.

$$\frac{\Gamma \vdash t : \forall X : *_l . T_1 \qquad \Gamma \vdash T_2 : *_l}{\Gamma \vdash t[T_2] : [T_2/X]T_1}$$

Trivial.

Case.

$$\frac{\Gamma \vdash p : t_1 = t_2 \qquad \Gamma \vdash t[T_2] : [t_1/y]T'}{\Gamma \vdash t[T_2] : [t_2/y]T'}$$

Similar to the previous case.

Case. Part v.

Case.

$$\frac{t_1 \downarrow t_2 \qquad \Gamma \vdash t_1 : T \qquad \Gamma \vdash t_2 : T \qquad \Gamma \; Ok}{\Gamma \vdash join : t_1 = t_2}$$

Trivial.

Case.

$$\frac{\Gamma \vdash p : t_1' = t_2' \qquad \Gamma \vdash join : [t_1'/y]T')}{\Gamma \vdash join : [t_2'/y]T'}$$

Similar to the previous case.

□

12.3.2 Hereditary Substitution

The ordering on types is defined as follows:

Definition 12.3.2.14.

The ordering $>_\Gamma$ is defined as the least relation satisfying the universal closure of the following formulas:

$$\begin{array}{lll} \Pi x : T_1.T_2 & >_\Gamma & T_1 \\ \Pi x : T_1.T_2 & >_\Gamma & [t/x]T_2, \textit{ where } \Gamma \vdash t : T_1. \\ \forall X : *_l.T & >_\Gamma & [T'/X]T, \textit{ where } \Gamma \vdash T' : *_l. \end{array}$$

Just as we have seen before this is the ordering used in the proofs of the properties of the hereditary substitution function, which state next. As one might have expected this is a well-founded ordering.

Theorem 12.3.2.15. *[Well-Founded Ordering] The ordering $>_\Gamma$ is well-founded on types T such that $\Gamma \vdash T : *_l$ for some l.*

Proof. This proof is similar to the same proof for SSF$^+$. It depends on the following function, and intermediate result.

Definition 12.3.2.16.

The depth of a type T is defined as follows:

$$\begin{array}{lcl} depth(t) & = & 0, \textit{ where } t \textit{ is any term.} \\ depth(X) & = & 1 \\ depth(\Pi x : T.T') & = & depth(T) + depth(T') \\ depth(\forall X : *_l.T) & = & depth(T) + 1 \end{array}$$

We use the metric (l, d) in lexicographic combination, where l is the level of a type T, and d is the depth of T in the proof of the next lemma.

Lemma 12.3.2.17. *[Well-Founded Measure] If* $T >_\Gamma T'$ *then* $(l, d) > (l', d')$*, where* $\Gamma \vdash T : *_l$*,* $depth(T) = d$*,* $\Gamma \vdash T : *_{l'}$*, and* $depth(T') = d'$*.*

Proof. Similar to the proof of the same lemma for SSF^+. □

□

The type-syntactic-equality relation respects this ordering.

Lemma 12.3.2.18. *If* $\Gamma \vdash T' \approx T''$ *and* $T >_\Gamma T'$ *then* $T >_\Gamma T''$*.*

Proof. This is a proof by case analysis on the kinding derivation of $\Gamma \vdash T : *_p$, with a case analysis on the derivation of $T >_\Gamma T'$.

Case.

$$\frac{\Gamma(X) = *_p \qquad p \leq q \qquad \Gamma \; Ok}{\Gamma \vdash X : *_q}$$

This case cannot arise, because we do not have $X >_\Gamma T$ for any type T.

Case.

$$\frac{\Gamma \vdash T_1 : *_p \qquad \Gamma \vdash T_2 : *_q}{\Gamma \vdash T_1 \to T_2 : *_{max(p,q)}}$$

By analysis of the derivation of the assumed ordering statement, we must have $T' \equiv T_1$ or $T' \equiv T_2$. If $T' \equiv T_1$ and $p \geq q$ then we have the required kind derivation for T'. If $p < q$ then by level weakening $\Gamma \vdash T_1 : *_q$, and we have the required kinding derivation for T'. The case for when $T' \equiv T_2$ is similar.

Case.

$$\frac{\Gamma \vdash T_1 : *_p \qquad \Gamma \vdash T_2 : *_q}{\Gamma \vdash T_1 + T_2 : *_{max(p,q)}}$$

By analysis of the derivation of the assumed ordering statement, we must have $T' \equiv T_1$ or $T' \equiv T_2$. If $T' \equiv T_1$ and $p \geq q$ then we have the required kind derivation for T'. If $p < q$ then by level weakening $\Gamma \vdash T_1 : *_q$, and we have the required kinding derivation for T'. The case for when $T' \equiv T_2$ is similar.

Case.

$$\frac{\Gamma, X : *_r \vdash T : *_s}{\Gamma \vdash \forall X : *_r . T : *_{max(r,s)+1}}$$

By analysis of the derivation of the assumed ordering statement, we must have $T' \equiv [T''/X]T$, for some type T'' with $\Gamma \vdash T'' : *_r$. Let $t = max(r, s) + 1$. Clearly, $s < t$, hence by level weakening $\Gamma, X : *_r \vdash T : *_t$ and by substitution

for kinding $\Gamma \vdash [T''/X]T : *_t$, and we have the required kinding derivation for T'.

□

The following lemma is used in the proof of totality of the hereditary substitution function.

Lemma 12.3.2.19. *[Congruence of Products] If* $\Gamma \vdash \psi \approx \Pi y : T_1.T_2$ *and* ψ *is a subexpression of* T'' *then* $T'' >_\Gamma T_1$ *and* $T'' >_{\Gamma,y:T_1} T_2$.

Proof. The possible form for ψ is only a Π-type. So it suffices to show that if $\Gamma \vdash \Pi y : T_1'.T_2' \approx \Pi y : T_1.T_2$ and ψ is a subexpression of T'' then $T'' >_\Gamma T_1$ and $T'' >_{\Gamma,y:T_1} T_2$.

It must be the case that $T'' >_\Gamma T_1'$ and $T'' >_{\Gamma,y:T_1} T_2'$, because T_1' and T_2' are both subexpressions of T''. By injectivity of Π-types for typed equality we obtain $\Gamma \vdash T_1' \approx T_1$ and $\Gamma, y : T_1 \vdash T_2' \approx T_2$. Finally, by Lemma 12.3.2.18 we know $T'' >_\Gamma T_1$ and $T'' >_{\Gamma,y:T_1} T_2$. □

The hereditary substitution function is fully defined in Figure 61. We do not repeat the definition of $ctype_T$ for DSSF$^=$, because it is exactly the same as the previous system. Next we have the properties of the hereditary substitution function. All of the proofs are similar to the proofs for SSF$^+$, so we omit them here.

Lemma 12.3.2.20. *[Total, Type Preserving, and Sound w.r.t Reduction]*

Suppose $\Gamma \vdash t : T$ *and* $\Gamma, x : T, \Gamma' \vdash t' : T'$. *Then there exists a term* t'' *and a type* T'' *such that* $[t/x]^T t' = t''$, $\Gamma, [t/x]\Gamma' \vdash t'' : T''$, *and* $\Gamma, \Gamma' \vdash T'' \approx [t/x]T'$,

and $[t/x]t' \rightsquigarrow^* [t/x]^T t'$.

$[t/x]^T x = t$

$[t/x]^T y = y$
Where y is a variable distinct from x.

$[t/x]^T join = join$

$[t/x]^T(\lambda y : T'.t') = \lambda y : T'.([t/x]^T t')$

$[t/x]^T(\Lambda X : *_l.t') = \Lambda X : *_l.([t/x]^T t')$

$[t/x]^T(t_1\ t_2) = ([t/x]^T t_1)\ ([t/x]^T t_2)$
Where $([t/x]^T t_1)$ is not a λ-abstraction, $([t/x]^T t_1)$ and t_1 are λ-abstractions, or $ctype_T(x, t_1)$ is undefined.

$[t/x]^T(t_1\ t_2) = [([t/x]^T t_2)/y]^{T''} s_1'$
Where $([t/x]^T t_1) \equiv \lambda y : T''.s_1'$ for some y and s_1' and t_1 is not a λ-abstraction, and $ctype_T(x, t_1) = \Pi y : T''.T'$.

$[t/x]^T(t'[T']) = ([t/x]^T t')[T']$
Where $[t/x]^T t'$ is not a type abstraction or t' and $[t/x]^T t'$ are type abstractions.

$[t/x]^T(t'[T']) = [T'/X]s_1'$
Where $[t/x]^T t' \equiv \Lambda X : *_l.s_1'$, for some X, s_1' and $\Gamma \vdash T' : *_q$, such that, $q \leq l$ and t' is not a type abstraction.

Figure 61. Hereditary Substitution Function for Stratified System $F^=$

Corollary 12.3.2.21. *Suppose* $\Gamma \vdash t : T$ *and* $\Gamma, x : T, \Gamma' \vdash t' : T'$. *Then* $\Gamma, [t/x]\Gamma' \vdash [t/x]^T t' : [t/x]T'$.

Lemma 12.3.2.22. *[Redex Preserving] If* $\Gamma \vdash t : T$, $\Gamma, x : T, \Gamma' \vdash t' : T'$ *then* $|rset(t', t)| \geq |rset([t/x]^T t')|$.

Proof. Just as we have seen for the previous systems this proof depends on the following function:

$rset(x) = \emptyset$

$rset(join) = \emptyset$

$rset(\lambda x : T.t) = rset(t)$

$rset(\Lambda X : *_l.t) = rset(t)$

$rset(t_1\ t_2)$

$$\begin{array}{lll} = & rset(t_1, t_2) & \text{if } t_1 \text{ is not a } \lambda\text{-abstraction.} \\ = & \{t_1\ t_2\} \cup rset(t_1', t_2) & \text{if } t_1 \equiv \lambda x : T.t_1'. \end{array}$$

$rset(t''[T''])$

$$\begin{array}{lll} = & rset(t'') & \text{if } t'' \text{ is not a type absraction.} \\ = & \{t''[T'']\} \cup rset(t''') & \text{if } t'' \equiv \Lambda X : *_l.t'''. \end{array}$$

□

Lemma 12.3.2.23. *[Normality Preserving] If* $\Gamma \vdash n : T$ *and* $\Gamma, x : T' \vdash n' : T'$ *then there exists a normal term* n'' *such that* $[n/x]^T n' = n''$.

12.3.3 Concluding Normalization

We are now ready to conlcude normalization. The interpretation of types are defined exactly the same way as they were for STLC (Definition 6.2.0.7). We do not repeat that definition here. The next two lemmas are used in the proof of the type soundness theorem.

Corollary 12.3.3.24. *[Type Substitution for the Interpretation of Types] If* $n \in [\![T']\!]_{\Gamma, X:*_l, \Gamma'}$ *and* $\Gamma \vdash T : *_l$ *then* $[T/X]n \in [\![[T/X]T']\!]_{\Gamma, [T/X]\Gamma'}$.

Proof. By the definition of the interpretation of types, $\Gamma, X : *_l, \Gamma' \vdash n : T'$ and by Lemma 12.3.1.2, $\Gamma, [T/X]\Gamma' \vdash [T/X]n : [T/X]T'$. Finally, by the definition of the interpretation of types, $[T/X]n \in [\![[T/X]T']\!]_{\Gamma, [T/X]\Gamma'}$. □

Lemma 12.3.3.25. *[Semantic Equality] If* $\Gamma \vdash p : t_1 = t_2$ *then* $[\![[t_1/x]T]\!]_\Gamma = [\![[t_2/x]T]\!]_\Gamma$.

Proof. We first prove the left to right containment. Suppose $t \rightsquigarrow^* n \in [\![[t_1/x]T]\!]_\Gamma$. Then by the definition of the interpretation of types, $\Gamma \vdash n : [t_1/x]T$. By assumption we know $\Gamma \vdash p : t_1 = t_2$, hence, by applying the conversion type-checking rule $\Gamma \vdash n : [t_2/x]T$. Finally, by the definition of the interpretation of types, $n \in [\![[t_2/x]T]\!]_\Gamma$. Therefore, $t \in [\![[t_2/x]T]\!]_\Gamma$. The opposite direction is similar. □

We now conclude type soundness for $\text{SSF}^=$, and hence normalization.

Lemma 12.3.3.26. *[Hereditary Substitution for the Interpretation of Types] If* $n' \in [\![T']\!]_{\Gamma,x:T,\Gamma'}$, $n \in [\![T]\!]_\Gamma$, *then* $[n/x]^T n' \in [\![[n/x]T']\!]_{\Gamma,[n/x]\Gamma'}$.

Proof. This proof is similar to the same proof for SSF^+. □

Theorem 12.3.3.27. *[Type Soundness] If* $\Gamma \vdash t : T$ *then* $t \in [\![T]\!]_\Gamma$.

Proof. This is a proof by induction on the structure of the typing derivation of t.

Case.

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

By regularity $\Gamma \vdash T : *_l$ for some l, hence $[\![T]\!]_\Gamma$ is nonempty. Clearly, $x \in [\![T]\!]_\Gamma$ by the definition of the interpretation of types.

Case.

$$\frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x : T_1.t : \Pi x : T_1.T_2}$$

By the induction hypothesis and the definition of the interpretation of types $t \in [\![T_2]\!]_{\Gamma,x:T_1}$, $t \rightsquigarrow^! n \in [\![T_2]\!]_{\Gamma,x:T_1}$ and $\Gamma, x : T_1 \vdash n : T_2$. Thus, by applying the λ-abstraction type-checking rule, $\Gamma \vdash \lambda x : T_1.n : \Pi x : T_1.T_2$, hence by the definition of the interpretation of types $\lambda x : T_1.n \in [\![\Pi x : T_1.T_2]\!]_\Gamma$. Since $\lambda x : T_1.t \rightsquigarrow^! \lambda x : T_1.n \in [\![\Pi x : T_1.T_2]\!]_\Gamma$ we know by the definition of the interpretation of types $\lambda x : T_1.t \in [\![\Pi x : T_1.T_2]\!]_\Gamma$.

Case.

$$\frac{\Gamma \vdash t_1 : \Pi x : T_1.T_2 \qquad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\ t_2 : [t_2/x]T_2}$$

It suffices to show that there exists a normal term n such that $t_1\ t_2 \rightsquigarrow^! n \in [\![[t_2/x]T_2]\!]_\Gamma$. By the induction hypothesis and the definition of the interpretation of types $t_1 \rightsquigarrow^! n_1 \in [\![\Pi x : T_1.T_2]\!]_\Gamma$, $\Gamma \vdash n_1 : \Pi x : T_1.T_2$, $t_2 \rightsquigarrow^! n_2 \in [\![T_1]\!]_\Gamma$, and $\Gamma \vdash n_2 : T_1$. Clearly,

$$\begin{array}{rcl} t_1\ t_2 & \rightsquigarrow^* & n_1\ n_2 \\ & = & [n_1/z](z\ n_2), \end{array}$$

for some fresh variable $z \notin FV(n_1, n_2, T_1, T_2, x)$. By Lemma 12.3.2.20, Lemma 12.3.2.23, and Corollary 12.3.2.21 $[n_1/x](z\ n_2) \rightsquigarrow^* [n_1/x]^{T_1}(z\ n_2)$, $[n_1/x]^{T_1}(z\ n_2)$ is normal, and $\Gamma \vdash [n_1/x]^{T_1}(z\ n_2) : [n_2/x]T_2$. Thus, $t_1\ t_2 \rightsquigarrow^! [n_1/x]^{T_1}(z\ n_2)$. It suffices to show that $\Gamma \vdash [n_1/x]^{T_1}(z\ n_2) : [t_2/x]T_2$. This is justified by the following typing derivation:

$$\dfrac{\dfrac{\Gamma \vdash t_2 : T_1 \quad \Gamma \vdash n_2 : T_1 \qquad n_2 \downarrow t_2}{\Gamma \vdash join : n_2 = t_2}\ \text{Join} \qquad \Gamma \vdash [n_1/x]^{T_1}(z\ n_2) : [n_2/x]T_2}{\Gamma \vdash [n_1/x]^{T_1}(z\ n_2) : [t_2/x]T_2}\ \text{Conv}$$

Therefore, $[n_1/x]^{T_1}(z\ n_2) \in [\![[t_2/x]T_2]\!]_\Gamma$ which implies that $t_1\ t_2 \in [\![[t_2/x]T_2]\!]_\Gamma$.

Case.

$$\dfrac{t_1 \downarrow t_2 \qquad \Gamma\ Ok}{\Gamma \vdash join : t_1 = t_2}$$

Clearly, $join \in [\![t_1 = t_2]\!]_\Gamma$ by the definition of the interpretation of types.

Case.

$$\dfrac{\Gamma \vdash t_0 : t_1 = t_2 \qquad \Gamma \vdash t : [t_1/x]T}{\Gamma \vdash t : [t_2/x]T}$$

By the induction hypothesis, $t \in [\![[t_1/x]T]\!]_\Gamma$. By the definition of the interpretation of types, $t \rightsquigarrow^!\ n \in [\![[t_1/x]T]\!]_\Gamma$. By assumption we know, $\Gamma \vdash t_0 : t_1 = t_2$. Thus, by Lemma 12.3.3.25, $[\![[t_1/x]T]\!]_\Gamma = [\![[t_2/x]T]\!]_\Gamma$. Therefore, $n \in [\![[t_2/x]T]\!]_\Gamma$, hence, by the definition of the interpretation of types, $t \in [\![[t_2/x]T]\!]_\Gamma$.

Case.

$$\dfrac{\Gamma, X : *_p \vdash t : T}{\Gamma \vdash \Lambda X : *_p.t : \forall X : *_p.T}$$

By the induction hypothesis, $t \in [\![T]\!]_{\Gamma, X:*_p}$, so by the definition of the interpretation of types, $t \rightsquigarrow^!\ n \in [\![T]\!]_{\Gamma, X:*_p}$ and $\Gamma, X : *_p \vdash n : T$. We can apply

the Λ-abstraction type-checking rule to obtain $\Gamma \vdash \Lambda X : *_p.n : \forall X : *_p.T$, thus $\Lambda X : *_p.n \in [\![\forall X : *_p.T]\!]_\Gamma$. Since $\Lambda X : *_p.t \rightsquigarrow^! \Lambda X : *_p.n$ by definition of the interpretation of types $\Lambda X : *_p.t \in [\![\forall X : *_p.T]\!]_\Gamma$.

Case.

$$\frac{\Gamma \vdash t : \forall X : *_l.T_1 \qquad \Gamma \vdash T_2 : *_l}{\Gamma \vdash t[T_2] : [T_2/X]T_1}$$

By the induction hypothesis $t \in [\![\forall X : *_l.T_1]\!]_\Gamma$, so by the definition of the interpretation of types $t \rightsquigarrow^! n \in [\![\forall X : *_l.T_1]\!]_\Gamma$ and $\Gamma \vdash n : \forall X : *_l.T_1$. We do a case split on whether or not n is a Λ-abstraction. We can apply the type-instantiation type-checking rule to obtain $\Gamma \vdash n[T_2] : [T_2/X]T_1$ and by the definition of the interpretation of types $n[T_2] \in [\![[T_2/X]T_1]\!]_\Gamma$. Therefore, $t \in [\![[T_2/X]T_1]\!]_\Gamma$. Suppose $n \equiv \Lambda X : *_l.n'$. Then $t[T_2] \rightsquigarrow^* (\Lambda X : *_l.n')[T_2] \rightsquigarrow [T_2/X]n'$. By inversion $n' \in [\![T_1]\!]_{\Gamma, X:*_l}$. Therefore, by Lemma 12.3.3.24 $[T_2/X]n' \in [\![[T_2/X]T_1]\!]_\Gamma$ and $t[T_2] \in [\![[T_2/X]T_1]\!]_\Gamma$, since $t[T_2] \downarrow [T_2/X]n'$.

□

Corollary 12.3.3.28. *[Normalization] If $\Gamma \vdash t : T$, then there exists a normal form n, such that $t \rightsquigarrow^! n$.*

CHAPTER 13

THE $\lambda\Delta$-CALCULUS

So far in this chapter we have introduced proofs of normalization for several extensions of SSF, which is an intuitionistic logic. In fact, to the knowledge of the author the only type theories to which the hereditary substitution proof technique has been applied to are intuitionistic. So a natural question is, can hereditary substitution be used to prove normalization of a classical type theory? This would then imply that hereditary substitution can be used to provide a constructive proof of normalization for a non-constructive theory.

In this section we answer this question positively. We show normalization using hereditary substitution for the classical type theory the $\lambda\Delta$-calculus. The $\lambda\Delta$-calculus was introduced in Section 3.2. We do not repeat its definition. The reader can find the syntax and reduction rules in Figure 17 and the typing rules in Figure 18. We define negation just as it is in intuitionistic type theory, that is, $\neg A =^{def} A \to \bot$, where $\bot$ is absurdity. Arbitrary syntactically defined normal forms will be denoted by the meta-variables n and m, and arbitrary typing contexts will be denoted by the meta-variable Γ. We assume at all times that all variables in the domain of Γ are unique. In addition we rearrange the objects in Γ freely without indication.

13.1 Basic Syntactic Lemmas

The following meta-results are well-known so we omit their proofs. We do not always explicitly state the use of these results. The first two properties are weakening

and substitution for the typing relation.

Lemma 13.1.0.1. *[Weakening for Typing] If* $\Gamma \vdash t : T$ *then* $\Gamma, x : T' \vdash t : T$ *for any fresh variable* x *and type* T'*.*

Proof. Straightforward induction on the assumed typing derivation. □

Lemma 13.1.0.2. *[Substitution for Typing] If* $\Gamma \vdash t : T$ *and* $\Gamma, x : T, \Gamma' \vdash t' : T'$ *then* $\Gamma \vdash [t/x]t' : T'$*.*

Proof. Straightforward induction on the second assumed typing derivation. □

The final three properties are, confluence, type preservation and inversion of the typing relation. The proof of the confluence and type preservation can be found in [113] and the proof of the latter is trivial.

Theorem 13.1.0.3. *[Confluence] If* $t_1 \rightsquigarrow^* t_2$ *and* $t_1 \rightsquigarrow^* t_3$*, then there exists a term* t_4*, such that,* $t_2 \rightsquigarrow^* t_4$ *and* $t_3 \rightsquigarrow^* t_4$*.*

Theorem 13.1.0.4. *[Preservation] If* $\Gamma \vdash t : T$ *and* $t \rightsquigarrow t'$ *then* $\Gamma \vdash t' : T$*.*

Theorem 13.1.0.5. *[Inversion]*

i. If $\Gamma \vdash x : T$ *then* $x \in \Gamma$*.*

ii. If $\Gamma \vdash \lambda x : T_1.t : T_1 \to T_2$ *then* $\Gamma, x : T_1 \vdash t : T_2$*.*

iii. If $\Gamma \vdash \Delta x : \neg T.t : T$ *then* $\Gamma, x : \neg T \vdash t : \bot$*.*

Proof. This can be shown by straightforward induction on the assumed typing derivations. □

At this point we have everything we need to state and prove correct the hereditary substitution function.

13.2 An Extension

Since the $\lambda\Delta$-calculus is an extension of STLC, we might expect that the hereditary substitution function for the $\lambda\Delta$-calculus is also an extension of the hereditary substitution function for STLC. In this section we show that this extension is non-trivial by first considering the naive extension, and then discussing why it does not work. Following this, we give the final extension and prove it correct.

13.2.1 Problems with a Naive Extension

Lets consider the definition of the hereditary substitution function for STLC extended with two new cases. The first case for the Δ-abstraction whose definition parallels the definition for the λ-abstraction. The second is a new application case which handles newly created structural redexes and is defined following the same pattern as the case which handles β-redexes. We use the same termination metric we previously used.

Definition 13.2.1.1.

The naive hereditary substitution function is defined as follows:

$$[t/x]^A x = t$$

$$[t/x]^A y = y$$

$$[t/x]^A(\lambda y : A'.t') = \lambda y : A'.([t/x]^A t')$$

$$[t/x]^A(\Delta y : A'.t') = \Delta y : A'.([t/x]^A t')$$

$[t/x]^A(t_1\ t_2) = ([t/x]^A t_1)\,([t/x]^A t_2)$

Where $([t/x]^A t_1)$ *is not a* λ*-abstraction or* Δ*-abstraction, or both* $([t/x]^A t_1)$ *and* t_1 *are* λ*-abstractions or* Δ*-abstractions.*

$[t/x]^A(t_1\ t_2) = [s_2'/y]^{A''} s_1'$

Where $([t/x]^A t_1) = \lambda y : A''.s_1'$ *for some* y, s_1' *and* A'', $[t/x]^A t_2 = s_2'$, *and* $\mathsf{ctype}_A(x, t_1) = A'' \to A'$.

$[t/x]^A(t_1\ t_2) = \Delta z : \neg A'.[\lambda y : A'' \to A'.(z\,(y\,s_2))/y]^{\neg(A'' \to A')} s$

Where $([t/x]^A t_1) = \Delta y : \neg(A'' \to A').s$ *for some,* y s, *and* $A'' \to A'$, $([t/x]^A t_2) = s_2$ *for some* s_2, $\mathsf{ctype}_A(x, t_1) = A'' \to A'$, *and* z *is completely fresh.*

There is one glaring issue with this definition and it lies in the final case. We know from Lemma 13.2.3.4 and Lemma 13.2.3.5 that $\mathsf{ctype}_A(x, t_1) = A'' \to A'$ implies that $A \geq A'' \to A' < \neg(A'' \to A')$. Thus, this definition is not well founded! To fix this issue instead of naively following the structural reduction rule we immediately simultaneously hereditarily reduce all redexes created by replacing y with the linear λ-abstraction $\lambda y : A'' \to A'.(z\,(y\,s_2))$. To accomplish this we will define mutually with the hereditary substitution function a new function called the hereditary structural substitution function.

13.2.2 A Correct Extension

In order to reduce structural redexes in the definition of the hereditary substitution we will define by induction mutually with the hereditary substitution function a function called the hereditary structural substitution function. This function will use the notion of a multi-substitution. These are given by the following grammar:

$$\Theta ::= \cdot \mid \Theta, (y, z, t)$$

We denote the hereditary structural substitution function by $\langle\Theta\rangle^{A}_{A'}t'$ and hereditary substitution by $[t/x]^{A}t'$. The type of all the first projections of the elements of Θ is $\neg(A \to A')$ and the type of the second projections is $\neg A'$. Both functions are defined by mutual induction using the metric (A, f, t'), where $f \in \{0, 1\}$, in lexicographic combination with the ordering on types, the natural number ordering, and the strict subexpression on terms. The meta-variable f labels each function and is equal to 0 in the definition of the hereditary substitution function and is equal to 1 in the definition of the hereditary structural substitution function. Again, in the definitions of the hereditary substitution and hereditary structural substitution function it is assumed that all variables have been renamed as to prevent variable capture. The following is the final definition of the hereditary substitution function for the $\lambda\Delta$-calculus.

Definition 13.2.2.2.

The hereditary substitution function is defined as follows:

$\langle\Theta\rangle^{A_1}_{A_2}x = \lambda y : A_1 \to A_2.(z\,(y\,t))$

Where $(x, z, t) \in \Theta$, for some z and t, and y is fresh in x, z, and t.

$\langle\Theta\rangle^{A_1}_{A_2}x = x$

Where $(x, z, t) \notin \Theta$ for any z or t.

$\langle\Theta\rangle^{A_1}_{A_2}(\lambda y : A.t) = \lambda y : A.\langle\Theta\rangle^{A_1}_{A_2}t$

$\langle\Theta\rangle^{A_1}_{A_2}(\Delta y : A.t) = \Delta y : A.\langle\Theta\rangle^{A_1}_{A_2}t$

$\langle\Theta\rangle^{A_1}_{A_2}(x\,t') = z\,[t/y]^{A_1}s$

Where $(x, z, t) \in \Theta$, $t' \equiv \lambda y : A_1.t''$, *for some* y *and* t'', *and* $\langle\Theta\rangle^{A_1}_{A_2} t'' = s$.

$\langle\Theta\rangle^{A_1}_{A_2}(x\ t') = z\ (\Delta z_2 : \neg A_2.s)$
Where $(x, z, t) \in \Theta$, $t' \equiv \Delta y : \neg(A_1 \to A_2).t''$, *for some* y *and* t'', *and* $\langle\Theta, (y, z_2, t)\rangle^{A_1}_{A_2} t'' = s$, *for some fresh* z_2.

$\langle\Theta\rangle^{A_1}_{A_2}(x\ t') = z\ s'$
Where $(x, z, t) \in \Theta$, t' *is not an abstraction, and* $\langle\Theta\rangle^{A_1}_{A_2} t' = s'$.

$\langle\Theta\rangle^{A_1}_{A_2}(t_1\ t_2) = s_1\ s_2$
Where t_1 *is either not a variable, or it is both a variable and* $(t_1, z', t') \notin \Theta$ *for any* t' *and* z', $\langle\Theta\rangle^{A_1}_{A_2} t_1 = s_1$, *and* $\langle\Theta\rangle^{A_1}_{A_2} t_2 = s_2$.

$[t/x]^A x = t$

$[t/x]^A y = y$

$[t/x]^A(\lambda y : A'.t') = \lambda y : A'.([t/x]^A t')$

$[t/x]^A(\Delta y : A'.t') = \Delta y : A'.([t/x]^A t')$

$[t/x]^A(t_1\ t_2) = ([t/x]^A t_1)\ ([t/x]^A t_2)$
Where $([t/x]^A t_1)$ *is not a* λ*-abstraction or* Δ*-abstraction, or both* $([t/x]^A t_1)$ *and* t_1 *are* λ*-abstractions or* Δ*-abstractions.*

$[t/x]^A(t_1\ t_2) = [s_2'/y]^{A''} s_1'$
Where $([t/x]^A t_1) = \lambda y : A''.s_1'$ *for some* y, s_1' *and* A'', $[t/x]^A t_2 = s_2'$, *and* $\mathsf{ctype}_A(x, t_1) = A'' \to A'$.

$[t/x]^A(t_1\ t_2) = \Delta z : \neg A'.\langle(y, z, s_2)\rangle^{A''}_{A'} s$
Where $([t/x]^A t_1) = \Delta y : \neg(A'' \to A').s$ *for some* y s, *and* $A'' \to A'$, $([t/x]^A t_2) = s_2$ *for some* s_2, $\mathsf{ctype}_A(x, t_1) = A'' \to A'$, *and* z *is fresh.*

We can see in the final case of the hereditary substitution function that the cut type has decreased. Hence, this case is now well founded. Lets consider an example which illustrates how our new definition operates.

Example 13.2.2.3.

Consider the terms $t \equiv \Delta f : \neg(b \to b).(f\,(\Delta f' : \neg(b \to b).(f'\,(\lambda z : b.z))))$ *and* $t' \equiv x\,u$*, where* u *is a free variable of type* b*. Again, our goal is to compute* $[t/x]^{(b\to b)}t'$ *using the definition of the hereditary substitution function in Definition 13.2.2.2. Now*

$$[t/x]^{(b\to b)}(x\,u) = \Delta z_1 : \neg b.(z_1\,(\Delta z_2 : \neg b.(z_2\,u))),$$

because

$$\mathsf{ctype}_{(b\to b)}(x, x) = (b \to b), \qquad [t/x]^{(b\to b)}x = t, \quad [t/x]^{(b\to b)}u = u,$$

and for some fresh variable z_1 *of type* $\neg b$

$$\begin{array}{l}\Delta z_1 : \neg b.\langle (f, z_1, u)\rangle_b^b(f\,(\Delta f' : \neg(b \to b).(f'\,(\lambda z : b.z)))) \;\; = \\ \Delta z_1 : \neg b.(z_1\,(\Delta z_2 : \neg b.(z_2\,u)))\end{array}$$

where

$$\begin{array}{l}\langle (f, z_1, u)\rangle_b^b(f\,(\Delta f' : \neg(b \to b).(f'\,(\lambda z : b.z)))) \qquad = \\ z_1\,(\Delta z_2 : \neg b.\langle (f, z_1, u), (f', z_2, u)\rangle_b^b(f'\,(\lambda z : b.z)))\end{array}$$

because

$$(f, z_1, u) \in \langle (f, z_1, u)\rangle, \Delta f' : \neg(b \to b).(f'\,(\lambda z : b.z)) \equiv \Delta f' : \neg(b \to b).(f'\,(\lambda z : b.z)),$$

and for some fresh variable z_2 *of type* $\neg b$

$$\langle (f, z_1, u), (f', z_2, u)\rangle_b^b(f'\,(\lambda z : b.z)) \;\; = \;\; z_2\,u$$

because

$$(f', z_2, u) \in \langle (f, z_1, u), (f', z_2, u)\rangle, \qquad \lambda z : b.z \equiv \lambda z : b.z, \quad \langle (f, z_1, u)\rangle_b^b z = z$$

In the next section we prove the definition of the hereditary substitution function correct.

13.2.3 Main Properties

Just as we did for the previous system we now prove the properties of hereditary substitution. We introduce some notation to make working with multi-substitutions a bit easier. The sets of all first, second, and third projections of the triples in Θ are denoted Θ^1, Θ^2, and Θ^3 respectively. We denote the assumption of all elements of Θ^i having the type T as $\Theta^i : T$. This latter notation is used in typing contexts to indicate the addition of all the variables in Θ^j for $j \in \{1, 2\}$ to the context with the specified type. We denote this as $\Gamma, \Theta^j : T, \Gamma'$ for some contexts Γ and Γ'. The notation $\Gamma \vdash \Theta^3 : T$ is defined as for all $t \in \Theta^3$ the typing judgment $\Gamma \vdash t : T$ holds. Finally, we denote terms in Θ^3 being normal as $\mathsf{norm}(\Theta^3)$.

All of the following properties will depend on a few properties of the ctype function. They are listed in the following lemma.

Lemma 13.2.3.4. *[Properties of ctype]*

i. If $\mathsf{ctype}_T(x, t) = T'$ then $\mathsf{head}(t) = x$ and $T' \leq T$.

ii. If $\Gamma, x : T, \Gamma' \vdash t : T'$ and $\mathsf{ctype}_T(x, t) = T''$ then $T' \equiv T''$.

Proof. We prove part one first. This is a proof by induction on the structure of t.

Case. Suppose $t \equiv x$. Then $\mathsf{ctype}_T(x, x) = T$. Clearly, $\mathsf{head}(x) = x$ and T is a subexpression of itself.

Case. Suppose $t \equiv t_1\ t_2$. Then $\mathsf{ctype}_T(x, t_1\ t_2) = T''$ when $\mathsf{ctype}_T(x, t_1) = T' \to T''$. Now $t > t_1$ so by the induction hypothesis $\mathsf{head}(t_1) = x$ and $T' \to T''$

is a subexpression of T. Therefore, $\mathsf{head}(t_1\, t_2) = x$ and certainly T'' is a subexpression of T.

We now prove part two. This is also a proof by induction on the structure of t.

Case. Suppose $t \equiv x$. Then $\mathsf{ctype}_T(x, x) = T$. Clearly, $T \equiv T$.

Case. Suppose $t \equiv t_1\, t_2$. Then $\mathsf{ctype}_T(x, t_1\, t_2) = T_2$ when $\mathsf{ctype}_T(x, t_1) = T_1 \to T_2$. By inversion on the assumed typing derivation we know there exists type T'' such that $\Gamma, x : T, \Gamma' \vdash t_1 : T'' \to T'$. Now $t > t_1$ so by the induction hypothesis $T_1 \to T_2 \equiv T'' \to T'$. Therefore, $T_1 \equiv T''$ and $T_2 \equiv T'$.

□

Lemma 13.2.3.5. *[Properties of ctype Continued]*

i. If $\Gamma, x : T, \Gamma' \vdash t_1\, t_2 : T'$, $\Gamma \vdash t : T$, $[t/x]^T t_1 = \lambda y : T_1.t'$, and t_1 is not a λ-abstraction, then t_1 is in head normal form and there exists a type A such that $\mathsf{ctype}_T(x, t_1) = A$.

ii. If $\Gamma, x : T, \Gamma' \vdash t_1\, t_2 : T'$, $\Gamma \vdash t : T$, $[t/x]^T t_1 = \Delta y : \neg(T'' \to T').t'$, and t_1 is not a Δ-abstraction, then there exists a type A such that $\mathsf{ctype}_T(x, t_1) = A$.

Proof. We prove part one first. This is a proof by induction on the structure of $t_1\, t_2$.

The only possibilities for the form of t_1 is x or $s_1\, s_2$. All other forms would not result in $[t/x]^T t_1$ being a λ-abstraction and t_1 not. If $t_1 \equiv x$ then there exist a type T''

such that $T \equiv T'' \to T'$ and $\mathsf{ctype}_T(x, x\ t_2) = T'$ when $\mathsf{ctype}_T(x, x) = T \equiv T'' \to T'$ in this case. We know T'' to exist by inversion on $\Gamma, x : T, \Gamma' \vdash t_1\ t_2 : T'$.

Now suppose $t_1 \equiv s_1\ s_2$. Now knowing t_1 to not a λ-abstraction implies that s_1 is also not a λ-abstraction or $[t/x]^T t_1$ would be an application instead of a λ-abstraction. So it must be the case that $[t/x]^T s_1$ is a λ-abstraction and s_1 is not. Since $s_1 < t_1$ we can apply the induction hypothesis to obtain there exists a type A such that $\mathsf{ctype}_T(x, s_1) = A$. Now by inversion on $\Gamma, x : T, \Gamma' \vdash t_1\ t_2 : T'$ we know there exists a type T'' such that $\Gamma, x : T, \Gamma' \vdash t_1 : T'' \to T'$. We know $t_1 \equiv s_1\ s_2$ so by inversion on $\Gamma, x : T, \Gamma' \vdash t_1 : T'' \to T'$ we know there exists a type A'' such that $\Gamma, x : T, \Gamma' \vdash s_1 : A'' \to (T'' \to T')$. By part two of Lemma 13.2.3.4 we know $A \equiv A'' \to (T'' \to T')$ and $\mathsf{ctype}_T(x, t_1) = \mathsf{ctype}_T(x, s_1\ s_2) = T'' \to T'$ when $\mathsf{ctype}_T(x, s_1) = A'' \to (T'' \to A')$, because we know $\mathsf{ctype}_T(x, s_1) = A$.
The proof of part two is similar to the proof of part one. □

The first two properties of the hereditary substitution function are totality and type preservation.

Lemma 13.2.3.6. *[Totality and Type Preservation]*

i. If $\Gamma \vdash \Theta^3 : A$ *and* $\Gamma, \Theta^1 : \neg(A \to A') \vdash t' : B$*, then there exists a term* s *such that* $\langle\Theta\rangle^A_{A'} t' = s$ *and* $\Gamma, \Theta^2 : \neg A' \vdash s : B$.

ii. If $\Gamma \vdash t : A$ *and* $\Gamma, x : A, \Gamma' \vdash t' : B$*, then there exists a term* s *such that* $[t/x]^A t' = s$ *and* $\Gamma, \Gamma' \vdash s : B$.

Proof. This is a mutually inductive proof using the lexicographic combination (A, f, t')

of our ordering on types, the natural number ordering where $f \in \{0, 1\}$, and the strict subexpression ordering on terms. We first prove part one and then part two. In both parts we case split on t'.

Part One.

Case. Suppose t' is a variable x. Then either there exists a term a such that $(x, z, a) \in \Theta$ or not. Suppose so. Then $\langle\Theta\rangle^{A}_{A'}x = \lambda y : A \to A'.(z\,(y\,a))$ where y is fresh in x, z and a. Now suppose there does not exist any term a or z such that $(x, z, a) \in \Theta$. Then $\langle\Theta\rangle^{A}_{A'}x = x$. Typing clearly holds, because if $(x, z, a) \in \Theta$ then $B \equiv \neg(A \to A')$ and we know $\Gamma, \Theta^2 : \neg A' \vdash \lambda y : A \to A'.(z\,(y\,a)) : B$ or $x \notin \Theta^1$ then it must be the case that $x : B \in \Gamma$, hence, by assumption and weakening for typing $\Gamma, \Theta^2 : \neg A' \vdash x : B$.

Case. It must be the case that $B \equiv B_1 \to B_2$ for some types B_1 and B_2. Suppose $t' \equiv \lambda y : B_1.t'_1$. Then $\langle\Theta\rangle^{A}_{A'}t' = \langle\Theta\rangle^{A}_{A'}(\lambda y : B_1.t'_1) = \lambda y : B_1.\langle\Theta\rangle^{A}_{A'}t'_1$. Now sense $(A, 1, t') > (A, 1, t'_1)$ we may apply the induction hypothesis to obtain that there exists a term s such that $\langle\Theta\rangle^{A}_{A'}t'_1 = s$, and $\Gamma, \Theta^2 : \neg A', y : B_1 \vdash s : B_2$. Thus, by definition and the typing rule for λ-abstractions we obtain $\langle\Theta\rangle^{A}_{A'}t' = \lambda y : B_1.s$ and $\Gamma, \Theta^2 : \neg A' \vdash \lambda y : B_1.s : B_1 \to B_2$.

Case. Suppose $t' \equiv \Delta y : \neg B.t'_1$. Similar to the previous case.

Case. Suppose $t' \equiv t'_1\ t'_2$. We have two cases to consider.

Case. Suppose $t'_1 \equiv x$ for some variable x. In each case $B \equiv \bot$.

Case. Suppose $t'_2 \equiv \lambda y : A.t''_2$, for some y and t''_2, $(x, z, t) \in \Theta$. Since

$(A, 1, t') > (A, 1, t_2'')$ and the typing assumptions hold by inversion we can apply the induction hypothesis to obtain $\langle\Theta\rangle^A_{A'} t_2'' = s$ for some term s and $\Gamma, \Theta^2 : \neg A', y : A \vdash s : A'$. Furthermore, sense $(A, 1, t') > (A, 0, s)$, the previous typing condition and the typing assumptions we also know from the induction hypothesis that $[t/y]^A s = s'$ for some term s' and $\Gamma, \Theta^2 : \neg A' \vdash s' : A'$. Finally, by definition we know $\langle\Theta\rangle^A_{A'} t' = z\,([t/y]^A s) = z\,s'$ and by using the application typing rule that $\Gamma, \Theta^2 : \neg A' \vdash z\,s' : B$.

Case. Suppose $t_2' \equiv \Delta y : \neg(A \to A').t_2''$, for some y and t_2'', $(x, z, t) \in \Theta$. Since $(A, 1, t') > (A, 1, t_2'')$ we know from the induction hypothesis that $\langle\Theta, (y, z_2, t)\rangle^A_{A'} t_2'' = s$ for some fresh variable z and term s, and $\Gamma, \Theta^2 : \neg A', z_2 : \neg A' \vdash s :\!\!\perp$. Finally, $\langle\Theta\rangle^A_{A'} t' = z\,(\Delta z_2 : \neg A'.s)$ by definition, and by using the application typing rule $\Gamma, \Theta^2 : \neg A' \vdash z\,(\Delta z_2 : \neg A'.s) : B$.

Case. Suppose t_2' is not an abstraction, and $(x, z, t) \in \Theta$. Since $(A, 1, t') > (A, 1, t_2')$ we know from the induction hypothesis that $\langle\Theta\rangle^A_{A'} t_2' = s$ for some term s and $\Gamma, \Theta^2 : \neg A' \vdash s : A \to A'$. Finally, $\langle\Theta\rangle^A_{A'} t' = z\,s$ by definition, and by using the application typing rule $\Gamma, \Theta^2 : \neg A' \vdash z\,s : B$.

Case. Suppose $(x, z, t'') \notin \Theta$ for any term t'' and z. Since $(A, 1, t') > (A, 1, t_2')$ we know from the induction hypothesis that $\langle\Theta\rangle^A_{A'} t_2' = s$ for some term s and $\Gamma, \Theta^2 : \neg A' \vdash s : A \to A'$. Finally, $\langle\Theta\rangle^A_{A'} t' = x\,s$

by definition, and by using the application typing rule $\Gamma, \Theta^2 : \neg A' \vdash x\ s : B$.

Case. Suppose t'_1 is not a variable. This case follows easily from the induction hypothesis.

Part two.

Case. Suppose t' is either x or a variable y distinct from x. Trivial in both cases.

Case. Suppose $t' \equiv \lambda y : A_1.t'_1$. By inversion we know there exists a type A_2 such that $\Gamma, x : A, \Gamma', y : A_1 \vdash t'_1 : A_2$. We also know that t'_1 is a strict subexpression of t', hence we can apply the second part of the induction hypothesis to obtain $[t/x]^A t'_1 = s_1$ and $\Gamma, \Gamma', y : A_1 \vdash s_1 : A_2$ for some term s_1. By the definition of the hereditary substitution function

$$\begin{array}{rcl} [t/x]^A t' & = & \lambda y : A_1.[t/x]^A t'_1 \\ & = & \lambda y : A_1.s_1. \end{array}$$

It suffices to show that $\Gamma, \Gamma' \vdash \lambda y : A_1.s_1 : A_1 \to A_2$. By simply applying the typing rule Lam using $\Gamma, \Gamma', y : A_1 \vdash s_1 : A_2$ we obtain $\Gamma, \Gamma' \vdash \lambda y : A_1.s_1 : A_1 \to A_2$.

Case. Suppose $t' \equiv \Delta y : \neg B.t'_1$. Similar to the previous case.

Case. Suppose $t' \equiv t'_1\ t'_2$. By inversion we know $\Gamma, x : A, \Gamma' \vdash t'_1 : B' \to B$ and $\Gamma, x : A, \Gamma' \vdash t'_2 : B'$ for some type B'. Clearly, t'_1 and t'_2 are strict subexpressions of t'. Thus, by the second part of the induction hypothesis there exists terms s_1 and s_2 such that $[t/x]^A t'_1 = s_1$ and $[t/x]^A t'_2 = s_2$, and $\Gamma, \Gamma' \vdash s_1 : B' \to B'$ and $\Gamma, \Gamma' \vdash s_2 : B'$. We case split on whether or not s_1 is a λ-abstraction

or a Δ-abstraction and t_1' is not, or s_1 and t_1' are both a λ-abstraction or a Δ-abstraction. We only consider the non-trivial cases when $s_1 \equiv \lambda y : B'.s_1'$ and t_1' is not a λ-abstraction, and $s_1 \equiv \Delta y : \neg(B' \to B).s_1'$ and t_1' is not a Δ-abstraction. Consider the former.

Now by Lemma 13.2.3.4 it is the case that there exists a B'' such that $\mathsf{ctype}_A(x, t_1') = B''$, $B'' \equiv B' \to B$, and B is a subexpression of A, hence $A > B'$. By the definition of the hereditary substitution function $[t/x]^A(t_1'\ t_2') = [s_2/y]^{B'} s_1'$. Therefore, by the induction hypothesis there exists a term s such that $[s_2/y]^A s_1' = s$ and $\Gamma, \Gamma' \vdash s : B$.

At this point consider when $s_1 \equiv \Delta y : \neg(B' \to B).s_1'$ and t_1' is not a Δ-abstraction. Again, by Lemma 13.2.3.4 it is the case that there exists a B'' such that $\mathsf{ctype}_A(x, t_1') = B''$, $B'' \equiv B' \to B$ and $B' \to B$ is a subexpression of A. Hence, $A > B'$. Let r be a fresh variable of type $\neg B$. Then by the induction hypothesis, there exists a term s'', such that, $\langle (y, r, s_2) \rangle_B^{B'} s_1' = s''$ and $\Gamma, r : \neg B \vdash s'' : \bot$. Therefore, $[t/x]^A(t_1'\ t_2') = \Delta r : \neg B.\langle (y, r, s_2) \rangle_B^{B'} s_1' = \Delta r : \neg B.s''$, and by the Δ-abstraction typing rule $\Gamma \vdash \Delta r : \neg B.s'' : B$.

□

The next property shows that the hereditary substitution function is normality preserving. The proof of normality preservation depends on the following auxiliary result.

Lemma 13.2.3.7. *For any Θ, A and A', if $n_1\ n_2$ is normal then $\mathsf{head}(\langle \Theta \rangle_{A'}^A (n_1\ n_2))$ is a variable.*

Proof. This is a proof by induction on the form of $n_1\, n_2$. In every case where n_1 is a variable and $(n_1, z, t) \in \Theta$ for some term t and variable z, we know by definition that $\langle\Theta\rangle^A_{A'}(n_1\, n_2) = z\, t_2$ for some variable z and term t_2. In the case where n_1 is a variable and $(n_1, z, t) \notin \Theta$ for some term t and variable z, we know by definition that $\langle\Theta\rangle^A_{A'}(n_1\, n_2) = (\langle\Theta\rangle^A_{A'} n_1)\,(\langle\Theta\rangle^A_{A'} n_2)$. Now by hypothesis and definition $\langle\Theta\rangle^A_{A'} n_1 = n_1$. Thus, $(\langle\Theta\rangle^A_{A'} n_1)\,(\langle\Theta\rangle^A_{A'} n_2) = n_1\,(\langle\Theta\rangle^A_{A'} n_2)$ and we know n_1 is a variable. The final case is when n_1 is not a variable. Then it must be the case that n_1 is a normal application. So by the induction hypothesis $\mathsf{head}(\langle\Theta\rangle^A_{A'} n_1)$ is a variable. Therefore, $\mathsf{head}(\langle\Theta\rangle^A_{A'}(n_1\, n_2)) = \mathsf{head}((\langle\Theta\rangle^A_{A'} n_1)\,(\langle\Theta\rangle^A_{A'} n_2)) = \mathsf{head}(\langle\Theta\rangle^A_{A'} n_1)$ is a variable. □

Lemma 13.2.3.8. *[Normality Preservation]*

i. If $\mathsf{norm}(\Theta^3)$, $\Gamma \vdash \Theta^3 : A$ *and* $\Gamma, \Theta^1 : \neg(A \to A') \vdash n' : B$, *then there exists a normal form* m *such that* $\langle\Theta\rangle^A_{A'} n' = m$.

ii. If $\Gamma \vdash n : A$ *and* $\Gamma, x : A, \Gamma' \vdash n' : B$ *then there exists a term* m *such that* $[n/x]^A n' = m$.

Proof. This is a mutually inductive proof using the lexicographic combination (A, f, n') of our ordering on types, the natural number ordering where $f \in \{0, 1\}$, and the strict subexpression ordering on terms. We first prove part one and then part two. In both parts we case split on n'.

Part One.

Case. Suppose n' is a variable x. Then either there exists a normal form m and variable z, such that, $(x, z, m) \in \Theta$ or not. Suppose so. Then $\langle\Theta\rangle^A_{A'} x =$

$\lambda y : A \to A'.(z\,(y\,m))$ where y is fresh in x, z and m. Clearly, $\lambda y : A \to A'.(z\,(y\,m))$ is normal. Now suppose there does not exist any term m or z such that $(x, z, m) \in \Theta$. Then $\langle\Theta\rangle^{A}_{A'}x = x$ which is clearly normal.

Case. Suppose $n' \equiv \lambda y : B_1.n'_1$. Then $\langle\Theta\rangle^{A}_{A'}n' = \langle\Theta\rangle^{A}_{A'}(\lambda y : B_1.n'_1) = \lambda y : B_1.\langle\Theta\rangle^{A}_{A'}n'_1$. Now sense $(A, 1, n') > (A, 1, n'_1)$ we may apply the induction hypothesis to obtain that there exists a term m such that $\langle\Theta\rangle^{A}_{A'}n'_1 = m$. Thus, by definition we obtain $\langle\Theta\rangle^{A}_{A'}n' = \lambda y : B_1.m$.

Case. Suppose $n' \equiv \Delta y : \neg B.n'_1$. Similar to the previous case.

Case. Suppose $n' \equiv n'_1\,n'_2$. We have two cases to consider.

Case. Suppose $n'_1 \equiv x$ for some variable x.

Case. Suppose $n'_2 \equiv \lambda y : A.n''_2$, for some y and n''_2, $(x, z, n) \in \Theta$. Since $(A, 1, n') > (A, 1, n''_2)$ and the typing assumptions hold by inversion we can apply the induction hypothesis to obtain $\langle\Theta\rangle^{A}_{A'}n''_2 = m$ for some term m. We know from Lemma 13.2.3.8 that $\Gamma, \Theta^2 : \neg A', y : A \vdash m : A'$. Furthermore, sense $(A, 1, n') > (A, 0, m)$, the previous typing condition and the typing assumptions we also know from the induction hypothesis that $[t/y]^A m = m'$ for some term m'. Finally, by definition we know $\langle\Theta\rangle^{A}_{A'}n' = z\,([n/y]^A m) = z\,m'$. It is easy to see that $z\,m'$ is normal.

Case. Suppose $n'_2 \equiv \Delta y : \neg(A \to A').n''_2$, for some y and n''_2, $(x, z, n) \in \Theta$. Since $(A, 1, n') > (A, 1, n''_2)$ we know from the induction hypothesis that $\langle\Theta, (y, z_2, n)\rangle^{A}_{A'}n''_2 = m$ for some normal form m, and $\Gamma, \Theta^2 :$

$\neg A', z_2 : \neg A' \vdash m : \bot$. Finally, $\langle\Theta\rangle^A_{A'} n' = z\,(\Delta z_2 : \neg A'.m)$ by definition.

Case. Suppose n'_2 is not an abstraction, and $(x, z, n) \in \Theta$. Since $(A, 1, n') > (A, 1, n'_2)$ we know from the induction hypothesis that $\langle\Theta\rangle^A_{A'} n'_2 = m$ for some normal form m. Finally, $\langle\Theta\rangle^A_{A'} n' = z\,m$ by definition.

Case. Suppose $(x, z, n'') \notin \Theta$ for any term n'' and z. Since $(A, 1, n') > (A, 1, n'_2)$ we know from the induction hypothesis that $\langle\Theta\rangle^A_{A'} n'_2 = m$ for some term m. Finally, $\langle\Theta\rangle^A_{A'} n' = x\,m$ by definition.

Case. Suppose n'_1 is not a variable. This case follows easily from the induction hypothesis and Lemma 13.2.3.7.

Part two.

Case. Suppose n' is either x or a variable y distinct from x. Trivial in both cases.

Case. Suppose $n' \equiv \lambda y : B_1.n'_1$. We also know that n'_1 is a strict subexpression of n', hence we can apply the second part of the induction hypothesis to obtain $[n/x]^A n'_1 = m_1$ for some normal form m_1. By the definition of the hereditary substitution function

$$\begin{array}{rcl} [n/x]^A n' & = & \lambda y : A_1.[n/x]^A n'_1 \\ & = & \lambda y : B_1.m_1. \end{array}$$

Clearly, $\lambda y : B_1.m_1$ is normal.

Case. Suppose $n' \equiv \Delta y : \neg B.n'_1$. Similar to the previous case.

Case. Suppose $t' \equiv t'_1\,t'_2$. Clearly, n'_1 and n'_2 are strict subexpressions of n'. Thus, by the induction hypothesis there exists normal forms n_1 and n_2 such that

$[n/x]^A n_1' = m_1$ and $[n/x]^A n_2' = m_2$. We case split on whether or not m_1 is a λ-abstraction or a Δ-abstraction and n_1' is not, or m_1 and n_1' are both a λ-abstraction or a Δ-abstraction. We only consider the non-trivial cases when $m_1 \equiv \lambda y : B'.m_1'$ and n_1' is not a λ-abstraction, and $m_1 \equiv \Delta y : \neg(B' \to B).m_1'$ and n_1' is not a Δ-abstraction. Consider the former.

Now by Lemma 13.2.3.4 it is the case that there exists a B'' such that $\mathsf{ctype}_A(x, t_1') = B''$, $B'' \equiv B' \to B$, and B is a subexpression of A, hence $A > B'$. By the definition of the hereditary substitution function $[n/x]^A(n_1'\ n_2') = [m_2/y]^{B'} m_1'$. Therefore, by the induction hypothesis there exists a normal form m such that $[m_2/y]^A m_1' = m$.

At this point consider when $m_1 \equiv \Delta y : \neg(B' \to B).m_1'$ and n_1' is not a Δ-abstraction. Again, by Lemma 13.2.3.4 it is the case that there exists a B'' such that $\mathsf{ctype}_A(x, t_1') = B''$, $B'' \equiv B' \to B$ and $B' \to B$ is a subexpression of A. Hence, $A > B'$. Let r be a fresh variable of type $\neg B$. Then by the induction hypothesis, there exists a term m'', such that, $\langle (y, r, m_2) \rangle_B^{B'} m_1' = m''$ and Therefore, $[n/x]^A(n_1'\ n_2') = \Delta r : \neg B.\langle (y, r, m_2) \rangle_B^{B'} m_1' = \Delta r : \neg B.m''$.

□

The final correctness property of the hereditary substitution function is soundness with respect to reduction. We need one last piece of notation. Suppose $\Theta = (x_1, z_1, t_1), \ldots, (x_i, z_i, t_i)$ for some natural number i. Then $\langle \Theta \rangle^{\uparrow_{A'}^A}\ t' =^{def} [\lambda y : A \to A'.(z_i\ (y\ t_i))/x_i](\cdots([\lambda y : A \to A'.(z_1\ (y\ t_1))/x_1]t_1)\cdots)$.

Lemma 13.2.3.9. *[Soundness with Respect to Reduction]*

i. If $\Gamma \vdash \Theta^3 : A$ *and* $\Gamma, \Theta^1 : \neg(A \to A') \vdash t' : B$*, then* $\langle\Theta\rangle^{\uparrow^A_{A'}}\, t' \rightsquigarrow^* \langle\Theta\rangle^A_{A'} t'$.

ii. If $\Gamma \vdash t : A$ *and* $\Gamma, x : A, \Gamma' \vdash t' : B$ *then* $[t/x]t' \rightsquigarrow^* [t/x]^A t'$.

Proof. This is a mutually inductive proof using the lexicographic combination (A, f, t') of our ordering on types, the natural number ordering where $f \in \{0, 1\}$, and the strict subexpression ordering on terms. We first prove part one and then part two. In both parts we case split on t'.

Part One.

Case. Suppose t' is a variable x. Then either there exists a term a such that $(x, z, a) \in \Theta$ or not. Suppose so. Then by definition we know $\langle\Theta\rangle^{\uparrow^A_{A'}}\, x = \lambda y : A \to A'.(z\,(y\,a))$, for some fresh variable y. Now $\langle\Theta\rangle^A_{A'} x = \lambda y : A \to A'.(z\,(y\,a))$, where we choose the same y. Thus, $\langle\Theta\rangle^{\uparrow^A_{A'}}\, x \rightsquigarrow^* \langle\Theta\rangle^A_{A'} x$. Now suppose there does not exist any term a or z such that $(x, z, a) \in \Theta$. Then $\langle\Theta\rangle^A_{A'} x = \langle\Theta\rangle^{\uparrow^A_{A'}}\, x = x$. Thus, $\langle\Theta\rangle^{\uparrow^A_{A'}}\, x \rightsquigarrow^* \langle\Theta\rangle^A_{A'} x$.

Case. Suppose $t' \equiv \lambda y : B_1.t'_1$. This case follows from the induction hypothesis.

Case. Suppose $t' \equiv \Delta y : \neg B.t'_1$. Similar to the previous case.

Case. Suppose $t' \equiv t'_1\, t'_2$. We have two cases to consider.

Case. Suppose $t'_1 \equiv x$ for some variable x.

Case. Suppose $t'_2 \equiv \lambda y : A.t''_2$, for some y and t''_2, $(x, z, t) \in \Theta$. Now

$$\begin{aligned}
&\langle\Theta\rangle^{\uparrow^A_{A'}}\, (x\,(\lambda y : A.t''_2)) \\
&= (\lambda y : A \to A'.(z\,(y\,t)))\,(\lambda y : A.(\langle\Theta\rangle^{\uparrow^A_{A'}}\, t''_2)) \\
&\rightsquigarrow z\,((\lambda y : A.(\langle\Theta\rangle^{\uparrow^A_{A'}}\, t''_2))\,t) \\
&\rightsquigarrow z\,([t/y](\langle\Theta\rangle^{\uparrow^A_{A'}}\, t''_2))
\end{aligned}$$

Since $(A, 1, t') > (A, 1, t_2'')$ we can apply the induction hypothesis to obtain $\langle\Theta\rangle^{\uparrow^A_{A'}}\, t_2'' \rightsquigarrow^* \langle\Theta\rangle^A_{A'} t_2''$. Hence,

$$z\,([t/y](\langle\Theta\rangle^{\uparrow^A_{A'}}\, t_2'')) \quad \rightsquigarrow^* \quad z\,([t/y](\langle\Theta\rangle^A_{A'} t_2''))$$

Furthermore, sense $(A, 1, t') > (A, 0, \langle\Theta\rangle^A_{A'} t_2'')$, we also know from the induction hypothesis that

$$\begin{array}{lll} z\,([t/y](\langle\Theta\rangle^A_{A'} t_2'')) & \rightsquigarrow^* & z\,([t/y]^A(\langle\Theta\rangle^A_{A'} t_2'')) \\ & = & \langle\Theta\rangle^A_{A'} t' \end{array}$$

Case. Suppose $t_2' \equiv \Delta y' : \neg(A \to A').t_2''$, for some y and t_2'', $(x, z, t) \in \Theta$.

Now using a fresh variable z_2 we know

$$\begin{array}{l} \langle\Theta\rangle^{\uparrow^A_{A'}}\, (x\,(\Delta y' : \neg(A \to A').t_2'')) \\ \quad = (\lambda y : A \to A'.(z\,(y\,t)))\,(\Delta y' : \neg(A \to A').(\langle\Theta\rangle^{\uparrow^A_{A'}}\, t_2'')) \\ \quad \rightsquigarrow z\,((\Delta y' : \neg(A \to A').(\langle\Theta\rangle^{\uparrow^A_{A'}}\, t_2''))\,t) \\ \quad \rightsquigarrow z\,(\Delta z_2 : \neg A'.([\lambda y : A \to A'.(z_2\,(y\,t))/y'](\langle\Theta\rangle^{\uparrow^A_{A'}}\, t_2''))) \\ \quad = z\,(\Delta z_2 : \neg A'.(\langle\Theta, (y', z_2, t)\rangle^{\uparrow^A_{A'}}\, t_2'')) \end{array}$$

Since $(A, 1, t') > (A, 1, t_2'')$ we know from the induction hypothesis that $\langle\Theta, (y', z_2, t)\rangle^{\uparrow^A_{A'}}\, t_2'' \rightsquigarrow^* \langle\Theta, (y', z_2, t)\rangle^A_{A'} t_2''$. Thus,

$$\begin{array}{lll} z\,(\Delta z_2 : \neg A'.(\langle\Theta, (y', z_2, t)\rangle^{\uparrow^A_{A'}}\, t_2'')) & \rightsquigarrow^* & z\,(\Delta z_2 : \neg A'.(\langle\Theta, (y', z_2, t)\rangle^A_{A'} t_2'')) \\ & = & \langle\Theta\rangle^A_{A'} t'. \end{array}$$

Case. Suppose t_2' is not an abstraction, and $(x, z, t) \in \Theta$. Since $(A, 1, t') > (A, 1, t_2')$ we know from the induction hypothesis that $\langle\Theta\rangle^{\uparrow^A_{A'}}\, t_2' \rightsquigarrow^* \langle\Theta\rangle^A_{A'} t_2'$. Thus,

$$\begin{array}{lll} \langle\Theta\rangle^{\uparrow^A_{A'}}\, t' & = & \langle\Theta\rangle^{\uparrow^A_{A'}}\, (x\, t_2') \\ & = & z\,(\langle\Theta\rangle^{\uparrow^A_{A'}}\, t_2') \\ & \rightsquigarrow^* & z\,(\langle\Theta\rangle^A_{A'} t_2') \\ & = & \langle\Theta\rangle^A_{A'} (x\, t_2') = \langle\Theta\rangle^A_{A'} t'. \end{array}$$

Case. Suppose $(x, z, t'') \notin \Theta$ for any term t'' and z. Since $(A, 1, t') > (A, 1, t_2')$ we know from the induction hypothesis that $\langle\Theta\rangle^{\uparrow^A_{A'}}\, t_2' \rightsquigarrow^* \langle\Theta\rangle^A_{A'} t_2'$. Thus,

$$\begin{array}{rcl}\langle\Theta\rangle^{\uparrow^A_{A'}}\, t' & = & \langle\Theta\rangle^{\uparrow^A_{A'}}\,(x\; t'_2) \\ & = & x\;(\langle\Theta\rangle^{\uparrow^A_{A'}}\; t'_2) \\ & \rightsquigarrow^* & x\;(\langle\Theta\rangle^A_{A'} t'_2) \\ & = & \langle\Theta\rangle^A_{A'}(x\; t'_2) = \langle\Theta\rangle^A_{A'} t'.\end{array}$$

Case. Suppose t'_1 is not a variable. This case follows easily from the induction hypothesis.

Part two

Case. Suppose t' is a variable x or y distinct from x. Trivial in both cases.

Case. Suppose $t' \equiv \lambda y : B_1.s$. Then $[t/x](\lambda y : B_1.s) = \lambda y : B_1.([t/x]s)$. Now s is a strict subexpression of t' so we can apply the second part of the induction hypothesis to obtain $[t/x]s \rightsquigarrow^* [t/x]^A s$. At this point we can see that since $\lambda y : B_1.[t/x]s \equiv [t/x](\lambda y : B_1.s)$ we may conclude that $\lambda y : B_1.[t/x]s \rightsquigarrow^* \lambda y : B_1.[t/x]^A s$.

Case. Suppose $t' \equiv \Delta y : \neg B.s$. Similar to the previous case.

Case. Suppose $t' \equiv t'_1\; t'_2$. By Lemma 13.2.3.6 there exists terms s_1 and s_2 such that $[t/x]^A t'_1 = s_1$ and $[t/x]^A t'_2 = s_2$. Since t'_1 and t'_2 are strict subexpressions of t' we can apply the second part of the induction hypothesis to obtain $[t/x]t'_1 \rightsquigarrow^* s_1$ and $[t/x]t'_2 \rightsquigarrow^* s_2$. Now we case split on whether or not s_1 is a λ-abstraction and t'_1 is not, a Δ-abstraction and t'_1 is not, or s_1 is not a λ-abstraction or a Δ-abstraction. If s_1 is not a λ-abstraction or a Δ-abstraction then $[t/x]^A t' = ([t/x]^A t'_1)\,([t/x]^A t'_2) \equiv s_1\; s_2$. Thus, by two applications of the induction hypothesis, $[t/x]t' \rightsquigarrow^* [t/x]^A t'$, because $[t/x]t' = ([t/x]t'_1)\,([t/x]t'_2)$. Suppose $s_1 \equiv \lambda y : B'.s'_1$ and t'_1 is not a λ-abstraction. By Lemma 13.2.3.4

there exists a type B'' such that $\mathsf{ctype}_A(x, t_1') = B''$, $B'' \equiv B' \to B$, and B'' is a subexpression of A. Then by the definition of the hereditary substitution function $[t/x]^A(t_1'\ t_2') = [s_2/y]^{B'} s_1'$. Now we know $A > B'$ so we can apply the second part of the induction hypothesis to obtain $[s_2/y]s_1' \rightsquigarrow^* [s_2/y]^{B'} s_1'$. By knowing that $((\lambda y : B'.s_1')\ s_2) \rightsquigarrow ([s_2/y]s_1')$ and by the previous fact we know $(\lambda y : B'.s_1')\ s_2 \rightsquigarrow^* [s_2/y]^{B'} s_1'$. We now make use of the well known result of full β-reduction. The result is stated as

$$\frac{\begin{array}{ll} a \rightsquigarrow^* a' & \\ b \rightsquigarrow^* b' & a'\ b' \rightsquigarrow^* c \end{array}}{a\ b \rightsquigarrow^* c}$$

where a, a', b, b', and c are all terms. We apply this result by instantiating a, a', b, b', and c with $[t/x]t_1'$, s_1, $[t/x]t_2'$, s_2, and $[s_2/y]^{B'} s_1'$ respectively. Therefore, $[t/x](t_1'\ t_2') \rightsquigarrow^* [s_2/y]^{B'} s_1'$.

Suppose $s_1 \equiv \Delta y : \neg(B' \to B).s_1'$ and t_1' is not a Δ-abstraction. By Lemma 13.2.3.4 there exists a type B'' such that $\mathsf{ctype}_A(x, t_1') = B''$, $B'' \equiv B' \to B$, and B'' is a subexpression of A. Then by the definition of the hereditary substitution function $[t/x]^A(t_1'\ t_2') = \Delta z : \neg B.\langle (y, z, s_2)\rangle_B^{B'} s_1'$, where z is fresh variable. Now

$$\begin{array}{rcl} [t/x](t_1'\ t_2') & = & ([t/x]t_1')\,([t/x]t_2') \\ & \rightsquigarrow^* & s_1\ s_2 \\ & \equiv & (\Delta y : \neg(B' \to B).s_1')\ s_2 \\ & \rightsquigarrow & \Delta z : \neg B.[\lambda y' : B' \to B.(z\,(y'\ s_2))/y]s_1' \\ & = & \Delta z : \neg B.(\langle (y, z, s_2)\rangle^{\uparrow_B^{B'}}\ s_1') \end{array}$$

It suffices to show that $\Delta z : \neg B.(\langle (y, z, s_2)\rangle^{\uparrow_B^{B'}}\ s_1') \rightsquigarrow^* \Delta z : \neg B.\langle (y, z, s_2)\rangle_B^{B'} s_1'$, but this follows from the induction hypothesis, because $(A, 0, t') > (B', 1, s_1')$.

□

Using these properties it is now possible to conclude normalization for the $\lambda\Delta$-calculus.

13.3 Concluding Normalization

We now define the interpretation $[\![T]\!]_\Gamma$ of types T in typing context Γ. This is in fact the same interpretation of types that was used to show normalization using hereditary substitution of the various extensions of SSF.

Definition 13.3.0.1.

The interpretation of types $[\![T]\!]_\Gamma$ *is defined by:*

$$n \in [\![T]\!]_\Gamma \iff \Gamma \vdash n : T$$

We extend this definition to non-normal terms t *in the following way:*

$$t \in [\![T]\!]_\Gamma \iff \exists n.t \rightsquigarrow^! n \in [\![T]\!]_\Gamma$$

Finally, we have the main lemma hereditary substitution for the interpretation of types.

Lemma 13.3.0.2. *[Hereditary Substitution for the Interpretation of Types] If* $n \in [\![T]\!]_\Gamma$ *and* $n' \in [\![T']\!]_{\Gamma,x:T,\Gamma'}$*, then* $[n/x]^T n' \in [\![T']\!]_{\Gamma,\Gamma'}$*.*

Proof. We know by Lemma 13.2.3.6 that there exists a term s such that $[n/x]^T n' = s$ and $\Gamma, \Gamma' \vdash s : T'$, and by Lemma 13.2.3.8 s is normal. Therefore, $s \in [\![T']\!]_{\Gamma,\Gamma'}$. □

Using the previous lemma and the properties of the hereditary substitution function we can now prove type soundness.

Theorem 13.3.0.3. *[Type Soundness] If* $\Gamma \vdash t : T$ *then* $t \in [\![T]\!]_\Gamma$*.*

Proof. This is a proof by induction on the assumed typing derivation.

Case.

$$\frac{}{\Gamma, x : A \vdash x : A} \text{ Ax}$$

Trivial.

Case.

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A.t : A \to B} \text{ Lam}$$

By the induction hypothesis $t \in [\![B]\!]_{\Gamma, x:A}$. By the definition of the interpretation of types $t \rightsquigarrow^! n \in [\![B]\!]_{\Gamma, x:A}$ and $\Gamma, x : A \vdash n : B$. Thus, by applying the λ-abstraction type-checking rule, $\Gamma \vdash \lambda x : A.n : A \to B$, hence by the definition of the interpretation of types $\lambda x : A.n \in [\![A \to B]\!]_\Gamma$. Therefore, $\lambda x : A.t \rightsquigarrow^! \lambda x : A.n \in [\![A \to B]\!]_\Gamma$.

Case.

$$\frac{\Gamma, x : \neg A \vdash t : \bot}{\Gamma \vdash \Delta x : \neg A.t : A} \text{ Delta}$$

Similar to the previous case.

Case.

$$\frac{\Gamma \vdash t_2 : A \quad \Gamma \vdash t_1 : A \to B}{\Gamma \vdash t_1\, t_2 : B} \text{ App}$$

By the induction hypothesis we know $t_1 \in [\![A \to B]\!]_\Gamma$ and $t_2 \in [\![A]\!]_\Gamma$. So by the definition of the interpretation of types we know there exists normal forms n_1 and n_2 such that $t_1 \rightsquigarrow^* n_1 \in [\![A \to B]\!]_\Gamma$ and $t_2 \rightsquigarrow^* n_2 \in [\![A]\!]_\Gamma$. Assume y is a fresh variable in n_1 and n_2 of type A. Then by hereditary substitution for the interpretation of types (Lemma 13.3.0.2) $[n_1/y]^A(y\, n_2) \in [\![B]\!]_\Gamma$. It suffices to show that $t_1\, t_2 \rightsquigarrow^* [n_1/y]^A(y\, n_2)$. This is an easy consequence of soundness with respect to reduction (Lemma 13.2.3.9), that is, $t_1\, t_2 \rightsquigarrow^* n_1\, n_2 = [n_1/y](y\, n_2)$ and by soundness with respect to reduction $[n_1/y](y\, n_2) \rightsquigarrow^* [n_1/y]^A(y\, n_2)$. Therefore, $t_1\, t_2 \in [\![B]\!]_\Gamma$.

□

Finally, we conclude normalization for the $\lambda\Delta$-calculus using hereditary substitution.

Corollary 13.3.0.4. *[Normalization] If $\Gamma \vdash t : T$ then there exists a term n such that $t \rightsquigarrow^! n$.*

13.4 Related Work

We first compare the proof method normalization using hereditary substitution with other known proof methods. The $\lambda\Delta$-calculus could have been proven weakly and strongly normalizing by translation to $\lambda\mu$-calculus. It is true that this is not as complicated as the proof method here, but a proof by translation does not yield a direct proof.

A direct proof of weak and strong normalization could have been given using the Tait-Girard reducibility method. However, we claim that the proof method used

here is less complicated. The statement of the type soundness theorem is qualitatively less complex due to the fact that there is no need to universally quantify over the set of well-formed substitutions. We are able to prove type soundness on open terms directly. Additionally, the formalization of normalization using hereditary substitution does not require recursive types to define the semantics of types which are required when formalizing a proof using reducibility.

R. David and K. Nour give a short proof of normalization of the $\lambda\Delta$-calculus in [46]. There they use a rather complicated lexicographic combination to give a completely arithmetical proof of strong normalization. While they show strong normalization their proof method is comparable to using hereditary substitution. As we mentioned in the introduction hereditary substitution is the constructive content of normalization proofs using the lexicographic combination of an ordering on types and the strict subexpression ordering on terms. It is currently unknown if hereditary substitution can be extended to show strong normalization, but we conjecture that the constructive content of the proof of Lemma 3..6 in David and Nour's work would yield a hereditary substitution like function. Furthermore, for simply typed theories we believe it is enough to show weak normalization and never need to show strong normalization. It is well-known due to the work of G. Barthe et al. in [18] that for the entire left hand side of the λ-cube weak normalization implies strong normalization. We conjecture that this result would extend to the left hand side of the classical λ-cube given in [19]. Thus, showing normalization using hereditary substitution is less complicated than the work of David and Nour's.

Similar to the work of David and Nour is the work of F. Joachimski and R. Matthes. In [71] they prove weak and strong normalization of various simply typed theories. The proof method used is induction on various lexicographic combinations similar to hereditary substitution. After proving weak normalization of each type theory they extract the constructive content of the proof yielding a normalization function which depends on a substitution function similar to the hereditary substitution function. In contrast once hereditary substitution is defined for a type theory we can easily define a normalization function. Note that the following function is the computational content of the type-soundness theorem (Theorem 13.3.0.3).

Definition 13.4.0.1.

We define a normalization function for the $\lambda\Delta$-calculus using hereditary substitution as follows:

$$\mathsf{norm}\ x = x$$

$$\mathsf{norm}\ (\lambda x : A.t) = \lambda x : A.(\mathsf{norm}\ t)$$

$$\mathsf{norm}\ (\Delta x : A.t) = \Delta x : A.(\mathsf{norm}\ t)$$

$$\mathsf{norm}\ (t_1\ t_2) = [n_1/r]^A(r\ n_2)$$

Where $\mathsf{norm}\ t_1 = n_1$, $\mathsf{norm}\ t_2 = n_2$, *A is the type of t_1, and r is fresh in t_1 and t_2.*

This function is similar to the normalization functions in Joachimski and Matthes' work. We could use the above normalization function to decide $\beta\eta$-equality for the $\lambda\Delta$-calculus. Indeed this one of the main application of hereditary substitution.

A. Abel in 2006 shows how to implement a normalizer using sized heteroge-

neous types which is a function similar to the hereditary substitution function in [2]. He then uses hereditary substitution to prove normalization of the type level of a type theory with higher-order subtyping in [4]. This results in a purely syntactic metatheory. C. Keller and T. Altenkirch recently implemented hereditary substitution as a normalization function for the simply typed λ-calculus in Agda [73]. Their results show that hereditary substitution can be used to decide $\beta\eta$-equality. They found hereditary substitution to be convenient to use in a total type theory, because it can be implemented without a termination proof. This is because the hereditary-substitution function can be recognized as structurally recursive, and hence accepted directly by Agda's termination checker.

CONCLUSION

Bugs are the bane of software development, and in order to rid the world of such problems we must revisit the very foundations of programming languages. We have argued that a programming language must be mathematically defined. This allows programmers and language designers the ability to reason about the software being developed in the language. Furthermore, we have argued that the programming languages of the future must contain some notion of a logic which allows for the verification of the programs written in the languages.

We introduced two new dependently-typed functional programming languages that contain a logical fragment called Freedom of Speech and Separation of Proof from Program. We proved logical consistency of the logical fragment of the former. Furthermore, we introduced a new type theory called Dualized Type Theory that shows promise of being a logical foundation of induction and co-induction. All of these new theories constitute foundations of programming languages that directly support verification of the software developed in them.

Finally, we showed how to adapt the hereditary substitution proof technique for showing logical consistency by establishing normalization of several extensions of Stratified System F. We also proved normalization by hereditary substitution of the $\lambda\Delta$-calculus; the first classical type theory to be proved consistent using hereditary substitution. These proofs show that hereditary substitution is a valuable tool that can be used to establish consistency of the logical fragment of various programming

languages.

We believe the future is bright, and that investigations such as the ones in this thesis will lead to new and exciting programming languages. These new languages will have the means to prevent major bugs in safety critical devices from reaching the public, but there is a lot more work to be done.

The following list addresses future work related to each theory:

- The freedom of speech language has a call-by-value operational semantics, but this requires many unfortunate value restrictions. It would be worthwhile to explore the language design when call-by-name is adopted. Some questions to ask are is the language more elegant? That is, do we trade the value restrictions for some other restrictions?

- We only presented the design of the Sep^3, but the analysis still needs to be done. In particular, consistency should be established for the proof fragment, and type safety should be proven for the entire language.

- DTT is a nice start at understanding if bi-intuitionistic logic can be the basis of a logical framework for induction and co-induction, but it is not clear that DTT can be conservatively extended with such features and maintain type safety and consistency. An exploration of these features is an important future goal. Furthermore, the language design containing abstractions from the semantics (abstract Kripke graphs, and worlds on types) is an unfortunate consequence of the failure of cut-elimination due to subtraction, are there alternate designs

that could be used which prevent the need for these abstractions?

- Hereditary substitution still needs to be explored with respect to more expressive type theories. For example, can it be used to prove normalization for a type theory with large eliminations? How about impredicative theories like system F? My conjecture is that it can, but it will require new ideas, because the orderings we have used to prove correctness of the hereditary substitution function are too weak.

REFERENCES

[1] From lambda calculus to cartesian closed categories. *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 376–402, 1980.

[2] A. Abel. Implementing a normalizer using sized heterogeneous types. In *In Workshop on Mathematically Structured Functional Programming, MSFP*, 2006.

[3] A. Abel, B. Pientka, D. Thibodeau, and A. Setzer. Copatterns: programming infinite structures by observations. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL)*, pages 27–38. ACM, 2013.

[4] A. Abel and D. Rodriguez. Syntactic metatheory of higher-order subtyping. In *Proceedings of the 22nd international workshop on Computer Science Logic*, CSL '08, pages 446–460, Berlin, Heidelberg, 2008. Springer-Verlag.

[5] U. Acar, A. Ahmed, and M. Blume. Imperative self-adjusting computation. 2008.

[6] R. Adams. *A Modular Hierarchy of Logical Frameworks*. PhD thesis, 2004.

[7] A. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *Lecture Notes in Computer Science*. ESOP, 2006.

[8] A. Ahmed, D. Dreyer, and A. Rossberg. State-dependent representation independence. *POPL*, 2009.

[9] T. Altenkirch and C. Mcbride. Generic programming within dependently typed programming. In *In Generic Programming, 2003. Proceedings of the IFIP TC2 Working Conference on Generic Programming, Schloss Dagstuhl*, pages 1–20. Kluwer Academic Publishers, 2003.

[10] R. Amadio and P. Curien. *Domains and lambda-calculi*. Cambridge tracts in theoretical computer science. Cambridge University Press, 1998.

[11] J. Andrews and Y. Zhang. General test result checking with log file analysis. *IEEE Trans. Softw. Eng.*, 29:634–648, July 2003.

[12] P. Andrews. Churchś type theory. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Spring 2009 edition, 2009.

[13] A. Appel and D. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.*, 23(5):657–683, 2001.

[14] A. Armstrong, S. Foster, and G. Struth. Dependently typed programming based on automated theorem proving. *CoRR*, abs/1112.3833, 2011.

[15] D. Aspinall and J. Sevcík. Formalising java's data race free guarantee. In *In 20th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2007)*, pages 22–37. Springer, 2007.

[16] S. Awodey. Type theory and homotopy. *Preprint*, 2010.

[17] H. Barendregt. Lambda calculi with types. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, pages 117–309. Oxford University Press, 1992.

[18] G. Barthe, J. Hatcliff, and M. H. SÃrensen. Weak normalization implies strong normalization in a class of non-dependent pure type systems. *Theoretical Computer Science*, 269(1-2):317 – 361, 2001.

[19] G. Barthe, J. Hatcliff, and M. Sørensen. A notion of classical pure type system (preliminary version). *Electronic Notes in Theoretical Computer Science*, 6:4–59, 1997.

[20] G. Bellin. Natural deduction and term assignment for co-heyting algebras in polarized bi-intuitionistic logic. 2004.

[21] G. Bellin. A term assignment for dual intuitionistic logic. *LICS'05-IMLA'05*, 2005.

[22] G. Bellin. Categorical proof theory of co-intuitionistic linear logic. *Submitted to LOMECS*, 2012.

[23] G. Bellin, M. Carrara, D. Chiffi, and A. Menti. A pragmatic dialogic interpretation of bi-intuitionism. *Submitted to Logic and Logical Philosophy*, 2014.

[24] C. Biasi and F. Aschieri. A term assignment for polarized bi-intuitionistic logic and its strong normalization. *Fundam. Inf.*, 84(2):185–205, April 2008.

[25] blogs.consumerreports.org. Consumer reports cars blog: Japan investigates reports of prius brack problem, 2010.

[26] E. Brady. Idris —: systems programming meets full dependent types. In *Proceedings of the 5th ACM workshop on Programming languages meets program verification*, PLPV '11, pages 43–54, New York, NY, USA, 2011. ACM.

[27] N. De Bruijn. The mathematical language automath, its usage, and some of its extensions. In M. Laudet, D. Lacombe, L. Nolin, and M. Schtzenberger, editors, *Symposium on Automatic Demonstration*, volume 125 of *Lecture Notes in Mathematics*, pages 29–61. Springer Berlin / Heidelberg, 1970. 10.1007/BFb0060623.

[28] L. Cardelli. A polymorphic lambda-calculus with type:type. Technical report, 1986.

[29] F. Cardone and R. Hindley. History of lambda-calculus and combinatory logic. 2006.

[30] C. Casinghino, V. Sjöberg, and S. Weirich. Step-indexed normalization for a language with general recursion. In *MSFP '12*, 2012.

[31] C. Casinghino, V. Sjöberg, and S. Weirich. Combining proofs and programs in a dependently typed langauge. In *POPL '14*, 2014.

[32] A. Church. A set of postulates for the foundation of logic. *The Annals of Mathematics*, 34(4):pp. 839–864, 1933.

[33] A. Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5(2):pp. 56–68, 1940.

[34] R. Constable, S. Allen, S. Allen, H. Bromley, W. Cleaveland, J. Cremer, R. Harper, H. Douglas, T. Knoblock, N. Mendler, P. Panangaden, S. Smith, J. Sasaki, and S. Smith. *Implementing Mathematics with The Nuprl Proof Development System*. Prentice Hall, 1986.

[35] T. Coquand. An analysis of girard's paradox. In *In Symposium on Logic in Computer Science*, pages 227–236. IEEE Computer Society Press, 1986.

[36] T. Coquand. A new paradox in type theory. In *Logic, Methodology and Philosophy of Science IX : Proceedings of the Ninth International Congress of Logic, Methodology, and Philosophy of Science*, pages 7–14. Elsevier, 1994.

[37] T. Coquand. Type theory. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Spring 2010 edition, 2010.

[38] T. Coquand and G. Huet. The calculus of constructions. *Inf. Comput.*, 76(2-3):95–120, 1988.

[39] P. Cousot. The verification grand challenge and abstract interpretation. In B. Meyer and J. Woodcock, editors, *Verified Software: Theories, Tools, Experiments*, volume 4171 of *Lecture Notes in Computer Science*, pages 227–240. Springer, Berlin, Germany, December 2007.

[40] T. Crolard. Subtractive logic. *Theor. Comput. Sci.*, 254(1-2):151–185, 2001.

[41] T. Crolard. A formulae-as-types interpretation of subtractive logic. *J. Log. and Comput.*, 14(4):529–570, August 2004.

[42] R. Crole. *Categories for Types*. Cambridge Mathematical Textbooks. Cambridge University Press, 1994.

[43] P. Curien. Abstract machines, control, and sequents. In G. Barthe, P. Dybjer, L. Pinto, and J. Saraiva, editors, *Applied Semantics*, volume 2395 of *Lecture Notes in Computer Science*, pages 123–136. Springer Berlin Heidelberg, 2002.

[44] P. Curien and H. Herbelin. The duality of computation. In *Proceedings of the fifth ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 233–243. ACM, 2000.

[45] N. Danner and D. Leivant. Stratified polymorphism and primitive recursion. *Mathematical. Structures in Comp. Sci.*, 9(4):507–522, 1999.

[46] R. David and K. Nour. A short proof of the strong normalization of the simply typed lambdamu-calculus. *SCHEDAE INFORMATICAE*, 12:27–33, 2003.

[47] D. Dreyer, A. Ahmed, and L. Birkedal. Logical step-indexed logical relations. *Logical Methods in Computer Science*, 7(2), 2011.

[48] P. Dybjer. Representing inductively defined sets by wellorderings in martin-löf's type theory. *Theoretical Computer Science*, 176(1-2):329, 1997.

[49] H. Eades. The semantics of advanced programming languages. Full version, available at http://metatheorem.org/wp-content/papers/thesis-full.pdf, 2014.

[50] H. Eades and A. Stump. Hereditary substitution for stratified system f. In *Proof-Search in Type Theories (PSTT)*, 2010.

[51] H. Eades and A. Stump. Using the hereditary substitution function in normalization proofs, 2011.

[52] H. Eades and A. Stump. Hereditary Substitution for the $\lambda\Delta$-Calculus. *ArXiv e-prints*, September 2013.

[53] P. Martin-Löf. Cohen et al., editor. *Constructive mathematics and computer programming*, volume 1, North-Holland, Amsterdam., 1982.

[54] S. Feferman. Predicativity. In S. Shapiro, editor, *The Oxford Handbook of Philosophy of Mathematics and Logic*, pages 590–624. Oxford University Press, 2005.

[55] A. Filinski. Declarative continuations: An investigation of duality in programming language semantics. In D. Pitt, D. Rydeheard, P. Dybjer, A. Pitts, and A. Poign, editors, *Category Theory and Computer Science*, volume 389 of *Lecture Notes in Computer Science*, pages 224–249. Springer Berlin Heidelberg, 1989.

[56] S. Fogarty, E. Pasalic, J. Siek, and W. Taha. Concoqtion: indexed types now! In *Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, PEPM '07, pages 112–121, New York, NY, USA, 2007. ACM.

[57] H. Geuvers, R. Krebbers, and J. McKinna. The lambda-mu-t-calculus. *Annals of Pure and Applied Logic*, 2012.

[58] H. Geuvers and G. Nijmegen. Proof assistants: history, ideas and future, Feburary 2009.

[59] J. Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1971.

[60] J. Girard, Y. Lafont, and P. Taylor. *Proofs and Types (Cambridge Tracts in Theoretical Computer Science)*. Cambridge University Press, April 1989.

[61] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge University Press, 1990.

[62] G. Gonthier. A computer-checked proof of the four colour theorem. 2005.

[63] R. Goré, Linda Postniece, and Alwen Tiu. Cut-elimination and proof-search for bi-intuitionistic logic using nested sequents. In C. Areces and R. Goldblatt, editors, *Advances in Modal Logic*, pages 43–66. College Publications, 2008.

[64] T. Griffin. A formulae-as-types notion of control. In *In Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 47–58. ACM Press, 1990.

[65] C. Gunter. *Semantics of Programming Languages: Structures and Techniques*. Foundations of Computing. MIT Press, 1992.

[66] J. Heijenoort. *From Frege to Gödel: a source book in mathematical logic, 1879-1931*. Source books in the history of the sciences. Harvard University Press, 1967.

[67] J. Hintikka. *From Dedekind to Gödel: essays on the development of the foundations of mathematics*. Synthese library. Kluwer Academic Publishers, 1995.

[68] H. Hofmann. *Extensional Concepts in Intensional Type Theory*. Thesis. University of Edinburgh, Department of Computer Science, 1995.

[69] M. Hofmann and T. Streicher. The Groupoid Model Refutes Uniqueness of Identity Proofs. In *Ninth Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 208–212. IEEE Computer Society, 1994.

[70] M. Hofmann and T. Streicher. The groupoid interpretation of type theory. In *Twenty-five years of constructive type theory (Venice, 1995)*, volume 36 of *Oxford Logic Guides*, pages 83–111. Oxford Univ. Press, New York, 1998.

[71] F. Joachimski and R. Matthes. Short proofs of normalization for the simply-typed lambda-calculus, permutative conversions and gödel's t, 1999.

[72] S. Peyton Jones, D. Vytiniotiss, S. Weirich, and G. Washburn. Simple unification-based type inference for gadts. In *Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, ICFP '06, pages 50–61, New York, NY, USA, 2006. ACM.

[73] C. Keller and T. Altenkirch. Hereditary substitutions for simple types, formalized. In *Proceedings of the third ACM SIGPLAN workshop on Mathematically structured functional programming*, MSFP '10, pages 3–10, New York, NY, USA, 2010. ACM.

[74] G. Kimmell, A. Stump, H. Eades, P. Fu, T. Sheard, S. Weirich, C. Casinghino, V. Sjoberg, N. Collins, and K. Y. Ahn. Equational reasoning about programs with general recursion and call-by-value semantics. In *PLPV*, 2012.

[75] D. Kimura and M. Tatsuta. Dual Calculus with Inductive and Coinductive Types. In R. Treinen, editor, *Rewriting Techniques and Applications (RTA)*, pages 224–238, 2009.

[76] S. Kleene and J. Rosser. The inconsistency of certain formal logics. *The Annals of Mathematics*, 36(3):pp. 630–636, 1935.

[77] C. Klein, J. Clements, C. Dimoulas, C. Eastlund, M. Felleisen, M. Flatt, J. McCarthy, J. Rafkind, S. Tobin-Hochstadt, and R. Findler. Run your research: on the effectiveness of lightweight mechanization. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '12, pages 285–296, New York, NY, USA, 2012. ACM.

[78] G. Klein and T. Nipkow. Verified bytecode verifiers. *Theor. Comput. Sci.*, 298:583–626, April 2003.

[79] J. Krivine. Realizability in classical logic. *Panoramas et synthèses*, 27:197–229, 2009. Interactive models of computation and program behaviour. Société Mathématique de France.

[80] J. Lambek. *Substructural Logics*, volume 2, chapter From Categorical Grammar to Bilinear Logic. Oxford Science Publications, 1993.

[81] J. Lambek. Cut elimination for classical bilinear logic. *Fundam. Inform.*, 22(1/2):53–67, 1995.

[82] F. Lawvere and S. Schanuel. *Conceptual Mathematics: A First Introduction to Categories*. Conceptual Mathematics: A First Introduction to Categories. Cambridge University Press, 2009.

[83] D. Leivant. Finitely stratified polymorphism. *Inf. Comput.*, 93(1):93–113, 1991.

[84] X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '06, pages 42–54, New York, NY, USA, 2006. ACM.

[85] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[86] J. Lévy. An algebraic interpretation of the $\lambda\beta\kappa$-calculus; and an application of a labelled λ-calculus. *Theoretical Computer Science*, 2(1):97 – 114, 1976.

[87] D. Licata and R. Harper. A formulation of Dependent ML with explicit equality proofs. Technical Report CMU-CS-05-178, Carnegie Mellon University Department of Computer Science, 2005.

[88] Nathan M.-L. and T. Sheard. Erasure and polymorphism in pure type systems. In *Proceedings of the Theory and practice of software, 11th international conference on Foundations of software science and computational structures*, FOSSACS'08/ETAPS'08, pages 350–364, H. Berlin, 2008. Springer-Verlag.

[89] P. Martin-Löf and G.Sambin. *Intuitionistic type theory*. Studies in proof theory. Bibliopolis, 1984.

[90] C. McBride and J. McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.

[91] E. Mendelson. *Introduction to Mathematical Logic*. Chapman & Hall/CRC, 5th edition, 2009.

[92] G. Mints. *A short introduction to intuitionistic logic*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[93] C. Murthy. Classical proofs as programs: How, what and why. Technical report, Cornell University, 1991.

[94] G. Neis, D. Dreyer, and A. Rossberg. Non-parametric parametricity. *ICFP*, 2009.

[95] T. Nipkow. Higher-Order Critical Pairs. *Proceedings of Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 342–349, 1991.

[96] B. Nordström, K. Petersson, and J. Smith. *Programming in Martin-Löf's Type Theory*. Oxford University Press, 1990.

[97] B. Nordstrom, K. Petersson, and J. Smith. *Programming in Martin-Löf's Type Theory: An Introduction*. Oxford University Press, USA, July 1990.

[98] U. Norell. Towards a practical programming language based on dependent type theory. PhD Thesis, 2007.

[99] The German Federal Bureau of Aircraft Accidents. Investigation report, 2004.

[100] M. Parigot. Free deduction: An analysis of "computations" in classical logic. In A. Voronkov, editor, *Logic Programming*, volume 592 of *Lecture Notes in Computer Science*, pages 361–380. Springer Berlin / Heidelberg, 1992.

[101] M. Parigot. Lambda-mu-calculus: An algorithmic interpretation of classical natural deduction. In A. Voronkov, editor, *Logic Programming and Automated Reasoning*, volume 624 of *Lecture Notes in Computer Science*, pages 190–201. Springer Berlin / Heidelberg, 1992. 10.1007/BFb0013061.

[102] F. Pfenning and C. Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In *Automated Deduction — CADE-16*, volume 1632 of *Lecture Notes in Computer Science*, pages 679–679. Springer Berlin / Heidelberg, 1999.

[103] B. Pierce. *Basic Category Theory for Computer Scientists*. Foundations of Computing. MIT Press, 1991.

[104] B. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[105] B. Pierce. *Advanced Topics in Types and Programming Languages*. The MIT Press, 2004.

[106] L. Pinto and T. Uustalu. Proof search and counter-model construction for bi-intuitionistic propositional logic with labelled sequents. In M. Giese and A. Waaler, editors, *Automated Reasoning with Analytic Tableaux and Related Methods*, volume 5607 of *Lecture Notes in Computer Science*, pages 295–309. Springer Berlin Heidelberg, 2009.

[107] A. Pitts. Existential types: Logical relations and operational equivalence. In K. Larsen, S. Skyum, and G. Winskel, editors, *Automata, Languages and Programming*, volume 1443 of *Lecture Notes in Computer Science*, pages 309–326. Springer Berlin / Heidelberg, 1998. 10.1007/BFb0055063.

[108] A. Platzer and C. Edmund. Formal verification of curved flight collision avoidance maneuvers: A case study. In A. Cavalcanti and D. Dams, editors, *FM*, volume 5850 of *LNCS*, pages 547–562. Springer, 2009.

[109] G. Plotkin. Lambda-definability and logical relations. *University of Edinburgh School of Artificial Intelligence Memorandum SAI-RM-4*, 1973.

[110] D. Prawitz. *Natural Deduction: A Proof-Theoretical Study*. Dover Publications, 1965.

[111] D. Prawitz. *Logical Consequence from a Constructivist Point of View*, volume 1, pages 671–695. Oxford University Press, 2005.

[112] C. Rauszer. Semi-boolean algebras and their applications to intuitionistic logic with dual operations,. *Fundamenta Mathematicae*, 83:219–249, 1974.

[113] J. Rehof and M. Sørensen. The Lambda-Delta-calculus. In *Proceedings of the International Conference on Theoretical Aspects of Computer Software (TACS)*, pages 516–542, 1994.

[114] Research Triangle Institute. The Economic Impacts of Inadequate Infrastructure for Software Testing, 2002. Sponsored by the Department of Commerce's National Institute of Standards and Technology.

[115] Reuters. Toyota to recall 436,00 hybrids globally-document, February 2010.

[116] J. Reynolds. Towards a theory of type structure. In *Colloque sur la Programmation*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425, New York, 1974. Springer-Verlag.

[117] J. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998.

[118] J. C. Reynolds. Programming methodology. chapter What Do Types Mean?: From Intrinsic to Extrinsic Semantics, pages 309–327. Springer-Verlag New York, Inc., New York, NY, USA, 2003.

[119] H. Schellinx. Some syntactical observations on linear logic. *Journal of Logic and Computation*, 1(4):537–559, 1991.

[120] P. Selinger. Control categories and duality: on the categorical semantics of the lambda-mu calculus. *Mathematical Structures in Computer Science*, 11(02):207–260, 2001.

[121] P. Sewell, F. Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strnisa. Ott: Effective tool support for the working semanticist. In *Journal of Functional Programming*, volume 20, pages 71–122, 2010.

[122] C. Shan. Higher-order modules in system fω and haskell. http://www.cs.rutgers.edu/ ccshan/xlate/xlate.pdf, 2006. Online; accessed 08-16-13.

[123] T. Sheard. Type-Level Computation Using Narrowing in Ωmega. In *Programming Languages meets Program Verification*, volume 1643, 2006.

[124] V. Sjoberg, C. Casinghino, K. Y. Ahn, N. Collins, H. Eades, P. Fu, G. Kimmell, T. Sheard, A. Stump, and S. Weirich. Irrelevance, heterogeneous equality, and call-by-value dependent type systems. In J. Chapman and P. B. Levy, editors, Proceedings Fourth Workshop on *Mathematically Structured Functional Programming,* Tallinn, Estonia, 25 March 2012, volume 76 of *Electronic Proceedings in Theoretical Computer Science*, pages 112–162. Open Publishing Association, 2012.

[125] M. Sorensen. Strong normalization from weak normalization in typed lambda-calculi. *Information and Computation*, 133:35–71, 1997.

[126] M. Sørensen and P. Urzyczyn. *Lectures on the Curry-Howard Isomorphism*. Number v. 10 in Studies in Logic and the Foundations of Mathematics. Elsevier Science, 2006.

[127] R. Statman. Logical relations and the typed lambda-calculus. *Information and Control*, 65(23):85 – 97, 1985.

[128] T. Streicher. *Semantics of Type Theory: Correctness, Completeness, and Independence Results*. Progress in theoretical computer science. Birkhäuser, 1991.

[129] T. Streicher. *Investigations Into Intensional Type Theory*. PhD thesis, Habilitation- sschrift, Ludwig-Maximilians-Universität Müchen, November 1993.

[130] A. Stump, M. Deters, A. Petcher, T. Schiller, and T. Simpson. Verified programming in guru. In *Proceedings of the 3rd workshop on Programming languages meets program verification*, PLPV '09, pages 49–58, New York, NY, USA, 2008. ACM.

[131] A. Stump, M. Deters, A. Petcher, T. Schiller, and T. Simpson. Verified Programming in Guru. In T. Altenkirch and T. Millstein, editors, *Programming Languges meets Program Verification (PLPV)*, 2009.

[132] E. Sumii and B. Pierce. Logical relation for encryption. *J. Comput. Secur.*, 11(4):521–554, July 2003.

[133] M. Tatsuta and G. Mints. A simple proof of second-order strong normalization with permutative conversions. *Annals of Pure and Applied Logic*, 136:134–155, 2005.

[134] The Coq Development Team. The coq proof assistant reference manual, 2008.

[135] thedetroitbureau.com. Nhtsa memo on regenerative braking, April 2011.

[136] A. Troelstra. *History of constructivism in the 20th century*. ITLI Prepublication Series ML-91-05, 1991.

[137] United States Federal Bureau of Investigation. 2005 FBI Computer Crime Survey.

[138] D. Vytiniotis and V. Koutavas. Relating step-indexed logical relations and bisimulations. Technical Report MSR-TR-2009-25, Microsoft Research, March 2009.

[139] D. Vytiniotis and S. Weirich. Dependent types: Easy as PIE. In M. T. Morazán and H. Nilsson, editors, *Draft Proceedings of the 8th Symposium on Trends in Functional Programming*, pages XVII–1—XVII–15. Dept. of Math and Computer Science, Seton Hall University, April 2007. TR-SHU-CS-2007-04-1.

[140] P. Wadler. Call-by-value is dual to call-by-name. *SIGPLAN Not.*, 38(9):189–201, August 2003.

[141] P. Wadler. Call-by-value is dual to call-by-name – reloaded. In J. Giesl, editor, *Term Rewriting and Applications*, volume 3467 of *Lecture Notes in Computer Science*, pages 185–203. Springer Berlin Heidelberg, 2005.

[142] P. Wadler. The girard,reynolds isomorphism (second edition). *Theoretical Computer Science*, 375(1-3):201 – 226, 2007. Festschrift for John C. Reynolds 70th birthday.

[143] K. Watkins, I. Cervesato, F. Pfenning, and D. Walker. A concurrent logical framework: The propositional fragment. In S. Berardi, M. Coppo, and F. Damiani, editors, *Types for Proofs and Programs*, volume 3085 of *Lecture Notes in Computer Science*, pages 355–377. Springer Berlin / Heidelberg, 2004.

[144] H. Weyl. Das kontinuum. Translated as Weyl 1994, 1918.

[145] W.Howard. The formulae-as-types notion of construction. 1969-1980.

[146] H. Xi. Dead code elimination through dependent types. In *The First International Workshop on Practical Aspects of Declarative Languages*, pages 228–242, San Antonio, January 1999. Springer-Verlag LNCS vol. 1551.

[147] H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '03, pages 224–235, New York, NY, USA, 2003. ACM.

[148] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 249–257, Montreal, June 1998.

[149] H. Xi and F. Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 214–227, San Antonio, January 1999.

[150] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using model checking to find serious file system errors. *ACM Trans. Comput. Syst.*, 24:393–423, November 2006.

[151] C. Zenger. Indexed types. *Theoretical Computer Science*, 187(1Äì2):147 – 165, 1997.