THE SEMANTIC ANALYSIS OF ADVANCED PROGRAMMING LANGUAGES

by

Harley Daniel 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

June 30

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 Daniel Eades III

has been approved by the Examining Committee for the thesis requirement for the Doctor of Philosophy degree in Computer Science at the June 30 graduation.

Thesis Committee: _____
Aaron Stump, Thesis Supervisor

_____
Cesare Tinelli

_____
Stephanie Weirich

_____
Gregory Landini

_____
Kasturi Varadarajan

TABLE OF CONTENTS

0.1 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 [100, 123].

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 [18]. 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 [120]. 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 [101]. 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 of-

ficially ruled by the German Federal Bureau of Aircraft Accidents Investigation that the on-board collision detection was indeed faulty [87].

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 [65] 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 plans 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 plans to collide if the systems are operational [96]. 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 article has been formally defined in a tool called Ott [106]. The full Ott specification of every type theory defined with in this article can be found in the appendix. Ott is a tool for writing definitions of programming languages, type theories, and λ-calculi. 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 article. It clearly stands as a successful step towards the solution of the second major problem outlined above.

Part I

Background

Figure 0.2.0.1. Time line of the History of Type Theory

0.2 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 days of Bertrand Russell and Alonzo Church. They are as we consider them the founding fathers of type theory. We then move to discuss more modern type theories. The time line depicted in Fig. 0.2.0.1 gives a summary of the most significant breakthroughs in type theory. The history given here follows this time line exactly. 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.

0.2.1 The Early Days of Type Theory (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 [57, 55], 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 gives a nice and simple definition of the simple type theory in [77] and we summarize this in the following definition.

Definition 0.2.1.1. *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. 0.2.1.1. 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 paper 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 [27, 23] [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 [26]. 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 Appendix **??** 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 [64]. 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 [27, 11]. 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,

[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: Thierry Coquand's Calculus of Constructions, Per Martin-Löf's Type Theory, Michel Parigot's $\lambda\mu$-Calculus, and Philip Wadler's Dual Calculus.

and Church. Some really great references on this early history and more can be found in [23, 31, 16]. We now move on to modern type theory where we will cover a large part of type theory as it stands today.

0.2.2 Modern Type Theory (1961 - Present)

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 [50, 16, 27], the second is called Curry style [16, 103], and the third is in the form of a pure type system. We introduce pure type systems in Sec. 0.5.2. 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 specifying 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 0.2.2.2 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 0.2.2.2 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

[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

Syntax:

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

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

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

Figure 0.2.2.3. Type-checking algorithm for the Church-style simply typed λ-calculus

of inference rules. The typing judgment for STLC is defined in Figure 0.2.2.3. The typing judgment depends on a typing context Γ which for now can be considered as a list 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

---

when deriving conclusions from the inference rules deriving the judgment there is always a deterministic choice on how to proceed.

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 0.2.2.2. *We apply each rule starting with its conclusion:*

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

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. The syntax and reduction relation of this theory is defined in Fig. 0.2.2.4 and the typing judgment is defined in Fig. 0.2.2.5. 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:

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

Recall that these rules are read bottom up. Up until now we have considered the typing judgment as simply a checking procedure with the type as one of the inputs,

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:

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

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

Figure 0.2.2.4. Syntax and reduction rules for the Curry-style simply-typed λ-calculus

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 [16].

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 would arise. The first of these is Gödel's system T. In this theory Gödel extends

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

Figure 0.2.2.5. Type-checking algorithm for the Curry-style simply typed λ-calculus

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:

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

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

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

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

$$\frac{t_2 \rightsquigarrow t_2'}{t_1\ t_2 \rightsquigarrow t_1\ t_2'}\ \text{R\_App2} \qquad \frac{t \rightsquigarrow t'}{\mathsf{S}\ t \rightsquigarrow \mathsf{S}\ t'}\ \text{R\_Succ}$$

Figure 0.2.2.6. Syntax and reduction rules for Gödel's system T

STLC with natural numbers and primitive recursion. In [50] 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 Fig. 0.2.2.6 and the type-checking relation is defined in Fig. 0.2.2.7.

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

$$\frac{}{\Gamma, x : T, \Gamma' \vdash x : T}\ \textsc{Var} \qquad \frac{}{\Gamma \vdash 0 : \mathsf{Nat}}\ \textsc{Zero}$$

$$\frac{}{\Gamma \vdash \mathsf{S} : \mathsf{Nat} \to \mathsf{Nat}}\ \textsc{Succ} \qquad \frac{}{\Gamma \vdash \mathsf{R} : T \to ((T \to (\mathsf{Nat} \to T)) \to (\mathsf{Nat} \to T))}\ \textsc{Rec}$$

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

Figure 0.2.2.7. Type-checking algorithm for the Gödel's system T

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:

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

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

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

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

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

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 [50].

The authors of [50] consider system T to be a step forward computationally, but a step backward logically. We will see in Sect. 0.3 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 0.2.2.3. *Some interesting functions in system T:*

Addition:	$\mathbf{add}\, x\, y := \lambda x : \mathsf{Nat}.(\lambda y : \mathsf{Nat}.(\mathsf{R}\, x\, (\lambda z : \mathsf{Nat}.(\lambda w : \mathsf{Nat}.(\mathsf{S}\, z)))\, y))$
Multiplication:	$\mathbf{mult}\, x\, y := \lambda x : \mathsf{Nat}.(\lambda y : \mathsf{Nat}.(\mathsf{R}\, 0\, (\lambda z : \mathsf{Nat}.(\lambda w : \mathsf{Nat}.(\mathbf{add}\, x\, z)))\, y))$
Exponentiation:	$\mathbf{exp}\, x\, y := \lambda x : \mathsf{Nat}.(\lambda y : \mathsf{Nat}.(\mathsf{R}\, (\mathsf{S}\, 0)\, (\lambda z : \mathsf{Nat}.(\lambda w : \mathsf{Nat}.(\mathbf{exp}\, x\, z)))\, y))$
Predecessor:	$\mathbf{pred}\, x := \lambda x : \mathsf{Nat}.(\mathsf{R}\, 0\, (\lambda z : \mathsf{Nat}.(\lambda w : \mathsf{Nat}.w))\, x)$

Example 0.2.2.4. *We give an example reduction of addition. We use numeral syntax for natural numbers, but this should be read as syntactic sugar. That is,* $1 \equiv \mathsf{S}\, 0$, $2 \equiv \mathsf{S}\, (\mathsf{S}\, 0)$, *etc.*

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

Notice that the example reduction given in Ex. 0.2.2.4 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 [50]. The proof is similar to how we show strong normalization for STLC in Sect. 0.7.3. 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 [49, 50, 16]. 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 [128]. 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 [102, 103]. Reynolds being in the field of programming languages was investigating polymorphism. That is the ability to define universal (or generic[3]) 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 [128].

The syntax for terms, types, and the reduction rules are defined in Fig. 0.2.2.8 and the definition of the typing relation is defined in Fig. 0.2.2.9. 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

[3] Throughout this article 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 0.2.2.8. Syntax and reduction rules for system F

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

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

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

Figure 0.2.2.9. Type-checking algorithm 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 0.2.2.5. *Example functions in system F:*

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

Note that in the previous example we used the definition

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

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 [50]. 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 $n\,[X]\,f\,v \equiv f^n v$, 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 Sect. 0.7. 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 where used to organize the types of his theory. This organization made it impossible to instantiate a type with itself.

Syntax:

$$\begin{array}{lcl} K & ::= & 1 \mid 2 \mid \ldots \\ T & ::= & X \mid T \to T' \mid \forall X : K.T \\ t & ::= & x \mid \lambda x : T.t \mid \Lambda X : K.t \mid t_1\ t_2 \mid t[T] \end{array}$$

Full β-reduction:

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

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

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

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

Figure 0.2.2.10. Syntax and reduction rules for SSF

Predicative systems are less expressive than impredicative systems [68]. This means that there are functions definable in an impredicative theory which are not definable in its predicative version. In [68, 37] 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 Sec. 0.7.1. The syntax and reduction rules for SSF are defined in Fig. 0.2.2.10, kinding rules in Fig. 0.2.2.11, and typing rules in Fig. 0.2.2.12. The objective of SSF is to enforce the property of 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

$$\frac{}{\Gamma, X : K, \Gamma' \vdash X : K}\ \textsc{K\_Var} \qquad \frac{\Gamma \vdash T_1 : K \quad \Gamma \vdash T_2 : K'}{\Gamma \vdash T_1 \to T_2 : \mathsf{max}(K, K')}\ \textsc{K\_Arrow}$$

$$\frac{\Gamma, X : K \vdash T : K'}{\Gamma \vdash \forall X : K . T : \mathsf{max}(K + 1, K')}\ \textsc{K\_Forall}$$

Figure 0.2.2.11. Kind-checking rules for the SSF

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

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

$$\frac{\Gamma \vdash t : \forall X : K . T' \quad \Gamma \vdash T : K}{\Gamma \vdash t[T] : [T/X]T'}\ \textsc{TypeApp}$$

Figure 0.2.2.12. Type-checking rules for the SSF

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

$$\frac{\Gamma, X : K \vdash T : K'}{\Gamma \vdash \forall X : K.T : \mathsf{max}(K+1, K')} \quad \text{K\_FORALL.}$$

This is the rule which enforces predicativity. It does this by making sure the level of $\forall X : K.T$ 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 $K < max(K+1, K')$ for all K and K'. Hence, resulting in the enforcement of our desired property.

A understandable question one could ask at this point is, are predicative theories enough? 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 [44]. In fact Hermann Weyl proposed a predicativist foundation of mathematics. In his book [130] 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.

Open Problem. 2.2.5. *It seems as if there are degrees of impredicativity. For example, system F has a weaker form of impredicativity, because no paradoxes exist in system F, this follows from the fact that we know it is consistent, but there are other impredicative systems which do contain paradoxes. In fact we will see such a system in Sect. 0.5. How many degrees of impredicativity are there and how much is too much?*

I do not know of any research investigating the previous open problem, but it seems to me that it is an important question.

System $\mathbf{F}^{\omega}$. In Girard's thesis [49] 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 Fig. 0.2.2.13, the kinding rules in Fig. 0.2.2.14, and the typing rules in Fig. 0.2.2.15. 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

Syntax:

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

Full β-reduction:

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

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

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

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

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

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

Figure 0.2.2.13. Syntax and reduction rules for system F^ω

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

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

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

Figure 0.2.2.14. Kinding rules of system F^ω

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

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

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

Figure 0.2.2.15. Type-checking algorithm for the system F^ω

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 [107]. 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 0.5. Using dependent types and type-level computation we could amongst other things define and use only a single copy of the natural numbers.

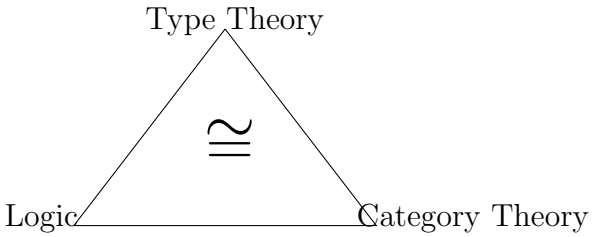
Logically, through the three perspectives of computation (see Section 0.3) sys-

tem 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 types 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.

0.3 The Three Perspectives of Computation

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. Calling this beautiful relationship the three perspectives of computation 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 three perspectives of computation. 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.

0.3.1 Type Theory and Logic

Intuitionism began with Luitzen Brouwer. Implicit in his work was an interpretation of the formulas of propositional and predicate intuitionistic logic as computational 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. 0.3.1.1 defines the assignment of proof terms using these constructs to formulas of intuitionistic propositional logic.

Definition 0.3.1.1. *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 0.2 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 [131] 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 [52, 131, 78, 81, 112, 121]. 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 [128, 49]. We will see other advances to this correspondence with logic in Section 0.5 where we discuss dependent types.

There is one requirement a type theory must meet in order for it to correspond to a logic. We know from logic that proofs must be finite. So if computational constructs such as objects of type theory are to be considered proofs then they 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 [97]. It is widely known that showing a type theory to be consistent – through the remainder of the paper 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 0.7.

0.3.2 Type Theory and 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 0.3.2.2. *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:

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

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

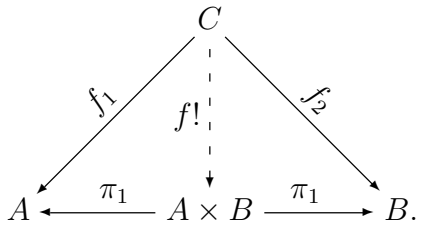
In order to interpret STLC we will need a category with some special features. The first of these is the final object.

Definition 0.3.2.3. *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.

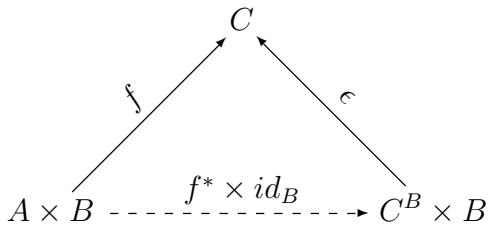
Definition 0.3.2.4. *An object of a category* $\mathcal{C}$ *denoted* $A \times B$ *is called a binary*

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 0.3.2.5. *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 0.3.2.6. *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 article. The interested reader should see [34, 53, 67, 92] for excellent introductions to the subject.

We now have everything needed to interpret STLC as a category. Our interpretation follows that of [53]. 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 0.3.2.7. *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:

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

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

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

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.

0.3.3 The Impact of the Three Perspectives of Computation on Programming Languages

The reader may now be wondering what the benefits are of the three perspectives of computation if there are any at all. The three perspectives of computation are all just that. They provide a unique angle on computation. To paraphrase [139] 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[4] .

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

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

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 allowed 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 [112].

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.

0.4 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?

0.4.1 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 Michel Parigot constructed the $\lambda\mu$-calculus in 1992 [89]. 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 [88]. This allowed for Parigot to define a computational perspective of free deduction which he called the $\lambda\mu$-calculus. We now briefly introduce the $\lambda\mu$-calculus. The syntax and reduction rules are in Fig. 0.4.1.16. We can think of the language of the $\lambda\mu$-calculus as an extension of the λ-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.

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

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 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 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.

$$\frac{}{x : \Gamma, A^x \vdash A, \Delta}\ \textsc{Var} \qquad \frac{t : \Gamma, A^x \vdash B, \Delta}{\lambda x.t : \Gamma \vdash A \to B, \Delta}\ \textsc{Lam}$$

$$\frac{s : \Gamma \vdash A^\alpha, \Delta}{\mu\alpha.s : \Gamma \vdash A, \Delta}\ \textsc{Mu} \qquad \frac{\begin{array}{c} t_2 : \Gamma' \vdash A, \Delta' \\ t_1 : \Gamma \vdash A \to B, \Delta \end{array}}{t_1\ t_2 : \Gamma, \Gamma' \vdash B, \Delta, \Delta'}\ \textsc{App}$$

$$\frac{t : \Gamma \vdash A, \Delta}{[\alpha]t : \Gamma \vdash A^\alpha, \Delta}\ \textsc{NameApp}$$

Figure 0.4.1.17. Type-checking algorithm for the $\lambda\mu$-calculus

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 Fig. 0.4.1.17. 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 [5] Think of $e : \Gamma \vdash \Delta$ as e being a witness[6] 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 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

[5] This is not the only formalization we could have used. See [35] for another example which is closer to the style we have been using for the earlier type theories.

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

$\neg A =^{def} A \rightarrow \bot$. 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 0.4.1.1. *The following rules are admissible w.r.t. the $\lambda\mu$-calculus:*

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

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 0.4.1.2. *The following rules are derivable using the typing rules and the rules of Lemma 0.4.1.1:*

$$\frac{t : \Gamma, A^x \vdash \bot, \Delta}{\lambda x.t : \Gamma \vdash \neg A, \Delta}\ \textsc{NegInt1} \qquad \frac{t_1 : \Gamma \vdash \neg A, \Delta \qquad t_2 : \Gamma \vdash A, \Delta}{t_1\ t_2 : \Gamma \vdash \bot, \Delta}\ \textsc{NegElim1}$$

$$\frac{\alpha \textit{ fresh in } \Delta \qquad s : \Gamma, A^x \vdash \Delta}{\lambda x.\mu\alpha.s : \Gamma \vdash \neg A, \Delta}\ \textsc{NegInt2} \qquad \frac{t_1 : \Gamma \vdash \neg A, \Delta \qquad t_2 : \Gamma \vdash A, \Delta}{t_1\ t_2 : \Gamma \vdash \Delta}\ \textsc{NegElim2}$$

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

Example 0.4.1.3. *The proof of $\neg\neg A \to A$ is as follows:*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{y : \neg\neg A^y \vdash \neg\neg A, A^\alpha}\ \textsc{Var}}{x : \neg\neg A^y, A^x \vdash A}\ \textsc{Var}}{[\alpha]x : \neg\neg A^y, A^x \vdash A^\alpha}\ \textsc{NameApp}}{\lambda x.\mu\beta.[\alpha]x : \neg\neg A^y \vdash \neg A, A^\alpha}\ \textsc{NegInt2}}{y\,(\lambda x.\mu\beta.[\alpha]x) : \neg\neg A^y \vdash \bot, A^\alpha}\ \textsc{NegElim1}}{[\beta'](y\,(\lambda x.\mu\beta.[\alpha]x)) : \neg\neg A^y \vdash A^\alpha}\ \textsc{BtmElim}}{\mu\alpha.[\beta'](y\,(\lambda x.\mu\beta.[\alpha]x)) : \neg\neg A^y \vdash A}\ \textsc{Mu}}{\lambda y.\mu\alpha.[\beta'](y\,(\lambda x.\mu\beta.[\alpha]x)) : \cdot \vdash \neg\neg A \to A}\ \textsc{Lam}$$

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.* [47]

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

Definition 0.4.1.4. *The following defines exceptions within the $\lambda\mu$-calculus:*

$$\begin{aligned} \mathsf{catch}_\alpha\, t &:= \mu\alpha.[\alpha]t \\ \mathsf{throw}_\alpha\, t &:= \mu\beta.[\alpha]t, \text{ where } \beta \text{ is fresh} \end{aligned}$$

Using our reduction rules with the addition of $\mu\alpha.[\alpha]t \rightsquigarrow t$ provided that α is fresh in t[7], we can easily define some nice reduction rules for these definitions.

Definition 0.4.1.5. *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 [47].

Open Problem. 4.1.5. *This is more of an investigation. Most type theories are given in natural deduction form with introduction and elimination rules. Why is this? This we believe is a matter of perspective. Some would claim that natural deduction provides a simple "natural" formulation of computation and that sequent style formulations would burden the programmer, because they are used to programming with natural deduction. However, sequent style type theories yield more symmetries with left and right rules. We will talk more about symmetries and dualities in Sec. 0.4.3.*

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

We believe that the argument against sequent style programming languages is not convincing. It seems sequent style formulations reveal more structure which yield better programming constructs. Thus, it would be worth while investigating using sequent style type theories as programming languages.

Task: *Define a classical and/or intuitionistic sequent style type theory with programming in mind. Investigate the possibility of constructing a surface language for programming in this core type theory. Questions to consider are, does the sequent style formulation provide more of a burden for the programmer? Can the surface language relieve some of this? A lot of research has gone into sequent style logics. Proof search for example is usually conducted on sequent style logics. Can we pull research from proof search into a sequent style type theory to yield more automation?*

0.4.2 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 Morten Sørensen in Rehof's thesis [99]. 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

Syntax:

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

Reduction:

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

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

Figure 0.4.2.18. Syntax and reduction rules for the $\lambda\Delta$-calculus

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

$$\frac{\Gamma, x : \neg A \vdash t : \bot}{\Gamma \vdash \Delta x : \neg A.t : A}\ \textsc{Delta} \qquad \frac{\begin{array}{c} \Gamma \vdash t_2 : A \\ \Gamma \vdash t_1 : A \to B \end{array}}{\Gamma \vdash t_1\ t_2 : B}\ \textsc{App}$$

Figure 0.4.2.19. Type-checking algorithm for the $\lambda\Delta$-calculus

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 Fig. 0.4.2.18[8] and the typing rules in Fig. 0.4.2.19. We give the formulation a la Church, but the formulation a la Curry does exist.

[8]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 [99].

We can see that the syntax really is just the extension of STLC with the $\Delta x : T.t$ control 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 0.4.2.7. *The proof of $\neg\neg A \to A$ in the $\lambda\Delta$-calculus:*

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

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

Context:

$$\begin{aligned} |\Gamma, A^x| &:= |\Gamma|, x : A \\ |\Delta, A^\alpha| &:= |\Delta|, x : \neg A, \textit{where } x \textit{ is fresh in } |\Delta| \end{aligned}$$

Terms and Statements:

$$
\begin{array}{lcl}
|x| & := & x \\
|\alpha| & := & y, \textit{ for some fresh variable } y \\
|\lambda x : A.t| & := & \lambda x : A.|t| \\
|t_1\ t_2| & := & |t_1|\,|t_2| \\
|\mu\alpha : A.s| & := & \Delta x : \neg A.|s|^{\alpha}_{x} \\
|[\alpha]t|^{\alpha}_{x} & := & x\,|t| \\
|[\alpha]t|^{\beta}_{x} & := & z\,|t|, \textit{ where } \alpha \textit{ is distinct from } \beta \textit{ and } z \textit{ is fresh in } t
\end{array}
$$

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 0.4.2.9.

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 0.4.2.10. *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*

$$\begin{aligned} |x_1 : A_1, \ldots, x_i : A_i| &:= {A_1}^{x_1}, \ldots, {A_i}^{x_i} \equiv \Gamma_\mu \\ |y_1 : \neg B_1, \ldots, y_j : \neg B_j| &:= {B_1}^{\alpha_1}, \ldots, {B_j}^{\alpha_j} \equiv \Delta \end{aligned}$$

Terms:

$$\begin{aligned} |x| &:= x \\ |\lambda x : A.t| &:= \lambda x : A.|t| \\ |t_1\ t_2| &:= |t_1|\,|t_2| \\ |\Delta x : \neg A.t| &:= \mu\alpha.[\beta]|t|, \textit{ where } \alpha \not\equiv \beta \end{aligned}$$

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 0.4.2.11. *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.

0.4.3 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 correspondence. 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.

0.4.4 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 [35]. 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 thesis [45]. 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 [105]. However, Selinger's work did not provide an involution of duality. In [105] 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.

Open Problem. 4.4.11. *We mentioned in Sect. 0.4.2 that there are no published proofs of the equivalence of the $\lambda\mu$-calculus. In that section we also provide a sketch of how to go about proving their equivalence. However, another question is, since we can model the $\lambda\mu$-calculus as a control category due to Selinger can we model the $\lambda\Delta$-calculus using the same interpretation as the $\lambda\mu$-calculus. The answer is most certainly yes, but since the $\lambda\Delta$-calculus is slightly less complicated than the $\lambda\mu$-calculus would its control categorical model be less complicated than the $\lambda\mu$-calculus'? We believe it would. Even more so it would provide a way of showing that these beautiful dualities we have been discussing are indeed present in the $\lambda\Delta$-calculus. They are indeed less obvious due the $\lambda\Delta$-calculus being a natural deduction formulation of classical logic.*

Task: *Give a control categorical semantics of the $\lambda\Delta$-calculus and show that it is a less complex internal language to control categories than the $\lambda\mu$-calculus'. Then use this semantics to show that CBN is dual to CBV in the $\lambda\Delta$-calculus.*

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 0.4.4.13. *CBV is defined by the following rules:*

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

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 0.4.4.14. *CBN is defined by the following rules:*

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

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 [35]. 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:

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

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 [35]:

> *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*

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

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

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

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

Figure 0.4.4.20. The Syntax and Reduction Rules for the $\bar{\lambda}\mu\tilde{\mu}$-Calculus

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 Fig. 0.4.4.20 and its typing rules are in Fig. 0.4.4.21.

The new type $A - B$ is the dual to implication called subtraction. It is logically

Commands:

$$\frac{\Gamma \vdash v : A \mid \Delta \quad \Gamma \mid e : A \vdash \Delta}{\langle v \mid e \rangle : (\Gamma \vdash \Delta)} \textsc{Cut}$$

Terms:

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

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

Contexts:

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

$$\frac{\Gamma \vdash v : A \mid \Delta \quad \Gamma \mid e : B \vdash \Delta}{\Gamma \mid v \cdot e : A \to B \vdash \Delta} \textsc{Ctx} \qquad \frac{\Gamma \mid e : B \vdash \beta : A, \Delta}{\Gamma \mid \beta\lambda.e : B - A \vdash \Delta} \textsc{Colam}$$

Figure 0.4.4.21. The Typing Rules for the $\bar{\lambda}\mu\tilde{\mu}$-Calculus

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, 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:

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

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 | 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 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.

0.4.5 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 [126]. 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 Fig. 0.4.5.22. 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

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

Figure 0.4.5.22. Syntax of the Dual Calculus

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 Fig. 0.4.5.23 and the typing

$$\frac{}{(a \cdot \alpha).\alpha \rightsquigarrow a}\ \textsc{EtaR} \qquad \frac{}{x.(x \cdot k) \rightsquigarrow k}\ \textsc{EtaL}$$

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

$$\frac{}{[k]\mathrm{not} \cdot\ \mathrm{not}[a] \rightsquigarrow a \cdot k}\ \textsc{BetaNeg} \qquad \frac{}{\mathsf{inl}\, a \cdot [k, l] \rightsquigarrow a \cdot k}\ \textsc{BetaCoProd1}$$

$$\frac{}{\mathsf{inr}\, a \cdot [k, l] \rightsquigarrow a \cdot l}\ \textsc{BetaCoProd2} \qquad \frac{}{\langle a, b\rangle \cdot\ \mathrm{fst}\, k \rightsquigarrow a \cdot k}\ \textsc{BetaProd1}$$

$$\frac{}{\langle a, b\rangle \cdot\ \mathrm{snd}\, k \rightsquigarrow b \cdot k}\ \textsc{BetaProd2}$$

Figure 0.4.5.23. Reduction Rules for the Dual Calculus

rules are in Fig. 0.4.5.24. 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 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

Terms :

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

$$\frac{\Gamma \vdash \Delta \leftarrow a \rightarrow A}{\Gamma \vdash \Delta \leftarrow \mathsf{inl}\, a \rightarrow A \vee B}\ \textsc{t\_CoProdL} \qquad \frac{\Gamma \vdash \Delta \leftarrow b \rightarrow B}{\Gamma \vdash \Delta \leftarrow \mathsf{inr}\, b \rightarrow A \vee B}\ \textsc{t\_CoProdR}$$

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

Coterms :

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

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

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

Statements :

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

Figure 0.4.5.24. Typing Rules for the Dual Calculus

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 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 0.4.5.15. *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 [126] just like Curien and Herbelin did in [35]. However, in a follow up paper Wadler showed that his duality of the dual calculus into itself is an involution [127]. 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.

0.5 Dependent Type Theory

All the type theories we have seen thus far consist of what are called "simple 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 three perspectives of computation was a system called Automath which was pioneered by de Bruijn in the 1970's [21]. 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.

0.5.1 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 [43, 75]. It is considered the first full dependent type theory. Type Theory is defined by giving a syntax and deriving three judgments. Martin-Löf placed 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

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

Figure 0.5.1.25. The syntax of Martin-Löf's Type Theory

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 Fig. 0.5.1.25. 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: $\mathsf{case}\, s\, \mathsf{of}\, x, y.t$ and $\mathsf{case}\, s\, \mathsf{of}\, x.t_1, y.t_2$. 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 $\mathsf{case}\, s\, \mathsf{of}\, x.t_1, y.t_2$ 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,

$$\frac{}{\Gamma \vdash \bot : \mathsf{Type}}\ \textsc{K\_Bottom} \qquad \frac{}{\Gamma \vdash \top : \mathsf{Type}}\ \textsc{K\_Unit}$$

$$\frac{\Gamma, x : A \vdash B : \mathsf{Type}}{\Gamma \vdash \Sigma x : A.B : \mathsf{Type}}\ \textsc{K\_Ext} \qquad \frac{\Gamma \vdash A : \mathsf{Type} \quad \Gamma \vdash B : \mathsf{Type}}{\Gamma \vdash A \times B : \mathsf{Type}}\ \textsc{K\_Prod}$$

$$\frac{\Gamma, x : A \vdash B : \mathsf{Type}}{\Gamma \vdash \Pi x : A.B : \mathsf{Type}}\ \textsc{K\_Pi} \qquad \frac{\Gamma \vdash A : \mathsf{Type} \quad \Gamma \vdash B : \mathsf{Type}}{\Gamma \vdash A \to B : \mathsf{Type}}\ \textsc{K\_Arrow}$$

$$\frac{\Gamma \vdash A : \mathsf{Type} \quad \Gamma \vdash B : \mathsf{Type}}{\Gamma \vdash A + B : \mathsf{Type}}\ \textsc{K\_Coprod}$$

Figure 0.5.1.26. Kinding for Martin-Löf's Type Theory

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:

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

The kinding rules are defined in Fig. 0.5.1.26. 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

$$\frac{}{\Gamma \vdash \top : \mathsf{True}}\ \textsc{L\_True} \qquad \frac{\Gamma \vdash A : \mathsf{True} \quad \Gamma \vdash B : \mathsf{True}}{\Gamma \vdash A \times B : \mathsf{True}}\ \textsc{L\_Prod}$$

$$\frac{\Gamma, x : A \vdash B : \mathsf{True}}{\Gamma \vdash \Pi x : A.B : \mathsf{True}}\ \textsc{L\_Foralli} \qquad \frac{\Gamma \vdash t : A \quad \Gamma \vdash \Pi x : A.B : \mathsf{True}}{\Gamma \vdash [t/x]B : \mathsf{True}}\ \textsc{L\_Foralle}$$

$$\frac{\Gamma, X : \mathsf{True} \vdash A : \mathsf{True}}{\Gamma \vdash X \to A : \mathsf{True}}\ \textsc{L\_Impi} \qquad \frac{\Gamma \vdash X : \mathsf{True} \quad \Gamma \vdash X \to A : \mathsf{True}}{\Gamma \vdash A : \mathsf{True}}\ \textsc{L\_Impe}$$

$$\frac{\Gamma \vdash A : \mathsf{True}}{\Gamma \vdash A + B : \mathsf{True}}\ \textsc{L\_Ori1} \qquad \frac{\Gamma \vdash B : \mathsf{True}}{\Gamma \vdash A + B : \mathsf{True}}\ \textsc{L\_Ori2}$$

$$\frac{\Gamma \vdash A + B : \mathsf{True} \quad \Gamma, A : \mathsf{True} \vdash C : \mathsf{True} \quad \Gamma, B : \mathsf{True} \vdash C : \mathsf{True}}{\Gamma \vdash C : \mathsf{True}}\ \textsc{L\_Ore} \qquad \frac{\Gamma \vdash t : A \quad \Gamma \vdash [t/x]B : \mathsf{True}}{\Gamma \vdash \Sigma x : A.B : \mathsf{True}}\ \textsc{L\_Exti}$$

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

Figure 0.5.1.27. Validity for Martin-Löf's Type Theory

Fig. 0.5.1.27. 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 Fig. 0.5.1.28. 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

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

$$\frac{x, y \text{ fresh in } C \quad \Gamma \vdash s : \Sigma x : A.B \quad \Gamma, x : A, y : B \vdash c : C}{\Gamma \vdash \mathsf{case}\ s\ \mathsf{of}\ x, y.c : C}\ \text{T\_Case1} \qquad \frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B}{\Gamma \vdash (a, b) : A \times B}\ \text{T\_Prod} \qquad \frac{\Gamma \vdash c : A \times B}{\Gamma \vdash \pi_1\ c : A}\ \text{T\_}$$

$$\frac{\Gamma \vdash c : A \times B}{\Gamma \vdash \pi_2\ c : B}\ \text{T\_Prod2} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A.t : \Pi x : A.B}\ \text{T\_Pi} \qquad \frac{\Gamma \vdash t' : A \quad \Gamma \vdash t : \Pi x : A.B}{\Gamma \vdash t\ t' : [t'/x]B}$$

$$\frac{x \text{ fresh in } B \quad \Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A.t : A \to B}\ \text{T\_Arrow} \qquad \frac{\Gamma \vdash t' : A \quad \Gamma \vdash t : A \to B}{\Gamma \vdash t\ t' : B}\ \text{T\_App2} \qquad \frac{\Gamma \vdash B : \mathsf{Type} \quad \Gamma \vdash a : A}{\Gamma \vdash a : A + B}\ \text{T\_C}$$

$$\frac{\Gamma \vdash A : \mathsf{Type} \quad \Gamma \vdash b : B}{\Gamma \vdash b : A + B}\ \text{T\_CoProd2} \qquad \frac{\Gamma \vdash s : A + B \quad \Gamma, x : A \vdash c : C \quad \Gamma, y : B \vdash c' : C}{\Gamma \vdash \mathsf{case}\ s\ \mathsf{of}\ x.c, y.c' : C}\ \text{T\_Case2} \qquad \frac{\Gamma \vdash A : \mathsf{Type}}{\Gamma, x : \bot \vdash \mathsf{abort} : A}\ \text{T}$$

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

Figure 0.5.1.28. Typing Rules for Martin Löf's Type Theory

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

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

$$\frac{\Gamma \vdash t_1 : \Pi x : A.B}{\Gamma \vdash t_1 = \lambda x : A.(t_1\,x) : \Pi x : A.B}\ \textsc{Eq\_Eta} \qquad \frac{\Gamma \vdash a : A \quad \Gamma, x : A \vdash c : C \quad \Gamma, y : B \vdash c' : C}{\Gamma \vdash \mathsf{case}\ a\ \mathsf{of}\ x.c, y.c' = [a/x]c : [a/x]C}\ \textsc{E}$$

$$\frac{\Gamma \vdash b : B \quad \Gamma, x : A \vdash c : C \quad \Gamma, y : B \vdash c' : C}{\Gamma \vdash \mathsf{case}\ b\ \mathsf{of}\ x.c, y.c' = [b/x]c' : [b/x]C}\ \textsc{Eq\_Case2} \qquad \frac{\Gamma \vdash t : T \quad \Gamma \vdash a : [t/x]A \quad \Gamma, x : T, y : A \vdash b : B}{\Gamma \vdash \mathsf{case}\,(t, a)\ \mathsf{of}\ x, y.b = [t/x][a/y]b : [t/x]}$$

Figure 0.5.1.29. Equality for Martin-Löf's Type Theory

$$\frac{\Gamma \vdash s : A + B \quad \Gamma, x : A \vdash c : C \quad \Gamma, y : B \vdash c' : C}{\Gamma \vdash \mathsf{case}\ s\ \mathsf{of}\ x.c, y.c' : C}\ \textsc{T\_Case2}.$$

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 [75] 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 Fig. 0.5.1.29 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:

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

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:

$$\textit{Kinding}\quad \frac{\Gamma \vdash A : \mathsf{Type} \quad \Gamma \vdash a : A \quad \Gamma \vdash b : A}{\Gamma \vdash \mathsf{Id}\ A\ a\ b : \mathsf{Type}} \qquad \textit{Typing}\quad \frac{\Gamma \vdash a = b : A}{\Gamma \vdash \mathsf{tt} : \mathsf{Id}\ A\ a\ b} \qquad \frac{\Gamma \vdash t : \mathsf{Id}\ A\ a\ b}{\Gamma \vdash a = b : \mathsf{Id}\ A\ a\ b}$$

Using these rules we can prove all of the usual axioms of identity: reflexivity, transitivity, and symmetry [75]. 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 [113, 114].

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

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

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 [15]. See [113, 114, 58, 59] 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 [75]. 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 [40].

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 [29, 30]. 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, because it allows for impredicativity over terms as well as types. 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 [85].

0.5.2 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 $*$[9] called the λ-cube [16]. The following expresses the language of this class of types theories.

Definition 0.5.2.1. *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.

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

[9] 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.

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

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

$$\frac{\Gamma \vdash a : b \quad \Gamma \vdash a' : s \quad (x : a') \notin \Gamma}{\Gamma, x : a' \vdash a : b}\ \textsc{Weakening} \qquad \frac{\Gamma \vdash a' : b \quad \Gamma \vdash a : \Pi x : b.b'}{\Gamma \vdash a\ a' : [a'/x]b'}\ \textsc{App} \qquad \frac{\Gamma \vdash a : b \quad \Gamma \vdash b' : s \quad b}{\Gamma \vdash a : b'}$$

$$\frac{\Gamma \vdash a : s_1 \quad \Gamma, x : a \vdash b : s_2 \quad (s_1, s_2) \in R}{\Gamma \vdash \Pi x : a.b : s_2}\ \textsc{Pi} \qquad \frac{\Gamma, x : a \vdash b : b' \quad \Gamma \vdash \Pi x : a.b' : s}{\Gamma \vdash \lambda x : a.b : \Pi x : a.b'}\ \textsc{Lam}$$

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 article 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 $R = \{(*,*),(\Box,*),(\Box,\Box)\}$.

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

Figure 0.5.2.30. Syntax for the Separated Calculus of Constructions

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 [32]. 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 Fig. 0.5.2.30[10]. This formulation simply extends system 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 Fig. 0.5.2.31, the kinding rules are defined in Fig. 0.5.2.32, the typing rules in Fig. 0.5.2.33, and finally the equality rules in Fig. 0.5.2.34. We

[10]We do not have a citation for where this formulation can be found. It was learned by the author from Hugo Herbelin at the 2011 Oregon Programming Language Summer School.

$$\frac{}{\Gamma \vdash \mathsf{Type} : *}\ \text{S\_Type} \qquad \frac{\Gamma, X : K \vdash K' : *}{\Gamma \vdash \Pi X : K.K' : *}\ \text{S\_Prod1}$$

$$\frac{\Gamma, x : T \vdash K : *}{\Gamma \vdash \Pi x : T.K : *}\ \text{S\_Prod2}$$

Figure 0.5.2.31. Sorting Rules for the Separated Calculus of Constructions

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

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

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

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

Figure 0.5.2.32. 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 0.5.2.33. Typing Rules for the Separated Calculus of Constructions

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

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

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

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

$$\frac{t_2 \approx t_2'}{t_1\, t_2 \approx t_1\, t_2'}\ \text{R\_App2} \qquad \frac{t_1 \approx t_2}{T\, t_1 \approx T\, t_2}\ \text{R\_App3}$$

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

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

Figure 0.5.2.34. The Equality for the Separated Calculus of Constructions

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 article. 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. Throughout the article so far we have mentioned the connection of type theory to programming language research. In the next section we give more details of this connection. We will also discuss several real world applications of

the type theories we have discussed above. Following the next section will be the final section which discusses how to reason about type theories at the meta-level. However, before discussing the design of programming languages we first give some open problems in dependent type theory.

Open Problems in Dependent Type Theory. In Section 0.4 we introduced classical type theory. Take note that every dependent type theory we have introduced in this section is intuitionistic. It turns out that very little research on classical dependent type theory has been done. Gilles Barthe et al. defined what they call classical pure type systems in [17]. What they did was take the Δ-operator from the $\lambda\Delta$-calculus and added it to PTS'. Then they did some preliminary meta-theory of classical pure type systems. Another paper on classical dependent type theory was by Herbelin [56], which showed that if one takes the extensional version of Martin-Löf's type theory, full β-reduction, and the μ-abstraction from the $\lambda\mu$-calculus then the type theory becomes inconsistent. He then goes on to show that by choosing a particular reduction strategy like CBV or CBN this problem goes away. Paul Levy shows in [72] that taking Marin-Löf's Type Theory and adding Peirce's law which when combined with the axiom of choice results in unwanted stuck terms. He shows that this occurs with any chosen reduction strategy. Nuria Brede extends Nuprl proof assistants core type theory called computational type theory with the μ-operator from the $\lambda\mu$-calculus in [20]. They are not concerned with extracting computational content from the classical logic, but rather to just allow for proofs to reason using classical reasoning. These are the only advances in classical dependent type theory that we

know of. This suggests that while the research on intuitionistic dependent type theory has been throughly investigated over the course of the last forty years, since Automath, classical dependent type theory has not. This is startling, since classical type theory provides a lot of conveniences and beautiful structure. Hence, we arrive at the following open problem[11].

Open Problem. 5.2.2. *Is it possible to formulate a classical dependent type theory which is consistent with full β-reduction with the presence of advanced typing features?*

We believe this is an achievable goal, but we are unsure of whether we may need to come up with new operators different from the μ-abstraction and the Δ-abstraction. Another open problem presents itself with respect to the dual calculus.

Open Problem. 5.2.3. *If we choose CBV as a reduction strategy and extend the dual calculus with dependency what are the duals of dependent product? It seems it would be a strong disjoint union over the output. Is this true? This implies that the dual to disjoint union would be a product type over the output. How does equality work in such a language?*

[11]We are not the only people concerned about this gap. Robin Adams has begun a four year program to try and solve this very problem. Hugo Herbelin has been working on a very promising idea called delimited control. I do not discuss his work here, because I have just learned of it.

0.6 Dependent Types, Proof Assistants, and Programming Languages

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 [93]. 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 [125, 19, 76, 8, 86, 115, 73]. Now dependent types are very powerful and provide a rich environment for verification. They also 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 [109]. 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 [46, 139]. 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 Algebraic Data Types (GADT) [135]. These have been added to Haskell [61]. 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 [10, 14, 33, 66, 138]. 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 [13] 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. The proof assistant Coq is based on an extension of Coquand's CoC called the Calculus of Inductive Constructions [119]. 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 [51]. A C compiler has been formally verified with in Coq [70, 69]. This project is called CompCert. Agda is the second proof assistant based on dependent type theory. The core of Agda is Martin-Löf's Type Theory. However, we are not aware of any large scale mathematics in Agda. NuPrl is another proof assistant based on Martin-Löf's Type Theory [28]. Finally, Twelf is a proof assistant based on a restricted version of Martin-Löf's Type Theory called LF [91]. More information on proof assistants can be found in [48]. 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 [136]. 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 [132] 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 [137].

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 the three perspectives of computation, and is called the proof fragment. Garrin Kimmel et al. show in [63] 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 termi-

nating. 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 [79]. The erasure was investigated in a similar setting as Sep^3 in [110]. 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 [22] 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 three perspectives of computation, and can be used to proof check large scale mathematics. Dependent types are the future of programming languages.

0.7 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 [112].

Definition 0.7.0.1. *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 three perspectives of computation 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 0.7.0.2. *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 [112]. 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 [111]. 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 [112]. 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'.

Open Problem. 7.0.2. *Are there any dependent type theories where weak normalization implies strong normalization?*

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.

0.7.1 Hereditary Substitution

In [97] 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 [50, 9, 71]. These proofs have a particular structure and are completely constructive. Kevin Watkins was the first to make their constructive content explicit [129]. 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 [5].

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 0.7.1.4. *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 [129]. Adams extended this work to dependent types in his thesis [5]. We will show how to use hereditary substitution to show weak normalization. Let's consider how to define hereditary substitution for STLC.

0.7.2 Hereditary Substitution for STLC

The definition of the hereditary substitution depends on a partial function called *ctype*. It is defined by the following definition.

Definition 0.7.2.5. *The partial function ctype 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 [42].

Lemma 0.7.2.6.

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 0.7.2.7. *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 [41] 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 0.7.2.6, $T'' \to T'$ 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 0.7.2.6 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 seman-

tics 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 0.7.2.8. *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 0.7.2.9. *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 0.7.2.10. *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 0.7.2.11. *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 [98] 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 0.7.2.12. *If $n' \in [\![T']\!]_{\Gamma,x:T,\Gamma'}$, $n \in [\![T]\!]_\Gamma$, then $[n/x]^T n' \in [\![T']\!]_{\Gamma,\Gamma'}$.*

Proof. By Lemma 0.7.2.8 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 0.7.2.9 $\hat{n}$ 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 0.7.2.13. *If* $\Gamma \vdash t : T$ *then* $t \in [\![T]\!]_\Gamma$.

Corollary 0.7.2.14. *If* $\Gamma \vdash t : T$ *then* $t \rightsquigarrow^! n$.

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 [41]. The advantage of hereditary substitution is that it shows promise of being less complex than other normalization techniques. This means that it would be easier to formalize in proof assistants. However, there is a big drawback of hereditary substitution and that is it is not known how many type theories can be proven normalizing using it. Which leads us to a few open problems.

Open Problem. 7.2.14. *Can system T be proven normalizing using hereditary substitution?*

The solution this open problem has alluded us for quite sometime. It seems as if it

would be a straightforward extension of the proof of normalization for SLTC, but the natural number recursor makes it very hard to put an ordering on the types. The following open problem is even harder to solve.

Open Problem. 7.2.15. *Can system F be proven normalizing using hereditary substitution?*

Clearly no natural number ordering will exist for the types of system F. If one did then we could prove consistency of second order arithmetic using a natural number ordering. We know this is impossible. However, it may be possible to use a semantic technique to prove the properties of the hereditary substitution function. We are currently pursuing the idea to use categorical models to prove the properties of the hereditary substitution function for system F. In [60] Felix Joachimski and Ralph Matthes define a similar function to hereditary substitution and concludes weak and strong normalization of STLC, system T, and system F. It maybe possible to adopt some of their work to hereditary substitution to be able to solve the previous two open problems.

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 [3]. 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 [62]. 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.

0.7.3 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 T using an interpretation of types based on set theory with comprehension. He called this interpretation saturated sets. Later, John Yves Girard, against popular belief[12], 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 [50]. We will summarize how to show strong normalization of STLC using Tait's

[12]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 [50].

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.

Definition 0.7.3.17. *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 [50].

Lemma 0.7.3.18. *If $t \in [\![T]\!]$, then $t \in \mathsf{SN}$.*

Lemma 0.7.3.19. *If $t \in [\![T]\!]$ and $t \rightsquigarrow t'$ then $t' \in [\![T]\!]$.*

Lemma 0.7.3.20. *If t is neutral, $t' \in [\![T]\!]$ and $t \rightsquigarrow t'$ then $t \in [\![T]\!]$.*

Lemma 0.7.3.21. *If t is neutral and normal then $t \in [\![T]\!]$.*

The proof that $[\![T]\!]$ defined in Def. 0.7.3.26 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 0.7.3.22. *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 0.7.3.23. *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 $\Gamma = \{\sigma \mid \Gamma \vdash \sigma\}$. We now have everything we need to show that all typeable terms are reducible, hence, strongly normalizing.

Theorem 0.7.3.24. *If $\sigma \in [\![\Gamma]\!]$ and $\Gamma \vdash t : T$ then $\sigma\, t \in [\![T]\!]$.*

Corollary 0.7.3.25. *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 0.7.3.26. *The interpretation of types are defined as follows:*

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

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. 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 0.7.3.27. *Well-formed substitutions:*

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

Definition 0.7.3.28. *Well-formed reducibility candidate substitutions:*

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

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

Definition 0.7.3.29. *Well-formed substitution with respect to a well-formed reducibility candidate substitution:*

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

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 0.7.3.30. *$[\![T]\!]_\rho$ 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 0.7.3.31. *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 0.7.3.32. *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 0.7.3.33. *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 0.7.3.34. *If* $\Gamma \vdash t : T$, $\Gamma \vdash \sigma$ *and* $\Gamma \vdash \rho$ *then,* $\sigma t \in [\![T]\!]_\rho$.

Corollary 0.7.3.35. *If* $\cdot \vdash t : T$ *then* $t \in SN$.

The proof of the corollary follows from Theorem 0.7.3.34 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 [50].

0.7.4 Logical Relations

Logical relations are straightforward extensions of reducibility sets. They can be thought of as unary 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 $n > 2$. Logical relations have been used in a wide range of applications, from consistency proofs all the way to encryption.

Andrew Pitts used logical relations to show when two inhabitants of the dis-

joint union type are equivalent in [95]. Karl Crary gives a nice introduction to logical relations in [94] 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 [117]. 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.

0.7.4.1 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? There is a partial solution to this question and an existing open problem.

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 out-

standing 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 [12] 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 [6]. Since this earlier work a number of applications of step-indexed logical relations have been conducted, e.g. [4, 7, 84, 124]. One drawback of using step-indexed logical relations is that the proofs usually involve tedious computations of step indices. In [39] 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 [24] 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 0.7.4.36. $\cdot \vdash t : T$ *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[13]. We do not show any proofs here, but they can all be found in the extended version of [6] which can be accessed through Ahmed's web page[14].

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.

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

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

Definition 0.7.4.37. *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 as follows:

Lemma 0.7.4.38. *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 0.7.4.39. *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 0.7.4.40. *We define logical relations for values and then we extend this definition to expressions (terms t).*

Logical relations for values:

$$\begin{aligned}
\mathcal{V}[\![\alpha]\!]_\rho &= \rho(\alpha) \\
\mathcal{V}[\![X]\!]_\rho &= \{v \mid \cdot \vdash v : X\} \\
\mathcal{V}[\![T_1 \to T_2]\!]_\rho &= \{\lambda x : T.t \mid \cdot \vdash \lambda x : T_1 : T_1 \to T_2 \wedge \forall v.v \in \mathcal{V}[\![T_1]\!]_\rho \implies [v/x]t \in \mathcal{E}[\![T_2]\!]_\rho\} \\
\mathcal{V}[\![\mu\alpha.T]\!]_\rho &= \{\mathit{fold}\ v \mid \forall v.\mathit{unfold}\ (\mathit{fold}\ v) \in \mathcal{V}[\![[\mu\alpha.T/\alpha]T]\!]_\rho.
\end{aligned}$$

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 0.7.4.41. *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 0.7.4.42.

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

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 0.7.4.43. *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 0.7.4.44. *Logical relations for values:*

$$\begin{aligned}
\mathcal{V}[\![\alpha]\!]_\rho &= \rho(\alpha) \\
\mathcal{V}[\![X]\!]_\rho &= \{(k,v) \in Atom^{value} \mid \cdot \vdash v : X\} \\
\mathcal{V}[\![T_1 \to T_2]\!]_\rho &= \{(k, \lambda x : T.t) \in Atom^{value} \mid \cdot \vdash \lambda x : T_1.t : T_1 \to T_2 \wedge \\
&\quad \forall j \leq k. \forall v. (j,v) \in \mathcal{V}[\![T_1]\!]_\rho \implies (j, [v/x]t) \in \mathcal{T}[\![T_2]\!]_\rho\} \\
\bar{\mathcal{V}}_n[\![\mu\alpha.T]\!]_\rho &= \{(k, fold\ v) \in Atom^{value} \mid k < n \wedge \forall j < k. (j,v) \in \mathcal{V}[\![T]\!]_{\rho[\alpha \mapsto \bar{\mathcal{V}}_k[\![\mu\alpha.T]\!]_\rho]}\} \\
\mathcal{V}[\![\mu\alpha.T]\!]_\rho &= \bigcup_{n \geq 0} \bar{\mathcal{V}}_n[\![\mu\alpha.T]\!]_\rho
\end{aligned}$$

The next definition extends the previous to terms.

Definition 0.7.4.45. *Logical relations extended to terms:*

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

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 0.7.4.46. *Well-formed term-variable substitutions:*

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

Definition 0.7.4.47. *Well-formed type-variable contexts:*

$$\begin{aligned}\mathcal{D}[\![\cdot]\!] &= \{\emptyset\} \\ \mathcal{D}[\![\Delta, \alpha]\!] &= \{\rho[\alpha \mapsto \mathcal{I}] \mid \rho \in \mathcal{D}[\![\Delta]\!] \wedge \mathcal{I} \in Type\}\end{aligned}$$

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 0.7.4.48. *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 0.7.4.49. *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 0.7.4.50. *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. Step-index logical relations have been used with recursive types, but not for general recursive programs and inconsistent type theories. We arrive at the following open problem.

Open Problem. 7.4.50. *Suppose we have a dependent type theory where it has*

been separated into two fragments, a logical fragment and a programmatic fragment. It does not matter whether it has a collapsed syntax or not. The logical fragment is at least dependent system F and the programmatic fragment is at least CoC with general recursion and Type : Type. *Furthermore, the type theory has freedom of speech. Thus, programs contain proofs and proofs contain programs.*

Is it possible to combine logical relations and step-index logical relations to show consistency of the logical fragment?

We said at the beginning of this section that Casinghino does exactly this in [24], but their type theory does not contain Type : Type.

Part II

Design

Part III

Analysis

APPENDIX A

SAMPLE APPENDIX

A.1 Appendix One

A.2 Appendix Two

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 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.

[4] U. Acar, A. Ahmed, and M. Blume. Imperative self-adjusting computation. 2008.

[5] R. Adams. *A Modular Hierarchy of Logical Frameworks*. PhD thesis, 2004.

[6] A. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *Lecture Notes in Computer Science*. ESOP, 2006.

[7] A. Ahmed, D. Dreyer, and A. Rossberg. State-dependent representation independence. *POPL*, 2009.

[8] 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.

[9] R. Amadio and P. Curien. *Domains and lambda-calculi*. Cambridge tracts in theoretical computer science. Cambridge University Press, 1998.

[10] J. Andrews and Y. Zhang. General test result checking with log file analysis. *IEEE Trans. Softw. Eng.*, 29:634–648, July 2003.

[11] P. Andrews. Churchś type theory. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Spring 2009 edition, 2009.

[12] 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.

[13] A. Armstrong, S. Foster, and G. Struth. Dependently typed programming based on automated theorem proving. *CoRR*, abs/1112.3833, 2011.

[14] 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.

[15] S. Awodey. Type theory and homotopy. *Preprint*, 2010.

[16] 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.

[17] 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.

[18] blogs.consumerreports.org. Consumer reports cars blog: Japan investigates reports of prius brack problem, 2010.

[19] 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.

[20] N. Brede. *lambda-mu-PRL - A Proof Refinement Calculus for Classical Reasoning in Computational Type Theory.* PhD thesis, University of Potsdam, 2009.

[21] 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.

[22] L. Cardelli. A polymorphic lambda-calculus with type:type. Technical report, 1986.

[23] F. Cardone and R. Hindley. History of lambda-calculus and combinatory logic. 2006.

[24] C. Casinghino, V. Sjöberg, and S. Weirich. Step-indexed normalization for a language with general recursion. In *MSFP '12*, 2012.

[25] C. Chen and H. Xi. Combining programming with theorem proving. In *Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, ICFP '05, pages 66–77, New York, NY, USA, 2005. ACM.

[26] A. Church. A set of postulates for the foundation of logic. *The Annals of Mathematics*, 34(4):pp. 839–864, 1933.

[27] A. Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5(2):pp. 56–68, 1940.

[28] 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.

[29] T. Coquand. An analysis of girard's paradox. In *In Symposium on Logic in Computer Science*, pages 227–236. IEEE Computer Society Press, 1986.

[30] 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.

[31] T. Coquand. Type theory. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Spring 2010 edition, 2010.

[32] T. Coquand and G. Huet. The calculus of constructions. *Inf. Comput.*, 76(2-3):95–120, 1988.

[33] 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.

[34] R. Crole. *Categories for Types*. Cambridge Mathematical Textbooks. Cambridge University Press, 1994.

[35] P. Curien and H. Herbelin. The duality of computation. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, ICFP '00, pages 233–243, New York, NY, USA, 2000. ACM.

[36] N. Danner. Ordinals and ordinal functions representable in the simply typed lambda calculus. *Annals of Pure and Applied Logic*, 97(1-3):179 – 201, 1999.

[37] N. Danner and D. Leivant. Stratified polymorphism and primitive recursion. *Mathematical. Structures in Comp. Sci.*, 9(4):507–522, 1999.

[38] R. David and K. Nour. A short proof of the strong normalization of the simply typed lambdamu-calculus. *SCHEDAE INFORMATICAE*, 12:27–33, 2003.

[39] D. Dreyer, A. Ahmed, and L. Birkedal. Logical step-indexed logical relations. *Logical Methods in Computer Science*, 7(2), 2011.

[40] P. Dybjer. Representing inductively defined sets by wellorderings in martin-löf's type theory. *Theoretical Computer Science*, 176(1-2):329, 1997.

[41] H. Eades and A. Stump. Hereditary substitution for stratified system f. In *Proof-Search in Type Theories (PSTT)*, 2010.

[42] H. Eades and A. Stump. Using the hereditary substitution function in normalization proofs, 2011.

[43] Per Martin-Löf. Cohen et al., editor. *Constructive mathematics and computer programming*, volume 1, North-Holland, Amsterdam., 1982.

[44] S. Feferman. Predicativity. In S. Shapiro, editor, *The Oxford Handbook of Philosophy of Mathematics and Logic*, pages 590–624. Oxford University Press, 2005.

[45] A. Filinski. *Declarative Continuations and Categorical Duality*. Rapport. Univ., 1989.

[46] 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.

[47] H. Geuvers, R. Krebbers, and J. McKinna. The lambda-mu-t-calculus. *Annals of Pure and Applied Logic*, 2012.

[48] H. Geuvers and G. Nijmegen. Proof assistants: history, ideas and future, Feburary 2009.

[49] J. Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1971.

[50] J. Girard, Y. Lafont, and P. Taylor. *Proofs and Types (Cambridge Tracts in Theoretical Computer Science)*. Cambridge University Press, April 1989.

[51] G. Gonthier. A computer-checked proof of the four colour theorem. 2005.

[52] 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.

[53] C. Gunter. *Semantics of Programming Languages: Structures and Techniques*. Foundations of Computing. MIT Press, 1992.

[54] R. Harper and C. Stone. A type-theoretic interpretation of standard ml. Book Chapter 898, Carnegie Mellon University, 1998.

[55] 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.

[56] H. Herbelin. On the degeneracy of σ-types in the presence of computational classical logic. In Pawel Urzyczyn, editor, *Typed Lambda Calculi and Applications, 7th International Conference, TLCA 2005, Nara, Japan, April 21-23, 2005, Proceedings*, volume 3461 of *LNCS*, pages 209–220. Springer, 2005.

[57] J. Hintikka. *From Dedekind to Gödel: essays on the development of the foundations of mathematics*. Synthese library. Kluwer Academic Publishers, 1995.

[58] H. Hofmann. *Extensional Concepts in Intensional Type Theory*. Thesis. University of Edinburgh, Department of Computer Science, 1995.

[59] 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.

[60] F. Joachimski and R. Matthes. Short proofs of normalization for the simply-typed lambda-calculus, permutative conversions and gödel's t, 1999.

[61] 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.

[62] 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.

[63] 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.

[64] S. Kleene and J. Rosser. The inconsistency of certain formal logics. *The Annals of Mathematics*, 36(3):pp. 630–636, 1935.

[65] 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.

[66] G. Klein and T. Nipkow. Verified bytecode verifiers. *Theor. Comput. Sci.*, 298:583–626, April 2003.

[67] F. Lawvere and S. Schanuel. *Conceptual Mathematics: A First Introduction to Categories*. Conceptual Mathematics: A First Introduction to Categories. Cambridge University Press, 2009.

[68] D. Leivant. Finitely stratified polymorphism. *Inf. Comput.*, 93(1):93–113, 1991.

[69] 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.

[70] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[71] 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.

[72] P. Levy. Martin-löf clashes with griffin, operationally. 2001.

[73] 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.

[74] P. Martin-Löf. Analytic and synthetic judgements in type theory. In Paolo Parrini, William Demopoulos, David Devidi, Robert Disalle, and Wayne Myrvold, editors, *Kant and Contemporary Epistemology*, volume 54 of *The Western Ontario Series in Philosophy of Science*, pages 87–99. Springer Netherlands, 1994.

[75] P. Martin-Löf and G.Sambin. *Intuitionistic type theory*. Studies in proof theory. Bibliopolis, 1984.

[76] C. McBride and J. McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.

[77] E. Mendelson. *Introduction to Mathematical Logic*. Chapman & Hall/CRC, 5th edition, 2009.

[78] G. Mints. *A short introduction to intuitionistic logic*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[79] Nathan Mishra-Linger and Tim 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.

[80] G. Morrisett. Compiling with types, 1995.

[81] C. Murthy. Classical proofs as programs: How, what and why. Technical report, Cornell University, 1991.

[82] A. Nanevski and G. Morrisett. Dependent type theory of stateful higher-order functions. Technical report, 2005.

[83] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: dependent types for imperative programs. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, pages 229–240, New York, NY, USA, 2008. ACM.

[84] G. Neis, D. Dreyer, and A. Rossberg. Non-parametric parametricity. *ICFP*, 2009.

[85] B. Nordstrom, K. Petersson, and J. Smith. *Programming in Martin-Löf's Type Theory: An Introduction*. Oxford University Press, USA, July 1990.

[86] U. Norell. Towards a practical programming language based on dependent type theory. PhD Thesis, 2007.

[87] The German Federal Bureau of Aircraft Accidents. Investigation report, 2004.

[88] 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.

[89] M. Parigot. Lambda-mu-calculus: An algorithmic interpretation of classical natural deduction. In Andrei 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.

[90] M. Parigot. Proofs of strong normalisation for second order classical natural deduction. *Journal of Symbolic Logic*, 62(4):1461–1479, 1997.

[91] 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.

[92] B. Pierce. *Basic Category Theory for Computer Scientists*. Foundations of Computing. MIT Press, 1991.

[93] B. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[94] B. Pierce. *Advanced Topics in Types and Programming Languages*. The MIT Press, 2004.

[95] A. Pitts. Existential types: Logical relations and operational equivalence. In Kim Larsen, Sven Skyum, and Glynn Winskel, editors, *Automata, Languages and Programming*, volume 1443 of *Lecture Notes in Computer Science*, pages 309–326. Springer Berlin / Heidelberg, 1998. 10.1007/BFb0055063.

[96] A. Platzer and C. Edmund. Formal verification of curved flight collision avoidance maneuvers: A case study. In Ana Cavalcanti and Dennis Dams, editors, *FM*, volume 5850 of *LNCS*, pages 547–562. Springer, 2009.

[97] D. Prawitz. *Natural Deduction: A Proof-Theoretical Study*. Dover Publications, 1965.

[98] D. Prawitz. *Logical Consequence from a Constructivist Point of View*, pages 671–695. Volume 1 of Shapiro [108], 2005.

[99] J. Rehof and M. Sørensen. The lambdadelta-calculus. In *Proceedings of the International Conference on Theoretical Aspects of Computer Software*, TACS '94, pages 516–542, London, UK, 1994. Springer-Verlag.

[100] 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.

[101] Reuters. Toyota to recall 436,00 hybrids globally-document, February 2010.

[102] 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.

[103] J. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998.

[104] D. Schmidt. *The Structure of Typed Programming Languages*. MIT Press, Cambridge, MA, USA, 1994.

[105] 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.

[106] P. Sewell, F. Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. StrniŠa. Ott: Effective tool support for the working semanticist. *J. Funct. Program.*, 20(1):71–122.

[107] 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.

[108] S. Shapiro. *The Oxford Handbook of Philosophy of Mathematics and Logic*. Oxford University Press, 2005.

[109] T. Sheard. Type-Level Computation Using Narrowing in Ωmega. In *Programming Languages meets Program Verification*, volume 1643, 2006.

[110] 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 James Chapman and Paul Blain 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.

[111] M. Sorensen. Strong normalization from weak normalization in typed lambda-calculi. *Information and Computation*, 133:35–71, 1997.

[112] 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.

[113] T. Streicher. *Semantics of Type Theory: Correctness, Completeness, and Independence Results*. Progress in theoretical computer science. Birkhäuser, 1991.

[114] T. Streicher. *Investigations Into Intensional Type Theory*. PhD thesis, Habilitation- sschrift, Ludwig-Maximilians-Universität Müchen, November 1993.

[115] 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.

[116] M. Sulzmann, M. Chakravarty, S. Peyton Jones, and K. Donnelly. System f with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*, TLDI '07, pages 53–66, New York, NY, USA, 2007. ACM.

[117] E. Sumii and B. Pierce. Logical relation for encryption. *J. Comput. Secur.*, 11(4):521–554, July 2003.

[118] W. Tait. Infinitely long terms of transfinite type. In J.N. Crossley and M.A.E. Dummett, editors, *Formal Systems and Recursive Functions*, volume 40 of *Studies in Logic and the Foundations of Mathematics*, pages 176 – 185. Elsevier, 1965.

[119] The Coq Development Team. The coq proof assistant reference manual, 2008.

[120] thedetroitbureau.com. Nhtsa memo on regenerative braking, April 2011.

[121] A. Troelstra. *History of constructivism in the 20th century*. ITLI Prepublication Series ML-91-05, 1991.

[122] A. Troelstra and H. Schwichtenberg. *Basic proof theory (2nd ed.)*. Cambridge University Press, New York, NY, USA, 2000.

[123] United States Federal Bureau of Investigation. 2005 FBI Computer Crime Survey.

[124] D. Vytiniotis and V. Koutavas. Relating step-indexed logical relations and bisimulations. Technical Report MSR-TR-2009-25, Microsoft Research, March 2009.

[125] D. Vytiniotis and S. Weirich. Dependent types: Easy as PIE. In Marco T. Morazán and Henrik 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.

[126] P. Wadler. Call-by-value is dual to call-by-name. *SIGPLAN Not.*, 38(9):189–201, August 2003.

[127] P. Wadler. Call-by-value is dual to call-by-name – reloaded. In Jürgen Giesl, editor, *Term Rewriting and Applications*, volume 3467 of *Lecture Notes in Computer Science*, pages 185–203. Springer Berlin Heidelberg, 2005.

[128] P. Wadler. The girard,reynolds isomorphism (second edition). *Theoretical Computer Science*, 375(1-3):201 – 226, 2007. Festschrift for John C. Reynolds 70th birthday.

[129] K. Watkins, I. Cervesato, F. Pfenning, and D. Walker. A concurrent logical framework: The propositional fragment. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *Types for Proofs and Programs*, volume 3085 of *Lecture Notes in Computer Science*, pages 355–377. Springer Berlin / Heidelberg, 2004.

[130] H. Weyl. Das kontinuum. Translated as Weyl 1994, 1918.

[131] W.Howard. The formulae-as-types notion of construction. 1969-1980.

[132] 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.

[133] H. Xi. Dependently Typed Data Structures. In *Proceedings of Workshop of Algorithmic Aspects of Advanced Programming Languages (WAAAPL '99)*, pages 17–32, Paris, September 1999.

[134] H. Xi. Dependent Types for Program Termination Verification. In *Proceedings of 16th IEEE Symposium on Logic in Computer Science*, pages 231–242, Boston, June 2001.

[135] 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.

[136] 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.

[137] 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.

[138] 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.

[139] C. Zenger. Indexed types. *Theoretical Computer Science*, 187(1Äì2):147 – 165, 1997.