Cedille2: A proof theoretic redesign of the Calculus of Dependent Lambda Eliminations

by

Andrew Marmaduke

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

May 2024

Thesis Committee: Aaron Stump, Thesis Supervisor
Cesare Tinelli
J. Garrett Morris
Sriram Pemmaraju
William J. Bowman

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

– Some wise dude

ACKNOWLEDGMENTS

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

ABSTRACT

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

PUBLIC ABSTRACT

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

PREFACE

Chapter 1

INTRODUCTION

Type theory is a tool for reasoning about assertions of some domain of discourse. When applied to programming languages, that domain is the expressible programs and their properties. Of course, a type theory may be rich enough to express detailed properties about a program, such that it halts or returns an even number. Therein lies a tension between what properties a type theory can faithfully (i.e. consistently) encode and the complexity of the type theory itself. If the theory is too complex then it may be untenable to prove that the type theory is well-behaved. Indeed, the design space of type theories is vast, likely infinite. When incorporating features the designer must balance complexity against capability.

Modern type theory arguably began with Martin-Löf in the 1970s and 1980s when he introduced a dependent type theory with the philosophical aspirations of being an alternative foundation of mathematics [29, 30]. Soon after in 1985, the Calculus of Constructions (CC) was introduced by Coquand [11, 12]. Inductive data (e.g. natural numbers, lists, trees) was shown by Guevers to be impossible to derive in CC [20]. Nevertheless, inductive data was added as an extension by Pfenning [36] and the Calculus of Inductive Constructions (CIC) became the basis for the proof assistant Rocq [34].

In the early 1990s Barendregt introduced a generalization to Pure Type Systems (PTS) and studied CC under his now famous λ-cube [5, 4]. The λ-cube demonstrated how CC could be deconstructed into four essential sorts of functions. At its base was the Simply Typed Lambda Calculus (STLC) a type theory introduced in the 1940s by Church to correct logical consistency issues in his (untyped) λ-calculus [8]. The STLC has only basic functions found in all programming languages. System F, a type theory introduced by Girard [22, 23] and independently by Reynolds [39], is obtained from STLC by adding quantification over types (i.e. polymorphic functions). Adding a

copy of STLC at the type-layer, functions from types to types, yields System F^{ω}. Finally, the addition of quantification over terms or functions from terms to types, completes CC. While this is not the only path through the λ-cube to arrive at CC it is the most well-known and the most immediately relevant.

Perhaps surprisingly, all the systems of the λ-cube correspond to a logic. In the 1970s Curry circulated his observations about the STLC corresponding to intuitionistic propositional logic [24]. Reynolds and Girard's combined work demonstrated that System F corresponds to second-order intuitionistic propositional logic [22, 39, 40]. Indeed, Barendregt extended the correspondence to all systems in his λ-cube noting System F^{ω} as corresponding to higher-order intuitionistic propositional logic and CC as corresponding to higher-order intuitionistic predicate logic [4]. Fundamentally, the Curry-Howard correspondence associates programs of a type theory with proofs of a logic, and types with formula. However, the correspondence is not an isomorphism because the logical view does not possess a unique assignment of proofs. The type theory contains potentially *more* information than the proof derivation.

Cedille is a programming language with a core type theory based on CC [42, 43]. However, Cedille took an alternative road to obtaining inductive data than what was done in the 1980s. Instead, CC was modified to add the implicit products of Miquel [31], the dependent intersections of Kopylov [26], and an equality type over untyped terms. The initial goal of Cedille was to find an efficient way to encode inductive data. This was achieved in 2018 with Mendler-style lambda encodings [14]. However, the design of Cedille sacrificed certain properties such as the decidability of type checking. Decidability of type checking was stressed by Kreisel to Scott as necessary to reduce proof checking to type checking because a proof does not, under Kreisel's philosophy, diverge [41]. This puts into contention if Cedille corresponds to a logic at all. What remains is to describe the redesign of Cedille such that it does have decidability of type checking and to argue why this state of affairs is preferable. However, completing this journey requires a deeper introduction into the type theories of the λ-cube.

$$
\begin{aligned}
t &::= x \mid \mathfrak{b}(\kappa_1, x : t_1, t_2) \mid \mathfrak{c}(\kappa_2, t_1, \ldots, t_{\mathfrak{a}(\kappa_2)}) \\
\kappa_1 &::= \lambda \mid \Pi \\
\kappa_2 &::= \star \mid \Box \mid \text{app}
\end{aligned}
$$

$$
\begin{array}{rcl}
\mathfrak{a}(\star) = \mathfrak{a}(\Box) = 0 & \qquad \lambda\, x : t_1.\, t_2 := \mathfrak{b}(\lambda, x : t_1, t_2) & \qquad \star := \mathfrak{c}(\star) \\
\mathfrak{a}(\text{app}) = 2 & (x : t_1) \to t_2 := \mathfrak{b}(\Pi, x : t_1, t_2) & \qquad \Box := \mathfrak{c}(\Box) \\
& t_1\ t_2 := \mathfrak{c}(\text{app}, t_1, t_2) &
\end{array}
$$

Figure 1.1: Syntax for System F^ω.

1.1 System F^ω

The following description of System F^ω differs from the standard presentation in a few important ways:

1. the syntax introduced is of a generic form which makes certain definitions more economical,

2. a bidirectional PTS style is used but weakening is replaced with a well-formed context relation.

These changes do not affect the set of proofs or formula that are derivable internally in the system.

Syntax consists of three forms: variables ($x, y, z, \ldots$), binders ($\mathfrak{b}$), and constructors ($\mathfrak{c}$). Every binder and constructor has an associated discriminate or tag to determine the specific syntactic form. Constructor tags have an associated arity ($\mathfrak{a}$) which determines the number of arguments, or subterms, the specific constructor contains. A particular syntactic expression will be interchangeably called a syntactic form, a term, or a subterm if it exists inside another term in context. See Figure 1.1 for the complete syntax of F^ω. Note that the grammar for the syntax is defined using a BNF-style [15] where $t ::= f(t_1, t_2, \ldots)$ represents a recursive definition defining a category of syntax, t, by its allowed subterms. For convenience a shorthand form is defined for each tag to maintain a more familiar appearance with standard syntactic definitions. Thus, instead of writing $\mathfrak{b}(\lambda, (x : A), t)$ the more common form is used: $\lambda\, x :$

$$
\begin{aligned}
FV(x) &= \{x\} \\
FV(\mathfrak{b}(\kappa_1, x : t_1, t_2)) &= FV(t_1) \cup (FV(t_2) - \{x\}) \\
FV(\mathfrak{c}(\kappa_2, t_1, \ldots, t_{\mathfrak{a}(\kappa_2)})) &= FV(t_1) \cup \cdots \cup FV(t_{\mathfrak{a}(\kappa_2)})
\end{aligned}
$$

$$
\begin{aligned}
[y := t]x &= x \\
[y := t]y &= t \\
[y := t]\mathfrak{b}(\kappa_1, x : t_1, t_2) &= \mathfrak{b}(\kappa_1, x : [y := t]t_1, [y := t]t_2) \\
[y := t]\mathfrak{c}(\kappa_2, t_1, \ldots, t_{\mathfrak{a}(\kappa_2)}) &= \mathfrak{c}(\kappa_2, [y := t]t_1, \ldots, [y := t]t_{\mathfrak{a}(\kappa_2)})
\end{aligned}
$$

Figure 1.2: Operations on syntax for System F^{ω}, including computing free variables and susbtitution.

$A.\, t$. Whenever the tag for a particular syntactic form is known the shorthand will always be used instead.

Free variables of syntax is defined by a straightforward recursion that collects variables that are not bound in a set. Likewise, substitution is recursively defined by searching through subterms and replacing the associated free variable with the desired term. See Figure 1.2 for the definitions of substitution and computing free variables. However, there are issues with variable renaming that must be solved. A syntactic form is renamed by consistently replacing bound and free variables such that there is no variable capture. For example, the syntax $\lambda\, x : A.\, y\ x$ cannot be renamed to $\lambda\, y : A.\, y\ y$ because it captures the free variable y with the binder λ. More critically, variable capture changes the meaning of a term. There are several rigorous ways to solve variable renaming including (non-exhaustively): De Bruijn indices (or levels) [13], locally-nameless representations [7], nominal sets [37], locally-nameless sets [38], etc. All techniques incorporate some method of representing syntax uniquely with respect to renaming. For this work the variable bureaucracy will be dispensed with. It will be assumed that renaming is implicitly applied whenever necessary to maintain the meaning of a term. For example, $\lambda\, x : A.\, y\ x = \lambda\, z : A.\, y\ z$ and the substitution $[x := t]\lambda\, x : A.\, y\ x$ unfolds to $\lambda\, x : [x := t]A.\, [z := t](y\ x)$.

$$\frac{t_1 \rightsquigarrow t_1'}{\mathfrak{b}(\kappa, x : t_1, t_2) \rightsquigarrow \mathfrak{b}(\kappa, x : t_1', t_2)} \qquad \frac{t_2 \rightsquigarrow t_2'}{\mathfrak{b}(\kappa, x : t_1, t_2) \rightsquigarrow \mathfrak{b}(\kappa, x : t_1, t_2')}$$

$$\frac{t_i \rightsquigarrow t_i' \qquad i \in 1, \ldots, \mathfrak{a}(\kappa)}{\mathfrak{c}(\kappa, t_1, \ldots t_i, \ldots t_{\mathfrak{a}(\kappa)}) \rightsquigarrow \mathfrak{c}(\kappa, t_1, \ldots t_i', \ldots t_{\mathfrak{a}(\kappa)})}$$

$$(\lambda\, x{:}A.\, b)\ t \rightsquigarrow [x := t]b$$

Figure 1.3: Reduction rules for System F^ω.

The syntax of F^ω has a well understood notion of reduction (or dynamics, or computation) defined in Figure 1.3. This is an *inductive* definition of a two-argument relation on terms. A given rule of the definition is represented by a collection of premises $(P_1, \ldots, P_n)$ written above the horizontal line and a conclusion (C) written below the line. An optional name for the rule (Example) appears to the right of the horizontal line. An inductive definition induces a structural induction principle allowing reasoning by cases on the rules and applying the induction hypothesis on the premises. During inductive proofs it is convenient to name the derivation of a premise $(\mathcal{D}_1, \ldots, \mathcal{D}_n)$. Moreover, to minimize clutter during proofs the name of the rule is removed.

$$\frac{P_1 \qquad \ldots \qquad P_n}{C}\ \textsc{Example} \qquad \frac{\overset{\mathcal{D}_1}{P_1} \qquad \ldots \qquad \overset{\mathcal{D}_n}{P_n}}{C}$$

Inductive definitions build a finite tree of rule applications concluding with axioms (or leafs). Axioms are written without premises and optionally include the horizontal line. The reduction relation for F^ω consists of three rules and one axiom. Relations defined in this manner are always the *least* relation that satisfies the definition. In other words, any related terms must have a corresponding inductive tree witnessing the relation.

The reduction relation (or step relation) models function application anywhere in a term via its axiom, called the β-rule. This relation is antisymmetric.

$$\frac{}{t\ R^*\ t}\ \textsc{Reflexive} \qquad\qquad \frac{t\ R\ t' \qquad t'\ R^*\ t''}{t\ R^*\ t''}\ \textsc{Transitive}$$

Figure 1.4: Reflexive-transitive closure of a relation R.

There is a *source* term s and a *target* term t, $s \rightsquigarrow t$, where t is the result of one function evaluation in s. Alternatively, $s \rightsquigarrow t$ is read as s *steps* to t. Note that if there is no λ-term applied to an argument (i.e. no function ready to be evaluated) for a given term t then that term cannot be the source term in the reduction relation. A term that cannot be a source is called a *value*. If there exists some sequence of terms related by reduction that end with a value, then all source terms in the sequence are *normalizing*. If *all* possible sequences of related terms end with a value for a particular source term s, then s is *strongly normalizing*. Restricting the set of terms to a normalizing subset is critical to achieve decidability of the reduction relation.

For any relation $-R-$, the reflexive-transitive closure ($-R^*-$) is inductively defined with two rules as shown in Figure 1.4. In the case of the step relation the reflexive-transitive closure, $s \rightsquigarrow^* t$, is called the *multistep relation*. Additionally, when $s \rightsquigarrow^* t$ then s *multisteps* to t. It is easy to show that any reflexive-transitive closure is itself transitive.

Lemma 1.1. *Let R be a relation on a set A and let $a, b, c \in A$. If $a\ R^*\ b$ and $b\ R^*\ c$ then $a\ R^*\ c$.*

Proof. By induction on $a\ R^*\ b$.

Case: $\dfrac{}{t\ R^*\ t}$

It must be the case the $a = b$.

Case: $\dfrac{\overset{\mathcal{D}_1}{t\ R\ t'} \qquad \overset{\mathcal{D}_2}{t'\ R^*\ t''}}{t\ R^*\ t''}$

Let $z = t'$, then we have $a\ R\ z$ and $z\ R^*\ b$. By the inductive hypothesis (IH) we have $z\ R^*\ c$ and by the transitive rule we have $a\ R^*\ c$ as desired.

□

Two terms are *convertible*, written $t_1 \equiv t_2$, if $\exists\ t'$ such that $t_1 \rightsquigarrow^* t'$ and $t_2 \rightsquigarrow^* t'$. Note that this is not the only way to define convertibility in a type theory, but it is the standard method for a PTS. Convertibility is used in the typing rules to allow syntax forms to have continued valid types as terms reduce. It may be tempting to view conversion as the reflexive-symmetric-transitive closure of the step relation, but transitivity is not an obvious property. In fact, proving transitivity of conversion is often a significant effort, beginning with the confluence lemma.

Lemma 1.2 (Confluence)**.** *If $s \rightsquigarrow^* t_1$ and $s \rightsquigarrow^* t_2$ then there exists a t' such that $t_1 \rightsquigarrow^* t'$ and $t_2 \rightsquigarrow^* t'$.*

Proof. See Appendix A for a proof of confluence involving a larger reduction relation. Note that F^ω's step relation is a subset of this relation and thus is confluent. □

Theorem 1.3 (Transitivity of Conversion)**.** *If $a \equiv b$ and $b \equiv c$ then $a \equiv c$*

Proof. By premises we know $\exists u, v$ such that $a \rightsquigarrow^* u$, $b \rightsquigarrow^* u$, $b \rightsquigarrow^* v$, and $c \rightsquigarrow^* v$. By confluence, $\exists z$ such that $u \rightsquigarrow^* z$ and $v \rightsquigarrow^* z$. By transitivity of multistep reduction, $a \rightsquigarrow^* z$ and $c \rightsquigarrow^* z$. Therefore, $a \equiv c$. □

Figure 1.5 defines the typing relation on terms for F^ω. As previously mentioned this formulation is different from standard presentations. Four relations are defined mutually:

1. $\Gamma \vdash t \rhd T$, to be read as T is the inferred type of the term t in the context Γ or, t infers T in Γ;

2. $\Gamma \vdash t \blacktriangleright T$, to be read as T is the inferred type, possibly after some reduction, of the term t in the context Γ or, t reduction-infers T in Γ;

$$\frac{\Gamma \vdash t \rhd A \qquad A \rightsquigarrow^* B}{\Gamma \vdash t \blacktriangleright B}\ \textsc{RedInf} \qquad \frac{\Gamma \vdash t \rhd A \qquad \Gamma \vdash B \blacktriangleright K \qquad A \equiv B}{\Gamma \vdash t \lhd B}\ \textsc{Chk}$$

$$\frac{}{\vdash \varepsilon}\ \textsc{CtxEm} \qquad \frac{x \notin \mathrm{FV}(\Gamma) \qquad \vdash \Gamma \qquad \Gamma \vdash A \blacktriangleright K}{\vdash \Gamma, x : A}\ \textsc{CtxApp}$$

$$\frac{\vdash \Gamma}{\Gamma \vdash \star \rhd \Box}\ \textsc{Axiom} \qquad \frac{\vdash \Gamma \qquad (x : A) \in \Gamma}{\Gamma \vdash x \rhd A}\ \textsc{Var}$$

$$\frac{\Gamma \vdash A \blacktriangleright \Box \qquad \Gamma, x : A \vdash B \blacktriangleright \Box}{\Gamma \vdash (x : A) \to B \rhd \Box}\ \textsc{Pi1} \qquad \frac{\Gamma \vdash A \blacktriangleright K \qquad \Gamma, x : A \vdash B \blacktriangleright \star}{\Gamma \vdash (x : A) \to B \rhd \star}\ \textsc{Pi2}$$

$$\frac{\Gamma \vdash (x : A) \to B \blacktriangleright K \qquad \Gamma, x : A \vdash t \rhd B}{\Gamma \vdash \lambda\, x : A.\, t \rhd (x : A) \to B}\ \textsc{Lam} \qquad \frac{\Gamma \vdash f \blacktriangleright (x : A) \to B \qquad \Gamma \vdash a \lhd A}{\Gamma \vdash f\ a \rhd [x := a]B}\ \textsc{App}$$

Figure 1.5: Typing rules for System F^ω. The variable K is a metavariable representing either $\star$ or $\Box$.

3. $\Gamma \vdash t \lhd T$, to be read as T is checked against the inferred type of the term t in the context Γ or, t checks against T in Γ;

4. $\vdash \Gamma$, to be read as the context Γ is well-formed, and thus consists only of types that themselves have a type

Note that there are two Pi rules that restrict the domain and codomain pairs of function types to three possibilities: $(\Box, \Box)$, $(\star, \star)$, and $(\Box, \star)$. This is exactly what is required by the λ-cube for this definition to be F^ω. For the unfamiliar reading these rules is arcane, thus exposition explaining a small selected set is provided.

$$\frac{\vdash \Gamma}{\Gamma \vdash \star \rhd \Box}\ \textsc{Axiom}$$

The axiom rule has one premise, requiring that the context is well-formed. It concludes that the constant term $\star$ has type $\Box$. Intuitively, the term $\star$ should be viewed as a universe of types, or a type of types, often referred to as a *kind*. Likewise, the term $\Box$ should be viewed as a universe of kinds, or a kind of kinds. An alternative idea would be to

change the conclusion to $\Gamma \vdash \star \rhd \star$. This is called the *type-in-type* rule, and it causes the type theory to be inconsistent [22, 25]. Note that there is no way to determine a type for $\square$. It plays the role of a type only.

$$\frac{\vdash \Gamma \qquad (x : A) \in \Gamma}{\Gamma \vdash x \rhd A}\ \textsc{Var}$$

The variable rule is a context lookup. It scans the context to determine if the variable is anywhere in context and then the associated type is what that variable infers. This rule is what requires the typing relation to mention a context. Whenever a type is inferred or checked it is always desired that the context is well-formed. That is why the variable rule also requires the context to be well-formed as a premise, because it is a leaf relative to the inference relation. Without this additional premise there could be typed terms in ill-formed contexts.

$$\frac{\Gamma \vdash f \blacktriangleright (x : A) \to B \qquad \Gamma \vdash a \lhd A}{\Gamma \vdash f\ a \rhd [x := a]B}\ \textsc{App}$$

The application rule infers the type of the term f and reduces that type until it looks like a function-type. Once a function type is required it is clear that the type of the term a must match the function-type's argument-type. Thus, a is checked against the type A. Finally, the inferred result of the application is the codomain of the function-type B with the term a substituted for any free occurrences of x in B. This substitution is necessary because this application could be a type application to a type function. For example, let $f = \lambda\, X : \star.\, \mathrm{id}\ X$ where id is the identity term. The inferred type of f is then $(X : \star) \to X \to X$. Let $a = \mathbb{N}$ (any type constant), then $f\ \mathbb{N} \rhd [X := \mathbb{N}](X \to X)$ or $f\ \mathbb{N} \rhd \mathbb{N} \to \mathbb{N}$.

While this presentation of F^ω is not standard Lennon-Bertrand demonstrated that it is equivalent to the standard formulation [27]. In fact, Lennon-Bertrand showed that a similar formulation is logically equivalent for the stronger CIC. Thus, standard metatheoretical results such as preservation and strong normalization still hold.

Lemma 1.4 (Preservation of F^ω)**.** *If* $\Gamma \vdash s \lhd T$ *and* $s \rightsquigarrow^* t$ *then* $\Gamma \vdash t \lhd T$

Proof. See Appendix B for a proof of preservation of a conservative extension of F^ω, and thus a proof of preservation for F^ω itself. □

Theorem 1.5 (Strong Normalization of F^ω)**.** *If* $\Gamma \vdash t \triangleright T$ *then* t *and* T *are strongly normalizing*

Proof. System F^ω is a subsystem of CC which has several proofs of strong normalization. See (non-exhaustively) proofs using saturated sets [19], model theory [44], realizability [33], etc. □

With strong normalization the convertibility relation is decidable, and moreover, type checking is decidable. Let *red* be a function that reduces its input until it is either $\star$, $\Box$, a binder, or in normal form. Note that this function is defined easily by applying the outermost reduction and matching on the resulting term. Let *conv* test the convertibility of two terms. Note that this function may be defined by reducing both terms to normal forms and comparing them for syntactic identity. Both functions are well-defined because F^ω is strongly normalizing. Then the functions *infer*, *check*, and *wf* can be mutually defined by following the typing rules. Thus, type inference and type checking is decidable for F^ω.

While it is true that F^ω only has function types as primitives several other data types are internally derivable using function types. For example, the type of natural numbers is defined:

$$\mathbb{N} = (X : \star) \to X \to (X \to X) \to X$$

Likewise, pairs and sum types are defined:

$$A \times B = (X : \star) \to (A \to B \to X) \to X$$

$$A + B = (X : \star) \to ((A \to X) \times (B \to X)) \to X$$

The logical constants true and false are defined:

$$\top = (X : \star) \to X \to X$$

$$\bot = (X : \star) \to X$$

Negation is defined as implying false:

$$\neg A = A \to \bot$$

These definitions are called *Church encodings* and originate from Church's initial encodings of data in the λ-calculus [9, 10]. Note that if there existed a term such that $\vdash t \lhd \bot$ then trivially for *any* type T we have $\vdash t\ T \lhd T$. Thus, $\bot$ is both the constant false and the proposition representing the principle of explosion from logic. Moreover, this allows a concise statement of the consistency of F^ω.

Theorem 1.6 (Consistency of System F^ω)**.** *There is no term t such that* $\vdash t \lhd \bot$

Proof. Suppose $\vdash t \lhd \bot$. Let n be the value of t after it is normalized. By preservation $\vdash n \lhd \bot$. Deconstructing the checking judgment we know that $\vdash n \rhd T$ and $T \equiv \bot$, but $\bot$ is a value and values like n infer types that are also values. Thus, $T = \bot$ and we know that $\vdash n \rhd \bot$. By inversion on the typing rules $n = \lambda\, X : \star.\, b$, and we have $X : \star \vdash b \rhd X$. The term b can only be $\star$, $\Box$, or X, but none of these options infer type X. Therefore, there does not exist a term b, nor a term n, nor a term t. □

Recall that induction principles cannot be derived internally for any encoding of data [20]. This is not only cumbersome but unsatisfactory as the natural numbers are in their essence the least set satisfying induction. Ultimately, the issue is that these encodings are too general. They admit theoretical elements that F^ω is not flexible enough to express nor strong enough to exclude.

1.2 Calculus of Constructions and Cedille

As previously mentioned, CC is one extension away from F^ω on the λ-cube. Indeed, the two rules Pi1 and Pi2 can be merged to form CC:

$$\frac{\Gamma \vdash A \blacktriangleright K_1 \qquad \Gamma, x : A \vdash B \blacktriangleright K_2}{\Gamma \vdash (x : A) \to B \rhd K_2}\ \textsc{Pi}$$

where now both K_1 and K_2 are metavariables representing either $\star$ or $\Box$. Note that no other rules, syntax, or reductions need to be changed. Replacing Pi1 and Pi2 with this new Pi rule is enough to obtain a complete and faithful definition of CC.

With this merger types are allowed to depend on terms. From a logical point of view, this is a quantification over terms in formula. Hence, why CC is a predicate logic instead of a propositional one according to the Curry-Howard correspondence. Yet, there is a question about what exactly quantification over terms means. Surely it does not mean quantification over syntactic forms.

It means, at minimum, quantification over well-typed terms, but from a logical perspective these terms correspond to proofs. In first order predicate logic the domain of quantification ranges over a set of *individuals*. The set of individuals represents any potential set of interest with specific individuals identified through predicates expressing their properties. With proofs the situation is different. A proof has meaning relative to its formula, but this meaning may not be relevant as an individual in predicate logic. For example, the proof 2 for a Church encoded natural number is intuitively data, but a proof that 2 is even is intuitively not. In CC, both are merely proofs that can be quantified over.

Cedille alters the domain of quantification from proofs to (untyped) λ-caluclus terms. Thus, for Cedille, the proof 2 becomes the encoding of 2 and the proof that 2 is even can *also* be the encoding of 2. This is achieved through a notion of *erasure* which removes type information and auxiliary syntactic forms from a term. Additionally, convertibility is modified to be convertibility of λ-calculus terms. However, erasure as it is defined in Cedille enables diverging terms in inconsistent contexts. The result by Abel and Coquand, which applies to a wide range of type theories including Cedille, is one way to construct a diverging term [1].

If terms are able to diverge, in what sense are they a proof? What a proof is or is not is difficult to say. As early as Aristotle there are documented forms of argument, Aristotle's syllogisms [3]. More than a millennium later Euclid's *Elements* is the most well-known example of a mathematical text containing what a modern audience would call proofs. Moreover, visual renditions of *Elements*, initiated by Byrne, challenge the notion of a proof being an algebraic object [6]. However, the study of proof as a mathematical object dates first to Frege [16] followed soon after by Peano's formalism of arithmetic [35] and Whitehead and Russell's *Principia Mathematica* [46]. For the kinds of logics

discussed by the Curry-Howard correspondence, structural proof theories, the originator is Gentzen [17, 18]. Gentzen's natural deduction describes proofs as finite trees labelled by rules. Note that this is, of course, a very brief history of mathematical proof.

All of these formulations may be justified as acceptable notions of proof, but the purpose of proof from an epistemological perspective is to provide justification. It is unsatisfactory to have a claimed proof and be unable to check that it is constructed only by the rules of the proof theory. This is the situation with Cedille, although rare, there are terms where reduction diverges making it impossible to check a type. However, it is unfair to levy this criticism against Cedille alone, as well-known type theories also lack decidability of type checking. For example, Nuprl with its equality reflection rule [2], and the proof assistant Lean with its notion of casts [32]. Moreover, Lean has been incredibly successful in formalizing research mathematics including the Liquid Tensor Experiment [28] and Tao's formalization of The Polynomial Freiman-Ruzsa Conjecture [45]. Indeed, not having decidability of type checking does to necessarily prevent a tool from producing convincing arguments.

Ultimately, the definition of proof is a philosophical one with no absolute answer, but this work will follow Gentzen and Kreisel in requiring that a proof is a finite tree, labelled by rules, supporting decidable proof checking. The reader need only asks themselves which proof they would prefer if the option was available: one that potentially diverges, or one that definitely does not. If it is the latter, then striving for decidable type theories that are capable enough to reproduce the results obtained by proof assistants like Lean is a worthy goal.

1.3 Thesis

Cedille is a powerful type theory capable of deriving inductive data with relatively modest extension and modification to CC. However, this capability comes at the cost of decidability of type checking and thus, in the opinion of Kreisel, the cost of a Curry-Howard correspondence to a proof theory. A redesign of Cedille that focuses on maintaining a proof-theoretic view recovers

decidability of type checking while still solving the original goals of Cedille. Although this redesign does prevent some constructions from being possible, the new balance struck between capability and complexity is desirable because of a well-behaved metatheory.

1.4 Contributions

Chapter 2 defines the Cedille2 Core (CC2) theory, including its syntax, and typing rules. Erasure from Cedille is rephrased as a projection from proofs to objects. Basic metatheoretical results are proven including: confluence, preservation, and classification.

Chapter 3 models CC2 in F^{ω} obtaining a strong normalization result for proof normalization. This model is a straightforward extension of a similar model for CC. Critically, proof normalization is not powerful enough to show consistency nor object normalization. Additionally, CC2 is shown to be a conservative extension of F^{ω}.

Chapter 4 models CC2 in CDLE obtaining consistency for CC2. Although CDLE is not strongly normalizing it still possess a realizability model which justifies its logical consistency. CC2 is closely related to CDLE which makes this models straightforward to accomplish. Moreover, a selection of axioms added to CC2 is shown to recover much of CDLEs features.

Chapter 5 proves object normalization from proof normalization and consistency. The φ, or cast, rule is the only difficulty after proof normalization and consistency. However, any proof can be translated into a new proof that contains no cast rules. Applying this observation yields an argument to obtain full object normalization.

Chapter 6 with normalization for both proofs and objects a well-founded type checker is defined. This implementation leverages normalization-by-evaluation and other basic techniques like pattern-based unification. The tool it benchmarked to demonstrate reasonable performance.

Chapter 7 contains derivations of generic inductive data, quotient types, large eliminations, constructor subtyping, and inductive-inductive data. All

of these constructions are possible in Cedille but require modest modifications to derive in Cedille2.

Chapter 8 concludes with a collection of open conjectures and questions. Cedille2 at the conclusion of this work is still in its infancy.

Chapter 2

THEORY DESCRIPTION AND BASIC METATHEORY

The theory described in this chapter is a variation of the core theory of Cedille [43]. It is closely related with the significant differences occurring with the equality type. This variation has two primary goals. First, to have decidable type checking (and thus decidable conversion checking). Second, to retain as many constructions as possible from Cedille. This chapter focuses on the description of the theory and some basic metatheory. By basic, we mean properties that are provable by induction on the various derivations or are otherwise provable using straightforward methods.

Syntax for the theory is described in Figure 2.1. Unlike other presentations a generic syntax tree is used with a tag to indicate different syntactic forms. There are three basic syntactic constructs: variables, binders, and constructors. A generic presentation enables occasional economic benefits in presenting other derivations. However, a more standard syntax is defined in terms of the generic one. The specific syntactic forms and the generic forms are used interchangeably whichever is more convenient.

Formally, syntax is worked with as a locally nameless set following the axioms of Pitts [**2023pitts˙lns**]. For the sake of presentation these details are elided. This means that freshness of variables and capture avoiding substitution are largely taken for granted in the exposition of the theory. Moreover, identity of syntactic terms is assumed to be alpha equivalence. Meaning that, again, bureaucracy around variables is taken for granted. Thus, substitution is defined simply:

$$[x := v]y = v \text{ if } x = y$$
$$[x := v]y = y \text{ if } x \neq y$$
$$[x := v]\mathfrak{b}(\kappa_1, x : t_1, t_2) = \mathfrak{b}(\kappa_1, x : [x := v]t_1, [x := v]t_2)$$
$$[x := v]\mathfrak{c}(\kappa_2, t_1, \ldots, t_{\mathfrak{a}(\kappa_2)}) = \mathfrak{c}(\kappa_2, [x := v]t_1, \ldots, [x := v]t_{\mathfrak{a}(\kappa_2)})$$

$$
\begin{aligned}
t &::= x \mid \mathfrak{b}(\kappa_1, x : t_1, t_2) \mid \mathfrak{c}(\kappa_2, t_1, \ldots, t_{\mathfrak{a}(\kappa_2)}) \\
\kappa_1 &::= \lambda_m \mid \Pi_m \mid \cap \\
\kappa_2 &::= \star \mid \Box \mid \bullet_m \mid \mathrm{pair} \mid \mathrm{proj}_1 \mid \mathrm{proj}_2 \mid \mathrm{eq} \mid \mathrm{refl} \mid J \mid \vartheta \mid \delta \mid \phi \\
m &::= \omega \mid 0 \mid \tau \\
&\mathfrak{a}(\star) = \mathfrak{a}(\Box) = 0 \\
&\mathfrak{a}(\mathrm{proj}_1) = \mathfrak{a}(\mathrm{proj}_2) = \mathfrak{a}(\mathrm{refl}) = \mathfrak{a}(\vartheta) = \mathfrak{a}(\delta) = 1 \\
&\mathfrak{a}(\bullet_m) = 2 \\
&\mathfrak{a}(\mathrm{pair}) = \mathfrak{a}(\mathrm{eq}) = \mathfrak{a}(\varphi) = 3 \\
&\mathfrak{a}(J) = 6
\end{aligned}
$$

$$
\begin{aligned}
\star &:= \mathfrak{c}(\star) & t.1 &:= \mathfrak{c}(\mathrm{proj}_1, t) \\
\Box &:= \mathfrak{c}(\Box) & t.2 &:= \mathfrak{c}(\mathrm{proj}_2, t) \\
\lambda_m\, x{:}t_1.\, t_2 &:= \mathfrak{b}(\lambda_m, x : t_1, t_2) & t_1 =_{t_2} t_3 &:= \mathfrak{c}(\mathrm{eq}, t_1, t_2, t_3) \\
(x : t_1) \to_m t_2 &:= \mathfrak{b}(\Pi_m, x : t_1, t_2) & \mathrm{refl}(t) &:= \mathfrak{c}(\mathrm{refl}, t) \\
(x : t_1) \cap t_2 &:= \mathfrak{b}(\cap, x : t_1, t_2) & \vartheta(t) &:= \mathfrak{c}(\vartheta, t) \\
t_1 \bullet_m t_2 &:= \mathfrak{c}(\bullet_m, t_1, t_2) & \delta(t) &:= \mathfrak{c}(\delta, t) \\
[t_1, t_2, t_3] &:= \mathfrak{c}(\mathrm{pair}, t_1, t_2, t_3) & \varphi(t) &:= \mathfrak{c}(\varphi, t)
\end{aligned}
$$

$$
J(t_1, t_2, t_3, t_4, t_5, t_6) := \mathfrak{c}(J, t_1, t_2, t_3, t_4, t_5, t_6)
$$

Figure 2.1: Generic syntax, there are three constructors, variables, a generic binder, and a generic non-binder. Each are parameterized with a constant tag to specialize to a particular syntactic consruct. The non-binder constructor has a vector of subterms determined by an arity function computed on tags. Standard syntactic constructors are defined in terms of the generic forms.

$$
\begin{aligned}
|x| &= x & |f \bullet_\tau a| &= |f| \bullet_\tau |a| \\
|\star| &= \star & |[t_1, t_2, T]| &= |t_1| \\
|\Box| &= \Box & |t.1| &= |t| \\
|\lambda_0\, x\!:\!A.\, t| &= |t| & |t.2| &= |t| \\
|\lambda_\omega\, x\!:\!A.\, t| &= \lambda_\omega\, x.\, |t| & |x =_A y| &= |x| =_{|A|} |y| \\
|\lambda_\tau\, x\!:\!A.\, t| &= \lambda_\tau\, x\!:\!|A|.\, |t| & |\mathrm{refl}(t)| &= \lambda_\omega\, x.\, x \\
|(x : A) \to_m B| &= (x : |A|) \to_m |B| & |J(A, P, x, y, e, w)| &= |e| \bullet_\omega |w| \\
|(x : A) \cap B| &= (x : |A|) \cap |B| & |\vartheta(e)| &= |e| \\
|f \bullet_0 a| &= |f| & |\delta(e)| &= |e| \\
|f \bullet_\omega a| &= |f| \bullet_\omega |a| & |\varphi(f, e)| &= \lambda_\omega\, x.\, x
\end{aligned}
$$

Figure 2.2: Erasure of syntax, for type-like and kind-like syntax erasure is homomorphic, for term-like syntax erasure reduces to the untyped lambda calculus.

$$
\frac{t_1 \rightsquigarrow t_1'}{\mathfrak{b}(\kappa, x : t_1, t_2) \rightsquigarrow \mathfrak{b}(\kappa, x : t_1', t_2)} \qquad \frac{t_2 \rightsquigarrow t_2'}{\mathfrak{b}(\kappa, x : t_1, t_2) \rightsquigarrow \mathfrak{b}(\kappa, x : t_1, t_2')}
$$

$$
\frac{t_i \rightsquigarrow t_i' \qquad i \in 1, \ldots, \mathfrak{a}(\kappa)}{\mathfrak{c}(\kappa, t_1, \ldots t_i, \ldots t_{\mathfrak{a}(\kappa)}) \rightsquigarrow \mathfrak{c}(\kappa, t_1, \ldots t_i', \ldots t_{\mathfrak{a}(\kappa)})}
$$

$$
\begin{aligned}
(\lambda_m\, x\!:\!A.\, b) \bullet_m t &\rightsquigarrow [x := t]b \\
[t_1, t_2, A].1 &\rightsquigarrow t_1 \\
[t_1, t_2, A].2 &\rightsquigarrow t_2 \\
J(A, P, x, y, \mathrm{refl}(z), w) &\rightsquigarrow w \bullet_0 z \\
\vartheta(\mathrm{refl}(t.1)) &\rightsquigarrow \mathrm{refl}(t) \\
\vartheta(\mathrm{refl}(t.2)) &\rightsquigarrow \mathrm{refl}(t)
\end{aligned}
$$

Figure 2.3: Reduction rules for arbitrary syntax.

$$\begin{aligned} \mathrm{dom}_\Pi(\omega, K) &= \star & \mathrm{codom}_\Pi(\omega) &= \star \\ \mathrm{dom}_\Pi(\tau, K) &= K & \mathrm{codom}_\Pi(\tau) &= \square \\ \mathrm{dom}_\Pi(0, K) &= K & \mathrm{codom}_\Pi(0) &= \star \end{aligned}$$

Figure 2.4: Domain and codomains for function types. The variable K is either $\star$ or $\square$.

$$\frac{\vdash \Gamma}{\Gamma \vdash \star \rhd \square}\ \textsc{Axiom} \qquad \frac{\vdash \Gamma \qquad (x : A) \in \Gamma}{\Gamma \vdash x \rhd A}\ \textsc{Var}$$

$$\frac{\Gamma \vdash t \rhd A \qquad A \rightsquigarrow^* B}{\Gamma \vdash t \blacktriangleright B}\ \textsc{RedInf} \qquad \frac{\Gamma \vdash t \rhd A \qquad \Gamma \vdash B \blacktriangleright K \qquad A \equiv B}{\Gamma \vdash t \lhd B}\ \textsc{Chk}$$

$$\frac{}{\vdash \varepsilon}\ \textsc{CtxEm} \qquad \frac{\vdash \Gamma \qquad \Gamma \vdash A \blacktriangleright K \qquad x \notin \mathrm{FV}(\Gamma)}{\vdash \Gamma, x : A}\ \textsc{CtxApp}$$

$$\frac{\Gamma \vdash A \blacktriangleright \mathrm{dom}_\Pi(m, K) \qquad \Gamma, x : A \vdash B \blacktriangleright \mathrm{codom}_\Pi(m)}{\Gamma \vdash (x : A) \to_m B \rhd \mathrm{codom}_\Pi(m)}\ \textsc{Pi}$$

$$\frac{\Gamma \vdash A \blacktriangleright \mathrm{dom}_\Pi(m, K) \qquad \Gamma, x : A \vdash t \rhd B \qquad x \notin \mathrm{FV}(|t|) \text{ if } m = 0}{\Gamma \vdash \lambda_m\, x{:}A.\, t : (x : A) \to_m B}\ \textsc{Lam}$$

$$\frac{\Gamma \vdash f \blacktriangleright (x : A) \to_m B \qquad \Gamma \vdash a \lhd A}{\Gamma \vdash f \bullet_m a \rhd [x := a]B}\ \textsc{App}$$

Figure 2.5: Inference rules for function types, including erased functions. The variable K is either $\star$ or $\square$.

$$\frac{\Gamma \vdash A \blacktriangleright \star \qquad \Gamma, x : A \vdash B \blacktriangleright \star}{\Gamma \vdash (x : A) \cap B \rhd \star}\ \textsc{Int} \qquad \frac{\Gamma \vdash T \blacktriangleright (x : A) \to_\tau B \qquad \Gamma \vdash t \lhd A \qquad \Gamma \vdash s \lhd [x := t]B \qquad |t| =_\beta |s|}{\Gamma \vdash [t, s; T] \rhd (x : A) \cap B}\ \textsc{Pair}$$

$$\frac{\Gamma \vdash t \blacktriangleright (x : A) \cap B}{\Gamma \vdash t.1 \rhd A}\ \textsc{Fst} \qquad \frac{\Gamma \vdash t \blacktriangleright (x : A) \cap B}{\Gamma \vdash t.2 \rhd [x := t.1]B}\ \textsc{Snd}$$

Figure 2.6: Inference rules for intersection types.

$$\frac{\Gamma \vdash A \blacktriangleright \star \qquad \Gamma \vdash a \lhd A \qquad \Gamma \vdash b \lhd A}{\Gamma \vdash a =_A b \rhd \star}\ \textsc{Eq} \qquad \frac{\Gamma \vdash t \rhd A}{\Gamma \vdash \mathrm{refl}(t) \rhd t =_A t}\ \textsc{Refl}$$

$$\frac{\Gamma \vdash A \blacktriangleright \star \qquad \Gamma \vdash P \lhd (x\ y : A) \to_\tau (e : x =_A y) \to_\tau \star \qquad \Gamma \vdash x \lhd A \qquad \Gamma \vdash y \lhd A \qquad \Gamma \vdash e \lhd x =_A y \qquad \Gamma \vdash w \lhd (a : A) \to_0 P \bullet_\tau a \bullet_\tau a \bullet_\tau \mathrm{refl}(a)}{\Gamma \vdash J(A, P, x, y, e, w) \rhd P \bullet_\tau x \bullet_\tau y \bullet_\tau e}\ \textsc{J}$$

$$\frac{\Gamma \vdash e \blacktriangleright a.i =_T b.j \qquad \Gamma \vdash a \blacktriangleright (x : A) \cap B \qquad \Gamma \vdash b \lhd (x : A) \cap B}{\Gamma \vdash \vartheta(e) \rhd a =_{(x:A)\cap B} b}\ \textsc{Prm}$$

$$\frac{\Gamma \vdash f \blacktriangleright (a : A) \to_\omega (x : A') \cap B \qquad A \equiv A' \qquad \Gamma \vdash e \lhd (a : A) \to_\omega a =_A (f \bullet_\omega a).1 \qquad \mathrm{FV}(|e|) = \varnothing}{\Gamma \vdash \varphi(f, e) \rhd (a : A) \to_\omega (x : A) \cap B}\ \textsc{Cast}$$

$$\frac{\Gamma \vdash e \lhd \mathrm{ctt} =_{\mathrm{cBool}} \mathrm{cff}}{\Gamma \vdash \delta(e) \rhd (X : \star) \to_0 X}\ \textsc{Sep}$$

Figure 2.7: Inference rules for equality types where $\mathrm{cBool} := (X : \star) \to_0 (x : X) \to_\omega (y : X) \to_\omega X$; $\mathrm{ctt} := \lambda_0 X : \star.\, \lambda_\omega x : X.\, \lambda_\omega y : X.\, x$; and $\mathrm{cff} := \lambda_0 X : \star.\, \lambda_\omega x : X.\, \lambda_\omega y : X.\, y$. Also, $i, j \in \{1, 2\}$

Chapter 3

PROOF NORMALIZATION AND RELATIONSHIP TO SYSTEM F^{ω}

Chapter 4

CONSISTENCY AND RELATIONSHIP TO CDLE

Chapter 5

OBJECT NORMALIZATION

A φ_i-proof is a proof that allows i nested φ syntactic constructs. For example, a φ_0-proof allows no φ subterms, a φ_1-proof allows φ subterms but no nested φ subterms, and a φ_2-proof allows φ_1 subterms. Defined inductively, a φ_0-proof is a proof with no φ syntactic constructs and a φ_{i+1}-proof is a proof with φ_i-proof subterms.

For any φ_i-proof p there is a strictification $s(p)$ that is a φ_0-proof in Figure 5.1.

Lemma 5.1 (Strictification Preserves Inference)**.** *Given* $\Gamma \vdash t \rhd A$ *then* $\Gamma \vdash s(t) \rhd A$

Proof. By induction on the typing rule, the φ rule is the only one of interest:

Case:
$$\frac{A \overset{\mathcal{D}_2}{\equiv} A' \qquad \Gamma \vdash e \lhd (a : A) \overset{\mathcal{D}_3}{\to_\omega} a =_A (f \bullet_\omega a).1 \qquad \mathrm{FV}(|e|) \overset{\mathcal{D}_4}{=} \varnothing}{\Gamma \vdash \varphi(f, e) \rhd (a : A) \to_\omega (x : A) \cap B}$$

Need to show that $\Gamma \vdash s(\varphi(a, f, e)) \rhd (x : A) \cap B$ which reduces to: $\Gamma \vdash s(f) \bullet_\omega s(a) \rhd (x : A) \cap B$. By the IH we know that $s(f)$ infers the same function type, and that $s(a)$ infers the same argument type, therefore the application rule concludes the proof.

□

Lemma 5.2 (Strict Proofs are Normalizing)**.** *Given* $\Gamma \vdash t \rhd A$ *then* $s(t)$ *is strongly normalizing*

Proof. Direct consequence of strong normalization of proofs □

$$
\begin{aligned}
s(x) &= x & s([s,t,T]) &= [s(s), s(t), s(T)] \\
s(\star) &= \star & s(t.1) &= s(t).1 \\
s(\Box) &= \Box & s(t.2) &= s(t).2 \\
s(\lambda_m\, x\!:\!A.\, t) &= \lambda_m\, x\!:\!s(A).\, s(t) & s(x =_A y) &= s(x) =_{s(A)} s(y) \\
s((x:A) \to_m B) &= (x : s(A)) \to_m s(B) & s(\mathrm{refl}(t)) &= \mathrm{refl}(s(t)) \\
s((x:A) \cap B) &= (x : s(A)) \cap s(B) & s(\vartheta(e)) &= \vartheta(s(e)) \\
s(f \bullet_m a) &= s(f) \bullet_m s(a) & s(\delta(e)) &= \delta(s(e))
\end{aligned}
$$

$$
\begin{aligned}
s(J(A,P,x,y,r,w)) &= J(s(A), s(P), s(x), s(y), s(r), s(w)) \\
s(\varphi(a,f,e)) &= s(f) \bullet_\omega s(a)
\end{aligned}
$$

Figure 5.1: Strictification of a proof.

Lemma 5.3 (Strict Objects are Normalizing)**.** *Given* $\Gamma \vdash t \rhd A$ *then* $|s(t)|$ *is strongly normalizing*

Proof. Proof Idea:

Proof reduction tracks object reduction in the absence of φ constructs. Thus, the normalization of a proof provides an upper-bound on the number of reductions an object can take to reach a normal form. □

A proof, $\Gamma \vdash t_1 \rhd A$, is contextually equivalent to another proof, $\Gamma \vdash t_2 \rhd A$, if there is no context with hole of type A whose object reduction diverges for t_1 but not t_2. In other words, if a context can be constructed that distinguishes the terms based on their object reduction.

Lemma 5.4. *A* φ_1*-proof,* p*, is contextually equivalent to its strictification,* $s(p)$

Proof. Proof by induction on the typing rule for p, focus on the application rule:

Case: $\dfrac{\Gamma \vdash f \blacktriangleright (x:A) \to_m B \overset{\mathcal{D}_1}{} \qquad \Gamma \vdash a \lhd A \overset{\mathcal{D}_2}{}}{\Gamma \vdash f \bullet_m a \rhd [x := a]B}$

In particular, we care about when $f = \varphi(v, b, e).2$ and $m = \omega$. Note that the first projection has a proof-reduction that yields a which makes it unproblematic.

We know that $s(v) = v$ because f is a φ_1-proof. Let v_n be the normal form of v and note that $|v_n|$ is also normal. Likewise, we have e_n and $|e_n|$ normal.

Suppose there is a context $C[\cdot]$ where $|p|$ diverges but $|s(p)|$ normalizes. (Note that the opposite assumption is impossible). If $|v_n|$ is a variable, then reduction in $|p|$ is blocked (contradiction). Otherwise $|v_n| = \lambda\, x.\, x\ t_1\ \cdots\ t_n$ where t_i are normal.

Now it must be the case that $|e \bullet_\omega v| = |e_n| \bullet_\omega |v_n|$ is normalizing. Thus, we have a refl proof that $v_n = (f \bullet_\omega v_n).1$. (Note, this proof *must* be refl because $\mathrm{FV}(|e|) = \varnothing$). But, this implies convertibility, thus $|v_n| =_\beta |f| \bullet_\omega |v_n|$, but this must mean more concretely that $|f| \bullet_\omega |v_n| \rightsquigarrow |v_n|$. Yet $|f| \bullet_\omega |v_n| \bullet_\omega a$ is strongly normalizing because it is $s(p)$. Therefore, p in this case is strongly normalizing which refutes the assumption yielding a contradiction.

□

Lemma 5.5. *If t_1 is strongly normalizing and contextually equivalent to t_2 then t_2 is strongly normalizing*

Proof. Immediate by the definition of contextual equivalence. □

Theorem 5.6. *A φ_i-proof p is strongly normalizing for all i*

Proof. By induction on i.

Case: $i = 0$

Immediate because $s(p) = p$ and strict proofs are strongly normalizing.

Inductive Case:

Suppose that φ_i-proof is strongly normalizing. Goal: show that φ_{i+1}-proof is strongly normalizing.

□

Chapter 6

CEDILLE2: SYSTEM IMPLEMENTATION

Chapter 7

CEDILLE2: INTERNALLY DERIVABLE CONCEPTS

Chapter 8

CONCLUSION AND FUTURE WORK

Appendix A

PROOF OF CONFLUENCE

Appendix B

PROOF OF PRESERVATION

Hello!

BIBLIOGRAPHY

[1] Andreas Abel and Thierry Coquand. "Failure of normalization in impredicative type theory with proof-irrelevant propositional equality". In: *Logical Methods in Computer Science* 16 (2020).

[2] Stuart F Allen et al. "The Nuprl open logical environment". In: *Automated Deduction-CADE-17: 17th International Conference on Automated Deduction Pittsburgh, PA, USA, June 17-20, 2000. Proceedings 17*. Springer. 2000, pp. 170–176.

[3] Aristotle. *Analytica Priora et Posteriora*. Oxford University Press, 1981. ISBN: 9780198145622.

[4] HENK BARENDREGT. "Introduction to generalized type systems". In: *Journal of Functional Programming* 1.2 (1991), pp. 125–154.

[5] Henk Barendregt and Kees Hemerik. "Types in lambda calculi and programming languages". In: *ESOP'90: 3rd European Symposium on Programming Copenhagen, Denmark, May 15–18, 1990 Proceedings 3*. Springer. 1990, pp. 1–35.

[6] Oliver Byrne. *Oliver Byrne's Elements of Euclid*. Art Meets Science, 2022. ISBN: 978-1528770439.

[7] Arthur Charguéraud. "The locally nameless representation". In: *Journal of automated reasoning* 49 (2012), pp. 363–408.

[8] Alonzo Church. "A formulation of the simple theory of types". In: *The journal of symbolic logic* 5.2 (1940), pp. 56–68.

[9] Alonzo Church. "A set of postulates for the foundation of logic". In: *Annals of mathematics* (1932), pp. 346–366.

[10] Alonzo Church. "A set of postulates for the foundation of logic". In: *Annals of mathematics* (1933), pp. 839–864.

[11] Thierry Coquand. "Une théorie des constructions". PhD thesis. Universiteé Paris VII, 1985.

[12] Thierry Coquand and Gérard Huet. *The calculus of constructions*. Tech. rep. RR-0530. INRIA, May 1986. URL: https://hal.inria.fr/inria-00076024.

[13] Nicolaas Govert De Bruijn. "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem". In: *Indagationes Mathematicae (Proceedings)*. Vol. 75. 5. Elsevier. 1972, pp. 381–392.

[14] Denis Firsov, Richard Blair, and Aaron Stump. "Efficient Mendler-style lambda-encodings in Cedille". In: *Interactive Theorem Proving: 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings 9*. Springer. 2018, pp. 235–252.

[15] Robert W Floyd. "A descriptive language for symbol manipulation". In: *Journal of the ACM (JACM)* 8.4 (1961), pp. 579–584.

[16] Gottlob Frege. "Begriffsschrift, a Formula Language, Modeled upon that of Arithmetic, for Pure Thought [1879]". In: *From Frege to Gödel: A Source Book in Mathematical Logic* 1931 (1879).

[17] Gerhard Gentzen. "Untersuchungen über das logische schließen. I." In: *Mathematische zeitschrift* 35 (1935).

[18] Gerhard Gentzen. "Untersuchungen über das logische Schließen. II." In: *Mathematische zeitschrift* 39 (1935).

[19] Herman Geuvers. "A short and flexible proof of strong normalization for the calculus of constructions". In: *International Workshop on Types for Proofs and Programs*. Springer. 1994, pp. 14–38.

[20] Herman Geuvers. "Induction is not derivable in second order dependent type theory". In: *International Conference on Typed Lambda Calculi and Applications*. Springer. 2001, pp. 166–181.

[21] Herman Geuvers and Mark-Jan Nederhof. "Modular proof of strong normalization for the calculus of constructions". In: *Journal of Functional Programming* 1.2 (1991), pp. 155–189.

[22] Jean-Yves Girard. "Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur". PhD thesis. Universiteé Paris VII, 1972.

[23] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and types*. Vol. 7. Cambridge university press Cambridge, 1989.

[24] William A Howard. "The formulae-as-types notion of construction". In: *To HB Curry: essays on combinatory logic, lambda calculus and formalism* 44 (1980), pp. 479–490.

[25] Antonius JC Hurkens. "A simplification of Girard's paradox". In: *Typed Lambda Calculi and Applications: Second International Conference on Typed Lambda Calculi and Applications, TLCA'95 Edinburgh, United Kingdom, April 10–12, 1995 Proceedings 2*. Springer. 1995, pp. 266–278.

[26] A. Kopylov. "Dependent intersection: a new way of defining records in type theory". In: *18th Annual IEEE Symposium of Logic in Computer Science, 2003. Proceedings.* 2003, pp. 86–95. DOI: 10.1109/LICS.2003.1210048.

[27] Meven Lennon-Bertrand. "Complete Bidirectional Typing for the Calculus of Inductive Constructions". In: *ITP 2021-12th International Conference on Interactive Theorem Proving*. Vol. 193. 24. 2021, pp. 1–19.

[28] *Liquid Tensor Experiment*. https://github.com/leanprover-community/lean-liquid. 2022.

[29] Per Martin-Löf. "An Intuitionistic Theory of Types: Predicative Part". In: *Logic Colloquium '73*. Ed. by H.E. Rose and J.C. Shepherdson. Vol. 80. Studies in Logic and the Foundations of Mathematics. Elsevier, 1975, pp. 73–118. DOI: https://doi.org/10.1016/S0049-237X(08)71945-1.

[30] Per Martin-Löf and Giovanni Sambin. *Intuitionistic type theory*. Vol. 9. Bibliopolis Naples, 1984.

[31] Alexandre Miquel. "The Implicit Calculus of Constructions Extending Pure Type Systems with an Intersection Type Binder and Subtyping". In: *Typed Lambda Calculi and Applications*. Ed. by Samson Abramsky. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 344–359. ISBN: 978-3-540-45413-7.

[32] Leonardo de Moura and Sebastian Ullrich. "The lean 4 theorem prover and programming language". In: *Automated Deduction–CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings 28*. Springer. 2021, pp. 625–635.

[33] C-HL Ong and Eike Ritter. "A generic Strong Normalization argument: application to the Calculus of Constructions". In: *International Workshop on Computer Science Logic*. Springer. 1993, pp. 261–279.

[34] Christine Paulin-Mohring. "Inductive definitions in the system Coq rules and properties". In: *Typed Lambda Calculi and Applications*. Ed. by Marc Bezem and Jan Friso Groote. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 328–345. ISBN: 978-3-540-47586-6.

[35] Giuseppe Peano. *Arithmetices principia: Nova methodo exposita*. Fratres Bocca, 1889.

[36] Frank Pfenning and Christine Paulin-Mohring. "Inductively defined types in the Calculus of Constructions". In: *Mathematical Foundations of Programming Semantics*. Ed. by M. Main et al. New York, NY: Springer-Verlag, 1990, pp. 209–228. ISBN: 978-0-387-34808-7.

[37] Andrew M Pitts. *Nominal sets: Names and symmetry in computer science*. Cambridge University Press, 2013.

[38] Andrew M. Pitts. "Locally Nameless Sets". In: *Proc. ACM Program. Lang.* 7.POPL (2023). DOI: 10.1145/3571210. URL: https://doi.org/10.1145/3571210.

[39] John C Reynolds. "Towards a theory of type structure". In: *Programming Symposium: Proceedings, Colloque sur la Programmation Paris, April 9–11, 1974*. Springer. 1974, pp. 408–425.

[40] John C Reynolds. "Types, abstraction and parametric polymorphism". In: *Information Processing 83, Proceedings of the IFIP 9th World Computer Congres*. 1983, pp. 513–523.

[41] Dana Scott. "Constructive validity". In: *Symposium on automatic demonstration*. Springer. 1970, pp. 237–275.

[42] Aaron Stump. "The calculus of dependent lambda eliminations". In: *Journal of Functional Programming* 27 (2017), e14.

[43] Aaron Stump and Christopher Jenkins. *Syntax and Semantics of Cedille*. 2021. arXiv: 1806.04709 [cs.PL].

[44] Jan Terlouw. "Strong normalization in type systems: A model theoretical approach". In: *Annals of Pure and Applied Logic* 73.1 (1995), pp. 53–78.

[45] *The Polynomial Freiman-Ruzsa Conjecture*. https://github.com/teorth/pfr. 2024.

[46] Alfred North Whitehead and Bertrand Russell. "Principia Mathematica". In: (1927).