Logic Programming and Type Inference with the Calculus of Constructions

Matthew Mirman

CMU-CS-14-110

May 2014

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Frank Pfenning
Karl Crary

Submitted in partial fulfillment of the requirements for the degree of Masters in Computer Science.

Keywords: Logic Programming, Pure Type System, Type Inference, Higher Order Unification, Caledon Language, Higher Order Abstract Syntax, Metaprogramming, Universe Checking

For my grandfather.

Abstract

In this thesis I present a higher order logic programming language, Caledon, with a pure type system and a Turing complete type inference and implicit argument system based on the same logic programming semantics. Because the language has dependent types and type inference, terms can be generated by providing type constraints. I design the dynamic semantics of this language to be the same used to perform type inference, such that there is no disparity between compilation and running. The lack of distinction between compilation and execution permits certain metaprogramming techniques which are normally either unavailable or only possible with second thought extensions. The addition of control structures such as implicit arguments, shared holes, polymorphism, and nondeterminism control makes programming computation during type inference more natural. As a consequence of these extensions, unification problems must be generated to solve for terms in addition to the usual problems generated to solve for types. Furthermore, because every result of execution is a term in the consistent calculus of constructions, Caledon can be considered an interactive theorem prover with a less orthogonal combination of proof search and proof checking than has previously been designed or implemented.

Acknowledgments

I thank my advisor, Frank Pfenning for listening patiently to all my outlandish ideas.

Contents

Chapter 1

Introduction

Higher order logic programming languages such as λProlog represent a significant step toward the development of an effective self-contained language for the writing of programming languages [52]. LF [33] on the other hand was designed as a meta-logical framework which was later used as the basis for the higher order backtracking logic programming language Elf [60]. Both of these approaches, however, have drawbacks. λProlog lacks a type system broad enough to express many invariants for its own code. Elfs type system is so conservative that it lacks polymorphism, a fundamental requirement for building libraries. But, if a dependently-typed logic programming language like Elf is extended with polymorphism, it becomes powerful enough to use the proofs generated by its own proof search as functioning code.

In this thesis we present a logic programming language, Caledon, intended to extend the typed logic languages to general usability and permit higher order programming. Caledon, much like Haskell is not intended to be a theorem proving language, but instead a general purpose language. It is given certain mechanics of a theorem proving language to simplify the process of designing embedded domain specific languages, and for metaprogramming, but these features do not result in a total, consistent language.

In most languages, type inference is employed to generate types for type directed compilation or static analysis. In order to take advantage of a logic programming languages ability to generate its own code elegantly, we modify the purpose of type inference so that it can be considered as the execution of the language. In this way, type inference can be worded as a translation from language terms to unification problems. In this paper, we consider unification problems to be the target language for the compilation of a correctly parsed program. Since the "Calculus of Constructions" [16] contains polymorphism, dependent types, and type functions, it is used as the basis for the source language. While type inference for the "Calculus of Constructions" and even LF is undecidable[28] and thus often abandoned as a goal in its entirety. We, on the other hand, specify the type inference procedure with the same semantics as the language itself, thus making the procedure programmable.

To control the use of inference programmatically, implicit arguments similar to those in Agda are included. As implicit arguments may be filled by proof search in addition to simple unification, our implicit arguments are similar to the type classes of Haskell. These features result in the source terms being changed by type inference.

1.1 Logic Programming

Logic programming languages such as Prolog were originally designed as part of the "AI program", in much the same way Lisp was. Automated reasonings natural goal was to be able to arbitrarily prove theorems. A logic programming language was a set of axioms and a predicate. If the predicate could be proven through those axioms, the automated theorem prover would halt. These proof search procedures were then constrained into useful programming semantics. When performed in a backtracking manner, the proof-search process represented a formulation of procedural code with powerful pattern matching. The Caledon language is a higher-order backtracking logic

programming language in the style of Elf [60]. In this section I present some basic intuition for logic programming, rather than explaining it technically, and demonstrate the descriptive power of the system implemented in Caledon.

1.1.1 Basics

We begin by defining addition on unary numbers in Caledon shown in shown in 1.1.

```
1 defn add : nat -> nat -> nat -> prop
2     | addZ = add zero A A
3     | addS = add (succ A) B (succ C)
4                 <- add A B C
```

Figure 1.1: Addition in Caledon

One might notice that this definition is incredibly similar to its Haskell counterpart shown in 1.2.

```
1 add :: nat -> nat -> nat
2 add Zero a = a
3 add (Succ a) b = Succ c
4     where c = add a b
```

Figure 1.2: Addition in Haskell

We can read the logic programming definition as we would read the functional definition with pattern matching, knowing that an intelligent compiler would be able to convert the first into the second. Search allows one to define essentially nondeterministic programs. A common use for logic programming has been to search for solutions to combinatorial games such as tic-tac-toe, without the programmer worrying about the order of the search. As this tends to produce inefficient code, this use style is dis-

couraged. Rather, a more procedural view of logic programming is encouraged where pattern match and search is performed in the order it appears.

```
1 defn p : T_1 -> ... -> T_r -> prop
2   >| n1 = p T_1 ... T_r <- p_1,1 ... <- p_1,k_1
3 ...
4   >| nN = p T_1 ... T_r <- p_n,1 ... <- p_n,k_n
5
6 query prg = p t1 ... tr
```

Figure 1.3: Format of a Caledon Logic Program

In this view, a program of the form 1.3 should be considered a program which first attempts to prove using axiom n1 by matching prg with "p T1 ... Tr" and then attempting to prove $p_{1,1}$ and so on.

1.1.2 Higher Order Programming

Fortunately, higher order functions need not be restricted to patterns. Definitions provide even more ways to generalize code.

A great example is the function application operator from Haskell. We can define this in Caledon as shown in 1.4

```
1 fixity right 0 @
2 defn @ : (At -> Bt) -> At -> Bt
3   as ?\ At Bt . \ f : At -> Bt . \ a : At . f a
```

Figure 1.4: Definitions for expressive syntax

In many cases, allowing these definitions allows for significant simplification of syntax. The reader familiar with languages like Elf, Haskell, and Agda might notice the implicit abstraction of the type variables At and Bt in the type of @ in 1.4. The rest of

this paper is concerned with formalizing these implicit abstractions and letting them have as much power as possible. For example, one might make these abstractions explicit by instead declaring at the beginning 1.5

```
1 infixr 0 @
2 (@) :: forall At Bt . (At -> Bt) -> At -> Bt
3 (@) = \ f . \ a . f a
```

Figure 1.5: Explicit Haskell style abstractions

However, in a dependently typed language, every function type is a dependent product (forall). This makes it necessary to provide a new (explicit) implicit dependent product - ?∀ or ?Π.

```
1 fixity right 0 @
2 defn @ : {At Bt:prop} (At -> Bt) -> At -> Bt
3   as ?\ At Bt . \ f . \ a . f a
```

Figure 1.6: The (explicit) implicit equivalent of 1.4

Haskell also has type classes. For example, the type of show can be seen in 1.7.

```
1 show :: Show a => a -> String
```

Figure 1.7: The type of show

In Caledon, these can be written similarly as in 1.8

```
1 defn show : showC A => A -> string
2 defn show : {unused : showC A } A -> string
```

Figure 1.8: Equivalent types for show in Caledon

However, since implicit arguments are a natural extension of the dependent type system in Caledon, no restrictions are made on the number of arguments, or difficulty of computing. Unfortunately, since computation is primarily accomplished by the logic programming fragment of the language rather than the functional fragment of the language, the correspondence between these programmable implicit arguments and type classes is not one to one. It is possible to replicate virtually all of the functionality of type classes in the implicit argument system, but the syntax required to do so can become verbose. Rather than attempting to simulate type classes, more creative uses are possible, such as computing the symbolic derivative of a type for use in a (albeit, slow and unnecessary) generic zipper library, or writing programs that compile differently with different types in different environments.

1.2 Initial Examples

In the previous section I gave an introduction to the notion of logic programming using both the familiar language of Haskell and the new language of Caledon. In this section I will build upon these ideas by introducing logic programming with polymorphism through building a set of standard polymorphic type logic library.

There are a few ways of defining sums in Caledon.

```
1 defn and : type -> type -> type
2     | pair = [A B : type] A -> B -> and A B
3
4 defn fst : and A B -> A -> type
5     | fstImp = fst (pair Av Bv) Av
6
7 defn snd : and A B -> B -> type
8     | sndImp = snd (pair Av Bv) Bv
```

Figure 1.9: Logical conjunction

In this first, simplest way (as seen in figure 1.9, we define a predicate for and and predicates for construction and projection. This method has the advantage of doubling as a form of sequential predicate.

```
1
2 query main = and (print ''hello '') (print '' world!'')
```

Figure 1.10: Use of logical conjunction

In the figure 1.10 the query will output “hello world!”.

```
1
2  defn and : type -> type -> type
3    as \ a : type . \ b : type .
4         [ c : type ] (a -> b -> c) -> c
5
6  defn pair : A -> B -> and A B
7    as ?\ A B : type .
8         \ a b .
9         \ c : type .
10        \ proj : A -> B -> c .
11          proj a b
12
13 defn fst : and A B -> A
14   as ?\ A B : type .
15        \ pair : [c : type] (A -> B -> c) -> c .
16          pair A (\ a b . a)
17
18 defn snd : and A B -> A
19   as ?\ A B : type .
20        \ pair : [c : type] (A -> B -> c) -> c .
21          pair B (\ a b . b)
```

Figure 1.11: Church style conjunction

In the case of figure 1.11, we do not add any axioms without their proofs. In this example we also introduce the dependent product written in the form $[a : t_1]t_2$.

This case mimics the version usually seen in the Calculus of Constructions and has the advantage of the projections being functions rather than predicates.

```
1
2  defn churchList : type    > type
3    as \A : type . [ lst : type] lst    > (A    > lst    > lst )    > lst
4
5  defn consCL : [ B : type ] B -> churchList B -> churchList B
6    as \ C : type .
7       \ V : C .
8       \ cl : churchList C .
9       \ lst : type .
10      \ nil : lst .
11      \ cons : C -> lst -> lst .
12      cons V (cl lst nil cons)
13
14 defn nilCL : [ B : type ] churchList B
15   as \ C : type .
16      \ lst : type .
17      \ nil : lst .
18      \ cons : C -> lst -> lst .
19        nil
20
21 defn mapCL : { A B } ( A -> B) -> churchList A -> churchList B
22   as ?\ A B : type .
23       \ F : A -> B.
24       \ cl : churchList A .
25       \ lst : type .
26       \ nil : lst .
27       \ cons : B -> lst -> lst .
28         cl lst nil (\ v . cons (F v))
29
30 defn foldrCL : { A B } ( A -> B -> A) -> A -> churchList B -> A
31   as ?\ A B : type .
32       \ F : A -> B -> A .
33       \ bc : A .
34       \ cl : churchList B .
35         cl A bc (\ v : B . \ c : A . F c v)
```


Figure 1.12: Church style list

In the Church form of a list, folds and maps are possible to implement as functions rather than predicates. However, their implementation is verbose and doesn't permit more complex functions.

```
1
2  defn list : type -> type
3     | nil = list A
4     | cons = A -> list A -> list A
5
6  defn concatList : list A -> list A -> list A -> type
7     | concatListNil = [ L : list A ] concatList nil L L
8     | concatListCons =
9           concatList (cons (V : T) A) B (cons V C)
10       <- concatList A B C
11
12 defn concatList : list A -> list A -> list A -> type
13    | concatListNil = [ L : list A ] concatList nil L L
14    | concatListCons =
15          concatList (cons (V : T) A) B (cons V C)
16       <- concatList A B C
17
18 defn mapList : (A -> B) -> list A -> list B -> type
19    | mapListNil = [ F : A -> B] mapList F nil nil
20    | mapListCons = [F : A -> B]
21            mapList F (cons V L) (cons (F V) L')
22           <- mapList F L L'
23
24 defn pmapList : (A -> B -> type) -> list A -> list B -> type
25    | pmapListNil = [ F : A -> B -> type] pmapList F nil nil
26    | pmapListCons = [F : A -> B -> type]
27            pmapList F (cons V L) (cons V' L')
28            <- F V V'
29            <- mapList F L L'
```

Figure 1.13: Logic List

The logic programming version can be seen in figure 1.13. It is important to note that we can now map a predicate over a list rather than just mapping a function over a list.

Chapter 2

Type System

In this section, I introduce the specifics of the "Caledon Implicit Calculus of Constructions" ($CICC$). The internal Caledon type system is an extension of the well known "Calculus of Constructions" with the addition of implicit bindings and explicit type constraints for implicit instantiation. While the inspiration for this comes from the theorem prover Agda, it appears as though no formal treatment has been provided. This section provides a background on pure type systems and the history of the "Calculus of Constructions" and introduces a formal definition and treatment of $CICC$ with η conversions for the purpose of type checking and proof search.

The type system of Caledon is designed after two different formalisms for working with implicit arguments: the "Bicolored Calculus of Constructions" (CC^{Bi}) [44] and the "Implicit Calculus of Constructions" (ICC) [55].

It is comprised of two parts: The "Caledon Implicit Calculus of Constructions" ($CICC$), and the "Implicit Caledon Implicit Calculus of Constructions" ($CICC^{-}$). $CICC$ is a custom modification of CC^{Bi} with Church-style binders and explicit constraints and externally named binders reminiscent of module type theory [21]. $CICC^{-}$ Is intended to be a combination of the first two, or a partial erasure system for $CICC$.

2.1 Pure Type Systems

The type system for Caledon is a pure type system [45] extended with explicit recursive types and implicit types. In this section, I discuss what a pure type system is and what its properties are.

Pure type systems are generalizations of the lambda cube [5] which allow for arbitrary relationships between terms and types. With proper selection of constants, sorts, axioms, and relations, pure type systems can embed the "Calculus of Constructions" [16] and many other type systems one might want to construct.

As generalizations, these systems are important, as it has been proven by Jutting [40] that type checking for normalizing pure type systems with finite axiom sets are decidable. Thus, by showing how a system is a pure type system and is normalizing, you get decidability of type checking nearly for free.

It has also been shown that these systems have utility. Roorda [70] gave an implementation of a functional programming language with pure type system and demonstrated its utility.

A pure type system is a set S of sorts, $A \subseteq S \times S$ of axioms, and a relation $R \subseteq S \times S \times S$ along with the following grammar and inference rules:

Definition 2.1.1 ***(PTS Syntax)***

$$E ::= V \mid S \mid E\ E \mid \lambda V : E.E \mid \Pi V : E.E$$

Definition 2.1.2 ***(PTS Typing Rules)***

$$\frac{}{\cdot \vdash}\ \mathsf{WF-E} \qquad \frac{\Gamma \vdash T : s \quad x \notin DV(\Gamma)}{\Gamma, x : T \vdash}\ \mathsf{WF-S}$$

$$\frac{\Gamma \vdash \quad (c, s) \in A}{\Gamma \vdash c : s}\ \mathsf{axioms}$$

$$\frac{\Gamma \vdash A : s \quad s \in S}{\Gamma, x : A \vdash x : A} \text{ start}$$

$$\frac{\Gamma \vdash A : B \quad \Gamma \vdash C : s \quad s \in S}{\Gamma, x : C \vdash A : B} \text{ weakening}$$

$$\frac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2 \quad (s_1, s_2, s_3) \in R}{\Gamma \vdash (\Pi x : A.B) : s_3} \text{ product}$$

$$\frac{\Gamma \vdash F : (\Pi x : A.B) \quad \Gamma \vdash V : A \quad \textit{x is free for V in B}}{\Gamma \vdash FV : [V/x]B} \text{ application}$$

$$\frac{\Gamma, x : A \vdash F : B \quad \Gamma \vdash (\Pi x : A.B) : s \quad s \in S}{\Gamma \vdash (\lambda x : A.F) : (\Pi x : A.B)} \text{ abstraction}$$

$$\frac{\Gamma \vdash A : B \quad \Gamma \vdash B \equiv_{\beta\eta\nu *} B' \quad \Gamma \vdash B' : s \quad s \in S}{\Gamma \vdash A : B'} \text{ conversion}$$

As Barendregt [5] points out, the common type theories with only functions can be recast as pure type systems by choice of axioms. In the simplest example, the only axioms chosen are $(*, \Box)$ along with the single relationship $(*, *, *)$. This system describes the simply typed lambda calculus, where only terms can depend on terms. We say that $A \to B \equiv \Pi x : A.B$ iff $x \notin FV(B)$.

Theorem 2.1.3 *Subject Reduction: If* $\Gamma \vdash A : T$ *and* $A \Rightarrow_\beta B$ *then* $\Gamma \vdash B : T$

Geuvers and Nederhof [26] proved subject reduction for any calculus on the λ cube. This property can be proved syntactically by induction on the structure of the typing derivation and there exist Elf and Agda verified proofs of this property. Note that this is a useful property to maintain, even in the face of inconsistency of a system, because at the very least, the property allows for a consistent understanding of typing terms.

Theorem 2.1.4 *Uniqueness of Types: If* $\Gamma \vdash A : T$ *and* $\Gamma \vdash A : T'$ *then* $T \equiv_\beta T'$

The uniqueness of types with respect to β reduction has also been shown for any system on the λ cube. This last property is important to showing the decidability of type inference in the Caledon language without implicits.

Lemma 2.1.5 *Strengthening*

$$\frac{\Gamma, x : T \vdash M : U \quad x \notin FV(M) \cup FV(U)}{\Gamma \vdash M : U} \text{ strength}$$

As it turns out, the strengthening lemma has important implications to the generation of bindings during proof search.

2.2 The Calculus of Constructions

The type system for Caledon is based on the "Calculus of Constructions" as defined by Coquand et al. [16]. Since Caledon might extend the "Calculus of Constructions," it is important to view it as a pure type system. As Barendregt [5] points out, the common type theories can be recast as pure type systems by choice of axioms.

Roorda and Jeuring [70] gave an implementation of a functional programming language with pure type system and demonstrated its utility.

This is the pure type system where.

Definition 2.2.1 ***(PTS for*** CC***)***

$$A = \{*, \Box\} \tag{2.1}$$

$$S = \{(* : \Box)\} \tag{2.2}$$

$$R = \{(*, *, *), (*, \Box, \Box), (\Box, \Box, \Box), (\Box, *, *)\} \tag{2.3}$$

In this system, terms can depend on terms and types, and types can depend on types and terms. This pure type system has the well known strong normalization property,

implying the termination of all lambda terms typeable by CC [25] [26]. It is necessary to be careful with the types of equalities allowed in the conversion rule, since if there are more allowed equalities, then certain proofs become significantly more complex.

Definition 2.2.2 $\Gamma \vdash_{CC} P : T : K$ *means* $\Gamma \vdash_{CC} P : T$ *and* $\Gamma \vdash_{CC} T : K$

2.2.1 Consistency of the Calculus of Constructions

Definition 2.2.3 $\mathsf{Term}_{CC} = \{M : \exists T, \Gamma.\Gamma \vdash_{cc} M : T\}$

Theorem 2.2.4 ***(Strong Normalization)*** $\forall M \in \mathsf{Term}_{CC}.SN(M)$

The easiest to digest proof is also due to Geuvers [25] but other proofs have also been proposed. This proof has the convenient property that it does not depend too much on the definition of the set SN (strongly normalizing). The only properties required are that $S \subseteq SN$ where S is the set of sorts in the system. In the case of CC $\Box, * \in SN$. It also requires that $\Pi x : A.B \in SN$, and $\lambda x : A.B \in SN$ for any A, B. For those familiar with Geuvers' proof, it is important to note that the proof requires that saturated subsets of SN are closed under arbitrary intersection and function space generation.

This proof is restricted to normalization in the calculus where only β reduction is considered and not η conversion. The proof for the calculus with full reduction properties is in Geuvers [27]. While normalization of terms in Caledon without considering proof search does not involve η conversion, unification could potentially involve η expansion and it is helpful to maintain the consistency of the "Calculus of Constructions" even when η reduction is considered. It is also important to note that in the Curry-style calculus where types are omitted from lambda abstractions, the Church-Rosser theorem under η conversion appears essentially for free [55]. Strong normalization follows, and by replacement of types, strong normalization follows for the Curry-style calculus.

2.2.2 Impredicativity in the Calculus of Constructions

It is also important to note that while the term language of the calculus of constructions is strongly normalizing, the predicates in the calculus of constructions are impredicative, meaning that small types (meaning propositions) can be generalized over all small types. This adds yet another layer of computation into the Caledon language which might not terminate. Furthermore, the predicate language is insufficient to prove properties of larger types. For example, we'd be unable to encode or prove many notions from category theory given that they would be required to apply to the category of predicates. Luo [43] solves this by introducing universes into the "Extended Calculus of Constructions". As defined by Luo, the Extended Calculus of Constructions is no longer a pure type system.

Harper and Pollack [32] provided an analysis of a few pure type systems with universes. Agda, Idris, Coq, and Plastic [12] all implement this technique. Idris uses constraint satisfaction to allow for implicit universes and thus does not allow the universes to become an annoyance in code. While not discussed in detail in the thesis, the Caledon implementation similarly produces constraints on implicit universes during unification and reduces the cyclic universe checking problem to dynamic transitive closure.

2.2.3 Theorems in Caledon

It is important to note that in the programming language Caledon, programs might not terminate. The search language itself is not consistent, and is not a theorem verification language like Elf. Rather, it is language for writing theorem-proving programs.

Definition 2.2.5 *In the Caledon language,* $\mathsf{prop} = *$ *and* $\mathsf{type} = \Box$

It is important to note that in Caledon code, type will never appear. This is because $\mathsf{type} : T$ for any T is not provable in the Calculus of Constructions, and every term that appears in the language must have a provable type.

Definition 2.2.6 *If* $\Gamma \vdash_{CICC} P : T : \mathsf{prop}$ *in the Caledon language, then* T *is a theorem, and* P *is a proof.*

Terms can be considered this way since the Calculus of Constructions is consistent. No unbounded computation is necessary to normalize P. Specifically, proof search is not involved in the normalization procedure.

When CC was first developed, theorems were proven and generated by explicitly defining constructors and destructors for records and sum types. Later, the inductive Calculus of Constructions was developed [15] which more accurately forms the basis of the Coq programming language. These types of inductive constructions have been omitted from Caledon. Instead, predicates are specified by assuming axioms relating them together. This omits the confusion that would be generated from having both inductively defined data and predicates in the system, as dependently typed logic programming treats predicates as both data and code.

Unfortunately, simple CC does not make for an expressive or useful theorem proving language. This can significantly limit the utility of theorem searching techniques. In order to make it more useful, a predicative hierarchy of universes was appended to the language which would allow for inclusion of more meta-mathematics techniques. As it turns out, this universe hierarchy is can be essential to meta-programming in Caledon. While most of the rest of thesis assumes a simple impredicative universe, I explain the universe construction here, and allow the reader to extrapolate for the remainder of the thesis.

The original calculus of constructions with a universe hierarchy included an impredicative type.

Definition 2.2.7 ***(PTS for*** CC_ω***)***

$$A = \{\mathsf{prop}\} \cup \{\mathsf{type}_i | i \in \mathbb{N}\}$$

$$S = \{(\mathsf{prop} : \mathsf{type}_0)\} \cup \{(\mathsf{type}_i : \mathsf{type}_{i+1}) | i \in \mathbb{N}\}$$

$$R = \{(\mathsf{type}_j, \mathsf{type}_i, \mathsf{type}_k) | j \leq k \wedge i \leq k\} \cup \{(\mathsf{prop}, t, t) | t \in A\} \cup \{(t, \mathsf{prop}, t) | t \in A\}$$

While type checking with the addition of the impredicative universe is theoretically equivalently as difficult as without it, in practice it turns out to not be very useful for proof or program writing, and languages like Coq, Agda and Idris now omit it for the system with only predicative universes.

The addition of a universe hierarchy into the Caledon Language is possible without too much added difficulty. Experimentation shows that the omitting the predicative prop allows for a simpler implementation of type inference. In this case, typechecking and unification are performed with the usually inconsistent assumption that type : type, and a cycle checker is later applied to ensure that there is no instance of $\mathsf{type}_i : \mathsf{type}_i$ necessary.

In the case where an impredicative universe is included, the extra axiom prop : type would need to be included, which would mean searching among two possibilities, rather than a single possibility when attempting to find a type to replace a proof hole with. The added nondeterminism tends to cause an explosion in the time complexity of type inference.

2.3 Caledon Implicit Calculus of Constructions

In these sections I lay out the type system for Caledon. I first describe the target system, on which has meaningful theorems, $CICC$. This is the interpreter output of successfully running a Caledon program. I then describe the $CICC$ syntax and statics. Caledons exposed type system is a variant of the Implicit Calculus of Constructions, ICC [53],

which for the rest of the paper I will refer to as $CICC^{-}$. The $CICC^{-}$ statics are realized by the interpretation of the unification problem form (UPF) generated by elaboration. The syntactic pipeline is as follows:

$$CICC^{-} \rightarrow UPF \rightarrow CICC$$

Proving the consistency of $CICC$ requires a further elaboration into CC

$$E ::= V \mid S \mid E\ E \mid \lambda V : T.E \mid ?\lambda V, V : T.E \mid \Pi V : E.E \mid ?\Pi V, V : E.E \mid E\{V : E = E\}$$

Figure 2.1: Syntax of $CICC$

In the rest of the thesis, $?\Pi v : E_1.E_2$ shall refer to $?\Pi v, v : E_1.E_2$ and $?\lambda v : E_1.E_2$ shall refer to $?\lambda v, v : E_1.E_2$.

The *dependent* explicit and implicit products are written $\Pi v : E.E$ and $?\Pi v : A.E$. The *non-dependent* explicit and implicit products are written $T \rightarrow T$ and $T \Rightarrow T$ respectively.

Before providing typing rules, we first supply a few definitions which clarify the notion of a constrained name of a term.

Definition 2.3.1 *The constrained names on a term, written $CN(M)$ is a set defined as follows:*

$$CN(M\{x = E\}) = \{x\} \cup CN(M) \tag{2.4}$$

$$CN(\mathsf{otherwise}) = \emptyset \tag{2.5}$$

The constrained names on a term are defined to be the constraints placed at the top level on a term. For example, in the term $\mathsf{some_p red}$ type $\mathsf{nat}\{A = nat\}\{B = nat\}$ the constrained names are A and B.

Definition 2.3.2 *The generalized names for a term, written $GN(M)$ is a set defined as follows:*

$$GN(?\Pi n, x : T.M) = \{n\} \cup GN(M) \cup GN(T) \tag{2.6}$$

$$GN(\mathsf{otherwise}) = \emptyset \tag{2.7}$$

The definition of generalized names and of constrained names are in a way complimentary. Only the generalized names of the type of a term may be constrained.

Definition 2.3.3 *The bound names for a term, written $BN(M)$ is a set defined as follows:*

$$BN(?\lambda n, x : T.M) = \{n\} \cup BN(M) \tag{2.8}$$

$$BN(\mathsf{otherwise}) = \emptyset \tag{2.9}$$

As defined above, bound names and bound variables can no longer be treated the same in the semantics. Specifically, $?\lambda x : A.B$ does not have the same semantics as $?\lambda y : A.[y/x]B$. This implies that alpha conversion is now limited.

There are a few ways to deal with this. The most attractive possibility is to interpret names as a kind of record modifier. This can be seen as saying $\{x : T = N\} : \{x : N\}$, and $?\lambda x : N.B$ is really just $\lambda y : \{x : N\}.[y.x/x]N$ where $.x : \{x : N\} \to N$.

```
1  defn nat : prop
2    as [a : prop] a -> (a -> a) -> a
3
4  defn nat_1 : nat -> prop
5    as \ N : nat . [a : nat -> prop] a zero -> succty a -> a N
6
7  defn rec : nat -> prop -> prop
8    as \ nm : nat . \ N : kind . nat_1 nm * N
9
10 defn get : [N : kind] [nm : nat] nat_1 nm * N -> N
11   as \ N : kind . \nm : nat . \ c : (nat_1 nm, N) . snd c
12
13 defn put : [N : kind] [nm : nat] nat_1 nm ->  N -> nat_1 nm * N
14   as \ N : kind . \nm : nat . \nmnm : nat_1 nm . \ c : N . pair nmnm N
```

Figure 2.2: Definitions for extraction, written in CC with Caledon syntax

We can further convert this into traditional dependent types by constructing type invariants as seen in 2.2. Note that we shall refer loosely to the encoding in church numerals of a name x as $\bar{x}$. $\bar{\bar{x}}$ refers to the inhabitant of the type $\mathsf{nat}_1\ \bar{x}$. get and put shall be used as aliases for the definitions outlined above.

Then $?\lambda nm, x : A.B$ and $?\Pi nm, x : A.B$ becomes $\lambda y : \mathsf{rec}\ n\bar{m}\ A.[get\ A\ n\bar{m}\ y/x]B$ and $?\Pi y : \mathsf{rec}\ n\bar{m}\ A.[getAn\bar{m}y/x]B$.

Similarly, $N\{x\ :\ T\ =\ A\}$ would become $N\ (\mathsf{put}\ T\ \bar{x}\ \bar{\bar{x}}\ A)$.

One might notice that N in get is of type kind. In simple CC, this is unfortunately not an actual type. Rather, it refers to the use of either type or prop. Allowing N : type is not permitted in the standard CC since it is quantified. This is possible in CC_ω however. In this case, kind would always refer to the next universe after the highest universe mentioned in the program. In the Caledon language implementing simple CC we are free to define kind as type_1, since it will always be larger than any type or kind mentioned

in a Caledon program. Fortunately, Geuvers' proof [27] of strong normalization in the presence of η conversion applies to the Calculus of Construction with one impredicative universe and two predicative universes.

This intuitive conversion leads to the following typing rules for $CICC$.

The simplest, albeit not the most constructive method of defining typeability for $CICC$ can be obtained by a simple projection into CC.

Definition 2.3.4 ***(Projection from*** $CICC$ ***to*** CC***)***

$$[\![v]\!]_{ci} := v \tag{2.10}$$

$$[\![s]\!]_{ci} := s \tag{2.11}$$

$$[\![E_1\ E_2]\!]_{ci} := [\![E_1]\!]_{ci}\ [\![E_2]\!]_{ci} \tag{2.12}$$

$$[\![E_1\ \{x : T = E\}]\!]_{ci} := [\![E_1]\!]_{ci}\ (\mathsf{put}\ [\![T]\!]_{ci}\ \bar{x}\ \bar{\bar{x}}; [\![E_2]\!]_{ci}) \tag{2.13}$$

$$[\![\lambda nm, v : T.E]\!]_{ci} := \lambda v : [\![T]\!]_{ci} . [\![E]\!]_{ci} \tag{2.14}$$

$$[\![?\lambda nm, v : T.E]\!]_{ci} := \lambda y : \mathsf{rec}\ n\bar{m}\ [\![T]\!]_{ci} . [\![[\mathsf{get}\ [\![T]\!]_{ci}\ n\bar{m}\ y/v]E]\!]_{ci} \quad \textit{where } y \textit{ is fresh} \tag{2.15}$$

$$[\![\Pi v : T.E]\!]_{ci} := \Pi v : [\![T]\!]_{ci} . [\![E]\!]_{ci} \tag{2.16}$$

$$[\![?\Pi nm, v : T.E]\!]_{ci} := \Pi y : \mathsf{rec}\ n\bar{m}\ [\![T]\!]_{ci} . [\![[\mathsf{get}\ [\![T]\!]_{ci}\ n\bar{m}\ y/v]E]\!]_{ci} \quad \textit{where } y \textit{ is fresh} \tag{2.17}$$

It is significant that church numerals be used for the representation of the name in the record, as no extra axioms need to be included in the context of the translation for the translation to be valid. This necessity is seen in **??**.

Definition 2.3.5 ***(Typing for*** $CICC$***)*** *We say* $\Gamma \vdash_{ci} A : T$ *iff* $[\![\Gamma]\!]_{ci} \vdash_{cc} [\![A]\!]_{ci} : [\![T]\!]_{ci}$

What is then important is that we can translate back from the calculus of constructions after transformations have occurred.

Theorem 2.3.6 ***(Projection Substitution)***

$[\![[A/x]B]\!]_{ci} = [[\![A]\!]_{ci}/x]\,[\![B]\!]_{ci}$ *provided* x *is free for* A *in* B.

The proof of this theorem is by induction on the structure of B.

Case 1 *Suppose B is the variable x.*

$$[\![[A/x]x]\!]_{ci} = [\![A]\!]_{ci} = [[\![A]\!]_{ci}/x]x = [[\![A]\!]_{ci}/x]\,[\![x]\!]_{ci}$$

Case 2 *Suppose B is a different variable v.*

$$[\![[A/x]v]\!]_{ci} = v = [[\![A]\!]_{ci}/x]v = [[\![A]\!]_{ci}/x]\,[\![v]\!]_{ci}$$

Case 3 *The most interesting case is $B = ?\lambda nm, v : T.E$*

Then

$$[[\![A]\!]_{ci}/x]\,[\![?\lambda nm, v : T.E]\!]_{ci} = [[\![A]\!]_{ci}/x](\lambda v : \mathsf{rec}\ n\bar{m}\ \ [\![T]\!]_{ci}\,.\,[\![\mathsf{get}\ \ [\![T]\!]_{ci}\ \ n\bar{m}\ y/v]E]\!]_{ci})$$

where y is fresh.

Then

$$= (\lambda v : \mathsf{rec}\ n\bar{m}\ ([[\![A]\!]_{ci}/x]\,[\![T]\!]_{ci}).[[\![A]\!]_{ci}/x]\,[\![[\mathsf{get}\ \ [\![T]\!]_{ci}\ \ n\bar{m}\ y/v]E]\!]_{ci})$$

$$= (\lambda v : \mathsf{rec}\ n\bar{m}\ \ [\![[A/x]T]\!]_{ci}\,.[[\![A]\!]_{ci}/x][\mathsf{get}\ \ [\![T]\!]_{ci}\ \ n\bar{m}\ y/v]\,[\![E]\!]_{ci})$$

Then by the induction hypothesis on T and E and the fact that $[[\![A]\!]_{ci}/x]\mathsf{get} = \mathsf{get}$ and $[[\![A]\!]_{ci}/x]\mathsf{rec} = \mathsf{rec}$ and $x \neq y$ since y is fresh, we get:

$$= (\lambda v : \mathsf{rec}\ n\bar{m}\ \ [\![[A/x]T]\!]_{ci}\,.[\mathsf{get}\ \ [\![[A/x]T]\!]_{ci}\ \ n\bar{m}\ y/v][[\![A]\!]_{ci}/x]\,[\![E]\!]_{ci})$$

since we know that v isn't free in A by the fact that x is free for A in B (provided x is actually used anywhere in E) we get the following:

$$= (\lambda v : \mathsf{rec}\ n\bar{m}\ \ [\![[A/x]T]\!]_{ci}\,.\,[\![\mathsf{get}\ \ [\![[A/x]T]\!]_{ci}\ \ n\bar{m}\ y/v][A/x]E]\!]_{ci})$$

$$= [\![?\lambda nm, v : [A/x]T.[A/x]E]\!]_{ci}$$

$$= [\![[A/x](?\lambda nm, v : T.E)]\!]_{ci}$$

Lemma 2.3.7 ***(Reduction Translation)*** *Forall* $M, N \in \mathsf{Term}_{ci}$ *if* $M \rightarrow_{\beta\eta *} N$ *then* $[\![M]\!]_{ci} \rightarrow_{\beta\eta *} [\![N]\!]_{ci}$

Proof of this lemma is by lexicographic induction on the structure of $M \rightarrow N$ and M.

Theorem 2.3.8 ***(Semantic Equivalence)***

Forall $M \in \mathsf{Term}_{ci}$ *such that* $[\![M]\!]_{ci} \rightarrow_{\beta\eta *} N'$ *and* $\Gamma \vdash M : T$ *implies that there exists* $M' \in \mathsf{Term}_{cicc}$ *such that* $M \rightarrow_{\beta\eta *} M'$ *and* $[\![M']\!]_{ci} \equiv N'$

The proof is by induction on the form of $[\![M]\!]_{ci} \rightarrow_{\beta\eta} N'$ and then induction to arbitrary numbers of reductions.

Here we only show the case of β reduction. η reduction/expansion is analogous.

Now assume $[\![M]\!]_{ci} \rightarrow_{\beta\eta} N'$ is of the form $[\![E]\!]_{ci} \rightarrow_{\beta} V$. Then it is of the form $(\lambda x : T_a.R)E_2 \rightarrow_{beta} [[\![E_2]\!]_{ci}/x]R$ where $V = [[\![E_2]\!]_{ci}/c]R$ and $(\lambda x : T_a.R)E_2 = [\![E_1]\!]_{ci}\, E_2 = [\![E]\!]_{ci}$.

Then since $[\![E_1]\!]_{ci} = (\lambda x : T_a.R)$ we know that $E_1 = \lambda x : T'_a.R'$ or that $E_1 = ?\lambda nm, v : T'_a.R'$. We examine these two cases.

Case 1 *We begin with the case of* $E_1 = \lambda x : T'_a.R'$

Then because $\Gamma \vdash_{ci} M : T$ and $M = (\lambda x : T'_a.R')E'_2$, we know that $\Gamma \vdash_{ci} (\lambda x : T'_a.R') : \Pi x : T'_a.T_2$. and thus $[\![\Gamma]\!]_{ci} \vdash_{cc} [\![(\lambda x : T'_a.R')]\!]_{ci} : [\![\Pi x : T'_a.T_2]\!]_{ci}$.

Then $[\![\Gamma]\!]_{ci} \vdash_{cc} (\lambda x : [\![T'_a]\!]_{ci}\,.\,[\![R']\!]_{ci})\,[\![E_2]\!]_{ci} : [\![T]\!]_{ci}$ is the result of the "app" rule.

Since the "app" rule has to have been applied, we know that $[\![M]\!]_{ci} = [\![(\lambda x : T'_a.R')E_2]\!]_{ci} = (\lambda x : [\![T'_a]\!]_{ci}\,.\,[\![R']\!]_{ci})\,[\![E_2]\!]_{ci}$.

Thus, $[\![E'_2]\!]_{ci} = E_2$ and $[\![T'_a]\!]_{ci} = T_a$ and $[\![R']\!]_{ci} = R$.

Since we are performing a β reduction, we know that x is free for $[\![E_2]\!]_{ci}$ in $[\![R']\!]_{ci}$. Then by the projection substitution theorem 2.3.4, $[\![[E_2/x]R']\!]_{ci} = [[\![E_2]\!]_{ci}/x]\,[\![R']\!]_{ci}$ where $[\![R']\!]_{ci} = R$.

Thus, $[\![[E_2/x]R']\!]_{ci} = V$

Case 2 *The last case is where* $E_1 = ?\lambda nm, x : T'.R'$

Because $\Gamma \vdash_{ci} M : T$ and $M = (?\lambda nm, x : T'_a.R')E'_2$ we know that $\Gamma \vdash_{ci} (?\lambda x : T'_a.R') : ?\Pi nm, x : T'_a.T_2$ clearly.

Then $[\![\Gamma]\!]_{ci} \vdash_{cc} [\![?\lambda nm, x : T'_a.R']\!]_{ci} E_2 : [\![T]\!]_{ci}$ must be the result of the is the result of the "app" rule.

Thus, we know that $[\![\Gamma]\!]_{ci} \vdash_{cc} [\![?\lambda nm, x : T'_a.R']\!]_{ci} : \Pi y : T_3.T_4$ for some T_3 and T_4.

By uniqueness of forms and inversions, we get that

$$\Pi y : T_3.T_4 = \Pi y : \mathsf{rec} n\bar{m}\, [\![T'_a]\!]_{ci}\, .\, [\![[\mathsf{get}\, [\![T'_a]\!]_{ci}\, n\bar{m}y/x]T_2]\!]_{ci}$$

since

$$[\![?\lambda nm, x : T'_a.R']\!]_{ci} = \lambda y : \mathsf{rec} n\bar{m}\, [\![T'_a]\!]_{ci}\, .\, [\![[\mathsf{get}\, [\![T'_a]\!]_{ci}\, n\bar{m}y/x]R']\!]_{ci}$$

Thus, we know that

$$[\![M]\!]_{ci} = [\![(?\lambda nm, x : T'_a.R')]\!]_{ci}\, (\mathsf{put}\, [\![T']\!]_{ci}\, n\bar{m}n\bar{\bar{m}}\, [\![E_2]\!]_{ci})$$

Thus, $M = (?\lambda nm, x : T'_a.R')\{nm : T'_2 = E_2\}$ and $[\![T'_a]\!]_{ci} = T$ and $[\![R']\!]_{ci} = R$. Since we are performing a β reduction, we know that x is free for $[\![E_2]\!]_{ci}$ in $[\![R']\!]_{ci}$. Then by the projection substitution theorem 2.3.4, $[\![[E_2/x]R']\!]_{ci} = [[\![E_2]\!]_{ci}/x]\, [\![R']\!]_{ci}$ where $[\![R']\!]_{ci} = R$

Thus, $[\![[E_2/x]R']\!]_{ci} = V$

Theorem 2.3.9 ***(Strong Normalization)*** $\forall M \in \mathsf{Term}_{ci}.SN(M)$

Strong normalization is a consequence of the reduction translation lemma 2.3.7.

That we can cleanly translate into the calculus of constructions without loss or gain of semantic translation implies strong normalization for $CICC$ with β reduction and η expansion. This is the most important theorem of the section, as it implies that type-checking a Caledon statement will allow that statement to be compiled to pattern form to be used in proof search.

That $CICC$ is simply an extension of CC and not a modification of CC implies we have the completeness theorem, 2.3.10.

Theorem 2.3.10 ***(Completeness)*** $\forall M, T \in \mathsf{Term}_{cc}. \vdash_{cc} M : T \implies \vdash_{ci} M : T$

This theorem is trivial since the syntax of CC is a subset of the syntax of $CICC$.

Chapter 3

Operational Semantics

In this chapter, I lay out and justify an operational specification and semantics for Caledon. I first discuss the history of the technique of unification.

Caledon's unification and proof search algorithm is based on the method designed for Elf by Pfenning et al. [60]. This algorithm does not terminate on all inputs and in this chapter, I begin by characterizing the inputs it is designed to terminate on. In the next section, I describe the techniques for evaluating terms to forms for unification. I then describe the algorithm used for higher order unification. The last sections pertain to proof search, which are the traditional semantics of the language. In the last section, I specify a technique for controlling of nondeterminism during proof search, which allows the programmer to choose between a complete breadth first search and an imperative depth first search.

3.1 History

Huet [37] gave the first semi-decision algorithm for unification of terms in the lambda calculus. Later, Miller and Nadathur [51] proved that for terms in the pattern fragment of the lambda calculus, unification was decidable. Pfenning and Elliot [59] demon-

strated unification for the typed lambda calculus and considered solving the dynamic pattern fragment where non pattern equations are postponed. Elliott [23] gave a more efficient algorithm for unification in the context of dependent types. Later Pfenning [61] did the same thing for unification in the "Calculus of Constructions", although without a mixed prefix. The most succinct presentation is from the 1991 paper describing the workings of Elf [60]. While the unification algorithm implemented in the interpreter for Caledon is an extension of that presented in Pfenning [61], I briefly cover here the main ideas from the presentation of the paper describing the workings of Elf, and extend those ideas later.

3.2 Forms for Unification

Definition 3.2.1 *Spine Form*

$$N ::= P \mid \lambda V : N.N \tag{3.1}$$

$$P ::= V \mid PN \tag{3.2}$$

We write $\Pi V : N.P$ as a synonym for $\Pi\ N\ (\lambda V : N.P)$ in the rest of this thesis. This simplifies the presentation of the unification algorithm, as then Π can be considered a traditional constructor that can also be used to direct the unification procedure.

Spine terms have the incredibly useful property of always being in head normal form, meaning that the head of every term is a constructor, and every argument is either a constructor or lambda term.

3.2.1 Higher Order Patterns

While spine form is restrictive enough that its terms are always in head normal form, it is not restrictive enough for unification problems to be decidable. Miller [49] showed

that for any unification instance given in the pattern fragment shown in 3.2.2, unification is decidable.

Pattern form is specified with respect to partial permutations ϕ, which are injective mappings from finite domains to finite domains.

Definition 3.2.2 *Pattern Form: Note that Δ is the existential context and Γ is the universal context.*

$$\frac{\Delta; \Gamma \vdash A \ \mathsf{Pat} \quad \Delta; \Gamma, u \vdash M \ \mathsf{Pat}}{\Delta; \Gamma \vdash \lambda u : A.M \ \mathsf{Pat}} \ \mathsf{P/ABS}$$

$$\frac{\Delta; \Gamma \vdash M_1 \ \mathsf{Pat} \quad \cdots \quad \Delta; \Gamma \vdash M_m \ \mathsf{Pat}}{\Delta; \Gamma \vdash c \ M_1 \cdots M_m \ \mathsf{Pat}} \ \mathsf{P/CON} \quad \frac{\Delta; \Gamma \vdash M_1 \ \mathsf{Pat} \quad \cdots \quad \Delta; \Gamma \vdash M_m \ \mathsf{Pat} \quad u \in \Gamma}{\Delta; \Gamma \vdash u \ M_1 \cdots M_m \ \mathsf{Pat}} \ \mathsf{P/VAR}$$

$$\frac{\phi \textit{ is a partial permutation} \quad x \in \Delta}{\Delta; u_1, \cdots u_p \vdash x \ u_{\phi(1)} \cdots u_{\phi(m)} \ \mathsf{Pat}} \ \mathsf{P/PROP} \quad \frac{\Delta; \Gamma \vdash M' \ \mathsf{Pat} \quad M \equiv_\eta M'}{\Delta; \Gamma \vdash M \ \mathsf{Pat}} \ \mathsf{P/VAR}$$

3.2.2 Canonical Forms

The unification algorithm in Pfenning [61] for the "Calculus of Constructions", which the meta-theory of Caledon is based on, relies on expressions being presented in β-*normal* η*-long form* (or *canonical form*), meaning that they are η expanded to conform to their type signature. In the initial publication of this paper, it was taken as a hypothesis that every well-typed term in CC has a unique β-normal η-long form. This is now known to be the case [1].

Definition 3.2.3 *Canonical Forms*

$$\frac{(s_1, s_2) \in A}{\Gamma \vdash s_1 \ \Rightarrow \ s_2} \ \mathsf{F/ax} \quad \frac{\Gamma \vdash A \ \Rightarrow \ s_1 \quad \Gamma, x : A \vdash B \ \Rightarrow \ s_2 \quad (s_1, s_2, s_3) \in R}{\Gamma \vdash \Pi x : A.B \ \Rightarrow \ s_3} \ \mathsf{F/prod}$$

$$\frac{\Gamma, x : A \vdash M \;\Rightarrow\; B \quad \Gamma \vdash A \;\Rightarrow\; s}{\Gamma \vdash \lambda x : A.M \;\Rightarrow\; \Pi x : A.B}\ \mathsf{F/lam}$$

$$\frac{\Gamma \vdash h\ M_1 \cdots M_n : D \quad \Gamma \vdash M_1 \;\Rightarrow\; A_1 \quad \cdots \quad \Gamma \vdash M_n \;\Rightarrow\; A_n}{\Gamma \vdash h\ M_1 \cdots M_n \;\Rightarrow\; D}\ \mathsf{F/app}$$

where D is atomic

It has been proven that the standard "Calculus of Constructions" is β-normal η-long form strongly normalizing. Unfortunately, normalization into this form is not possible without type information. Later, a typed substitution algorithm will be given which ensures normalization into this form.

The notions of canonical form and of a higher order pattern are also trivially extensible into Church-style CC^{Bi} (ie, ICC* from [6]), where strong normalization into β-normal η-long form is also provable, as is shown by Barras and Bernardo [6].

3.3 Substitution

The higher order unification algorithm described is only defined on the pattern form provided in the previous section. As the pattern fragment is a restriction on β-normal η-long form, it is necessary to provide a normalizing substitution that preserves β-normal η-long form. The presentation here revolves around hereditary substitution as described by Pfenning et al. [60] for a calculus with only β-reduction.

Other presentations are possible, such as traditional substitution followed by "Normalization by Evaluation" for $\beta\eta$-conversions[1]. While this is a proven total method in a typed setting, it's mechanics are complex and not particularly enlightening. Keller et al. [41] extended hereditary substitution to η-expansion, but only in the simply typed case. The presentation here extends this version into CC.

3.3.1 Untyped Substitution

Definition 3.3.1 ***(Hereditary Substitution)***

$$[S/x]x := S \quad [S/x]y := y \quad [S/x]P\ T := \mathsf{H}([S/x]P, [S/x]T)$$

$$[S/x]\lambda v : T.P := \lambda v' : [S/x]T.[S/x][v'/v]P \ \textit{ where } v' \textit{ is new.}$$

$$\mathsf{H}(\lambda v : T.P, A) := [A/v]P$$

$$\mathsf{H}(PA_1, A_2) := P\ A_1\ A_2 \quad \mathsf{H}(V, A) := V\ A$$

It is important to note the alpha conversion in the λ case, as alpha conversion will be lost on some terms when implicits are added.

A hereditary substitution is not necessarily terminating as is shown by substitution replicating the ω-combinator.

$$[(\lambda x : T.x\ x)/x](x\ (\lambda x : T.x\ x))$$

This is not defined, as it expands to the well known $\mathsf{H}(\lambda x.x\ x, \lambda x.x\ x)$.

If the pattern and substitution are well typed terms in the Calculus of Constructions, by strong normalization, this version of hereditary substitution is total.

Theorem 3.3.2 *Substitution Theorem:*

If $\Gamma, x : T \vdash A : T' : \mathsf{prop}$ *and* $\Gamma \vdash S : T : \mathsf{prop}$ *then* $\Gamma \vdash [S/x]_o A : [S/x]_o T' : \mathsf{prop}$

By consistency, $[S/x]_o A$ can be normalized to strong head normal form. Thus, the hereditary substitution $[S/x]A$ is defined.

3.3.2 Typed Substitution

Performing substitutions that maintain β-normal η-long form is important to ensuring decidability of unification. Unfortunately this is not possible without some type information, as arbitrary η expansion has no stop condition. Keller [41] gave a hereditary substitution algorithm that results in canonical forms for simply typed lambda calculus. 3.3.4 solves this by performing a typed substitution under a context, generating type information.

Theorem 3.3.3 ***(Reduction Decomposition)*** *If* $\Gamma \vdash A \Rightarrow_{\beta\eta*} B$ *then there exists some* C *such that* $A \Rightarrow_{\beta*} C$ *and* $\Gamma \vdash C \Rightarrow_{\eta*} B$

The property in 3.3.3 stating that any reduction can be shown equivalent to first a series of β reductions and then a series of η expansions forms the basis of the following algorithm.

For this section it is convenient to include Π in the spine form:

$$N ::= P \mid \lambda V : N.N \mid \Pi V : N.N$$

Definition 3.3.4 ***(Typed Hereditary Substitution)***

$$[S/x : A]^n_\Gamma P := \mathsf{E}([S/x : A]^p_\Gamma P) \tag{3.3}$$

$$[S/x : A]^n_\Gamma(\lambda y : B.N) := \lambda y : [S/x]B.[S/x : A]^n_{\Gamma,y:B}N \tag{3.4}$$

$$\mathsf{E}(M \uparrow P) := M \tag{3.5}$$

$$\mathsf{E}(N \downarrow A) := \eta_A^{-1}(N) \tag{3.6}$$

$$\eta^{-1}_{\Pi x:A.B}(N) := \lambda z : A.\eta_B^{-1}(N\ \eta_A^{-1}(z)) \textit{ where } z \textit{ is fresh} \tag{3.7}$$

$$\eta_P^{-1}(N) := N \tag{3.8}$$

$$[N/x : A]^p_\Gamma x := N \uparrow A \tag{3.9}$$

$$[S/x : A]^p_\Gamma y := y \downarrow \Gamma(x) \tag{3.10}$$

$$[S/x : A]^p_\Gamma P\ N := \mathsf{H}_\Gamma([S/x : A]^p_\Gamma P, [S/x : A]^n_\Gamma N) \tag{3.11}$$

$$\mathsf{H}_\Gamma((\lambda v : A_1.N) \uparrow \Pi v' : A_1.A_2, P) := [P/v : A_1]^n_\Gamma N \uparrow [P/v']^n A_2 \tag{3.12}$$

$$\mathsf{H}_\Gamma(P \downarrow \Pi y : B_1.B_2, N) := P\ N \downarrow [N/y]^n B_2 \tag{3.13}$$

In these last two rules, when initializing the dependent product, the substitution must be neither typed nor η-expanding as this leads to an unnecessary circularity. Types need not be in η-long form when used for η-expanding in substitution, since they are not unified against anything.

Also, in the hereditary part of the recurrence, it is possible to omit cases for improperly formatted terms. $A \downarrow T$ has the invariant that A must be already in a canonical form within the context it is used since it existed previously in the equation. Similarly, $A \uparrow T$ has the invariant that A must be locally in a canonical form. This permits significant reduction of steps.

In general, it is provable that for any PTS that is strongly normalizing for β reduction and η expansion, this algorithm will terminate and substitution will be defined. This is a direct consequence of 3.3.3. Since every step of the algorithm applies either an η

expansion or a β reduction, the algorithm must halt when substituting for well typed terms.

The useful property of this hereditary substitution is spelled out in 3.3.6

Theorem 3.3.5 ***(Soundness of η Expansion)*** *If $\Gamma \vdash F : A$ and F is in β-normal form then $\eta_A^{-1}(F)$ is in η-long form*

The proof of this theorem is trivial since all that occurs is the complete η expansion of every possible term.

Note, in the following theorems 3.3.63.3.73.3.8 the $\Gamma \vdash N$ Norm means that N is in β-normal η-long form (ignoring types).

Theorem 3.3.6 ***(Soundness of Hereditary Substitution)*** *If $\Gamma \vdash S : A$ and $\Gamma \vdash S$ Norm and $\Gamma \vdash N$ Norm and x is free for S in N and $[S/x : A]^n N$ is defined, then $\Gamma \vdash ([S/x : A]^n_\Gamma N)$ Norm*

Theorem 3.3.7 ***(Soundness of Hereditary Application-1)*** *If $\Gamma \vdash P : \Pi v' : A_1.A_2$ and $\Gamma \vdash N : A_1$ and $\Gamma \vdash P$ Norm and $\Gamma \vdash N$ Norm then $E(H_\Gamma(P \downarrow \Pi v' : A_1.A_2, N))$ Norm*

Theorem 3.3.8 ***(Soundness of Hereditary Application-2)*** *If $\Gamma \vdash (\lambda v : A_1.N) : \Pi v' : A_1.A_2$ and $\Gamma \vdash P : A_1$ and $\Gamma \vdash P$ Norm and $\Gamma \vdash N$ Norm and v is free for P in N then $\Gamma \vdash E(H_\Gamma((\lambda v : A_1.N) \uparrow \Pi v' : A_1.A_2, P))$ Norm*

The proof for these theorems is by mutual induction: for substitution on the structure of $[S/x : A]^n_\Gamma N = E([S/x : A]^p_\Gamma N)$, for application-1 on the structure of $E(H_\Gamma(P \downarrow \Pi v' : A_1.A_2, N))$, and for application-2 on the structure of $E(H_\Gamma((\lambda v : A_1.N) \uparrow \Pi v' : A_1.A_2, P))$.

Case 1 ***Substitution*** *Suppose N is of the form $(\lambda y : B.N')$.*

$$[S/x : A]^n_\Gamma(\lambda y : B.N') = \lambda y : [S/x]B.[S/x : A]^n_{\Gamma, y:B} N'$$

And thus by the induction hypothesis, $\Gamma \vdash [S/x : A]^n_{\Gamma, y:B} N'$ Norm and thus $\Gamma \vdash \lambda y : [S/x]B.[S/x : A]^n_\Gamma N'$ Norm

Case 2 ***Substitution*** *Suppose N is of the form $P\ N'$.*

$$[S/x : A]^n_\Gamma N = E(H_\Gamma([S/x : A]^n_\Gamma P, [S/x : A]^n_\Gamma N'))$$

And thus by the induction hypothesis, $\Gamma \vdash [S/x : A]^n_\Gamma P$ Norm and

$$\Gamma \vdash [S/x : A]^n_\Gamma N' \text{ Norm}$$

Thus,

$$E(H_\Gamma([S/x : A]^n_\Gamma P, [S/x : A]^n_\Gamma N'))$$

by mutual induction using 3.3.7.

Case 3 ***Substitution*** *Suppose N is of the form x.*

$$[S/x : A]^n_\Gamma x = E(S \uparrow A) = S$$

and by argument, $\Gamma \vdash S$ Norm already.

Case 4 ***Substitution*** *Suppose N is of the form y.*

$$[S/x : A]^n_\Gamma y = E(y \downarrow \Gamma(y)) = \eta^{-1}_{\Gamma(y)}(y)$$

which by theorem 3.3.5 should be in β-normal, η-long form.

Case 5 ***Application-1***

$$E(H_\Gamma(P \downarrow \Pi v' : A_1.A_2, N)) = E(PN \downarrow [N/v']A_2) = \eta^{-1}_{[N/v']A_2}(PN)$$

Since $\Gamma \vdash N$ Norm and $\Gamma \vdash P$ Norm,

Thus, PN is in β-normal form, and so $\Gamma \vdash \eta^{-1}_{[N/v']A_2}(PN)$ Norm

Case 6 ***Application-2***

$$E(H_\Gamma((\lambda v : A_1.N) \uparrow \Pi v' : A_1.A_2, P)) = E([P/v : A_1]^n_\Gamma N \uparrow [P/v']^n A_2) = [P/v : A_1]^n_\Gamma N$$

Since $\Gamma \vdash (\lambda v : A_1.N) : \Pi v' : A_1.A_2$, we know that $(\lambda v : A_1.N)P$ must normalize and thus $[P/v : A_1]^n_\Gamma N$ is defined. Thus, since $\Gamma \vdash P$ Norm and $\Gamma \vdash N$ Norm and $\Gamma \vdash P : A_1$, we know that $\Gamma \vdash [P/v : A_1]^n_\Gamma N$ Norm

Thus, $\Gamma \vdash E(H_\Gamma((\lambda v : A_1.N) \uparrow \Pi v' : A_1.A_2, P))$ Norm

3.4 Higher Order Unification

Checking for the equivalence of two full lambda terms has long been known to be only semi-decidable. The matter becomes even more complicated when checking for the equality of terms with variables bound by both existential and universal quantifiers. Research from the past thirty years has constrained the problem to a decidable subset known as the pattern fragment.

3.4.1 Unification Terms

Definition 3.4.1 *Unification Terms:*

$$U ::= U \wedge U \mid \forall V : T.U \mid \exists V : T.U \mid T \doteq T \mid \top$$

When $\doteq$ is taken to mean $\equiv_{\beta\eta\alpha *}$, the unification problem is to determine whether a statement U is "true" in the standard sense, and provide a proof of the truth of the statement.

Unification problems of the form $\forall x : T_1.\exists y : T_2.U$ can be solved by solving those of the form $\exists y : \Pi x : T_1.T_2.\forall x : T_1.[y\ x/y]U$ in the process known as raising. Unification

statements can always quantified over unused variables: $U \implies Qx : T.U$ where $Q ::= \exists \mid \forall$.

Thus, statements can always be converted to the form

$$\exists y_1 \cdots y_n . \forall x_1 \cdots x_k . S_1 \doteq V_1 \wedge \cdots S_r \doteq V_r$$

3.4.2 Unification Term Meaning

We can provide an provability relation of a unification formula based on the obvious logic.

Definition 3.4.2 *$\Gamma \Vdash F$ can be interpreted as Γ implies F is provable.*

$$\frac{\Gamma \vdash M : A \quad M \equiv_{\beta\eta\alpha *} N \quad \Gamma \vdash N : A}{\Gamma \Vdash M \doteq N}\ \mathsf{equiv} \qquad \frac{}{\Gamma \Vdash \top}\ \mathsf{true} \qquad \frac{\Gamma \Vdash F \quad \Gamma \Vdash G}{\Gamma \Vdash F \wedge G}\ \mathsf{conj}$$

$$\frac{\Gamma \Vdash [M/x]F \quad \Gamma \vdash M : A}{\Gamma \Vdash \exists x : A.F}\ \mathsf{exists} \qquad \frac{\Gamma, x : A \Vdash F}{\Gamma \Vdash \forall x : A.F}\ \mathsf{forall}$$

While a truly superb logic programming language might be able to convert this very declarative specification into a runnable program, the essentially nondeterministic rule for existential quantification in a unification formula prevents an obvious deterministic algorithm from being extracted.

3.4.3 Higher Order Unification for CC

I now present an algorithm, similar to that presented in [60], for unification in the "Calculus of Constructions". Because we have already presented typed hereditary substitution with η-expansion, the presentation here will address itself to types in the substitutions.

$F \longrightarrow F'$ shall mean that F can be transformed to F' without modifying the provability. An equation $F[G]$ will stand as notation for highlighting G under the formulae context F. As an example, if we were to examine the formula $\forall x.\forall n.\exists y.(y \doteq x \wedge \forall z.\exists r.[xz \doteq r])$ but were only interested in the last portion, we might instead write it as $\forall x.F[\forall z.\exists r.[xz \doteq r]]$ Again, ϕ shall be an injective partial permutation.

Furthermore, rather than explicitly writing down the result of unification, we shall use $\exists x.F \longrightarrow \exists x.[L/x]F$ to stand for $\exists x.F \longrightarrow \exists x.x \doteq L \wedge [L/x]F$

The unification rules are symmetrical, so any rule of the form $M \doteq N$ is equivalent to $N \doteq M$ practically.

Also, for the purpose of typed normalizing hereditary substitution, a formula prefix $F[e]$ of the form $Qx_1 : A_1.E_1 \wedge \cdots Qx_n : A_n.e$ shall be considered as a context $x_1 : A_1, \cdots, x_n : A_n$ when written $\nu^{-1}(F)$.

Case 1 *Lam-Any*

$$F[\lambda x : A.M \doteq N] \longrightarrow F[\forall x : A.M \doteq \mathsf{H}_{\nu^{-1}(F),x:A}(N, x)]$$

Because application is normalizing, the "Lam-Any" can cover the case where N is also a λ abstraction.

Case 2 *Lam-Lam*

$$F[\lambda x : A.M \doteq \lambda x : A'.N] \longrightarrow F[A \doteq A' \wedge \forall x : A.M \doteq N]$$

While the "Lam-Lam" rule is not explicitly necessary as it is covered by the "Lam-Any" rule, when working in a substitutive system with explicit names rather than De-Bruijn indexes, this helps to reduce the number of substitutions from an original name.

These reductions make the assumption that no variable name is bound more than once. Strange as this assumption might sound, it is equivalent to working entirely with

DeBruijn indexes. Another option is to α-convert everywhere and annotate new variables with their original names, then α-convert back to the original after unification. Another option is to use DeBruijn indexes. DeBruijn indexes have their own drawbacks here, as certain transformation such as "Raising" or the "Gvar-Uvar" rules involve insertion of multiple variables into the context at an arbitrary point, which requiring the lifting of many variable names.

Case 3 *Uvar-Uvar*

$$F[\forall y : A.G[yM_1 \cdots M_n \doteq yN_1 \cdots N_n]] \longrightarrow F[\forall y : A.G[M_1 \doteq \wedge N_1 \cdots \wedge M_1 \cdots N_n]]$$

Case 4 *Identity*

$$F[M \doteq M] \longrightarrow F[\top]$$

Case 5 *Raising*

$$F[\forall y : A.\exists x : B.G] \longrightarrow F[\exists x' : (\Pi y : A.B).\forall y : A.[x'y/x : B]_{F,x:\Pi y:A.B,y:A}G]$$

This rule is important, as correct or incorrect application of this rule can result in terminating or non terminating reduction sequences.

Case 6 *Exists-And*

$$F[(\exists x : A.E_1) \wedge E_2] \longrightarrow F[\exists x : A.E_1 \wedge E_2]$$

Case 7 *Forall-And*

Moving the universal quantifier to capture a conjunction is critical, since if done incorrectly, existential variables might be defined with respect to universal quantifiers that they were not previously in the scope of.

$$F[(\forall x : A.E_1) \wedge E_2] \longrightarrow F[\forall x : A.E_1 \wedge E_2]$$

provided no existential variables are declared in E_2.

While this restriction prevents most applications of this rule, equations can still be flattened to the form

$$Qx_1 : A_1 \cdots Qx_n : A_n.M_1 \doteq N_1 \wedge \cdots \wedge M_m \doteq N_n$$

transforming E_2 first with the "Raising" rule until an "Exists-And" transformation is possible, then repeating until E_2 no longer contains any existentially quantified variables. This process is always terminating, although potentially significantly slower.

The following cases are based on unification of a formula of the form

$$\Gamma[\exists x : \Pi u_1 : A_1 \cdots \Pi u_n : A_n.AF[\forall y_1 : A'_1.G_1[\cdots \forall y_p : A'_p.G_p[xy_{\phi(1)} \cdots y_{\phi(n)} \doteq M] \cdots]]]$$

Case 8 *Gvar-Uvar-Outside*

M has the form $yM_1 \cdots M_m$ some y universally quantified outside of x and $y : \Pi v_1 : B_1 \cdots \Pi v_m : B_m.B$.

Then we can *imitate* y with x'. Let $L = \lambda u_1 : A_1 \cdots \lambda u_n : A_n.y(x_1u_1 \cdots u_n) \cdots (x_mu_1 \cdots u_n)$.

Then we can transition to

$$\exists x_1 : \Pi u_1 : A_1 \cdots \Pi u_n : A_n.B_1 \cdots \exists x_m : \Pi u_1 : A_1 \cdots \Pi u_n : A_n.[x_{m-1}u_1 \cdots u_n/v_{m-1} : B_{m-1}] \cdots$$

$$[x_1u_1 \cdots u_n/v_1 : B_1]_{\Gamma, x_1:T_{x_1}, \cdots, x_{m-1}:T_{x_n}} B_m.[L/x : T_x]_{\Gamma, x_1:T_{x_1}, \cdots, x_m:T_{x_n}} F$$

Case 9 *Gvar-Uvar-Inside*

If M has the form $y_{\phi(i)}M_1 \cdots M_m$ for $1 \leq i \leq n$ then we can project x to $y_{\phi(i)}$.

Here we can perform the same transition as in the "Gvar-Uvar-Outside" case but let $L = \lambda u_1 : A_1 \cdots \lambda u_n : A_n.u_i(x_1u_1 \cdots u_n) \cdots (x_mu_1 \cdots u_n)$.

Case 10 *Gvar-Gvar-Same*

M has the form $xy_{\psi(1)} \cdots y_{\psi(n)}$.

In this case we pick the *unique* permutation ρ such that $\rho(k) = \psi(i)$ for all i such that $\psi(i) = \phi(i)$.

Then letting $L = \lambda u_1 : A_1 \cdots \lambda u_n : A_n.x' u_{\rho(1)} \cdots u_{\rho(n)}$, we can transition to

$$\exists x' : \Pi u_1 : A_\rho(1) \cdots \Pi u_l : A_\rho(l).A[L/x : T_x]_{\Gamma, x' : T'_x} F$$

Case 11 *Gvar-Gvar-Diff*

M has the form $zy_{\psi(1)} \cdots y_{\psi(m)}$ for some existentially quantified variable $z : \Pi v_1 B_1 \cdots \Pi v_m : B_m.B$ distinct from x and partial permutation ψ.

In this case, we can only transition if z is existentially quantified consecutively outside of x.

In this case, we perform a dual imitation.

Let ψ' and ϕ' be partial permutations such that for all i and j such that $\psi(i) = \phi(j)$ then there is some k such that $\psi'(k) = i$ and $\psi'(k) = j$

Then let the L, L' be as follows.

$$L = \lambda u_1 : A_1 \cdots \lambda u_n : A_n.x' u_{\phi'(1)} \cdots u_{\phi'(l)}$$

$$L = \lambda v_1 : B_1 \cdots \lambda v_m : B_m.x' u_{\psi'(1)} \cdots u_{\psi'(l)}$$

Then we can transition to

$$\Gamma[\exists x' : \Pi u_1 : A_{\phi'(1)} \cdots \Pi u_l : A_{\phi'(l)}.[L'/z : T_z]_{\Gamma, x' : T_{x'}}[L/x : T_x]_{\Gamma, x' : T_{x'}} F$$

While this case appears to be only rarely applicable, the "Raising" transition can be used to allow this rule to apply. Due to the restrictions of this case, and the potential for

non termination with unrestricted application of the "Raising" rule, it is the only case where the "Raising" rule is permitted.

3.4.4 Implementation

Because typed substitution is necessary, we must now keep track of types for existential variables. This can significantly complicate the implementation of the unification algorithm as the common technique of maintaining unbound existential variables with restrictions can no longer be blindly used, as existential variables must be maintained in the formula. If existential variables are maintained literally in the formula, the structure must provide the ability to add variables at both the top and bottom level. This complication if implemented naively can lead to a significantly less efficient structure. After experimentation, similar speeds have been observed when this structure is implemented as a zipper [38]. Unfortunately in this case, since variables are best implemented via DeBruijn indexes, variable reconstruction is no longer trivial. To reconstruct types, variable names might be included along side existential variable bindings. Again, this introduces significant complications. In order to perform substitution for an existential variable, the context of existential variables would have to have existential DeBruijn indexes continually swapped for their names. This is most readily implementable as having all existential variable instances also mention their names.

Another potential option is to maintain the type of the existential variable with each mention of the existential variable. While in this situation it is simplest to implement reconstruction, types might be enormous and it would be preferable to mention them only once.

The last option is to perform unification with untyped substitution in certain cases. While there is no proof at the moment that unification on the pattern subset of the "Calculus of Constructions" with untyped substitution for only the existential substitutions

is total, it is not unbelievable. Furthermore, omitting typed substitution does not alter the correctness of the algorithm, only the potential totality.

Ideally, knowledge that type checking terminated would be convincing enough so it is not necessary to continue with the reconstruction. However, reconstruction is necessary for implementing the multi-pass proof search described previously. Furthermore, reconstruction is useful since the exposed typing rules are incoherent, meaning there could be multiple coercions to demonstrate a subtyping relation. In these cases, it is desirable to see what was inferred by type inference.

3.5 Proof Search

In a traditional logic programming language, the order of declaration of quantified arguments is irrelevant, and the context can be considered an unordered set (even though for implementation reasons it is not). In a dependently typed logic programming language where types direct proof search, types must be maintained in the context and the context thus must maintain order. Since search dynamically poses unification problems, which may not be entirely solvable until later in the search, unification and proof search can be made to be naturally mutually recursive procedures when terms of the form $T \in T$ are permitted in the logic. As it is important to maintain the mixed quantifier prefix throughout proof search, it is desirable to provide a version of the algorithm where unification and proof search are not distinct procedures. Pfenning et al. [60] gave a succinct formulation where inhabitance and immediate implication were represented directly in the unification calculus.

3.5.1 Search

Definition 3.5.1 *Unification Calculus with Search*

$$U ::= U \wedge U \mid \forall V : T.U \mid \exists V : T.U \mid U \doteq U \mid \top \mid T \in T \mid T \in T >> T \in T$$

The following new transformations are added to represent proof search:

$$G_{\Pi} : M \in \Pi x : A.B \longrightarrow \forall x : A.\exists y : B.y \doteq Mx \wedge y \in B \tag{3.14}$$

$$G^1_{\mathsf{Atom}} : \forall x : A.F[M \in C] \longrightarrow \forall x : A.F[x \in A >> M \in C]$$

where C is an atomic type (3.15)

$$G^2_{\mathsf{Atom}} : F[M \in C] \longrightarrow \forall x : A.F[c_0 \in A >> M \in C]$$

where $c_0 : A$ is a constant and C is an atomic type (3.16)

$$D_{\Pi} : F[N \in \Pi x : A.B >> M \in C] \longrightarrow F[\exists x : A(Nx \in B >> M \in C) \wedge x \in A] \tag{3.17}$$

$$D_{\mathsf{Atom}} : F[N \in aN_1 \cdots N_n >> M \in aM_1 \cdots M_n] \longrightarrow F[N_1 \doteq M_1 \wedge \cdots \wedge N_n \doteq M_n \wedge N \doteq M] \tag{3.18}$$

3.5.2 Proof Sharing

In a pure setting, significant improvements to the efficiency of the implementation can be made by extending the quantifiers of the unification calculus to include forced inhabitant existential quantification.

$$U ::= U \wedge U \mid \forall V : T.U \mid \exists V : T.U \mid \exists_f V : T.U \mid U \doteq U \mid \top \mid T \in T >> T \in T$$

$$G^1_{\mathsf{Atom}} : \forall x : A.F[\exists_f V : T.\top] \longrightarrow \forall x : A.F[x \in A >> M \in C]$$

$$G^2_{\mathsf{Atom}} : \exists_f x : A.F[\exists_f V : T.\top] \longrightarrow \forall x : A.F[x \in A >> M \in C]$$

$$D_\Pi : N \in \Pi x : A.B >> M \in C \longrightarrow \exists_f x : A(Nx \in B >> M \in C)$$

In this situation, it is permitted to use the results of future searches for the solution of the current search. While this sharing is optimal from an operational standpoint, it can make reasoning about the behavior of impure logic programs very difficult. Given that Caledon is an impure programming language, reasoning about program behavior comes before optimizing proof search. It is the subject of future research to determine proof sharing techniques that do not interfere with effects.

Chapter 4

Type Inference

In this section, I introduce the type inference system for Caledon. I first discuss the inference rules and erasure form of $CICC$ dubbed $CICC^{-}$, a system based on the "Implicit Calculus of Constructions" (ICC) [67]. I then introduce a unification algorithm for handling constraints generated by $CICC^{-}$ and finally describe the construction of these constraints and the elaboration technique.

ICC is an extension to the standard "Calculus of Constructions" which allows a declaration that in all uses of a function, the argument be omitted and chosen during typechecking based on a provability relation.

Standard CC, and even standard LF can be unnecessarily verbose, as seen in the example 4.1.

Ideally, one omits redundant types whose values are parameterized and can be inferred from context.

Omitting these types gives rise to the notion of an implicit type system. The Hindley-Milner [34] system for inferring principle types in system F is a special case of the system where implicit universally quantified type variables are automatically resolved.

```
1  defn churchList : prop -> prop
2    as \ A : prop . [lst : prop -> prop] ([C] lst C) -> (A -> [C] lst C -> lst
         C) -> [C] lst C
3
4  defn mapCL : [A : prop] [B : prop] (A -> B) -> churchList A -> churchList B
5    as \ A     : prop                    .
6       \ B     : prop                    .
7       \ F     : A -> B                  .
8       \ cl    : churchList A            .
9       \ lst   : prop -> prop            .
10      \ nil   : [B] lst B               .
11      \ cons  : B -> [B] lst B -> lst B .
12
13           cl lst nil (\v . cons (F v))
14
15 defn mapResult : churchList natural
16   as mapCL natural boolean (\ a : natural . isZero a) someList
```

Figure 4.1: Maping over the church encoding of a list

4.1 Implicit Calculus of Constructions

Miquel [55] provides a more general system than that seen in Hindley-Milner, ICC to allow for implicit arguments. Here, I briefly explain the system and some of the relevant theoretical results that have been obtained. Because maintaining flexibility is important to future extensions of Caledon, I present the implicit calculus in terms of Pure Type Systems.

$$E ::= V \mid S \mid E\ E \mid \lambda V.E \mid \Pi V : E.E \mid \forall V : E.E$$

Figure 4.2: Syntax of ICC

Miquel's presentation of ICC uses Curry-style λ bindings with types omitted. The typing rules for ICC are predominantly the same as those for Pure Type Systems, except that I provide an extra rule for abstraction, application, and formation of implicitly quantification. The abstraction rule also must conform to the syntax of the Curry-style λ bindings.

$$\frac{\Gamma, x : A \vdash_{ICC} F : B \quad \Gamma \vdash_{ICC} (\Pi x : A.B) : s \quad s \in S}{\Gamma \vdash_{ICC} (\lambda x.F) : (\Pi x : A.B)} \text{ abstraction}$$

$$\frac{\Gamma, x : T \vdash_{ICC} M : U \quad \Gamma \vdash_{ICC} (\forall x : T.U) : s \quad s \in S \quad x \notin FV(M)}{\Gamma \vdash_{ICC} M : (\forall x : T.U)} \text{ gen}$$

$$\frac{\Gamma \vdash_{ICC} M : \forall x : T.U \quad \Gamma \vdash_{ICC} N : T}{\Gamma \vdash_{ICC} M : U[N/x]} \text{ inst}$$

$$\frac{\Gamma \vdash_{ICC} A : s_1 \quad \Gamma, x : A \vdash_{ICC} B : s_2 \quad (s_1, s_2, s_3) \in R}{\Gamma \vdash_{ICC} (\forall x : A.B) : s_3} \text{ imp} - \text{prod}$$

$$\frac{\Gamma, x : T \vdash_{ICC} M : U \quad x \notin FV(M) \cup FV(U)}{\Gamma \vdash_{ICC} M : U} \text{ strength}$$

$$\frac{\Gamma \vdash_{ICC} \lambda x.(Mx) : T \quad x \notin FV(M)}{\Gamma \vdash_{ICC} M : T} \text{ ext}$$

Figure 4.3: Typing for ICC

In the formulation in 4.3, there is no way to control the type of the argument used explicitly. Similarly, there is no mechanism for this in the syntax shown in 4.2. In the implemented version, this is not the case, as a notion of explicit binding has been provided.

In addition, in the formulation, neither the strengthening rule nor the rule of exten-

sionality are admissible. These rules are necessary to show subject reduction.

4.1.1 Subtyping

Definition 4.1.1 ***Subtyping relation:*** $\Gamma \vdash_{ICC} T \leq T' \equiv \Gamma, x : T \vdash_{ICC} x : T'$

Lemma 4.1.2 ***(Subtyping is a preordering)***

$$\frac{\Gamma \vdash_{ICC} T : s}{\Gamma \vdash_{ICC} T \leq T}\ \mathsf{sym} \qquad \frac{\Gamma \vdash_{ICC} T_1 \leq T_2 \quad \Gamma \vdash_{ICC} T_2 \leq T_3}{\Gamma \vdash_{ICC} T_1 \leq T_3}\ \mathsf{trans} \qquad \frac{\Gamma \vdash_{ICC} M \leq T \quad \Gamma \vdash_{ICC} T \leq T'}{\Gamma \vdash_{ICC} M : T'}\ \mathsf{sub}$$

Lemma 4.1.3 *Domains of products are contravariant and codomains are covarient:*

$$\frac{\Gamma \vdash_{ICC} T' \leq T \quad \Gamma, x : T' \vdash_{ICC} U \leq U'}{\Gamma \vdash_{ICC} \Pi x : T.U \leq \Pi x : T'.U'} \qquad \frac{\Gamma \vdash_{ICC} T' \leq T \quad \Gamma, x : T' \vdash_{ICC} U \leq U'}{\Gamma \vdash_{ICC} \forall x : T.U \leq \forall x : T'.U'}$$

4.1.2 Results

There are two main results that follow from this calculus.

Theorem 4.1.4 ***(Subject Reduction)*** *If* $\Gamma \vdash_{ICC} M : T$ *and* $M \rightarrow_{\beta\eta *} M'$ *then* $\Gamma \vdash_{ICC} M' : T$

Definition 4.1.5 $\mathsf{Term}_{ICC} = \{M : \exists T, \Gamma . \Gamma \vdash_{ICC} M : T\}$

Because this calculus is Curry-style, Church-Rosser is provable. While the internal representation and external presentation of Caledon is not necessarily Curry-style, it is possible to mimic a Church-style encoding into a Curry-style encoding through the use of type ascriptions and evaluation-delaying terms. Technically, the calculus will no longer have the Church-Rosser property if evaluation-delaying terms are included. However, evaluation-delaying terms are ineffectual when added to a strongly normalizing calculus.

4.2 Inference for CICC

In order to make use of the implicit system of $CICC$, an inference relation must be provided. This is accomplished by extending the typing rules and providing a mapping from the extended type derivation and term to an original type derivation and term.

Let $\Gamma \vdash A : T \wedge B : T'$ stand for $\Gamma \vdash A : T$ and $\Gamma \vdash B : T'$.

Definition 4.2.1 ***($CICC^-$ Extended Typing Rules)***

$$\frac{\Gamma \vdash_{i^-} M :?\Pi n, x : T.U \quad \Gamma \vdash_{i^-} N : T \quad n \notin DV(\Gamma)}{\Gamma \vdash_{i^-} M : [N/x]U} \ \mathsf{inst/f}$$

$$\frac{\Gamma, x : T \vdash_{i^-} M : [N/x]U \wedge N : T \quad \Gamma \vdash_{i^-} (?\Pi n, x : T.U) : K \quad n \notin FV(M) \cup DV(\Gamma)}{\Gamma \vdash_{i^-} M :?\Pi n, x : T.U} \ \mathsf{abs/f}$$

$$\frac{\Gamma, x : T \vdash_{i^-} M : U \quad x \notin FV(M) \cup FV(U) \cup DV(\Gamma)}{\Gamma \vdash_{i^-} M : U} \ \mathsf{strength}$$

$$\frac{\Gamma \vdash_{i^-} M :?\Pi n, x : T.U \quad \Gamma \vdash_{i^-} N : T \quad n \notin GN(M) \quad n \notin BN(U)}{\Gamma \vdash_{i^-} M\{n = N\} : [N/x]U} \ \mathsf{inst/b}$$

In $CICC$, as in CC, the strengthening rule is admissible, while in $CICC^-$, it is not.

The rule abs/f might appear to not make sense at first glance since it abstracts to a known term, but it can be considered equivalent to an existential pack without the pack proof term, since $x \notin FV(M)$

Conversion is now restricted to β to accommodate the Church-Rosser theorem which is necessary to prove subject reduction.

We do not need semantic related properties and thus the semantics of $CICC^-$ is unimportant, since we will be elaborating to the sublanguage $CICC$ before evaluating and type checking further.

However, the substitution theorem holds.

Theorem 4.2.2 ***(Substitution)***

$$\frac{\Gamma, x : T_1, \Gamma' \vdash_{i^-} M : T_2 \quad \Gamma \vdash_{i^-} N : T_1}{\Gamma, [N/x]\Gamma' \vdash_{i^-} [N/x]M : [N/x]T_2} \ \mathsf{subst}$$

Theorem 4.2.3 ***(Subject Reduction)*** *If* $\Gamma \vdash_{i^-} M : T$ *and* $M \rightarrow_{\beta *} M'$ *then* $\Gamma \vdash_{i^-} M' : T$

4.2.3 is at the moment believed to be true, although no full formalization exists. Provided reductions are restricted to β conversion, the Church-Rosser theorem is simply provable and the proof of subject reduction is similar to that in the traditional "Calculus of Constructions."

Without the abs/f rule, subject reduction becomes unnecessary for the metatheory since the single direction subtyping relation is sufficient. However, unification becomes difficult to implement.

Unfortunately, the projection function now requires more information than is available syntactically, and thus must be given on the typing derivation.

Definition 4.2.4 ***(Projection from*** $CICC^-$ ***to*** $CICC$***)***

$$\left[\!\left[\frac{}{\cdot \vdash_{i^-}} \ \mathsf{wf/e}\right]\!\right]^{c}_{ci^-} := \cdot$$

$$\left[\!\left[\frac{\overset{\mathcal{D}}{\Gamma \vdash_{i^-} x : T} \quad \cdots}{\Gamma, x : T \vdash_{i^-}} \ \mathsf{wf/s}\right]\!\right]^{c}_{ci^-} := [\![\Gamma \vdash_{i^-}]\!]^{c}_{ci^-} \, , [\![\mathcal{D}]\!]_{ci^-}$$

$$\left[\!\left[\frac{\cdots}{\Gamma, x : A \vdash_{i^-} x : A} \ \mathsf{start}\right]\!\right]_{ci^-} := x$$

$$\left[\!\left[\frac{\cdots}{\Gamma, x : A \vdash_{i^-} c : s} \ \mathsf{axioms}\right]\!\right]_{ci^-} := c$$

$$\left[\!\left[\frac{\Gamma \vdash \overset{\mathcal{D}_1}{T} : s_1 \quad \Gamma, x : \overset{\mathcal{D}_2}{T} \vdash U : s_2 \quad \cdots}{\Gamma \vdash_{i^-} (\Pi x : T.U) : s} \ \mathsf{prod}\right]\!\right]_{ci^-} := \Pi x : [\![\mathcal{D}_1]\!]_{ci^-} \, . \, [\![\mathcal{D}_2]\!]_{ci^-}$$

$$\left[\!\left[\frac{\Gamma \vdash \overset{\mathcal{D}_1}{T} : s_1 \quad \Gamma, x : \overset{\mathcal{D}_2}{T} \vdash U : s_2 \quad \cdots}{\Gamma \vdash_{i^-} (?\Pi n, x : T.U) : s} \ \mathsf{prod}*\right]\!\right]_{ci^-} :=?\Pi n, x : [\![\mathcal{D}_1]\!]_{ci^-} \, . \, [\![\mathcal{D}_2]\!]_{ci^-}$$

$$\left[\!\!\left[\dfrac{\Gamma, x : T \overset{\mathcal{D}_1}{\vdash_{i^-}} M : U \qquad \dfrac{\Gamma \vdash \overset{\mathcal{D}_2}{T} : s_1 \quad \Gamma, x : T \overset{\mathcal{D}_3}{\vdash} U : s_2 \quad \cdots}{\Gamma \vdash_{i^-} (\Pi x : T.U) : s}\ \mathsf{prod} \quad \cdots}{\Gamma \vdash_{i^-} \lambda x : T.M : (\Pi x : T.U)}\ \mathsf{gen} \right]\!\!\right]_{ci^-} := \lambda x : [\![\mathcal{D}_2]\!]_{ci^-} . [\![\mathcal{D}_1]\!]_{ci^-}$$

$$\left[\!\!\left[\dfrac{\Gamma, x : T \overset{\mathcal{D}_1}{\vdash_{i^-}} M : U \qquad \dfrac{\Gamma \vdash \overset{\mathcal{D}_2}{T} : s_1 \quad \Gamma, x : T \overset{\mathcal{D}_3}{\vdash} U : s_2 \quad \cdots}{\Gamma \vdash_{i^-} (?\Pi n, x : T.U) : s}\ \mathsf{prod}* \quad \cdots}{\Gamma \vdash_{i^-} ?\lambda n, x : T.M : (?\Pi n, x : T.U)}\ \mathsf{gen}* \right]\!\!\right]_{ci^-} :=?\lambda n, x : [\![\mathcal{D}_2]\!]_{ci^-} . [\![\mathcal{D}_1]\!]_{ci^-}$$

$$\left[\!\!\left[\dfrac{\Gamma \vdash_{i^-} \overset{\mathcal{D}_1}{M} : \Pi x : T.U \quad \Gamma \vdash_{i^-} \overset{\mathcal{D}_2}{N} : T}{\Gamma \vdash_{i^-} MN : U[N/x]}\ \mathsf{app} \right]\!\!\right]_{ci^-} := [\![\mathcal{D}_1]\!]_{ci^-}\ [\![\mathcal{D}_2]\!]_{ci^-}$$

$$\left[\!\!\left[\dfrac{\Gamma \vdash_{i^-} M :\overset{\mathcal{D}_1}{?\Pi} n, x : T.U \quad \Gamma \vdash_{i^-} \overset{\mathcal{D}_2}{N} : T \quad \cdots}{\Gamma \vdash_{i^-} M\{n = N\} : U[N/x]}\ \mathsf{inst/b} \right]\!\!\right]_{ci^-} := [\![\mathcal{D}_1]\!]_{ci^-}\ \{n : [\![\Gamma \vdash T : \mathsf{kind}]\!]_{ci^-} = [\![\mathcal{D}_2]\!]_{ci^-}\}$$

$$\left[\!\!\left[\dfrac{\Gamma \vdash_{i^-} M :\overset{\mathcal{D}_1}{?\Pi} n, x : T.U \quad \Gamma \vdash_{i^-} \overset{\mathcal{D}_2}{N} : T \quad \cdots}{\Gamma \vdash_{i^-} M : U[N/x]}\ \mathsf{inst/f} \right]\!\!\right]_{ci^-} := [\![\mathcal{D}_1]\!]_{ci^-}\ \{n = [\![\mathcal{D}_2]\!]_{ci^-}\}$$

$$\left[\!\!\left[\dfrac{\Gamma, x : T \overset{\mathcal{D}}{\vdash_{i^-}} M : U \quad \cdots}{\Gamma \vdash_{i^-} M : U}\ \mathsf{strength} \right]\!\!\right]_{ci^-} := [\![\mathcal{D}]\!]_{ci^-}$$

$$\left[\!\!\left[\dfrac{\Gamma \vdash_{i^-} \overset{\mathcal{D}}{M} : [N/x]U \quad \cdots}{\Gamma \vdash_{i^-} M :?\Pi n, x : T.U}\ \mathsf{abs/f} \right]\!\!\right]_{ci^-} :=?\lambda n, x : T. [\![\mathcal{D}]\!]_{ci^-}$$

Theorem 4.2.5 *(Soundness of extraction)*

$$\Gamma \vdash_{i^-} \quad \Longrightarrow [\![\Gamma \vdash_{i^-}]\!]^c_{ci} \vdash_{i^-} \tag{4.1}$$

$$\Gamma \vdash_{i^-} A : T \implies [\![\Gamma \vdash_{i^-}]\!]^c_{ci} \vdash_{i^-} [\![\Gamma \vdash_{i^-} A : T]\!]_{ci} \tag{4.2}$$

Since $CICC$ permits η equivalence and $CICC^-$ does not, the extraction in the reverse direction is no longer sound. For our purposes, this is not objectionable since $CICC$ is known to be consistent and there is no reason to convert back into $CICC^-$, as it is used entirely as a pre-elaboration language. Once terms are typechecked and type inferred in $CICC^-$, they are typechecked in $CICC$ and normalized in $CICC$. While the reverse extraction is generally not sound, normal terms with normal types are clearly typeable in $CICC^-$.

4.2.1 Subtyping

Similar to ICC, these rules result in a subtyping relation, which will be of importance during type inference and elaboration.

Definition 4.2.6 *Subtyping relation:* $\Gamma \vdash_{i^-} T \leq T' \equiv \Gamma, x : T \vdash_{i^-} x : T'$ *where* x *is new.*

Lemma 4.2.7 *Subtyping is a preordering:*

$$\frac{\Gamma \vdash_{i^-} T : s}{\Gamma \vdash_{i^-} T \leq T}\ \mathsf{refl} \qquad \frac{\Gamma \vdash_{i^-} T_1 \leq T_2 \quad \Gamma \vdash_{i^-} T_2 \leq T_3}{\Gamma \vdash_{i^-} T_1 \leq T_3}\ \mathsf{trans}$$

$$\frac{\Gamma \vdash_{i^-} M \leq T \quad \Gamma \vdash_{i^-} T \leq T'}{\Gamma \vdash_{i^-} M : T'}\ \mathsf{sub}$$

This theorem is an application of the substitution lemma.

Lemma 4.2.8 *Domains of products are contravariant and codomains are covariant:*

$$\frac{\Gamma \vdash_{i^-} T' \leq T \quad \Gamma, x : T' \vdash_{i^-} U \leq U'}{\Gamma \vdash_{i^-} \Pi x : T.U \leq \Pi x : T'.U'} \qquad \frac{\Gamma \vdash_{i^-} T' \leq T \quad \Gamma, x : T' \vdash_{i^-} U \leq U'}{\Gamma \vdash_{i^-} \forall x : T.U \leq \forall x : T'.U'}$$

Unlike traditional subtyping relations where an explicit subtyping rule must be included in the type system, this system's subtyping relation is much easier to manage during unification, because it is simply a macro for a provability relation.

This allows one to implement higher order unification with minimal modification, as in a lattice unification algorithm. The modification is made to the search procedure, and subtyping constraints are realized as search terms.

However, with the addition of the strengthening rule, this kind of modification is not entirely necessary, since it is provable that this subtyping relation is symmetric 4.2.9, and thus an entirely symmetric unification algorithm can be presented.

Theorem 4.2.9 is not obvious at first glance, so I will provide intuitive justification first.

In $CICC$, by uniqueness of types, $\Gamma \vdash x : A$ and $\Gamma \vdash x : B$ implies $A \equiv_{\beta\eta *} B$. In $CICC^{-}$ however, there is no such uniqueness of types properties. Rather, the inst/f and abs/f rules permit one to respectively add and initialize an implicit argument, then abstract implicitly upon an unused argument.

Thus if $\Gamma, x :?\Pi n, z : T.A \vdash_{i^-} x : A$ by implicit instantiation of the argument n, we might also derive $\Gamma, x : A \vdash_{i^-} x :?\Pi n, z : T.A$ given that $z \notin FV(x)$ and that $\Gamma, x :?\Pi n, z : T.A \vdash_{i^-} x : A$ implies $\Gamma, x :?\Pi n, z : T.A \vdash_{i^-}$ which implies $\Gamma \vdash_{i^-} x : (?\Pi n, z : T.A) : K$.

Theorem 4.2.9 ***(Symmetry)*** *$\Gamma \vdash_{i^-} A \leq B$ implies $\Gamma \vdash_{i^-} B' \leq A'$. where $A \equiv_\beta A'$ and $B \equiv_\beta B'$*

Proof: This is proved by induction on the structure of the proof of $\Gamma, x : A \vdash_{i^-} x : B$. Here I only consider the cases relevant to the new fragment.

Case 1 *We begin with the non admissible strengthening rule.*

$$\frac{\Gamma, x : A, z : T \vdash_{i^-} x : B \qquad z \notin FV(x) \cup FV(B) \cup DV(G)}{\Gamma, x : A \vdash_{i^-} x : B}\ \text{strength}$$

from this we can derive via the induction hypothesis, $\Gamma, x : B', z : T \vdash_{i^-} x : A'$ and then reapply strengthening.

$$\frac{\Gamma, x : B', z : T \vdash_{i^-} x : A' \qquad z \notin FV(x) \cup FV(A') \cup DV(G)}{\Gamma, x : B' \vdash_{i^-} x : A'}\ \text{strength}$$

Case 2 *In this case we cover implicit instantiation.*

$$\frac{\Gamma, x : A \vdash_{i^-} x :?\Pi n, z : T.B \qquad \Gamma, x : A \vdash_{i^-} N : T \qquad z \notin DV(\Gamma, x : A)}{\Gamma, x : A \vdash_{i^-} x : [N/z]B}\ \text{inst/f}$$

Suppose z is not in $FV(B)$ then $[N/z]B \equiv B$. From this the following proof is possible:

The first steps are the following few derivations:

$$\frac{\Gamma, \vdash_{i^-} B' : K \qquad z \notin FV(B') \cup FV(K)}{\Gamma, z : T', \vdash_{i^-} B' : K}\ \text{stren}$$

$$\frac{\Gamma \vdash_{i^-} T' : K' \qquad \Gamma, z : T' \vdash_{i^-} B' : K}{\Gamma \vdash_{i^-} ?\Pi n, z : T'.B' : K}\ \text{form/f}$$

$$\frac{B \equiv_\beta B' \qquad z \notin B'}{z \notin FV(B')}$$

From these, we can derive the following result about B'.

$$\frac{\dfrac{\Gamma, z : T' \vdash_{i^-} B' : K}{\Gamma, z : T', x : B' \vdash_{i^-} x : B'}\ \text{start} \qquad \Gamma \vdash_{i^-} ?\Pi n, z : T'.B' : K \qquad z \notin FV(x) \cup DV(\Gamma)}{\Gamma, x : B' \vdash_{i^-} x :?\Pi n, z : T'.B'}\ \text{abs/f}$$

Finally, we can derive the desired result:

$$\frac{\dfrac{\Gamma, x : B' \vdash_{i^-} x :?\Pi n, z : T'.B' \qquad \dfrac{IH(\Gamma, x : A \vdash_{i^-} x :?\Pi n, z : T.B)}{\Gamma, x :?\Pi n, z : T'.B' \vdash_{i^-} x : A'}}{\Gamma, x : B' \vdash_{i^-} x : A'}\ \text{subst} \qquad z \notin FV(B')}{\Gamma, x : [N/z]B' \vdash_{i^-} x : A'}$$

The only unexplained axioms here are $\Gamma \vdash_{i^-} B' : K$ and $\Gamma \vdash_{i^-} T' : K$ in this proof. Because $\Gamma, x : A' \vdash_{i^-} x :?\Pi n, z : T'.B'$ is true, we know that $\Gamma, x : A' \vdash_{i^-} ?\Pi n, z : T'.B' : K$ and thus that $\Gamma, x : A' \vdash_{i^-} ?\Pi T' : K'$ and $\Gamma, x : A', z : T' \vdash_{i^-} B' : K$. By strengthening we can infer $\Gamma \vdash_{i^-} B' : K$ and $\Gamma \vdash_{i^-} T' : K$.

On the other hand, if z is in $FV(B)$ we achieve different proofs. Now we can infer that $x \notin FV(N)$, but we can not show that $[N/z]B' \equiv B'$.

By the induction hypothesis, we can infer $\Gamma, x :?\Pi n, z : T'.B' \vdash_{i^-} x : A'$.

First, we know that $\Gamma, x : A \vdash_{i^-} [N/z]B : K$ by well formedness of the judgement $\Gamma, x : A \vdash_{i^-} x : [N/z]B'$ and the conversion rule.

$$\dfrac{\dfrac{\dfrac{\Gamma, x : A \vdash_{i^-} [N/z]B' : K \qquad x \notin FV(B')}{\Gamma \vdash_{i^-} [N/z]B' : K}\ \text{strength}}{\Gamma, x : [N/z]B' \vdash_{i^-} x : [N/z]B'}\ \text{start} \qquad z \notin FV(N)}{\Gamma, x : [N/z]B', z : T \vdash_{i^-} x : [N/z]B'}\ \text{strength}$$

thus, we can use the abs/f rule to construct a form we can use in substitution.

$$\dfrac{\Gamma, x : [N/z]B', z : T \vdash_{i^-} (x : [N/z]B') \wedge N : T' \qquad \Gamma, x : [N/z]B' \vdash_{i^-} ?\Pi n, z : T'.B' : K \qquad z \notin GV(M; \Gamma)}{\Gamma, x : [N/z]B' \vdash_{i^-} x :?\Pi n, z : T'.B'}$$

where $GV(M; \Gamma) = FV(M) \cup DV(\Gamma)$.

Finally, we get the following derivation.

$$\dfrac{\Gamma, x : [N/z]B' \vdash_{i^-} x :?\Pi n, z : T'.B' \qquad \Gamma, x :?\Pi n, z : T'.B' \vdash_{i^-} x : A'}{\Gamma, x : [N/z]B' \vdash_{i^-} x : A'}\ \text{subst}$$

Case 3 *In this case we examine the* abs/f *rule.*

$$\dfrac{\Gamma, x : A, z : T \vdash_{i^-} x : [N/z]B \wedge N : T \qquad \Gamma, x : A \vdash_{i^-} ?\Pi n, z : T'.B' : K \qquad z \notin GV(M; \Gamma, x)}{\Gamma, x : A \vdash_{i^-} x :?\Pi n, z : T.B}\ \text{abs/f}$$

From this we can infer that $z \notin FV(A)$. This is useful since we can derive:

$$\Gamma, z : T, x : A \vdash_{i^-} x : [N/z]B \wedge N : T$$

We can then apply the induction hypothesis to get the following:

$$\Gamma, z : T, x : [N'/z]B' \vdash_{i^-} x : A'$$

From this we can infer $\Gamma, x : [N'/z]B' \vdash_{i^-} x : A'$ by strengthening since $z \notin FV(x) \cup FV(A')$.

Furthermore, we can infer that $\Gamma \vdash_{i^-} N' : T'$ since $N' \equiv_\beta N$ so $\Gamma \vdash_{i^-} N' : T$ by subject reduction and $T' \equiv_\beta T$ so $\Gamma \vdash_{i^-} N' : T'$ by conversion.

We can also derive $\Gamma, x :?\Pi n, z : T'.B' \vdash_{i^-} x :?\Pi n, z : T'.B'$ by the start rule.

We get the following proof:

$$\frac{\Gamma, x :?\Pi n, z : T'.B' \vdash_{i^-} x :?\Pi n, z : T'.B' \qquad \Gamma \vdash_{i^-} N' : T' \qquad z \notin DV(\Gamma)}{\Gamma, x :?\Pi n, z : T'.B' \vdash_{i^-} x : [N'/x]B'} \text{ inf/f}$$

Finally, with the knowledge that $z \notin FV(A')$, we can derive the following:

$$\frac{\Gamma, x :?\Pi n, z : T'.B' \vdash_{i^-} x : [N'/x]B' \qquad \Gamma, x : [N'/x]B' \vdash_{i^-} x : A'}{\Gamma, x :?\Pi n, z : T'.B' \vdash_{i^-} x : A'} \text{ subst}$$

□

4.3 Semantics for $CICCI$

4.3.1 Substitution With Implicits

The formulation of hereditary substitution in the presence of implicit arguments is not unlike the presentation of hereditary substitution without implicit arguments with additional required checks.

Definition 4.3.1 ***(Implicit Typed Hereditary Substitution)***

$$[S/x : A]^n_\Gamma(?\lambda y : B.N) :=?\lambda y : B.[S/x : A]^n_{\Gamma,y:B}N$$

$$\eta^{-1}_{?\Pi x:A.B}(N) :=?\lambda x : A.N\ \{x = \eta^{-1}_A(x)\}$$

since N being typeable by $?\Pi x$ means that x can not appear free in N

$$\mathsf{H}_\Gamma(P \downarrow ?\Pi y : B_1.B_2, \{v := N\}) := P\ \{v := N\} \downarrow [N/y : B_1]^n_\Gamma B_2$$

$$\mathsf{H}_\Gamma((?\lambda v : A_1.N) \uparrow ?\Pi v : A_1.A_2, \{v := P\}) := [P/v]^n_{\Gamma\vdash v:A_1}N \uparrow A_2$$

$$\mathsf{H}(?\lambda v : T.P \uparrow \_, A) :=?\lambda v : T.\mathsf{H}(P, A)$$

4.3.2 Unification With Implicits

Now we can use the convenient fact that $\Gamma \vdash A \leq B$ implies $\Gamma \vdash B \leq A$ to extend the unification rules provided before to apply to $CICC^{-}$.

Case 1 *?Lam-?Lam-same*

$$F[?\lambda n, x : A.M \doteq ?\lambda n, y : A.N] \longrightarrow$$
$$F[\forall x : A.M \doteq [x/y]N] \quad (4.3)$$

Note that in this rule, the external name on the left matches the external name on the right.

Case 2 *?Forall-?Forall-same*

$$F[?\Pi n, x : A.M \doteq ?\Pi n, y : A'.N] \longrightarrow$$
$$F[A \doteq A' \wedge \forall x : A.M \doteq [x/y]N] \quad (4.4)$$

In this rule the external name on the left must match the external name on the right.

Case 3 *?Lam-?Lam-same*

This case is distinct from the simple case of equality of dependent products in that an implicit abstraction could lie at the head of the spine if it had not already been constrained.

$$F[(?\lambda n, x : A.M)\ R_1 \cdots R_n \doteq (?\lambda n, y : A'.N)\ R'_1 \cdots R'_m] \longrightarrow$$
$$F[A \doteq A' \wedge \forall x : A.H(\cdots H(M, R_1) \cdots R_n) \doteq H(\cdots H([x/y]N, R'_1) \cdots, R'_n)] \quad (4.5)$$

Again, the external name on the left must match the external name on the right.

Case 4 *?Lam-Unbound*

if $n \notin BN(N)$ then

$$F[(?\lambda n, x : A.M)\ R_1 \cdots R_n \doteq N] \longrightarrow$$
$$F[\exists x : A.H(\cdots H(M, R_1) \cdots R_n) \doteq N \wedge x \in A] \quad (4.6)$$

Here we perform a search for $x \in A$ to satisfy the `inst/f` rule. Rather than adding a potentially unused constriction to the right hand side, we observe that such a constriction could be inferred implicitly.

Case 5 *Uvar-Uvar-BothConst*

Rather than simply adding a case for universal variables with the possibility of a constriction in the argument list, we must modify the already existing case to "search" for constrictions on both sides. We first have a case which matches constrictions on both sides.

Suppose $n \notin CN(y\ N_1 \cdots N_n) \cup CN(y\ M_1 \cdots M_n)$

$$F[\forall y : A.G[y\ M_1 \cdots M_{r-1}\ \{n = A\}\ M_r \cdots M_n \doteq yN_1 \cdots N_{r'-1}\ \{n = A'\}\ N_{r'} \cdots N_n]] \longrightarrow$$
$$F[\forall y : A.G[yM_1 \cdots M_n \doteq yN_1 \cdots N_n \wedge A \doteq A']] \quad (4.7)$$

Case 6 *Uvar-Uvar-OneConst*

This case matches when there is only a constriction on one side.

Suppose $n \notin CN(y\ N_1 \cdots N_n) \cup CN(y\ M_1 \cdots M_n)$

$$F[\forall y : A.G[y\ M_1 \cdots M_{r-1}\ \{n = A\}\ M_r \cdots M_n \doteq yN_1 \cdots N_n]] \longrightarrow$$
$$F[\forall y : A.G[yM_1 \cdots M_n \doteq yN_1 \cdots N_n]] \quad (4.8)$$

In this case we have no reference point to unify A against, and we do not know its type, so we can simply ignore it.

Case 7 *Uvar-Uvar-Eq*

If $CN(y\ N_1 \cdots N_n) \cup CN(y\ M_1 \cdots M_n) = \emptyset$

$$F[\forall y : A.G[yM_1 \cdots M_n \doteq yN_1 \cdots N_n]] \longrightarrow$$
$$F[\forall y : A.G[M_1 \doteq \wedge N_1 \cdots \wedge M_1 \cdots N_n]] \quad (4.9)$$

This last case behaves exactly as the old "Uvar-Uvar-Eq" except that we require there to be no constrained names on either side.

Chapter 5

Implementation

Implementation of Caledon has a few unique properties, not all related to the exposed logic of the language. In this chapter, I discuss some details of the specification and implementation of Caledon. The algorithm for actually performing higher order unification and type inference is unusual in Caledon, because Caledon uses a zipper-style context implemented by a finger-tree based sequence, and does not perform linear passes on the unification problem. I define families as a coherent set of axioms for proof search. Nondeterminism control is discussed as a way of letting the programmer choose between sequential and concurrent execution and between efficient and complete searches. Finally, I define methods of interacting with the world.

5.1 Type Inference

While the generation of unification problems in Caledon is straightforward as described in the previous sections, the implementation of the higher order pattern unification algorithm is convoluted. If unification were to be implemented in Ollibot [64], a much simpler, yet significantly less efficient implementation might be possible.

Universally quantified and lambda quantified variables are easily represented by

DeBruijn indexes, since no universal quantifiers are ever introduced between two pre-existing universal quantifiers. On the other hand, existential variables are introduced at any location. Rather than complicating substitutions, existential variable instances are represented by unique names with depth indexes, instead of height indexes. Because existential variables are not explicitly named in the context, each existential variable instance carries its own type, lifted to the location of the instance.

Since such a representation requires traversals and modifications of a tree structure, a natural solution is to use a zipper to represent the entire structure. Because traversing the structure downward also builds a context of universal variables and passed conjunctive paths, the same zipper structure can likewise be used as the type context for DeBruijn variable lookup. In an imperative language with effects, one might think that using a vector to hold the zipper context of the unification problem is optimal. However, since nondeterminism is essential to proof search, complications arise if the structure is shared among threads manually. A pure data structure based on finger trees [35] known as a sequence turns out to be an ideal choice of structure. Concatenation in this structure is constant time, and splitting and lookup are logarithmic. While logarithmic lookup time is slightly worse than the constant lookup time for a vector, its benefit is that it automatically shares relevant unchanged sections between threads.

5.2 Type Families

Permitting entirely polymorphic axioms at the top level significantly complicates efficient proof search and can introduce unintended falsehoods. To remedy this, axioms are grouped together by conclusion, ensuring program definitions are local and not spread out across the source code. Grouping axioms in this way optimizes a proof search, because it is now possible to limit the search for axioms to the same family of the head of the goal. Such grouping thus acts as an automated and enforced version of the freeze

command from Elf. Mutually recursive definitions are automatically inferred.

$$\frac{\Pi x : T_1.T_2\mathsf{ty}}{T_2\ \mathsf{ty}}\ \Pi - \mathsf{ty}$$

$$\frac{?\Pi x : T_1.T_2\mathsf{ty}}{T_2\ \mathsf{ty}}\ ?\Pi - \mathsf{ty}$$

$$\frac{}{\mathsf{prop\ ty}}\ \mathsf{prop} - \mathsf{ty}$$

$$\frac{T_2@n \quad x \neq n}{\Pi x : T_1.T_2@n}\ \Pi - \mathsf{fam}$$

$$\frac{T_2@n \quad x \neq n}{?\Pi x : T_1.T_2@n}\ ?\Pi - \mathsf{fam}$$

$$\frac{}{n@n}\ \mathsf{var} - \mathsf{fam}$$

$$\frac{T_1@n}{T_1\ T_2@n}\ \mathsf{app} - \mathsf{fam}$$

$$\frac{T_1@n}{T_1\ \{x = T_2\}@n}\ ?\mathsf{app} - \mathsf{fam}$$

Figure 5.1: The family relationship

In figure 5.2, using the fam relation ensures that an axiom belongs to a family, whereas the ty relation ensures that a family type actually results in a type.

Grouping axioms together as families and preventing entirely polymorphic results ensures that entirely polymorphic results will be consistent. It is possible to relax this constraint at the expense of full program speed. Speed results because the only time

a polymorphic axiom is introduced into the proof search context is locally within an axioms assumptions.

5.3 Controlled Nondeterminism

A logic program need not only be a deterministic depth first pattern search. For purely declarative axioms, the depth first strategy is usually, in fact, incomplete and not representative of what should and should not halt. Depth first search, however, can be more efficient in many cases and when used intentionally will constrict nondeterminism. Concurrency in a program with IO has well known uses and advantages. The breadth first proof search strategy can conveniently be represented by a program where every pattern-match forks and then executes concurrently. This is not ideal if used indiscriminately. An ideal implementation allows one to control the patterns that are searched in parallel and in sequence. In the following snippet of the code, the distinction between breadth first and depth first queries are used to emulate the concept of concurrency and cause the program to have more complex behavior.

```
1
2 query main = runBoth false
3
4 defn runBoth : bool -> type
5   >| run0 = runBoth A
6            <- putStr ''ttt ''
7            <- A =:= true
8    | run1 = runBoth A
9            <- putStr ''vvvv''
10           <- A =:= true
11   | run2 = runBoth A
12           <- putStr ''qqqq''
13           <- A =:= true
14  >| run3 = runBoth A
15           <- putStr '' jjj ''
16           <- A =:= false
```

Figure 5.2: Nondeterminism control

In example 5.2, the query main prints to the screen something similar to ttt vqvqvqvq jjj. This happens because, despite proof search failing on the first three axioms due to an incorrect match, the fail is deferred until after IO has been performed. The middle axioms, "run1" and "run2" are declared to be breadth first axioms, while "run0" and "run3" are declared to be depth first axioms. The declaration of an axiom as being depth first implies that it's entire tree must be searched for a successful proof before the next axiom can be attempted. While breadth first axioms make no such guarantee, it is also not guaranteed that they will run concurrently. If for example they both make calls to predicates which each only have a single depth first axiom, they will not be any more concurrent than before.

```
1
2 defn fail : type
3   as true =:= false
4
5 defn then : type -> type -> type
6   >| then-A = Fst `then` Snd <- Fst <- fail
7   >| then-B = Fst `then` Snd <- Snd
8
9 query main = print ``Hello `` `then` print ``world!''
```

Figure 5.3: Sequential predicate

The predicate then in figure 5.3 executes its first routine and subsequently its second sequentially. To understand how this works, it is helpful to step through the query main. When main is called, it will first initiate a goal of the form then (print Hello) (print world!). Because the axioms then-A and then-B are both preceded by "¿—", the then-A is attempted, the search of which is constrained to be entire. Since (print Hello) succeeds and does not nondeterministically branch, fail will be initiated as the next goal. "fail" will certainly fail, and the current branch will end. then-B is the next attempt. It will succeed since (print world!) succeeds.

```
1
2 defn while : type -> type -> type
3     | while-A = Fst `while` Snd <- Fst
4     | while-B = Fst `while` Snd <- Snd
5
6 query main = print ``aaaa'' `while` print ``bbbb''
```

Figure 5.4: Concurrent predicate

The predicate while in figure 5.4 executes its first routine at the same time as its

second. The result of main might then be abababab. It is important to note that in this notion of concurrency, neither thread is the "original" thread.

5.4 IO and Builtin Values and Predicates

This section is both a specification and a guide to future implementers of logic programming languages with proof search.

When programming in Caledon, searching for items of type prop might uncover IO, and thus IO can be performed during typechecking. This can be understood as the set of axioms differing based on the environment available.

IO is performed when the evaluation function encounters a query for a built-in IO performing function.

```
1 eval : Proposition -> Environment Formula
2 eval (a ∈ ''print'' Str) = ( if gvar a then print str else ()
3                                        ; return (a ≐ printImp Str)
4                                        )
```

It is important to include the check that a has not already been resolved so that repeated IO actions are not performed when nondeterministically proof searching.

It is tempting to define predicates that take input as "taking as an argument a function that uses the input." This is in fact a valid way to define such functions and permits for hints of directionality in the types. However, it is still possible to escape from the confines of abstraction to build a predicate without obvious input directionality.

```
1 builtin readLine : (string -> prop) -> prop
2
3 defn readLinePredicate : string -> prop
4   as \ s : string . readLine (\ t . t =:= s )
```

Ensuring variables do not escape their intended scope is necessary to ensuring that

the intended IO action is only executed once, and not multiple times during proof search.

While it is possible to reason about nondeterministic IO, it is desirable to also have actions that cannot be executed twice, for which nondeterminism is not possible.

For this, the notion of a monad is useful. In this setting, IO is presented as a series of built-in axioms.

```
1 defn io : prop -> prop
2     | bind      = io A -> (A -> io B) -> io B
3     | return    = A -> io A
4     | ioReadLine = io string
5     | ioPrint     = string -> io unit
```

We can now no longer write $A \in \mathsf{readLine}$ though since $\mathsf{readLine} : \mathsf{prop}$ is a value and thus has no inhabitants.

Instead, an interpretation predicate can be created which maps these dummy IO actions to real actions.

```
1 defn run : io A -> A -> prop
2   >| runBind = run (bind IOA F) V
3              <- run IOA A
4              <- run (F A) V
5   >| runReturn    = run (return V) V
6   >| runReadLine = run ioReadLine A <- readLineEscape A
7   >| runPrint     = run (ioPrint S) one <- print S
```

Since the type system is the "Calculus of Constructions," IO actions constructed from io will be total, severely limiting their utility. More complex IO actions and interpreters can be generated, most importantly, ones involving recursion or infinite loops.

Chapter 6

Programming with Caledon

6.1 Typeclasses

As previously written, implicit arguments alongside polymorphism and proof search can subsume Haskell-style type classes.

The easiest way to see this is in an implementation of what is known as the "Show" type class in Haskell. In a logic programming language, a predicate that can be used to print a data type can also be used to read a data type, so here we shall discuss a "serialize" type class.

```
1
2 defn serializeBool : bool -> string -> type
3   >| serializeBool-true = serializeBool true ''true''
4   >| serializeBool-false = serializeBool false ''false''
```

Figure 6.1: Serializing booleans

```
query readQuery = exists B : bool. serializeBool B ''true''
query printQuery = exists S : string . serializeBool false S
```

Figure 6.2: Bidirectional serializing

Given the predicate 6.1 executes its matches in parallel, both of the queries in 6.2 will resolve.

The serialize predicate is a useful one, and we would like it to be polymorphic in all types for which we implement a serialize function. This is possible using implicit arguments.

We first create an open type for the type class serializable.

```
open serializable : [ T ] { serializer : T -> string -> type } type
```

Figure 6.3: The type of the type class serializable

We then define a function "serialize" which unpacks the implicit dependency of the type serializable.

```
defn serialize : {T}{ serializable : T -> string -> type } T -> string ->
    type
    | serializeImp =
        [ Serializer : T -> string -> type ]
        [ Serializable : serializable T { serializer = Serializer }]
        serialize { serializable = Serializable } V S
      <- Serializer V S
```

Figure 6.4: The implementation of the function serialize

```
instance serialize-bool = serializable bool { serializer = serializeBool }
instance serialize-nat = serializable nat { serializer = serializeNat }
```

Figure 6.5: Instances of serializable

To implement an instance of the serializable type class, one adds an instance axiom to the environment as in 6.6

Use of the function then omits the implementation of the "serializable" argument and type argument, such that they might be resolved automatically as in **??**

```
1 query readQueryBool = exists B . serialize B ''true''
2 query printQueryBool = exists S . serialize false S
3
4 query printQueryNat = exists S . serialize (succ (succ zero)) S
5 query readQueryNat = exists S : nat . serialize S ''(succ (succ zero))''
```

Figure 6.6: Instances of serializable

This process can be extended to open type classes which bypass the family requirement, allowing future instances to be declared. It is difficult to fully discuss uses of this capability in the confines of this paper. However, type class computation has been known to the Haskell community for quite some time. Moreover, it has been used in applications ranging from embedding an imperative computation monad with local variable use and assignment rules similar to those of C, to an RPC framework which creates end points based on functions with arbitrarily complex type signatures.

6.2 Linear Predicates

One major drawback of the type class paradigm outlined in the previous section is the inability for a typeclass to uniquely determine membership of type in a type class based on floating predicates in the environment with matching signatures. While ideal behavior is possible for theorems in the "Calculus of Constructions" which exhibit ideal parametricity, the type type has the trivial inhabitant type. Thus implementations will nearly always resolve to this version.

```
1 defn serializeable : [T] { serializer : T    > string    > type } type
2    | serializable-auto-find = [T] [ Serializer : T -> string -> type ]
3       serializable T { serializer = Serializer }
```

Figure 6.7: This would be nice

For example, it would be very nice if the predicates in figure 6.7 automatically resolved to a reasonable instance of serializable.

```
1 defn serialize : {T} { serializer : T    > string    > type } T -> string ->
     type
2   as ?\T : type . ?\ serializer : T -> string -> type . \v : T . \s : string
3    . serializer v s
```

Figure 6.8: Nice implementation

We could implement serialize as above in the definition in figure 6.8. However, this kind of implementation will fail to determine the correct instance of "serializer."

```
1 query writeBool = serialize true ''true''
2 ===>
3 query writeBool =
4    serialize
5        {T = bool}
6        { serializer = \ x : bool . \ y : string . type }
7        true ''true''
```

Figure 6.9: Trivial Failure

Rather, it will infer a trivial predicate, as seen in the query in figure 6.9

```
1 defn show : {T}{shower : T -> string } T -> string
2   as ?\T : type . ?\ shower : T -> string . \ v : T . shower v
3
4 query writeBool = print (show true)
5 ===>
6 query writeBool = print (show {T = bool}{shower = \ x : bool . nil })
```

Figure 6.10: Functions also fail

One might think that functions, as an alternative to predicates, are immune to inhabitation by trivial and incorrect values as in the above scenario. However, unless specified with their properties (tedious), functions have similar drawbacks, as is seen in the program in figure 6.10.

```
1  defn show : {T}
2              {shower : T -> string}
3              {reader : string -> maybe T}
4              {comp1  : [v] reader (shower v) =:= just v}
5              {comp2  : [v] fromJust (reader s) (\x . shower x =:= s) type}
6              T -> string
7    as ?\ T : type
8     . ?\ shower : T -> string
9     . ?\ reader : _
10    . ?\ comp1  : _
11    . ?\ comp2  : _
12    . \ v : T
13    . shower v
```

Figure 6.11: Kind of a success using proofs

One can sometimes work around this by including metatheorems about the implicit functions, as in the figure 6.11. However, proving the metatheorems is often tedious,

impossible, or sometimes just plain slow for the compiler.

One method under investigation to solve this ambiguity problem uses substructural dependent quantification, in which types indicate that the function argument can be used only once in the term, but unlimited times in types. While this is a subject of ongoing work on my part, I have included a description of my ideas.

```
1  defn serializeBool : bool -o string -o type
2     | serializeBool-true = serializeBool true ''true''
3     | serializeBool-false = serializeBool false ''false''
4
5  defn serialize : {T}{ serializer : T -o string -o type } T -> string -> type
6    as ?\ T
7     . ?\ serializer : T -o string -o type ]
8     .  \ v : T
9     .  \ s : string
10    . serializer v s
```

Figure 6.12: Linear types would be useful here

$F : A \multimap B$ shall mean that the function F only consumes a single resource of type A. $F : \forall_o x : A.B$ shall mean the same in a dependent setting.

The problem is solved in the figure 6.12 since the only function which linearly consumes a single boolean and a single string in the program and outputs a type is "serializeBool". Other functions that do this might be added later, but functions of the form $(\lambda x : \mathsf{bool}.\lambda s : \mathsf{string}.\mathsf{type})$ are not possible. Unfortunately, something along the lines of $(\lambda x : \mathsf{bool}.\lambda s : \mathsf{string}.\mathsf{isString}s \wedge \mathsf{isBool}x)$ might be possible, but these are significantly more manageable provided one is careful.

Fortunately, the fact that we are working with higher order abstract syntax in the "Calculus of Constructions" means that linear dependent products are actually implementable within Caledon.

```
1  defn restriction : type
2     | linear = restriction
3     | unused = restriction
4
5  defn restrictor : restriction -> restriction -> restriction -> type
6     | restrictor-linear1 = restrictor linear unused linear
7     | restrictor-linear2 = restrictor linear linear unused
8     | restrictor-unused = restrictor unused unused unused
9
10
11 defn restrict : restriction -> [ T : type ] [ P : T -> type ] ( [ x : T ] P
       x ) -> type
12    | restrict-unused =
13      restrict unused T (\ x : T . P) (\ x : T . G)
14
15    | restrict-linear =
16      restrict linear T (\ x : T . T) (\ x : T . x)
17
18    | restrict-app =
19       restrict Ba T (\ x : T . P x (G x)) (\ x : T . (F x) (G x))
20    <- restrict Bb T (\ x : T . [z : Q x] P x z ) (\ x : T . F x)
21    <- restrict Bc T (\ x : T . Q x ) (\ x : T . G x)
22
23    | restrict-lam =
24       restrict B T (\ x : T . [y : A] P x y) (\ x : T . \ y : A . F y x)
25    <- [ y : A ] restrict B T (\ x : T . P x y) (\ x : T . F y x)
26
27    | restrict-eta =
28       restrict B T (\ x : T . [y : A x] P x y) (\ x : T . \ y : A x. F x y)
29    <- restrict B T (\ x : T . [y : A x] P x y) (\ x : T . F x)
```

Figure 6.13: Linearity in Caledon

Substructural representations with higher order abstract syntax in Elf is due to Crary [18].

```
1  fixity lambda lolli
2  fixity lambda llam
3
4  defn lolli : [ T : type ] ( T -> type ) -> type
5     | llam = [T]{TyF}[F : [x : T] TyF x]
6              restrict linear T TyF F => ( lolli x : T . TyF x)
7
8  defn lapp : {A : type} { T : A -> type } [ f : lolli x : A . T x ] [ a : A ]
       T a -> type
9     | lapp-imp = lapp (llam _ _ F _) V (F V)
10
11 fixity arrow -o
12 defn -o : type -> type -> type
13   as \t : type . \t2 : type . lolli t (\x : t . t2)
```

Figure 6.14: Linear Dependent Product

In the figure 6.13 the predicate "restrict linear" encodes the test that an arbitrary function, even one with a dependent type, is linear [8] if its argument is used exactly once in the term and potentially many times in the type. "lolli" is the linear dependent type constructor and "llam" is the linear function constructor. Provided the code from figure 6.13 was in the environment, figure 6.12 in fact works.

More of these cases could be made less ambiguous through use of an ordered dependent type constructor, but this is significantly more complicated to define, although certainly possible.

```
1  defn sum : nat -o nat -o nat -o type
2     | sum-zero = [N : nat]
3                   [Sum : nat -o nat -o type]
4                   [Sum' : nat -o type]
5                   [Sum'' : type]
6       lapp sum zero Sum -> lapp Sum N Sum' -> lapp Sum' N Sum'' -> Sum''
7     | sum-succ = [N M R : nat]
8                   [Sum1 Sum2 : nat -o nat -o type ]
9                   [ Sum1' Sum2' : nat -o type ]
10                  [ Sum1'' Sum2'' : type ]
11          lapp sum N Sum1 ->
12          lapp Sum1 M Sum1' ->
13          lapp Sum1' R Sum1'' -> Sum1''
14      -> lapp sum (succ N) Sum2 ->
15          lapp Sum1 M Sum2' ->
16          lapp Sum2' (succ R) Sum2'' -> Sum2''
```

Figure 6.15: Use of a linear type

Of course, as seen in figure 6.15, actually using this linear dependent product is a bit absurd. It requires flattening application into a logic programming form where the target is a type variable.

```
1  fixity application lapp
2  defn sum : nat -o nat -o nat -o type
3     | sum-zero = [N : nat] *APP=lapp* sum zero N N
4     | sum-succ = [N M R : nat]
5           *APP=lapp* sum N M R
6       -> *APP=lapp* sum (succ N) M (succ R)
```

Figure 6.16: Example of a syntax for flattening application

That the end result is a type variable means that the family checking algorithm is

no longer applicable. In this case, it is helpful to either hard code linearity or provide a syntax for flattening successive applications using a predicate, as seen in figure 6.16

Syntax for applications could potentially extend the notion of a family such that predicates using these applications could also be frozen. In this case, the applicator x used with $*APP = x*$ would be declared, so that the compiler could check that its type matches the form $\{A : \mathsf{type}\}\{T : A \to \mathsf{type}\}[f : \mathsf{some} - \mathsf{product} x : A.Tx][a : A]Ta \to \mathsf{type}$.

Chapter 7

Conclusion

7.1 Results

For this thesis, I designed a logic programming language with a type system based on the "Calculus of Constructions" which integrated the notion of an implicitly quantified type in a manner useful for automating proof search. I demonstrated a series of reductions from this language to the "Calculus of Constructions" where the output of the language could be interpreted as meaningful theorems. I provided an abstract machine for the language based on higher order unification with proof search, and I demonstrated an elaboration method to this machine.

The semantics of the language based on this compilation and evaluation joined the notions of type inference and traditional evaluation in a way that does not appear to have been examined in great detail in the past.

I provided a method to constrain proof search of a predicate to a small subset of the axioms in the environment using families. I demonstrated a way to explicitly control whether a predicate was searched in a breadth first or depth first manner, allowing constructs similar to fork and join to be defined.

I gave examples of usage of the Caledon language and demonstrated functionality

equivalent to type classes and ways to extend the applicability of this feature using library defined linearity checking.

Finally, I provided an implementation of Caledon in Haskell and provided a standard library both of which can be found at https://github.com/mmirman/caledon/. Since previous dependently typed logic languages did not include polymorphism, standard libraries were not reasonable or possible to include. However, I included polymorphism, so that useful generic lists, type logic, printing, monad and functor libraries became possible.

7.2 Future Work

This thesis presents a new language, and more work can easily be envisioned to provide a greater framework for proving theorems about it. In general, compilation for a language where the programs are theorems for a consistent logic allows significant optimization capability. In Twelf, totality, modes, and worlds allowed predicates to be converted to programs. In general, running Caledon programs in the current implementation is slow, as types need to be recorded and searched during runtime. Algorithms that take advantage of totality checking [2], uniqueness checking [3], worlds checking[3], mode checking[3], and universe checking [32], could be implemented and applied as they were for Twelf and Agda. It would be useful to have a type system for a logic programming language which could ensure closed predicates were theorems.

More work needs to be done to automate type class instancing, as was demonstrated in the section on Linearity. While implemented, universe checking during unification has yet to be proven entirely correct.

The future holds further investigation into additional applications of the language, and it is clear that much more interesting programs can be written with Caledon. While derivatives of one holed types are possible in Caledon, automatically providing traver-

sals for these zipper types is an unexplored topic. While I have demonstrated a concise method of creating concurrency, I look forward to designing libraries for controlling concurrency using the IO primitives.

Bibliography

[1] Andreas Abel. Towards normalization by evaluation for the $\beta\eta$-calculus of constructions (extended version). *Functional and Logic Programming*, pages 224–239, 2010. 3.2.2, 3.3

[2] Thorsten Altenkirch and Nils Anders Danielsson. Termination checking in the presence of nested inductive and coinductive types. In *Note supporting presentation given at the Workshop on Partiality and Recursion in Interactive Theorem Provers, Edinburgh, UK*, 2010. 7.2

[3] Penny Anderson and Frank Pfenning. Verifying uniqueness in a logical framework. *Theorem Proving in Higher Order Logics*, pages 109–129, 2004. 7.2

[4] H. Barendregt, W. Dekkers, and R. Statman. Lambda calculus with types. *Handbook of logic in computer science*, 2:118–310, 1992.

[5] Henk Barendregt. Introduction to generalized type systems. *Journal of functional programming*, 1(2):125–154, 1991. 2.1, 2.1, 2.2

[6] Bruno Barras and Bruno Bernardo. The implicit calculus of constructions as a programming language with dependent types. *Foundations of Software Science and Computational Structures*, pages 365–379, 2008. 3.2.2

[7] Gilles Barthe. Extensions of pure type systems. In *Typed Lambda Calculi and Applications*, pages 16–31. Springer, 1995.

[8] N. Benton, G. Bierman, V. De Paiva, and M. Hyland. A term calculus for intuitionistic linear logic. *Typed Lambda Calculi and Applications*, pages 75–90, 1993. 6.2

[9] Jean-Philippe Bernardy and Marc Lasson. Realizability and parametricity in pure type systems. *Foundations of Software Science and Computational Structures*, pages 108–122, 2011.

[10] EDWIN BRADY. Idris, a general purpose dependently typed programming language: Design and implementation.

[11] Edwin Brady, Christoph Herrmann, and Kevin Hammond. Lightweight invariants with full dependent types. In *Draft Proceedings of Trends in Functional Programming*, volume 2008. Citeseer, 2008.

[12] Paul Callaghan and Zhaohui Luo. An implementation of lf with coercive subtyping & universes. *Journal of Automated Reasoning*, 27(1):3–27, 2001. 2.2.2

[13] Adam Chlipala. Certified programming with dependent types, 2011.

[14] Adam Chlipala, Leaf Petersen, and Robert Harper. Strict bidirectional type checking. In *Proceedings of the 2005 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 71–78. ACM, 2005.

[15] Thierry Coquand and Christine Paulin. Inductively defined types. In *COLOG-88*, pages 50–66. Springer, 1990. 2.2.3

[16] Thierry Coquand, Gerard Huet, et al. The calculus of constructions. 1986. 1, 2.1, 2.2

[17] Thierry Coquand, Randy Pollack, and Makoto Takeyama. A logical framework with dependently typed records. *Typed lambda calculi and applications*, pages 1086–1086, 2003.

[18] Karl Crary. Higher-order representation of substructural logics. *ACM Sigplan Notices*, 45(9):131–142, 2010. 6.2

[19] Rowan Davies and Frank Pfenning. Intersection types and computational effects. In *ACM Sigplan Notices*, volume 35, pages 198–208. ACM, 2000.

[20] G. Dowek, T. Hardin, and C. Kirchner. Higher order unification via explicit substitutions. *Information and Computation*, 157(1):183–235, 2000.

[21] Derek Dreyer, Karl Crary, and Robert Harper. A type system for higher-order modules. *ACM SIGPLAN Notices*, 38(1):236–249, 2003. 2

[22] Joshua Dunfield and Frank Pfenning. *Tridirectional typechecking*, volume 39. ACM, 2004.

[23] C. Elliott. Higher-order unification with dependent function types. In *Rewriting Techniques and Applications*, pages 121–136. Springer, 1989. 3.1

[24] Martın H Escardó. The intrinsic topology of a martin-löf universe.

[25] Herman Geuvers. A short and flexible proof of strong normalization for the calculus of constructions, 1994. 2.2, 2.2.1

[26] Herman Geuvers and Mark-Jan Nederhof. Modular proof of strong normalization for the calculus of constructions. *Journal of Functional Programming*, 1(2):155–189, 1991. 2.1, 2.2

[27] Jan Herman Geuvers. *Logics and type systems*. Citeseer, 1993. 2.2.1, 2.3

[28] Gilles Gowek. The undecidability of typability in the lambda-pi-calculus. In M. Bezem and J.F. Groote, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, pages 139–145, Utrecht, The Netherlands, March 1993. Springer-Verlag LNCS 664. 1

[29] Masami Hagiya and Yozo Toda. On implicit arguments. *Logic, Language and Computation*, pages 10–30, 1994.

[30] C.V. Hall, K. Hammond, S.L. Peyton Jones, and P.L. Wadler. Type classes in haskell. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(2):109–138,

1996.

[31] Robert Harper and Frank Pfenning. A module system for a programming language based on the lf logical framework. *Journal of Logic and Computation*, 8(1):5–31, 1998.

[32] Robert Harper and Robert Pollack. Type checking with universes. *Theoretical computer science*, 89(1):107–136, 1991. 2.2.2, 7.2

[33] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Symposium on logic in Computer Science*, pages 194–204, June 1987. 1

[34] Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, 146:29–60, 1969. 4

[35] Ralf Hinze and Ross Paterson. Finger trees: a simple general-purpose data structure. *Journal of Functional Programming*, 16(2):197–218, 2006. 5.1

[36] James G Hook and Douglas J Howe. Impredicative strong existential equivalent to type: type. Technical report, Cornell University, 1986.

[37] Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975. 3.1

[38] Gérard Huet. Functional pearl. *J. functional programming*, 7(5):549–554, 1997. 3.4.4

[39] Antonius Hurkens. A simplification of girard's paradox. *Typed Lambda Calculi and Applications*, pages 266–278, 1995.

[40] LSV Jutting. Typing in pure type systems. *Information and Computation*, 105(1): 30–41, 1993. 2.1

[41] Chantal Keller and Thorsten Altenkirch. Normalization by hereditary substitutions. *proceedings of Mathematical Structured Functional Programming*, 2010. 3.3, 3.3.2

[42] Andres Löh, Conor McBride, and Wouter Swierstra. A tutorial implementation of a dependently typed lambda calculus. *Fundamenta Informaticae*, 102(2):177–207,

2010.

[43] Zhaohui Luo. Ecc, an extended calculus of constructions. In *Logic in Computer Science, 1989. LICS'89, Proceedings., Fourth Annual Symposium on*, pages 386–395. IEEE, 1989. 2.2.2

[44] Marko Luther. More on implicit syntax. *Automated Reasoning*, pages 386–400, 2001. 2

[45] James McKinna and Robert Pollack. Pure type systems formalized. *Typed Lambda Calculi and Applications*, pages 289–305, 1993. 2.1

[46] Paul-André Melliès and Benjamin Werner. A generic normalisation proof for pure type systems. In *Types for Proofs and Programs*, pages 254–276. Springer, 1998.

[47] Albert R Meyer and Mark B Reinhold. Type is not a type. In *Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 287–295. ACM, 1986.

[48] Dale Miller. *A theory of modules for logic programming*. University of Pennsylvania, Department of Computer and Information Science, 1986.

[49] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of logic and computation*, 1(4):497–536, 1991. 3.2.1

[50] Dale Miller. Unification under a mixed prefix. *Journal of symbolic computation*, 14 (4):321–358, 1992.

[51] Dale Miller and Gopalan Nadathur. Higher-order logic programming. In *Third International Conference on Logic Programming*, pages 448–462. Springer, 1986. 3.1

[52] Dale Miller and Gopalan Nadathur. An overview of λprolog. In *Proc. of the 5th Int. Conf. on Logic Programming*, 1988. 1

[53] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform

proofs as a foundation for logic programming. *Annals of Pure and Applied logic*, 51(1):125–157, 1991.

[54] Alexandre Miquel. *Le Calcul des Constructions implicite: syntaxe et sémantique*. PhD thesis, PhD thesis, Université Paris 7, 2001.

[55] Alexandre Miquel. The implicit calculus of constructions extending pure type systems with an intersection type binder and subtyping. *Typed Lambda Calculi and Applications*, pages 344–359, 2001. 2, 2.2.1, 4.1

[56] Matthew Mirman. Logic programming and type inference with the calculus of constructions. 2013.

[57] Ulf Norell. *Towards a practical programming language based on dependent type theory*. Chalmers University of Technology, 2007.

[58] F. Pfenning. Partial polymorphic type inference and higher-order unification. In *Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 153–163. ACM, 1988.

[59] F. Pfenning and C. Elliot. Higher-order abstract syntax. *ACM SIGPLAN Notices*, 23 (7):199–208, 1988. 3.1

[60] F. Pfenning et al. Logic programming in the lf logical framework. *Logical frameworks*, pages 149–181, 1991. 1, 1.1, 3, 3.1, 3.3, 3.4.3, 3.5

[61] Frank Pfenning. Unification and anti-unification in the calculus of constructions. In *Logic in Computer Science, 1991. LICS'91., Proceedings of Sixth Annual IEEE Symposium on*, pages 74–85. IEEE, 1991. 3.1, 3.2.2

[62] Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical structures in computer science*, 11(04):511–540, 2001.

[63] Frank Pfenning and Carsten Schürmann. System description: Twelfa meta-logical framework for deductive systems. In *Automated DeductionCADE-16*, pages 202–

206. Springer, 1999.

[64] Frank Pfenning and Robert J Simmons. Substructural operational semantics as ordered logic programming. In *Logic In Computer Science, 2009. LICS'09. 24th Annual IEEE Symposium on*, pages 101–110. IEEE, 2009. 5.1

[65] Brigitte Pientka and Frank Pfenning. Optimizing higher-order pattern unification. In Franz Baader, editor, *Automated Deduction CADE-19*, volume 2741 of *Lecture Notes in Computer Science*, pages 473–487. Springer Berlin Heidelberg, 2003. ISBN 978-3-540-40559-7. doi: 10.1007/978-3-540-45085-6_40. URL `http://dx.doi.org/10.1007/978-3-540-45085-6_40`.

[66] Benjamin C Pierce and David N Turner. Local type inference. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(1):1–44, 2000.

[67] Robert Pollack. Implicit syntax. In *Informal Proceedings of First Workshop on Logical Frameworks, Antibes*, 1990. 4

[68] Florian Rabe and Carsten Schürmann. A practical module system for lf. In *Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*, pages 40–48. ACM, 2009.

[69] Jason Reed. Redundancy elimination for lf. *Electronic Notes in Theoretical Computer Science*, 199:89–106, 2008.

[70] J-W Roorda and JT Jeuring. Pure type systems for functional programming. 2001. 2.1, 2.2

[71] P. Severi. Type inference for pure type systems. *Information and Computation*, 143 (1):1–23, 1998.

[72] Lionel Vaux. A note on an implicit calculus with annotated terms: introducing universal dependent types., 2004.

[73] M. Wenzel. Type classes and overloading in higher-order logic. *Theorem Proving in*

Higher Order Logics, pages 307–322, 1997.