Encoding the Calculus of Constructions in a Higher-Order Logic[*]

Amy Felty
AT&T Bell Laboratories
600 Mountain Ave.
Murray Hill, NJ 07974 USA

Abstract

We present an encoding of the calculus of constructions (CC) *in a higher-order intuitionistic logic* ($\mathcal{I}$) *in a direct way, so that correct typing in* CC *corresponds to intuitionistic provability in a sequent calculus for* $\mathcal{I}$*. In addition, we demonstrate a direct correspondence between proofs in these two systems. The logic* $\mathcal{I}$ *is an extension of hereditary Harrop formulas (hh) which serve as the logical foundation of the logic programming language* λ*Prolog. Like hh,* $\mathcal{I}$ *has the uniform proof property, which allows a complete non-deterministic search procedure to be described in a straightforward manner. Via the encoding, this search procedure provides a goal directed description of proof checking and proof search in* CC.

1 Introduction

The motivations for encoding the Calculus of Constructions [4] in a higher-order logic are twofold. First, it is well-known that in CC, types can be viewed as formulas and terms as proofs in an intuitionistic logic (in a manner similar to that described in Howard [11]). We want to provide some insight into the correspondence between these two languages by providing a direct encoding of one into the other. Second, we are interested in proof search in CC. The encoding provides a high-level description of a search procedure based on logic programming.

Intuitively, the correspondence between CC types and formulas in higher-order logic is fairly direct. A functional type $P \to Q$ corresponds to an implication in higher-order logic. Taking this idea further, introductions and eliminations of the type arrow in type derivations correspond directly to the introduction and elimination rules for implication in higher-order logic. More generally, a type in CC has the structure $(x:P)Q$ where P and Q are types and x is a variable of type P bound in this expression. (The arrow form is an abbreviation for the case when the variable x does not appear in Q.) This more general form can be viewed as a formula where x is universally quantified in Q. However, the analogy that applied to introduction and elimination of implication is not as direct for universal quantification. In the elimination rule, for example, from the assumption $\forall x Q$ we can conclude that any instance $[R/x]Q$ holds as long as R has the same type as x. In higher-order logic, this type is a simple type. In the corresponding CC rule, the variable x and term R can have arbitrary CC type. Although CC types include the types of the simply-typed λ-calculus, they also include much more. As a result of this mismatch, although the main ideas are rather simple, carrying out the full formalization of the correspondence between these two languages is more difficult than one might expect. The encoding presented here is an extension of an encoding in Felty [8] of the Logical Framework (LF) in a slightly less expressive logic than the one used here. Although LF types are more expressive than simple types, they are less expressive than CC types. Correctly handling the polymorphism of CC requires a significant extension over the LF encoding.

In our encoding, types will correspond to predicates over terms. Informally, the CC type $(x:P)Q$ represents a functional type in the following sense: if f is a function of this type, and R is a term of type P, then fR (f applied to R) has the type Q where all occurrences of x are replaced by R. Such term/type pairs will be mapped to universally quantified implications, *e.g.*, $f:((x:P)Q)$ is mapped to $\forall x\ (\llbracket x:P \rrbracket \supset \llbracket fx:Q \rrbracket)$ (where the double brackets denote the encoding operation). In CC, types can appear inside terms. If we prefix the above formula with a λ-abstraction over f, we obtain a representation of a type which, as we will see, can then occur inside encoded terms.

The behavior of the search procedure suggested by

[*]In *Proceedings of the 8th Annual IEEE Symposium on Logic in Computer Science, June 1993.* ©1993 by the Institute of Electrical and Electronics Engineers, Inc. All rights reserved.

the encoding is similar to the complete procedure for CC presented in Dowek [5]. One way in which our procedure differs is that in order to implement an automated version, Dowek's procedure would require unification on CC terms to solve constraints that arise during search. In our procedure, terms are encoded as simply-typed λ-terms, and thus unification on such terms is all that is required.

In addition to providing a search procedure, the encoding presented here provides a framework in which to study how theorem proving techniques designed for one system can be applied to proof search in the other. For example, some early work by Bledsoe [2] develops techniques for automatic discovery of substitutions for set variables, a class of higher-order variables. Studying these techniques via the encoding may provide insight into further automating search in CC and into Bledsoe's technique itself. In the other direction, additional insight into automating theorem proving in higher-order logic might be gained by studying the behavior of search procedures for CC such as Dowek's [5] via the encoding.

In the next two sections we present the two languages we are concerned with. In Section 2 we present the meta-logic $\mathcal{I}$, and in Section 3 we present the Calculus of Constructions. Then in Section 4 we present the encoding of CC into $\mathcal{I}$. We discuss its correctness in Section 5. In Section 6 we describe an implementation of a search procedure based on the encoding, and finally in Section 7 we conclude and discuss future work.

2 A Higher-Order Meta-Logic

The types and terms of $\mathcal{I}$ are essentially those of the simple theory of types [3]. We assume a fixed set of *primitive types*, which includes at least the symbol o, the type for propositions. *Function types* are constructed using the binary infix symbol $\rightarrow$; if τ and σ are types, then so is $\tau \rightarrow \sigma$. The type constructor $\rightarrow$ associates to the right. The *order* of a primitive type is 0 while the order of a function type $\tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau_0$, where $n \geq 0$ and τ_0 is primitive, is one greater than the maximum order of $\tau_1, \ldots, \tau_n$.

For each type τ, we assume that there are denumerably many constants and variables of that type. Simply typed λ-terms are built in the usual way using constants, variables, applications, and abstractions. Equality between λ-terms is taken to mean $\beta\eta$-convertibility. We shall assume that the reader is familiar with the usual notions and properties of substitution and α, β, and η conversion for the simply typed λ-calculus. See Hindley and Seldin [10] for a fuller discussion of these basic properties. If x is a variable and t is a term then $[t/x]$ denotes the operation of substituting t for all free occurrences of x, systematically changing bound variables in order to avoid variable capture.

A constant or variable p of type $\tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow o$ is called a *predicate* constant or variable. An *atomic formula* is a term of type o of the form $pt_1 \ldots t_n$ where p is a predicate constant or variable. The predicate p is the *head* of this atomic formula. The logical connectives are defined by introducing suitable constants as in Church [3]. The constants $\wedge$ (conjunction) and $\supset$ (implication) are both of type $o \rightarrow o \rightarrow o$, and $\forall_\tau$ (universal quantification) is of type $(\tau \rightarrow o) \rightarrow o$, for each type τ. The expression $\forall_\tau(\lambda z\ t)$ is written simply as $\forall_\tau z\ t$ or $\forall z\ t$ when the type τ can be inferred from context.

Intuitionistic provability for $\mathcal{I}$ can be given in terms of sequent calculus proofs. A *sequent* is a pair $\mathcal{P} \longrightarrow B$, where $\mathcal{P}$ is a finite (possibly empty) set of formulas and B is a formula. The set $\mathcal{P}$ is this sequent's *antecedent* and B is its *succedent*. The expression $B, \mathcal{P}$ denotes the set $\mathcal{P} \cup \{B\}$. This notation is used even if $B \in \mathcal{P}$. The inference rules for sequents are presented in Figure 1. The following provisos are also attached to the two inference rules for quantifier introduction: in $\forall$-R c is a constant of type τ that does not occur free in the lower sequent, and in $\forall$-L t is a term of type τ.

A *proof* of the sequent $\mathcal{P} \longrightarrow B$ is a finite tree constructed using these inference rules such that the root is labeled with $\mathcal{P} \longrightarrow B$ and the leaves are labeled with *initial sequents*, that is, sequents $\mathcal{P}' \longrightarrow B'$ such that $B' \in \mathcal{P}'$. The non-terminals in such a tree are instances of the inference figures in Figure 1. Since we do not have an inference figure for $\beta\eta$-conversion, we shall assume that in building a proof, two formulas are equal if they are $\beta\eta$-convertible. In a sequent $\mathcal{P} \longrightarrow B$, we say that the formulas in $\mathcal{P}$ are *assumptions* and B is a *goal*. If this sequent has a proof, we write $\mathcal{P} \vdash_I B$ and say that goal B is provable from assumptions $\mathcal{P}$.

Definition 1 Let $\mathcal{P}$ be a finite set of $\mathcal{I}$ formulas. The expression $|\mathcal{P}|$ denotes the smallest set of pairs $\langle \mathcal{G}, D \rangle$ of finite sets of formulas $\mathcal{G}$ and formula D, such that

- If $D \in \mathcal{P}$ then $\langle \emptyset, D \rangle \in |\mathcal{P}|$.
- If $\langle \mathcal{G}, D_1 \wedge D_2 \rangle \in |\mathcal{P}|$ then $\langle \mathcal{G}, D_1 \rangle \in |\mathcal{P}|$ and $\langle \mathcal{G}, D_2 \rangle \in |\mathcal{P}|$.
- If $\langle \mathcal{G}, \forall_\tau x D \rangle \in |\mathcal{P}|$ then $\langle \mathcal{G}, [t/x]D \rangle \in |\mathcal{P}|$ for all terms t of type τ.

$$\frac{B, C, \mathcal{P} \longrightarrow A}{B \wedge C, \mathcal{P} \longrightarrow A}\wedge\text{-L} \qquad \frac{\mathcal{P} \longrightarrow B \qquad C, \mathcal{P} \longrightarrow A}{B \supset C, \mathcal{P} \longrightarrow A}\supset\text{-L} \qquad \frac{[t/x]B, \mathcal{P} \longrightarrow A}{\forall_\tau x\, B, \mathcal{P} \longrightarrow A}\forall\text{-L}$$

$$\frac{\mathcal{P} \longrightarrow B \qquad \mathcal{P} \longrightarrow C}{\mathcal{P} \longrightarrow B \wedge C}\wedge\text{-R} \qquad \frac{B, \mathcal{P} \longrightarrow C}{\mathcal{P} \longrightarrow B \supset C}\supset\text{-R} \qquad \frac{\mathcal{P} \longrightarrow [c/x]B}{\mathcal{P} \longrightarrow \forall_\tau x\, B}\forall\text{-R}$$

Figure 1: Left and Right Introduction Rules for $\mathcal{I}$

- If $\langle \mathcal{G}, G \supset D\rangle \in |\mathcal{P}|$ then $\langle \mathcal{G} \cup \{G\}, D\rangle \in |\mathcal{P}|$.

This inference system has the *uniform proof* property as defined in Miller et al. [12] and described by the following theorem.

Theorem 2 Let $\mathcal{P}$ be a finite set of formulas and let B be a formula. The sequent $\mathcal{P} \longrightarrow B$ has a proof if and only if it has a proof in which every sequent containing a non-atomic formula as its succedent is the conclusion of a right introduction rule.

Proof: The proof is by induction on the height of an arbitrary proof. The cases for the left rules use the following three lemmas proved by induction on the structure of the formula in the succedent. (1) If there is a uniform proof of $B, C, \mathcal{P} \longrightarrow A$, then there is a uniform proof of $B \wedge C, \mathcal{P} \longrightarrow A$. (2) If there are uniform proofs of $\mathcal{P} \longrightarrow B$ and $C, \mathcal{P} \longrightarrow A$, then there is a uniform proof of $B \supset C, \mathcal{P} \longrightarrow A$. (3) If there is a uniform proof of $[t/x]B, \mathcal{P} \longrightarrow A$, then there is a uniform proof of $\forall_\tau x\, B, \mathcal{P} \longrightarrow A$.

Based on this property, we can describe the following high-level non-deterministic search procedure for proofs in this logic. This procedure is described by the following operations. Here G is a goal which we are trying to prove from the set of assumptions $\mathcal{P}$.

AND: If G is $G_1 \wedge G_2$ then try to show that both G_1 and G_2 follow from $\mathcal{P}$.

AUGMENT: If G is $D \supset G'$ then try to show that G' follows from $\mathcal{P} \cup \{D\}$.

GENERIC: If G is $\forall_\tau x\, G'$ then pick a new constant c of type τ and try to show $[c/x]G'$.

BACKCHAIN: If G is atomic and there is a pair $\langle \mathcal{G}, G\rangle \in |\mathcal{P}|$, then attempt to prove each of the formulas in $\mathcal{G}$ from $\mathcal{P}$. If $\mathcal{G}$ is empty, then we are done.

We say that a subformula occurs *positively* (*negatively*) in a formula if it occurs on the left of an even (odd) number of implications. In the logic of *hh* [12], which serves as the logical foundation of the logic programming language λProlog, all atomic formulas occurring positively in assumptions or negatively in goals cannot have a variable at the head. The further restriction that atomic formulas are not allowed to contain occurrences of $\supset$ is imposed. Allowing variables at the head as well as occurrences of $\supset$ in atomic formulas as we do provides a significant extension to the logic. However, in *hh*, goal formulas can also be of the form $G_1 \vee G_2$ and $\exists_\tau x\, G$. The properties stated here do not extend if we permit these connectives in goal formulas.

In Miller et al. [12], it was shown that unification can be used to implement the BACKCHAIN operation for *hh* and is sufficient for determining substitutions. However, because we allow variables at the head of arbitrary atomic subformulas in $\mathcal{I}$, unification is no longer enough. Consider the following subset of $|\mathcal{P}|$ obtained by modifying the third clause of Definition 1.

Definition 3 Let $\mathcal{P}$ be a finite set of closed $\mathcal{I}$ formulas. Here, we assume the variables bound by universal quantifiers in each formula are distinct. The expression $|\mathcal{P}|_0$ denotes the smallest set of pairs $\langle \mathcal{G}, D\rangle$ of finite sets of formulas $\mathcal{G}$ and formula D, such that

- If $D \in \mathcal{P}$ then $\langle \emptyset, D\rangle \in |\mathcal{P}|_0$.
- If $\langle \mathcal{G}, D_1 \wedge D_2\rangle \in |\mathcal{P}|_0$ then $\langle \mathcal{G}, D_1\rangle \in |\mathcal{P}|_0$ and $\langle \mathcal{G}, D_2\rangle \in |\mathcal{P}|_0$.
- If $\langle \mathcal{G}, \forall_\tau x D\rangle \in |\mathcal{P}|_0$ then $\langle \mathcal{G}, D\rangle \in |\mathcal{P}|_0$.
- If $\langle \mathcal{G}, G \supset D\rangle \in |\mathcal{P}|_0$ then $\langle \mathcal{G} \cup \{G\}, D\rangle \in |\mathcal{P}|_0$.

Informally, an implementation of BACKCHAIN for *hh* can be described as follows. Choose a pair $\langle \mathcal{G}, A\rangle$ in $|\mathcal{P}|_0$ such that A is atomic. Replace all free variables with logic variables in A and attempt to unify A with the current atomic goal G. If unification succeeds, apply the resulting substitution to the formulas in $\mathcal{G}$ and attempt to prove each of them. Since A in the chosen pair is atomic, it will have the form $pt_1 \ldots t_n$.

In hh, p must be a constant and thus any substitution of variables will result in an atomic formula. In $\mathcal{I}$, however, if p is a variable, and A does not unify with the goal, we still have to consider substitutions that transform A into a non-atomic formula G, and then consider the set $|\{G\}|_0$. For example, consider a substitution for p that transforms A to an implication $B \supset A'$ where A' is atomic. We must now consider pairs in $|\{B \supset A'\}|_0$ where the second element is atomic, in this case just $\langle\{B\}, A'\rangle$, and try to unify A with A'. If successful B becomes an additional subgoal that must be proven along with the formulas in $\mathcal{G}$. Otherwise, we have to continue repeating the procedure. Thus, in $\mathcal{I}$, we have additional non-determinism caused by having to "guess" at least the part of the substitution that determines the logical structure of the formula we are backchaining on before unification can be used. In Section 6, we discuss a modification which eliminates this non-determinism, and results in an incomplete but still powerful search procedure. We discuss how this incompleteness affects the completeness of proof search for encoded CC.

3 The Calculus of Constructions

The syntax of terms of the Calculus of Constructions (CC) is given by the following grammar.

$$\text{Type} \mid \text{Prop} \mid x \mid PQ \mid [x:P]Q \mid (x:P)Q$$

Here x is a syntactic variable ranging over variables, and P and Q are syntactic variables ranging over terms. We assume a denumerable set of CC variables. The variable x is bound in the expressions $[x:P]Q$ and $(x:P)Q$. The former binding operator corresponds to the usual notion of λ-abstraction, while the latter corresponds to abstraction in dependent types. We write $P \to Q$ for $(x:P)Q$ when x does not occur in Q.

Terms that differ only in the names of bound variables are identified. If x is an object-level variable and N is an object then $[N/x]$ denotes the operation of substituting N for all free occurrences of x, systematically changing bound variables in order to avoid variable capture.

The following four kinds of *assertions* are derivable in the CC type theory.

$\vdash \Gamma$ context	(Γ is valid context)
$\Gamma \vdash K : \text{Type}$	(K is a type in Γ)
$\Gamma \vdash A : K$	(A has type K in Γ)
$\Gamma \vdash M : A$	(M has proposition A in Γ)

For the latter three assertions, we say that K, A, or M, respectively, is a *well-typed term* in Γ. These assertions separate terms into three levels. Here K and L range over terms at the first level called *types*, which appear in expressions on the left in the second kind of judgment. A and B range over second-level expressions called *families*, which appear on the left in the third kind of assertion. Finally, M and N range over third-level expressions called *objects*, which appear on the left in the fourth kind of assertion. The constant Type forms a 0-level class with a single element. We say that Type is a *kind*. Only a subclass of families can appear on the right in derivable assertions of the fourth kind. We call terms in this subclass *propositions*. Propositions also correspond to the subclass of families that occur on the left in derivable assertions of the third kind in the special case when K is Prop. In addition, Γ ranges over contexts. The empty context is denoted by $\langle\rangle$. We will use P, Q, and R to range over arbitrary objects, families, and types. Although the inference rules will not be presented in a way that distinguishes the three levels of terms, we make them explicit here so that we may later discuss their corresponding notions in the encoding into $\mathcal{I}$. We write $\Gamma \vdash \alpha$ for an arbitrary assertion of one of the later three forms above, where α is called a CC *judgment*. In deriving an assertion of this form, we always assume that we start with a valid context Γ.

CC has the property that all well-typed terms are strongly normalizing. The notion of $\beta\eta$-conversion is defined by the rules in Figure 2. The (η) rule has the proviso that the variable x cannot appear free in P. We write $P =_\beta Q$ if $P =_{\beta\eta} Q$ has a derivation that doesn't use the (η) rule. We say that a term is *primitive* if it is Type, Prop or a proposition that is $\beta\eta$-convertible to a term of the form $xP_1 \ldots P_n$ where $n \geq 0$ and x is a variable. All well-typed terms have unique β-normal and $\beta\eta$-normal forms. In addition, they have unique $\beta\eta$-long forms [5]. In particular, the $\beta\eta$-long form of a term is the η-long form of its $\beta\eta$-normal form. A term in $\beta\eta$-long form has the form $[x_1 : P_1] \cdots [x_n : P_n](y_1 : Q_1) \cdots (y_m : Q_m)(zR_1 \ldots R_p)$ where $n, m, p \geq 0$, z is a variable, Prop, or Type, $(zR_1 \ldots R_p)$ has primitive proposition, type, or kind, and $P_1, \ldots, P_n, Q_1, \ldots, Q_m, R_1, \ldots, R_p$ are in $\beta\eta$-long form. We write $\beta(P)$ and $\beta\eta l(P)$ to denote the β-normal and $\beta\eta$-long forms, respectively, of an arbitrary term P.

We present a version of the typing rules of CC such that the terms in any derivable assertion are in β-normal form. These rules are given in Figure 3. In these rules, s, s_1, and s_2, are either Type or Prop.

$$([x:R]P)Q =_{\beta\eta} [Q/x]P \quad (\beta)$$

$$[x:R]Px =_{\beta\eta} P \quad (\eta)$$

$$\frac{P_1 =_{\beta\eta} P_2 \qquad Q_1 =_{\beta\eta} Q_2}{[x:P_1]Q_1 =_{\beta\eta} [x:P_2]Q_2}\ (\xi\text{-ABS})$$

$$\frac{P_1 =_{\beta\eta} P_2 \qquad Q_1 =_{\beta\eta} Q_2}{(x:P_1)Q_1 =_{\beta\eta} (x:P_2)Q_2}\ (\xi\text{-PROD})$$

$$\frac{P_1 =_{\beta\eta} P_2 \qquad Q_1 =_{\beta\eta} Q_2}{P_1Q_1 =_{\beta\eta} P_2Q_2}\ (\text{CONG})$$

$$P =_{\beta\eta} P \quad (\text{REFL})$$

$$\frac{P =_{\beta\eta} Q}{Q =_{\beta\eta} P}\ (\text{SYM})$$

$$\frac{P =_{\beta\eta} R \qquad R =_{\beta\eta} Q}{P =_{\beta\eta} Q}\ (\text{TRANS})$$

Figure 2: $\beta\eta$-Convertibility in CC

$$\vdash \langle\rangle \text{ context} \quad (\text{EMPTY-CTX})$$

$$\frac{\vdash \Gamma \text{ context} \qquad \Gamma \vdash P : s}{\vdash \Gamma, x:P \text{ context}}\ (\text{INTRO})$$

$$\frac{\vdash \Gamma \text{ context} \qquad \Gamma \vdash P : Q}{\vdash \Gamma, P:Q \text{ context}}\ (\text{LEMMA})$$

$$\Gamma \vdash \text{Prop} : \text{Type} \quad (\text{PROP-TYPE})$$

$$\frac{P:Q \in \Gamma}{\Gamma \vdash P : Q}\ (\text{INIT})$$

$$\frac{\Gamma \vdash P : s_1 \qquad \Gamma, x:P \vdash Q : s_2}{\Gamma \vdash (x:P)Q : s_2}\ (\text{PROD})$$

$$\frac{\Gamma \vdash R : s_1 \quad \Gamma, x:R \vdash Q : s_2 \quad \Gamma, x:R \vdash P : Q}{\Gamma \vdash [x:R]P : (x:R)Q}\ (\text{ABS})$$

$$\frac{\Gamma \vdash P_1 : (x:Q_1)Q_2 \qquad \Gamma \vdash P_2 : Q_1}{\Gamma \vdash \beta(P_1P_2) : \beta([P_2/x]Q_2)}\ (\text{APP})$$

Figure 3: CC Typing Rules

In (INTRO), (PROD), and (ABS), we assume that the variable x does not already occur as the left hand side of a context item in Γ. Items introduced into contexts by (LEMMA) will be called *context lemmas.* A *derivation* in CC of the assertion $\Gamma \vdash \alpha$ is a finite tree constructed using these inference rules with root $\Gamma \vdash \alpha$. It can be shown that for a given derivation of an arbitrary assertion $\Gamma \vdash P : Q$ using the usual presentation of CC, *e.g.*, Coquand and Huet [4], there is a corresponding derivation of $\beta(\Gamma) \vdash \beta(P) : \beta(Q)$ and also of $\beta\eta l(\Gamma) \vdash \beta\eta l(P) : \beta\eta l(Q)$ with the same basic structure using the rules in Figure 3. Here, $\beta(\Gamma)$ represents the context with every right hand side of a pair introduced by (INTRO) replaced by its β-normal form, and both the left and right of every context lemma replaced by their β-normal forms. $\beta\eta l(\Gamma)$ is defined similarly for $\beta\eta$-long forms.

A *canonical derivation* in CC is a derivation such that (1) every occurrence of (INIT) is followed by a series of applications of (APP) such that the term on the right in the conclusion of the last one is a primitive proposition or type, and (2) every left premise of (APP) is either the conclusion of (INIT) or (APP). It can be shown that any derivable assertion has a canonical derivation. In addition, all terms in judgments in such derivations are not only in β-normal form, but also in $\beta\eta$-long form. However, given an arbitrary derivable assertion, there is not necessarily a canonical assertion with the same basic structure.[1] Because of the restricted form of canonical derivations, we "lose" some proofs. We can, in a sense, gain them back using a technique similar to that used for LF in Felty [8]. Given an arbitrary derivation of $\Gamma \vdash \alpha$, we can define a function which "reads off" a series of context lemmas Δ. It is then possible to obtain a canonical derivation of the same basic structure of $\Gamma, \Delta \vdash \alpha$.

In this paper, we will only consider canonical derivations. By restricting CC in this way, we obtain an inference system whose proofs correspond directly to those we can build using our search procedure. Each uniform proof in $\mathcal{I}$ built from encoded assumptions has a corresponding canonical derivation in CC of similar structure, and vice versa.

[1] We do not give a precise definition of "same basic structure" here, though it is possible to do so. An example of a proof transformation that does not preserve "same basic structure" is cut-elimination.

4 Encoding the Calculus of Constructions

Given an assertion $\Gamma \vdash \alpha$, the encoding we present here will map Γ to a set of assumptions and map α to a goal to be proven from these assumptions. Such an encoding operates on judgment pairs. We will also need to define a translation of CC terms to simply-typed λ-terms. These two encodings are defined mutually recursively since encoding a proposition or type of the form $(x:P)Q$ will require introducing a new variable, say f, encoding the judgment $f : ((x:P)Q)$, and abstracting over f. The resulting formula is a statement describing properties of an arbitrary term of this type.

Here, we assume the variables of CC are divided into six denumerable sets, two for each of the three levels of terms: objects, families, and types. For objects, these sets will be denoted $\mathcal{V}_o^1$ and $\mathcal{V}_o^2$. We assume all free object variables in the assertion to be encoded are in $\mathcal{V}_o^1$. The translation will choose variables from $\mathcal{V}_o^2$ when it needs "new" object variables. Similarly, we have sets $\mathcal{V}_f^1$, $\mathcal{V}_f^2$, $\mathcal{V}_t^1$, and $\mathcal{V}_t^2$.

In this section, since we encode CC in $\mathcal{I}$, we consider $\mathcal{I}$ as the metalanguage. We introduce meta-level type ob which will be the type of encoded CC objects. We assume a fixed bijective mapping ρ_o from variables of $\mathcal{V}_o^1$ and $\mathcal{V}_o^2$ to meta-variables of type ob. We will also assume a fixed bijective mapping ρ_f from variables of $\mathcal{V}_f^1$ and $\mathcal{V}_f^2$ to meta-variables of type $ob \to o$. CC families will be mapped to predicates over objects. Finally, we assume a mapping ρ_t from type variables to meta-variables of type $(ob \to o) \to o$. CC types will correspond to "predicates over predicates over" objects. The union of these three mappings will be denoted ρ. For readability in our presentation, these mappings will often be implicit. A variable x will represent both a CC variable and its corresponding meta-variable given by ρ. It will always be clear from context which is meant.

There are four kinds of applications in CC, an object applied to an object, an object to a family, a family to an object, and a family to a family. Similarly there are four kinds of abstraction, where the abstracted variable is either an object or family and the resulting term is either an object or family. To encode such terms, we introduce a constant for each kind of application and abstraction. We also introduce constants for the CC constants Prop and Type. These constants and their types are given in Figure 4. Note that the types of the constants in the figure are all of order three or less. The order of the types here corresponds directly to the number of levels of terms, in this case three with a level 0 containing only the constant Type. It should be possible to extend the encoding presented here to generalized type systems (GTS) as in Barendregt [1]. Such systems may have any number of levels. The number of levels corresponds directly to the maximum order of the types of the constants introduced to encode terms.

We are now ready to define the encoding of CC terms. We denote the encoding of term P as $\langle\!\langle P \rangle\!\rangle$. We denote the encoding of judgment $P:Q$ as $[\![P:Q]\!]$. The encoding on terms is defined in Figure 5. We use syntactic variables to denote the class to which each CC term belongs. Here, f is an object variable from $\mathcal{V}_o^2$ and g is a family variable from $\mathcal{V}_f^2$. We assume that the variables chosen during the translation of a single term are all distinct. It is easy to see that objects are mapped to terms of type ob, families to terms of type $ob \to o$, and types to terms of type $(ob \to o) \to o$. This encoding also has the following property.

Lemma 4 Given CC terms P and Q, and variable x, $[\langle\!\langle Q \rangle\!\rangle/x]\langle\!\langle P \rangle\!\rangle = \langle\!\langle [Q/x]P \rangle\!\rangle$.

The translations of CC context items and judgments to $\mathcal{I}$ formulas is defined in Figure 6. It is a partial function since it is defined by cases and undefined when no case applies. It will in fact always be defined on valid context items and judgments. Note the direct mapping of ()-abstraction in CC to instances of universal quantification and implication in $\mathcal{I}$ formulas, as discussed earlier. In the first two clauses of the definition, the bound variable is mapped to a variable at the meta-level bound by universal quantification. In the third conjunct, the left hand side of the implication asserts the fact that the bound variable has a certain proposition or type, while the right hand side contains the translation of the body of the proposition or type which may contain occurrences of this bound variable. The base case occurs when there is no leading ()-abstraction on the left, resulting in an atomic formula. The following property follows from the definitions of the encodings on terms and judgments.

Lemma 5 Let P and Q be CC terms such that $[\![P : Q]\!]$ is well-defined and doesn't use the first clause of the definition in Figure 6. Then $[\![P : Q]\!] = (\langle\!\langle Q \rangle\!\rangle\, \langle\!\langle P \rangle\!\rangle)$.

The convertibility relation for encoded CC terms can be expressed as a set of $\mathcal{I}$ formulas. We introduce the following three binary predicates which will be used to express convertibility at the levels of ob-

$$\begin{aligned} prop &: (ob \to o) \to o & typ &: ((ob \to o) \to o) \to o \\ ap_{oo} &: ob \to ob \to ob & abs_{oo} &: (ob \to ob) \to (ob \to o) \to ob \\ ap_{of} &: ob \to (ob \to o) \to ob & abs_{fo} &: ((ob \to o) \to ob) \to ((ob \to o) \to o) \to ob \\ ap_{fo} &: (ob \to o) \to ob \to ob \to o & abs_{of} &: (ob \to (ob \to o)) \to (ob \to o) \to ob \to o \\ ap_{ff} &: (ob \to o) \to (ob \to o) \to ob \to o & abs_{ff} &: ((ob \to o) \to (ob \to o)) \to ((ob \to o) \to o) \to ob \to o \end{aligned}$$

Figure 4: Constants for Encoding CC Terms

$$\begin{aligned} & & & & \langle\!\langle [x\!:\!A]M \rangle\!\rangle &:= (abs_{oo}\ \lambda x.\langle\!\langle M \rangle\!\rangle\ \langle\!\langle A \rangle\!\rangle) \\ \langle\!\langle x \rangle\!\rangle &:= \rho(x) & & & \langle\!\langle [x\!:\!K]M \rangle\!\rangle &:= (abs_{fo}\ \lambda x.\langle\!\langle M \rangle\!\rangle\ \langle\!\langle K \rangle\!\rangle) \\ \langle\!\langle \mathrm{Type} \rangle\!\rangle &:= typ & & & \langle\!\langle [x\!:\!A]B \rangle\!\rangle &:= (abs_{of}\ \lambda x.\langle\!\langle B \rangle\!\rangle\ \langle\!\langle A \rangle\!\rangle) \\ \langle\!\langle \mathrm{Prop} \rangle\!\rangle &:= prop & & & \langle\!\langle [x\!:\!K]B \rangle\!\rangle &:= (abs_{ff}\ \lambda x.\langle\!\langle B \rangle\!\rangle\ \langle\!\langle K \rangle\!\rangle) \\ \langle\!\langle MN \rangle\!\rangle &:= (ap_{oo}\ \langle\!\langle M \rangle\!\rangle\ \langle\!\langle N \rangle\!\rangle) & & & \langle\!\langle (x\!:\!A)B \rangle\!\rangle &:= \lambda f.[\![f : (x\!:\!A)B]\!] \\ \langle\!\langle MB \rangle\!\rangle &:= (ap_{of}\ \langle\!\langle M \rangle\!\rangle\ \langle\!\langle B \rangle\!\rangle) & & & \langle\!\langle (x\!:\!K)B \rangle\!\rangle &:= \lambda f.[\![f : (x\!:\!K)B]\!] \\ \langle\!\langle AN \rangle\!\rangle &:= (ap_{fo}\ \langle\!\langle A \rangle\!\rangle\ \langle\!\langle N \rangle\!\rangle) & & & \langle\!\langle (x\!:\!A)L \rangle\!\rangle &:= \lambda g.[\![g : (x\!:\!A)L]\!] \\ \langle\!\langle AB \rangle\!\rangle &:= (ap_{ff}\ \langle\!\langle A \rangle\!\rangle\ \langle\!\langle B \rangle\!\rangle) & & & \langle\!\langle (x\!:\!K)L \rangle\!\rangle &:= \lambda g.[\![g : (x\!:\!K)L]\!] \end{aligned}$$

Figure 5: Encoding CC Terms

jects, families, and types.

$$\begin{aligned} conv_o &: ob \to ob \to o \\ conv_f &: (ob \to o) \to (ob \to o) \to o \\ conv_t &: ((ob \to o) \to o) \to ((ob \to o) \to o) \to o \end{aligned}$$

The $\mathcal{I}$ formulas expressing convertibility for families are given in Figure 7. In this figure and the next, we leave of outermost quantifiers and assume universal quantification over all free variables written as capital letters (possibly with subscripts). The first two formulas encode the (β) rule. Since we only consider canonical derivations, the encoded terms will always be in η-long form, and so we do not include formulas for the (η) rule. We next have two formulas encoding (CONG), two formulas encoding (ξ-ABS), and two formulas encoding (ξ-PROD). The last three formulas encode reflexivity, symmetry, and transitivity. Convertibility for objects is similar except that there are no formulas corresponding to the two for (ξ-PROD), while convertibility for types includes only two such formulas together with formulas expressing reflexivity, symmetry, and transitivity. The following theorem expresses the correctness for this specification.

Lemma 6 Let A and B be families, and $\mathcal{P}$ be the set of formulas encoding convertibility for families given in Figure 7 together with those for convertibility for objects and types. Then $A =_\beta B$ if and only if $\mathcal{P} \longrightarrow (conv_f\ \langle\!\langle A \rangle\!\rangle\ \langle\!\langle B \rangle\!\rangle)$ has a sequent proof.

Proof: This lemma is proved by induction on a derivation of $A =_\beta B$ and Lemma 5. The induction is simultaneous with similar statements for the other two convertibility relations. The proofs are an extension of those given in Felty [7] for a specification of convertibility for LF.

A few remaining assumptions in $\mathcal{I}$ are necessary in order to have a direct correspondence between derivability in CC and provability in $\mathcal{I}$. We include the formulas in Figure 8. The first formula expresses the (PROP-TYPE) rule. The next four formulas express the (PROD) rule at the level of propositions and types. The remaining four rules provide β-conversion for encoded CC terms appearing in both goals and assumptions. While an $\mathcal{I}$ proof may contain nodes with terms representing CC terms that are not necessarily in $\beta\eta$-long form, the corresponding node in the canonical CC derivation will contain the corresponding $\beta\eta$-long form. There are no explicit formulas for the (INIT), (ABS), and (APP) rules. These rules are what we are encoding directly by translating context items. As a result of this translation, the (ABS) rule, for example, corresponds directly to applications of the $\wedge$-R, $\forall$-R, and $\supset$-R inference rules for $\mathcal{I}$.

5 CC Derivations as $\mathcal{I}$ Proofs

In this section, we state the theorems and sketch the proofs showing that for any canonical derivation in CC, there is a corresponding proof in $\mathcal{I}$. We then discuss the correspondence in the reverse direction.

We say that Γ is a *pre-context* if for every pair $P\!:\!Q$

$$
\begin{aligned}
[\![[x\!:\!R]P : (x\!:\!R)Q]\!] &:= [\![R : s_1]\!] \wedge \forall x \left([\![x\!:\!R]\!] \supset [\![Q : s_2]\!]\right) \wedge \forall x \left([\![x\!:\!R]\!] \supset [\![P : Q]\!]\right) \\
[\![P : (x\!:\!R)Q]\!] &:= [\![R : s_1]\!] \wedge \forall x \left([\![x\!:\!R]\!] \supset [\![Q : s_2]\!]\right) \wedge \forall x \left([\![x\!:\!R]\!] \supset [\![Px : Q]\!]\right) \\
&\ \text{where } P \text{ has no leading []-abstraction.} \\
[\![P : Q]\!] &:= \langle\!\langle Q\rangle\!\rangle\ \langle\!\langle P\rangle\!\rangle \quad \text{where } Q \text{ has no leading ()-abstraction.}
\end{aligned}
$$

Figure 6: Translating CC Context Items and Judgments

$$
\begin{aligned}
&(\mathit{conv}_f\ (\mathit{ap}_{\mathit{ff}}\ (\mathit{abs}_{\mathit{ff}}\ B\ K)\ A)\ (B\ A)) \\
&(\mathit{conv}_f\ (\mathit{ap}_{\mathit{fo}}\ (\mathit{abs}_{\mathit{of}}\ B\ A)\ M)\ (B\ M)) \\
&(\mathit{conv}_f\ A_1\ A_2) \wedge (\mathit{conv}_f\ B_1\ B_2) \supset (\mathit{conv}_f\ (\mathit{ap}_{\mathit{ff}}\ A_1\ B_1)\ (\mathit{ap}_{\mathit{ff}}\ A_2\ B_2)) \\
&(\mathit{conv}_f\ A_1\ A_2) \wedge (\mathit{conv}_o\ N_1\ N_2) \supset (\mathit{conv}_f\ (\mathit{ap}_{\mathit{fo}}\ A_1\ N_1)\ (\mathit{ap}_{\mathit{fo}}\ A_2\ N_2)) \\
&\forall x(\mathit{conv}_f\ B_1x\ B_2x) \wedge (\mathit{conv}_f\ A_1\ A_2) \supset (\mathit{conv}_f\ (\mathit{abs}_{\mathit{of}}\ B_1\ A_1)\ (\mathit{abs}_{\mathit{of}}\ B_2\ A_2)) \\
&\forall x(\mathit{conv}_f\ B_1x\ B_2x) \wedge (\mathit{conv}_t\ K_1\ K_2) \supset (\mathit{conv}_f\ (\mathit{abs}_{\mathit{ff}}\ B_1\ K_1)\ (\mathit{abs}_{\mathit{ff}}\ B_2\ K_2)) \\
&(\mathit{conv}_f\ A_1\ A_2) \wedge (\mathit{conv}_f\ B_1\ B_2) \supset (\mathit{conv}_f\ \lambda f.((\mathit{prop}\ A_1) \wedge \forall x(A_1x \supset (\mathit{prop}\ B_1)) \wedge \forall x(A_1x \supset (B_1\ (\mathit{ap}_{\mathit{oo}}\ f\ x)))) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \lambda f.((\mathit{prop}\ A_2) \wedge \forall x(A_2x \supset (\mathit{prop}\ B_2)) \wedge \forall x(A_2x \supset (B_2\ (\mathit{ap}_{\mathit{oo}}\ f\ x))))) \\
&(\mathit{conv}_f\ A_1\ A_2) \wedge (\mathit{conv}_t\ K_1\ K_2) \supset (\mathit{conv}_f\ \lambda f.((\mathit{typ}\ K_1) \wedge \forall x(K_1x \supset (\mathit{prop}\ A_1)) \wedge \forall x(K_1x \supset (A_1\ (\mathit{ap}_{\mathit{of}}\ f\ x)))) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \lambda f.((\mathit{typ}\ K_2) \wedge \forall x(K_2x \supset (\mathit{prop}\ A_2)) \wedge \forall x(K_2x \supset (A_2\ (\mathit{ap}_{\mathit{of}}\ f\ x))))) \\
&(\mathit{conv}_f\ P\ P) \\
&(\mathit{conv}_f\ Q\ P) \supset (\mathit{conv}_f\ P\ Q) \\
&(\mathit{conv}_f\ P\ R) \wedge (\mathit{conv}_f\ R\ Q) \supset (\mathit{conv}_f\ P\ Q)
\end{aligned}
$$

Figure 7: Convertibility for Propositions

$$
\begin{aligned}
&(\mathit{typ}\ \mathit{prop}) \\
&(\mathit{prop}\ A) \wedge \forall x(Ax \supset (\mathit{prop}\ B) \supset (\mathit{prop}\ \ \lambda f.((\mathit{prop}\ A) \wedge \forall x(Ax \supset (\mathit{prop}\ B)) \wedge \forall x(Ax \supset (B\ (\mathit{ap}_{\mathit{oo}}\ f\ x)))))) \\
&(\mathit{typ}\ K) \wedge \forall x(Kx \supset (\mathit{prop}\ B) \supset (\mathit{prop}\ \ \lambda f.((\mathit{typ}\ K) \wedge \forall x(Kx \supset (\mathit{prop}\ B)) \wedge \forall x(Kx \supset (B\ (\mathit{ap}_{\mathit{of}}\ f\ x)))))) \\
&(\mathit{prop}\ A) \wedge \forall x(Ax \supset (\mathit{typ}\ L) \supset (\mathit{typ}\ \ \lambda g.((\mathit{prop}\ A) \wedge \forall x(Ax \supset (\mathit{typ}\ L)) \wedge \forall x(Ax \supset (L\ (\mathit{ap}_{\mathit{fo}}\ g\ x)))))) \\
&(\mathit{typ}\ K) \wedge \forall x(Kx \supset (\mathit{typ}\ L) \supset (\mathit{typ}\ \ \lambda g.((\mathit{typ}\ K) \wedge \forall x(Kx \supset (\mathit{typ}\ L)) \wedge \forall x(Kx \supset (L\ (\mathit{ap}_{\mathit{ff}}\ g\ x)))))) \\
&(\mathit{conv}_t\ K\ L) \wedge (\mathit{conv}_f\ A\ B) \wedge LB \supset KA \\
&(\mathit{conv}_f\ A\ B) \wedge (\mathit{conv}_o\ M\ N) \wedge BN \supset AM \\
&(\mathit{conv}_t\ K\ L) \wedge (\mathit{conv}_f\ A\ B) \wedge LB \wedge (KA \supset G) \supset G \\
&(\mathit{conv}_f\ A\ B) \wedge (\mathit{conv}_o\ M\ N) \wedge BN \wedge (AM \supset G) \supset G
\end{aligned}
$$

Figure 8: Formulas Expressing Some CC Typing and Conversion Rules

in Γ, $[\![P:Q]\!]$ is well-defined. We write $[\![\Gamma]\!]$ to denote the set containing $[\![P:Q]\!]$ for every $P:Q$ in Γ.

In this section, we write $\mathcal{P}_{CC}$ to denote the set of assumptions containing the formulas in Figures 7 and 8 as well as the convertibility formulas for objects and types discussed but not shown.

Lemma 7 Let P, Q, P', Q', and R be CC terms such that $P =_{\beta\eta} P'$ and $Q =_{\beta\eta} Q'$, and $[\![P:Q]\!]$ is well-defined. Let x be a variable and Γ a pre-context.

1. $\mathcal{P}_{CC}, [\![\Gamma]\!] \vdash_I [\![P:Q]\!]$ iff $\mathcal{P}_{CC}, [\![\Gamma]\!] \vdash_I (\langle\!\langle Q\rangle\!\rangle\ \langle\!\langle P'\rangle\!\rangle)$.
2. If $\mathcal{P}_{CC}, [\![\Gamma]\!] \vdash_I [\![([R/x]P) : ([R/x]Q)]\!]$, then $\mathcal{P}_{CC}, [\![\Gamma]\!] \vdash_I [\langle\!\langle R\rangle\!\rangle/x][\![P:Q]\!]$.
3. $\mathcal{P}_{CC}, [\![\Gamma]\!], [\![x:Q]\!] \vdash_I A$ iff $\mathcal{P}_{CC}, [\![\Gamma]\!], [\![x:Q']\!] \vdash_I A$, for any $\mathcal{I}$ formula A.
4. $\mathcal{P}_{CC}, [\![\Gamma]\!] \vdash_I [\![P:Q]\!]$ iff $\mathcal{P}_{CC}, [\![\Gamma]\!] \vdash_I [\![P:Q']\!]$.

Proof: (1) and (2) are proved by induction on the structure of Q. (3) follows directly from the fact that the last two formulas in Figure 8 are in $\mathcal{P}_{CC}$. (4) is proved by induction on a derivation of $Q =_{\beta\eta} Q'$.

Theorem 8 Let Γ be a valid context and let P and Q be CC terms. If $\Gamma \vdash P:Q$ has a canonical derivation in CC, then $\mathcal{P}_{CC}, [\![\Gamma]\!] \longrightarrow [\![P:Q]\!]$ has a sequent proof in $\mathcal{I}$.

Proof: The theorem follows from Lemmas 4, 5, 6, and 7, and induction on the height of a canonical CC derivation. It illustrates directly how to construct an $\mathcal{I}$ proof from a CC derivation.

The correspondence in the reverse direction is not as direct. In fact, there are actually "too many" $\mathcal{I}$ proofs, not all of which correspond directly to CC derivations. The extra proofs result from the last four formulas of Figure 8 which allow too much freedom in the conversion of terms. If we restrict the way these subformulas are used so that CC terms are always reduced to normal form, we obtain a direct correspondence in this direction also. More specifically, consider the backwards construction of a sequent proof of an encoded assertion. For all rules except the $\forall$-L rule, whenever all encoded CC terms appearing in formulas in the conclusion are in normal form, those in the premises are also in normal form. We can enforce a normal-form invariant by requiring that all terms in the premise of an application of $\forall$-L get immediately normalized. We must also augment the specification of conversion with formulas defining when a term is in normal form.

One way to formalize this correspondence is to modify the definition of the encoding so that subgoals for normalization are placed everywhere they are needed in the assumptions and the goal. To do so requires dividing the encoding of judgments into two mutually recursive functions, one for encoding goals and the other for encoding assumptions. In particular, normalization formulas must always occur positively in goals and negatively in assumptions so that they always appear as goals and never as assumptions. However, having two functions complicates the encoding of CC terms since whenever families occur inside terms, there must be both a positive and a negative version. Although this complicates the encoding technically, it adds no new difficulty to the proofs of the correspondence between $\mathcal{I}$ proofs and CC derivations. In fact, this correspondence can be proven by extending the proof in Felty [7], where both positive and negative encodings are used to encode the LF type theory. The specification for terms in normal form for LF given there can also be extended directly to CC.

6 Implementing a Search Procedure

We consider two examples taken from Dowek [5]. First, we start with the following context.

T: Prop
R: $T \to T \to$ Prop
eq: $T \to T \to$ Prop
$antisym$: $(x:T)(y:T)(R\ x\ y) \to (R\ y\ x) \to (eq\ x\ y)$
a: T
b: T
u: $(R\ a\ b)$
v: $(R\ b\ a)$

Using the inference rules of Figure 3, we can prove that the object $(antisym\ a\ b\ u\ v)$ has proposition $(eq\ a\ b)$. The encoded version of this type judgment is:

$$\begin{array}{l}(ap_{fo}\ (ap_{fo}\ eq\ a)\ b\\ \qquad (ap_{oo}\ (ap_{oo}\ (ap_{oo}\ (ap_{oo}\ antisym\ a)\ b)\ u)\ v))\end{array}$$

which has a simple proof from the encoded context shown below.

$$\begin{array}{l}(prop\ T)\\ \forall x((T\ x) \supset \forall y((T\ y) \supset (prop\ (ap_{fo}\ (ap_{fo}\ R\ x)\ y))))\\ \forall x((T\ x) \supset \forall y((T\ y) \supset (prop\ (ap_{fo}\ (ap_{fo}\ eq\ x)\ y))))\\ \forall x((T\ x) \supset \forall y((T\ y) \supset \forall w((ap_{fo}\ (ap_{fo}\ R\ x)\ y\ w) \supset\\ \quad \forall z((ap_{fo}\ (ap_{fo}\ R\ y)\ x\ z) \supset\\ \qquad (ap_{fo}\ (ap_{fo}\ eq\ x)\ y\\ \qquad\quad (ap_{oo}\ (ap_{oo}\ (ap_{oo}\ (ap_{oo}\\ \qquad\qquad antisym\ x)\ y)\ w)\ z))))))\end{array}$$

$(T\ a)$
$(T\ b)$
$(ap_{fo}\ (ap_{fo}\ R\ a)\ b\ u)$
$(ap_{fo}\ (ap_{fo}\ R\ b)\ a\ v)$

For illustration purposes in this and the next example, when using the second clause of the definition of the encoding in Figure 6, we leave off the first two conjuncts, *i.e.*, we used only $\forall x\ (\llbracket x:R\rrbracket \supset \llbracket Px : Q\rrbracket)$. (The additional conjuncts provide assumptions that are not needed in these examples.)

As a slightly more complex example, consider the following context.

$A:\mathrm{Prop}$
$B:\mathrm{Prop}$
$I:\mathrm{Prop} \rightarrow \mathrm{Prop}$
$u:(P:\mathrm{Prop})((I\ P) \rightarrow P)$
$v:(I\ (A \rightarrow B))$
$w:A$

In CC, we can show that the object $(u\ (A \rightarrow B)\ v\ w)$ has proposition B. To do so, the proposition P in the fourth context item must be instantiated with the functional term $(A \rightarrow B)$. By the encoding on terms, $\langle\!\langle A \rightarrow B\rangle\!\rangle$ is:

$$\lambda f.((prop\ A) \wedge \\ \quad \forall x(Ax \supset (prop\ B)) \wedge \forall x(Ax \supset (B\ (ap_{oo}\ f\ x)))).$$

We write Q below to abbreviate this term. Thus, in $\mathcal{I}$, we must prove $(B\ (ap_{oo}\ (ap_{oo}\ (ap_{of}\ u\ Q)\ v)\ w))$ from the following assumptions.

$(prop\ A)$
$(prop\ B)$
$\forall P((prop\ P) \supset (prop\ (ap_{ff}\ I\ P)))$
$\forall P((prop\ P) \supset$
$\quad \forall z((ap_{ff}\ I\ P\ z) \supset (P\ (ap_{oo}\ (ap_{of}\ u\ P)\ z))))$
$(ap_{ff}\ I\ Q\ v)$
$(A\ w)$

We can begin the corresponding sequent proof in $\mathcal{I}$ by applying $\forall$-L to the fourth assumption and instantiating P with Q. The rest of the proof follows easily.

Using the search procedure for $\mathcal{I}$ described in Section 2, we can implement a search procedure for CC encoded judgments. We can use such a procedure for both type checking and proof search in CC. For type checking, both terms in a judgment are given and the search is straightforward. The head of the CC term to be type checked completely determines which assumption to use in backchaining. In the first example, the head of the object in the original goal is *antisym*, and so the formula for *antisym* (the fourth formula in the encoded context above) is the one that must be used by the BACKCHAIN operation. In the $\mathcal{I}$ formulas in the above examples, we call the rightmost atomic formula the *head subformula*. In an interpreter that uses logic variables to determine substitutions, the variables x, y, z, w in the head subformula of the *antisym* assumption can be replaced by logic variables M_1, M_2, M_3, M_4 which get instantiated to a, b, u, v, respectively, upon unification with the goal. The resulting subgoals are $(T\ a)$, $(T\ b)$, $(ap_{fo}\ (ap_{fo}\ R\ a)\ b\ u)$, and $(ap_{fo}\ (ap_{fo}\ R\ b)\ a\ v)$, which follow immediately. Type checking the second example is not as simple. The head u of the CC term in the goal determines which assumption to use as before, but we cannot simply unify the head subformula $(P\ (ap_{oo}\ (ap_{of}\ u\ P)\ z))))$ with the goal $(B\ (ap_{oo}\ (ap_{oo}\ (ap_{of}\ u\ Q)\ v)\ w))$. In this case, we need to instantiate P with Q to obtain a non-atomic subformula. In general, for type checking encoded CC judgments, the logical structure needed in an instantiation of a variable head is completely determined by the CC term. Here, the fact that u in the goal is applied to one more argument than u in the head subformula of the assumption indicates that we need one universal quantifier over one implication where the right subformula is atomic. In general, if the goal has n extra arguments, the substitution must result in a formula of the form $\forall x_1(G_1 \supset \cdots \forall x_n(G_n \supset A)\cdots)$ where A is atomic. In addition, the n extra arguments in the goal determine what the structure of the subgoals $G_1, \ldots, G_n$ should be.

Proof search in CC corresponds to stating a proposition or type and searching for an object or family that inhabits it. In this case, we start with a logic variable to represent the object or family we want to find and fill it in incrementally using unification during BACKCHAIN steps on assumptions. For example, we could start with the goal $(ap_{fo}\ (ap_{fo}\ eq\ a)\ b\ M)$ with M a logic variable. Using BACKCHAIN on the *antisym* assumption with new logic variables M_1, M_2, M_3, M_4, the variable M gets partially instantiated to the term $(ap_{oo}\ (ap_{oo}\ (ap_{oo}\ (ap_{oo}\ antisym\ M_1)\ M_2)\ M_3)\ M_4)$. It is not until further BACKCHAIN steps that these variables get instantiated to a, b, u, v, respectively, completing the search.

Although this example is simple, the second example and search in general is much more complicated. We list several problems and discuss ways of overcoming them.

One problem, as discussed in Section 2, is that the BACKCHAIN operation needed for $\mathcal{I}$ is highly nondeterministic. Consider again the second example above, this time starting with the goal $(B\ M)$ where M is a logic variable. We need to introduce a substitution for P that transforms the head subformula to a non-atomic formula, but we have no indication of what this substitution should be. However, we can modify the interpreter so that it provides a procedure that is incomplete, but behaves like the *transitively complete* search procedure for CC given in Dowek [5]. Informally, a procedure is transitively complete if when trying to prove a typing assertion $\Gamma \vdash P : Q$, it is possible to prove a series of typing "lemmas" eventually leading to the desired result, *i.e.*, there are provable assertions

$$\begin{array}{l} \Gamma \vdash P_1 : Q_1 \\ \Gamma, P_1{:}Q_1 \vdash P_2 : Q_2 \\ \vdots \\ \Gamma, P_1{:}Q_1, \ldots, P_n{:}Q_n \vdash P : Q \end{array}$$

where $n \geq 0$. The modification to our procedure can be described simply as follows: don't restrict the BACKCHAIN to work on atomic formulas, always attempt BACKCHAIN before any of the other search operations, and always use unification to attempt to unify the current goal with the head subformula of an assumption. In the second example above, one lemma must be proved before we can prove that $(u\ (A \rightarrow B)\ v\ w)$ has proposition B, namely that $(u\ (A \rightarrow B)\ v)$ has proposition $A \rightarrow B$. In terms of search, this means that we must find a proof of $A \rightarrow B$ before finding a proof of B. Finding a proof of $A \rightarrow B$ means solving the subgoal $(\mathit{prop}\ A) \wedge \forall x((A\ x) \supset (\mathit{prop}\ B)) \wedge \forall x((A\ x) \supset (B\ (M'x)))$ where M' is a logic variable. Instead of applying the AND, GENERIC and AUGMENT operations, we directly unify the formula with $(P\ (ap_{oo}\ (ap_{of}\ u\ P)\ z))))$. There is one solution and the remaining subgoals are solved trivially, instantiating M' to $\lambda x.(ap_{oo}\ (ap_{oo}\ (ap_{of}\ u\ Q)\ v)\ x)$. We then add the solved subgoal as an assumption:

$$\begin{array}{l} (\mathit{prop}\ A) \wedge \forall x((A\ x) \supset (\mathit{prop}\ B)) \wedge \\ \quad \forall x((A\ x) \supset (B\ (ap_{oo}\ (ap_{oo}\ (ap_{of}\ u\ Q)\ v)\ x))). \end{array}$$

The goal $(B\ M)$ is now easily solved by backchaining on the third conjunct of this assumption, obtaining $(ap_{oo}\ (ap_{oo}\ (ap_{of}\ u\ Q)\ v)\ w)$ for logic variable M.

In general, the proofs that cannot be discovered without first proving a lemma are those that need to use a context item of the form $Q : (x_1 : P_1) \cdots (x_n : P_n)(vS_1 \ldots S_q)$ where v is a variable, and the term to be substituted for this variable is a proposition (or abstraction over a proposition) that is not primitive. Induction principles for example are expressed in this form. As in Dowek [5], our procedure handles the case when a goal directly represents the property to be proved by induction, but not the case when a generalization of the induction hypothesis is needed.

Even with the modified procedure, extra control is likely to be needed to decide which assumptions to use in backchaining at each step. It will often be the case that more than one can be applied. One way to provide such control is to implement a tactic style interactive theorem prover which allows a user to guide search step by step as well as incorporate some heuristic search procedures. An example of such a system for CC is Coq [6]. A λProlog implementation of tactic style search is presented in Felty [9]. Using this implementation, a tactic theorem prover for encoded CC can be directly implemented. In such a theorem prover, it is possible for a user to supply generalized induction hypotheses and other hints directly when needed. In addition, in a tactic theorem prover, it may often be possible to provide specialized procedures for finding such hypotheses automatically.

Finally, although the examples here didn't show it, when one of the last four formulas in Figure 8 is needed, there is much non-determinism in choosing terms that are $\beta\eta$-equivalent. However, as discussed in the previous section, we actually want to work with terms in normal form. If the specification of conversion in Figure 7 is replaced by a normalization procedure, this non-determinism will be eliminated.

7 Conclusion and Future Work

We have demonstrated the formal correspondence between two distinct languages by presenting an encoding of one in the other. As mentioned, this encoding provides a framework in which to study how theorem proving techniques designed for one system can be applied to proof search in the other. We have discussed in some detail the search procedure for the encoded language, CC, that is derived from a search procedure for the meta-logic $\mathcal{I}$. In addition, we mentioned several other theorem proving techniques worth further investigation such as those of Bledsoe and Dowek. Although developed exclusively for one of the two languages, such techniques may be able to aid in providing automatic support for search in the other language. One technique of interest that has not yet been mentioned is unification. Studying unification of CC terms as reflected in the higher-order logic setting, for example, should provide additional insight into this complex but important operation.

We have not considered the possibility of translating $\mathcal{I}$ formulas into CC. We consider here the subset of $\mathcal{I}$ without conjunction. (Any formula in $\mathcal{I}$ can in fact be mapped to an equivalent set of $\mathcal{I}$ formulas that do not contain conjunction.) This translation is particularly simple, mainly because any simple type is also a type in CC. Let $\mathcal{P}$ be a set of assumptions and G a goal. We build a CC context Γ as follows. First, for each constant c of type τ appearing in the formulas in $\mathcal{P}$ and in G, add $c : \tau'$ to Γ where τ' is τ with all occurrences of o replaced by Prop. Second, for each formula $D \in \mathcal{P}$, introduce a new constant k not used in the translation to this point. Add to the end of Γ the CC pair $k : D'$ where D' is essentially D with $B \supset C$ written as $(x{:}B)C$, $\forall_\tau x\ B$ written as $(x{:}\tau)B$, and all occurrences of o replaced by Prop. In G, make the same replacements for implication, universal quantifiers, and occurrences of o to obtain a CC term G'. Then, proving the sequent $\mathcal{P} \longrightarrow G$ in $\mathcal{I}$ corresponds to finding a term P such that $\Gamma \vdash P : G'$ has a type derivation in CC.

As mentioned earlier, the encoding presented here should extend directly to generalized type systems so that the number of levels of terms in a type system corresponds directly to the maximum order of the types of constants used to encode terms. In Felty [7], an encoding is given for the LF type theory with an encoding on terms similar to the one presented here. Both the number of levels in LF and the maximum order of types of constants in the encoding is two. A more direct encoding where constants for application and abstraction are not needed is presented in Felty [8]. There, application and abstraction in LF are encoded directly as application and abstraction in the metalanguage. Furthermore, each LF variable is mapped to a constant of the metalanguage such that the order of the LF type and the corresponding simple type are exactly the same. Although the simple type may have less information, it captures the basic structure of the LF type. Such types can be of arbitrary order. We cannot, however, give such a direct encoding of CC in $\mathcal{I}$ in the same way. In particular, a variable f of polymorphic type in CC cannot be mapped to a constant f of the simply-typed λ-calculus. There is no simple type that can capture the structure of a polymorphic type. However, such a direct encoding should be possible in a higher-order logic that extends $\mathcal{I}$ so that the terms of the logic are the terms of the polymorphic λ-calculus.

References

[1] Hank Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):124–154, April 1991.

[2] W. W. Bledsoe. A maximal method for set variables in automatic theorem proving. *Machine Intelligence*, 9:53–100, 1979.

[3] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[4] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, February/March 1988.

[5] Gilles Dowek. *Démonstration Automatique dans le Calcul des Constructions.* PhD thesis, L'Université Paris VII, December 1991.

[6] Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Christine Paulin-Mohring, and Benjamin Werner. The coq proof assistant user's guide. Technical Report 134, INRIA, December 1991.

[7] Amy Felty. *Specifying and Implementing Theorem Provers in a Higher-Order Logic Programming Language.* PhD thesis, University of Pennsylvania, Technical Report MS-CIS-89-53, August 1989.

[8] Amy Felty. Encoding dependent types in an intuitionistic logic. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 215–251. Cambridge University Press, 1991.

[9] Amy Felty. Implementing tactics and tacticals in a higher-order logic programming language. *Journal of Automated Reasoning*, To appear.

[10] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinatory Logic and Lambda Calculus.* Cambridge University Press, 1986.

[11] William A. Howard. The formulae-as-type notion of construction, 1969. In *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490. Academic Press, 1980.

[12] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.