

# Reasoning in Extensional Type Theory with Equality

Chad E. Brown

Universität des Saarlandes, Saarbrücken, Germany, cebrown@ags.uni-sb.de

**Abstract.** We describe methods for automated theorem proving in extensional type theory with primitive equality. We discuss a complete, cut-free sequent calculus as well as a compact representation of cut-free (ground) proofs as extensional expansion dags. Automated proof search can be realized using a few operations to manipulate extensional expansion dags with variables. These search operations form a basis for complete search procedures. Procedures based on these ideas are implemented in the higher-order theorem prover TPS.

## 1 Introduction

Church's type theory [12] is a form of higher-order logic which is sufficiently powerful to represent much of traditional mathematics. The original Hilbert-style proof theory in [12] does not provide a convenient calculus for automated deduction. In an effort to study automated deduction for higher-order logic, fragments of Church's type theory have been considered. In particular, the higher-order theorem proving system TPS has traditionally searched for proofs in elementary type theory. Elementary type theory is Church's type theory without axioms of extensionality, descriptions, choice, or infinity. Andrews introduced a Hilbert-calculus  $\mathcal{T}$  for elementary type theory in [2]. Three important steps regarding the development of automated reasoning in elementary type theory can be sketched as follows:

1. In [2] Andrews proved a cut-free sequent calculus complete relative to  $\mathcal{T}$ .
2. Miller [15] demonstrated that every theorem of elementary type theory has an *expansion proof* as described in [15; 16; 6; 4].
3. Procedures were developed to search for expansion proofs by manipulating expansion trees (with progressively instantiated variables). The implementation of such search procedures in TPS are described in [3; 6].

We can generalize the three steps above and add another to outline a method for studying automated reasoning in a logic.

1. Develop a cut-free ground calculus for the logic.
2. Develop a compact representation of cut-free ground proofs.
3. Design a set of search operations for approximating compact ground proofs (using variables).

4. Establish completeness of search by verifying that some selection of the search operations will lead to a proof, if a proof exists.

A ground expansion proof can be approximated by an expansion tree with expansion variables. First-order methods can be naturally generalized to search for expansion proofs. In particular, TPS combines mating search with Huet's higher-order pre-unification. For completeness, one must also consider *primitive substitutions* for set variables. A primitive substitution introduces a logical constant or projection for a variable at the head of a literal. Primitive (and general) substitutions in TPS are discussed in [6] on page 331. Such enumeration techniques place practical limitations on the search space. Intuitively, TPS attempts to converge towards a ground expansion proof by adding appropriate connections, performing higher-order unification steps and performing primitive substitutions.

To complete the analysis of automated reasoning in elementary type theory, such a search procedure would need to be proven complete. That is, one must show that some sequence of search operations applied to expansion trees successfully terminates if an expansion proof exists. This has never been carried out for expansion proofs. One reason is TPS traditionally has performed primitive substitutions (and most quantifier duplications) in a pre-processing step. Since new quantifiers can be introduced during search (via primitive substitutions), one cannot expect restricting primitive substitution applications to pre-processing leads to a complete search procedure.

Another interesting fragment of Church's type theory is extensional type theory. Extensional type theory adds principles of Boolean and functional extensionality to elementary type theory. The system LEO [8] searches using a resolution calculus for extensional type theory. TPS can now also prove theorems of extensional type theory using an appropriately modified notion of an expansion proof. We also extend the notion of expansion proof to include reasoning with primitive equality.

One advantage of working with extensional type theory is that the theory is closer to mathematical reasoning. For example, in mathematics one does not distinguish between  $A \cup B$  and  $B \cup A$ . These sets could be different in models of elementary type theory, but not in models of extensional type theory.

Another advantage of working with extensional type theory is that one can simplify the search for instantiations of set variables. (This was, in fact, the motivation for adding extensionality reasoning to TPS.) While primitive substitutions are still necessary, one can obtain certain restrictions on which logical constants must be available for primitive substitutions. Also, one can represent some theorems in a natural manner which avoids introducing certain set variables. In particular, the use of primitive equality instead of Leibniz equality completely eliminates the set variables introduced by Leibniz equality.

A lifting lemma is proven for the resolution calculus in [8] in order to show completeness of search. However, this lifting lemma required a *Flex-Flex rule* to apply a substitution to flex-flex pair. Since such a rule is notoriously branching, LEO does not actually apply such a rule during search. One of the goals

of the development of extensional expansion proofs and corresponding search procedures was to show completeness of search without requiring operations on flex-flex pairs. In the end, we not only show completeness of search without any *Flex-Flex rule*, but also without considering connections between two flexible nodes.

We begin by describing extensional type theory relative to a signature of logical constants. By formulating extensional type theory in this way, we make precise the possible restrictions on set instantiations and hence primitive substitutions. We proceed by describing each of the four steps outlined above relative to (fragments of) extensional type theory. We describe a cut-free sequent calculus (complete with respect to an appropriate semantics). We describe extensional expansion proofs as extensional expansion dags (generalizing expansion trees) which satisfy certain properties. We indicate completeness of ground extensional expansion proofs relative to extensional type theory. When searching for extensional expansion proofs, we use variables which are progressively instantiated. We give a set of operations one can perform during search on extensional expansion dags (with variables). A lifting argument shows completeness of search once one has completeness of the ground case. Finally, we indicate some theorems which can be proven automatically in TPS using the new methods. This paper describes the work contained in [11], focusing on the results for automated proof search in extensional type theory.

## 2 Terms and Propositions

In first-order logic, one first defines terms (inductively) then atomic propositions (using terms and relations) and finally propositions (inductively). In higher-order logic, one can inductively define the terms of type  $\alpha$  and let propositions be the terms of a particular type  $o$  of truth values. However, in order to express propositions as a term of type  $o$ , one must use logical constants such as  $\neg_{oo}$ ,  $\vee_{ooo}$  and  $\Pi_{o(o\alpha)}^\alpha$ . For completeness of proof search, one must consider primitive substitutions for each such logical constant. Since one of our purposes is to restrict primitive substitutions, it is worthwhile to define propositions at one level higher than terms (as is done in the first-order case).

First, we define the set of (simple) types inductively.  $o$  is the type of truth values,  $\iota$  is a type of individuals and  $(\alpha\beta)$  is a type of functions from  $\beta$  to  $\alpha$  whenever  $\alpha$  and  $\beta$  are types. Suppose  $\mathcal{S}$  is a set of typed logical constants,  $\mathcal{V}$  is a set of typed variables and  $\mathcal{P}$  is a set of typed parameters. For each type  $\alpha$ , we assume the set  $\mathcal{V}_\alpha$  (variables of type  $\alpha$ ) and the set  $\mathcal{P}_\alpha$  (parameters of type  $\alpha$ ) are infinite. Next, we can inductively define the set  $wff_\alpha(\mathcal{S})$  of terms of type  $\alpha$  using logical constants in  $\mathcal{S}$ , variables, parameters, application and  $\lambda$ -abstraction. We make the dependence on  $\mathcal{S}$  explicit in order to consider different sets of logical constants.

To formulate higher-order logic, Church [12] assumed the signature included logical constants  $\neg_{oo}$ ,  $\vee_{ooo}$  and  $\Pi_{o(o\alpha)}^\alpha$  for each type  $\alpha$ . From these, one can define the other logical operators as in [12]. Equality at type  $\alpha$  can be defined

by Leibniz equality. The alternative pursued in [5] is to have primitive equality  $=_{o\alpha}^\alpha$  in the signature for each type  $\alpha$ . The logical connectives and quantifiers can be defined from primitive equality (assuming full extensionality). In the general case, we will not assume any logical constants to be in the signature  $\mathcal{S}$ . We will always assume, however, that  $\mathcal{S}$  is a subset of the collection

$$\{\top_o, \perp_o, \neg_{oo}, \wedge_{ooo}, \vee_{ooo}, \supset_{ooo}, \equiv_{ooo}\} \\ \cup \{II_{o(o\alpha)}^\alpha \mid \alpha \in \mathcal{T}\} \cup \{\Sigma_{o(o\alpha)}^\alpha \mid \alpha \in \mathcal{T}\} \cup \{=_{o\alpha}^\alpha \mid \alpha \in \mathcal{T}\}.$$

Without (enough) logical constants we should not consider the type theory to be “higher-order logic” since one cannot define the same sets using the restricted language as one can define using higher-order logic. Each collection of logical constants yields a fragment of higher-order logic.

The set  $prop(\mathcal{S})$  of *propositions* over a signature  $\mathcal{S}$  is defined inductively.

- If  $\mathbf{A} \in wff_o(\mathcal{S})$ , then  $\mathbf{A} \in prop(\mathcal{S})$ .
- If  $\alpha$  is a type and  $\mathbf{A}, \mathbf{B} \in wff_\alpha(\mathcal{S})$ , then  $[\mathbf{A} \doteq^\alpha \mathbf{B}] \in prop(\mathcal{S})$ .
- $\top \in prop(\mathcal{S})$ .
- If  $\mathbf{M} \in prop(\mathcal{S})$ , then  $[\neg \mathbf{M}] \in prop(\mathcal{S})$ .
- If  $\mathbf{M}, \mathbf{N} \in prop(\mathcal{S})$ , then  $[\mathbf{M} \vee \mathbf{N}] \in prop(\mathcal{S})$ .
- If  $\mathbf{M} \in prop(\mathcal{S})$ , then  $[\forall x_\alpha \mathbf{M}] \in prop(\mathcal{S})$ .

We use the notation  $\top, \neg, \vee$  and  $\forall$  to distinguish these constructors (at the level of propositions) from the corresponding logical constants  $\top, \neg, \vee$  and  $II^\alpha$ .

Suppose  $\mathcal{S}$  is a signature with  $\neg \in \mathcal{S}$  and  $\mathbf{A} \in wff_o(\mathcal{S})$ . Then  $\neg \mathbf{A} \in wff_o(\mathcal{S})$  is both a term of type  $o$  and a proposition. Also,  $\mathbf{A}$  is a term of type  $o$  and a proposition. Hence  $\neg \mathbf{A}$  is a proposition, but is *not* a term of type  $o$ .

We assume the usual notions of  $\alpha, \beta$  and  $\eta$  conversion for terms. These can be easily extended (by induction) to propositions along with notions of substitution,  $\beta$ -normalization,  $\beta\eta$ -normalization, etc. We use  $\mathbf{M}^\downarrow$  to denote the  $\beta\eta$ -normal form of  $\mathbf{M}$  (whether  $\mathbf{M}$  is a term or a proposition). Likewise, the notions of free and bound variables is defined as usual for terms and propositions. A term or proposition is *closed* if it contains no free variables. A *sentence* is a closed proposition.

An advantage of adding this extra level of propositions is that many theorems can be stated as propositions without assuming any logical constants are in  $\mathcal{S}$ . For example, the surjective form of Cantor’s Theorem can be stated as the sentence

$$\neg \exists g_{oi} \forall f_{oi} \exists j_i \cdot g j \doteq^{oi} f \tag{1}$$

which is in  $prop(\emptyset)$ . (We use  $\exists$  as shorthand for  $\neg \forall \neg$ .) If we insisted on representing Cantor’s Theorem as a term of type  $o$ , then we would need several logical constants. By separating the levels of terms and propositions, we can distinguish between the expressive power of a fragment of type theory from the proof strength of the fragment.

### 3 Sequent Calculus

For any signature  $\mathcal{S}$  of logical constants, a sequent calculus  $\mathcal{G}_{\beta\text{fb}}^{\mathcal{S}}$  is defined in [11] (where sequents are multisets of sentences). Using this proof theory, we define the  $\mathcal{S}$ -fragment of extensional type theory by the set of theorems of  $\mathcal{G}_{\beta\text{fb}}^{\mathcal{S}}$  (provable sequents containing only one sentence). The rules of  $\mathcal{G}_{\beta\text{fb}}^{\mathcal{S}}$  are shown in Figure 1. The rules  $\mathcal{G}(\forall^W)$  and  $\mathcal{G}(f^W)$  require the usual condition that the parameter  $W$  be new with respect to the sequent in the conclusion of the rule. The rules  $\mathcal{G}(\neg\forall, \mathcal{S})$  and  $\mathcal{G}(\neg\equiv, \mathcal{S})$  depend directly on the signature  $\mathcal{S}$ . (The more logical constants are in  $\mathcal{S}$ , the more terms can be used in these rules.) The rules  $\mathcal{G}(Init^{\bar{\cdot}})$  and  $\mathcal{G}(Dec)$  apply for any  $n \geq 0$  and parameter  $H$  (of the appropriate type).

|  |   |   |   |
|--|---|---|---|
| $\frac{\Gamma, \mathbf{M}, \mathbf{M}}{\Gamma, \mathbf{M}} \mathcal{G}(Contr)$   | $\frac{\Gamma, \mathbf{M}^\dagger}{\Gamma, \mathbf{M}} \mathcal{G}(\beta\eta)$  | $\frac{}{\Gamma, \top} \mathcal{G}(\top)$   | $\frac{\Gamma, \mathbf{M}}{\Gamma, \neg\neg\mathbf{M}} \mathcal{G}(\neg\neg)$ |
| $\frac{\Gamma, \mathbf{M}, \mathbf{N}}{\Gamma, [\mathbf{M} \vee \mathbf{N}]} \mathcal{G}(\vee)$  |   | $\frac{\Gamma, \neg\mathbf{M} \quad \Gamma, \neg\mathbf{N}}{\Gamma, \neg[\mathbf{M} \vee \mathbf{N}]} \mathcal{G}(\neg\vee)$  |   |
| $\frac{\Gamma, [[W/x]\mathbf{M}]}{\Gamma, [\forall x_\alpha \mathbf{M}]} \mathcal{G}(\forall^W)$   | $\frac{\Gamma, \neg[[C/x]\mathbf{M}] \quad \mathbf{C} \in \text{cuff}_\alpha(\mathcal{S})}{\Gamma, \neg[\forall x_\alpha \mathbf{M}]} \mathcal{G}(\neg\forall, \mathcal{S})$  |   |   |
| $\frac{\Gamma, \mathbf{A}^\sharp \quad \mathbf{A} \in \text{cuff}_o(\mathcal{S})}{\Gamma, \mathbf{A}} \mathcal{G}(\sharp)$   | $\frac{\Gamma, \neg(\mathbf{A}^\sharp) \quad \mathbf{A} \in \text{cuff}_o(\mathcal{S})}{\Gamma, \neg\mathbf{A}} \mathcal{G}(\neg\sharp)$  |   |   |
| $\frac{\Gamma, [[\mathbf{G} W] \equiv^\alpha [\mathbf{H} W]]}{\Gamma, [\mathbf{G} \equiv^{\alpha\beta} \mathbf{H}]} \mathcal{G}(f^W)$  | $\frac{\Gamma, \neg[[\mathbf{G} \mathbf{B}] \equiv^\alpha [\mathbf{H} \mathbf{B}]] \quad \mathbf{B} \in \text{cuff}_\beta(\mathcal{S})}{\Gamma, \neg[\mathbf{G} \equiv^{\alpha\beta} \mathbf{H}]} \mathcal{G}(\neg\equiv, \mathcal{S})$ |   |   |
| $\frac{\Gamma, \neg\mathbf{A}, \mathbf{B} \quad \Gamma, \neg\mathbf{B}, \mathbf{A}}{\Gamma, [\mathbf{A} \equiv^o \mathbf{B}]} \mathcal{G}(b)$  |   | $\frac{\Gamma, \mathbf{A}, \mathbf{B} \quad \Gamma, \neg\mathbf{A}, \neg\mathbf{B}}{\Gamma, \neg[\mathbf{A} \equiv^o \mathbf{B}]} \mathcal{G}(\neg\equiv^o)$  |   |
| $\frac{\Gamma, [\mathbf{A} \equiv^t \mathbf{C}] \quad \Gamma, [\mathbf{B} \equiv^t \mathbf{D}]}{\Gamma, \neg[\mathbf{A} \equiv^t \mathbf{B}], [\mathbf{C} \equiv^t \mathbf{D}]} \mathcal{G}(EU\text{ni}f_1)$           | $\frac{\Gamma, [\mathbf{A} \equiv^t \mathbf{D}] \quad \Gamma, [\mathbf{B} \equiv^t \mathbf{C}]}{\Gamma, \neg[\mathbf{A} \equiv^t \mathbf{B}], [\mathbf{C} \equiv^t \mathbf{D}]} \mathcal{G}(EU\text{ni}f_2)$                            |   |   |
| $\frac{\Gamma, [\mathbf{A}^1 \equiv \mathbf{B}^1] \quad \dots \quad \Gamma, [\mathbf{A}^n \equiv \mathbf{B}^n]}{\Gamma, [H \overline{\mathbf{A}^n}], \neg[H \overline{\mathbf{B}^n}]} \mathcal{G}(Init^{\bar{\cdot}})$ |   | $\frac{\Gamma, [\mathbf{A}^1 \equiv \mathbf{B}^1] \quad \dots \quad \Gamma, [\mathbf{A}^n \equiv \mathbf{B}^n]}{\Gamma, [[H \overline{\mathbf{A}^n}] \equiv^t [H \overline{\mathbf{B}^n}]]} \mathcal{G}(Dec)$ |   |

**Fig. 1.** Sequent Rules for  $\mathcal{G}_{\beta\text{fb}}^{\mathcal{S}}$

The rules  $\mathcal{G}(\sharp)$  and  $\mathcal{G}(\neg\sharp)$  provides the connection between the logical constants in  $\mathcal{S}$  and the level of propositions using the operation taking a term  $\mathbf{A} \in \text{wff}_o(\mathcal{S})$  to a proposition  $\mathbf{A}^\sharp$ . The definition of  $\mathbf{A}^\sharp$  depends on the head of the term  $\mathbf{A}$ . In particular,  $\top^\sharp$  is  $\top$ ,  $[\neg\mathbf{B}]^\sharp$  is  $\neg\mathbf{B}$ ,  $[\mathbf{B} \vee \mathbf{C}]^\sharp$  is  $[\mathbf{B} \vee \mathbf{C}]$ ,  $[\mathbf{D} \equiv^\alpha \mathbf{E}]^\sharp$  is  $[\mathbf{D} \equiv^\alpha \mathbf{E}]$  and  $[I^\alpha \mathbf{F}]^\sharp$  is  $[\forall x_\alpha. \mathbf{F} x]$ . For other logical constants, we rely on

the usual methods of representing different operations in terms of  $\top$ ,  $\vee$  and  $\forall$ . For example,  $\perp^\#$  is  $\neg\top$  and  $[\mathbf{B} \supset \mathbf{C}]^\#$  is  $[\neg\mathbf{B} \vee \mathbf{C}]$ . If the head of  $\mathbf{A}$  is not a logical constant, then  $\mathbf{A}^\#$  is simply defined to be  $\mathbf{A}$ . (Note that the operation is not recursive. For example,  $[\neg\neg\mathbf{B}]^\#$  is  $\neg\neg\mathbf{B}$  as opposed to  $\neg\neg\mathbf{B}$ .)

In [11], a model class  $\mathfrak{M}_{\beta\text{fb}}(\mathcal{S})$  is defined for the  $\mathcal{S}$ -fragment of extensional type theory. The sequent calculus  $\mathcal{G}_{\beta\text{fb}}^{\mathcal{S}}$  is proven sound and complete with respect to  $\mathfrak{M}_{\beta\text{fb}}(\mathcal{S})$  (see Theorems 3.4.14 and 5.7.18 of [11]). As a consequence of completeness, we can prove the surjective form of Cantor’s “theorem” (1) cannot be proven in the  $\emptyset$ -fragment of extensional type theory. (The intuitive reason for this is the need to use  $\neg$  in the definition of the diagonal set.) In fact, one cannot prove (1) in the  $\mathcal{S}$ -fragment of extensional type theory even if  $\mathcal{S}$  is  $\{\top, \perp, \wedge, \vee\}$  (see Corollary 6.7.9 in [11]). There is a concrete model in  $\mathfrak{M}_{\beta\text{fb}}(\{\top, \perp, \wedge, \vee\})$  in which (1) is false.

We do not include a cut rule in  $\mathcal{G}_{\beta\text{fb}}^{\mathcal{S}}$  since we are interested in representing (and searching for) cut-free proofs. We know cut is admissible in  $\mathcal{G}_{\beta\text{fb}}^{\mathcal{S}}$  as a consequence of completeness (see Corollary 5.7.19 of [11]).

We do not include an initial rule for deriving general sequents of the form  $\Gamma, \neg\mathbf{M}, \mathbf{M}$  or a reflexivity rule for deriving general sequents of the form  $\Gamma, [\mathbf{A} \equiv \mathbf{A}]$ . The sequent calculus is complete without such rules (essentially because one can reduce to special cases of  $\mathcal{G}(\text{Init}^\equiv)$  and  $\mathcal{G}(\text{Dec})$ ). The reason we do not include a general initial rule or a general reflexivity rule is to avoid needing to perform arbitrary  $\beta\eta$ -unification when proving lifting results to show completeness of automated search.

While we do include a contraction rule in  $\mathcal{G}_{\beta\text{fb}}^{\mathcal{S}}$ , there are only a limited number of situations in which contraction is necessary to obtain a proof. For example, the contraction rule is often used along with the rules  $\mathcal{G}(\neg\forall, \mathcal{S})$  and  $\mathcal{G}(\neg \equiv \rightarrow, \mathcal{S})$  in order to allow multiple instantiations. In the sequent calculus  $\mathcal{G}_{\beta\text{fb}}^{\mathcal{S}}$ , one may also need to make multiple uses of formulas in instances of the rules  $\mathcal{G}(\text{Init}^\equiv)$ ,  $\mathcal{G}(\text{EUnif}_1)$  and  $\mathcal{G}(\text{EUnif}_2)$ . Finally, we may need to use contraction to provide a copy of an equation for an instances of  $\mathcal{G}(\text{Dec})$  as well as an instance of either  $\mathcal{G}(\text{EUnif}_1)$  or  $\mathcal{G}(\text{EUnif}_2)$ . The next example demonstrates different essential applications of contraction in conjunction with the other rules of  $\mathcal{G}_{\beta\text{fb}}^{\mathcal{S}}$ .

For purposes of illustration, the main example we will consider in this paper is THM615:

$$H_{\text{io}}[H \top =^t H \perp] \equiv^t H \perp$$

This is a sentence in  $\text{prop}(\mathcal{S})$  if we assume  $\top, \perp, =^t \in \mathcal{S}$ . There is a reasonably simple proof of THM615. Either  $[H \top =^t H \perp]$  is true or false. If true, then THM615 is equivalent to  $[H \top =^t H \perp]$  which we have assumed true. If false, then THM615 is equivalent to  $[H \perp =^t H \perp]$  which is true by reflexivity.

This short proof of THM615 can be formalized using of the cut rule with cut formula  $[H \top =^t H \perp]$ . Since cut is admissible, there must be a derivation of THM615 without using cut. The only possible rules of  $\mathcal{G}_{\beta\text{fb}}^{\mathcal{S}}$  which can be used to conclude THM615 are contraction or the decomposition rule  $\mathcal{G}(\text{Dec})$ . Since  $[H \top =^t H \perp] \equiv^o \perp$  is not a theorem, one cannot expect a derivation of THM615 to end with the decomposition rule. Instead, one can derive the

sequent

$$[H_{\iota o}[H \top =^{\iota} H \perp] \doteq^{\iota} H \perp], [H \top =^{\iota} H \perp] \doteq^o \perp \quad (2)$$

and complete the derivation of THM615 using decomposition followed by contraction.

The rule  $\mathcal{G}(\mathfrak{b})$  (for Boolean extensionality) reduces deriving (2) to deriving (3) and (4):

$$[H_{\iota o}[H \top =^{\iota} H \perp] \doteq^{\iota} H \perp], \neg[H \top =^{\iota} H \perp], \perp \quad (3)$$

$$[H_{\iota o}[H \top =^{\iota} H \perp] \doteq^{\iota} H \perp], \neg\perp, [H \top =^{\iota} H \perp]. \quad (4)$$

The sequent (4) is derived using  $\mathcal{G}(\neg\#)$ ,  $\mathcal{G}(\neg\neg)$  and  $\mathcal{G}(\top)$  (since  $\neg\perp\#$  is  $\neg\neg\top$ ).

Nontrivial equality reasoning is required to derive (3). One can conclude (3) using  $\mathcal{G}(EUnif_1)$  and contraction from (5) and (6):

$$[[H \top] \doteq [H.[H \perp] = [H \perp]]], \neg[H \top \doteq H \perp], \perp \quad (5)$$

$$[[H \perp] \doteq [H \perp]], \neg[H \top \doteq H \perp], \perp \quad (6)$$

The sequent (6) contains an instance of reflexivity and has an easy derivation (with several steps since there is no general rule for reflexivity). Using  $\mathcal{G}(Dec)$  and  $\mathcal{G}(\mathfrak{b})$ , (5) can be reduced to deriving (7) and (8):

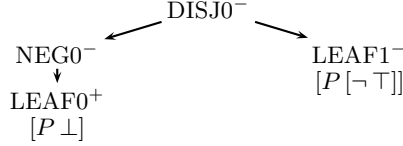
$$\neg\top, [[H \perp] = [H \perp]], \neg[H \top \doteq H \perp], \perp \quad (7)$$

$$\top, \neg[[H \perp] = [H \perp]], \neg[H \top \doteq H \perp], \perp \quad (8)$$

Both (7) and (8) are straightforward to derive.

## 4 Extensional Expansion Dags

While one can formulate automated search based on a sequent calculus, it is more common to choose a proof representation which eliminates certain redundancies. Expansion proofs provide a compact representation of cut-free proofs in elementary type theory. In particular, an expansion proof contains the essential information regarding instantiations and which atoms are used in initial sequents without recording all the information about the order of sequent rule applications. In [15] and [16] Dale Miller defined expansion proofs consisting of expansion trees with an acyclic dependence relation and a complete mating. A complete mating [3] is a set of connections which spans every vertical path. The notion of an expansion proof has been generalized in [11] to extensional expansion proofs. An extensional expansion proof is an extensional expansion dag (instead of an expansion tree) with an acyclic dependence relation and no



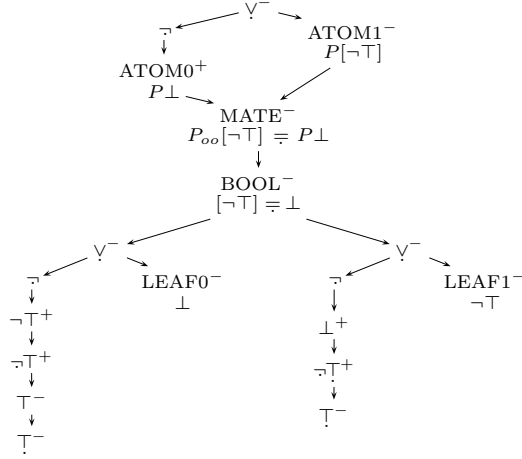
**Fig. 2.** Expansion Tree

*unsolved parts* (instead of no unspanned vertical paths). A formal definition of extensional expansion proofs can be found in [11] (see Definition 7.4.18). Here we describe the different elements of extensional expansion proofs by considering examples and noting the relationship to the sequent calculus  $\mathcal{G}_{\beta\text{fb}}^S$ .

Consider the sentence  $\neg[P_{oo} \perp_o] \vee [P [\neg \top]]$ . An expansion tree for this formula is shown in Figure 2.

We could try to connect LEAF0 to LEAF1, but we will never be able to syntactically unify  $[P \perp]$  and  $[P [\neg \top]]$ . Extensionally, we can prove these are equal since the arguments  $\perp$  and  $\neg \top$  are equivalent, hence equal by Boolean extensionality. An extensional expansion dag which would provide a proof of the sentence is shown in Figure 3. We make the connection between ATOM0 and ATOM1 explicit via the connection node MATE. The node MATE corresponds to the goal of showing  $[P \perp]$  and  $[P [\neg \top]]$  are equal. Whenever we create a mate node by connecting two atoms, the two atoms must have the same parameter in the head position. The children of the mate node correspond to showing the arguments of the two atoms are equal. The process of connecting two atoms by creating a mate node and forming the children of the new mate node corresponds to an application of the  $\mathcal{G}(\text{Init}^=)$  rule in the sequent calculus  $\mathcal{G}_{\beta\text{fb}}^S$ . In Figure 3, the node BOOL corresponds to showing  $\perp$  and  $[\neg \top]$  are equal (and is related to the  $\mathcal{G}(\text{b})$  sequent rule). One child of BOOL corresponds to showing  $[\neg \top]$  implies  $\perp$  (which uses the positive node for  $[\neg \top]$ , but not the negative node for  $\perp$ ). The other child of BOOL corresponds to showing  $\perp$  implies  $[\neg \top]$  (which uses the positive node for  $\perp$ , but not the negative node for  $[\neg \top]$ ). Many of the nodes below BOOL simply pass from logical constants in terms to the level of propositions. Such nodes correspond to applications of the sequent rules  $\mathcal{G}(\#)$  and  $\mathcal{G}(\neg\#)$ .

In Figure 4 we show an extensional expansion proof of THM615, omitting a few minor portions. (The nodes labelled by  $\supset$  actually stand for a  $\vee$  node and a  $\neg$  node.) We can directly compare this extensional expansion proof to the sequent calculus derivation described in the previous section. Note that the root node EQN0 is a negative equation node with two children, DEC0 and EQNGOAL0, both of which correspond to the same negative equation. EQN0 behaves like an application of contraction giving two copies of the equation. The child DEC0 is a decomposition node which will correspond to the principal formula in applications of the rule  $\mathcal{G}(\text{Dec})$ . The child EQNGOAL0 is an equation goal node which will correspond to a principal formula in applications of the rule  $\mathcal{G}(\text{EUnif}_1)$  and



**Fig. 3.** Extensional Expansion Dag

$\mathcal{G}(EUnif_2)$ . We obtained the subgoal sequent (2) by applying  $\mathcal{G}(Contr)$  and  $\mathcal{G}(Dec)$ . This corresponds to passing from the root node EQN0 to the children DEC0 and EQNGOAL0 and then to the child BOOL0 of DEC0. Hence the sequent (2) corresponds to the set  $\{EQNGOAL0, BOOL0\}$ . Each sequent  $\Gamma$  which occurs in the sequent derivation of THM615 corresponds to a set of nodes in the extensional expansion dag:

| sequent | set of nodes                          |
|---------|---------------------------------------|
| (3)     | $\{EQNGOAL0, EQN0, \perp^-\}$         |
| (5)     | $\{EQN2, EQN1, \perp^-\}$             |
| (6)     | $\{EQN4, EQN1, \perp^-\}$             |
| (7)     | $\{\top^+, EQN1, EQNGOAL3, \perp^-\}$ |

This correspondence can be used to translate between sequent derivations and extensional expansion proofs (see Theorems 7.9.1 and 7.10.12 in [11]).

## 5 Search Operations

While searching for an extensional expansion proof, we manipulate extensional expansion dags containing (free) expansion variables. Some search operations extend the structure by adding information (e.g., nodes or edges). Other search operations partially instantiating variables using projection and imitation terms (as in Huet's pre-unification algorithm). One search operation (*Flex-Rigid Mate*) instantiates a variable (using an imitation term) and adds a connection.

First we consider a search operation which can always be applied eagerly without sacrificing completeness. We refer to this operation as development.

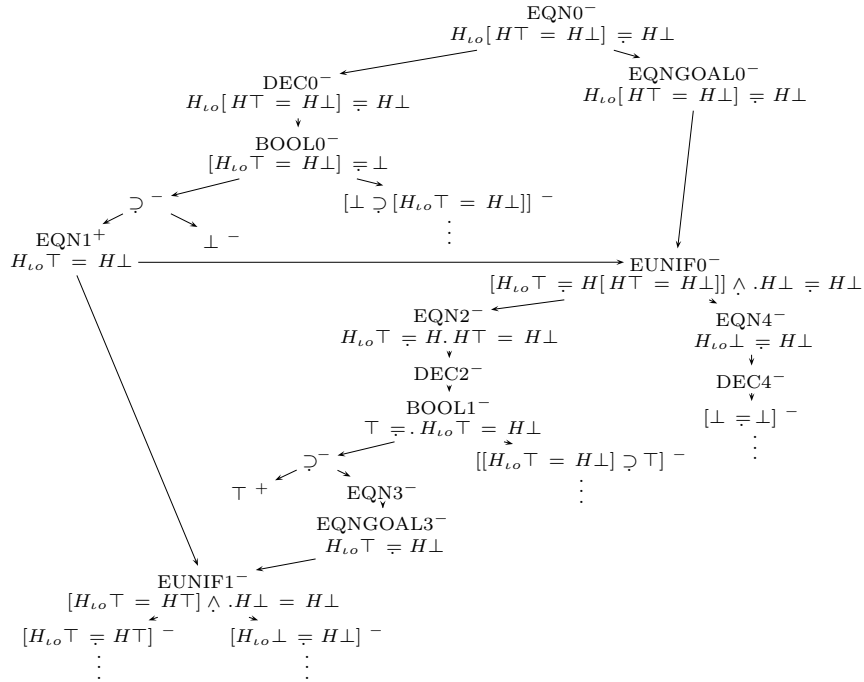


Fig. 4. Extensional Expansion Proof for THM615

- **Development** The operation is technically defined by a large number of cases. We only provide a few example cases. If  $a$  is a leaf node with shallow formula  $[\neg\mathbf{M}]$  and polarity  $p$ , then add a new successor node  $b$  to  $a$  with shallow formula  $\mathbf{M}$  and polarity  $-p$ . If  $a$  is a positive leaf node with shallow formula  $[\forall x_\alpha \mathbf{M}]$ , then let  $y_\alpha$  be a new variable and add a new positive node  $b$  with shallow formula  $[y/x]\mathbf{M}$  and add a new expansion arc from  $a$  to  $b$  labelled with a new variable  $y_\alpha$ .

The remaining operations are relative to a given set of nodes. A set  $\mathfrak{p}$  of nodes is called an *unsolved part* (or *u-part*) of an extensional expansion dag if the set satisfies certain closure conditions (see Definition 7.4.4 in [11]). For example, we require every child of a conjunctive node in  $\mathfrak{p}$  (e.g., a negative  $\forall$  node) to be in  $\mathfrak{p}$  and we require that every disjunctive node in  $\mathfrak{p}$  has some child in  $\mathfrak{p}$ . These conditions are analogous to conditions defining the vertical paths of a matrix representation of a formula. We also require a condition for  $\mathfrak{p}$  arising from the existence of connection nodes: If  $a$  and  $b$  are connected by a node  $c$  and  $a, b \in \mathfrak{p}$ , then  $c \in \mathfrak{p}$ .

Given any expansion node in a u-part, we can increase its multiplicity.

- **Duplication** Suppose  $\mathfrak{p}$  is a u-part and  $e \in \mathfrak{p}$  is a positive expansion node with shallow formula  $[\forall x_\alpha \mathbf{M}]$ . A *Duplication step* creates a new child of  $e$  with shallow formula  $[y_\alpha/x]\mathbf{M}$  where  $y_\alpha$  is a new (free) expansion variable.

Three operations add connections between two nodes in a u-part.

- **Rigid-Rigid Mate** Suppose  $\mathfrak{p}$  is a u-part,  $a \in \mathfrak{p}$  is a positive atomic node with shallow formula  $[P \overline{\mathbf{A}^n}]$  and  $b \in \mathfrak{p}$  is a negative atomic node with shallow formula  $[P \overline{\mathbf{B}^n}]$  where  $P$  is a parameter. A *Rigid-Rigid Mate step* creates a new mate node with shallow formula  $[[P \overline{\mathbf{A}^n}] \overset{\circ}{=} [P \overline{\mathbf{B}^n}]]$  beneath  $a$  and  $b$ .
- **E-Unification** Suppose  $\mathfrak{p}$  is a u-part,  $a \in \mathfrak{p}$  is a positive equation node with shallow formula  $[\mathbf{A} \overset{t}{=} \mathbf{B}]$  and  $b \in \mathfrak{p}$  is a negative equation goal node with shallow formula  $[\mathbf{C} \overset{t}{=} \mathbf{D}]$ . An *E-unification step* creates a new E-unification node with shallow formula  $[[\mathbf{A} \overset{t}{=} \mathbf{C}] \wedge [\mathbf{B} \overset{t}{=} \mathbf{D}]]^1$  beneath  $a$  and  $b$ .
- **Symmetric E-Unification** This is analogous to the E-unification step except the new node has shallow formula  $[[\mathbf{A} \overset{t}{=} \mathbf{D}] \wedge [\mathbf{B} \overset{t}{=} \mathbf{C}]]$ .

We next consider two operations which instantiate an expansion variable. The first operation corresponds to pre-unification and the second operation corresponds to applying primsubs.

- **Flex-Rigid** Suppose  $\mathfrak{p}$  is a u-part and  $a \in \mathfrak{p}$  is a negative equation node with shallow formula  $[x \overline{\mathbf{A}^n} \overset{t}{=} F \overline{\mathbf{B}^m}]$  (or  $[F \overline{\mathbf{B}^m} \overset{t}{=} x \overline{\mathbf{A}^n}]$ ) where  $x$  is a variable and  $F$  is a parameter. A *Flex-Rigid step* instantiates  $x$  by imitating  $F$  or by projecting onto any argument of  $x$  with an appropriate type.
- **Primitive Substitution** Suppose  $\mathfrak{p}$  is a u-part and  $a \in \mathfrak{p}$  is a positive flexible node with shallow formula  $[p \overline{\mathbf{A}^n}]$  where  $p$  is a (set) variable. A *Primitive Substitution step* instantiates  $p$  by imitating any logical constant in  $\mathcal{S}$  or projecting onto any argument of  $p$  with an appropriate type.

Finally, we consider an operation which mates a positive flexible node with a negative atomic node.

- **Flex-Rigid Mate** Suppose  $\mathfrak{p}$  is a u-part,  $a \in \mathfrak{p}$  is a positive flexible node with shallow formula  $[p \overline{\mathbf{A}^n}]$  and  $b \in \mathfrak{p}$  is a negative atomic node with shallow formula  $[Q \overline{\mathbf{B}^m}]$  where  $p$  is a (set) variable and  $Q$  is a parameter. A *Flex-Rigid Mate step* instantiates  $p$  by imitating  $Q$  and creates a new mate node beneath  $a$  and  $b$ .

A search procedure proceeds by applying the search operations above (starting with an initial extensional expansion dag) until one obtains a pre-solved extensional expansion dag (i.e., an extensional expansion dag such that every u-part is pre-solved).

We have not introduced search operations which operate on negative flex-flex equation nodes or that connect two flexible nodes. The fact that such (highly branching) rules are not necessary for completeness is significant. The fact that flex-flex pairs (and flex-flex connections) can always be delayed corresponds to the fact that flex-flex pairs are always delayed in higher-order pre-unification [14]. In higher-order pre-unification, if one reaches a set of disagreement pairs such that every pair is flex-flex, then the set is *pre-unified* and there exist many (easy

---

<sup>1</sup> Technically, this proposition is  $\neg[\neg[\mathbf{A} \overset{t}{=} \mathbf{C}] \vee \neg[\mathbf{B} \overset{t}{=} \mathbf{D}]]$

to construct) solutions to the unification problem. In the context of extensional expansion dags, we say a u-part  $\mathfrak{p}$  is *pre-solved* if there is either a negative flex-flex equation node in  $\mathfrak{p}$  or a negative flexible node in  $\mathfrak{p}$ . Extensional expansion proofs are required to contain no u-parts. We can weaken this condition to say every unsolved part is pre-solved. Given an appropriate extensional expansion dag such that every unsolved part is pre-solved, one can easily construct an extensional expansion proof (see Theorem 8.3.3 in [11]). The idea for solving u-parts containing flex-flex pairs is the same as in higher-order pre-unification: one imitates a new parameter  $C_i$  of base type. The idea for solving u-parts containing negative flexible nodes is analogous: one imitates the logical constant  $\top_o$ . Using this idea, one can solve any collection of pre-solved u-parts simultaneously.

Among the operations presented above, the least directed are primitive substitution steps. A primitive substitution can involve imitating any logical constant in  $\mathcal{S}$  whenever a u-path contains a positive flexible node. In general, the set  $\mathcal{S}$  is infinite (e.g., if  $\Pi^\alpha \in \mathcal{S}$  for every type  $\alpha$ ). In an effort to minimize the problem of primitive substitutions, one can consider smaller signatures of logical constants. Some restrictions on the signature  $\mathcal{S}$  do not result in any essential incompleteness. For example, assuming  $\{\neg, \vee\} \subseteq \mathcal{S}$ , there is no need to include  $\wedge, \supset$  or  $\equiv$  in the signature  $\mathcal{S}$ . In particular, performing primitive substitutions for  $\wedge$  in addition to primitive substitutions for  $\vee$  and  $\neg$  does not increase the theorems one can prove. Furthermore, assuming  $\{\neg, \vee\} \subseteq \mathcal{S}$ , there is no need to include  $\Pi^\alpha, \Sigma^\alpha$  or  $=^\alpha$  in  $\mathcal{S}$  for propositional types  $\alpha$  (i.e., types  $\alpha$  which are constructed solely from type  $o$ ). If one is concerned only with the theory of propositional types (as in [13; 1]), then one can reduce the signature (and hence primitive substitutions) to  $\{\neg, \vee\}$  (or other such minimal signatures) without sacrificing completeness. This fact does not hold if one does not assume extensionality. Consider the simple proposition involving only propositional types:

$$\exists s_{o(oo)} \forall f_{oo}. s f \equiv \exists p_o f p \tag{9}$$

Essentially, (9) expresses the existence of the  $\Sigma^o$  (existence) operator for type  $o$ . Assuming extensionality, the witness  $[\lambda f_{oo}. f A_o \vee [f \neg A]]$  (expressed using only  $\vee, \neg$  and an arbitrary parameter  $A_o$ ) can be used to prove the theorem. On the other hand, there is a non-extensional model which includes interpretations for  $\neg$  and  $\vee$  in which (9) is false.

One can also consider other restrictions (such as eliminating all logical constants by choosing  $\mathcal{S}$  to be empty) which do rule out the possibility of proving certain theorems of extensional type theory (e.g., the surjective form of Cantor's theorem). On the other hand,  $\mathfrak{M}_{\beta\text{fb}}(\emptyset)$  provides a model theory for the  $\emptyset$ -fragment of extensional type theory,  $\mathcal{G}_{\beta\text{fb}}^\emptyset$  provides a cut-free proof theory and the search operations above provide for automated search without primitive substitutions (except projections).

## 6 Completeness of Search

In order to prove completeness of search, we assume there is a ground extensional expansion proof  $Q^*$  of a sentence  $M$ . Given  $Q^*$  we can define the notion of a

*lifting map*  $f = \langle \varphi, f \rangle$  from an extensional expansion dag  $\mathcal{Q}$  (with variables) to  $\mathcal{Q}^*$  (where  $\varphi$  is a ground substitution and  $f$  maps the nodes of  $\mathcal{Q}$  to the nodes of  $\mathcal{Q}^*$ ). We can also define a measure  $\|f : \mathcal{Q} \rightarrow \mathcal{Q}^*\| < \omega^3$  for each such lifting map (see Definitions 8.4.3 and 8.4.4 in [11]). Using  $\mathcal{Q}^*$  we can guide the application of search operations to  $\mathcal{Q}$  and use the lifting map  $f$  to maintain the relationship between the current extensional expansion dag  $\mathcal{Q}$  and the known proof  $\mathcal{Q}^*$ .

A key step for proving completeness is to verify one can always make progress. That is, given any ground extensional expansion proof  $\mathcal{Q}^*$ , extensional expansion dag  $\mathcal{Q}$  (where  $\mathcal{Q}$  is not already pre-solved) and lifting map  $f$  from  $\mathcal{Q}$  to  $\mathcal{Q}^*$ , there must be some search operation one can apply to  $\mathcal{Q}$  to obtain  $\mathcal{Q}'$  and some lifting map  $f'$  from  $\mathcal{Q}'$  to  $\mathcal{Q}^*$  such that  $\|f' : \mathcal{Q}' \rightarrow \mathcal{Q}^*\| < \|f : \mathcal{Q} \rightarrow \mathcal{Q}^*\|$ . In other words, there must be some search operation which results in an extensional expansion dag *closer* to the goal proof  $\mathcal{Q}^*$ .

Once one knows progress can always be made, one can argue completeness of search as follows: There is a trivial lifting map from an initial extensional expansion dag (with one node corresponding to  $\mathbf{M}$ ) to a ground extensional expansion proof  $\mathcal{Q}^*$  of  $\mathbf{M}$ . Using the progress property, one can appropriately choose search operations until one obtains a pre-solved extensional expansion dag. Since  $\|f : \mathcal{Q} \rightarrow \mathcal{Q}^*\|$  is an ordinal, the process will terminate in a pre-solved extensional expansion dag after a finite number of appropriate choices. From a pre-solved extensional expansion dag, one can obtain a ground extensional expansion proof (which may actually be different from  $\mathcal{Q}^*$ ).

This completeness argument is further refined as in [11]. First, one can assume that no development step can be applied to the given extensional expansion proof  $\mathcal{Q}^*$ . Using this assumption, one can show that given any extensional expansion dag  $\mathcal{Q}$  and lifting map  $f : \mathcal{Q} \rightarrow \mathcal{Q}^*$ , if any development step applies to  $\mathcal{Q}$ , then this development step results in an extensional expansion dag  $\mathcal{Q}'$  and lifting map  $f' : \mathcal{Q}' \rightarrow \mathcal{Q}^*$  such that  $\|f' : \mathcal{Q}' \rightarrow \mathcal{Q}^*\| < \|f : \mathcal{Q} \rightarrow \mathcal{Q}^*\|$  (see Lemma 8.6.17 in [11]). Consequently, development steps are a form of *don't-care* non-determinism.

Assuming no development step can be applied to  $\mathcal{Q}$ , then one can show a progress property relative to u-parts (which are not pre-solved). In particular, for any u-part  $\mathfrak{p}$  either  $\mathfrak{p}$  is pre-solved or there is a search operation (other than development) relative to  $\mathfrak{p}$  which results in an extensional expansion dag  $\mathcal{Q}'$  with a lifting map  $f' : \mathcal{Q}' \rightarrow \mathcal{Q}^*$  such that  $\|f' : \mathcal{Q}' \rightarrow \mathcal{Q}^*\| < \|f : \mathcal{Q} \rightarrow \mathcal{Q}^*\|$  (see Lemma 8.8.4 in [11]). Consequently, choosing the (not pre-solved) u-part  $\mathfrak{p}$  is also a form of *don't-care* non-determinism while choosing the operation relative to  $\mathfrak{p}$  is a form of *don't-know* non-determinism. We can always restrict our attention to u-parts which are not pre-solved since search ends precisely when every u-part is pre-solved.

Finally, instead of simply maintaining a current extensional expansion dag  $\mathcal{Q}$  and a search lifting map  $f$ , one can also represent information regarding quantifier duplications and set variables. Using this information one can impose certain conditions on the order of search operations. In particular, one can perform quantifier duplications (on a single expansion node), then primitive substitutions (for certain set variables) and lastly create connections and perform unification. (If

new expansion nodes or set variables are created during search, then new duplications or primitive substitutions for these new objects are allowed.) Completeness of this ordered form of search follows from Theorem 8.8.6 of [11].

## 7 Examples

The MS04-2 search procedure implemented in TPS uses a bounded best-first search strategy with iterative deepening. Under the basic flag settings, MS04-2 only considers u-parts which are not presolved and only considers the options described in Section 5. Flags provide weights for ordering different options.

Using MS04-2, TPS can prove THM615 automatically in less than a second. Other examples TPS can prove automatically in less than a second using MS04-2 include  $\mathbf{E}_1^{ext}$ ,  $\mathbf{E}_2^{ext}$  and  $\mathbf{E}_3^{ext}$  from [7].

The example  $\mathbf{E}^{Dec}$  in [7] motivates a special decomposition rule in LEO for functions. The formulation in [7] is with respect to Leibniz equality:

$$\forall X_{\alpha\alpha} \forall Y_{\alpha} [f_{\alpha\alpha(\alpha\alpha)} X Y \doteq^{\alpha} g_{\alpha\alpha(\alpha\alpha)} X Y] \wedge \forall Z_{\alpha} [h_{\alpha\alpha} Z \doteq^{\alpha} j_{\alpha\alpha} Z] \supset f h \doteq^{\alpha\alpha} g j$$

In this proposition, we use the notation  $[\mathbf{A} \doteq^{\alpha} \mathbf{B}]$  to denote Leibniz equality of  $\mathbf{A}$  and  $\mathbf{B}$  at the level of propositions:  $[\forall q_{\alpha\alpha} . [q \mathbf{A}] \supset [q \mathbf{B}]]$ . We can form an example EDEC2 by replacing all instances of Leibniz equality with primitive equality: The proof of  $\mathbf{E}^{Dec}$  (using Leibniz equality) requires two primitive substitutions using  $=^{\iota}$  followed by unification steps including 8 imitations and 4 projections. Even when flag settings are optimized for this example (while still only allowing basic search operations), TPS takes over 2 hours to find the proof. On the other hand, EDEC2 requires no primitive substitutions and very little unification (3 imitation steps and 1 projection step) and can be proven in less than a second.

Our final example is a theorem of topology we call THM616:

$$\begin{aligned} & \forall G_{o(o\iota)} [G \subseteq OPEN_{o(o\iota)} \supset OPEN [\bigcup G]] \\ & \supset \forall B_{o\iota} . \forall x_{\iota} [B x \supset \exists D . OPEN D \wedge D x \wedge D \subseteq B] \supset OPEN B \end{aligned}$$

where  $\bigcup$  abbreviates  $\lambda G_{o(o\alpha)} \lambda x_{\alpha} \exists S_{o\alpha} . G S \wedge S x$ . THM616 is a modified version of BLEDSOE-FENG-SV-10 (studied in [9; 10]) where closure of  $OPEN$  under unions is stated in a more natural manner. However, THM616 is only a theorem if extensionality is assumed (whereas BLEDSOE-FENG-SV-10 is a theorem of elementary type theory). Consequently, no previous TPS search procedure could possibly prove THM616. By combining the extensionality reasoning described here with the set constraint reasoning described in [10], TPS can prove THM616 in about 10 seconds.

## 8 Conclusion

We have described a complete theorem proving procedure for extensional type theory with primitive equality. Furthermore, the theorem proving techniques

provide a basis for reasoning in various fragments of extensional type theory in which one varies the logical constants available. The study of such fragments allows one to measure the primitive substitutions necessary to find an automated proof of a theorem.

## References

1. Peter B. Andrews. A reduction of the axioms for the theory of propositional types. *Fundamenta Mathematicae*, 52:345–350, 1963.
2. Peter B. Andrews. Resolution in type theory. *Journal of Symbolic Logic*, 36:414–432, 1971.
3. Peter B. Andrews. On connections and higher-order logic. *Journal of Automated Reasoning*, 5:257–291, 1989.
4. Peter B. Andrews. Classical type theory. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume 2, chapter 15, pages 965–1007. Elsevier Science, 2001.
5. Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Kluwer Academic Publishers, second edition, 2002.
6. Peter B. Andrews, Matthew Bishop, Sunil Issar, Dan Nesmith, Frank Pfenning, and Hongwei Xi. TPS: A theorem proving system for classical type theory. *Journal of Automated Reasoning*, 16:321–353, 1996.
7. Christoph Benzmüller. *Equality and Extensionality in Automated Higher-Order Theorem Proving*. PhD thesis, Universität des Saarlandes, 1999.
8. Christoph Benzmüller and Michael Kohlhase. System description: LEO — a higher-order theorem prover. In Claude Kirchner and Hélène Kirchner, editors, *Proceedings of the 15th International Conference on Automated Deduction*, volume 1421 of *Lecture Notes in Artificial Intelligence*, pages 139–143, Lindau, Germany, 1998. Springer-Verlag.
9. W. W. Bledsoe and Guohui Feng. Set-Var. *Journal of Automated Reasoning*, 11:293–314, 1993.
10. Chad E. Brown. Solving for set variables in higher-order theorem proving. In Andrei Voronkov, editor, *Proceedings of the 18th International Conference on Automated Deduction*, volume 2392 of *Lecture Notes in Artificial Intelligence*, pages 408–422, Copenhagen, Denmark, 2002. Springer-Verlag.
11. Chad E. Brown. *Set Comprehension in Church’s Type Theory*. PhD thesis, Department of Mathematical Sciences, Carnegie Mellon University, 2004.
12. Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.
13. Leon Henkin. A theory of propositional types. *Fundamenta Mathematicae*, 52:323–344, 1963.
14. Gérard P. Huet. A unification algorithm for typed  $\lambda$ -calculus. *Theoretical Computer Science*, 1:27–57, 1975.
15. Dale A. Miller. *Proofs in Higher-Order Logic*. PhD thesis, Carnegie Mellon University, 1983. 81 pp.
16. Dale A. Miller. A compact representation of proofs. *Studia Logica*, 46(4):347–370, 1987.