Encoding a Dependent-Type λ-Calculus in a Logic Programming Language

Amy Felty
INRIA Sophia-Antipolis
2004, Route des Lucioles
06565 Valbonne Cedex, France

Dale Miller
Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104–6389 USA

Abstract

Various forms of typed λ-calculi have been proposed as specification languages for representing wide varieties of object logics. The logical framework, LF, is an example of such a dependent-type λ-calculus. A small subset of intuitionistic logic with quantification over simply typed λ-calculus has also been proposed as a framework for specifying general logics. The logic of hereditary Harrop formulas with quantification at all non-predicate types, denoted here as \( hh^\omega \), is such a meta-logic that has been implemented in both the Isabelle theorem prover and the λProlog logic programming language. Both frameworks provide for specifications of logics in which details involved with free and bound variable occurrences, substitutions, eigenvariables, and the scope of assumptions within object logics are handled correctly and elegantly at the "meta" level. In this paper, we show how LF can be encoded into \( hh^\omega \) in a direct and natural way by mapping the typing judgments in LF into propositions in the logic of \( hh^\omega \). This translation establishes a very strong connection between these two languages: the order of quantification in an LF signature is exactly the order of a set of \( hh^\omega \) clauses, and the proofs in one system correspond directly to proofs in the other system. Relating these two languages makes it possible to provide implementations of proof checkers and theorem provers for logics specified in LF by using standard logic programming techniques which can be used to implement \( hh^\omega \).

1 Introduction

The design and construction of computer systems that can be used to specify and implement large collections of logics has been the goal of several different research projects. In this paper we shall focus on two approaches to designing such systems. One approach is based on the use of dependent-type λ-calculi as a meta-language while another approach is based on the use of a very simple intuitionistic logic as a meta-language. The Logical Framework (LF) [HHP87] and the Calculus of Constructions (CC) [CH88] are two examples of dependent-type calculi that have been proposed as meta-logics. The Isabelle
Theorem prover [Pau89] and the λProlog logic programming language [NM88] provide implementations of a common subset of intuitionistic logic, called \( \text{hh} \omega \) here, that can be used to specify a wide range of logics. Both Isabelle and λProlog can turn specifications of logics into proof checkers and theorem provers by making use of the unification of simply typed λ-terms and goal-directed, tactic-style search.

In this paper, we shall show that these two meta-languages are essentially of the same expressive power. This is done by showing how to translate LF specifications and judgments into a collection of \( \text{hh} \omega \) formulas such that correct typing in LF corresponds to intuitionistic provability in \( \text{hh} \omega \). Besides answering the theoretical question about the precise relationship between these meta-languages, this translation also describes how LF specifications of an object logic can be implemented using unification and goal-directed search since these techniques provide implementations of \( \text{hh} \omega \).

In Section 2 we present the meta-logic \( \text{hh} \omega \) and in Section 3 we present LF. Section 4 presents a translation of LF into \( \text{hh} \omega \) and Section 5 contains a proof of its correctness. Section 6 provides examples of this translation and Section 7 concludes.

2 The Meta-Logic

Let \( S \) be a fixed, finite set of primitive types (also called sorts). We assume that the symbol \( o \) is always a member of \( S \). Following Church [Chu40], \( o \) is the type for propositions. The set of types is the smallest set of expressions that contains the primitive types and is closed under the construction of function types, denoted by the binary, infix symbol \( \rightarrow \). The Greek letters \( \tau \) and \( \sigma \) are used as syntactic variables ranging over types. The type constructor \( \rightarrow \) associates to the right. If \( \tau_0 \) is a primitive type then the type \( \tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau_0 \) has \( \tau_1, \ldots, \tau_n \) as argument types and \( \tau_0 \) as target type. The order of a primitive type is 0 while the order or a non-primitive type is one greater than the maximum order of its argument types.

For each type \( \tau \), we assume that there are denumerably many constants and variables of that type. Constants and variables do not overlap and if two constants (or variables) have different types, they are different constants (or variables). A signature is a finite set \( \Sigma \) of constants and variables whose types are such that their argument types do not contain \( o \). A constant with target type \( o \) is a predicate constant. We often enumerate signatures by listing their members as pairs, written \( a : \tau \), where \( a \) is a constant of type \( \tau \). Although attaching a type in this way is redundant, it makes reading signatures easier.

Simply typed λ-terms are built in the usual way. The logical constants are given the following types: \( \wedge \) (conjunction) and \( \supset \) (implication) are both of type \( o \rightarrow o \rightarrow o \); \( \top \) (true) is of type \( o \); and \( \forall_\tau \) (universal quantification) is of type \( (\tau \rightarrow o) \rightarrow o \), for all types \( \tau \) not containing \( o \). A formula is a term of type \( o \). The logical constants \( \wedge \) and \( \supset \) are written in the familiar infix form. The expression \( \forall_\tau (\lambda x t) \) is written simply as \( \forall_\tau x t \).

If \( x \) and \( t \) are terms of the same type 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. The expression \( [t_1/x_1, \ldots, t_n/x_n] \) will denote the simultaneous substitution of the terms \( t_1, \ldots, t_n \) for the variables \( x_1, \ldots, x_n \), respectively.