Axiomatization of a Functional Logic Language

Francois Bronsard*
Uday S. Reddy†

Department of Computer Science
The University of Illinois at Urbana-Champaign
Urbana, IL 61801
Net: {bronsard, reddy}@cs.uiuc.edu

Abstract

Functional logic languages incorporate logic programming capabilities of constraint solving within a functional language framework. We consider a prototypical functional logic language which supports definite descriptions, i.e., terms of the form "the x such that p". Its semantics is defined in terms of flat ordered structures with ⊥ and ⊤ elements. These elements are used to represent the absence and ambiguity, respectively, of objects denoted by descriptions. We provide an equational axiomatization of the language and show that it is complete for this semantics.

1 Introduction

Functional programming and logic programming were originally considered two disparate programming paradigms. Investigations revealed that the two paradigms have many similarities [Red86b]. However, logic programming enriches the previous programming paradigms by the distinctive notion of computation by constraint solving. It was found that this notion was not incompatible with the concepts of functional programming. Our previous work [Red85, Red86a, Red87] showed this by designing paradigmatic languages which are functional, yet incorporate the notion of solving via the operational mechanism called narrowing. Such languages were termed functional logic languages or constraint functional languages [DG89]. It was also found that such languages had close relationships to the techniques of theorem proving with equality [Fay79, Hul80, Sla74] (see also [Der83, GM84]), single assignment languages [Lin85, NPA86] as well as classical formalisms like the Hilbert's ε operator [HA50, Lei69].

In this paper, we give an axiomatic treatment of a simple, but representative, functional logic language. The language treated here incorporates what may be called definite descriptions [Pot88]. In common mathematical parlance, one finds descriptions of the forms "the x such that p". For example, the supremum of a set S in a partial order

---

* Bronsard was supported by NSERC, Canada, and FCAR, Canada
† The work by Reddy was supported by a grant from Motorola Corp.
is "the element x such that every member of S is smaller than x, and, for all elements y bounding the members of S, x is smaller than y". Similar descriptions are also useful for specifying programs. For instance, the maximum of a list can be expressed by: \( \text{maxList}(l) = \text{the } x \text{ such that } x \in l \land \forall y \in l : x \geq y \). These descriptions allow one to write high-level executable specifications for programs.

A definite description is meaningful only in the situation that there is a unique x satisfying the stated property. For theoretical purposes, we also have to deal with the situations where there is no such x. Our approach is to use ordered structures for interpretation of terms and descriptions so that a description that refers to no objects is interpreted as \( \bot \) and a description that refers to multiple objects is interpreted as \( \top \). This is the same interpretation as that of [Red87]. There are, no doubt, many other possibilities. See [Pot88] for a discussion of these issues. Other similar notations have been used in the context of program synthesis by Darlington et al. [DFP86, DG89] and Bauer et al. [Bau87].

Specifications written in a functional logic language are often unusable as programs because they involve excessive, even infinite, search. We propose that such specifications may be formally transformed to efficient programs using automated reasoning tools. We are in the process of constructing an automated program transformation assistant called FOCUS, [Red88a, Red88b, Red89], whose specification language includes the functional logic language presented here. It is of central importance that such an automated reasoning system employ a sound and complete axiomatization of the language involved. Soundness guarantees the correctness of the program derived using the system, while completeness provides the confidence that it can be used to derive every "possible" program. The notion of "every possible program" must be interpreted with caution because programs can represent undecidable theories where no completeness is possible. What we are after is a complete deductive framework for the first-order aspects of the language. This is formalized by defining a class of models as in first-order predicate calculus.

Our axiomatization of the language is based on an axiomatization of the if-then-else operator given by Bloom and Tindell [BT83] and extended by Guessarian and Meseguer [GM87]. Section 3 presents this axiomatization with a few modifications to account for the \( \top \) element in the boolean sort.

In the remaining sections, we give the axiomatizations for an operator called choice and an abstraction construct called Some. A definite description involves the choice of a value from the semantic domain satisfying a given property. The choice operator captures a finitary version of this choice, and the Some construct generalizes it to a quantifier. Technically each section can be divided into 4 steps, first an axiomatization of the operator or the construct together with a proof that the axioms respect the desired semantics; then, some more axioms to express the relationship of the construct under study with if-then-else; and, finally, a proof that the combination of theses axioms forms a complete system.

2 Preliminaries

Functions Let \( \Sigma \) be a many-sorted signature. We assume at least the following sort distinctions: Data for data values and Bool for boolean values (the case of multiple data