A Fibrational Semantics for Logic Programs

Y. Kinoshita1* and A.J. Power2**

1 Electrotechnical Laboratory, Tsukuba, 305 Japan;
  e-mail: yoshiki@etl.go.jp, fax: +81-298-58-5184
2 Department of Computer Science, University of Edinburgh, JH9 3JZ, Scotland;
  e-mail: ajp@dcs.ed.ac.uk

Abstract. We introduce a new semantics for logic programming languages. It generalises the traditional Herbrand universe semantics, and specialises the semantics of logical relations, as used in analysing parametricity in functional and imperative programming languages. We outline a typed logic programming language, give it this semantics, and show how it supports structured development of logic programs as advocated by Sterling et al. In particular, it gives semantics for some dynamic aspects of logic programs.

1 Introduction

In 1989, Power and Sterling wrote "A notion of map between logic programs" [10], in which they began a stepwise structured development of logic programs (see [7],[12] for further developments). Although their work was generally expressed syntactically, there was a new semantics for logic programs implicit in their setting: each logic program was assigned a monoid, a $V$-semilattice (modulo some isomorphisms), and an action of the monoid on the $V$-semilattice. Then a map of logic programs was defined in terms of that structure, and that was the fundamental definition upon which the technical development of the paper depended.

The semantics implicit in that paper generalised the usual semantics for logic programs, as explained for instance in Lloyd's book [8]. Moreover, if one extends from the usual single sorted account of logic programs to add the mildest of typing structure, the semantics amounts to a special case of structures developed in studying functional and imperative programs, namely the structure of fibrations, central to the theory of logical relations, see [6]. So, in this paper, we make the semantics of Power and Sterling's paper explicit. This generalises the usual semantics based on Herbrand universes; it allows us to give a semantic account of some dynamic aspects of logic programs; and it fits into the more general setting of logical relations. It also gives us freedom to consider developing a semantics for higher-order logic programming languages such as $\lambda$-PROLOG (see [9]) and

---

* The first author acknowledges the support of MITI Cooperative Architecture Project.
** The second author gratefully acknowledges the support of EPSRC grant GR/J84205 Frameworks for programming language semantics and logic.
other general logic programming languages such as those based on linear logic (see [5] for example), as well as giving semantic support for the use of logic programming languages as specification languages for conventional imperative and functional programming languages via logical relations.

In order to illustrate the structure most vividly, we adopt a mild typing structure of tuples on logic programs, allowing products. Our semantics is given by an indexed category with finite products (see Section 3.2). We have a small category $C$ with finite products, to represent the category of sorts and terms; and to each object $T$ of $C$, a category $p(T)$ of formulae of sort $T$ and proofs that one implies another; finite products are used, with appropriate coherence, to model tuples of data in $C$ and to model finite conjunctions of formulae of a sort in each $p(T)$; and the functoriality of $p(T)$ in $T$ yields substitution of a term in a formula. This association makes an indexed category, which is known to be an equivalent notion to fibration, thus, the title. Technically, fibration is easier to handle, but indexed category is easier to understand. So, we develop our theory using indexed category here.

To organize the paper, we first recall the central definitions and result of [10] in Section 2. In Section 3, we develop those pieces of category theory we will need. Then we define many sorted first order languages in Section 4. We give our semantics, several examples, and describe and prove we have a generic model for each program in Section 5. In Section 6, we explain how this all allows us to give a semantic definition of most general unifier, which in turn allows us in Section 7 to give a semantic account of computations and extensions as defined by Power and Sterling.

2 Program Maps and Extensions in Traditional Setting

In this section, we recall the central definitions and results of [10]. There is a new semantics implicit in this analysis, and the main point of this paper is to make that semantics explicit.

A base in a first order language $\mathcal{L}$ is a finite set of atomic formulae in $\mathcal{L}$. Elements of a base will be called basic formulae. A definite logic program $P$ in $\mathcal{L}$ is a quadruple $P = (V_P, F_P, B_P, C_P)$ where $V_P$ is a set of variables, $F_P$ is a set of function symbols, $B_P$ is a base in $\mathcal{L}$ with variable and function symbols only from $V_P$ and $F_P$, and $C_P$ is a finite set of definite clauses, all of whose atomic formulae are elements of $B_P$.

A $P$-substitution is a substitution that fixes all variables that do not appear in $V_P$, and all variables in $V_P$ are moved to terms that are expressible by $V_P$ and $F_P$. A $P$-instance of a basic formula is obtained by a $P$-substitution.

The definition of definite logic programs given here specifies more information than the standard definition for definite logic programs given, for example, in [8]. The sets $V_P$, $F_P$, $B_P$ are implicit in Lloyd's treatment. Typically, these sets are understood from the context.

Given a program $P$, let $\&_P$ denote the set of all finite sets of $P$-instances of elements of $B_P$. The empty set (of $P$-instances of elements of $B_P$) will be