Denotational Versus Declarative Semantics for Functional Programming *

Juan Carlos González Moreno  
*Dpto. L.S.I.I.S. (U.P.M., Spain)*

María Teresa Hortals González, Mario Rodríguez Artalejo  
*Dpto. de Informática y Automática. (U.C.M., Spain)*

Abstract.  
Denotational semantics is the usual mathematical semantics for functional programming languages. It is higher order (H.O.) in the sense that the semantic domain \( D \) includes \([D \rightarrow D]\) as a subdomain. On the other hand, the usual declarative semantics for logic programs is first order (F.O.) and given by the least Herbrand model. In this paper, we take a restricted kind of H.O. conditional rewriting systems as computational paradigm for functional programming. For these systems, we define both H.O. denotational and F.O. declarative semantics as two particular instances of algebraic semantics over continuous applicative algebras. For the declarative semantics, we prove soundness and completeness of rewriting, as well as an initiality result. We show that both soundness and completeness fail w.r.t. the denotational semantics and we present a natural restriction of rewriting that avoids unsoundness. We conjecture that this restricted rewriting is complete for computing denotationally valid F.O. results.

1 Introduction

This work stems from our interest in the integration of logic and functional programming. Though this topic has received much attention in the last years [6,2,7], most of the existing research refers to F.O. languages. The semantic foundation of H.O. logic + functional languages does still need further investigation. Existing approaches propose either to translate H.O. syntax into F.O. syntax [3,4] or to use H.O. logics with a F.O. semantics [13,5].

In our previous work [10], we used conditional narrowing for a restricted kind of H.O. rewriting systems as a computational paradigm for H.O. functional logic programming, and we established soundness and completeness results w.r.t. a F.O. declarative semantics induced by a syntactic H.O. into F.O. translation. Subsequently, we realized that the effect of the syntactic translation was equivalent to the direct definition of a F.O. least model semantics, given by an algebra over an infinitary Herbrand universe equipped with a continuous apply operation. We also observed that denotational semantics can be obtained in the same way as a least model over a different H.O. domain \( D \) which includes \([D \rightarrow D]\) as a subdomain. The aim of this paper is to show that these ideas lead to a new and interesting view of the semantics of functional languages, well suited to their integration with logic languages.

The organization of the paper is as follows. In Section 2 we develop the basic facts about applicative expressions and continuous applicative algebras (C.A.A.s), which will be the basis of our mathematical semantics. In Section 3 we define a Simple Functional Language (SFL) and explain how to use a kind of infinite H.O. rewriting as its operational semantics. In Section 4 we show that C.A.A.s and a least fixpoint technique can be used to specify both a F.O. declarative and a H.O. denotational semantics for SFL programs. We also present an initiality result for the declarative semantics. In Section 5 we develop several results about soundness and completeness of rewriting w.r.t. both semantics; they show that declarative semantics characterizes the operational behaviour of SFL programs more adequately than denotational semantics. Section 6 summarizes some conclusions and planned lines of future work.

*Research supported by the PRONTIC project TIC 89/0104*
2 Applicative Expressions and Algebras

First we introduce the basic facts about applicative expressions, algebras, expression evaluation and equations.

2.1 Signatures with constructors

A signature with constructors is any pair $\Sigma = (DC_\Sigma, FS_\Sigma)$ where $DC_\Sigma = \cup DC_\Sigma^n$ and $FS_\Sigma = \cup FS_\Sigma^n$ are ranked sets of constructors and function symbols, respectively. We assume that all the sets $DC_\Sigma^n, FS_\Sigma^n$ are mutually disjoint and use the notation $\text{rank}(\phi)$ for the rank of any symbol $\phi \in DC_\Sigma \cup FS_\Sigma$.

2.2 Expressions and patterns

Given a countably infinite set of variables $X, Y, Z \in \text{Var}$, we define the set of applicative expressions $e \in \text{Exp}_\Sigma$ of signature $\Sigma$ by the following syntax:

$$ e ::= x | c | f | (e_0 e_1) $$

An expression $(e_0 e_1)$ stands for the application of $e_0$ to $e_1$. As usual, we assume that application associates to the left and omit brackets accordingly. Note that any applicative expression can be written in exactly one of the 6 following forms:

- $X$ % variable
- $c$ % constructor
- $f$ % function symbol
- $(X e_1 \ldots e_m)$ % curried application of a variable
- $(c e_1 \ldots e_m)$ % curried application of a constructor
- $(f e_1 \ldots e_m)$ % curried application of a function symbol

A curried application of a constructor or function symbol is called partial, exact or exceeding according to the case that the number $m$ of arguments is less than, equal to or greater than the symbol’s rank. We define 2 special kinds of expressions as follows:

- **Patterns** $s, t \in \text{Ptr}_\Sigma$
  $$ t ::= X | c \in \text{Var} | f \in FS_\Sigma^n, n > 0 | (t_1 \ldots t_m) \in D, m \leq n $$

- **First order expressions** $e \in \text{FOExp}_\Sigma$
  $$ e ::= X | c | f | (e_1 \ldots e_n) \in D, 0 < n $$

We also define the set of first order patterns as $\text{FOPtr}_\Sigma = \text{Ptr}_\Sigma \cap \text{FOExp}_\Sigma$. Intuitively, the rank of a constructor is the exact number of arguments needed for building a data structure; the rank of a function symbol is the exact number of arguments needed for knowing how to evaluate the function call; patterns are nonevaluable expressions which represent (possibly H.O.) data; F.O. patterns represent data structures without functional components; F.O. expressions avoid H.O. variables and involve only exact curried applications.

**Convention:** For the rest of the paper, such expressions as "n-ary", "n arguments", etc. must be understood as referring to curried functions.

2.3 Continuous applicative algebras

We assume that the reader is familiar with the notion of Scott domain [14]. For the partial ordering of a given domain $D$ we use the notation $\sqsubseteq_D$ or simply $\sqsubseteq$ if $D$ is clear by the context.

A continuous applicative algebra (C.A.A.) of signature $\Sigma$ is any algebraic structure $\mathcal{A}$ consisting of a Scott domain $D_\mathcal{A}$ with ordering $\sqsubseteq_\mathcal{A}$ as carrier, a continuous binary operation $\circ_\mathcal{A} \in [D_\mathcal{A} \times D_\mathcal{A}]$ (meant as interpretation of application), which is required to be strict w.r.t. its first argument, and interpretations $\phi_\mathcal{A} \in D_\mathcal{A}$ for all symbols $\phi \in DC_\Sigma \cup FS_\Sigma$. The notion is similar to the applicative structures used in combinatory logic [11].

Any $x \in D_\mathcal{A}$ induces n-ary continuous functions $\{x\}_n^\mathcal{A}$ (for all $n \in \mathbb{N}$) defined by