A Step Towards a Methodology for Mercury Program Construction: A Declarative Semantics for Mercury*

Dante Baldan Baudouin Le Charlier Christophe Leclère Isabelle Pollet

Institut d'Informatique
Facultés Universitaires Notre-Dame de la Paix
Rue Grandgagnage 21
B-5000 Namur (Belgium)
{dba,ble,clc,ipo}@info.fundp.ac.be

Abstract

Declarative methodologies for logic program construction have been proposed for Prolog. They roughly consist of 1) building a purely logical version of the program based on a clear declarative semantics and 2) performing a number of checks about modes, types and multiplicity. We plan to define a similar methodology for Mercury. This choice is motivated by the fact that type, mode, and multiplicity must be explicitly specified in Mercury, allowing the compiler to perform the second step above. In order to propose a methodology to perform the first step, we need a declarative semantics for Mercury, which has not yet been explicitly defined. The goal of the paper is to propose such a semantics pursuing simplicity and naturalness. We chose to define the semantics with respect to a unique interpretation domain, called the "universe", which is a kind of higher-order version of the Herbrand universe. Based on this simple domain, the denotation of terms and goals is naturally defined as well as the models of programs. Although the declarative semantics is primarily introduced to improve "manual" program construction by programmers, it could be used in a synthesis context.

1 Introduction

The work presented in this paper originates from two assumptions: 1) constructing logic programs requires some form of explicit reasoning (i.e., a logic program is not both a specification and a program, it is just a program); 2) logic programs should be constructed according to a declarative approach that basically amounts to show that the intended model by the user (described through a specification) is a (distinguished) model of the program.

Declarative methodologies for logic program construction have been proposed for Prolog (see e.g., [Apt97, Dev90, HL92, Nai87]). They roughly consist of 1) building a purely logical version of the program based on a clear declarative semantics and 2) performing a number of checks about modes, types, and multiplicity, as well as some permutations of clauses and literals, to ensure that Prolog actually computes the relation described by the logical version. We plan to define a similar methodology for Mercury [SHC96]. This choice is motivated by the fact that type, mode, and multiplicity information can (and must) be explicitly specified in Mercury, allowing the

*Extended Abstracts of LOPSTR'98, Eighth International Workshop on Logic-based Program Synthesis and Transformation, 15-19 June 1998, Manchester, UK. Technical Report Series, Department of Computer Science, University of Manchester, ISSN 1361-6161. Report number UMCS-98-6-1. http://www.cs.man.ac.uk/cstechrep/titles98.html

compiler to automatically perform the second step described above. However, in order to propose a methodology to perform the first step, we need a declarative semantics for Mercury, which has not yet been explicitly defined. The goal of this paper is to propose such a semantics.

When designing our declarative semantics for Mercury, we have pursued two main objectives: simplicity and naturalness. This is because, in our opinion, the declarative semantics should be a straightforward mathematical underpinning of the intuitive meaning of the programs, allowing the programmer to check its understanding of programs both easily and explicitly. As a consequence, we chose to define the semantics with respect to a unique interpretation domain, called the "universe", which is a kind of higher-order version of the Herbrand universe used in e.g., [Dev90][1]. Based on this simple domain, the denotation of terms and goals (formulas) is naturally defined as well as the models of programs.

Notice that, although the declarative semantics is primarily introduced to improve "manual" program construction by programmers, it could also be used in a synthesis context [FLO97a, FLO97b] to automatically translate synthesized programs into equivalent Mercury programs (see Section 6).

The rest of this paper is organized as follows. Section 2 introduces the notation used in the paper. Section 3 describes a simplified abstract syntax of Mercury, on which the semantics is based. Section 4 introduces the universe of values and the denotation of types, terms and goals. Section 5 defines the semantics. Finally, Section 6 summarizes how we will use the declarative semantics in the program construction process.

2 Notation

Given two sets A and B, $A \to B$ is the set of the functions from A, the domain of f, to B, the codomain of f. A sequence of objects $X_1, \ldots, X_n$ (e.g. Cartesian product of sets, tuples of elements) is indicated by $\overline{X}$ and $|\overline{X}|$ is its length. If f is an n-ary function with $n > 1$, then $f(\overline{t})$ and $f[\overline{t}/\overline{d}]$ denote $f(t_1, \ldots, t_{|\overline{t}|})$ and the function f' that differs from f only on $\overline{t}$ where $f'(\overline{t}) = \overline{d}$, respectively. If f is a 1-ary function then $f(\overline{t})$ and $f[\overline{t}/\overline{d}]$ denote the sequence $f(t_1), \ldots, f(t_{|\overline{t}|})$ and the function f' that differs from f only on $\overline{t}$ where $f'(\overline{t}) = \overline{d}$, respectively. The application of a function f to a syntactic object t will be indicated by $f[\![t]\!]$. The set of the boolean values $\{\mathit{TRUE}, \mathit{FALSE}\}$ is Bool.

3 The Abstract Syntax

This section presents the abstract syntax $\mathcal{S}$ defining a kernel of Mercury leaving apart, for the sake of simplicity, other features such as the modes, the determinism, and the modules. The abstract syntax $\mathcal{S}$ is presented as a set of production rules in which each entity is defined in terms of its constituent parts (Backus-Naur-Form) as, for instance, in first order logic, where a term is either a variable or a constant or a compound term consisting of a function symbol of arity, say n, followed by n-terms: $\mathit{term} ::= \mathit{variable} \mid \mathit{constant} \mid f(\overline{\mathit{term}})$. The productions form a context-free grammar with attributes, i.e. with context conditions or specifications. For instance, in the above example, the context conditions require that $|\overline{\mathit{term}}| = n$. The context conditions will be given in natural language, the terminals used in $\mathcal{S}$ will be indicated in `typewriter`, and *italic* will be reserved for atomic or compound objects.

[1]This choice forces us to impose some limitations on recursive type definitions.

3.1 Symbols for Atomic Sets

The symbols for the atomic sets used in $\mathcal{S}$ are *var*, *constr*, *ident* indicating the program variables, the term constructors, and the identifiers (predicate and function names), respectively. Furthermore, $\mathcal{S}$ uses type variables, indicated by *tvar*, and type constructors, indicated by *tconstr*. As usual in abstract syntax, by "indicating", we mean that each occurrence of, say *var*, can be replaced by any program variable. In the following, the symbols for atomic sets will be referred to as atomic symbols.

3.2 Symbols for Constructed Sets

The abstract syntax $\mathcal{S}$ provides rules for constructing terms from atomic symbols. The most important difference with respect to first order logic is that $\mathcal{S}$ allows functions and predicates to be terms, i.e. $\mathcal{S}$ allows higher order terms or lambda expressions. A predicate term is composed by the terminal `pred` followed by a sequence of distinct variables (representing the arguments) and by the goal defining the predicate self. Similarly, a function term is composed by the terminal `func` followed by a sequence of distinct variables (representing the input), by a single variable (representing the output), and by the goal defining the function self. Also, $\mathcal{S}$ allows term applications, i.e. the application of an identifier (or a term) to a sequence of terms as arguments (higher order function application). Formally, a *term* is defined as follows:

$$\mathit{term} ::= \mathit{var} \mid \mathit{constr}\ \overline{\mathit{term}} \mid \texttt{pred}\ \overline{\mathit{var}}\ \mathit{goal} \mid \texttt{func}\ \overline{\mathit{var}}\ \mathit{var}\ \mathit{goal} \mid \texttt{apply}\ \mathit{ident}\ \overline{\mathit{term}} \mid \texttt{apply}\ \mathit{term}\ \overline{\mathit{term}}$$

The context information for the above rule is that the length of the sequences of terms match the arity of the corresponding functors. More specifically, we mean that the arity of *constr*, *ident*, and *term* match exactly the length of $\overline{\mathit{term}}$ in *constr* $\overline{\mathit{term}}$, `apply` *ident* $\overline{\mathit{term}}$, and in `apply` *term* $\overline{\mathit{term}}$. In the following similar conditions will be referred to as *arity constraints*. We remark that $\mathcal{S}$ does not support "currying", i.e., the application of a function term specifying only the first few arguments and leaving the remaining arguments unspecified. This feature of Mercury can be implemented in $\mathcal{S}$ by replacing incomplete function calls with suitable lambda expressions.

The second symbol for a set of constructed objects in $\mathcal{S}$ is *goal*. The most important difference with respect to first order logic is that $\mathcal{S}$ allows the calls of terms applied to other terms, known as higher order predicate calls and indicated by `call` *ident* $\overline{\mathit{term}}$ or `call` *term* $\overline{\mathit{term}}$. The arity constraints must be satisfied in `call` *ident* $\overline{\mathit{term}}$ and in `call` *term* $\overline{\mathit{term}}$, allowing no currying (see above). For the sake of simplicity, $\mathcal{S}$ does not contain all possible goals of Mercury: for example $\mathcal{S}$ does not present the universal quantification nor the DCG-rules because they are easily introduced as abbreviations. Formally, a *goal* is defined as follows:

$$\begin{array}{lll} \mathit{goal} & ::= & \texttt{true} \mid \texttt{fail} \mid \texttt{unif}\ \mathit{term}\ \mathit{term} \mid \texttt{call}\ \mathit{ident}\ \overline{\mathit{term}} \mid \texttt{call}\ \mathit{term}\ \overline{\mathit{term}} \mid \texttt{not}\ \mathit{goal} \\ & & \mathit{goal}\ \texttt{and}\ \mathit{goal} \mid \mathit{goal}\ \texttt{or}\ \mathit{goal} \mid \texttt{if}\ \mathit{goal}\ \texttt{then}\ \mathit{goal}\ \texttt{else}\ \mathit{goal} \mid \texttt{exists}\ \overline{\mathit{var}}\ \mathit{goal} \end{array}$$

where $\overline{\mathit{var}}$ is a sequence of distinct variables.

All terms, predicates and functions are typed in $\mathcal{S}$ (and in Mercury) using a polymorphic type system à la ML. The syntax of types in $\mathcal{S}$ allows the definition of predicate and function types specifying the types of the arguments and result (the last one only for function types). More precisely, the type of a predicate is specified by `pred` followed by the sequence $\overline{\mathit{type}}$ that represents the types of the arguments, and the type of a function is specified by `func` followed by the sequence composed by $\overline{\mathit{type}}$ and *type* that represent the types of arguments and output, respectively. Also, $\mathcal{S}$ allows the definition of new types using *constr*. For example, the definition of the polymorphic type $list(T)$ is given through the constructors $[]/0$ and $./2$ as follows: $list\ T\ \ ([]\ ,\ .(T, list(T)))$. The formal definition is:

$$type ::= tvar \mid tconstr\ \overline{type} \mid \texttt{pred}\ \overline{type} \mid \texttt{func}\ \overline{type}\ type \qquad typedef ::= tconstr\ \overline{tvar}\ \overline{(constr\ \overline{type})}$$

The arity constraints have to be satisfied in $tconstr\ \overline{type}$, $tconstr\ \overline{tvar}$, and $constr\ \overline{type}$. Also, for each term constructor *constr* there exists at most one *typedef* containing it (and at most once): this means that we do not allow types to have partially common structure. Moreover, there exists at most one type definition containing a given type constructor in the left hand side of the definition. At the concrete syntax level (i.e., in "real" Mercury), the same constructor symbol may occur in several type definitions but we assume that all occurrences of the considered symbol are replaced by different constructors when the program is translated in abstract syntax. We assume that $\overline{tvar}$ is a sequence of distinct variables and that all type variables occurring in $\overline{(constr\ \overline{type})}$ occurs also in $\overline{tvar}$, i.e., the types are transparent. The word "transparent" is meant to indicate that the parameters in a type of a subterm can be seen in the type of the term that contains it.

3.3 Programs

A program in $\mathcal{S}$ is a set of predicate and function definitions illustrated below.

$$preddef ::= ident\ \overline{var}\ goal \qquad funcdef ::= ident\ \overline{var}\ var\ goal$$

In both definitions we assume that the arity constraints are verified and that all free variables occurring in *goal* appear also in $\overline{var}$ and in $\overline{var}\ var$, respectively.

In this paper we assume that the program is well-typed, i.e., that there exists a most general type assignment for program variables and identifiers. For space constraints, we cannot give the rules characterizing well-typed programs (such rules are given in [LLP98]). In the following, the most general type assignment for a program will be referred to as *the* typing (of the program). A consequence of the well-typedness of a program is that the types of the arguments of a term can be more "specific" than the ones given by the typing of the considered term. For instance, let `call` *id* 1 and `call` *id* $[1,2]$ be two calls for *id* returning an integer and a list of integers, respectively. The most general type of *id* is $tv \to tv$ and thus, for each call there is a type substitution, i.e., an object of the form $\{tv_1/ty_1, \ldots, tv_n/ty_n\}$ with $tv_1, \ldots, tv_n$ distinct type variables and $ty_1, \ldots, ty_n$ types, that returns the types of the *id* in the call. Type substitutions will be indicated by α. This observation will be used in the following.

4 The Denotation of Types, Functions, and Predicates

We present the denotation of types, functions, and predicates into a domain of values, called the *universe* U.

4.1 The Universe of Values

Let *Var*, *Ident*, and *Constr* be enumerable and disjoint sets of variables, identifiers, constructors given once and for all. They can be seen as the possible values for the syntactic objects *var*, *ident*, and *constr*. Intuitively, the *universe* U of values is the set of ground terms closed under constructor application, function construction, and predicate construction. Formally, the *universe* U is the union of all sets of the increasing chain $U_0 \subseteq U_1, \ldots, U_i, \ldots$ defined below.

Definition 4.1 *Let U_0 be the set of first order ground terms constructed using Constr as follows: if $c \in Constr$ is an n-ary constructor and $d_1, \ldots, d_n$ are elements of U_0 then $c\overline{d} \in U_0$. For all $i > 0$, let us define U_{i+1} as follows: (1) let $Func_i$ be the set of all* total *functions f such that $f \in D_1 \times \cdots \times D_m \to U_i$ with $D_1, \ldots, D_m \subseteq U_i$; (2) let $Pred_i$ be the set of all predicates p such that $p \in D_1 \times \cdots \times D_n \to Bool$ where $D_1, \ldots, D_n \subseteq U_i$; (3) let $U_i^! \equiv U_i \cup Func_i \cup Pred_i$; (4) Finally, U_{i+1} is the smallest set such that (4.a) $U_i^! \subseteq U_{i+1}$; (4.b) if $c \in Constr$ is an n-ary constructor and $d_1, \ldots, d_n \in U_i^!$ then $c\overline{d} \in U_{i+1}$.*

We remark that U contains *total* functions while Mercury allows partial functions: this does not result in a less expressive domain of values because any partial function can be rewritten as a predicate and predicates are already contained in U. Notice also that the components of the function and predicate domains are contained in the family, denoted by $\mathcal{A}(U)$, of *admissible* subsets of U, i.e., sets contained in some U_i of the hierarchy defining U. This means that we will be able to model only this kind of functions and predicates, which, in turn, imposes some restrictions on the recursive definitions of functional and predicate types.

4.2 The Denotation of Types

Informally speaking, because of the polymorphism of the type system, the most general type annotation for a program P provides a family of domains for the program variables and a family of signatures for the identifiers in P.

More precisely, let $\mathsf{tconstr}$, tvar, and type be the set of type constructors, type variables and types. They can be seen as the concretization of *tconstr*, *tvar*, and *type* of the abstract syntax. Let $\mathsf{Tenv} \stackrel{\triangle}{=} (Var \cup Ident) \to \mathsf{type}$ and $\mathsf{Denv} \stackrel{\triangle}{=} \mathsf{tvar} \to \mathcal{A}(U)$ be the set of type environments and the set of domain environments, respectively. Elements of Denv will be indicated by δ. The denotation of types is given by means of two mutually recursive functions τ and $\mathcal{T}$. We introduce the denotations of types by induction on the definition of types in the abstract syntax.

Definition 4.2 *The denotation of a type $\mathsf{tc}(\overline{tv}) ::= \ldots |c_i\overline{ty_i}| \ldots$ is given by the following pair of functions:*

$$\tau : \mathsf{tconstr} \to \mathsf{Denv} \to \mathcal{A}(U) \quad \mathcal{T} : \mathsf{type} \to \mathsf{Denv} \to \mathcal{A}(U)$$

The most significant rules are:

$$\tau[\![\mathsf{tc}]\!]\delta \stackrel{\triangle}{=} \textstyle\bigcup_{i=1}^{n} \{c_i\overline{d_i} : \overline{d_i} \in \mathcal{T}[\![\overline{ty_i}]\!]\delta\} \quad \mathcal{T}[\![\mathsf{func}\ \overline{ty}\ ty]\!]\delta \stackrel{\triangle}{=} \overline{D} \to D \quad \mathcal{T}[\![\mathsf{pred}\ \overline{ty}]\!]\delta \stackrel{\triangle}{=} \overline{D} \to Bool$$

where $\overline{D} = \mathcal{T}[\![\overline{ty}]\!]\delta$ and $D = \mathcal{T}[\![ty]\!]\delta$.

4.3 The Denotation of Terms

The denotation of a term t requires some preliminary explanation. Given a term t, the typing for the program provides a type for all variables and identifiers occurring in t. Since types are polymorphic, the type of t corresponds to a possibly infinite family of sets of values. The possible values of each type variable occurring in the type of t are given by a function in Denv that associates a(n) (admissible) set of values to each type variable of the type of t. At this point, each program variable and each identifier occurring in t have a set of values where they can range. For fixing the denotation of t, we need other two functions: an identifier environment ι in $\mathsf{Ienv} \stackrel{\triangle}{=} Ident \to \mathsf{Denv} \to U$, and a variable environment ν in $\mathsf{Venv} \triangleq \mathit{Var} \to U$. Notice the asymmetry between the signatures of the identifier and variable environments. It is due to the fact that an identifier has arguments whose domains are known through the (polymorphic) type of the identifier. The denotation of a term t is formalized in the following function $\mathcal{V}$ that can be defined inductively on the syntax of a term. In the following we assume that P is well-typed and $\theta \in \mathsf{Tenv}$ is the most general typing of the program.

Definition 4.3 *Let Term be the set of terms. The denotation function $\mathcal{V}$ is as follows.*

$$\mathcal{V} : \mathit{Term} \to \mathsf{Tenv} \to \mathsf{Denv} \to \mathsf{Ienv} \to \mathsf{Venv} \to U \cup \{\mathit{error}\}$$

The most significant rules are the following. The denotation of a variable v is the value of v in the variable environment ν if $\nu[\![v]\!]$ is a value compatible with the type of v:

$$\mathcal{V}[\![v]\!]\theta\delta\iota\nu \triangleq \nu[\![v]\!] \text{ if } \nu[\![v]\!] \in \mathcal{T}[\![\theta[\![v]\!]]\!]\delta, \text{ otherwise } \mathit{error}$$

The denotation of a predicate $\mathtt{pred}\ \overline{v}\ g$ is a predicate in U that associates each value $\overline{d}$ in its domain (determined by the typing of $\overline{v}$) with the denotation of the goal g assuming that $\overline{v}$ is replaced by $\overline{d}$:

$$\mathcal{V}[\![\mathtt{pred}\ \overline{v}\ g]\!]\theta\delta\iota\nu \triangleq \lambda\overline{d} \in \mathcal{T}[\![\theta[\![\overline{v}]\!]]\!]\delta.\mathcal{G}[\![g]\!]\theta\delta\iota\nu[\overline{v}/\overline{d}]$$

The denotation of a function $\mathtt{func}\ \overline{v}\ v\ g$ is a function f that associates each value $\overline{d}$ in its domain (determined by the typing of $\overline{v}$) with a value $f\overline{d}$ in its codomain (determined by the typing of v) such that the denotation of the goal g is TRUE when substituting $\overline{d}$ and $f\overline{d}$ in place of $\overline{v}$ and v, respectively:

$$\mathcal{V}[\![\mathtt{func}\ \overline{v}\ v\ g]\!]\theta\delta\iota\nu \triangleq \mathsf{choice}\{f \in \overline{D} \to D : \forall\overline{d} \in \overline{D} : \mathcal{G}[\![g]\!]\theta\delta\iota\nu[\overline{v}/\overline{d}, v/f\overline{d}]\}$$

where choice indicates "any function" (axiom of choice), $\overline{D} \equiv \mathcal{T}[\![\theta[\![\overline{v}]\!]]\!]\delta$, and $D \equiv \mathcal{T}[\![\theta[\![v]\!]]\!]\delta$. We remark that the axiom of choice is required because there can be many different functions satisfying the requirement.

The most intricate rule is the one for the terms that are applications of functions. We recall that thanks to the well-typing of the program, the arguments and the result of each function application have a type that are instances of the most general type annotation for the function. This means that the domain environment δ', used in the following rule for computing f has to instantiate the type variables with sets of more specific values than δ.

$$\mathcal{V}[\![\mathtt{apply}\ id\ \overline{t}]\!]\theta\delta\iota\nu \triangleq f\overline{d}^{2}$$

where there exists a type substitution α such that

$$\overline{ty} \equiv \tau_y[\![\overline{t}]\!]\theta \quad \mathsf{func}\ \overline{ty}_1\ ty_2 \equiv \theta[\![id]\!] \quad \overline{ty} \equiv \overline{ty}_1\alpha \quad \delta' \equiv \lambda tv.\mathcal{T}[\![\alpha[\![tv]\!]]\!]\delta \quad f \equiv \iota[\![id]\!]\delta' \quad \overline{d} \equiv \mathcal{V}[\![\overline{t}]\!]\theta\delta\iota\nu$$

and τ_y is the extension of θ from type variables to type terms, defined as follows:

$$\tau_y : \mathit{Term} \to \mathsf{Tenv} \to \mathsf{type}$$

We present only the following rule:

$$\tau_y[\![\mathtt{apply}\ id\ \overline{t}]\!]\theta \triangleq ty''\alpha$$

where $\theta[\![id]\!] = \mathsf{func}\ \overline{ty'}\ ty''$, $\tau_y[\![\overline{t}]\!]\theta = \overline{ty'}\alpha$ for some type substitution α.

[2] If $\overline{d} \notin \mathit{dom}(f)$ then $f\overline{d} = \mathit{error}$.

4.4 The Denotation of Goals

Given a goal g, its value can be either $TRUE$ or $FALSE$ or $error$.

Definition 4.4 *Let Goal be the set of goals. The denotation function $\mathcal{G}$ is as follows.*

$$\mathcal{G} : Goal \to \mathsf{Tenv} \to \mathsf{Denv} \to \mathsf{Ienv} \to \mathsf{Venv} \to Bool \cup \{error\}$$

The complete definition cannot be supplied, thus, we present only the non trivial cases.

The denotation of `unif` $t_1\ t_2$ is the truth value of the equation $d_1 = d_2$, where $d_i = \mathcal{V}[\![t_i]\!]\theta\delta\iota\nu$ for $i = 1, 2$ and "=" means "the same element in U". If $d_1 = error$ or $d_2 = error$ then also $d_1 = d_2$ is evaluated to $error$. We remark that the evaluation of "=" could amount to solve an undecidable problem (for instance, if d_1 and d_2 are functions) and in that case we can safely return $error$ because also the goal under exam cannot solve the problem of function equivalence.

Given a higher order predicate call `call` $id\ \overline{t}$, its associated denotation is the value of the predicate corresponding to id applied to the arguments corresponding to $\overline{t}$ taking into account that the type of $\overline{t}$ can be more instantiated than the type of the arguments of id provided by the typing for the program. Formally:

$$\mathcal{G}[\![\texttt{call}\ id\ \overline{t}]\!]\theta\delta\iota\nu \triangleq p\overline{d}^{3}$$

where there exists a type substitution α such that

$$\overline{ty} \equiv \tau_y[\![\overline{t}]\!]\theta \quad \mathsf{pred}\ \overline{ty}_1 \equiv \theta[\![id]\!] \quad \overline{ty} \equiv \overline{ty}_1\alpha \quad \delta' \equiv \lambda tv.\mathcal{T}[\![\alpha[\![tv]\!]]\!]\delta \quad p \equiv \iota[\![id]\!]\delta' \quad \overline{d} \equiv \mathcal{V}[\![\overline{t}]\!]\theta\delta\iota\nu$$

The denotation of an existential quantification is defined as follows: if there exists $\overline{d} \in \mathcal{T}[\![\theta[\![\overline{v}]\!]]\!]\delta$ such that $\mathcal{G}[\![g]\!]\theta\delta\iota\nu[\overline{v}/\overline{d}] = error$ then

$$\mathcal{G}[\![\texttt{exists}\ \overline{v}\ g]\!]\theta\delta\iota\nu \triangleq error$$

Otherwise, if there exists $\overline{d} \in \mathcal{T}[\![\theta[\![\overline{v}]\!]]\!]\delta$ such that $\mathcal{G}[\![g]\!]\theta\delta\iota\nu[\overline{v}/\overline{d}] = TRUE$ then

$$\mathcal{G}[\![\texttt{exists}\ \overline{v}\ g]\!]\theta\delta\iota\nu \triangleq TRUE \text{ else } \mathcal{G}[\![\texttt{exists}\ \overline{v}\ g]\!]\theta\delta\iota\nu \triangleq FALSE$$

5 Models of Well-Typed Programs

We recall that a program is a set of predicate definitions of the form $(id\ \overline{v}\ g)$ and of function definitions of the form $(id\ \overline{v}\ v\ g)$. A model for a program P is an identifier environment $\iota \in \mathsf{Ienv}$ that satisfies all the predicate and function definitions in the following sense.

Definition 5.1 *Let P be a well-typed program and let $\theta \in \mathsf{Tenv}$ be its most general type annotation. A model of P is a function $\iota \in \mathsf{Ienv}$ such that for all $\delta \in \mathsf{Denv}$ and $\nu \in \mathsf{Venv}$: (1) For every predicate definition $(id\,\overline{v}\ g) \in P$:*

$$\forall \overline{d} \in \overline{D} : \iota[\![id]\!]\delta\overline{d} = \mathcal{G}[\![g]\!]\theta\delta\iota\nu[\overline{v}/\overline{d}]$$

where $\overline{D} \equiv \mathcal{T}[\![\theta[\![\overline{v}]\!]]\!]\delta$; (2) For every function definition $(id\ \overline{v}\ v\ g) \in P$:

$$\forall \overline{d} \in \overline{D} : \mathcal{G}[\![g]\!]\theta\delta\iota\nu[\overline{v}/\overline{d}, v/f\overline{d}] = TRUE$$

where $\overline{D} = \mathcal{T}[\![\theta[\![\overline{v}]\!]]\!]\delta$ and $D = \mathcal{T}[\![\theta[\![v]\!]]\!]\delta$.

[3] If $\overline{d} \notin dom(p)$ then $p\overline{d} = error$.

6 Discussion

The declarative semantics, which has just been defined, can be used to build logic descriptions [Dev90] in Deville's style, which is mainly based on structural induction and ensures —roughly speaking— that the user's intended model is the unique model of the program; the novelty is that logical descriptions can now be higher-order. Further works will adapt SLDNF-resolution [Llo87] to the context of Mercury to make it possible to prove that well typed and moded Mercury programs actually compute a set of answers covering the declarative semantics.

In the context of schema guided synthesis of logic programs [FLO97a, FLO97b], we guess that it is natural to see the isoinitial models of open programs as denotations of polymorphic functions and procedures in Mercury. This would ensure that well-moded Mercury translations of correct open programs [FLO97a, FLO97b] are operationally correct.

References

[Apt97] K. R. Apt. *From Logic Programming to Prolog.* Prentice Hall, 1997.

[Dev90] Y. Deville. *Logic Programming: Systematic Program Development.* MIT Press, 1990.

[FLO97a] P. Flener, K.-K. Lau, and M. Ornaghi. Correct-Schema-Guided Synthesis of Steadfast Programs. In M. Lowry and Y. Ledru, editors, *Proceedings of ASE'97*, Computer Society Press. IEEE, 1997.

[FLO97b] P. Flener, K.-K. Lau, and M. Ornaghi. On Correct Program Schemas. In N. E. Fuchs, editor, *Proceedings of LOPSTR'97*, LNCS. Springer-Verlag, 1997.

[HL92] J. Henrard and B. Le Charlier. Folon: An Environment for Declarative Construction of Logic Programs (Extended Abstract). In M. Bruynooghe and M. Wirsing, editors, *Proc. of PLILP'92*, volume 631 of *LNCS*, Leuven, Belgium, August 1992. Springer-Verlag.

[LLP98] B. Le Charlier, C. Leclère, and I.Pollet. A Declarative Semantics for Mercury (in French). Technical report, Institute d'Informatique, FUNDP, Namur, Belgique, 1998.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming.* Springer Series: Symbolic Computation–Artificial Intelligence. Springer-Verlag, second, extended edition edition, 1987.

[Nai87] L. Naish. Specification = Program + Types. In Kesav V. Nori, editor, *Proceedings of the 7th Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 287 of *LNCS*, pages 326–339, Pune, India, December 1987. Springer.

[SHC96] Z. Somogyi, F. Henderson, and T. Conway. The Execution Algorithm of Mercury, an Efficient Purely Declarative Logic Programming Language. *Journal of Logic Programming*, 29(1–3):17–64, October–November 1996.