On the Relationship between Logic Programs and Specifications*

Kung-Kiu Lau
Dept. of Computer Science
University of Manchester
Oxford Road, Manchester M13 9PL
United Kingdom
kung-kiu@cs.man.ac.uk

Mario Ornaghi
DSI
Università degli studi di Milano
Via Comelico 39/41, Milano
Italy
ornaghi@dsi.unimi.it

Abstract

The fundamental relation between a program P and its specification S is correctness: P satisfies S if and only if P is correct with respect to S. In logic programming, this relationship can be particularly close, since logic can be used to express both specifications and programs. Indeed logic programs are often regarded and used as (executable) specifications themselves. In this paper, we argue that the relation between S and P should be firmly set in the context of the underlying problem domain, which we call a framework $\mathcal{F}$, and we give a model-theoretic view of the correctness relation between specifications and programs in $\mathcal{F}$. We show that the correctness relation between S and P is always well-defined. It thus provides a basis for properly distinguishing between S and P. We use the subset example throughout, to illustrate our model-theoretic approach.

1 Introduction

The fundamental relationship between a logic program P and its specification S is *correctness*: P satisfies S if and only if P is correct with respect to S. The precise nature of this relationship depends on the chosen specification language L_S (and the notion of correctness that has been adopted). For instance, if L_S is also a logic language, then this relation could be very close (see e.g. [8]). Indeed logic programs are often regarded and used as (executable) specifications themselves. After all, a logic program is a Horn clause theory ([7]), and as such it can double as a definition. This would seem to suggest that L_S could be just Horn clause logic, and the notion of correctness is redundant.

In this paper, we argue that the relationship between S and P should be set firmly in the context of the underlying problem domain, which we shall call the *specification framework* $\mathcal{F}$. We will show that in $\mathcal{F}$ the relationship between S and P is always well-defined, and the notion of correctness is never redundant.

A specification framework $\mathcal{F}$ may be defined either formally or informally, but it should provide an unambiguous underpinning of the semantics of any given S and P, as well as the semantics of P's correctness with respect to S. The full meaning of S, including the *specified relation* r, is defined in $\mathcal{F}$. Since P does not contain axioms for

*This paper has been extracted from [14].

$\mathcal{F}$, we can always distinguish between S and P. Nevertheless, P is *correct* with respect to S if P computes r according to S.

This approach makes sense not only for (closed) informal and formal frameworks, but also for *open*, or parameterized frameworks. The latter support a modular approach to program derivation. For example, when composing two frameworks $\mathcal{F}$ and $\mathcal{G}$, the composite $\mathcal{F}[\mathcal{G}]$ inherits the correct programs derived in $\mathcal{F}$ and in $\mathcal{G}$. These programs, in turn, can be composed correctly in $\mathcal{F}[\mathcal{G}]$.

This paper is part of our research on specification frameworks and program derivation. Here we will concentrate on specifications and correctness. A more general discussion is given in the conclusions. Throughout the paper, we will use the *subset* relation as an illustration. In order to be brief, we will give an informal presentation and we will not treat correctness in open frameworks. Finally, we shall consider only definite programs, although our approach extends to normal programs.

2 Model-theoretic Correctness

When the underlying problem domain, or framework $\mathcal{F}$, is not formally defined, we can have only informal specifications. Nevertheless, we can (informally) define the correctness of a program P with respect to a specification S as follows:

> If the meaning of the specified relation r in S coincides completely with the meaning of r in the minimum Herbrand model of P, then P is *correct* with respect to S.

We illustrate the correctness relationship between S and P in an informal framework, by considering a Prolog program P1 for *subset* together with its informal specification. P1 uses unordered lists (possibly) containing duplicates to represent sets:

```
member(X, [X|_]).
member(X, [_|Xs])   :-  member(X, Xs).
                                                    (P1)
   subset([], _).
subset([X|Xs], Ys)  :-  member(X, Ys), subset(Xs, Ys).
```

The informal specification S1 of P1 is the following:

> "`subset(Small, Large)` is to be true if: `Small` and `Large` are two lists of integers and the set `Small` represents is a subset of the set that `Large` represents." (S1)

Here the informal (implicit) framework $\mathcal{F}$ is that of finite sets and lists of integers. However, 'informal' does not mean 'vague': we know exactly what sets and lists (and integers) are. The important point is that S1 receives its full, precise meaning in $\mathcal{F}$. In particular, the elements of a set can be listed in any order and may contain duplicates.

We can compare S1 with P1 by comparing the meaning of `subset` stated by S1 with its meaning in the minimum Herbrand model of P1. It is easy to see that these two meanings of `subset` coincide completely, i.e. P1 is correct with respect to S1. For example, for the query Q1:

$$? - \texttt{subset}([\texttt{A}, \texttt{B}], [1, 2, 3]). \qquad \text{(Q1)}$$

P1 produces the following nine expected answers (i.e. values of `A`, `B` such that `subset([A,B], [1,2,3])` is *true* according to S1):

```
A=1,B=1;  A=1,B=2;  A=1,B=3;
A=2,B=1;  A=2,B=2;  A=2,B=3;
A=3,B=1;  A=3,B=2;  A=3,B=3.
```

This example illustrates the central rôle of the informal framework $\mathcal{F}$ in the relationship between a specification S and a program P. Here $\mathcal{F}$ contains an (implicit) informal definition of (the theory of) finite sets and lists (of integers), which underlies S1, i.e. the full meaning of S1 is defined only in $\mathcal{F}$. On the other hand, P1 does not and cannot completely axiomatize such abstract data types. Consequently, as theories, programs cannot fully coincide with specifications. Nevertheless, $\mathcal{F}$ allows us to reason about the correctness of P with respect to S by comparing the meanings of the specified relation r in S and in the minimum Herbrand model of P.

When the framework $\mathcal{F}$ is defined formally, we can formally define the semantics of a specification S, and its correctness relation with a program P in terms of either proof theory or model theory. In this section, we shall consider the model-theoretic correctness relation between S and P, within a formal framework $\mathcal{F}$, following our approach to deductive synthesis of (both standard and constraint) logic programs (see e.g. [10, 12]).

In general, model-theoretic correctness is based on the comparison between specifications and the intended models of programs (see e.g. [4] for a brief survey). The distinguishing feature of our approach is that specifications, programs and correctness are defined within a framework $\mathcal{F}$ (the use of frameworks is discussed in [9, 11]).

We shall define $\mathcal{F}$ to be a full first-order logical theory, S a first-order formula in $\mathcal{F}$ that defines a (set of) relation(s) r, and P a Horn theory whose language contains the relation(s) r. Both $\mathcal{F}$ and P have intended models: the intended model of P is its minimum Herbrand model H, while the intended model I of $\mathcal{F}$ is determined by some intended model semantics. We define correctness as follows:

> P is *correct* with respect to S in $\mathcal{F}$ iff the minimum Herbrand model H of P and the intended model[1] are isomorphic when restricted to the relations r defined by S.

This is illustrated in Figure 1.[2] The new symbol r defined in S has an interpretation in

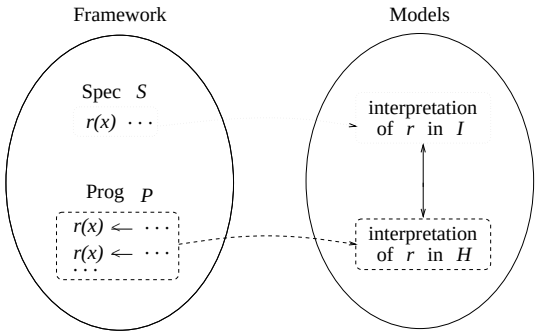

Figure 1: Model-theoretic relation between S and P.

I, and one in H. Correctness means that the two interpretations of r coincide (or, at least, are isomorphic).

[1] We will use the *isoinitial model* I of $\mathcal{F}$, as explained later.

[2] Dotted arrows denote semantic mappings, and the full double-headed arrow represents an isomorphism.

2.1 Specification Frameworks

A (specification) framework is an axiomatization of a problem domain. It has (i) a (many-sorted) signature of sort symbols, function symbols, including constant symbols, and relation symbols; and (ii) a set of first-order axioms for the function and relation symbols.

For example, we define a framework $\mathcal{SET}(Int)$ for sets as lists of integers as follows:

Framework $\mathcal{SET}(Int)$;
IMPORT: $\mathcal{LIST}(Int)$;
SORTS: $Int, List, Set$;
FUNCTIONS: $\{\}\ :\ List \rightarrow Set$;
RELATIONS: $\in\ :\ (Int, Set)$;
AXIOMS: $\forall L : List,\ x : Int.\ (x \in \{L\} \leftrightarrow mem(x, L))$;
$\forall A, B : Set.\ (A = B \leftrightarrow \forall x : Int.\ (x \in A \leftrightarrow x \in B))$.

$\mathcal{SET}(Int)$ imports a pre-defined framework $\mathcal{LIST}(Int)$ for lists of integers, which is the standard theory of lists (of integers) with the usual list operations, in particular the 'list membership' relation *mem*. *Int* stands for the standard (pre-defined) integer type. *Set* is constructed by the constructor $\{\}$. $\in$ is the usual 'set membership' relation. The first axiom defines $\in$ in terms of *mem*, while the second defines set equality in terms of $\in$.

In general, a framework $\mathcal{F}$ typically defines a new abstract data type T (e.g. *Set*) starting from pre-defined types (e.g. *Int* and *List*). T is constructed from *constructors* (e.g. $\{\}$), declared as functions. Axioms for the old sorts are imported, and new ones are added to define new functions and relations on T.

The syntax of a framework $\mathcal{F}$ is similar to that used in algebraic abstract data types (e.g. [17]), or in typed logic programming languages such as Gödel [6]. However, whilst an algebraic abstract data type is an *initial* model (defined below) of its specification, the intended model of $\mathcal{F}$, i.e. the abstract data type it axiomatizes, is an *isoinitial* model. If $\mathcal{F}$ has a *reachable* model, i.e. one where each element (of the domain) can be represented by a ground term, then isoinitial models can be characterized as follows:

> An *isoinitial model* I of $\mathcal{F}$ is a reachable model such that for any relation r defined in $\mathcal{F}$, ground instances $r(t)$ or $\neg r(t)$ are true in I iff they are true in all models of $\mathcal{F}$.

Such a model is also an initial model,[3] which for completeness we also define here:

> An *initial model* J of $\mathcal{F}$ is a reachable model such that for any relation r defined in $\mathcal{F}$, ground instances $r(t)$ are true in J iff they are true in all models of $\mathcal{F}$.

In general, the existence of an isoinitial model is of course not guaranteed. However, if a framework $\mathcal{F}$ has a reachable model I, then it can be shown that I is an isoinitial model of $\mathcal{F}$ if and only if for every closed atomic formula A (in $\mathcal{F}$), either $\mathcal{F} \vdash A$ or $\mathcal{F} \vdash \neg A$. This is a useful *existence condition* for isoinitial models. Furthermore, it can be shown that if $\mathcal{F}$ is a (possibly infinite) recursively enumerable axiomatization, then there

[3] This holds only for reachable models. For non-reachable models initiality and isoinitiality are independent properties (see [1]).

exists an isoinitial model in which relation symbols are interpreted by decidable relations, and function symbols by total computable functions. Indeed, we only ever construct such frameworks.

To construct such a framework $\mathcal{F}_n$ for axiomatizing a type T, we proceed incrementally. We start with a small framework $\mathcal{F}_0$ with an obvious isoinitial model I_0. For instance, if the freeness axioms (see [16]) hold in T, then $\mathcal{F}_0$ consists of just the constructor symbols in $\mathcal{F}_n$ together with their freeness axioms. The term model generated by the constructors is of course just the set of ground terms of T. It is an isoinitial model of their freeness axioms, i.e. it is an isoinitial model of $\mathcal{F}_0$.

Then we successively expand $\mathcal{F}_i$ into $\mathcal{F}_{i+1}$ by adding new function and relation symbols, together with their axioms, in such a way that I_i can be expanded into an isoinitial model I_{i+1} of $\mathcal{F}_{i+1}$.

For example, the following framework $\mathcal{NAT}$ axiomatizes the abstract data type of Peano arithmetic:

Framework $\mathcal{NAT}$;

SORTS:	Nat;		
FUNCTIONS:	0	:	$\rightarrow Nat$;
	s	:	$Nat \rightarrow Nat$;
	$+, *$	:	$(Nat, Nat) \rightarrow Nat$;
AXIOMS:	$\forall x.\,(\neg 0 = s(x)) \wedge \forall x, y.\,(s(x) = s(y) \rightarrow x = y)$;		
	$\forall x.\,(x + 0 = x)$;		
	$\forall x, y.\,(x + s(y) = s(x + y))$;		
	$\forall x.\,(x * 0 = 0)$;		
	$\forall x, y.\,(x * s(y) = x + x * y)$;		
	$\forall (H(0) \wedge \forall i.\,(H(i) \rightarrow H(s(i))) \rightarrow \forall x.\,H(x))$.		

and is constructed thus: $\mathcal{F}_0$ consists of only the constructors 0 and s, together with their freeness axiom, viz. the first axiom in $\mathcal{NAT}$. An isoinitial model I_0 of $\mathcal{F}_0$ is the term model generated by 0 and s. By interpreting 0 as the natural number *zero*, and s as the *successor* function, we can interpret the domain of I_0 as the natural numbers.

To get $\mathcal{F}_1$ we add the axioms for the function $+$, which define *sum* in a primitive recursive manner. The isoinitial model I_0 is thus expanded to I_1 with $+$ interpreted in this way. I_1 is an isoinitial model of $\mathcal{F}_1$.

To get the complete framework $\mathcal{NAT}$, we then add the function $*$ for *product*, and expand I_1 accordingly into the isoinitial model I of $\mathcal{NAT}$, viz. the abstract data type of natural numbers with the usual *sum* and *product* operations. In other words, $\mathcal{NAT}$ axiomatizes Peano arithmetic.

As we said, the existence of an isoinitial model is not guaranteed. In a framework $\mathcal{F}$, the lack of an isoinitial model corresponds to an incomplete characterisation of the symbols of the framework by the axioms. $\mathcal{F}$ is said to be *closed*, if it has an isoinitial model. Otherwise, it is said to be *open*. Particularly interesting open frameworks are *parametric frameworks* $\mathcal{F}(\Pi)$, where the *parameters* Π are a subsignature of that of $\mathcal{F}$.

A parametric framework $\mathcal{F}(\Pi)$can be characterised by its compositional properties. Let $\Sigma_{\mathcal{F}}$ be the signature of $\mathcal{F}$. Let $\mathcal{G}$ be a *closed* framework with a signature $\Sigma_{\mathcal{G}}$ such that Π is the intersection of $\Sigma_{\mathcal{G}}$ and $\Sigma_{\mathcal{F}}$. If $\mathcal{G}$ proves the parameter axioms of $\mathcal{F}$, i.e., those with signature Π, then the *closed $\mathcal{G}$-instance of* $\mathcal{F}$, written $\mathcal{F}[\Pi :: \mathcal{G}]$, is the set-theoretic union of the two frameworks.

An open framework $\mathcal{F}(\Pi)$ is *parametric* if every closed instance $\mathcal{F}[\Pi :: \mathcal{G}]$ is a closed framework.

For example, we can construct the framework $\mathcal{LIST}(X)$. External universal quantifiers have been omitted and variables of sorts *List* are in upper-case.

Framework $\mathcal{LIST}(X)$;

IMPORT:	$\mathcal{NAT}$;	
SORTS:	$Nat, X, List$;	
FUNCTIONS:	nil	$: \rightarrow List$;
	$.$	$: (X, List) \rightarrow List$;
	$nocc$	$: (X, List) \rightarrow Nat$;
RELATIONS:	$elemi$	$: (List, Nat, X)$;
AXIOMS:	$\neg nil = a.B \wedge (a_1.B_1 = a_2.B_2 \rightarrow a_1 = a_2 \wedge B_1 = B_2))$;	
	$nocc(x, nil) = 0$;	
	$a = b \rightarrow nocc(a, b.L) = nocc(a, L) + 1$;	
	$\neg a = b \rightarrow nocc(a, b.L) = nocc(a, L)$;	
	$elemi(L, 0, a) \leftrightarrow \exists B\,(L = a.B)$;	
	$elemi(L, s(i), a) \leftrightarrow \exists b, B\,(L = b.B \wedge elemi(B, i, a))$;	

For every closed instance $\mathcal{LIST}(X :: \mathcal{G})$, an isoinitial model of $\mathcal{LIST}(X :: \mathcal{G})$ is an expansion of the isoinitial models of $\mathcal{NAT}$ and $\mathcal{G}$, and is the usual structure (the term model generated by the constructors nil and $\cdot$) of lists of elements of X. $nocc(a, L)$ gives the number of occurrences of a in L. $elemi(L, i, a)$ means a occurs at position i in L.

For example, let $\mathcal{INT}$ be an axiomatisation of integers, which we omit here for conciseness. With $\mathcal{NAT}$ and $\mathcal{LIST}(Int :: \mathcal{INT})$ we have completed the construction of $\mathcal{SET}(Int)$. Note that, in the isoinitial model of $\mathcal{SET}$, *Set* is (interpreted as) the set of finite sets of integers.

Note, however, that $\mathcal{SET}$ is different from $\mathcal{NAT}$ and $\mathcal{LIST}(Int)$ in that the constructor $\{\ \}$ of $\mathcal{SET}$ does not satisfy the freeness axiom $\{X\} = \{Y\} \rightarrow X = Y$. This just means that in $\mathcal{SET}$ sets are represented by lists whose elements may appear in any order and may contain duplicates.

2.2 Specifications

In a given framework, we can add axioms for (total computable) functions and (decidable) relations. This mechanism allows us to define, or *specify*, new functions and relations.

We shall use two kinds of specifications: *if-and-only-if specifications* and *conditional specifications*. The former can be used to define new relations and to expand the language of the framework $\mathcal{F}$, whilst the latter are only used to specify different program behaviours and do not expand the language of $\mathcal{F}$.

An *if-and-only-if specification* S of a new relation r in $\mathcal{F}$ consists of a definition of the form $r(x) \leftrightarrow R(x)$ where $R(x)$ is any formula of the language of $\mathcal{F}$.

An *if-and-only-if* specification is therefore an *explicit* (non-recursive) definition. In the isoinitial model of $\mathcal{F}$, the meaning of the new symbol is completely determined by the definition, and we get a *unique* expanded model that interprets all the old symbols as before and the new one according to its definition. In other words, a specified relation has a unique interpretation in the isoinitial model of $\mathcal{F}$.

The model-theoretic relation between an *if-and-only-if* specification S of a relation r and a program P to compute r is illustrated in Figure 1, where the interpretation of r in I is that given by its explicit definition.

Explicit definitions can be used also to *expand* the specification language by new useful abbreviations. For example, the language of the kernel of the framework $\mathcal{LIST}$ for lists can be expanded by the specification of the usual predicates on lists, like:

$$mem(x, L) \leftrightarrow nocc(x, L) > 0$$
$$length(L, n) \leftrightarrow \forall i(\exists elemi(L, i, x) \leftrightarrow i < n)$$

However, we must use only *adequate* definitions, that is definitions which give rise to expansions of the isoinitial model (of $\mathcal{F}$) that are also isoinitial. In this connection, a useful criterion for adequacy is the existence of a totally correct program for the specified relation such that its completion can be proved in the framework. This means that we can actually use program synthesis to enrich the specification language by adequate definitions ([9]).

To return to the subset example, in $\mathcal{SET}$ we can define the subset relation $\subseteq$ in the usual way:

$$A \subseteq B \leftrightarrow \forall x.\,(x \in A \rightarrow x \in B) \tag{Q2}$$

Since we represent sets by lists, and our programs will compute on lists, we specify *sublist* as follows:

$$sublist(X, Y) \leftrightarrow \{X\} \subseteq \{Y\} \tag{Q3}$$

Note that (Q3) corresponds to the informal specification S1.

From the point of view of actually specifying programs, if-and-only-if specifications are often too restrictive and inflexible, and we would prefer weaker forms of specifications that admit multiple interpretations, corresponding to different program behaviours that are all correct for the problem at hand. To this end, we use *conditional specifications*. For instance, we could specify the sublist relation as follows:

$$IC(Y) \rightarrow (sublist(X, Y) \leftrightarrow OC(X) \wedge \{X\} \subseteq \{Y\}) \tag{Q4}$$

That is, we want $sublist(X, Y)$ to coincide with the *post-condition* $OC(X) \wedge \{X\} \subseteq \{Y\}$, where OC is an *output condition* stating that X does not contain duplicates, only if Y satisfies an *input-* or *pre-condition* IC stating that Y does not contain duplicates. Whenever the pre-condition IC does not hold, however, the post-condition, in particular the output condition OC, need not be satisfied.

Thus, unlike an if-and-only-if specification, a relation defined by a conditional specification C has many interpretations,[4] and we define correctness as follows:

> A program P is a *correct implementation* of a conditional specification C of a relation r iff the interpretation of r in the minimum Herbrand model H of P is (isomorphic to) one of the interpretations that satisfy C.

Thus a conditional specification may have different correct implementations.

What we have seen for specifications in a closed frameworks extends to parametric frameworks, where everything works, in a parameterized way, in all the closed instances. We omit the details for lack of space.

[4] Technically, there are many expansions of the isoinitial model of the framework.

2.3 Programs

Now we look at some examples of subset programs in $\mathcal{SET}$, and establish their model-theoretic, or *semantic correctness* by comparing their minimum Herbrand models with (the meaning of *sublist* stated by) the isoinitial model of $\mathcal{SET}$. We will consider both the if-and-only-if specification (Q3) and the conditional specification (Q4) of *sublist*.

We will define logic programs and their minimum Herbrand models in the usual way, and use sets only in specifications. Relations on sets will be linked to corresponding relations on lists via specifications like (Q3) or (Q4) in the previous section.

In general, as stated by Figure 1, for an if-and-only-if specification S of a relation r, in order to establish the correctness with respect to S of a program P (with minimum Herbrand model H) in a framework $\mathcal{F}$ (with isoinitial model I), we need to show that H is isomorphic to the restriction of I to the signature of P. For this purpose, we can make use of the following property:[5]

> In a framework $\mathcal{F}$ with isoinitial model I, if a program P with minimum Herbrand model H existentially terminates,[6] and we can prove its completion in $\mathcal{F}$, then H is isomorphic to the restriction of I to the signature of P.

For example, in $\mathcal{SET}$, we can show that the following program P1' is semantically correct with respect to (Q3):

$$\begin{array}{rcl} sublist(nil, Z) & \leftarrow & \\ sublist(x.Y, Z) & \leftarrow & mem(x, Z), sublist(Y, Z) \\ mem(x, x.Z) & \leftarrow & \\ mem(x, y.Z) & \leftarrow & mem(x, Z) \end{array} \qquad \text{(P1')}$$

since P1' existentially terminates and its completion can be proved in $\mathcal{SET}$.[7]

To avoid redundant answers, we need a better representation of sets than lists with duplicates. We will now show how we can use the conditional specification (Q4) to introduce input and output conditions that correspond to better list representations of sets. More importantly, we will explain in general how we can establish the semantic correctness of a program P with respect to a conditional specification C, i.e. how to compare the minimum Herbrand model of P with the many interpretations (in I) that satisfy C.

First, it is easy to see that a conditional specification can always be given as a pair of implications. In the case of (Q4), these are:

$$\begin{array}{l} sublist(X, Y) \rightarrow \neg IC(Y) \vee (OC(X) \wedge \{X\} \subseteq \{Y\}) \\ sublist(X, Y) \leftarrow IC(Y) \wedge (OC(X) \wedge \{X\} \subseteq \{Y\}) \end{array} \qquad (5)$$

That is, $sublist(X, Y)$ is any relation that is contained in the bigger relation *big*:

$$big(X, Y) \leftrightarrow \neg IC(Y) \vee (OC(X) \wedge \{X\} \subseteq \{Y\})$$

and that contains the smaller relation *small*:

$$small(X, Y) \leftrightarrow IC(Y) \wedge (OC(X) \wedge \{X\} \subseteq \{Y\})$$

[5] For brevity, we omit the proof here.

[6] That is, for every ground goal G, either $P \cup \{\leftarrow G\}$ finitely fails or the corresponding *SLD*-tree contains at least one success node.

[7] Again we omit the proof.

Thus a program P for computing *sublist* is a correct implementation with respect to (5) if it computes a relation that contains *small* and is contained in *big*. This is illustrated in Figure 2.

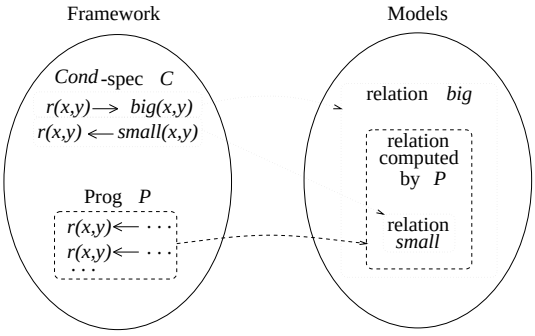

Figure 2: Model-theoretic relation between *big*, *small*, and P.

To compare the minimum Herbrand model of P with such relations, we proceed as follows:

(a) We assume $sublist(X,Y) \leftrightarrow big(X,Y)$, and we try to prove in $\mathcal{SET}$ the clauses of P. If a proof exists, then P computes a *subrelation* of *big*.[8]

(b) We assume $sublist(X,Y) \leftrightarrow small(X,Y)$, and we try to prove in $\mathcal{SET}$ the *only-if* part of the completion of P. If a proof exists and P existentially terminates, then *small* is contained in the relation computed by P.[9]

Therefore successful proofs in (a) and (b) will mean the semantic correctness of the implementation of P with respect to the chosen conditional specification.

As an example, consider the specification (Q4) where both the input and output conditions IC and OC are the relation $nocc_1$:[10]

$$nocc_1(L) \leftrightarrow \forall a.\, nocc(a,L) \leq 1$$

and consider the program P2':

$$\begin{array}{lcl} sublist(nil,Y) & \leftarrow & \\ sublist(x.X,Y) & \leftarrow & select(x,Y,H), sublist(X,H) \end{array} \qquad \text{(P2')}$$

where we use some *select* relation instead of membership.

According to (a), we assume $sublist(X,Y) \leftrightarrow big(X,Y)$, and we try to prove the clauses of P2'. The proofs go on as follows:

$$\begin{array}{lcl} sublist(nil,Y) & \leftrightarrow & \neg nocc_1(Y) \vee (nocc_1(nil) \wedge \{nil\} \subseteq \{Y\}) \\ & \leftrightarrow & true \\ sublist(x.X,Y) & \leftrightarrow & \neg nocc_1(Y) \vee (nocc_1(x.X) \wedge \{x.X\} \subseteq \{Y\}) \\ & \leftarrow & select(x,Y,H) \wedge (\neg nocc_1(H) \vee (nocc_1(X) \wedge \{X\} \subseteq \{H\})) \\ & \leftrightarrow & select(x,Y,H) \wedge sublist(X,H) \end{array}$$

[8] Indeed, in this case, the clauses of P are theorems of $\mathcal{SET} \cup \{\forall(sublist(X,Y) \leftrightarrow big(X,Y))\}$.

[9] A proof can be found in [13].

[10] $nocc_1(L)$ means L does not contain duplicates.

In the second proof, *select* can be chosen in different ways. For example, let us define $select_{big}$ by

$$select_{big}(x, Y, H) \leftrightarrow \neg nocc_1(Y) \vee (\{x.H\} \subseteq \{Y\} \wedge nocc_1(x.H))$$

It is easy to see that our proof can be completed for any *select* such that

$$select(x, Y, H) \rightarrow select_{big}(x, Y, H) \tag{6}$$

can be proved in the framework.

Now condition (a) only guarantees that the computed relation is contained in *big*. Indeed, many *select* relations satisfy (6), for example:

$$select(x, Y, H) \leftrightarrow x \in \{Y\} \wedge H = nil$$

However, with this choice of *select* the computed relation $sublist(X, Y)$ would be true only for some of the sublists X of Y.

Thus we have to compare the relation computed by P2' with *small*. We proceed according to (b), i.e. we assume $sublist(X, Y) \leftrightarrow small(X, Y)$, and try to prove the *only-if* part of the completion of P2', namely:

$$sublist(A, Y) \rightarrow A = nil \vee \exists x, X, H.\, (A = x.X \wedge select(x, Y, H) \wedge sublist(X, H))$$

Since $\forall A : List\, (A = nil \vee \exists x, X.\, A = x.X)$ is a theorem of the framework, we can give the proof by cases. The case $(A = nil)$ is obvious. The case $(A = x.X)$ is:

$$sublist(x.X, Y) \rightarrow \exists H.\, (select(x, Y, H) \wedge sublist(X, H))$$

The proof for this is:

$$\begin{array}{lcl} sublist(x.X, Y) & \leftrightarrow & nocc_1(Y) \wedge (nocc_1(x.X) \wedge \{x.X\} \subseteq \{Y\}) \\ & \rightarrow & \exists H.\, (select(x, Y, H) \wedge (nocc_1(H) \wedge (nocc_1(X) \wedge \{X\} \subseteq \{H\}))) \\ & \leftrightarrow & \exists H.\, (select(x, Y, H) \wedge sublist(X, H)) \end{array} \tag{7}$$

Let us define

$$select_{small}(x, Y, H) \quad \leftrightarrow \quad nocc_1(Y) \wedge \exists A, B\, (Y = concat(A, x.B) \wedge H = concat(A, B))$$

It is easy to see that, if

$$select_{small}(x, Y, H) \rightarrow select(x, Y, H) \tag{8}$$

can be proved, then so can (7).

Therefore, if we choose a *select* that satisfies both (6) and (8), then (a) and (b) hold and semantic correctness is guaranteed. For example, we can choose the following conditional specification:

$$nocc_1(Y) \rightarrow (select(x, Y, H) \leftrightarrow \exists A, B\, (Y = concat(A, x.B) \wedge H = concat(A, B))) \tag{9}$$

Now we can go on and derive the correct clauses for select. For lack of space we do not show this.

Without clauses for $select(x, Y, H)$, the program P2' cannot compute. However, we can say that it is correct. Indeed, for every program Q correct with respect to the above specification of *select*, P2' will be composed correctly with Q. We say that P2' is a correct open program, with open predicate *select*. Open programs allow us to treat correctness in parametric frameworks, where the framework parameters can be used in programs without giving any code for them. Again, we will not discuss this issue here, for reasons of space.

3 Conclusion

We have examined the model-theoretic relationship between logic programs and specifications, in the context of the underlying problem domain, or framework. The rôle of the framework is fundamental, since it provides the semantic underpinning for both specifications and programs, and it marks the main difference between our approach and the view that "logic programs are obviously correct since they are logical assertions". The latter is not quite satisfactory, since the meaning of correctness must be defined in terms of something other than logic programs themselves (see e.g. [5, p. 410]).

We have shown that in a framework, we can define correctness in a semantic way, either informally or formally. Informal correctness allows us to relate programs to their informal specifications. Formal correctness involves full first-order logical theories, and is defined in terms of model theory. In either case, we have shown that the relation between specifications and programs is always well-defined, and we can properly distinguish between them on the basis of the correctness relation.

A particular strength of our model-theoretic approach to correctness is its ability to handle open specifications and programs. In general we can use our approach to specify what are called generic classes in object-oriented programming, and define correctness of open methods with respect to to their specifications when they are inherited (see [11]). This kind of correctness, which we call steadfastness ([13]), that is preserved through inheritance hierarchies seems to be a promising tool for formal object-oriented software engineering.

Our semantics for open programs, their correctness, and the correctness of their composition is also different from other model-theoretic approaches such as [3, 2]. Again, the main difference lies in our use of a framework that allows us to reason about the correctness of program composition. This enables us to define steadfastness, which is the basis of formal correctness of object-oriented programs.

References

[1] A. Bertoni, G. Mauri and P. Miglioli. On the power of model theory in specifying abstract data types and in capturing their recursiveness. *Fundamenta Informaticae* VI(2):127–170, 1983.

[2] A. Bossi, M. Gabbrielli, G. Levi and M.C. Meo. A compositional semantics for logic programs. *Theoretical Computer Science* 122:3-47, 1994.

[3] A. Brogi, E. Lamma and P. Mello. Composing open logic programs. *J. Logic and Computation* 3(4):417-439, 1993.

[4] Y. Deville and K.-K. Lau. Logic program synthesis. *J. Logic Programming* 19,20:321–350, 1994. Special issue: Ten years of logic programming.

[5] J.H. Gallier. *Logic for Computer Science: Foundations for Automatic Theorem Proving*. Harper and Row, 1986.

[6] P.M. Hill and J.W. Lloyd. *The Gödel Programming Language*. MIT Press, 1994.

[7] W. Hodges. Logical features of Horn clauses. In D.M. Gabbay, C.J. Hogger, and J.A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, Vol 1, pages 449–503, Oxford University Press, 1993.

[8] R.A. Kowalski. The relation between logic programming and logic specification. In C.A.R. Hoare and J.C. Shepherdson, editors, *Mathematical Logic and Programming Languages*, pages 11–27, Prentice-Hall, 1985.

[9] K.-K. Lau and M. Ornaghi. On specification frameworks and deductive synthesis of logic programs. In L. Fribourg and F. Turini, editors, *Proc. LOPSTR 94 and META 94*, *Lecture Notes in Computer Science* 883, pages 104–121, Springer-Verlag, 1994.

[10] K.-K. Lau, M. Ornaghi and S.-Å. Tärnlund. The halting problem for deductive synthesis of logic programs. In P. van Hentenryck, editor, *Proc.* 11^{th} *Int. Conf. on Logic Programming*, pages 665–683, MIT Press, 1994.

[11] K.-K. Lau and M. Ornaghi. Towards an object-oriented methodology for deductive synthesis of logic programs. In M. Proietti, editor, *Proc. LOPSTR 95*, *LNCS* 1048:152–169, Springer-Verlag, 1996.

[12] K.-K. Lau and M. Ornaghi. A formal approach to deductive synthesis of constraint logic programs. In J.W. Lloyd, editor, *Proc. 1995 Int. Logic Programming Symp.*, pages 543–557, MIT Press, 1995.

[13] K.-K. Lau, M. Ornaghi and S.-.Å. Tärnlund. Steadfast logic programs. Submitted to *J. Logic Programming*.

[14] K.-K. Lau, M. Ornaghi. The Relationship between Logic Programs and Specifications — The Subset Example Revisited. To appear in *J. Logic Programming*.

[15] P. Miglioli, U. Moscato and M. Ornaghi. Abstract parametric classes and abstract data types defined by classical and constructive logical methods. *J. Symb. Comp.* 18:41–81, 1994.

[16] J.C. Shepherdson. Negation in logic programming. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 19–88. Morgan Kaufmann, 1988.

[17] M. Wirsing. Algebraic specification. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 675–788. Elsevier, 1990.