THE RELATIONSHIP BETWEEN LOGIC PROGRAMS AND SPECIFICATIONS — THE SUBSET EXAMPLE REVISITED

KUNG-KIU LAU AND MARIO ORNAGHI

▷ 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.

Address correspondence to Kung-Kiu Lau, Department of Computer Science, University of Manchester, Oxford Road, Manchester M13 9PL, United Kingdom, email: kung-kiu@cs.man.ac.uk, or Mario Ornaghi, Dipartimento di Scienze dell'Informazione, Universita' degli studi di Milano, Via Comelico 39/41, Milano, Italy, email: ornaghi@hermes.mc.dsi.unimi.it

Part of this work was done during the first author's visits to Fachgebiet Intellektik, Fachbereich Informatik, Technische Hochschule Darmstadt, Germany, and to Computing Science Department, Uppsala University, Sweden, supported by the European Union HCM project on Logic Program Synthesis and Transformation, contract no. 93/414.

THE JOURNAL OF LOGIC PROGRAMMING

655 Avenue of the Americas, New York, NY 10010 0743-1066/94/$7.00

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

We will show that this approach makes sense not only for (closed) informal and formal frameworks (Sections 2 and 3 respectively), but also for *open*, or parameterised frameworks (Section 4). Throughout the paper, we will use the *subset* relation as an illustration. In order to be brief and to avoid complex notational and theoretical details, we will give an informal presentation. Finally, we shall consider only definite programs, although our approach extends to normal programs.

2. Informal 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.

In this section, we illustrate the correctness relationship between S and P in an informal framework, by considering several Prolog programs for *subset* together with their informal specifications. These programs (and their specifications) are given by O'Keefe in [15] to compute `subset(Small, Large)` correctly according to different informal specifications, in order to find `Small` for a given `Large`, where `Small` and `Large` are lists representing sets in various ways.

O'Keefe's first Prolog[1]program P1 for `subset/2` 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:[2]

> "`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)

[1] That is Prolog without extension to support sets.

[2] We will paraphrase O'Keefe's specifications slightly for uniformity, whilst retaining their original meaning.

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:

$$? - \mathtt{subset}([\mathtt{A}, \mathtt{B}], [1, 2, 3]). \tag{Q1}$$

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.
```

O'Keefe's second Prolog program P2 represents sets by unordered lists without duplicates:

```
      select(X, [X|R], R).
   select(X, [H|T], [H|R])  :-  select(X, T, R).

            subset([], _).                                  (P2)
    subset([X|Xs], Set)     :-  select(X, Set, Set1),
                                subset(Xs, Set1).
```

The (paraphrased) informal specification S2 of P2 is the following:

> "Whenever `Large` is a list without duplicates, `subset(Small, Large)` is to be true if: `Small` is a list without duplicates; the set `Small` represents is a subset of the set that `Large` represents." (S2)

Note that unlike S1, now S2 contains a *pre-condition* on (the input list) `Large` that it does not contain duplicates. This means that the truth of `subset(Small, Large)` is defined only for values of `Large` that satisfy the pre-condition.

In general, whenever a specification S contains pre-conditions, we shall follow the pre-post-condition style of program development, and require a program P to satisfy S only when the pre-conditions in S are met:

> If S contains pre-conditions, then P is a *correct implementation* of S if P is correct with respect to S whenever the pre-conditions in S are met.

It is easy to see that P2 is a correct implementation of S2. For example, for the query Q1, P2 gives the six expected answers:

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

However, whenever the pre-condition in S2 is not satisfied, P2 does not satisfy S2. Indeed P2 happily accepts `Large` containing duplicates and produces `Small` containing duplicates.

The third Prolog program in [15] assumes that `Large` is sorted, and whenever this holds, the program yields a sorted `Small`:

```
        append([], L, L).
append([H|T], L, [H|R])  :-  append(T, L, R).

          subset([], _).                                  (P3)
    subset([X|Xs], Set)  :-  append(_, [X|Set1], Set),
                             subset(Xs, Set1).
```

It is easy to see that P3 is a correct implementation of the following specification:

> "Whenever `Large` is a sorted list without duplicates, `subset(Small, Large)` is to be true if: `Small` is a sorted list without duplicates; the set `Small` represents is a subset of the set that `Large` represents." (S3)

For example, for the query Q1, P3 gives the three expected answers:

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

Although it also accepts unsorted inputs, P3 does not satisfy S3 in these cases.

Finally, O'Keefe's fourth Prolog program is:

```
          subset([], _).
    subset([X|Xs], Set)  :-  select(X, Set, Set1),
                             subset(Xs, X, Set1).
                                                          (P4)
       subset([], _, _).
 subset([X|Xs], Z, Set)  :-  select(X, Set, Set1),
                             X@>Z, subset(Xs, X, Set1).
```

We can easily see that P4 is correct with respect to the following specification:

> "`subset(Small, Large)` is to be true if: `Large` is a list possibly containing duplicates; `Small` is a sorted list without duplicates; the set `Small` represents is a subset of the set that `Large` represents." (S4)

P4 will always give a sorted `Small` without duplicates regardless of whether `Large` is sorted or not, and whether `Large` contains duplicates or not. Thus the meaning of `subset` is the same in the minimum Herbrand model of P4 as in S4.

These examples illustrate 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 every S, i.e. the full meaning of S is defined only in $\mathcal{F}$. For instance, S3 and S4 even need list predicates for expressing *a list is ordered*, etc. On the other hand, the P's do not and cannot completely axiomatise such abstract data types and predicates. 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.

3. Model-theoretic Correctness

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 relation r,[3]and P a Horn theory whose language contains the relation r. Both $\mathcal{F}$ and P have intended models: the intended model of $\mathcal{F}$ is its *isoinitial model* I, while that of P is its minimum Herbrand model H. 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 isoinitial model I of S are isomorphic when restricted to the relations r defined by S.

This is illustrated in Figure 3.1.[4] The new symbol r defined in S has an interpre-

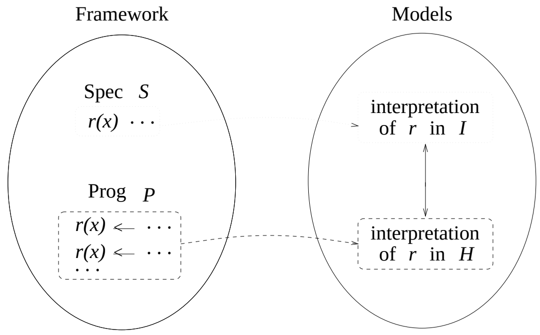

FIGURE 3.1. Model-theoretic relation between S and P.

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

3.1. Frameworks

A framework is an axiomatisation of a problem domain. It has (i) a (many-sorted) signature of sort symbols, function symbols,[5]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:

[3]Both S and r may be sets.

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

[5]Function symbols include constant symbols.

Framework $\mathcal{SET}(Int)$;

IMPORT: $\mathcal{LIST}(Int)$;

SORTS: $Int, List, Set$;

FUNCTIONS: $\{\,\} : List \to Set$;

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 axiomatises, 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 characterised 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 $\mathcal{F}$.[6]

Such a model is also an initial model,[7] 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 axiomatisation, then there 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 axiomatising a type T, we proceed incrementally. We start with a small framework $\mathcal{F}_0$ with an obvious isoinitial model I_0.

[6] Thus an isoinitial model is a *generic* model (see [16]).

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

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 $\mathcal{NAT}$ axiomatises the abstract data type of Peano arithmetic:

Framework $\mathcal{NAT}$;

SORTS: Nat;

FUNCTIONS: $0 \;:\; \to Nat$;
$s \;:\; Nat \to Nat$;
$+, * : (Nat, Nat) \to Nat$;

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

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, $\mathcal{NAT}$ axiomatises Peano arithmetic.[8]

Similarly, we can construct (part of) the framework $\mathcal{LIST}(Int)$:[9]

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

IMPORT: $\mathcal{NAT}, \mathcal{INT}$;

SORTS: $Nat, Int, List$;

FUNCTIONS: $nil \;:\; \to List$;
$. \;:\; (Int, List) \to List$;
$nocc : (Int, List) \to Nat$;

RELATIONS: $elemi : (List, Nat, Int)$;
...

[8] We have also included the axiom schema of induction for reasoning purposes.

[9] External universal quantifiers have been omitted. Variables of sorts *Int* and *List* are in lower-case and upper-case respectively.

$$\begin{array}{ll}\text{AXIOMS:} & \neg nil = a.B \wedge (a_1.B_1 = a_2.B_2 \rightarrow a_1 = a_2 \wedge B_1 = B_2));\\ & nocc(x, nil) = 0;\\ & a = b \rightarrow nocc(a, b.L) = nocc(a, L) + 1;\\ & \neg a = b \rightarrow nocc(a, b.L) = nocc(a, L);\\ & elemi(L, 0, a) \leftrightarrow \exists B\,(L = a.B);\\ & elemi(L, s(i), a) \leftrightarrow \exists b, B\,(L = b.B \wedge elemi(B, i, a));\\ & \ldots\end{array}$$

where $\mathcal{INT}$ is the (predefined) framework for the integer type *Int*.

$\mathcal{LIST}(Int)$ is essentially the standard axiomatisation of lists (of integers). The function $nocc(a, L)$ gives the number of occurrences of a in L. $elemi(L, i, a)$ means a occurs at position i in L. *nocc* and *elemi* enable us to define other useful list predicates.

It can be shown that an isoinitial model of $\mathcal{LIST}(Int)$ is an expansion of the isoinitial model of $\mathcal{NAT}$, and is the usual structure (the term model generated by the constructors *nil* and $\cdot$) of lists of integers.

With $\mathcal{NAT}$ and $\mathcal{LIST}(Int)$ 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.

3.2. Specifications

In a given framework, we can add axioms for (total computable) functions and (decidable) relations in the manner we have just described. This mechanism allows us to define or specify new functions and relations. In $\mathcal{LIST}(Int)$, for instance, we can define the the usual functions for length and concatenation, and the usual 'membership', 'length', 'concatenation', and 'permutation' relations. This illustrates how the language of a framework can be expanded by new specifications.

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 $\mathcal{F}$.

Thus explicit definitions can be used to *expand* the specification language by new useful abbreviations, such as the aforementioned ones for $\mathcal{LIST}$. 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 ([9]).[10]

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 3.2.

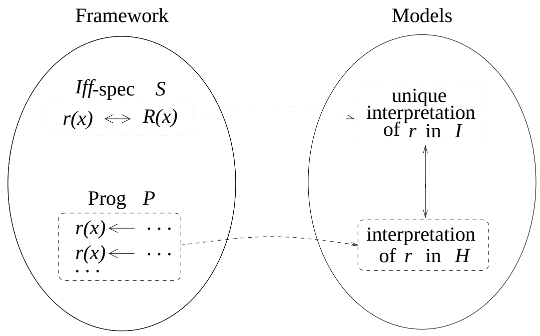

FIGURE 3.2. Model-theoretic relation between *if-and-only* specification S and P.

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

$$A \subseteq B \leftrightarrow \forall x.\, (x \in A \rightarrow x \in B) \tag{1}$$

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

$$\mathit{sublist}(X, Y) \leftrightarrow \{X\} \subseteq \{Y\} \tag{2}$$

Note that (2) corresponds to S1 of Section 2.

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:

$$IC(Y) \rightarrow (\mathit{sublist}(X, Y) \leftrightarrow OC(X) \wedge \{X\} \subseteq \{Y\}) \tag{3}$$

That is, we want $\mathit{sublist}(X, Y)$ 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 (see Section 2).

Thus, unlike an if-and-only-if specification, a relation defined by a conditional specification C has many interpretations,[11]and we define correctness as follows:

[10] This means that we can actually use program synthesis to enrich the specification language by adequate definitions.

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

> 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.

This is illustrated in Figure 3.3.

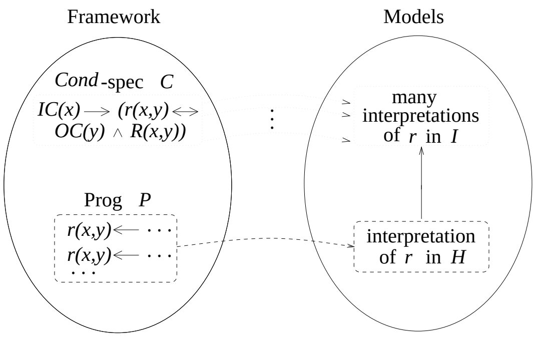

FIGURE 3.3. Model-theoretic relation between *conditional* specification C and P.

Thus a conditional specification may have different correct implementations. For example, one of the correct implementations of (3) is P2 in Section 2 (as we will show later), when the pre- and post-conditions in (3) correspond to those in S2.

3.3. 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 (2) and the conditional specification (3) 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 (2) or (3) in the previous section.

In general, as stated by Figure 3.2, 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:[12]

> In a framework $\mathcal{F}$ with isoinitial model I, if a program P with minimum Herbrand model H existentially terminates,[13] and we can prove its completion in $\mathcal{F}$, then H is isomorphic to the restriction of I to the signature of P.

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

[13] 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.

For example, in $\mathcal{SET}$, we can show that the following program P1' is semantically correct with respect to (2):

$$\begin{aligned} sublist(nil, Z) &\leftarrow \\ sublist(x.Y, Z) &\leftarrow mem(x, Z), sublist(Y, Z) \\ mem(x, x.Z) &\leftarrow \\ mem(x, y.Z) &\leftarrow mem(x, Z) \end{aligned} \tag{P1'}$$

since P1' existentially terminates and its completion can be proved in $\mathcal{SET}$.[14]

Now P1' of course corresponds to P1 in Section 2, and like P1, P1' produces redundant answers.

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 (3) 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 (3), these are:

$$\begin{aligned} sublist(X, Y) &\rightarrow \neg IC(Y) \vee (OC(X) \wedge \{X\} \subseteq \{Y\}) \\ sublist(X, Y) &\leftarrow IC(Y) \wedge (OC(X) \wedge \{X\} \subseteq \{Y\}) \end{aligned} \tag{5}$$

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*:

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

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 3.4.

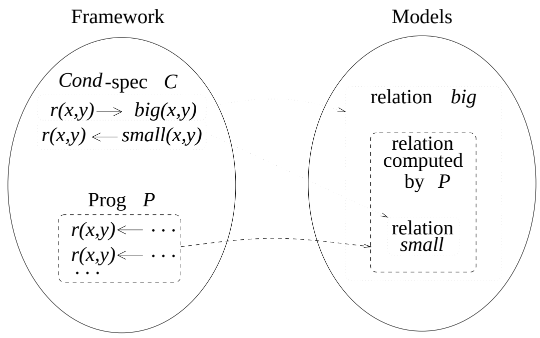

FIGURE 3.4. Model-theoretic relation between *big*, *small*, and P.

[14] Again we omit the proof.

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*.[15]

(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.[16]

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 (3) where both the input and output conditions IC and OC are the relation $nocc_1$:[17]

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

and consider the program P2':

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

where we use some *select* relation instead of membership, e.g. that in P2 in Section 2.

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:

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

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 $select(X,Y) \leftrightarrow small(X,Y)$, and try to prove the

[15] Indeed, in this case, the clauses of P are theorems of $\mathcal{SET} \cup \{\forall(sublist(X,Y) \leftrightarrow big(X,Y))\}$.
[16] A proof can be found in [13].
[17] $nocc_1(L)$ means L does not contain duplicates.

only-if part of the completion of P2', namely:

$$\begin{aligned} sublist(A,Y) \rightarrow A = nil \vee \exists x, X, H.\, (A = x.X \wedge select(x,Y,H) \wedge \\ sublist(X,H)) \end{aligned}$$

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 $sublist(nil,Y) \rightarrow nil = nil \vee \ldots$, which 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:

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

Let us define

$$\begin{aligned} &prec(x,y,L) && \leftrightarrow \exists i,j : Nat.\, (elemi(L,i,x) \wedge elemi(L,j,y) \wedge i < j) \\ &H \preceq Y && \leftrightarrow \forall a,b : Int.\, (prec(a,b,H) \rightarrow prec(a,b,Y)) \\ &select_{small}(x,Y,H) && \leftrightarrow nocc_1(Y) \wedge (\{Y\} = \{x.H\} \wedge nocc_1(x.H) \wedge H \preceq Y) \end{aligned}$$

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:

$$nocc_1(Y) \rightarrow (select(x,Y,H) \leftrightarrow \{x.H\} = \{Y\} \wedge nocc_1(x.H) \wedge H \preceq Y) \tag{9}$$

Now we can prove that the clauses for *select* in P2 in Section 2 are a correct implementation of (9).

The other programs in Section 2, viz. P3 and P4, can be specified by conditional specifications using the following if-and-only-if specification of the ordering relation:

$$ord(L) \leftrightarrow \forall i,x,y.\, (elemi(L,i,x) \wedge elemi(L,s(i),y) \rightarrow x < y)$$

P3 corresponds to the choice $IC(L) \leftrightarrow ord(L)$ and $OC(L) \leftrightarrow ord(L)$, while P4 corresponds to $OC(L) \leftrightarrow ord(L)$. We can show their correctness in the way we have illustrated above.

4. Open Frameworks, Specifications and Programs

A framework $\mathcal{F}$ is *open*, if it contains *parameters*, that is sort or relation symbols which are not completely characterised by the axioms of the framework. The incomplete axioms associated with a parameter are called *parameter axioms*. Unlike axioms that define symbols completely, parameter axioms only partially characterise the parameters, i.e. they define only properties to be satisfied by the actual parameters. The latter are usually symbols defined in another framework $\mathcal{G}$, and therefore parameter passing allows framework *composition*.

Specifications and programs in an open framework $\mathcal{F}$ may be open, i.e. they may contain the parameters of $\mathcal{F}$. When two frameworks are composed, their programs will be composed too. Open programs and program composition have been introduced in logic programming, e.g. in [3, 2]. Again, here the main difference in our approach is the use of frameworks and specifications. This enables us to define an important notion of correctness: when two frameworks are composed, together with their correct programs, correctness is automatically preserved, i.e. the resulting programs are guaranteed to be correct in the composite framework. This kind of correctness is applicable to *object-oriented* programs [11].

In this section, we will briefly explain model-theoretic correctness in open frameworks, as an extension of correctness in closed frameworks, and we will show that it is better to use isoinitial semantics for both frameworks and programs.

An open program or an open framework has a *class* of models. We will first explain how isoinitial semantics can capture the class of models of an open program P. As we pointed out in Section 3.1, an isoinitial model that is reachable is also an initial model. Now it should be obvious that a minimum Herbrand model is an initial model. Here we shall make use of these facts, and endow P with isoinitial semantics in such a way that, when an isoinitial model exists, then it is the minimum Herbrand model of P. This little departure from standard logic programming not only gives us semantic homogeneity, but, more importantly, it also allows us to reason about open programs.

By way of illustration, consider the following simple open program *car_prices*:

$$
\begin{array}{l}
\mathit{diag}^+(\leq) \\
\mathit{car}(\mathit{spider}). \\
\mathit{car}(\mathit{elan}). \\
\mathit{cheaper}(A,B) \leftarrow \mathit{price}(A,P), \mathit{price}(B,Q), P \leq Q
\end{array}
$$

where the positive diagram $\mathit{diag}^+(\leq) = \{0 \leq 0, 0 \leq 1, 1 \leq 1, \ldots\}$ corresponds to the predefined data type of integers with ordering $\leq$.

Clearly, the information on (the parameter) *price* is missing. Nevertheless the program *car_prices* has a minimum Herbrand model H_{bad} in which $\mathit{cheaper}(X,Y)$ is always false. Therefore, under minimum model semantics, H_{bad} would be the intended model of *car_prices*. This would be completely unsatisfactory, since the meaning of *price* is supposed to be open but H_{bad} interprets it as false. In order to handle open symbols like *price* properly, we need a semantics where no intended model exists when information on these symbols is incomplete.

Isoinitial semantics meets this requirement,[18] if we consider the *open completion* of (open) programs. In our example, the open completion $\mathit{Ocomp}(\mathit{car\_prices})$ is:

$$
\begin{array}{l}
\mathit{free}(\mathit{spider}, \mathit{elan}, 0, 1, 2, \ldots) = \{\neg \mathit{spider} = \mathit{elan}, \neg 0 = 1, \neg 0 = 2, \ldots\} \\
\mathit{diag}(\leq) = \{0 \leq 0, 0 \leq 1, 1 \leq 1, \neg 1 \leq 0, \neg 2 \leq 0, \neg 2 \leq 1, \ldots\} \\
\mathit{car}(X) \leftrightarrow X = \mathit{spider} \vee X = \mathit{elan} \\
\mathit{cheaper}(A,B) \leftrightarrow \exists P, Q\, (\mathit{price}(A,P) \wedge \mathit{price}(B,Q) \wedge P \leq Q)
\end{array}
$$

The open completion of a program contains:

- The freeness axioms for the constant and function symbols of the language (of the program), e.g. $\mathit{free}(\mathit{spider},\ \mathit{elan}, 0, 1, 2, \ldots)$.

- The complete diagram of any predefined predicates, e.g. $\mathit{diag}(\leq)$. This

[18] Isoinitial semantics gives a more abstract characterisation of open programs than other modifications of minimum model semantics, e.g. [3, 2].

diagram contains both positive and negative information.

- The completed definitions of the predicates defined by the program, e.g. *car* and *cheaper*.

The key difference between *Ocomp* and the standard completion *Comp* is that *Ocomp* does not contain any axioms for the open symbols, e.g. *price*. Thus, unlike in *Comp*, open symbols are not interpreted as false in *Ocomp*.

Note that *Ocomp*(*car_prices*) has a minimum Herbrand model, namely H_{bad}. It does not have an isoinitial model however. This corresponds to the fact that information on *price* is missing, i.e. we cannot use *car_prices* to prove anything about *price*. However, as soon as we supply a complete table of values for *price*, *Ocomp*(*car_prices*) has an isoinitial model. Thus by isoinitial semantics we can recognise that information is missing, and by completing this information in different ways, we can associate a whole class of models with an open program.

We can prove that, if a reachable isoinitial model of $Ocomp(P)$ exists, then it is isomorphic to the minimum Herbrand model of P. Therefore isoinitial semantics agrees (though it does not coincide) with minimum model semantics. We say that a program has an isoinitial model J iff its open completion has J as an isoinitial model.

Now it should be obvious how we can also associate a class of models with an open framework. For example, we can define an open framework for sets as lists of elements of an arbitrary sort *Elem* as follows:

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

(where $\mathcal{LIST}(Elem)$ can be constructed in an obvious way.)

Now *Elem* is open, and there are neither function symbols that use it nor parameter axioms for it. Therefore $\mathcal{SET}(Elem)$ has a class of models, of which the isoinitial model of $\mathcal{SET}(Int)$ is just an instance.

Each model of $\mathcal{SET}(Elem)$ can be seen as an isoinitial model of $\mathcal{C} \cup \mathcal{SET}(Elem)$, where $\mathcal{C}$ is any closed framework that axiomatises a particular interpretation $Elem^{\mathcal{C}}$ of *Elem*. We will call $\mathcal{C} \cup \mathcal{SET}(Elem)$ an *instance* of $\mathcal{SET}(Elem)$.

If all the instances of an open framework are closed frameworks, then we say that the open framework is *adequate* (see [9]).[19]For example, we can show that $\mathcal{SET}(Elem)$ is adequate. Without adequacy, framework composition and program correctness would be ill-defined and quite useless, and so we shall always use adequate open frameworks.

In an (adequate) open framework $\mathcal{F}$, the correctness relation between a(n open) specification S and an open program P can be defined in a similar manner to the way we have defined it for closed programs in closed frameworks. However, to do

[19]Informally, it means that all symbols, both open and closed, in a framework are (potentially) properly defined. This is possible through the use of parameter axioms.

so, we need to consider closed instances of P that correspond to closed instances of $\mathcal{F}$.

For example, for an open framework $\mathcal{F}$ with only open sort symbols, such as $\mathcal{SET}(Elem)$, we will consider *Herbrand instances* of $\mathcal{F}$ and P. For $\mathcal{SET}(Elem)$, Herbrand instances are instances $free(K) \cup \mathcal{SET}(Elem)$ where the closed framework $free(K)$ axiomatises the Herbrand universe $Elem^K$ generated by a suitable set K of constructors. Open programs in $\mathcal{SET}(Elem)$ can be closed in its Herbrand instances. For instance, in $\mathcal{SET}(Elem)$, the program P1' is open, since without constructors of *Elem* its open completion $Ocomp$(P1') does not have an isoinitial model. However, we can close P1' in a Herbrand instance $free(K) \cup \mathcal{SET}(Elem)$, by adding the constructors K to $Ocomp$(P1'). That is $Ocomp$(P1')$\cup free(K)$, which we will also call a Herbrand instance of P1', has an isoinitial model.

It should be obvious, in this simple example, that in *every* Herbrand instance $free(K) \cup \mathcal{SET}(Elem)$, the corresponding (Herbrand) instance of P1' is semantically correct with respect to (2). Thus we say that P1' is semantically correct in the open framework $\mathcal{SET}(Elem)$.

Similarly, the open program P2' is also semantically correct (with respect to (2)) in $\mathcal{SET}(Elem)$.

In general, we can informally define semantic correctness of a program P in an open framework $\mathcal{F}$ with only open sort symbols as follows:

> A program P is semantically correct (with respect to a specification S) in an open framework $\mathcal{F}$ iff in every Herbrand instance $free(K) \cup \mathcal{F}$ the corresponding (Herbrand) instance of P is semantically correct.

In other words, the model-theoretic relation between a(n open) specification S of a relation r and an open program P to compute r is as depicted in Figure 3.1, except that now we compare the interpretation of r in the isoinitial model I_K of the Herbrand instance $free(K) \cup \mathcal{F}$ of $\mathcal{F}$ with the interpretation of r in the isoinitial model H_K of the Herbrand instance $Ocomp(P) \cup free(K)$ of P, for *every* Herbrand instance (corresponding to a chosen set K of constructors) of $\mathcal{F}$ and P. Similarly, in Figures 3.2 and 3.3, we compare the isoinitial models of Herbrand instances of frameworks and programs.

Now, as we pointed out earlier, an open framework may also contain open relation symbols as parameters, together with parameter axioms that partially characterise them. For example, we may consider $\mathcal{LIST}(Elem, \lhd)$, where the parameter axioms for $\lhd$ only state that $\lhd$ is a total ordering relation. In this framework we can have open programs like P3 and P4 which assume a total ordering among list elements. In this situation, model-theoretic correctness is defined as before, but now Herbrand instances have to satisfy the total ordering axioms for the parameter $\lhd$, and open programs are closed by adding clauses for $\lhd$.

Finally, it is worthwhile to note that the proof techniques that we used in Section 3 to state model-theoretic correctness in closed framework can be extended to model-theoretic correctness in *adequate* open frameworks. To do so, we need only replace existential termination by parametric termination, which roughly speaking corresponds to termination of every instance of the program. For example, all the programs considered in this paper parametrically terminate, that is, they existentially terminate regardless of the domain of the elements and the total ordering on them.

5. 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.

Acknowledgements

We would like to thank the referees for their helpful comments and suggestions for improvement. Kung-Kiu Lau wishes to thank Wolfgang Bibel and Jonas Barklund for inviting him to Darmstadt and Uppsala respectively, and for their hospitality.

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. 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.

15. R.A. O'Keefe. What does subset mean? *ALP Newsletter* 8(3):10, August 1995.

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.