Forms of Logic Specifications: A Preliminary Study*

Kung-Kiu Lau[1] and Mario Ornaghi[2]

[1] Department of Computer Science
University of Manchester, Manchester M13 9PL, United Kingdom
kung-kiu@cs.man.ac.uk
[2] Dipartimento di Scienze dell'Informazione
Universita' degli studi di Milano, Via Comelico 39/41, Milano, Italy
ornaghi@dsi.unimi.it

Abstract. There is no universal agreement on exactly what form a specification should take, what part it should play in synthesis, and what its precise relationship with the specified program should be. In logic programming, the role of specification is all the more unclear since logic programs are often used as executable specifications. In this paper we take the view that specifications should be set in the context of the problem domain, which we call a *framework*. We conduct a preliminary study of two useful forms of logic specifications: *if-and-only-if* and *partial* specifications. First we set up a three-tier formalism for synthesis, the top-tier being a framework. Then within this formalism we define these two forms of specifications, and discuss their roles in synthesis.

1 Introduction

The purpose of program synthesis is to derive programs that are correct wrt to their (formal or informal) specifications. There is no universal agreement, however, on exactly what form a specification should take, what part it should play in synthesis, and what its precise relationship with the specified program should be. In logic programming, the role of specification is all the more unclear because a logic program itself is a Horn clause theory ([11]), and as such can double as a definition or specification. Indeed logic programs are often used as executable specifications, for instance in rapid prototyping.

In our work in deductive synthesis of logic programs (see e.g. [17, 18]), we maintain a strict distinction between specifications and programs. We take the view that program synthesis should take place in a three-tier formalism (with model-theoretic semantics). At the bottom level, we have *programs*, for computing (specified) relations. In the middle, we have *specifications* for defining

* Part of this work was done during the first author's visit to Computing Science Department, Uppsala University, Sweden, supported by the European Union HCM project on Logic Program Synthesis and Transformation, contract no. 93/414. He wishes to thank Jonas Barklund for his invitation and hospitality.

or specifying new relations (and functions). At the top, we have a *framework* that embodies an axiomatisation of (all the relevant knowledge of) the problem domain, that provides an unambiguous semantic underpinning for specifications and programs, as well as the correctness relationship between them.

In such a three-tier structure, a specification can serve two purposes: (i) to define new relation symbols, and (ii) to provide a starting point for, as well as help with, program synthesis. Furthermore, a specification can define relation symbols at two levels: (a) it can define symbols that can be used at the top level, i.e. relation symbols that expand the (language of the) framework; (b) it can define relation symbols that can be used at the bottom level, i.e. relation symbols that appear only in programs. Of course, symbols introduced at the framework level can also be used in programs. However, the converse is strictly not true.

In this paper, we first motivate a three-tier formalism for synthesis (Section 2), and then define and discuss two forms of specifications, viz. *if-and-only-if*, and *partial* specifications (Sections 4 and 5) in the context of a framework $\mathcal{F}$. We shall show that if-and-only-if specifications can be used to define symbols at both the framework and the program level, whereas partial specifications can define only program level symbols. However, as we shall show by means of examples, the latter specifications provide much more help than the former for practical program derivation.

We believe that a study and a classification of different forms of specifications is interesting, and could offer new possibilities to program synthesis. This paper is a preliminary step towards this goal.

2 A Three-tier Formalism

In this section, we use so-called *executable specifications*, to motivate a three-tier formalism for program synthesis. In logic programming, executable specifications are of course just logic programs. However, it is not clear exactly what the meaning of a logic program P is as an executable specification. Is it just the minimum Herbrand model of P? Is it the completion $Comp(P)$ of P? Or is it something else?

We would argue that the meaning of P as an executable specification is only well-defined in the context of the underlying problem domain, or *framework*. For example, consider the following Prolog program (in [22]) for *subset*:

```
member(X, [X|_]).
member(X, [_|Xs]) :- member(X, Xs).
                                                    (1)
   subset([], _).
subset([X|Xs], Ys) :- member(X, Ys), subset(Xs, Ys).
```

As a program, the meaning of (1) is of course completely defined, for example, by the usual minimum model semantics. However, as a specification, it is not at all clear that (1) specifies the *subset* relation (on sets), since it uses only lists. More precisely, (1) uses one particular representation of sets by lists, namely unordered lists possibly with duplicates. Clearly there are alternative programs/specifications for *subset* that use other list representations of sets. Sets can be represented by lists in various ways, since equality on sets differs from equality on lists. For example, $\{a, b\} = \{b, a\}$, whereas $a.b.nil \neq b.a.nil$. However, this information, as well as the definition of the chosen set representation by lists, is not (cannot be) contained in (1) itself; it can only be in the underlying framework.

For this reason, the intended meaning S of a program P used as an executable specification, needs to be specified separately. For instance, S for (1) might be the following informal specification (also in [22]):

> "`subset(X, Y)` is to be true if: `X` and `Y` are two lists of integers and the set `X` represents is a subset of the set that `Y` represents." (2)

Now, (2) uses the abstract data types of finite sets, lists, and integers, but it does not contain such theories. Therefore, its full and precise meaning is only defined in the the framework $\mathcal{F}$ of finite sets and lists of integers. For example, $\mathcal{F}$ contains the knowledge that the elements of a set can be listed in any order and the list may contain duplicates. This knowledge is assumed by (2).

This example therefore shows that to reason about synthesis, we need to have a three-tier formalism in which the meaning of a program P, at the *bottom* level, is defined by a specification S, at the *middle* level, while the meaning of S is in turn defined in the framework (in question), at the *top* level.

Thus we maintain a strict distinction between programs and specifications. In particular, by definition, so-called executable specifications do not provide a starting point for program synthesis, and so we do not regard them as proper specifications for this purpose. Rather, they are really programs that can and should be derived formally, especially if we want to know precisely what their (intended) meaning is.

It might seem that if we used a typed programming language such as Gödel ([10]), then the framework could be 'encoded' in executable specifications, and this would invalidate our view that executable specifications by themselves are not well-defined. However, other than *computational* knowledge, i.e. programs (typed or untyped), a framework also contains all the *non-computational* knowledge, e.g. induction principles, that is necessary for reasoning about programs. This allows us to write down specifications in a declarative style, and to derive from the same specification different (correct) programs.

3 Frameworks and Their Models

In this section, we briefly discuss frameworks and their model-theoretic semantics. This will set the scene for introducing specifications later.

A framework is an axiomatisation of a problem domain. It has

- a (many-sorted) signature of

- sort symbols;
- function declarations, for declaring constant and function symbols;
- relation declarations, for declaring relation symbols.

– a set of first-order axioms for the declared symbols.

A framework $\mathcal{F}$ is thus a full first-order logical theory. $\mathcal{F}$ has an intended model, viz. its *isoinitial model* I.

Definition 1. An *isoinitial model* I of $\mathcal{F}$ is a reachable model[3] 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 $\mathcal{F}$.

Example 1. For example, we can 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 framework $\mathcal{LIST}(Int)$ (defined in Example 2 below) 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*. The sort symbol *Int* stands for the standard (pre-defined) integer type. The sort *Set* is constructed by the *constructor* $\{\,\}$. The relation $\in$ is the usual 'set membership' relation. The first axiom defines $\in$ in terms of *mem*, while the second axiom defines set equality in terms of $\in$.

The isoinitial model of $\mathcal{SET}$ is the usual structure of lists of integers in which sets are represented as lists, i.e. the sort *Set* is interpreted as the set of finite sets of integers.

Note that the syntax of a framework is similar to that used in algebraic abstract data types (e.g. [24]).[4] However, algebraic abstract data types are usually defined by equational or Horn axioms, whereas frameworks are defined by full first-order axioms. Therefore, an algebraic abstract data type is an *initial* model of its specification, whereas a framework has an *isoinitial* model. Both initial and isoinitial theories enjoy the properties of 'no junk' and 'no confusion' ([8]), but only isoinitial theories handle negation properly.

The existence of an isoinitial model is not always guaranteed though. Indeed, we distinguish between two kinds of frameworks:

[3] That is a model where each element (of the domain) can be represented by a ground term.

[4] Or in typed logic programming languages such as Gödel [10].

– *Closed Frameworks.*
  A closed framework $\mathcal{F}$ is a theory that has an isoinitial model. This means that the axioms completely characterise the intended meaning (namely the interpretation in the isoinitial model) of the sort, function and relation symbols of the signature. It can be shown that if $\mathcal{F}$ has at least one reachable model, and for every ground atomic A, either A or $\neg A$ is provable in $\mathcal{F}$, then there exists an isoinitial model I of $\mathcal{F}$. Moreover, if $\mathcal{F}$ is a (possibly infinite) recursive set of axioms, then in I relation symbols are interpreted by decidable relations, and function symbols are interpreted by total computable functions.
– *Open Frameworks.*
  When an isoinitial model does not exist for a framework $\mathcal{F}$, we say that $\mathcal{F}$ is an *open* framework. In this case, the absence of I means that the axioms of $\mathcal{F}$ leave open the interpretation of some of the symbols of $\mathcal{F}$'s signature. Open frameworks are very useful in that they allow us to introduce parameters, and to deal with incomplete information that can be supplied via parameters (see [19]).

For simplicity, in this paper we will consider only specifications within closed frameworks. Such frameworks can be constructed incrementally, starting from a simple framework with an obvious isoinitial model, e.g. the term model generated by the constructor symbols (see e.g. [19]).

Example 2. A typical closed framework is (first-order) Peano arithmetic $\mathcal{NAT}$:[5]

Framework $\mathcal{NAT}$;

SORTS: Nat;

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

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

The isoinitial model $I_{\mathcal{NAT}}$ of $\mathcal{NAT}$ is the term model generated by its constructors 0 (zero) and s (successor), $+$ is the usual sum and $*$ is the usual product.

Finite structures, like integers with a fixed precision, or characters, can be axiomatised as well.[6] Now, suppose $\mathcal{INT}$ is a framework axiomatising integers. Then we can use $\mathcal{NAT}$ and $\mathcal{INT}$ to introduce the framework $\mathcal{LIST}(Int)$ for lists of integers:[7]

[5] We will omit the most external universal quantifiers from now on.
[6] By categorical axiomatisations where all the models are isomorphic.
[7] The sorts of the variables can be inferred from the context.

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

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

SORTS: $Nat, Int, List$;

FUNCTIONS: $nil \ : \to List$;

$\quad . \quad : (Int, List) \to List$;

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

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

AXIOMS: $\neg nil = a.B \wedge a_1.B_1 = a_2.B_2 \to a_1 = a_2 \wedge B_1 = B_2$;

$H(nil) \wedge \forall a, J \,.\, H(J) \to H(a.J) \to \forall L \,.\, H(L)$;

$nocc(x, nil) = 0$;

$a = b \to nocc(a, b.L) = s(nocc(a, L))$;

$\neg a = b \to nocc(a, b.L) = nocc(a, L)$;

$elemi(L, 0, a) \leftrightarrow \exists B \,.\, L = a.B$;

$elemi(L, s(i), a) \leftrightarrow \exists b, B \,.\, L = b.B \wedge elemi(B, i, a)$;

Lists can be constructed from the term model generated by the constructors $\cdot$ and *nil*. It can be shown that an isoinitial model $I_{\mathcal{LIST}}$ of $\mathcal{LIST}(Int)$ is an expansion of $I_{\mathcal{NAT}}$ and $I_{\mathcal{INT}}$. Indeed $I_{\mathcal{LIST}}$ is the usual structure (the term model generated by the constructors *nil* and $\cdot$) of lists of integers, where the function $nocc(a, L)$ gives the number of occurrences of a in L, and the relation $elemi(L, i, a)$ means a occurs at position i in L.

$\mathcal{LIST}(Int)$ contains $\mathcal{NAT}$, $nocc(a, L)$ and $elemi(L, i, a)$, and induction on lists, in order to provide a kernel with a sufficiently strong language and set of axioms. This kernel can be expanded mainly through if-and-only-if specifications, as we will see in the next Section.

Finally, since a framework introduces types, programs in a framework are typed.

4 If-and-only-if Specifications

In a given framework, as shown in the incremental construction of $\mathcal{SET}$ (and $\mathcal{LIST}$ and $\mathcal{NAT}$), we can define new (total computable) functions and (decidable) relations, by adding axioms which we call if-and-only-if specifications. In this section, we define and discuss such specifications. They are proper specifications, and they can also be used for constructing frameworks incrementally.

Definition 2. An *if-and-only-if specification* (or *iff specification* for short) S of a new *relation* r in a framework $\mathcal{F}$ consists of a definition of the form

$$r(x) \leftrightarrow R(x)$$

where $R(x)$ is any formula of the language of $\mathcal{F}$.

If $R(x)$ contains the defined symbol r, then we say that S is a *recursive* iff specification.

Recursive iff specifications have to be used with great care, since they may be ill-defined. For example, $r(x) \leftrightarrow \neg r(x)$ is logically inconsistent, whilst $r(x) \leftrightarrow r(x)$, though logically valid, does not characterise any specific intended interpretation of r (any interpretation satisfies it).

In contrast, non-recursive iff specifications are always consistent. In logic they are known as *explicit definitions*, and they have a clear declarative semantics:

> Let $\mathcal{F}$ be a framework and D_r be an explicit definition of a new relation r. Then, for every model M of the framework $\mathcal{F}$, there is a unique r-expansion M_r of M such that in M_r, r is interpreted according to D_r, whereas the old symbols are interpreted as in M.

For this reason, we will use mainly non-recursive iff specifications.

Thus, given a framework $\mathcal{F}$ and a (non-recursive) iff specification D_r, there is a unique r-expansion of the isoinitial model I of $\mathcal{F}$ that satisfies D_r. In other words, a relation defined by D_r has a unique interpretation in I. This is illustrated in Figure 1.

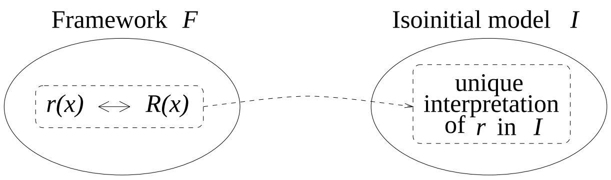

Fig. 1. Model-theoretic semantics of (non-recursive) *iff* specifications.

Example 3. In $\mathcal{SET}$, we can specify *subset* by the iff specification:

$$subset(A, B) \leftrightarrow \forall x \,.\, x \in A \rightarrow x \in B \tag{3}$$

In the sequel, instead of $subset(A, B)$ we will write $A \subseteq B$, for the sake of clarity.

Since sets are represented by lists in $\mathcal{SET}$, we can specify *sublist* by:

$$sublist(X, Y) \leftrightarrow \{X\} \subseteq \{Y\} \tag{4}$$

(in (3) A and B are sets, whereas in (4) X and Y are lists.)

In the isoinitial model of $\mathcal{SET}$, *subset* and *sublist* have a unique interpretation each.

Iff specifications are the most widely used form of specifications in logic program synthesis, e.g. [9, 5, 12, 17, 18]. In these papers,[8] various techniques are described for systematically deriving standard logic programs ([9, 5, 12,

[8] More details of these, and related papers, can be found in [6].

17]) or constraint logic programs ([18]). However, it should be noted that in e.g. [9, 5, 12], there is no (explicit) use of frameworks, or of model-theoretic semantics of iff specifications as described here, but they do make use of recursive iff specifications.

Iff specifications provide a starting point for program synthesis. For example in $\mathcal{SET}$ it is easy to see how we can derive the following program from (4):

$$
\begin{aligned}
sublist(nil, Z) &\leftarrow \\
sublist(x.Y, Z) &\leftarrow mem(x, Z), sublist(Y, Z) \\
mem(x, x.Z) &\leftarrow \\
mem(x, y.Z) &\leftarrow mem(x, Z)
\end{aligned}
\tag{5}
$$

(5) of course corresponds to the program (executable specification) (1) in Section 2. However, (5) has been (formally) derived from the specification (4) in $\mathcal{SET}$.

Besides synthesis, iff specifications can also be used to expand (the language of) a framework $\mathcal{F}$ by new symbols, since they have unique interpretations in the isoinitial model I of $\mathcal{F}$. However, for this purpose, we must use only *adequate* definitions D, viz. those that give rise to expansions of I which are also isoinitial models of the theory $\mathcal{F} \cup D$. Indeed, since D is of the form $\forall x \,.\, r(x) \leftrightarrow R(x)$, where R is any formula of the language, it may happen that the axioms of $\mathcal{F}$ are too weak to completely characterise the meaning of the new symbol r. For example, let $R(x)$ be a formula of $\mathcal{NAT}$ such that $\mathcal{NAT} \not\vdash R(0)$ and $\mathcal{NAT} \not\vdash \neg R(0)$ (such a formula exists); then the meaning of $r(0)$ is left open. Non-adequacy means that, starting from a closed framework $\mathcal{F}$, we get a framework containing an open symbol r. In model-theoretic terms, there is no longer an isoinitial model in the class of r-expansions of the models of $\mathcal{F}$.

A useful criterion for adequacy is the existence of a program P for computing the specified relation, such that P existentially terminates and the completion of P can be proved in the framework $\mathcal{F}$, enriched by the iff specifications of the predicates in P (see [16]). In other words, program synthesis can actually be used to expand $\mathcal{F}$ with adequate iff specifications.

Expansions of frameworks by new function symbols are equally important:

Definition 3. An (adequate) *iff specification* of a new *function* f in a framework $\mathcal{F}$ consists of a definition of the form

$$\forall x, y \,.\, y = f(x) \leftrightarrow r(x, y)$$

where $r(x, y)$ is a relation introduced by an (adequate) iff specification S_r, such that $\mathcal{F} \cup S_r \vdash \forall x \,.\, \exists! y \,.\, r(x, y)$.

An (adequate) iff specification S_f of a function introduces a new closed function symbol f, i.e. the (unique) f-expansion of the isoinitial model remains isoinitial and the meaning of f is completely characterised by the axioms of $\mathcal{F} \cup S_f$.

By adding new (closed) symbols, we can build up a framework for the problem domain in a natural and purely declarative way.

Example 4. We can expand $\mathcal{LIST}(Int)$ by adding the following (commonly used) functions and relations defined by iff specifications:

$$
\begin{array}{ll}
\text{FUNCTIONS:} & l : List \to Int; \\
& | : (List, List) \to List; \\
\text{RELATIONS:} & mem : (Int, List); \\
& len : (List, Nat); \\
& concat : (List, List, List); \\
& perm : (List, List); \\
& ord : (List); \\
\text{AXIOMS:} & mem(e, L) \leftrightarrow \exists i \,.\, elemi(L, i, e); \\
& len(L, n) \leftrightarrow \forall i \,.\, i < n \leftrightarrow \exists a \,.\, elemi(L, i, a); \\
& n = l(L) \leftrightarrow len(L, n); \\
& concat(A, B, L) \leftrightarrow \ (\forall i, a \,.\, i < l(A) \to \\
& \qquad\qquad\qquad\qquad (elemi(A, i, a) \leftrightarrow elemi(L, i, a))) \wedge \\
& \qquad\qquad\qquad\qquad (\forall j, b \,.\, elemi(B, j, b) \leftrightarrow \\
& \qquad\qquad\qquad\qquad elemi(L, j + l(A), b)); \\
& perm(A, B) \leftrightarrow \forall e \,.\, nocc(e, A) = nocc(e, B); \\
& C = A|B \leftrightarrow concat(A, B, C); \\
& ord(L) \leftrightarrow \forall i \,.\, elemi(L, i, e_1) \wedge elemi(L, s(i), e_2) \to e_1 \leq e_2.
\end{array}
$$

where l and $|$ are the usual functions for length and concatenation; and *mem*, *len*, *concat*, *perm*, and *ord* are the usual 'membership', 'length', 'concatenation', 'permutation', and 'ordered' relations.

All these definitions can be shown to be adequate.

We prefer (adequate) non-recursive iff specification for framework construction since they enable us to have a purely declarative specification language, i.e. to define the meaning of new symbols in a non-recursive way in terms of symbols that are already defined. For example, in $\mathcal{LIST}(Int)$ using the above additional definitions, we can specify sorting, declaratively as follows:

$$sort(X, Y) \leftrightarrow perm(X, Y) \wedge ord(Y)$$

However, recursive axioms cannot be avoided altogether, especially for axiomatising primitive operations on data types. For example, in $\mathcal{NAT}$, $+$ and $*$ cannot be defined non-recursively.

Thus our strategy for building a framework is as follows. First we define a sufficiently strong kernel framework by using *any* kind of axioms, both recursive and non-recursive. This requires skill and care, in order to avoid inconsistencies, and to guarantee the existence of an isoinitial model. Subsequently, we use adequate non-recursive iff specifications for introducing new symbols. Of course, recursive iff specifications are not forbidden, but they require more care.

5 Partial Specifications

For the purpose of program synthesis, in particular deriving different programs from the same specification, iff specifications are often too inflexible and unhelpful. For this purpose, it is better to use weaker forms of specifications that admit multiple interpretations, which we shall call *partial specifications.* In this section we define and discuss three of them: *super-and-sub*, *conditional* and *selector* specifications. These seem to us to be the most important and useful cases in practice.

5.1 Super-and-sub Specifications

Definition 4. A *super-and-sub specification* of a new relation r in a framework $\mathcal{F}$ consists of a definition of the form

$$\forall x \,.\, (R_{sub}(x) \rightarrow r(x)) \wedge (r(x) \rightarrow R_{super}(x))$$

where $R_{sub}(x)$ and $R_{super}(x)$ are two formulas of the language of $\mathcal{F}$ such that $\mathcal{F} \vdash \forall x \,.\, R_{sub}(x) \rightarrow R_{super}(x)$.

The implication $\forall x \,.\, R_{sub}(x) \rightarrow R_{super}(x)$ is satisfied by the isoinitial model I of $\mathcal{F}$.[9] Therefore the *sub* relation $R_{sub}(x)$ in I (i.e. the set of values $\mathbf{x}$ such that $I \models R_{sub}(\mathbf{x})$) is contained in the *super* relation $R_{super}(x)$, and the specified relation r is any relation that contains R_{sub} but is contained in R_{super}. This is illustrated in Figure 2.

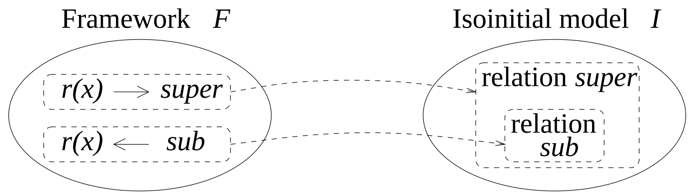

Fig. 2. Model-theoretic relation between R_{super} (*super*) and R_{sub} (*sub*).

Thus in contrast to one defined by an iff specification, a relation r defined by a super-and-sub specification S has many interpretations in the isoinitial model I of the framework $\mathcal{F}$. A *correct implementation* of S is any program P for computing r, such that the interpretation of r in P's minimum Herbrand model is a relation that contains R_{sub} and is contained in R_{super}.

It follows that a super-and-sub specification of a relation r can be implemented correctly in different ways. In particular, the relation R_{super} defines or suggests many possible implementations for r, but a correct implementation must compute a relation that contains R_{sub}. Thus R_{super} defines the search space for correct implementations, whilst R_{sub} guides the search for them. In

[9] It is in fact satisfied by all the models of $\mathcal{F}$, since it is a theorem of $\mathcal{F}$.

other words, R_{super} and R_{sub} delimit the search space for correct implementations. Consequently, super-and-sub specifications are very versatile and natural, and therefore helpful, for practical program synthesis.

In general, of course a partial specification is useful in practice only if we have methods to prove the correctness of programs. For super-and-sub specifications, the following theorem provides the necessary proof-theoretic characterisation of correctness:

Theorem 5. *Let S be a super-and-sub specification written as a pair of implications:*

$$\begin{aligned} r(x) &\rightarrow R_{super}(x) \\ r(x) &\leftarrow \ \ R_{sub}(x) \end{aligned}$$

and let P be a program for computing r, $Comp^+(P)$ be the if-part and $Comp^-(P)$ be the only-if-part of the completion of P. If

(a) $\forall x \,.\, \mathcal{F} \cup \{r(x) \leftrightarrow R_{super}(x)\} \vdash Comp^+(P)$,
(b) $\mathcal{F} \cup \{\forall x \,.\, r(x) \leftrightarrow R_{sub}(x)\} \vdash Comp^-(P)$, *and*
(c) P existentially terminates;

then P is a correct implementation of S.

Indeed (a) guarantees that the success set is contained in R_{super}, (b) guarantees that the finite-failure set contains R_{sub}, and (c) says that the success and finite failure sets are complementary.

This theorem is not only useful for proving the correctness of an implementation wrt a super-and-sub specification, but it can also provide a basis for a synthesis method.

Example 5. Suppose we are given the following iff specifications:

$$\begin{aligned} &nocc_1(L) && \leftrightarrow \forall x \,.\, nocc(x, L) \leq 1 \\ &select_{super}(x, Y, H) && \leftrightarrow \neg nocc_1(Y) \vee \neg x \in \{Y\} \vee nocc_1(x.H) \wedge \{x.H\} = \{Y\} \end{aligned}$$

$nocc_1(L)$ means that L does not contain duplicates, and $select_{super}(x, Y, H)$ means that H does not contain duplicates and contains all the elements of Y except x.

Suppose for every pair (x, Y) such that $x \in \{Y\}$ and $nocc_1(Y)$ holds, we want to compute at least one H such that $select_{super}(x, Y, H)$ holds. We can specify this as $select(x, Y, H)$ by a super-and-sub specification of the form

$$select_{sub}(x, Y, H) \rightarrow select(x, Y, H) \wedge select(x, Y, H) \rightarrow select_{super}(x, Y, H) \quad (6)$$

and look for a correct program P in the search space delimited by $select_{super}$ and $select_{sub}$, by first considering $select_{super}$ and then $select_{sub}$.

Note that while considering $select_{super}$ we can even leave $select_{sub}$ unspecified, and only require that

$$\begin{array}{l} select_{sub}(x, Y, H) \text{ be satisfied by at least one } H \text{ whenever } x \in \{Y\} \\ \text{and } nocc_1(Y) \text{ holds, and } select_{sub}(x, Y, H) \rightarrow select_{super}(x, Y, H). \end{array} \quad (7)$$

In this way, we do not (over)constrain the search space unnecessarily.

Now, the most obvious candidate for P is clearly one derivable from $select_{super}$ such that we can prove condition (a) of Theorem 5. We have:

$$
\begin{array}{lll}
select(x, x.X, X) & \leftrightarrow & select_{super}(x, x.X, X) \\
 & \leftrightarrow & \neg nocc_1(x.X) \vee \neg x \in x.X \vee nocc_1(x.X) \wedge \{x.X\} = \{x.X\} \\
 & \leftrightarrow & true \\
select(x, y.Y, y.H) & \leftrightarrow & select_{super}(x, y.Y, y.H) \\
 & \leftrightarrow & \neg nocc_1(y.Y) \vee \neg x \in y.Y \vee nocc_1(x.y.H) \wedge \\
 & & \{x.y.H\} = \{y.Y\} \\
 & \leftarrow & \neg nocc_1(Y) \vee \neg x \in Y \vee nocc_1(x.H) \wedge \{x.H\} = \{Y\} \\
 & \leftrightarrow & select(x, Y, H)
\end{array}
$$

So we have derived the program

$$
\begin{array}{l}
select(x, x.X, X) \leftarrow \\
select(x, y.Y, y.H) \leftarrow select(x, Y, H)
\end{array}
\tag{8}
$$

Obviously (8) satisfies condition (a) of Theorem 5.

Next we need to establish that (8) also meets condition (b) of Theorem 5 for a suitable $select_{sub}$. Looking at (8), we might surmise the following definition of $select_{sub}$:

$$
\begin{array}{ll}
select_{sub}(x, Y, H) \leftrightarrow & nocc_1(Y) \wedge nocc_1(x.H) \wedge \\
 & \exists U, V \mathbin{.} H = U|V \wedge Y = U|x.V
\end{array}
\tag{9}
$$

This satisfies the requirement (7) above. So, if we can prove condition (b) of Theorem 5, then (9) is a suitable definition.

The proof is:

$$
\begin{array}{lll}
select(x, A, B) & \leftrightarrow & nocc_1(A) \wedge nocc_1(x.B) \wedge \exists U, V \mathbin{.} B = U|V \wedge A = U|x.V \\
 & \rightarrow & (\exists V \mathbin{.} B = V \wedge A = x.V) \\
 & & \vee(\exists y, Y, V \mathbin{.} B = y.Y|V \wedge A = y.Y|x.V) \\
 & \rightarrow & \exists X, Y, H \mathbin{.} A = x.X \wedge B = X \vee A = y.Y \wedge B = y.H \\
select(x, y.Y, y.H) & \leftrightarrow & nocc_1(y.Y) \wedge nocc_1(x.y.H) \wedge \\
 & & \exists U, V \mathbin{.} y.H = U|V \wedge y.Y = U|x.V \\
 & \rightarrow & nocc_1(Y) \wedge nocc_1(x.H) \wedge \exists U, V \mathbin{.} H = U|V \wedge Y = U|x.V \\
 & \leftrightarrow & select(x, Y, H)
\end{array}
$$

Thus the program (8) is a correct implementation of the super-and-sub specification (6) where $select_{sub}$ is as defined by the iff specification (9).[10]

[10] Termination needs to be proved separately.

5.2 Conditional Specifications

Definition 6. A *conditional specification* of a new relation r in a framework $\mathcal{F}$ consists of a definition of the form

$$\forall x, y \,.\, IC(x) \rightarrow (r(x,y) \leftrightarrow OC(y) \wedge R(x,y))$$

where $IC(x)$ is a condition on x, $OC(y)$ is a condition on y, and $R(x,y)$ is a formula of the language of $\mathcal{F}$.

Roughly speaking, the relation $r(x,y)$ is defined if and when $IC(x)$ holds, and in this case $r(x,y)$ is defined to be $R(x,y)$ and $OC(y)$ will also hold. If we follow the pre-post-condition style of program specification (e.g. [14, 23]), then a conditional specification defines a program P to compute $r(x,y)$ in such a way that whenever $IC(x)$ is satisfied, P satisfies $OC(y) \wedge R(x,y)$. Therefore, we shall say that $IC(x)$ is an *input-* or *pre-condition* on x, $OC(y)$ is an *output condition* on y, and the formula $OC(y) \wedge R(x,y)$ is a *post-condition*.

Example 6. For example, we could specify the *sublist* relation by the following conditional specification:

$$IC(Y) \rightarrow (\mathit{sublist}(X,Y) \leftrightarrow OC(X) \wedge \{X\} \subseteq \{Y\}) \tag{10}$$

where $IC(Y)$ and $OC(X)$ state that Y and X do not contain duplicates. Note that here Y is the input and X the output.

The specification (10) defines $\mathit{sublist}(X,Y)$ to be $\mathit{subset}(\{X\},\{Y\})$ (and specifies that X does not contain duplicates) whenever Y does not contain duplicates.

Thus, like one defined by a super-and-sub specification, a relation defined by a conditional specification C has many interpretations in the isoinitial model I of the framework $\mathcal{F}$.

It is easy to see that a conditional specification

$$IC(x) \rightarrow (r(x,y) \leftrightarrow OC(y) \wedge R(x,y)) \tag{11}$$

can always be re-written as a pair of implications:

$$\begin{aligned} r(x,y) &\rightarrow \neg IC(x) \vee (OC(y) \wedge R(x,y)) \\ r(x,y) &\leftarrow \quad IC(x) \wedge (OC(y) \wedge R(x,y)) \end{aligned}$$

That is (11) becomes the super-and-sub specification:

$$\begin{aligned} r(x,y) \rightarrow R_{super} &\leftrightarrow \neg IC(x) \vee (OC(y) \wedge R(x,y)) \\ r(x,y) \leftarrow R_{sub} \quad &\leftrightarrow \quad IC(x) \wedge (OC(y) \wedge R(x,y)) \end{aligned}$$

For instance, the conditional specification (10) can be re-written as:

$$\begin{aligned} \mathit{sublist}(X,Y) \rightarrow \mathit{select}_{super} &\leftrightarrow \neg IC(Y) \vee (OC(X) \wedge \mathit{subset}(X,Y)) \\ \mathit{sublist}(X,Y) \leftarrow \mathit{select}_{sub} \quad &\leftrightarrow \quad IC(Y) \wedge (OC(X) \wedge \mathit{subset}(X,Y)) \end{aligned} \tag{12}$$

Therefore, for synthesising programs from a conditional specification C, we can apply the same proof method for super-and-sub specifications (in the previous subsection), by re-writing C as a pair of implications. The search for correct programs, however, is now governed by the input and output conditions, IC and OC respectively. Moreover, now R_{sub} is defined *a priori*, thus making the search easier.

Example 7. Consider the conditional specification (10), or the equivalent super-and-sub specification (12).

(i) Suppose both the input and output conditions IC and OC are the relation $nocc_1$. It can be shown that if we use the specification (6) for *select* (and the corresponding program (8)), then we can derive the following program for *sublist*:

$$\begin{array}{l} sublist(nil, Y) \leftarrow \\ sublist(x.X, Y) \leftarrow select(x, Y, H), sublist(X, H) \end{array} \tag{13}$$

following the strategy outlined in Example 5, except that here $select_{sub}$ is defined *a priori*. For brevity, we omit the derivation steps.

(ii) Alternatively, if in (10) we choose both IC and OC to be *ord* defined by the iff specification:

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

($ord(L)$ means that L is a sorted list), and then specify *append* by the following iff specification:

$$append(X, Y, Z) \leftrightarrow X|Y = Z \tag{14}$$

we can derive the following alternative program for *sublist*:

$$\begin{array}{ll} append(nil, X, X) & \leftarrow \\ append(x.X, Y, x.Z) & \leftarrow append(X, Y, Z) \\ sublist(nil, X) & \leftarrow \\ sublist(x.X, Y) & \leftarrow append(W, x.Z, Y), sublist(X, Z) \end{array} \tag{15}$$

Again, we omit the derivation steps.

5.3 Selector Specifications

Definition 7. A *selector specification* has the following form:

$$\begin{array}{ll} (i) & \forall x, y \,.\, IC(x) \wedge r(x, y) \rightarrow R(x, y) \wedge OC(y) \\ (ii) & \forall x \,.\, IC(x) \rightarrow \exists y \,.\, r(x, y) \end{array}$$

where r is the new specified relation and $R(x, y)$, $IC(x)$ and $OC(y)$ are formulas of the language of the framework.

This specification says that r is any relation contained in R, such that for every input x that satisfies the input condition $IC(x)$, r picks up, or *selects*, at least one output y.

This kind of specification is useful for deriving sub-programs in hierarchical program synthesis.

Example 8. Consider the hierarchical synthesis of a program for the relation *sort* defined by:

$$sort(X,Y) \leftrightarrow perm(X,Y) \land ord(Y)$$

in $\mathcal{LIST}(Int)$, and an (as yet) incomplete sorting program with the following structure:

$$\begin{array}{ll} sort(nil,nil) & \leftarrow \\ sort(x.nil,x.nil) & \leftarrow \\ sort(x.y.A,B) & \leftarrow split(x.y.A,I,J),\ldots \end{array}$$

where in the recursive clause the input $x.y.A$ is split into two shorter parts I and J, that can be used in various ways in the body. In this case *split* can be specified as follows:

$$\begin{array}{l} l(X) > 1 \land split(X,Y,Z) \rightarrow perm(X,Y|Z) \land l(Y) < l(X) \land l(Z) < l(X) \\ l(X) > 1 \rightarrow \exists Y,Z \,.\, split(X,Y,Z) \end{array}$$

That is, if X has length greater than 1, then *split* has to select a pair Y and Z, among all the pairs such that $Y|Z$ is a permutation of X, with the requirement the selected lists Y and Z must be shorter than X. A trivial example of a correct program is:

$$split(x.y.A,x.nil,y.A) \leftarrow \tag{16}$$

This choice gives rise to a program for insertion sort. Other choices give rise to other sorting algorithms.

For this kind of specifications, we have the following correctness theorem:

Theorem 8. *Let S be the following selector specification:*

$$\begin{array}{ll} (i) & \forall x,y \,.\, IC(x) \land r(x,y) \rightarrow R(x,y) \land OC(y) \\ (ii) & \forall x \,.\, IC(x) \rightarrow \exists y \,.\, r(x,y) \end{array}$$

Let P be a program for computing r. If

(a) $\mathcal{F} \cup \{\forall x,y \,.\, r(x,y) \leftrightarrow \neg IC(x) \lor R(x,y) \land OC(y)\} \vdash Comp^+(P)$, and
(b) for every goal $\leftarrow r(g,Y)$, where g is a ground term that satisfies $IC(g)$, $P \cup \{\leftarrow r(g,Y)\}$ successfully terminates;

then P is a correct implementation of S.

Example 9. For example, consider the program for *split* (16):

(a) We can prove

$$\begin{array}{rl} split(x.y.A,x.nil,y.A) \leftrightarrow & (\neg l(x.y.A) > 1 \lor (perm(x.y.A,x.nil|y.A) \\ & \land l(x.nil) < l(x.y.A) \land l(y.A) < l(x.y.A))) \\ & \leftrightarrow true \end{array}$$

(b) The successful termination of (16) for every goal $\leftarrow split(g,X,Y)$, where g is ground and has length greater than 1, is obvious.

Hence the program (16) is a correct implementation of S.

6 Discussion and Concluding Remarks

We have presented a model-theoretic view of frameworks and specifications. The model-theoretic semantics discussed here is used in [19] to define the model-theoretic relationship between frameworks, specifications and programs, in a three-tier formalism for logic program synthesis.

The kinds of specifications presented here are intended for *deductive synthesis*, and not for *constructive synthesis* (see [6] for a survey of different approaches to logic program synthesis). Our use of frameworks distinguishes our approach to specifications from related work in deductive synthesis. Moreover, we use a model-theoretic approach whereas others predominantly adopt a proof-theoretic approach. For example, in [9, 5, 12], frameworks are not used, and iff specifications are used in a proof-theoretic way to derive logic programs, and/or to prove them correct.

Nevertheless, there are similarities between our specifications and some of their verification conditions. For example, Clark [4, Section 4] (see also [13, p.158]) uses the following *correctness sentence*:

$$\forall x, y \,.\, I(x) \rightarrow (r(x,y) \leftrightarrow O(x,y)) \tag{17}$$

where $I(x)$ and $O(x,y)$ are input and output relations, to verify the total correctness of a program P to compute r. (17) is almost identical to our conditional specification, although it is not used for specification purposes.

In general, constructive synthesis follows the 'proofs as programs' approach (e.g. [1, 20]).[11] A specification consists of an input-output theorem of the form

$$\forall x \exists y \,.\, r(x,y) \tag{18}$$

From a proof of this theorem, a program to compute r can be extracted. This kind of specification is similar to our selector specification without an input condition, i.e. for every input x there is at least one y which satisfies $r(x,y)$ and is computed by the specified (or extracted) program.

In logic programming, the connection between deductive and constructive synthesis is particularly close. For example, in the constructive synthesis method of [2], the input-output relation $r(x,y)$ in (18) is itself an iff specification. In [15, 7], program extraction is performed by proving goals of the form

$$\forall x \exists y \,.\, q(x,y) \leftarrow r(x) \tag{19}$$

using an extended Prolog execution mechanism. Here, $r(x)$ can be regarded as an input, and $q(x,y)$ as an output. Therefore, (19) is similar to a selector specification of the form:

$$\begin{array}{l} r(x) \wedge q_P(x,y) \rightarrow q(x,y) \\ r(x) \rightarrow \exists y \,.\, q_P(x,y) \end{array}$$

[11] Note that Manna & Waldinger uses 'deductive synthesis' to refer to what we call 'constructive synthesis' here.

where $q(x,y)$ is the relation in the specification language, and q_P is the relation computed by the (extracted) program. It is also worth noting that if for every x such that $r(x)$ holds, the program computes all the y such that $q(x,y)$, i.e. if the answers of a goal $\leftarrow q_P(g,Y)$ are all the values v such that $q(g,v)$ is true in the isoinitial model of the framework, then (19) is a conditional specification.

Conditional specifications are widely used in so-called *formal methods*, e.g. VDM [14] and Z [23], in the form of pre- and post-conditions for specifying the effects of operations on the current computational state. However, these methods use such specifications in a purely proof-theoretic way.

Super-and-sub specifications, and selector specifications may also have been used by other researchers in deriving programs, but we are not aware of any formalisation of such specifications in existing literature.

Finally, it would seem that iff specifications are best suited to the incremental construction of a framework by adding adequate definitions via the synthesis of corresponding totally correct programs. For synthesis itself, it would seem that partial specifications are the most natural and versatile.

Acknowledgements

We wish to thank Alberto Pettorossi, and the referees, for their helpful comments and suggestions.

References

1. J.L. Bates and R.L. Constable. Proofs as programs. *ACM TOPLAS* 7(1):113-136, 1985.
2. A. Bundy, A. Smaill and G. Wiggins. The synthesis of logic programs from inductive proofs. In J.W. Lloyd, editor, *Proc. Esprit Symposium on Computational Logic*, pages 135–149, Springer-Verlag, 1990.
3. C.C. Chang and H.J. Keisler. *Model Theory*. North-Holland, 1973.
4. K.L. Clark. Predicate Logic as a Computational Formalism. Tech. Rep. 79/59, Imperial College, 1979.
5. K.L. Clark. The Synthesis and Verification of Logic Programs. Tech. Rep. 81/36, Imperial College, 1981.
6. Y. Deville and K.K. Lau. Logic program synthesis. *J. Logic Programming* 19,20:321–350, 1994. Special issue: Ten years of logic programming.
7. L. Fribourg. Extracting logic programs from proofs that use extended Prolog execution and induction. In D.H.D. Warren and P. Szeredi, editors, *Proc.* 7^{th} *Int. Conf. on Logic Programming*, pages 685–699, MIT Press, 1990.
8. J. Goguen and J. Meseguer. Unifying functional, object-oriented and relational programming with logical semantics. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 417–477, MIT Press, 1987.
9. Å. Hansson and S.-Å. Tärnlund. A natural programming calculus. In *Proc. IJCAI 79*, pages 348–355, 1979.
10. P.M. Hill and J.W. Lloyd. *The Gödel Programming Language*. MIT Press, 1994.

11. 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.
12. C.J. Hogger. Derivation of logic programs. *J. ACM*, 28(2):372–392, April 1981.
13. C.J. Hogger. *Introduction to Logic Programming*. Academic Press, 1984.
14. C.B. Jones. *Systematic Software Development Using VDM.* Prentice Hall, 2nd edition, 1990.
15. T. Kanamori and H. Seki. Verification of Prolog programs using an extension of execution. In E. Shapiro, editor, *Proc. 3^{rd} Int. Conf. on Logic Programming*, *Lecture Notes in Computer Science* 225, pages 475–489, 1986.
16. 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.
17. 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.
18. 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.
19. K.K. Lau and M. Ornaghi. The relationship between logic programs and specifications — the subset example revisited. To appear in *J. Logic Programming.*
20. Z. Manna and R. Waldinger. Fundamentals of deductive program synthesis. *IEEE Trans. on Soft. Eng.* 18(8):53–79, 1992.
21. 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.
22. R.A. O'Keefe. What does subset mean? *ALP Newsletter* 8(3):10, August 1995.
23. J.M. Spivey. *Understanding Z*. Cambridge University Press, 1988.
24. M. Wirsing. Algebraic specification. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 675–788. Elsevier, 1990.