On Specification Frameworks and Deductive Synthesis of Logic Programs*

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

[1] Department of Computer Science, University of Manchester
Oxford Road, 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@imiucca.csi.unimi.it

1 Introduction

Logic program synthesis methods can be roughly divided into those that use a formal specification and a formal synthesis process, and those that use an informal specification and an informal synthesis technique. A taxonomy and a survey of logic program synthesis methods can be found in [5].

In formal methods for logic program synthesis, the issue of what specification language is best has never loomed large, because first-order logic is the obvious candidate and hence the natural choice. This, in our view, has resulted in an over-simplistic view of logic program specifications.

In our own area of *deductive synthesis* of logic programs, for example, a specification is often just an *if-and-only-if* first-order definition, or a set of such definitions. Such a simplistic approach to specification has, in our experience, two major drawbacks:

- It cannot capture the entire body of knowledge needed for and used in the synthesis process.
  For example, our formalisation of deductive synthesis of logic programs in [9] reveals that the synthesis process has to make use of background knowledge not captured by the *if-and-only-if* definition of the specified relation. Such knowledge includes relevant theories, e.g. of data types, with induction schema, etc.
- It does not provide an adequate means to specify properties other than correctness.
  For example, in [10], we show that if we want to synthesise *steadfast* programs, i.e. programs that are modular in the sense that they define program units that remain correct (and hence unchanged) when integrated into larger

* The first author was partially supported by the EC HCM project on Logic Program Synthesis and Transformation, contract no. 93/414. The second author was partially supported by MURST.

programs, then we need to use axiomatisations which allow us to reason about program modularity and composition.

In our work [9, 10], therefore, we have used *specification frameworks* to provide the background for deductive synthesis of logic programs. In this paper, we take a closer look at such frameworks. We shall explain what they are, and how they can be used to specify properties such as correctness and modularity (and hence reusability).

Moreover, we shall show that there is a close two-way relationship between specification frameworks and deductive synthesis. In particular, a deductive synthesis process can provide a useful feedback mechanism which can not only check for desirable properties in the specification framework, but also improve the framework (with regard to such properties) using the result of the synthesis.

In our approach to modularity, we borrow many of the basic ideas developed in the algebraic approach (e.g. [6, 7, 14]). We shall briefly contrast the two approaches in Section 2.3.

2 Specification Frameworks and Specifications

In this section, we give an informal introduction to specification frameworks, and show why and how we distinguish between them and *specifications*. We also briefly explain the background and motivation for our approach.

2.1 Specification Frameworks

We take the view that program synthesis should take place in a general framework within which we can (a) specify, and (b) synthesise programs, for many different computational problems in that framework. The idea is that for (a) a framework should be *sufficiently expressive*, and for (b) it should act as a repository that contains all the knowledge that is needed for reasoning about the problem domain of interest.

The following are two examples of specification frameworks:

Example 1. A well-known example is Peano Arithmetic, which we will denote by $\mathcal{PA}$. Its language contains identity '=', successor 's', sum '+' and product '$\cdot$'. In $\mathcal{PA}$ every natural number $\mathbf{n}$ is represented by a *numeral* n.[3] Every computable function $f(x)$ can be expressed by a formula $F(x, z)$,such that for every pair of natural numbers $\mathbf{m}$ and $\mathbf{n}$, $\mathbf{m} = f(\mathbf{n})$ iff $F(m, n)$, where m and n are the numerals corresponding to $\mathbf{m}$ and $\mathbf{n}$ respectively, is provable in $\mathcal{PA}$. Thus, in $\mathcal{PA}$ we can *specify* any computable function by a formula; and therefore $\mathcal{PA}$ is *sufficiently expressive*.

Example 2. Another example is the following framework $\mathcal{LIST}$ for lists. We assume that the framework $\mathcal{PA}$ has already been defined. It contains the language

[3] Numerals are $0, s(0), s(s(0)), \ldots$

and the axioms of Peano Arithmetic and is enriched by the most useful primitive recursive functions and predicates. We use (i.e. we "import") $\mathcal{PA}$ in the following framework specifying lists, to introduce the *function* $len(L)$ (meaning the length of L) and the *relation* $elemi(L,i,e)$ (meaning the element e occurs at position i in L). Here, the specification of lists leaves undefined the list elements (namely the domain of the sort *Elem*) and the ordering $\leq$ on them, and thus *Elem* and $\leq$ act as *parameters*. We have a many-sorted language with (polymorphic) identity '='. The 'statement' **Import** $\mathcal{PA}$ incorporates the signature and the axioms of $\mathcal{PA}$, while the subsequent 'statements' add new sort, function and relation symbols to the signature, and introduce new axioms.

Framework $\mathcal{LIST}$;
Import $\mathcal{PA}$;

SORTS: $Elem, List$;

FUNCTIONS:
$nil : \to List$;
$\mid \;: (Elem, List) \to List$;
$len : List \to Nat$;

RELATIONS:
$\leq \;: (Elem, Elem)$;
$elemi : (List, Nat, Elem)$

AXIOMS:
$\forall a : Elem, B : List.\, \neg nil = a|B$
$\forall a_1, a_2 : Elem, B_1, B_2 : List.\, (a_1|B_1 = a_2|B_2 \to a_1 = a_2 \wedge B_1 = B_2)$
$\forall e : Elem, L : List.\, (elemi(L,0,e) \leftrightarrow \exists B : List.\, L = e|B)$
$\forall e : Elem, L : List, i : Nat.$
$(elemi(L,s(i),e) \leftrightarrow \exists a : Elem, B : List.\, (L = a|B \wedge elemi(B,i,e)))$
$H(nil) \wedge \forall x : Elem, J : List.\, (H(J) \to H(x|J)) \to \forall L : List.\, H(L)$

End $\mathcal{LIST}$.

The last axiom is the first-order induction schema on lists, where H is any (first-order) formula of the language of lists.

Natural numbers are not needed to specify list. However, the presence of *elemi* yields a more expressive language, which allows us to define in an easy and natural way many different relations on lists, as we will see later.

Thus we define a *specification framework* as a consistent first-order theory with a many-sorted signature, and with identity. We allow full first-order language for the sake of expressiveness. We will denote frameworks by $\mathcal{F}, \mathcal{G}, \ldots$

2.2 Specifications

A framework is thus a general theory, wherein many different computational problems can be specified in a clear, declarative manner. To specify a problem declaratively, we explicitly define the relation to be computed. We also list the goals to be solved by the synthesised program(s).

Definition 1. A *specification* in a framework $\mathcal{F}$ with language $\mathcal{L}_{\mathcal{F}}$, consists of:

- a *definition axiom* D_r:

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

  where r is a new predicate symbol and R is a formula in $\mathcal{L}_{\mathcal{F}}$.
- a set G of *goals* of the form $\exists r(t_0), \exists r(t_1), \ldots$

The link between specifications and programs is stated by the following definition.

Definition 2. Let $\langle D_r, G\rangle$ be a specification within a framework $\mathcal{F}$. A program P is *totally correct* in a model M of $\mathcal{F}$ *wrt* $\langle D_r, G\rangle$ if, for every goal $\exists r(t_k) \in G$:

- if $\exists r(t_k)$ is true in M, then there is at least a computed answer substitution[4] for $P \cup \{\leftarrow r(t_k)\}$;
- if σ is a computed answer substitution for $P \cup \{\leftarrow r(t_k)\}$, then $r(t_k)\sigma$ is true in M.

We define correctness in a model, since we will consider both closed frameworks, which single out one intended model, and open frameworks, which characterise a class of models.

Example 3. In the framework $\mathcal{LIST}$ of Example 2, we can specify many problems on lists. Firstly, we can specify many new relations on lists, using *definition axioms*. For example, the new relation symbols

- $sub(A, B)$, meaning A is a sub-list of B, i.e. every element of A is an element of B;
- $equiv(A, B)$, meaning A and B are equivalent, i.e. they contain the same elements; and
- $ord(L)$, meaning L is ordered,

can be introduced by the definition axioms D_{sub}, D_{equiv} and D_{ord}:

$$\begin{array}{ll} D_{sub} & : \forall A, B : List.\,(sub(A,B) \leftrightarrow \\ & \quad \forall e : Elem.\,(\exists i : Nat.elemi(A,i,e) \rightarrow \exists i : Nat.\,elemi(B,i,e))) \\ D_{equiv} & : \forall A, B : List.\,(equiv(A,B) \leftrightarrow sub(A,B) \wedge sub(B,A)) \\ D_{ord} & : \forall L : List.\,(ord(L) \leftrightarrow \\ & \quad \forall i : Nat, e_1, e_2 : Elem.\,(elemi(L,i,e_1) \wedge elemi(L,s(i),e_2) \rightarrow e_1 \le e_2)) \end{array}$$

Each relation symbol so defined, together with the corresponding definition axiom, can be added to the framework, thus enriching our language.

We can now specify, for example, the sorting problem by $\langle D_{sort}, G_{sort}\rangle$, with:

$$\begin{array}{l} D_{sort} \;:\; \forall L, S : List.\,(sort(L,S) \leftrightarrow equiv(L,S) \wedge ord(S)) \\ G_{sort} \;=\; \{\exists S : List.\,sort(l,S) \mid l \text{ is ground}\} \end{array}$$

The relations *ord* and *sort* depend on the ordering $\le: List \times List$; hence this specification specifies a class of sorting programs, depending on the interpretations of $\le: List \times List$.

[4] As defined in [11].

2.3 Background and Motivation

Various axiomatisations of data and programs can be found in the literature. For logic programming, Clark and Tärnlund [4] formulated a first-order theory of programs and data. Other examples of first-order theories of general interest to Computer Science are presented in [12].

However, a systematic use of axiomatisations to specify data and programs has been studied mainly in the field of algebraic specifications of abstract data types (ADT's). We refer in particular to the so-called *initial algebra approach*, which has been popularised by many authors (e.g. [8, 6, 7, 14]). In our approach, we incorporate many of their basic ideas (and their terminology): in particular, modularity (and parametric modules, like $\mathcal{LIST}$), and intended-model semantics.

However, there are some major and fundamental differences. In this section, we briefly outline them and thus contrast our approach with that of initial algebra.

We are concerned with program synthesis, and we have a different point of view on what a specification should be, for the purpose of synthesis. We distinguish between specification frameworks and specifications, and we do not impose any restrictions on the form of the axioms. In contrast, the initial algebra approach only allows restricted classes of axioms. Consequently, we cannot exploit the strengths of the initial algebra approach, namely that:

(a) Equational theory admits an initial model [8], unique up to isomorphism, and this result can be easily extended to Horn axioms (as in EQLOG[6]).

(b) There is a correspondence between truth in the initial model and provability for positive formulas. This property provides a kind of goal-solving completeness: a positive existential formula $\exists x A(x)$ (where A contains only $\wedge, \vee$) is true in the initial model iff there is an instance $A(t)$ provable in the theory.

However, as soon as we introduce negation, for example, we get into the following problems:

(a') The existence of an initial model is lost,[5] and there are no general and effective criteria to isolate those theories which do admit an initial model.

(b') Even if a theory $\mathcal{F}$ has an initial model I, a negated atomic formula may be true in I, but not provable; this destroys the goal-solving completeness property.

Both these points are critical in our approach: (a') because we do not want particular restrictions on the form of the axioms, and (b') because in a definition axiom $\forall x(r(x) \leftrightarrow R(x))$, $R(x)$ is any formula (using negation we may loose goal-solving completeness wrt the goals $\exists r(t)$).

Therefore we need some effective criterion for the existence of the intended model. If we use *isoinitial semantics* instead of initial semantics, we get such a criterion

[5] As is well-known in logic programming, the completion of a normal program may be inconsistent, or it may have many incomparable models.

Isoinitial semantics was introduced in [1], as a semantic characterisation of the computable ADT's. In this semantics, the intended model of a theory $\mathcal{F}$ is the *isoinitial model* (when it exists), i.e. a model J of $\mathcal{F}$ such that for every other model M of $\mathcal{F}$, there is a unique *isomorphic embedding* of J in M.

In other words, whereas initial semantics uses homomorphisms, isoinitial semantics uses isomorphic embeddings, which preserve relations and their negations. As a consequence, isoinitial models turn out to be always recursive, whilst this property is not guaranteed, in general, for initial models. An extensive comparison of the two kinds of semantics can be found in [1].

We will make use of the following criterion for the existence of isoinitial models.

Definition 3. We say that a model M of a theory $\mathcal{F}$ is *reachable* if every element of the domain of M is represented by some closed term of the language of $\mathcal{F}$.

We say that $\mathcal{F}$ is *atomically complete* if, for every closed atomic formula A, either $\mathcal{F} \vdash A$, or $\mathcal{F} \vdash \neg A$.

Theorem 4. *If a theory $\mathcal{F}$ has at least one reachable model, then $\mathcal{F}$ admits an isoinitial model if and only if it is atomically complete.*

This theorem gives atomic completeness as the key condition for the exsitence of isoinitial models. In our work, we will show that program synthesis can be used to test for atomic completeness, by the provability of the completion of synthesised programs.

3 Closed and Open Specification Frameworks

We will distinguish between *closed* and *open* frameworks. Informally speaking, an *open* framework leaves open the possibility of different interpretations of the symbols of its language, corresponding to different intended models, while a *closed* framework has a unique *intended model*, determining a unique interpretation.

For example, $\mathcal{PA}$ is a *closed* framework, since the meaning of 's', '$+$', '$\cdot$', '$=$' is completely defined and the intended model is the standard structure of natural numbers. In contrast, $\mathcal{LIST}$ is an *open* framework, since the meaning of *Elem* and $\leq : (Elem, Elem)$ is left open. Indeed, no axiom is given to characterise the domain of the sort *Elem* or the ordering $\leq$ on the elements.

In an open framework, we can distinguish between two kinds of symbols:

- *defining symbols*, namely symbols whose interpretation is left open;
- *defined symbols*, namely symbols whose meaning is defined by the axioms of the framework, possibly in terms of the defining symbols.

In Example 2, *Elem* and $\leq$ are defining symbols, while all the other symbols of the language are defined symbols.

We will write $\mathcal{F} : d_1, \ldots, d_n \Leftarrow p_1, \ldots, p_m$ to indicate that $\mathcal{F}$ is an open framework with *dependency* $d_1, \ldots, d_n \Leftarrow p_1, \ldots, p_m$, i.e. in $\mathcal{F}$, $d_1, \ldots, d_n$ are *defined* symbols and $p_1, \ldots, p_m$ are *defining* symbols. Thus $\mathcal{F} : d_1, \ldots, d_n \Leftarrow$ denotes a closed framework, i.e. all the symbols are defined by the axioms of the framework.

Thus defining symbols act as parameters, and our idea of an open framework is similar to a parametric (algebraic) ADT's. However, unlike the algebraic approach, we use the full first-order language, and therefore we need criteria for discriminating between 'good' and 'bad' frameworks, as we will now discuss.

The two key attributes of a good specification framework are *consistency* and *adequacy*. Consistency has the usual meaning of not containing contradictions, whereas adequacy is related to the dependency of the framework.

Intuitively speaking, a framework is *adequate* wrt a given set of dependencies, if whenever we completely specify the defining symbols, we obtain a complete specification of the defined symbols without adding any new axioms for them. This notion of adequacy wrt a dependency is the first point that we will develop for open frameworks.

3.1 Operations on Frameworks

Since the effort of writing adequate and consistent frameworks, as well as synthesising correct programs in such frameworks, may be considerably high, reusability of both specifications and programs is a very important issue: once we have produced a meaningful, consistent and adequate specification framework and correct and good software for it, we should be able to use them in a wide family of problems. Moreover, we should be able to accumulate and increase our knowledge of the more interesting problem domains, and to enlarge the class of reusable programs, leading to increasingly more complete and useful frameworks. An object-oriented approach, like the one developed in algebraic ADT's, would be very useful in this respect.

So, the second point of our study is the introduction of methods suitable for object-oriented specification and synthesis. We will consider the following operations:

- *Expansion.* Given an adequate and consistent open framework $\mathcal{F} : d_1, \ldots, d_n \Leftarrow p_1, \ldots, p_m$, we can *expand* $\mathcal{F}$, by adding a new (defined) relation symbol r by means of a definition axiom D_r, into a new framework $\mathcal{F} \cup D_r : r, d_1, \ldots, d_n \Leftarrow p_1, \ldots, p_m$.
  Since definition axioms give rise to conservative extensions, the expansion operation preserves consistency, but it may not preserve adequacy.
- *Consolidation.* An adequate open framework $\mathcal{F} : d_1, \ldots, d_n \Leftarrow p_1, \ldots, p_m$ can be *consolidated* in *any* closed framework $\mathcal{G} : p_1, \ldots, p_m, \ldots \Leftarrow$ to obtain the closed framework $\mathcal{F} \cup \mathcal{G} : d_1, \ldots, d_n, p_1, \ldots, p_m, \ldots \Leftarrow$. Such an operation thus enriches $\mathcal{G}$ by the defined symbols of $\mathcal{F}$. We shall call $\mathcal{F} \cup \mathcal{G}$ an *instance* of $\mathcal{F}$ (or, more precisely, an *instantiation* of $\mathcal{F}$ by $\mathcal{G}$).[6]
  This operation preserves adequacy, but it may not preserve consistency.

[6] Rather than the more clumsy alternative of 'a consolidation of $\mathcal{F}$ in $\mathcal{G}$'.

Therefore it is necessary to find conditions under which adequacy is preserved by expansions, and consistency by consolidations. For the latter, it is possible to find some sufficient conditions, but we will not discuss them here.

More interestingly, logic program synthesis is strictly related to the above operations. Firstly, it can be used to expand frameworks while preserving adequacy, as we will show in Section 6. Secondly, in an open framework $\mathcal{F}$, it is possible to synthesise *open* programs, which can be *reused* in every instantiation of $\mathcal{F}$, as we will explain in Section 6.3.

4 Closed Frameworks

Definition 5. A *closed* framework $\mathcal{F}$ with language $L_{\mathcal{F}}$ is an axiomatisation (i.e. a set of axioms $\mathcal{F} \subseteq \mathcal{L}_{\mathcal{F}}$) which satisfies the following properties:

- *Reachability.* There is at least one model of $\mathcal{F}$ reachable by a subset of the constant and function symbols of $L_{\mathcal{F}}$, called the *construction symbols*. The ground terms containing only construction symbols will be called *constructions*.
- *Freeness.* $\mathcal{F}$ proves the *freeness axioms* [13] for the construction symbols.
- *Atomic completeness.* $\mathcal{F}$ is atomically complete.

Thus, by Theorem 4, a closed framework is a theory admitting an isoinitial model. The existence of a set of construction symbols satisfying the freeness axioms is not strictly necessary (reachability sufficies). It is required here, since we deal with synthesis of logic programs and use Clark's equality theory[7] [3] for construction symbols. Of course, in logic programs only construction symbols can be used for constant and function symbols.

Example 4. As we have already seen, $\mathcal{PA}$ is a closed framework. The construction symbols in $\mathcal{PA}$ are '0' and 's', and the constructions are the numerals $0, s(0), s(s(0)), \ldots$ Note that sum '+' and product '$\cdot$' are not construction symbols, so they need not be considered in freeness axioms. Finally, $\mathcal{PA}$ is atomically complete. Indeed the closed atomic formulas of $\mathcal{PA}$ are of the form $t_1 = t_2$, where t_1, t_2 are closed terms, and, as is well-known, $\mathcal{PA} \vdash t_1 = t_2$ (if t_1 and t_2 denote the same natural number), or $\mathcal{PA} \vdash \neg t_1 = t_2$ (if they denote different numbers).

All the isoinitial models of a framework are isomorphic. For every closed framework $\mathcal{F}$, using the constructor symbols, we can build a particular recursive, isoinitial model I, that we will call the *canonical model*, in the following way:

- For every sort S, the domain of S is the set of constructions of sort S.
- A function symbol f is interpreted in I as the function f_I defined thus: for every tuple a of constructions (of the appropriate sorts), the value of $f_I(a)$ is the construction b such that $\mathcal{F} \vdash f(a) = b$.

[7] Containing freeness axioms.

– Every relation symbol r is interpreted in I as the relation r_I such that, for every tuple a of constructions, a belongs to r_I iff $\mathcal{F} \vdash r(a)$. In particular, '$=$' is the identity of constructions.

Example 5. In the canonical model of $\mathcal{PA}$, the domain (of its unique sort) is the set of numerals $\{0, s(0), s(s(0)), \ldots\}$, '$+$' is the sum of numerals, e.g. $\mathcal{PA} \vdash s(0) + s(0) = s(s(0))$; '$\cdot$' is the product of numerals, e.g. $\mathcal{PA} \vdash s(0) \cdot s(0) = s(0)$; and '$=$' is the identity of numerals ($\mathcal{PA} \vdash s^n(0) = s^m(0)$ iff $n = m$).

Let $\mathcal{F}$ be a closed framework and I its canonical model. By atomic completeness, for every relation symbol r and every tuple a of constructions, $r(a)$ is true in I iff it is true in every model M of $\mathcal{F}$, and $\neg r(a)$ is true in I iff it is true in every model M of $\mathcal{F}$. Therefore the canonical model is representative of any other model wrt the closed atomic and negated atomic formulas. That is, it yields the *intended semantics* of the function and relation symbols axiomatised by the framework.

4.1 Adequate Expansions of Closed Frameworks

As shown by the examples of the previous sections, definition axioms play a central rôle in our approach. In closed frameworks they are adequate when they completely characterise the defined relations, according to the following definition:

Definition 6. Let $\mathcal{F}$ be a closed framework and D_r

$$\forall \underline{x}(r(\underline{x}) \leftrightarrow R(\underline{x}))$$

be a definition axiom. We say that D_r *completely defines* r in $\mathcal{F}$ iff $\mathcal{F} \cup D_r$ satisfies the atomic completeness property.

If D_r completely defines r in $\mathcal{F}$, then $\mathcal{F} \cup D_r$ is in turn a closed framework. It is thus an *adequate expansion* of $\mathcal{F}$ by a new relation r, in the following sense:

– Let I be the canonical model of $\mathcal{F}$, and I_r that of $\mathcal{F} \cup D_r$. Then I_r is an *expansion* of I by r.
– The interpretation of the new symbol r in the expansion I_r is the following: for every tuple t of constructions (of the appropriate sorts), t belongs to r_{I_r} iff $\mathcal{F} \cup D_r \vdash r(t)$.

We recall that in an expansion of a model (see [2]) by new symbols, the interpretation of the old ones remains unchanged. Thus an adequate expansion $\mathcal{F} \cup D_r$ *preserves the semantics* of the symbols already present in $L_{\mathcal{F}}$.

This nice property holds only for completely defined relations, however. If D_r does not completely define a relation, then $\mathcal{F} \cup D_r$ is no longer a closed framework and a canonical model no longer exists. This means that, in general, we have to possibly reconsider the correctness of programs that have already been synthesised. Thus it is important to recognise the axioms D_r which completely define new relations.

5 Open Frameworks

From a general point of view, an open framework is a consistent set $\mathcal{F}$ of first-order axioms, for which we fix a dependency $d_1, \ldots, d_m \Leftarrow p_1, \ldots, p_m$. We require that the dependency satisfies the following properties:

(i) For every *defined* sort symbol S, identity '=' on S is a defined symbol. Moreover, there is (in the language of the framework) a non-empty set of constant and function symbols of sort S such that the corresponding freeness axioms are provable in $\mathcal{F}$; these are the *construction symbols* of S.
(ii) For every *defining* sort symbol D, identity '=' on D is a defining symbol. Moreover, the language of the framework does not contain constant and function symbols of sort D.

As in the previous section, the closed terms built by construction symbols are called *constructions*. Note that, by (ii), if D is any defining sort symbol, then the set of constructions of sort D is empty.

Example 6. The framework $\mathcal{LIST}$ has the dependency:

$$Nat, List, 0, nil, s, +, \cdot, |, len, elemi \;\Leftarrow\; Elem, \leq$$

We have the identities $=: (Nat, Nat)$, $=: (List, List)$, $=: (Elem, Elem)$; where the first two are defined symbols, whilst the third is a defining symbol.

The construction symbols of $List$ are the constant $nil \;: \rightarrow\; List$ and the function $| \;: (Elem, List) \rightarrow List$, and the only construction of sort $List$ is nil. Note that the only ground term of sort $List$ is nil, since the other ground terms depend on the ground terms of the defining sort $Elem$, and therefore will be known only in particular instances of the framework.

The construction symbols of the sort Nat are '0' and 's' and the constructions of sort Nat are the numerals.

The defining sort $Elem$ has no construction symbols and no constructors.

Let $\mathcal{F} : d_1, \ldots, d_m \Leftarrow p_1, \ldots, p_n$ be an open framework. If $n = 0$ (i.e. no defining symbol is given), then $\mathcal{F}$ is adequate iff it is a closed framework. Otherwise adequacy is defined as follows.

Definition 7. $\mathcal{F} : d_1, \ldots, d_m \Leftarrow p_1, \ldots, p_n$ is *adequate* wrt $d_1, \ldots, d_m \Leftarrow p_1, \ldots, p_n$ iff, for every closed framework $\mathcal{G} : p_1, \ldots, p_n, q_1, \ldots, q_h \Leftarrow$, $\mathcal{F} \cup \mathcal{G}$ is a closed framework such that its canonical model I^* is an expansion of the canonical model I of $\mathcal{G}$ by the new symbols $d_1, \ldots, d_m$.

Let $\mathcal{G} : p_1, \ldots, p_n, q_1, \ldots, q_h \Leftarrow$ be any closed framework that is consistent with $\mathcal{F} : d_1, \ldots, d_m \Leftarrow p_1, \ldots, p_n$; according to Section 3.1, we will say $\mathcal{F} \cup \mathcal{G}$ is an *instance* of $\mathcal{F}$ (or an *instantiation* of $\mathcal{F}$ by $\mathcal{G}$). Adequacy means that every (consistent) instantiation of $\mathcal{F}$ by $\mathcal{G}$ gives rise to an expansion of $\mathcal{G}$ by the symbols defined by $\mathcal{F}$, while preserving the semantics of the old symbols of $\mathcal{G}$.

5.1 Adequate Expansions of Open Frameworks

Using *adequate* open frameworks we have the following advantage: all the work which can be done directly in the open framework is done once and for all, i.e. it can be *reused* in all the instances of the framework. In particular, this holds for *reusable* programs (see Section 6.3) that can be synthesised in such frameworks. This explains why it is interesting to investigate the possibility of performing synthesis in open frameworks. We will show that synthesis of reusable open programs is possible for relations introduced by *adequate definition axioms*, defined as follows.

Definition 8. A definition axiom D_r is *adequate* in an open framework $\mathcal{F} : d_1, \ldots, d_m \Leftarrow p_1, \ldots, p_n$ iff, for every instance $\mathcal{F} \cup \mathcal{G}$, D_r completely defines r in $\mathcal{F} \cup \mathcal{G}$.

Axioms adequate in an open framework have the following nice property: they give rise to adequate expansions of the instances of the framework. More precisely, they add to every instance of the framework a new relation r, while preserving the semantics of the old symbols. Thus, if D_r is an adequate definition axiom, we say that $\mathcal{F} \cup D_r$ is an *adequate expansion* of $\mathcal{F}$ by r.

Note that in adequate expansions, the definition r depends only on the interpretation of the defining symbols in the various instances. Therefore the dependency of an expansion $\mathcal{F} \cup D_r$ is : $r, d_1, \ldots, d_m \Leftarrow p_1, \ldots, p_n$.

Finally, it will be convenient to relate adequacy of frameworks and of their expansions (by adequate definition axioms) to restricted classes of instances. Let Γ be a class of closed frameworks which can be used to instantiate an open framework $\mathcal{F}$. To define *adequacy* wrt Γ, it is sufficient to consider in Definition 7 and Definition 8 only the instantiations by closed frameworks $\mathcal{G} \in \Gamma$.

6 Deductive Synthesis and Specification Frameworks

In this section, first we briefly describe deductive synthesis of logic programs in a specification framework. Then we show how synthesis can be used to determine the adequacy of any expansion of a closed or open framework. For an adequate open framework, we also introduce the notion of open, reusable programs.

6.1 Deductive Synthesis of Logic Programs

First, as in [10], for a program P, we define:

- *free*(P) to be the freeness axioms for the constant and function symbols of P.
- *Comp*$^+(P) =$ *free*$(P) \cup Ax(P)$, where $Ax(P)$ is the *if*-part of *Comp*(P), the completion of P [3].

- $Comp^-(P, r)$ is the *only-if*-part of the completed definition of a predicate r in P:

$$\forall\,(r(x) \rightarrow E_1 \vee \cdots \vee E_k)$$

For convenience, we shall also write $Comp^-(P, r_1, \ldots, r_n)$ for multiple predicate symbols.

Now, let $\mathcal{F}$ be a general (open) framework and consider a specification consisting of a definition axiom D_0

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

and a set G of goals.

A *deductive synthesis* process (for this specification) generates a sequence of programs $P_1 \subseteq P_2 \subseteq \cdots \subseteq P_k \subseteq \cdots$, where, for every P_j, there is a corresponding set D_j of definition axioms, introducing the new predicate symbols of P_j .

We assume that the synthesis process ensures that at each step:

(i) $\mathcal{F} \cup D_j \vdash Comp^+(P_j)$;
(ii) P_j terminates , i.e. it either finitely fails or its *SLD*-tree has at least one success node, for every goal in G.

(i) ensures *partial correctness*. However, (i) and (ii) together do not ensure *total correctness*. In Sections 6.2 and 6.3, we will define the notion of total correctness in closed and open frameworks precisely. In the meantime, we state a criterion for halting synthesis with a totally correct program, which we presented in [10].

Definition 9. For a program P, let $r_1, \ldots, r_k$ be the relation symbols occurring in the head of at least one clause of P, and $q_1, \ldots, q_m$ be the ones occurring only in the bodies. Following our terminology for frameworks (without causing confusion), we say that P has the *dependency* $r_1, \ldots, r_k \Leftarrow q_1, \ldots, q_m$, and we write $P : r_1, \ldots, r_k \Leftarrow q_1, \ldots, q_m$. We call $r_1, \ldots, r_k$ the *defined predicates* and $q_1, \ldots, q_m$ the *defining predicates* of P.[8]

In every synthesis step j, we consider the relation symbols introduced by D_j as *defined symbols* of $\mathcal{F} \cup D_j$. Moreover, we require that the dependency of P_j *agrees* with that of $\mathcal{F} \cup D_j$ wrt the defined symbols, namely that the defined predicates of P_j are defined symbols of $\mathcal{F} \cup D_j$. A criterion for halting synthesis is then the following:

A Criterion for Halting Synthesis.
Let $P_j : r_1, \ldots, r_k \Leftarrow q_1, \ldots, q_m$ be the program synthesised in the current step j.
(i) If, for a defined predicate r_h, $\mathcal{F} \cup D_j \vdash Comp^-(P_j, r_h)$, then stop searching for clauses with head r_h.
(ii) If (i) holds for every defined predicate of P_j, and all the defining predicates $q_1, \ldots, q_m$ are defining symbols in $\mathcal{F}$, then halt synthesis.

[8] A defined predicate may occur both in the head and (*recursively*) in the body of a clause, whereas a defining one may occur *only* in the body.

As we mentioned in Section 3.1, an adequate (open) framework $\mathcal{F} : d_1, \ldots, d_n \Leftarrow p_1, \ldots, p_m$ completely characterises the defined symbols $d_1, \ldots, d_n$ in terms of the possible interpretations for the defining symbols $p_1, \ldots, p_m$. Correspondingly, a correct open program P should be a program computing some defined relation symbols of the framework, depending on the interpretation of other defining relation symbols. When (i) holds, the completed definition of r_h has been proved in the framework. Roughly speaking, this implies that we have already synthesised all the clauses to compute r_h, depending on the interpretation of the other relation symbols occurring in their bodies.

When (ii) holds, the open program P_j contains *all* the clauses to compute *all* its defined relations, depending *only* on the interpretation of the defining symbols of the framework. This implies *reusability*: for every model M of the framework, it is sufficient to add to P_j the clauses that correctly compute the defining relations in M, to obtain a complete program that is totally correct in M.

On the other hand, if such a P_j can be synthesised, then, as we will see in Section 6.3, the framework is adequate, i.e. synthesis can be used to analyse the adequacy of specification frameworks, and thus we have a useful feedback mechanism between specification and synthesis.

Synthesis in closed frameworks is a limiting case, where we halt synthesis only with programs with dependency $r_1, \ldots, r_k \Leftarrow$, since there are no defining symbols in the framework. We will prove that in this case we obtain (closed) programs which are totally correct (in the usual sense) in the intended model of the framework. Thus in a closed framework, program synthesis can be used both to find correct programs and to study adequate expansions, as we now show more formally.

6.2 Synthesis and Closed Frameworks

Let $\langle D_r, G \rangle$ be a specification in a closed framework $\mathcal{F}$ with canonical model I. We are interested in *adequate specifications*, namely specifications giving rise to adequate expansions. In an adequate specification, D_r completely defines r in $\mathcal{F}$, and $\mathcal{F} \cup D_r$ is a closed framework with a canonical model I_r. Thus the following definition is sensible:

Definition 10. For a given specification $\langle D_r, G \rangle$ and a given framework $\mathcal{F}$, a program P is *totally correct* in the *expansion* $\mathcal{F} \cup D_r$ iff it is totally correct in the canonical model I_r.

For the synthesis of totally correct programs, we can prove the following theorems.

We say that a program P *ground terminates* if, for every relation symbol r of P, and every ground goal $r(t)$, the *SLD*-tree for $P \cup \{\leftarrow r(t)\}$ contains at least one success node or is finitely failed.[9] The following theorem holds (see [10]):

[9] We assume a fair computation rule.

Theorem 11. *Let $\langle D_0, G\rangle$ be a specification in a closed framework $\mathcal{F}$, where r_0 is the relation defined by D_0. In a synthesis process, if a ground terminating program $P_j : r_0, \ldots r_k \Leftarrow$ can be synthesised such that $\mathcal{F} \vdash \mathrm{Comp}^-(P, r_0, \ldots r_k)$, then we have:*

(i) *$\mathcal{F} \cup D_j$ is an adequate expansion of $\mathcal{F}$ by the symbols introduced in D_j;*
(ii) *P_j is totally correct in the canonical model of $\mathcal{F} \cup D_j$ wrt the set of ground goals.*

This theorem is useful in a synthesis process where we do not know whether the specification $\langle D_0, G\rangle$ we start from is adequate or not. In such a case, we can use synthesis both to obtain a program answering the goals in G and to prove the adequacy of the specification. For every step j, we prove $Comp^+(P_j)$ and the termination of P_j, both for the goals in G and for all the ground goals $r_h(t)$.[10] We halt the synthesis process when we obtain a program $P_j : r_0, \ldots, r_k \Leftarrow$ such that $\mathcal{F} \cup D_j \vdash Comp^-(P_j, r_0, \ldots, r_k)$. If this process halts successfully, then we obtain an adequate expansion of our framework (by the relation symbols defined in D_j) and a totally correct program.

However, it may happen that during this process, we reach the following situation of uncertainty: the current P_j is such that

neither $\mathcal{F} \cup D_j \vdash \neg Comp^-(P_j, r_0, \ldots, r_k)$ nor $\mathcal{F} \cup D_j \vdash Comp^-(P_j, r_0, \ldots, r_k)$.

The following theorem[11] allows us to handle such a situation. In the theorem, 'P_j is totally correct in $\mathcal{F} \cup D_j$ wrt ground goals' means that, for every relation symbol r_i of P_j and every ground atom $r_i(t)$, $r_i(t)$ is true in the canonical model of $\mathcal{F} \cup D_j$ iff the *SLD*-tree for $P \cup \{\leftarrow r_i(t)\}$ contains at least one success node.

Theorem 12. *Let $P_j : r_0, \ldots, r_k \Leftarrow$ be the current program, and let D_j be definition axioms that completely define $r_0, \ldots, r_k$ in $\mathcal{F}$. If P_j is totally correct in $\mathcal{F} \cup D_j$ wrt ground goals, then $\mathrm{Comp}^-(P_j, r_0, \ldots, r_k)$ is consistent with $\mathcal{F} \cup D_j$, and can be added to it whilst preserving the canonical model.*

According to this theorem, in case we reach the above situation of uncertainty, then we study the definition axioms D_j. If we can prove (in some way) that D_j completely define the corresponding relations, then by Theorem 12 we can add $Comp^-(P_j, r_0, \ldots, r_k)$ as a new axiom, and we obtain both a totally correct program and an adequate expansion. Note that in this case we also have an improvement of the framework, since we add a stronger axiom.

On the other hand, if we discover that some definition axiom in D_j does not completely define the corresponding relation, we have to reconsider both the synthesis process and the framework. Indeed, it may be that in some synthesis step we made a bad 'choice', or that our framework is too weak to define the relations we want to compute.

---

[10] Ground termination is needed to apply Theorem 11.
[11] We omit the proof here for brevity.

6.3 Synthesis and Open Frameworks

To treat the correctness (i.e. the reusability) of an open program in an open framework, it is useful to compare the various Herbrand models of the program with the canonical models of the various instances of the framework. To this end, we introduce the following definitions.

Let M be a model for a language L_M and let C be a set of constant and function symbols.

We say that M is *C-generable* if it is reachable by C, and the freeness axioms of the function and constant symbols of C are satisfied in M. The closed terms built from C are the *constructions*.

Let M be a C-generable model with language L_M and let r be a relation symbol of L_M. The *positive diagram* of r in M is defined by:

$$diag^+(M, r) = \{r(t) \mid t \text{ are constructions and } M \models r(t)\}$$

For many relations, $diag^+(M, r_1, \ldots, r_k)$ is defined in the obvious way.

Now we can compare the positive diagrams with the Herbrand models of programs, and we introduce and define *steadfastness* (as in [10]) as follows:

Definition 13. Let $\mathcal{M}$ be a class of models, and assume that, for every model $M \in \mathcal{M}$, there is a set C_M of construction symbols of L_M such that M is C_M-generable. We say that

- M is a *steadfast model* of a program $P : r_1, \ldots, r_k \Leftarrow q_1, \ldots, q_m$ iff the constant and function symbols of P belong to C_M, and $diag^+(M, r_1, \ldots, r_k, q_1, \ldots, q_m)$ is the minimum Herbrand model of $Comp^+(P) \cup diag^+(M, q_1, \ldots, q_m)$;
- P is *steadfast* in $\mathcal{M}$ of models iff every $M \in \mathcal{M}$ is a steadfast model of P.

Steadfast programs are *reusable* in the following sense.

Property 14. *Let the program $P : r_1, \ldots, r_k \Leftarrow q_1, \ldots, q_m$ be steadfast in a class $\mathcal{M}$ of models, and let M be any model belonging to $\mathcal{M}$. For every program Q_M totally correct wrt $q_1, \ldots, q_m$ in M, if $P \cup Q_M$ terminates, then $P \cup Q_M$ is totally correct in M wrt $r_1, \ldots, r_k, q_1, \ldots, q_m$.*

Thus steadfastness of P works as a kind of 'parametric correctness' in a class of models, where P assumes in each model M the appropriate behaviour that depends on Q_M.

Now we introduce the notion of steadfastness in a class of instances of a framework (as defined in the previous subsection), as follows.

Let $\mathcal{F}$ be an open framework that is adequate wrt a class Γ of closed frameworks. Thus, for every $\mathcal{G} \in \Gamma$, the instance $\mathcal{F} \cup \mathcal{G}$ is a closed framework with a canonical model $I_{\mathcal{F}\cup\mathcal{G}}$. Let $\mathcal{I}_\Gamma$ be the class of canonical models $I_{\mathcal{F}\cup\mathcal{G}}$ such that $\mathcal{G} \in \Gamma$: we say that a program P is steadfast in the class of Γ-instances of $\mathcal{F}$ iff it is steadfast in $\mathcal{I}_\Gamma$.

Synthesis can be used to inform on the adequacy of expansions of open frameworks by new relation symbols, as stated by following theorem.

Theorem 15. *Let $\mathcal{F} : d_1, \ldots, d_n \Leftarrow p_1, \ldots, p_m$ be an open framework, and consider a class Γ of closed frameworks. Let $P_j : r_0, \ldots, r_k \Leftarrow q_1, \ldots, q_h$ be a synthesised program, and assume that all the defining predicates of P_j are defining symbols of $\mathcal{F}$. If*

(i) $\mathcal{F} \cup D_j \vdash \mathrm{Comp}^-(P_j, r_1, \ldots, r_k)$*; and*
(ii) for every $\mathcal{G} \in \Gamma$, there exists at least one program $Q : q_1, \ldots, q_m, \ldots \Leftarrow$, such $\mathcal{G} \vdash \mathrm{Comp}^-(Q, q_1, \ldots, q_m)$ and $P_j \cup Q$ terminates at least for the ground goals in $r_1, \ldots, r_k$;

then

(i) P_j is steadfast in the class of Γ-instances of $\mathcal{F}$; and
(ii) the definition axioms D_j are adequate in that class.

The proof is very simple. It follows from the fact that $\mathcal{P} \cup \mathcal{Q} \vdash \mathit{Comp}^-(P_j \cup Q, r_1, \ldots, r_k, q_1, \ldots, q_m)$, and from Theorem 11. More general facts about steadfast programs could be proved (see [10]), but they are not directly applicable to the methodology outlined here.

By Theorem 15, if we halt synthesis according to the criterion given in Section 6.1, then we obtain programs which are steadfast (and hence reusable) in the class of instances where a terminating $P_j \cup Q$, satisfying the hypotheses of the theorem, exists. This class may be a subclass of the class of all the instances. However, the theorem makes sense, since it applies to the class of instantiations by any framework $\mathcal{G}$ that can be obtained by our methodology and that (hence) contains a synthesised program $Q : q_1, \ldots, q_m, \ldots \Leftarrow$, such $\mathcal{G} \vdash \mathit{Comp}^-(Q, q_1, \ldots, q_m)$.

To obtain reusable programs P in this sense, a crucial point is the possibility of stating the termination of $P \cup Q$, for the various possible Q's, once and for all in the open framework. That is, we have to state termination of $P \cup Q$ without knowing Q, but only knowing some possible properties which can be expressed by the axioms of the open framework.

An example of parametric termination is given by the following program $P : p \Leftarrow q$:

$$\begin{array}{ll} p(X, 0, 0) & \leftarrow \\ p(X, s(Y), W) & \leftarrow p(X, Y, Z), q(Z, X, W) \end{array}$$

where the set of goals is $G = \{\exists Z.p(m, n, Z) \mid m, n \text{ are numerals}\}$. To obtain parametric termination, it is sufficient that the programs $Q : q, \ldots \Leftarrow$ used in the instantiations do not contain the predicate p. This requirement is sensible in our general approach, where p is to be computed by the reusable program $P : p \Leftarrow q$, and not by $Q : q, \ldots \Leftarrow$.

Another example is given by parametric sorting programs on lists $P : \mathit{sort} \Leftarrow \leq$. Many other examples can be given, coming from everyday programming practice. We intend to study in future general techniques to state parametric termination in relevant classes.

7 Conclusion

In this paper, we have described specification frameworks and their relationship with deductive synthesis of logic programs. We have shown that such frameworks are necessary to formalise not only the background knowledge needed for synthesis, but also notions related to reusable programs and their synthesis.

Hitherto, research in logic program synthesis has made use of *closed* specification frameworks in which all relation symbols are completely defined. However, such frameworks cannot capture the notion of reusable or steadfast programs. To do so, it is necessary to use *open* frameworks in which the meaning of some relation symbols is open, i.e. it can vary depending on a chosen model of the framework. Open frameworks thus provide a suitable backdrop to the synthesis of steadfast programs, and are therefore vital for an object-oriented approach to specification and synthesis. We have described such an approach, similar to the one developed in algebraic ADT's, for progam synthesis using the full first-order language and isoinitial semantics.

The relationship between synthesis and specification frameworks is very much a two-way affair. As well as reasoning 'forwards' from a specification (in a framework) via synthesis to the specified program, we can use the result of synthesis to inform on and improve the specification framework if necessary, i.e. we can reason 'backwards' from the program via synthesis to the specification framework.

Synthesis in a closed framework can tell us if the initial specification completely defines the specified relation. It can also enhance the framework by adding stronger axioms to it that correspond to the synthesised program.

Similarly, synthesis in an open framework can be used to determine if the initial specification is adequate, i.e. if it defines an adequate expansion of the framework.

In summary, the results presented and discussed in this paper should provide important foundations for our future work on object-oriented deductive synthesis of logic programs.

Acknowledgements

We are very grateful to the referees for their valuable comments and constructive suggestions which have enabled us to vastly improve this paper.

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. C.C. Chang and H.J. Keisler. *Model Theory*. North-Holland, 1973.
3. K.L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293-322. Plenum Press, 1978.

4. K.L. Clark and S.-Å. Tärnlund. A first order theory of data and programs. *Proc. IFIP 77*, pages 939–944. North-Holland, 1977.
5. Y. Deville and K.K. Lau. Logic program synthesis. To appear in *J. Logic Programming*, special issue on "Ten Years of Logic Programming", 1994.
6. J.A. Goguen and J. Meseguer. EQLOG: Equality, types,and generic modules for logic programming. In D. DeGroot and G. Lindstrom, editors, *Logic Programming: Functions, Relations, and Equations*, pages 295–363. Prentice-Hall, 1986.
7. J.A. 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.
8. J.A. Goguen, J.W. Thatcher and E. Wagner. An initial algebra approach to specification, correctness and implementation. In R. Yeh, editor, *Current Trends in Programming Methodology, IV*, pages 80-149. Prentice-Hall, 1978.
9. K.K. Lau and M. Ornaghi. An incompleteness result for deductive synthesis of logic programs. In D.S. Warren, editor, *Proc.* 10^{th} *Int. Conf. on Logic Programming*, pages 456–477, MIT Press, 1993.
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. J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2nd edition, 1987.
12. Z. Manna and R. Waldinger. *The Deductive Foundations of Computer Programming*. Addison-Wesley, 1993.
13. J.C. Shepherdson. Negation in Logic Programming. in J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 19-88. Morgan Kaufmann, 1988.
14. M. Wirsing. Algebraic specification. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 675–788. Elsevier, 1990.