A Formal View of Specification, Deductive Synthesis and Transformation of Logic Programs

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

Mario Ornaghi
Dipartimento di Scienze dell'Informazione
Universita' degli studi di Milano
Via Comelico 39/41, Milano, Italy
ornaghi@imiucca.csi.unimi.it

Abstract

We view logic programs as (first-order) axiomatisations, and deductive synthesis and transformation as operations on mathematical systems composed of axioms and "logical" systems such as SLD and SLDNF. In this framework, we distinguish between specification, deductive synthesis and transformation and discuss their inter-relationships.

1 Introduction

LOPSTR 92 concluded with a discussion on the distinction (if any) between synthesis and transformation. Most people accepted that the two areas were distinct (if not disjoint). However, it was not clear whether the distinction is mainly based on pragmatic considerations or on more fundamental reasons.

In this paper, we will propose and formalise a distinction between *specification*, *synthesis*, and *transformation*. This distinction is based on a formal framework described in [7] and [8]. In this framework, we view logic programs as (first-order) axiomatisations. Specification, synthesis and transformation are operations on mathematical systems which are composed of axioms and logical systems such as Classical Logic (*CL*), *SLD* and *SLDNF*. The distinction between specification, synthesis and transformation arises from the different behaviours of the mathematical systems involved in these operations.

However, we will also show that despite this distinction, there are close inter-relationships between specification, synthesis, and transformation via *deductive synthesis*. In this approach, the specification is given in the form of a set of axioms, and the synthesis process systematically derives a set of clauses which are logical consequences of these axioms.[1] Examples of deductive synthesis

[1] Note that Manna and Waldinger [10] use *deductive synthesis* to mean *proofs-as-programs*.

methods include [4, 5, 6].

In Section 2 we give an informal overview of our terminology and our proposed distinction. In Section 3 we summarise the formal terminology for our formal framework described in Section 4. Then we give a formalisation of deductive synthesis in Section 5, which provides a basis for our accounts of specification in Section 6, synthesis in Section 7, transformation in Section 8, and their inter-relationships in Section 9.

2 Overview

In this section we give an informal overview of our terminology and our proposed distinction. The formal terminology will be explained in the next section.

2.1 Specification

A *specification* is a triple

$$\langle S \cup D, Ans(G), CL \rangle$$

where CL, S, D and $Ans(G)$ have the following meaning.

CL is some inference system for first order classical logic. It indicates the *logic system* employed at the specification level.

S is a set of (first-order) axioms, called a *specification framework*. It should be as rich and complete as possible, to codify our knowledge of the problem domain, to define in a natural way many problems and to prove properties relevant for program synthesis. In this context, various induction schemas for instance are very useful.

D is an axiom defining in S the relation to be computed, G is a *set of goal formulas* involving such a relation and $Ans(G)$ is the set of the *possible answers* of G. By D we single out one of the problems definable in S, while by $Ans(G)$ we specify the following behaviour of the logic program that we wish to synthesise: "for every goal $g \in G$, search for the formulas of $Ans(g)$ provable in S".[2]

An example specification might be

$$\langle S_{list} \cup D_{prefix}, Ans(\{prefix(M,n) \mid n \text{ is ground}\}), CL \rangle \qquad (1)$$

where S_{list} is a (sufficiently rich) theory of lists, for instance containing identity axioms, axioms for list operations, induction schema on lists, etc; D_{prefix} is the *definition axiom* (called a *logic description* in [1]):

$$prefix(L,M) \leftrightarrow \forall i \leq length(L).\, L[i] = M[i]\ ;$$

Ans is the function associating with every goal $prefix(M,n)$ the set of formulas of the form $prefix(\tau, n)$ (τ being any term). Not every possible answer is a

[2] By G and Ans we can specify things like directionality [1], pre- and post-conditions, and other possible requirements.

valid answer. For example, $\mathit{prefix}(M, nil)$ has possible answers $\mathit{prefix}(nil, nil)$, $\mathit{prefix}(a.nil, nil)$, $\mathit{prefix}(b.a.nil, nil)$, and so on, but only the first one is an answer.

2.2 Synthesis

Synthesis is defined as getting a logic program P from $\langle S, Ans(G), CL\rangle$ such that

$$\langle S, Ans(G), CL\rangle \asymp \langle Ax(P), Ans(G), SLD\rangle$$

i.e. a formula of $Ans(G)$ has a proof in S using CL iff it has a proof in $Ax(P)$ using SLD. Here, $Ax(P)$ denotes the axiomatisation that corresponds to P, and we use SLD in a non-standard manner to denote a logical system (see Section 4.2).

For example, from the specification (1), we can get (somehow) the program P

$$\begin{array}{rcl} \mathit{prefix}(nil, M) & \leftarrow & \\ \mathit{prefix}(X.L_1, X.L_2) & \leftarrow & \mathit{prefix}(L_1, L_2) \end{array}$$

such that

$$\begin{array}{rl} & \langle S_{list} \cup D_{\mathit{prefix}}, Ans\{\mathit{prefix}(M, n) \mid n \text{ ground}\}, CL\rangle \\ \asymp & \langle Ax(P), Ans\{\mathit{prefix}(M, n) \mid n \text{ ground}\}, SLD\rangle \end{array}$$

Thus we view a specification $\langle S, Ans(G), CL\rangle$ as the definition of a problem in a general (possibly infinite) framework S, from which by synthesis we get $Ax(P)$ as a *small* (finite) subsystem, just sufficient to (efficiently) compute the required answers of the goal formulas. That is, S 'properly contains' $Ax(P)$ in an informal sense.

In general, specification frameworks are infinite first order axiomatisations; then every synthesised program is necessarily a smaller subsystem. This fact and the stronger incompleteness result of [8] show that specifications and programs are different[3] and that synthesis is a process of specialisation.

2.3 Transformation

Transformation is defined as getting a program P_2 from a given program P_1, such that P_1 and P_2 are equivalent programs. Then transformation maps programs into programs. Moreover, while synthesis extracts a small subsystem, transformation often goes through systems *equivalent* in CL, i.e. such that

$$\langle Ax(P_1) \cup D, CL\rangle \asymp \langle Ax(P_2) \cup D, CL\rangle$$

where D is a set of definition axioms, linking the predicate symbols of the two programs.

[3] Note that if S is finite, then there exists a logic program P whose completion is equivalent to S (see, e.g. [9, 11]) and this difference disappears.

For example, from the program P_1 (to compute Fibonacci numbers)

$$\begin{array}{rcl} f(0,s(0)) & \leftarrow & \\ f(s(0),s(0)) & \leftarrow & \\ f(s(s(I)),Z) & \leftarrow & f(I,Z_1), f(s(I),Z_2), +(Z_1,Z_2,Z) \end{array}$$

using the definition axiom

$$D \;:\; \forall x,y,z.(ff(x,y,z) \leftrightarrow f(x,y) \wedge f(s(x),z))$$

we can get the program P_2

$$\begin{array}{rcl} ff(0,s(0),s(0)) & \leftarrow & \\ ff(s(I),A,B) & \leftarrow & ff(I,H,A), +(I,H,B) \\ f(X,A) & \leftarrow & ff(X,A,B) \end{array}$$

such that P_1 and P_2 compute the same relation f. Note that

$$\langle Ax(P_1) \cup D, CL\rangle \asymp \langle Ax(P_2) \cup D, CL\rangle$$

i.e. in this example, transformation yields an axiomatisation $Ax(P_2)$ equivalent (using D) to the starting one.

Of course, there may be transformations that work as program specialisation, as discussed in the next subsection.

2.4 Partial Evaluation

Although we shall not discuss partial evaluation in this paper, it is interesting to see where it fits in our scheme of things. We define *partial evaluation* as follows: for a given program P_1 and a chosen goal I, using I as an input for partial evaluation, get a program P_2 such that

$$\langle Ax(P_1), Ans(I), SLD\rangle \asymp \langle Ax(P_2), Ans(I), SLD\rangle$$

i.e. for the answers of the chosen goal I, P_1 and P_2 should be 'equivalent'. In general $\langle Ax(P_2), CL\rangle$ is a proper subsystem of $\langle Ax(P_1), CL\rangle$. Thus we view partial evaluation as a form of *program specialisation*.

If we use *spec* and *prog* to denote sets of all specifications and programs respectively, we have

specification *specialisation*	$\equiv$	*spec* $\rightarrow$ *prog*
	$\equiv$	program synthesis
program *specialisation*	$\equiv$	*prog* $\rightarrow$ *prog*
		e.g. partial evaluation
program *transformation*	$\equiv$	*prog* $\rightarrow$ *prog*

Thus we regard partial evaluation as program transformation rather than as program synthesis.

Note that we also have

specification *transformation* $\equiv$ *spec* $\rightarrow$ *spec*

even though here we do not deal with specification transformation at all.

3 Formal Terminology

In this section we present a brief summary of the basic terminology for logical and mathematical systems.[4]

A *logical system* $\mathcal{L}$ is characterised by the set $\mathbf{P}_{\mathcal{L}}$ of the *proofs derivable in* $\mathcal{L}$. Every proof $\pi \in \mathbf{P}_{\mathcal{L}}$ has the attributes a, c where

- $a(\pi)$ is a set of formulas called the *assumptions* of π;
- $c(\pi)$ is a formula called the *consequence* of π.

For example, in a sequent system, if $A_1, \ldots, A_n \Rightarrow B_1, \ldots, B_m$ is the end sequent of a proof π, then $a(\pi) = \{A_1, \ldots, A_n\}$, $c(\pi) = B_1 \vee \cdots \vee B_n$.

For two logical systems $\mathcal{L}_1$ and $\mathcal{L}_2$, $\mathcal{L}_1$ is a *subsystem* of $\mathcal{L}_2$, or $\mathcal{L}_1 \sqsubseteq \mathcal{L}_2$,[5] iff there is a function

$$\rho \ : \ \mathbf{P}_{\mathcal{L}_1} \rightarrow \mathbf{P}_{\mathcal{L}_2}$$

such that for every $\pi \in \mathbf{P}_{\mathcal{L}_1}$:

$$a(\pi) = a(\rho(\pi)) \ \text{ and } \ c(\pi) = c(\rho(\pi)) \ .$$

$\mathcal{L}_1$ and $\mathcal{L}_2$ are *equivalent*, or $\mathcal{L}_1 \asymp \mathcal{L}_2$, iff $\mathcal{L}_1 \sqsubseteq \mathcal{L}_2$ and $\mathcal{L}_2 \sqsubseteq \mathcal{L}_1$.

A set of formulas Γ *proves* C in a logical system $\mathcal{L}$, or $\Gamma \vdash_{\mathcal{L}} C$, iff there is at least a proof $\pi \in \mathbf{P}_{\mathcal{L}}$ such that

$$a(\pi) \subseteq \Gamma \ \text{ and } \ C = c(\pi) \ .$$

Note that if $\mathcal{L}_1 \sqsubseteq \mathcal{L}_2$, then $\Gamma \vdash_{\mathcal{L}_1} C$ implies that $\Gamma \vdash_{\mathcal{L}_2} C$.

A *mathematical system* is a pair $\langle M, \mathcal{L} \rangle$, where M is a set of mathematical axioms[6] of a first-order language L_M (the signature of L_M contains the relation and function symbols axiomatised by M) and $\mathcal{L}$ is a logical system.

We say that π is a *proof* in $\langle M, \mathcal{L} \rangle$ iff π is a proof in $\mathcal{L}$ and $a(\pi) \subseteq M$. We shall use $\mathbf{P}_{\langle M, \mathcal{L} \rangle}$ to denote the set of the proofs in $\langle M, \mathcal{L} \rangle$.

The consequence $c(\pi)$ of a proof in $\langle M, \mathcal{L} \rangle$ will be called a *theorem* of $\langle M, \mathcal{L} \rangle$. To indicate that a formula H is a theorem of $\langle M, \mathcal{L} \rangle$, we will write $\langle M, \mathcal{L} \rangle \vdash H$.

For two mathematical systems $\langle M_1, \mathcal{L}_1 \rangle$ and $\langle M_2, \mathcal{L}_2 \rangle$ with (possibly) different languages L_{M_1} and L_{M_2}, $\langle M_1, \mathcal{L}_1 \rangle$ is *representable* in $\langle M_2, \mathcal{L}_2 \rangle$, or $\langle M_1, \mathcal{L}_1 \rangle \sqsubseteq \langle M_2, \mathcal{L}_2 \rangle$,[7] if there is a translation ρ_L of L_{M_1} into L_{M_2} and a function

$$\rho \ : \ \mathbf{P}_{\langle M_1, \mathcal{L}_1 \rangle} \rightarrow \mathbf{P}_{\langle M_2, \mathcal{L}_2 \rangle}$$

such that, for every $\pi \in \mathbf{P}_{\langle M_1, \mathcal{L}_1 \rangle}$, $\rho_L(c(\pi)) = c(\rho(\pi))$. Of course, the relation between the two systems depends on the properties of ρ_L. We will consider only functions ρ_L such that, for every $H \in L_{M_1} \cap L_{M_2}$, $\rho_L(H) = H$.

[4] For the sake of simplicity, we assume that all our systems work on formulas of first-order logic with equality.

[5] The relation $\sqsubseteq$ is reflexive and transitive.

[6] A mathematical axiom is any closed formula that is not a logical axiom.

[7] The relation $\sqsubseteq$ is reflexive and transitive.

This implies that, in the common language, every theorem of the first system is also a theorem of the second one.

$\langle M_1, \mathcal{L}_1\rangle$ and $\langle M_2, \mathcal{L}_2\rangle$ are *equivalent*, or $\langle M_1, \mathcal{L}_1\rangle \asymp \langle M_2, \mathcal{L}_2\rangle$, iff $\langle M_1, \mathcal{L}_1\rangle \sqsubseteq \langle M_2, \mathcal{L}_2\rangle$ and $\langle M_2, \mathcal{L}_2\rangle \sqsubseteq \langle M_1, \mathcal{L}_1\rangle$. This implies that, in the common language, the two systems have the same set of theorems.

Two relevant and remarkable examples of equivalence are:

$$M \vdash_{CL} N \Rightarrow \langle M \cup N, CL\rangle \asymp \langle M, CL\rangle \tag{2}$$

$$\langle M \cup \{\forall x.\,(r(x) \leftrightarrow F(x))\}, CL\rangle \asymp \langle M, CL\rangle\ . \tag{3}$$

(2) and (3) are well-known examples of *conservative extensions*.[8]

Finally we shall use

$$\langle M, \mathcal{A}, \mathcal{L}\rangle$$

(where $\mathcal{A}$ is a set of formulas), to denote the subsystem of $\langle M, \mathcal{L}\rangle$ such that: π is a proof in $\langle M, \mathcal{A}, \mathcal{L}\rangle$ iff π is a proof in $\langle M, \mathcal{L}\rangle$ and $c(\pi) \in \mathcal{A}$. In other words, the proofs in $\langle M, \mathcal{A}, \mathcal{L}\rangle$ are just the proofs in $\langle M, \mathcal{L}\rangle$ of formulas in $\mathcal{A}$. We call $\langle M, \mathcal{A}, \mathcal{L}\rangle$ the *restriction* of $\langle M, \mathcal{L}\rangle$ to $\mathcal{A}$.[9]

Restrictions of systems will be called *restricted systems*. For such systems, the definitions of $\sqsubseteq$ and $\asymp$ are as before.

4 A Formal Framework for Deductive Synthesis

In this section, we summarise our framework for deductive synthesis of logic programs in [7, 8]. For logic programming notation, we shall follow [9] closely. Atoms will be denoted by $A, A_1, \ldots$, literals by $L, L_1, \ldots$, clauses by $C, C_1, \ldots$, and substitutions by $\sigma, \theta, \ldots$ However, goals will be denoted by $\leftarrow G, \leftarrow G_1, \ldots$, where $G, G_1, \ldots$ are sequences of atoms or literals.

4.1 Logic Programs as Axiomatisations

In our formal framework, logic programs are regarded as (first-order) axiomatisations.

For a *definite* program P_D, the corresponding (finite) set of axioms, denoted by $Ax(P_D)$, is given by

$$Ax(P_D) = \{Ax(C) \mid C \text{ is a clause of } P_D\}$$

where[10]

$$\begin{array}{rcl} Ax(A \leftarrow) & = & \forall(A) \\ Ax(A \leftarrow A_1, \ldots, A_k) & = & \forall(A_1 \wedge \cdots \wedge A_k \rightarrow A)\ . \end{array}$$

[8] In (3), $\forall x.\,(r(x) \leftrightarrow F(x))$ is a *definition axiom* which introduces a new relation symbol $r(x)$.

[9] Note that some formulas in $\mathcal{A}$ may not have proofs in $\langle M, \mathcal{L}\rangle$ at all. In a specification, $\mathcal{A}$ is $Ans(G)$.

[10] We shall use $\forall(F)$ to denote the universal closure of a formula F.

For a *normal* program P_N, the corresponding axiomatisation is just its completion $Comp(P_N)$. However, for our purposes it will be more convenient, as will become evident later, to use a set of axioms which is logically equivalent to $Comp(P_N)$. We shall call this equivalent set $Comp^*(P_N)$, it contains:

- *Equality axioms.*[11]
- *Success axioms.* For every normal clause C of P_N, $Ax(C)$ (as defined above) is a success axiom of $Comp^*(P_N)$.
- *Default axioms.* For every predicate symbol $p(\underline{x})$, if there is no clause of P_N containing p in its head, then $\forall \underline{x}.\, \neg p(\underline{x})$ is a default axiom of $Comp^*(P_N)$.
- *Failure axioms.* For every predicate symbol $p(\underline{x})$ occurring in the head of some clause, if

  $$\forall \underline{x}.\, (p(\underline{x}) \leftrightarrow E_1 \vee \cdots \vee E_k)$$

  is the completed definition of $p(\underline{x})$, then

  $$\forall \underline{x}.\, (\neg E_1 \wedge \cdots \wedge \neg E_k \rightarrow \neg p(\underline{x}))$$

  belongs to $Comp^*(P_N)$. We call this the failure axiom of $p(\underline{x})$.

Equality axioms, success axioms, default axioms, and failure axioms are the only formulas which belong to $Comp^*(P_N)$.

4.2 Mathematical Systems in CL, SLD and SLDNF

We will consider logic program synthesis in the framework of mathematical systems based on the following (first-order) logical systems: classical logic *CL*, *SLD*-logic and *SLDNF*-logic.

A mathematical system based on *CL* is denoted by $\langle S, CL\rangle$, where S is a set of first-order mathematical axioms [12] and *CL* a deductive logical system for classical logic. S will be assumed to be *finite* or *recursive*. In examples, proofs in systems $\langle S, CL\rangle$ will be developed in an informal way, but it will be clear that they can be formalised.

Mathematical systems based on *SLD*-logic and *SLDNF*-logic are denoted by $\langle Ax(P_D), SLD\rangle$ and $\langle Comp^*(P_N), SLDNF\rangle$ respectively. For our purposes, we formalise the proofs in *SLD* and *SLDNF* logical systems as follows:[13]

The set $\mathbf{P}_{SLD}$ of *SLD*-proofs and the set $\mathbf{P}_{SLDNF}$ of *SLDNF*-proofs are respectively the sets of "proofs" of the forms

$$Ax(P_D) \Rightarrow_\Gamma G_D\theta \quad \text{and} \quad Comp^*(P_N) \Rightarrow_{\Gamma'} G_N\theta'$$

where P_D is a definite program, G_D a definite goal, and Γ is a *SLD*-refutation of $\leftarrow G_D$ with program P_D and computed answer θ; P_N is a normal

[11] See the equality theory on p. 79 in [9].
[12] That is, first-order closed formulas of some first-order language.
[13] We do not need to consider any precise representation of *SLD*- or *SLDNF*-refutations.

program, G_N a normal goal, and Γ' is a *SLDNF*-refutation of $\leftarrow G_N$ with program P_N and computed answer θ'. For *SLD*-proofs and *SLDNF*-proofs, the *assumptions* are $Ax(P_D)$ and $Comp^*(P_N)$, and the *consequence* is $\bigwedge(G_D\theta)$ and $\bigwedge(G_N\theta')$, respectively.[14]

5 Deductive Synthesis of Logic Programs

Now we introduce our model of deductive synthesis of logic programs from logic specifications.

5.1 The Initial Specification

First of all, we have to build the logic specification. We assume that we want to compute a relation expressed by a formula R of a specification framework $\langle S, CL\rangle$. The first step is to introduce a new relation symbol r through the definition axiom $\forall(r \leftrightarrow R)$. Using r, we give the set G of the goals to be answered and the set $Ans(G)$ of the related possible answers. In this way, we build a restricted mathematical system:

$$\langle S \cup D_0, Ans(G), CL\rangle \tag{4}$$

where $D_0 = \{\forall(r \leftrightarrow R)\}$. We call (4) the *initial specification*. Every $g \in G$ will be called a *goal* and every $g^* \in Ans(g)$ will be called a *possible answer* of g.

If the specification framework completely defines the relation r,[15] the valid answers can be defined as follows. A possible answer is a valid *answer* iff it is provable in $\langle S \cup D_0, CL\rangle$.

For example, consider the problem to decide whether or not a natural number is even and, if it is, to compute its integer half. An adequate specification framework is first-order Peano Arithmetic *PA*, containing the usual axioms for $=, s, +, \cdot$, and the first-order induction schema. The relation "Z is half of a natural number X" is defined in *PA* by the formula $Z + Z = X$; so we introduce the definition axiom D_0:

$$\forall Z, X.\, (\mathit{half}(Z, X) \leftrightarrow Z + Z = X)$$

and build the specification[16]

$$\langle PA \cup \{D_0\}, Ans_g(\{\mathit{half}(Z, n) \,|\, n \text{ is a numeral}\}), CL\rangle$$

where $Ans_g(\mathit{half}(Z, n)) = \{\mathit{half}(m, n) \,|\, m \text{ is a numeral}\}$[17]. Note that Ans_g specifies the desired behaviour; indeed, if no (valid) answer exists, then n is odd; otherwise there is a unique valid answer $h(m, n)$, containing the integer half m of n.

[14] We use $\bigwedge\{F_1, \ldots, F_n\}$ to denote $F_1 \wedge \cdots \wedge F_n$.
[15] That is, a tuple $t_1, \ldots, t_n$ is in r iff $r(t_1, \ldots, t_n)$ is provable. One can consider also parametric or loose specifications; this case will be briefly discussed in Section 6.
[16] A numeral is a ground term of the form $s(s(\ldots s(0)\ldots))$.
[17] That is, g in Ans_g stands for *ground*

5.2 The Synthesis Process

Starting from an initial specification $\langle S \cup D_0, Ans(G), CL\rangle$, the aim of a deductive synthesis process is to find a logic program P such that[18]

$$\langle Ax(P), Ans(G), SLD\rangle \asymp \langle S \cup D_0, Ans(G), CL\rangle \ .$$

Now in order to answer the goals of G, we may have to use some relation expressed (in the language of S) by some non-atomic formula H. Since clauses contain only atomic formulas, we represent H by a new relation symbol h, introduced by a *definition axiom* $\forall(h \leftrightarrow H)$. Thus the synthesis process can be formalised as a sequence of the form

$$\begin{aligned}\langle S \cup D_0, Ans(G), CL\rangle \Rightarrow \langle S \cup D_1 \cup Ax(P_1), Ans(G), CL\rangle \Rightarrow \cdots \\ \Rightarrow \langle S \cup D_n \cup Ax(P_n), Ans(G), CL\rangle\end{aligned}$$

where $D_0 \subseteq D_1 \subseteq \cdots \subseteq D_n$ are definition axioms, $P_1 \subseteq \cdots \subseteq P_n$ are logic programs, and

$$S \cup D_k \cup Ax(P_k) \vdash_{CL} Ax(P_{k+1}), \quad \text{for } 1 \leq k < n \ . \tag{5}$$

By insisting on (5) (and using (2) and (3)) we can prove the *partial correctness* of every P_h (see [7, 8] for details), but not total correctness. A synthesis process *correctly halts* when the final program P_n is *totally correct*, i.e. when the following holds:[19]

$$\langle Ax(P_n), Ans(G), SLD\rangle \asymp \langle S \cup D_0, Ans(G), CL\rangle \ .$$

Note that for total correctness, we do not require that $Ax(P_n)$ and S_0 are equivalent, but only that they are equivalent as systems *restricted* to $Ans(G)$.[20]

In order to have a practical criterion for total correctness which will enable us to know when to stop the synthesis process, we shall make use of *failure axioms*: we will stop as soon as we can prove that

$$\langle S \cup D_n \cup Ax(P_n), CL\rangle \vdash Fax_n$$

where Fax_n contains the failure axioms of the relations defined in D_n, according to the synthesised program P_n.

If the above condition holds, *termination* implies *total correctness*.[21] Of course, we need also some method to state halting, but this does not concern us here and we will only deal with examples where halting is obvious.

The above criterion works in the cases where the specification framework completely defines the relations involved in the synthesis process. But it can be extended to parametric specifications: here one has to prove the failure

[18] For the sake of simplicity, we shall assume that P is a definite program.

[19] Note that $\langle Ax(P_n), Ans(G), SLD\rangle \asymp \langle Ax(P_n), Ans(G), CL\rangle$.

[20] In general, $\langle Ax(P_n), CL\rangle$ is a proper subsystem of $\langle S_0, CL\rangle$, even if P_n is totally correct. We will use this fact in our distinction between synthesis and transformation.

[21] By termination we mean that for every goal of G, we obtain a positive or a negative answer. Since failure axioms guarantee that negative answers are correct answers, we have a program that is totally correct with respect to G.

axioms only for the non-parametric relations. A deeper study of failure axioms in program synthesis will be given in a subsequent paper.

Finally, we can extend our model to *SLDNF* to obtain a final program P such that

$$\langle \mathit{Comp}^*(P), \mathit{Ans}(g), \mathit{SLDNF}\rangle \asymp \langle S_0, \mathit{Ans}(g), \mathit{CL}\rangle\ .$$

Because the synthesis process is an incremental one, here we have to avoid non-monotonicity. When we add a clause C to our current program, we introduce (and prove) only its *success* axiom $Ax(C)$ in our axiomatisation. We only introduce a *default* axiom $\forall x.\, \neg p(x)$ if we can prove it.

Furthermore, we only introduce the *failure* axiom of a predicate p when we have introduced *all* the clauses having p in their heads. In addition, to ensure partial correctness, we need to prove all the failure axioms that we introduce. This implies that if in intermediate synthesis steps there are failure axioms which cannot be proved, then partial correctness does not hold. However, as soon as all the failure axioms become provable, then partial correctness is 'restored'.

We conclude this section with an example to illustrate our model of deductive synthesis.

Example 5.1 We start from the initial specification considered in Subsection 5.1:

$$\langle \mathit{PA} \cup \{D_0\}, \mathit{Ans}(\{\mathit{half}(Z,n) \mid n \text{ is a numeral}\}), \mathit{CL}\rangle\ .$$

We can get a clause for $\mathit{half}(0,X)$ from D_0:[22]

$$\mathit{half}(0,X) \leftrightarrow 0+0=X \leftrightarrow X=0\ .$$

This first step of the synthesis process gives:

$$\langle \mathit{PA} \cup \{D_0, Ax(\mathit{half}(0,X) \leftarrow X=0)\}, \{\mathit{half}(Z,n) \mid n \text{ is a numeral}\}, \mathit{CL}\rangle\ .$$

The corresponding failure axiom is

$$\forall Z,X.\, (\neg Z=0 \wedge \neg X=0 \rightarrow \neg \mathit{half}(Z,X))\ .$$

We can prove its negation in $PA \cup D_0$, and therefore our current program $\mathit{half}(0,X) \leftarrow X=0$ is only partially, but not totally, correct. So we have to search for some other clause.

Suppose we try to find a clause for $\mathit{half}(s(I),X)$ as follows:

$$\begin{array}{lcl} \mathit{half}(s(I),X) & \leftrightarrow & s(I)+s(I)=X \\ & \leftrightarrow & s(s(I+I))=X \\ & \leftrightarrow & \exists W.\, (s(s(W))=X \wedge \mathit{half}(I,W)) \end{array}$$

and so we get

$$\begin{array}{l} \langle \mathit{PA} \cup \{D_0, \\ Ax(\mathit{half}(0,X) \leftarrow X=0), Ax(\mathit{half}(s(I),X) \leftarrow s(s(W))=X, \mathit{half}(I,W))\}, \\ \{\mathit{half}(Z,n) \mid n \ \text{ is a numeral}\}, \mathit{CL}\rangle\ . \end{array}$$

[22] This is provable in $\mathit{PA} \cup D_0$.

Now the failure axiom is

$$\forall Z, X. ((\neg(X = 0 \land Z = 0) \land \neg\exists W, I. (s(s(W)) = X \land Z = s(I) \land \\ half(I, W)) \rightarrow \neg half(Z, X))$$

and it can be proved. So our program is totally correct, since it halts for all the goals considered. $\diamond$

6 Specification

In this section, we discuss the activity of constructing a specification for deductive synthesis of logic programs. In general, specification is of course only an informal process of translating an informal, and hence loose and often incomplete statement of some problem into a precise specification, which could be formal or informal. Clearly there is no formal method for proving the correctness of specifications.

In our framework, the aim of the specification phase is to build up a *logical* specification, formalised as a restricted mathematical system of the form $\langle S, \mathcal{G}, CL\rangle$.[23] The *specification framework* S may be completely given, though more usually it has to be built and maintained (i.e. added to whenever necessary). We shall discuss two ways of constructing such a specification.

Firstly, if we start with an informal (but precise) idea of the problem and know nothing about S, we can use a process similar to deductive (program) synthesis, and build up the specification, i.e. S and $\mathcal{G}$, in a stepwise manner. The idea is that if we can construct a clause set, i.e. a logic program P, which represents the problem we have in mind, then we can use $Ax(P)$ as S. Thus, we get a specification together with a program that satisfies it. However, in our framework, this process is not program synthesis, since every step gives rise to new axioms which add to the current incomplete specification of the problem in question. We will call it *specification synthesis* to distinguish it from program synthesis. Our specification synthesis corresponds to the construction of logic descriptions of [1]. We now give a simple example of specification synthesis.

Example 6.1 Consider the problem of finding all the prefixes of a list L. The first step of formalising the problem is to state a logic language for lists and its intended interpretation.

Suppose we choose the following language. It is many-sorted, with sorts *Els* (elements) and *Lists* (finite lists of elements), and its symbols are:

- the constant *nil* (the empty list) and the function symbol '.' such that $X.L$ is the list with head X and tail L;
- the (polymorphic) identity relation symbol $=$ on *Lists* and *Els*.

[23] Although we only consider CL, in general we may have different kinds of logics, such as modal, temporal or higher order logics.

Now, to specify our problem, we introduce the relation $\mathit{prefix}(L_1, L_2)$, meaning "$L_1$ is a prefix of L_2, i.e. L_2 is of the form $L_1 \ldots$", and try and find a set of clauses characterising *prefix* according to its intended meaning.

Suppose we try to define *prefix* by using the clause:

$$\mathit{prefix}(X.L_1, X.L_2) \leftarrow \mathit{prefix}(L_1, L_2)\ .$$

Now $Ax(\mathit{prefix}(X.L_1, X.L_2) \leftarrow \mathit{prefix}(L_1, L_2))$ agrees with our intended interpretation. However, the failure axiom

$$\forall(\neg\exists X, L_1, L_2 : (A = X.L_1 \wedge B = X.L_2 \wedge \mathit{prefix}(L_1, L_2)) \rightarrow \neg\mathit{prefix}(A, B))$$

imposes, for instance, that $\mathit{prefix}(A, B)$ is *false* if $A = nil$, contrary to the intended meaning of *prefix*. So we add the clause

$$\mathit{prefix}(nil, L) \leftarrow$$

where $Ax(\mathit{prefix}(nil, L))$ is valid in our intended interpretation.

Again, we analyse the failure axiom. It is now

$$\begin{aligned}&\forall(\ \neg A = nil \wedge \neg\exists X, L_1, L_2 : (A = X.L_1 \wedge B = X.L_2 \wedge \\ &\quad \mathit{prefix}(L_1, L_2)) \rightarrow \neg\mathit{prefix}(A, B))\end{aligned}$$

and agrees with our intended interpretation.

At this point, we have synthesised both a program and an axiomatisation. Of course, the program is correct with respect to the axiomatisation, by construction.[24] $\diamond$

Secondly, we can construct a specification, i.e. S and $\mathcal{G}$, in the framework of a given well-understood and expressive formal system, which we shall call a *specification framework*. In this situation, we can define *prefix* by a definition axiom, say D, directly in terms of the axioms of the specification framework SF, to yield the logical specification $\langle SF \cup D, Ans(\mathit{prefix}), CL\rangle$. Of course, such a situation is the norm in our framework.[25] We now illustrate with an example.

Example 6.2 Suppose we have to specify the *prefix* problem, but this time we are given the following specification framework for lists, which we denote by S_{list}. It contains the usual identity axioms and the axioms characterising the basic operations on lists, as follows:

- $\forall X, L.\,(\neg X.L = nil)$
- $\forall X_1, X_2, L_1, L_2.\,(X_1.L_1 = X_2.L_2 \rightarrow X_1 = X_2 \wedge Y_1 = Y_2)$
- induction schema on lists, for every formula H:

$$H(nil) \wedge \forall X, L.\,(H(L) \rightarrow H(X.L)) \rightarrow \forall L.\,H(L)\ .$$

[24] However, it is worth repeating that there is no formal way of proving the correctness of the axiomatisation with respect to our intended meaning.

[25] It is what we assumed in our model of deductive synthesis of programs.

Suppose S_{list} also contains, among other things,[26] the relation $occ(L,i,X)$, which means that X occurs at place i in L, characterised by the following axioms:

- $\forall L, X, Y, i.\,(occ(L,i,X) \wedge occ(L,i,Y) \rightarrow X = Y)$
- $\forall X, i.\,\neg occ(nil, i, X)$
- $\forall L, X, i.\,occ(X.L, 0, X)$
- $\forall L, X, Y, i.\,(occ(X.L, s(i), Y) \leftrightarrow occ(L, i, Y)$

Then in S_{list}, we can define *prefix* by the following *definition axiom*:

$$D_{prefix}: \quad prefix(L,M) \leftrightarrow \forall i, X.\,(occ(L,i,X) \rightarrow occ(M,i,X)) \tag{6}$$

and the initial specification for the *prefix* problem is

$$\langle S_{list} \cup D_{prefix}, Ans(\{prefix(M,n), n \text{ is ground}\}), CL\rangle$$

. $\diamond$

The specification constructed in Example 6.2 will be the starting point for (deductive) program synthesis. In contrast, the specification constructed in Example 6.1 above directly represents a program. This would seem to suggest that it would always be advantageous to use our deductive synthesis approach to do specification synthesis. However, this is not the case. In fact, there are many advantages doing program synthesis. For example, consistency problems can be avoided using program synthesis methods which preserve consistency. Furthermore, starting from the same specification, we can synthesise many different algorithms. Moreover, using correct synthesis methods, all the synthesised programs are correct with respect to the specification. Consequently, once we have evidence that our starting specification agrees with our intended meaning,[27] the correctness of programs is no longer a problem. Finally, during the synthesis, we can use all the knowledge developed in the specification framework.

Of course, the two methods for constructing specification can be mixed. For instance, it may happen that, while synthesising programs to compute the relations in question, we may discover that we need to add new axioms. In this way, we can enrich the given specification framework, thus obtaining a new one. We will discuss this further in Section 9.

To conclude this section, we now briefly discuss *parametric specification frameworks*.

Frameworks like *PA* are not parametric. For a non-parametric framework S,[28] we assume that there is an intended term model, i.e. a model whose elements can be represented by suitable ground "normal" terms (e.g. in *PA*

[26] Such as the sort *Nat* of natural numbers and the language *LN* and the axioms of *PA*.

[27] The problem of the correctness of the formal specification with respect to the informal problem remains, but it can now be studied once for all.

[28] For the sake of simplicity, we omit the logical system *CL* here.

natural numbers are represented by numerals). We say that S completely defines a relation R by a formula $F_R(\underline{X})$ iff the following holds: for every normal $\underline{t}$, if $\underline{t}$ is in R, then $F_R(\underline{t})$ is provable in S, otherwise $\neg F_R(\underline{t})$ is provable. This definition is very close to the notion of correctness in [1] and allows us to use negation in an appropriate way.

In general, parametric specifications cannot be used to completely define relations. For example, if in S_{list} the sort *Els* is a parameter, we have no ground terms for this sort. Then no relation containing some non-empty list in its domain can be completely defined, since we do not have ground terms denoting non-empty lists. Another example is the use of parametric relation symbols. For instance, we may assume some partial ordering relation without completely defining it (i.e. we give only the axioms of partial ordering).

An instance of a parametric framework S is a set of axioms $S_I \supset S$ completely characterising the parameters. Now, consider a class $\mathcal{R}$ of relations, containing a relation R_I for each possible instance S_I of S; we say that a formula $F_{\mathcal{R}}(\underline{X})$ *parametrically defines* $\mathcal{R}$ in S iff, for every instance S_I, $F_{\mathcal{R}}(\underline{X})$ completely defines (in S_I) the relation R_I. For example, in lists with parametric sort *Els*, for every instance I of *Els* we have a corresponding relation $\mathit{prefix}_I(X, Y)$, i.e. we have a class of relations; one can see that the definition axiom D_{prefix} parametrically defines this class. Synthesis from parametric definitions works as in the complete ones, using failure axioms as a sufficient criterion to stop synthesis. A deeper analysis of this will be one of our future research topics.

7 (Deductive) Program Synthesis

In this section, we will use the model of deductive synthesis described in Section 5 to illustrate the difference between specification synthesis and program synthesis for the *prefix* problem. We shall assume that the initial specification is given in a general specification framework (as in Example 6.2), and use our deductive synthesis model to synthesise a program for *prefix* from this specification.

Example 7.1 The initial specification is

$$\langle S_{list} \cup D_{prefix}, \{\mathit{prefix}(m,n) \mid n \text{ is ground}\}, \mathit{CL}\rangle$$

as defined in Example 6.2.

In order to compare program synthesis with specification synthesis, we choose synthesis steps that are identical to the ones in Example 6.1. The first step is to prove (in $S_{list} \cup D_{prefix}$):[29]

$$\begin{array}{rcl} \mathit{prefix}(X.L_1, X.L_2) & \leftrightarrow & \forall i, Y.\,(\mathit{occ}(X.L_1, i, Y) \rightarrow \mathit{occ}(X.L_2, i, Y)) \\ & \leftrightarrow & \forall i, Y.\,(\mathit{occ}(L_1, i, Y) \rightarrow \mathit{occ}(L_2, i, Y)) \\ & \leftrightarrow & \mathit{prefix}(L_1, L_2) \end{array}$$

[29]In particular, the second bi- implication is a useful theorem of S_{list} which could be added to our specification framework, thus enriching our knowledge base.

and so the result of this first step is the same as that of the first step in Example 6.1. However, here we have a formal proof of partial correctness in S_{list}.

Moreover, we can prove the negation of the failure axiom of the clause synthesised. Therefore total correctness does not hold, and we go on with a further synthesis step, which is:

$$\begin{array}{lcl} \mathit{prefix}(nil, M) & \leftrightarrow & \forall i, X.\, (occ(nil, i, X) \rightarrow occ(M, i, X)) \\ & \leftrightarrow & true \end{array}$$

At this point we can prove in S_{list} the failure axiom. Since our program now halts for the goals considered, we have a totally correct program P_{prefix} and we stop the synthesis. ◇

The difference with respect to specification synthesis is that here the correctness of every synthesis step has been proved in the specification framework S_{list}.

Furthermore, we can easily see that the mathematical systems

$\langle Ax(P_{prefix}), CL\rangle$

(corresponding to the extracted program P_{prefix}) is a *very small sub-theory* of the initial specification.[30] This sums up our view on (deductive) program synthesis: it extracts a small sub-theory (formalised as a logic program) which specialises the initial specification to just the goals specified.

In contrast, as we shall see in the next section, *program transformation* means transformation of a small (mathematical) system, representing a logic program only (but not a specification framework), into an equivalent system, representing another program.

8 Program Transformation

Program transformation starts from a program and gives rise to another program. In our framework, both logic programs and specifications are formalised as mathematical systems. However, mathematical systems for programs employ the logical system *SLD* whilst those for specifications employ *CL*.[31] If we consider *CL* as a logical computing mechanism, just as *SLD*, then specifications are "programs" using *CL* as the (inefficient) computing mechanism, and any distinction between logic specifications and logic programs, and hence between program synthesis and program transformation, can only be based on pragmatic considerations rather than logical ones.

However, we think that a distinction can be proposed that is based not on the distinction between *CL* and *SLD*, nor on the way of doing synthesis or transformation. Such a distinction can be based on a classification of mathematical systems into those which are *specifications* and those which represent logic programs. Clearly, the latter is a subclass of the former.

[30] Even considering the completion $\langle Comp^*(P_{prefix}), CL\rangle$, we obtain a proper subsystem.
[31] Our considerations hold for any specification logic and any logic programming system.

We say that a mathematical system $\langle S, CL\rangle$ *represents a program* iff $S = Ax(P)$, for some program P, or it is *equivalent* to some logic program P. Different equivalence notions may be considered. Two of them are, for example, the following ones.

$\langle S, CL\rangle$ is equivalent to P if there are definition axioms D such that[32]

$$\langle S \cup D, CL\rangle \asymp \langle Ax(P) \cup D, CL\rangle \tag{7}$$

and every definite goal provable in $Ax(P) \cup D$ is provable also in $Ax(P)$, i.e. no solution is lost by omitting D.

In the second definition, instead of (7), we require

$$\langle S \cup D, CL\rangle \asymp \langle Comp^*(P), CL\rangle \ .$$

In this case every finite axiomatisation represents a program. Indeed, S can be transformed into an equivalent $Comp^*(P)$ according to [9].

Thus deriving a program P from S does not extract a small part of S (as in the case of deductive program synthesis), but an *equivalent* axiomatisation. Rather it can be seen as a *transformation* of a "program" S, specified using the inefficient computing mechanism CL, into a program P such that, by the completeness of SLD, the (efficient) computing mechanism SLD can be used instead of CL.

In our framework, we regard (*deductive*) *program synthesis* as deriving a program from a mathematical system representing a *specification*; whereas deriving a program from a mathematical system representing a program is considered as *program transformation*. It is worth repeating that, in our framework, program synthesis derives a (very) small sub-theory of the initial specification, while program transformation derives an axiomatisation equivalent to that of the initial program.

For program transformation it is possible to apply techniques which cannot be applied, in general, to synthesis. Total correctness is not an issue here, since any adequate notion of equivalence should imply it. The main problem for transformation is then to preserve the equivalence of the mathematical systems involved, using CL or, at least, SLD.

Here we consider a transformation method working in CL, and based on the general model of Section 5 but with the following difference. Instead of proving total correctness, we prove (7) by proving[33]

$$\langle Ax(P_n) \cup D_n, CL\rangle \vdash S \ . \tag{8}$$

and we then show that no solution is lost by omitting D_n.

Let us explain with two simple examples.

[32] We compare the two axiomatisations not as systems restricted to any kind of goals, but as unrestricted mathematical systems based on CL. A translation ρ_L may be involved, corresponding to the definition axioms of the relations computed by P.

[33] Of course, this cannot be proved in program synthesis, since in general $Ax(P_n) \cup D_n$ (and even $Comp^*(P_n) \cup D_n$) is properly contained in $S \cup D_n$.

Example 8.1 This example is a transformation involving *SLD* programs. Therefore the logic programming system remains the same, and transformation modifies the programs. We start from the following program P_{fib} to compute Fibonacci numbers (we omit the usual clauses for $+$):

$$\begin{array}{rcl} f(0,s(0)) & \leftarrow & \\ f(s(0),s(0)) & \leftarrow & \\ f(s(s(I)),Z) & \leftarrow & f(I,Z_1), f(s(I),Z_2), +(Z_1,Z_2,Z) \end{array}$$

Of course, the corresponding mathematical system

$$\langle Ax(P_{fib}), CL\rangle$$

represents a program. We can transform it into a more efficient program in the usual way. First of all, we introduce a new relation ff, by the following definition axiom D_{ff}:

$$D_{ff} \;:\; ff(X,A,B) \leftrightarrow f(X,A) \wedge f(s(X),B)\;.$$

The transformation is carried out as deductive synthesis, but here, instead of total correctness, we test for the condition (8).

In the first transformation steps, we can prove, in $Ax(P_{fib}) \cup D_{ff}$, the axioms

$$Ax(ff(0,s(0),s(0)) \leftarrow)\;,\; Ax(ff(s(I),A,B) \leftarrow ff(I,H,A), +(I,H,B))\;. \quad (9)$$

However, no goal in f can be solved by $Ax(P_{fib})$, since there are no clauses for f. So we add the axiom:

$$Ax(f(X,A) \leftarrow ff(X,A,B))\;. \quad (10)$$

Let P_{ff} be the program containing the clauses in (9) and (10). To prove that no solution is lost by omitting D_{ff}, we can prove for instance that it is true in the minimum Herbrandt model. Hence we have our final equivalent program. $\diamond$

In the above example, transformation involves definite programs, and the computing mechanism is not changed. In the next example, we transform an equational program into a logic one. Transformation is carried out in *CL*, which contains both the starting equational system and the target *SLD* system.

Example 8.2 The initial "program" S is the following equational theory for computing $+$ for numerals (the usual axioms for identity will be assumed):

$$\{\forall x.\, x+0=x, \forall x,y.\, x+s(y)=s(x+y)\}$$

and the corresponding mathematical system is $\langle S, REW\rangle$, where REW is a rewriting system for example.

The first step to transform this equational system into a relational one is to introduce the definition axiom

$$D_0 \;:\; \forall x,y,z.\,(sum(x,y,z) \leftrightarrow x+y=z)\;.$$

Then to perform the transformation on S, we start a deductive synthesis process from the initial specification:

$$\begin{array}{l} \langle \{\forall x.\, x+0=x, \forall x,y.\, x+s(y)=s(x+y)\} \cup \\ \{\forall x,y,z.\, (sum(x,y,z) \leftrightarrow x+y=z)\}, \\ \{sum(m,n,Z) \mid m,n \text{ are numerals}\}, CL\rangle \end{array}$$

where the equational goals have been mapped into relational ones.

The first step is to derive:

$$sum(X,0,Z) \quad \leftrightarrow \quad X+0=Z \quad \leftrightarrow \quad X=Z$$

and we obtain a program P_0 containing the clause $sum(X,0,X) \leftarrow$. At this point, however, we can prove only one of the axioms of S, i.e.

$$\langle \{Ax(sum(X,0,X) \leftarrow), D_0\}\rangle \vdash \forall x.\, x+0=x\ .$$

So we go on as follows:

$$\begin{array}{rcl} sum(X,s(I),Z) & \leftrightarrow & X+s(I)=Z \quad \leftrightarrow \quad s(X+I)=Z \\ & \leftarrow & \exists W.\, (Z=s(W) \wedge X+I=W)^{34} \\ & \leftrightarrow & \exists W.\, (Z=s(W) \wedge sum(X,I,W)) \end{array}$$

and get the clause

$$sum(X,s(I),s(W)) \leftarrow sum(X,I,W)\ .$$

Now it can be proved easily that:

$$\begin{array}{rl} & \langle \{Ax(sum(X,s(I),s(W)) \leftarrow sum(X,I,W)), D_0\}, CL\rangle \\ \vdash & \forall x,i.\, x+s(i)=s(x+i)\ . \end{array}$$

To prove that no solution is lost by omitting D_0, we can show for instance that in the minimum Herbrandt model $sum(X,Y,Z)$ defines a total function $Z=f(X,Y)$. Therefore we have obtained our final logic program that is equivalent to the original equational one. $\diamond$

9 Mixing Specification with Synthesis and Transformation

Although in the previous sections we have made clear and formal distinctions between specification, synthesis and transformation, we have at the same time also hinted at their close inter-relationships by applying our deductive synthesis model to both specification and transformation. Indeed, to emphasise this, we will now give examples of how in methodological terms it may be desirable and possible to mix specification with synthesis and transformation respectively.

[34] The implication follows from identity axioms. A bi-implication does not hold, since S_0 does not contain axioms for successor s.

9.1 Specification and Synthesis

First we discuss specification and its relation with synthesis. In our deductive synthesis model, we assume that a specification is constructed using a (given) specification framework. The idea is that a specification framework is based on a well consolidated set of axioms. We assume that new specifications are always introduced by *definition axioms*, and synthesis gives rise to conservative extensions. Consequently, we have no consistency problems. Moreover, synthesis gives rise to extensions where the properties of the old relations are not affected by the new axioms. The latter point is important, since adding new programs does not affect the correctness of the old ones.

Of course, ideally we would want to be given specification frameworks that are expressive and powerful. In practice, however, often we have to build specification frameworks ourselves. In such a situation, we can imagine that we start from an inadequate axiomatisation and go through an initial experimental phase, whereby this axiomatisation is changed or enriched. In our framework, this phase may be performed as specification synthesis (as we saw in Section 6), during which experimental attempts are made at specifying new relations or synthesising new programs. Let us see some examples.

Example 9.1 The specification framework S_{list} of Section 6 may be seen as an initial stage in the process of building up a specification framework for lists. Let us try to define the permutation relation $perm(L_1, L_2)$. An obvious way is to say that every element X has the same number of occurrences in both L_1 and L_2. However, we cannot express the relation "number of occurrences" by a definition axiom. So this is a primitive relation (which we shall call $nocc(X, L, N)$) characterised by *new* axioms, which can be obtained for example by the following specification synthesis steps:

$$\begin{array}{rll} nocc(X, nil, 0) \leftarrow & & \text{(Failure axiom wrong.)} \\ nocc(X, X.L, s(I)) \leftarrow & nocc(X, L, I) & \text{(Failure axiom wrong.)} \\ nocc(X, Y.L, I) \leftarrow & \neg(X = Y), nocc(X, L, I) & \text{(Failure axiom correct.)} \end{array}$$

and the corresponding success and failure axioms characterise *nocc*.

At this point, we can define the relation $perm(L_1, L_2)$ as follows:

$$\begin{array}{l} \forall(perm(L_1, L_2) \leftrightarrow \\ \quad \forall X, N, M. \, (nocc(X, L_1, N) \wedge nocc(X, L_2, M) \rightarrow N = M)) \; . \end{array}$$

Now we have a richer specification context and we can go on with our specification. ◇

Example 9.2 Suppose we want to specify sorting problems. To do so, we require an ordering relation $\leq$ on the sort Els. This relation introduces a *constraint* on the parameter Els. To avoid constraint propagation in previously developed programs, it is better to introduce a new specification framework S_{olist}. It is equal to S_{list}, but the language now contains the new relation symbol $\leq$. Even if we do not give axioms on $\leq$, we can specify ordered lists for instance as follows:

$$\forall(ord(L) \leftrightarrow (occ(L,I,X) \wedge occ(L,s(I),Y) \rightarrow X \leq Y)) \; .$$

It is easy to see that we can synthesise a program for deciding ord. Now, we can specify $sort$ as follows:

$$\forall(sort(L,M) \leftrightarrow (ord(M) \wedge perm(L,M))) \; .$$

However, if we try to synthesise a sorting algorithm, we discover that $\leq$ must be a total ordering, and we have to add the corresponding axioms to our specification framework. Again such axioms introduce new constraints on the parameter Els, which are not needed, for example for the program deciding the predicate ord. Therefore it is better to introduce a new specification framework S_{tolist} containing the total ordering axioms on $\leq$, and carry on in this new framework. $\diamond$

In the above example, S_{olist} is a subclass of S_{list} which inherits all the programs already developed in S_{list}. S_{tolist} is, in turn, a subclass of S_{olist} and inherits from S_{olist}. In contrast, in Example 9.1, we added the new relation $nocc$ without introducing a new subclass, and just by updating S_{list}. This amounts to a pragmatic choice to ensure that the starting framework should be strong enough to define all the properties we consider basic for any kind of lists: length, concatenation, permutation, reverse, and so on. On the other hand, ordering is related to properties of the elements, and hence characterises particular kinds of lists, corresponding to subclasses.

Examples 9.1 and 9.2 show how abstract data types, classes and inheritance (as introduced in [3] for example) can be managed in logic program synthesis. Moreover, synthesis methods can be used both in program and in specification synthesis, so that we have a framework where synthesis and specification can be mixed in very natural way, without confusing one with the other. We believe that the study of methodologies using specification frameworks is an important issue in logic program synthesis.

9.2 Specification and Transformation

As well as mixing specification with synthesis, we can mix specification with transformation. We now give a simple example. In this example, we transform a program into another by first augmenting the specification framework with an axiom that enables the transformation to be performed. Note that the starting program and the target one are not equivalent systems; the transformation preserves correctness but not equivalence.

Example 9.3 Suppose we have the following axioms, defining a map set from lists into sets:

$$\begin{aligned} set(nil) &= \emptyset \\ set(X.L) &= \{X\} \cup set(L) \; . \end{aligned}$$

We can add the above axioms and some suitable axiomatisation of finite sets to S_{list} to get $S_{listset}$. Suppose we specify the relation "list union" lu as follows:

$$\forall(lu(A,B,C) \leftrightarrow set(A) \cup set(B) = set(C)) \ .$$

From this, we may infer or synthesise for instance the following program P_1 which is obviously correct:

$$\begin{array}{rcl} lu(A,nil,A) & \leftarrow & \\ lu(nil,A,A) & \leftarrow & \\ lu(X.A,Y.B,X.Y.C) & \leftarrow & lu(A,B,C) \end{array}$$

However, P_1 computes only part of the relation lu. For example, we know that $lu(a.nil,a.nil,a.nil)$ holds, but P_1 computes only $lu(a.nil,a.nil,a.a.nil)$. Even though we cannot prove the failure axioms of lu, nevertheless we may be satisfied if our aim is to find only one of the possible outputs.

On the other hand, we may wish to improve P_1 by transforming it. One way to do so would be to add the axiom (which we can prove)

$$\forall(lu(X.A,X.B,X.C) \leftrightarrow lu(X.A,X.B,X.X.C))$$

to our specification framework, and then use this axiom to transform P_1 into the following program P_2:

$$\begin{array}{rcl} lu(A,nil,A) & \leftarrow & \\ lu(nil,A,A) & \leftarrow & \\ lu(X.A,X.B,X.C) & \leftarrow & lu(A,B,C) \\ lu(X.A,Y.B,X.Y.C) & \leftarrow & \neg X = Y, lu(A,B,C) \end{array}$$

The two programs P_1 and P_2 represent *equivalent* systems in the following sense:

$$\forall(lu(X.A,X.B,X.C) \leftrightarrow lu(X.A,X.B,X.X.C)) \vdash Ax(P_1) \leftrightarrow Ax(P_2) \ .$$

In other words equivalence holds not in pure logic, but in our augmented specification framework, where we can prove

$$\forall(lu(X.A,X.B,X.C) \leftrightarrow lu(X.A,X.B,X.X.C)) \ .$$

$\diamond$

10 Conclusion

We have presented a formal mathematical framework for deductive synthesis of logic programs. In this framework, logic programs are viewed as axiomatisations, and specification, (deductive) synthesis and transformation are regarded as operations on mathematical systems made up of axiomatisations and logical systems such as classical logic, *SLD* and *SLDNF*. We have proposed a distinction based on the behaviour of the mathematical systems involved in these operations.

Program specification may be constructed in two different ways: the first is to build up an axiomatisation of the problem in hand (using our deductive synthesis model), the second is to define the problem by a formula in a given specification framework. The resulting specification is a restricted mathematical system $\langle S, \mathcal{G}, CL\rangle$. The correctness of a specification can only be stated informally.

Program synthesis starts from a specification $\langle S, \mathcal{G}, CL\rangle$, and extracts a program. This program represents a small sub-system of the initial specification, specialised to solve just the goals being considered. The correctness of the program can be formally established.

Program transformation works on small systems representing programs instead of specifications, and transforms them into equivalent systems. The equivalence can also be formally proved.[35]

This distinction naturally leads to an ideal scenario, where we start by specifying abstract data types and classes, and then synthesise and optimise (by transformation) logic programs. However, in practice, it is often desirable and possible to mix specification with synthesis and transformation.

With respect to classes, there is a natural notion of subclass (as a stronger specification framework), and subclasses inherit programs synthesised in superclasses. We believe that an important step is to develop a methodology not only for program derivation, but also for managing specification frameworks.

Finally, we have considered classical logic only. However, our model works equally well with different logics. Using suitable representation functions ρ and ρ_L, we can translate problems specified in a modal logic for example into computation problems which can be solved in *SLD*. Thus the notion of representation of mathematical systems provides a means to extend the above scenario to specifications based on logics different from classical logic.

Acknowledgements

We would like to thank the referees for their valuable comments which have improved this paper considerably. We are also grateful to LOPSTR 93 participants for making our workshop presentation a lively 'interactive' session.

References

[1] Y. Deville. *Logic Programming: Systematic Program Development*. Addison-Wesley, 1990.

[2] A. Eriksson, A.-L. Johansson and S.-Å.Tärnlund. Towards a Derivation Editor. In M. van Caneghem and D.H.D. Warren, editors, *Logic Programming and its Applications*, pages 117–126. Ablex Publ. Co., 1986.

[35] In [2], program synthesis and transformation are respectively classified as (i) a derivation from a definition D_d of data structures and a definition D_r about relations or functions; and (ii) a derivation from a program and the definitions D_d and D_r. Such a distinction is really very similar to ours in spirit.

1990.

[3] 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. 283,

[4] Å. Hansson and S.-Å. Tärnlund. A natural programming calculus. In *Proc. IJCAI-79*, pages 348–355, 1979.

[5] C.J. Hogger. Derivation of logic programs. *Journal of the ACM*, **28**, 372–392, 1981.

[6] K.K. Lau and S.D. Prestwich. Top-down synthesis of recursive logic procedures from first-order logic specifications. In D.H.D. Warren and P. Szeredi, editors, *Proc. 7^{th} Int. Conf. on Logic Programming*, pages 667–684. MIT Press, 1990.

[7] K.K. Lau and M. Ornaghi. Towards a formal framework for deductive synthesis of logic programs. Technical Report UMCS-92-11-2, Department of Computer Science, University of Manchester, November 1992.

[8] 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.

[9] J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2nd edition, 1987.

[10] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM TOPLAS*, 2(1):90–121, Jan 1980.

[11] J.C. Shepherdson. Negation in Logic Programming. in J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 19-88. Morgan Kaufmann, 1988.