On Correct Program Schemas

Pierre Flener,[1] Kung-Kiu Lau[2] and Mario Ornaghi[3]

[1] Department of Computer Science
Bilkent University, 06533 Bilkent, Ankara, Turkey
e-mail: pf@cs.bilkent.edu.tr
[2] Department of Computer Science
University of Manchester, Manchester M13 9PL, United Kingdom
e-mail: kung-kiu@cs.man.ac.uk
[3] Dipartimento di Scienze dell'Informazione
Universita' degli studi di Milano, Via Comelico 39/41, Milano, Italy
e-mail: ornaghi@dsi.unimi.it

Abstract. We present our work on the representation and correctness of program schemas, in the context of logic program synthesis. Whereas most researchers represent schemas purely syntactically as higher-order expressions, we shall express a schema as an open first-order theory that axiomatises a problem domain, called a *specification framework*, containing an open program that represents the template of the schema. We will show that using our approach we can define a meaningful notion of correctness for schemas, viz. that correct program schemas can be expressed as *parametric* specification frameworks containing templates that are *steadfast*, i.e. programs that are always correct provided their open relations are computed correctly.

1 Introduction

A program schema is an abstraction of a class of actual programs, in the sense that it represents their data-flow and control-flow, but does not contain (all) their actual computations or (all) their actual data structures. Program schemas have been shown to be useful in a variety of applications, such as proving properties of programs, teaching programming to novices, guiding both manual and (semi-)automatic synthesis of programs, debugging programs, transforming programs, and so on, both within and without logic programming. An overview of schemas and their applications can be found in [6].

In this paper, we present our work on two aspects of schemas: representation and correctness, in the context of logic program synthesis. In logic programming, most researchers represent their schemas as higher-order expressions, sometimes augmented by extra-logical annotations and features, so that actual (first-order) programs are obtained by applying higher-order substitutions to the schema. We shall take a different approach and show that a schema $\mathcal{S}$ can also be expressed as an *open* first-order theory $\mathcal{F}$ containing an *open* (first-order) program T, viz. a program in which some of the relations are left undefined. One advantage of this approach is that it simplifies the semantics of schemas and of their manipulations.

We shall endow a schema $\mathcal{S}$ with a formal (model-theoretic) semantics by defining $\mathcal{F}$ as a *specification framework*, i.e. an axiomatisation of the (possibly open) problem domain, and call T the *template* of $\mathcal{S}$. This allows us to define a meaningful notion of correctness for schemas. Indeed, we show that *correct* program schemas can be expressed as *parametric* specification frameworks containing templates that are *steadfast* open programs, i.e. programs that are always correct provided their open relations, i.e. their *parameters*, are computed correctly. Steadfastness is *a priori* correctness, and therefore correct schemas are *a priori* correctly reusable.

We shall also briefly indicate how to use correct schemas in practice. Using any kind of schemas requires suitable strategies, and we shall touch on some ideas for such strategies for correct schemas.

2 Program Schemas as Open Frameworks

Our approach to schemas (and program synthesis) is set in the context of a (fully) first-order axiomatisation $\mathcal{F}$ of the problem domain in question, which we call a *specification framework* $\mathcal{F}$. Specifications are given in $\mathcal{F}$, i.e. written in the language of $\mathcal{F}$. This approach enables us to define program correctness wrt specifications not only for closed programs but also for open programs, i.e. programs with parameters (open relations), in both closed and open frameworks. In this section, we briefly define specification frameworks, specifications, open programs.

2.1 Specification Frameworks

A specification framework is a full first-order logical theory (with identity) with an intended model:

Definition 1. (Specification Frameworks)
A *specification framework* $\mathcal{F}(\Pi)$ with parameters Π consists of*:*

- A (many-sorted) signature Σ of *sort*, *function* and *relation* symbols (together with their declarations).
  We distinguish between symbols of Σ that are *closed* (i.e. *defined* symbols) and those that are *open* (i.e. *parameters*). The latter are indicated by Π.
- A set of first-order axioms for the (declared) closed and open function and relation symbols of Σ.
  Axioms for the closed symbols may contain first-order *induction schemas.*
  Axioms for the open symbols, or parameters, are called *p-axioms.*

$\mathcal{F}(\Pi)$ is *open* if the set Π of parameters is not empty; it is *closed* otherwise.

A closed framework $\mathcal{F}$ axiomatises one problem domain, as an intended model (unique up to isomorphism). In our approach, intended models are *reachable isoinitial* models:

Definition 2. (Reachable Isoinitial Models)
A *reachable isoinitial model* i of $\mathcal{F}$ is a model such that i is reachable (i.e. the elements of its domain can be represented by ground terms) and, for any relation r defined in $\mathcal{F}$, ground instances $r(t)$ or $\neg r(t)$ are true in i iff they are true in all models of $\mathcal{F}$.

Example 1. (Closed Frameworks)
A typical closed framework is (first-order) Peano arithmetic $\mathcal{NAT}$ (we will omit the most external $\forall$ quantifiers):

Specification Framework $\mathcal{NAT}$;

SORTS:	Nat;
FUNCTIONS:	$0 \quad : \to Nat$;
	$s \quad : Nat \to Nat$;
	$+, * : (Nat, Nat) \to Nat$;
AXIOMS:	C-AXS$(0, s)$;
	$x + 0 = x$;
	$x + s(y) = s(x + y)$;
	$x * 0 = 0$;
	$x * s(y) = x + x * y$;

C-AXS$(0, s)$ contains Clark's Equality Theory (see [20]) for the constructors 0 and s, and the related (first-order) *induction schema* $H(0) \wedge (\forall i \,.\, H(i) \to H(s(i)) \to \forall x \,.\, H(x)$, where H stands for any formula of the language, i.e. the schema represents an infinite set of first-order axioms.

An isoinitial model of $\mathcal{NAT}$ is the term model generated by the constructors, equipped with the usual *sum* $(+)$ and *product* $(*)$.

The induction schema is useful for reasoning about properties of $+$ and $*$ that cannot be derived from the other axioms, e.g. associativity and commutativity. This illustrates the fact that in a framework we may have more than just an abstract data type definition, as we will see again later.

In general, a closed framework $\mathcal{F}$ is constructed incrementally from existing closed frameworks, and the new abstract data type axiomatised by $\mathcal{F}$ is completely defined thus. For example, a new sort T (possibly) depending on other pre-defined sorts is constructed from *constructors* declared as functions. The freeness axioms for the pre-defined sorts are imported and new axioms are added to define the (new) functions and relations on T.

The syntax of a framework $\mathcal{F}$ is thus similar to that used in algebraic abstract data types (e.g. [13, 29, 24]). However, whilst an algebraic abstract data type is an *initial* model ([12, 15]) of its specification, the intended model of $\mathcal{F}$ is an *isoinitial* model. Of course, a framework may have no intended (i.e. reachable isoinitial) model. We will only ever use frameworks with such models, i.e. adequate frameworks:

Definition 3. (Adequate Closed Frameworks)
A closed framework $\mathcal{F}$ is *adequate* if it has a reachable isoinitial model.

Now a framework $\mathcal{F}$ may also contain other forms of formulas, such as:

- *induction schemas* (as we saw in Example 1);
- *theorems*, i.e. proven properties of the problem domain (we will not encounter these in this paper);
- *specifications*, i.e. definitions of new symbols in terms of Σ symbols;
- (and even (*steadfast*)) *programs*.

However, such formulas are only admissible in $\mathcal{F}$ if their inclusion preserves $\mathcal{F}$'s adequacy (we will return to this in Section 2.2).

An open framework $\mathcal{F}(\Pi)$ has a non-empty set Π of parameters, that can be *instantiated* by closed frameworks as follows:

Definition 4. (Framework Instantiation)
Let $\mathcal{F}(\Pi)$ be an open framework with signature Σ, and $\mathcal{G}$ be a closed framework with signature Δ. If Π is the intersection of Σ and Δ, and $\mathcal{G}$ proves the p-axioms of $\mathcal{F}$, then the $\mathcal{G}$-*instance* of $\mathcal{F}$, denoted by $\mathcal{F}[\mathcal{G}]$, is the union of $\mathcal{F}$ and $\mathcal{G}$.

Instantiation may be defined in a more general way, involving renamings. Since renamings preserve adequacy and steadfastness, we can use this simpler definition without loss of generality.

Now we can define adequate open frameworks:

Definition 5. (Adequate Open Frameworks)
An open framework $\mathcal{F}(\Pi)$ is *adequate* if, for every adequate closed framework $\mathcal{G}$, the instance $\mathcal{F}(\Pi)[\mathcal{G}]$ is an adequate closed framework.

Adequacy means that parameter instantiation works properly, so we will also refer to adequate open frameworks as *parametric* frameworks.

Example 2. (Open Frameworks)
The following open framework axiomatises the (kernel of the) theory of lists with parametric element sort $Elem$ and parametric total ordering relation $\lhd$ (we use lower and upper case for elements and lists respectively):

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textbf{Specification Framework } \mathcal{LIST}(Elem, \lhd);} \\
\text{IMPORT:} & \mathcal{NAT}; \\
\text{SORTS:} & Nat, Elem, List; \\
\text{FUNCTIONS:} & nil \ : \ \to List; \\
 & \cdot \ : (Elem, List) \to List; \\
 & nocc : (Elem, List) \to Nat; \\
\text{RELATIONS:} & elemi : (List, Nat, Elem); \\
 & \lhd \ : (Elem, Elem);
\end{array}
$$

$$
\begin{array}{ll}
\text{AXIOMS:} & \text{C-AXS}(nil,\cdot); \\
& elemi(L,i,a) \leftrightarrow \exists h,T,j\,.\,L = h\cdot T \wedge \\
& (i = 0 \wedge a = h \vee i = s(j) \wedge elemi(T,j,a)); \\
& nocc(x,nil) = 0; \\
& a = b \rightarrow nocc(a, b\cdot L) = nocc(a,L)+1; \\
& \neg a = b \rightarrow nocc(a, b\cdot L) = nocc(a,L); \\
\text{P-AXIOMS:} & x \lhd y \wedge y \lhd x \leftrightarrow x = y; \\
& x \lhd y \wedge y \lhd z \rightarrow x \lhd z; \\
& x \lhd y \vee y \lhd x.
\end{array}
$$

where C-AXS$(nil,\cdot)$ contains Clark's Equality Theory (see [20]) for the list constructors $\cdot$ and nil, and the first-order induction schema $H(nil)\wedge(\forall a,J\,.\,H(J)\rightarrow H(a\cdot J))\rightarrow \forall L\,.\,H(L)$; the function $nocc(a,L)$ gives the number of occurrences of a in L, and $elemi(L,i,a)$ means a occurs at position i in L. The p-axioms are the parameter axioms for $\lhd$. In this case, they state that $\lhd$ must be a (non-strict) total ordering.

The parameters $Elem$ and $\lhd$ can be instantiated (after a possible renaming) by a closed framework proving the p-axioms. For example, suppose $\mathcal{INT}$ is a closed framework axiomatising the set Int of integers with total ordering $<$. Then $\mathcal{LIST}(Int,<)[\mathcal{INT}]$ becomes a closed framework with an isoinitial model where Int is the set of integers, Nat contains the natural numbers, and $List$ finite lists of integers. Note that $\mathcal{LIST}(Int,<)[\mathcal{INT}]$ contains the renaming of $Elem$ by Int and $\lhd$ by $<$. Note also that defined symbols can be renamed, when convenient. For example, we could rename $List$ by $ListInt$.

Whilst an adequate closed framework has *one* intended (isoinitial) model, an adequate open framework has a *class* of intended models.

2.2 Specifications

A framework is the context where a specification must be written, where it receives its proper meaning, and where we can reason about it and derive correct programs from it.

More formally, a specification S_δ in a framework is an axiom that defines a new relation δ in terms of the symbols Σ of the framework. Thus S_δ is a formula containing symbols from Σ and the new relation symbols δ:

Definition 6. (Specifications)
In a specification framework $\mathcal{F}(\Pi)$, a *specification* S_δ is a set of sentences that define new function or relation symbols δ in terms of the symbols Σ of $\mathcal{F}$. If S_δ contains symbols of Π, then it is called a *p-specification*.

S_δ can be interpreted as an *expansion operator* that associates with the isoinitial model i of $\mathcal{F}$ one or more classes of $(\Sigma+\delta)$-interpretations, that are the expansions of i defined by S_δ.

Definition 7. (Expansion)
Let j be a Σ-interpretation, and i be an expansion of j to $\Sigma + \delta$. We say that i is an *expansion* of j determined by a specification S (of δ) iff $\mathsf{i} \models S$.

We say that S_δ is *strict* if it defines just one expansion; it is *non-strict* if it defines more than one expansion. A more detailed discussion and classification of specifications can be found in [17].

For uniformity, in this paper, we shall use only *conditional specifications*, that is specifications of the form

$$\forall x : \mathsf{X}, \forall y : \mathsf{Y} \,.\, Q(x) \rightarrow (r(x,y) \leftrightarrow R(x,y))$$

where Q and R are formulas in the language of $\mathcal{F}$, and x:X, y:Y are (possibly empty) lists of sorted variables, with sorts in $\mathcal{F}$. Q is called the *input condition*, and R the *output condition* of the specification.

When Q is *true*, we drop it and speak of an *iff specification*. *Iff* specifications are strict, while in general a conditional specification is not.

In our approach, we maintain a clear distinction between frameworks and specifications. The latter introduce new symbols and assume their proper meaning only in the context of the framework. To distinguish the specified symbols from the signature of the framework, we will call them *s-symbols*. We also distinguish clearly between specifications and axioms.

Example 3. (Specifications)
In the open framework $\mathcal{LIST}(Elem, \lhd)$, we can specify the following functions and relations:

S-FUNCTIONS:
$$\begin{array}{l} l : List \rightarrow Nat; \\ | : (List, List) \rightarrow List; \end{array}$$

S-RELATIONS:
$$\begin{array}{rl} mem & : (Elem, List); \\ len & : (List, Nat); \\ append & : (List, List, List); \\ perm & : (List, List); \\ ord & : (List); \\ sort & : (List, List); \end{array}$$

SPECS:
$$\begin{array}{l} mem(e,L) \leftrightarrow \exists i \,.\, elemi(L,i,e); \\ len(L,n) \leftrightarrow \forall i \,.\, i < n \leftrightarrow \exists a \,.\, elemi(L,i,a); \\ n = l(L) \leftrightarrow len(L,n); \\ append(A,B,L) \leftrightarrow \; (\forall i, a \,.\, i < l(A) \rightarrow \\ \qquad\qquad\qquad\qquad (elemi(A,i,a) \leftrightarrow elemi(L,i,a))) \wedge \\ \qquad\qquad\qquad\qquad (\forall j, b \,.\, elemi(B,j,b) \leftrightarrow \\ \qquad\qquad\qquad\qquad elemi(L, j + l(A), b)); \\ perm(A,B) \leftrightarrow \forall e \,.\, nocc(e,A) = nocc(e,B); \\ C = A|B \leftrightarrow append(A,B,C); \end{array}$$

P-SPECS:
$$\begin{array}{l} ord(L) \leftrightarrow \forall i \,.\, elemi(L,i,e_1) \wedge elemi(L,s(i),e_2) \rightarrow e_1 \lhd e_2; \\ sort(L,S) \leftrightarrow perm(L,S) \wedge ord(S) \end{array}$$

As we will see in the next section, program predicates must be s-symbols. However, the specification of a program predicate may be non-strict and, in this case there may be many correct implementations, one for each expansion.

An s-symbol δ can be used also to expand the signature of the framework, in order to get a more expressive specification language. In this case, the specification S_δ is added to the axioms of the framework and δ is added to its signature. This operation will be called *framework expansion*.

We must use adequate framework expansions, i.e. expansions that preserve the adequacy of the framework. For example, the expansions of $\mathcal{LIST}(Elem, \lhd)$ by l, $|$, *mem*, *append*, *perm*, *ord* and *sort* in Example 3 can be shown to be adequate. In the following we will consider $\mathcal{F}$ thus expanded.

2.3 Closed and Open Programs

Open programs arise in both closed and open frameworks.

An *open program* may contain open relations, or parameters. The parameters of a program P are relations to be computed by other programs. They are not defined by P.

A relation in P is *defined* (by P) if and only if it occurs in the head of at least one clause of P. It is *open* if it is not defined (by P). An open relation in P is also called a *parameter* of P.

A program is *closed* if it does not contain open relations. We consider closed programs a special case of open ones.

Open programs are always given in the context of an (open or closed) framework $\mathcal{F}(\Pi)$. In $\mathcal{F}(\Pi)$, we will distinguish program sorts, i.e. sorts that can be used by programs. A closed program sort must have constructors (see axioms C-AXS(...)), and an open program sort may only be instantiated by program sorts. In programs, constant and function symbols may only be constructors. A program relation must be an s-symbol, i.e. it must have a specification.

Example 4. (Open Programs)
A possible open program for $sort(L, S)$ in $\mathcal{LIST}(Elem, \lhd)$ is the following:

$$\begin{array}{rl}
sort(L,S) \leftarrow & L = nil, S = nil \\
sort(L,S) \leftarrow & L = h.T, part(T,h,TL_1,TL_2), \\
& sort(TL_1,TS_1), sort(TL_2,TS_2), append(TS_1,h.TS_2,S) \\
part(L,p,S,B) \leftarrow & L = nil, S = nil, B = nil \\
part(L,p,S,B) \leftarrow & L = h.T, h \lhd p, part(T,p,TS,TB), \\
& S = h.TS \wedge B = TB \\
part(L,p,S,B) \leftarrow & L = h.T, \neg h \lhd p, part(T,p,TS,TB), \\
& S = TS \wedge B = h.TB
\end{array}$$

The s-symbols *sort* and *append* are specified in Example 3. The conditional specification of *part* can be found in Example 7.

2.4 Program Schemas

For representing schemas [1–6, 10, 14, 16, 21–23, 25–28], there are essentially two approaches, depending on the intended schema manipulations.

First, most researchers represent their schemas as higher-order expressions, sometimes augmented by extra-logical annotations and features, so that actual programs are obtained by applying higher-order substitutions to the schema. Such schemas could also be seen as first-order schemas, in the mathematical sense, namely designating an infinite set of programs that have the form of the schema. The reason why some declare them as higher-order is that they have applications in mind, such as schema-guided program transformation [7, 28, 11], where some form of higher-order matching between actual programs and schemas is convenient to establish applicability of the starting schema of a schematic transformation.

Second, Manna [21] advocates first-order schemas, where actual programs are obtained via an interpretation of the (relations and functions of the) schema. This is related to the approach we advocate here, namely that a schema $\mathcal{S}$ can also be represented as a (first-order) framework $\mathcal{F}$ containing an open program T, so that actual programs can be obtained by adding programs for some (but not necessarily all) of T's open relations. So there is no need to invent a new (or higher-order) schema language, at least in a first approximation (but see [6]).

Formally we define a program schema as follows:

Definition 8. (Program Schemas)
A *(program) schema* for a relation r is an open framework $\mathcal{S}(\Pi)$ containing a program P_r for r.
P_r is called the *template* of $\mathcal{S}(\Pi)$.
The p-axioms and the p-specifications are called the *constraints* of $\mathcal{S}(\Pi)$. Moreover, relation symbols of Π used only in specifications and (possibly) in p-axioms are called *s-parameters*.
A schema $\mathcal{S}$ *covers* a program P if ($\mathcal{S}$ and) its template can be instantiated to P.

We distinguish *s-parameters* from other parameters because in an instantiation by a closed framework $\mathcal{G}$ they can be replaced by *formulas* of the language of $\mathcal{G}$.[1] This does not hold for other parameters, since they must be instantiated by symbols of $\mathcal{G}$, in order to get a closed instance of the framework with a reachable isoinitial model.

Most definitions of schemas, with the laudable exception of the one by Smith [25, 26], reduce this concept to what we here call the template. Such definitions are thus merely syntactic, providing only a pattern of place-holders, with no concern about the semantics of the template, the semantics of the programs it covers, or the interactions between these place-holders. So a template by itself has no guiding power for teaching, programming, or synthesis, and the additional knowledge (corresponding to our constraints) somehow has to be hardwired into

[1] Of course, after the replacement, the p-axioms must be satisfied.

the system or person using the template. Despite the similarity, our definition is an enhancement of even Smith's definition, because we consider relational schemas (rather than "just" functional ones), open schemas (rather than just closed ones), and set up everything in the explicit, user-definable background theory of a framework (rather than in an implicit, predefined theory). The notion of constraint even follows naturally from, or fits naturally into, our view of correct schemas as (adequate) frameworks containing steadfast programs (see later), rather than as entities different from programs.

Example 5. (Program Schemas)
The schema in Figure 1 is our way of defining the divide-and-conquer schema. Note that the schema contains only p-axioms, and that $I_r, O_r, \ldots$ are s-parameters, i.e. they can be replaced by formulas in framework instantiations.

Schema $\mathcal{DC}(\mathsf{X}, \mathsf{Y}, \mathsf{H}, \prec, I_r, O_r, I_{dec}, O_{dec})$;

SORTS: $\mathsf{X}, \mathsf{Y}, \mathsf{H}$;

RELATIONS: $I_r, I_{dec} : (\mathsf{X})$;
$O_r : (\mathsf{X}, \mathsf{Y})$;
$O_{dec} : (\mathsf{X}, \mathsf{H}, \mathsf{X}, \mathsf{X})$;

P-AXIOMS:
$$I_{dec}(x) \land O_{dec}(x, hx, tx_1, tx_2) \to I_r(tx_1) \land tx_1 \prec x \land I_r(tx_2) \land tx_2 \prec x; \qquad (c_1)$$
$$I_{dec}(x) \to \exists h, x_1, x_2 \,.\, O_{dec}(x, h, x_1, x_2); \qquad (c_2)$$

P-SPECS:
$$I_r(x, y) \to (r(x, y) \leftrightarrow O_r(x, y)) \qquad (S_r)$$
$$I_r(x) \to (primitive(x) \leftrightarrow \neg I_{dec}(x)) \qquad (S_{prim})$$
$$I_{dec}(x) \to (decompose(x, hx, tx_1, tx_2) \leftrightarrow O_{dec}(x, hx, tx_1, tx_2)) \qquad (S_{dec})$$
$$I_r(x) \land \neg I_{dec}(x) \to (solve(x, y) \leftrightarrow O_r(x, y)) \qquad (S_{solve})$$
$$O_{dec}(x, hx, tx_1, tx_2) \land O_r(tx_1, ty_1) \land O_r(tx_2, ty_2) \to (compose(hx, ty_1, ty_2, y) \leftrightarrow O_r(x, y)) \qquad (S_{comp})$$

TEMPLATE:
$$r(x, y) \leftarrow primitive(x), solve(x, y)$$
$$r(x, y) \leftarrow \neg primitive(x), decompose(x, hx, tx_1, tx_2), r(tx_1, ty_1), r(tx_2, ty_2), compose(hx, ty_1, ty_2, y) \qquad (T_r)$$

Fig. 1. A divide-and-conquer schema.

3 Correct Schemas

A model-theoretic definition of correctness of open programs in a framework, called *steadfastness*, is given in [19]. Here, we give a less abstract, but more conventional definition. In this paper, for simplicity, we only give definitions and results that work for definite programs. Nevertheless they extend to normal programs, under suitable termination assumptions.

For closed programs in closed frameworks, we have the classical notion of (total) correctness:

Definition 9. (Total Correctness)
In a closed framework $\mathcal{F}$ with isoinitial model i, a closed program P_r for relation r is *totally correct* wrt its specification S_r

$$\forall x : \mathsf{X}, \forall y : \mathsf{Y}.\ I_r(x) \rightarrow (r(x,y) \leftrightarrow O_r(x,y)) \qquad (S_r)$$

iff for all $t : \mathsf{X}$ and $u : \mathsf{Y}$ such that $\mathsf{i} \models I_r(t)$ we have*:*

$$i \models O_r(t,u) \text{ iff } P_r \vdash r(t,u) \qquad (1)$$

If P_r satisfies the if-part of (1), it is *partially correct* (wrt S_r). If it satisfies the only-if part, then it is *total.*

Total correctness as defined here is unsatisfactory for logic programs, since it cannot deal with different cases of termination. In particular, we consider the following two cases:

(i) P_r is totally correct wrt to S_r, and terminates with either success or finite failure, for every ground goal $\leftarrow r(t,u)$ such that $\mathsf{i} \models I_r(t)$.
In this case, P_r correctly decides r, and we say that P_r is *correct* wrt $TC(r, S_r)$.

(ii) P_r is partially correct wrt S_r, and, for every ground $t : \mathsf{X}$ such that $\mathsf{i} \models I_r(t)$, the computation with open goal $\leftarrow r(t,y)$ terminates with at least one answer $y = u$.
In this case, P_r correctly computes a *selector* of r (i.e. a function or relation that, for every input x such that $I_r(x)$, selects at least one output y such that $O_r(x,y)$), and we say that P_r is *correct* wrt $PC(r, S_r)$.

$TC(r, S_r)$ and $PC(r, S_r)$ are called *termination requirements.*

It is easy to see that total correctness is too weak for case (i), since a totally correct P_r could fail to terminate for a false $r(t,u)$, and too strong for case (ii), since for computing a selector, we do not need success for every true $r(t,u)$). Therefore, a *specification of a program relation* r will be of the form $(S_r, S_1, \ldots, S_n, T_r \Leftarrow T_1, \ldots, T_n)$, i.e. it will include a termination requirement. Moreover, in the definition of steadfastness, we will consider correctness wrt (S_i, T_i) and (S_r, T_r), instead of total correctness.

Termination and termination requirements are an important issue. For lack of space, however, we will not further deal with them here.

The definition of correctness wrt (S_r, T_r) is still unsatisfactory. First, it defines the correctness of P_r in terms of the programs for the relations other than r, rather than in terms of their specifications. Second, all the programs for these relations need to be included in P_r (this follows from P_r being closed), even though it might be desirable to discuss the correctness of P_r without having to fully solve it (i.e. we may want to have an open P_r). So, the abstraction achieved through the introduction (and specification) of the new relations is wasted.

This leads us to the following notion of steadfastness of an open program in a closed framework.

Definition 10. (Steadfastness in a Closed Framework)
In a closed framework $\mathcal{F}$, let P_r be an open program for r, with parameters p_1, $\ldots$, p_n, specifications $S_r, S_1, \ldots, S_n$, and termination requirements $T_r, T_1, \ldots, T_n$.
P_r is *steadfast* in $\mathcal{F}$ if, for any closed programs $P_1, \ldots, P_n$ that compute $p_1, \ldots, p_n$ such that P_i is correct wrt (S_i, T_i), the (closed) program $P_r \cup P_1 \cup \ldots \cup P_n$ is correct wrt (S_r, T_r).

Now we can define steadfastness in an open framework:

Definition 11. (Steadfastness in an Open Framework)
In an open framework $\mathcal{F}(\Pi)$, let P_r be an open program for r, with parameters $p_1, \ldots, p_n$, specifications $S_r, S_1, \ldots, S_n$, and termination requirements $T_r, T_1, \ldots, T_n$.
P_r is *steadfast* in $\mathcal{F}(\Pi)$ if it is steadfast in every instance $\mathcal{F}[\mathcal{G}]$ for a closed framework $\mathcal{G}$.

This is similar to Deville's notion of 'correctness in a set of specifications' [5, p.76], except that his specifications and programs are not set within frameworks. Moreover, we also (but not in this paper, hence the simplified definition above) consider other cases of steadfastness, namely where *several* (but not necessarily all) defined relations of a program are known by their specifications, the other defined relations being known by their clauses only.

Now we can formally define correctness for program schemas:

Definition 12. (Correct Program Schemas)
A (program) schema for a relation r, i.e. an (adequate) open framework $\mathcal{S}(\Pi)$ containing a template P_r for r, is *correct* iff P_r is steadfast in $\mathcal{S}(\Pi)$.

Example 6. (Correct Program Schemas)
We will now show that the schema $\mathcal{S}$ in Example 5 is correct because ($\mathcal{S}$ is an adequate framework and) its template T_r:

$$\begin{aligned} r(x,y) &\leftarrow primitive(x), solve(x,y) \\ r(x,y) &\leftarrow \neg primitive(x), decompose(x,hx,tx_1,tx_2), \\ &\qquad r(tx_1,ty_1), r(tx_2,ty_2), compose(hx,ty_1,ty_2,y) \end{aligned} \tag{T_r}$$

is steadfast, if we add to it the following termination requirement:

$$\text{T-REQS: } PC(r,S_r) \Leftarrow TC(primitive, S_{primitive}), PC(solve, S_{solve}), \\ PC(decompose, S_{decompose}), PC(compose, S_{compose})$$

In fact we can derive the whole schema (including these termination requirements) from our attempt to prove that T_r is steadfast. Thus this example also serves to illustrate how we might derive correct schemas.

In the absence of constraints, an open program such as T_r has no fixed meaning, since it covers *every* program, which is obviously nonsensical. Indeed, it would suffice to instantiate *primitive* by *true*, and *solve* by the given program! However, we can give this template an informal intended semantics, as follows.

For an arbitrary relation r over formal parameters x and y, the program is to determine the value(s) of y corresponding to a given value of x. Two cases arise: either x has a value (when $primitive(x)$ holds) for which y can be easily directly computed (through $solve$), or x has a value (when $\neg primitive(x)$ holds) for which y cannot be so easily directly computed; the divide-and-conquer principle is then applied by:

1. *dividing* (through *decompose*) x into a term hx and two terms tx_1 and tx_2 that are both of the same sort as x but smaller than x according to some well-founded order,
2. *conquering* (through r) to determine values of ty_1 and ty_2 corresponding to tx_1 and tx_2, respectively,
3. *combining* (through *compose*) terms hx, ty_1, ty_2 to build y.

Just as the semantics of open programs is defined parametrically, we can do the same for this template, and whilst so doing, we can enforce the informal semantics and supply the corresponding axioms of the open relations (i.e. the constraints of the schema). We can do so by introducing an open framework $\mathcal{S}(I_r, O_r, \ldots)$ with a signature containing the sorts of the template and the open relation symbols $I_r, O_r, \ldots$ We can abduce the constraints of the schema by proving at an abstract level that T_r is steadfast in $\mathcal{S}$, wrt the specifications of r and the unknown axioms of the open relations the template introduces, and enforcing the informal semantics of the template during this proof. The proof itself must of course fail due to the lack of knowledge about r and the introduced open relations, but the reasons of this failure can be used to abduce the necessary relationships between r and these open relations. These relationships are of course the *constraints* on the open relations of the template!

Program T_r is steadfast in $\mathcal{S}$ if it is steadfast in every instance of $\mathcal{S}$. So let $\mathcal{F}$ be a *generic* instance $\mathcal{S}[\mathcal{G}]$, where $\mathcal{G}$ is a closed framework. Suppose the specification of r in $\mathcal{F}$ is:

$$\forall x : \mathsf{X}, \forall y : \mathsf{Y} \,.\, I_r(x) \rightarrow (r(x,y) \leftrightarrow O_r(x,y)) \qquad (S_r)$$

We have to find (at least) the p-specifications (in $\mathcal{F}$) S_{prim}, S_{solve}, S_{dec}, S_{comp} of $primitive$, $solve$, *decompose*, *compose*, respectively, such that T_r is a steadfast program for r in $\mathcal{F}$. For each S_i, let the input and output conditions be I_i and O_i respectively.

Suppose also that we only require that instances of the template T_r be partially correct and terminating (i.e. $PC(r, S_r)$ holds for each instance). Let t be a ground term such that $I_r(t)$, and consider the open goal $\leftarrow r(t, Y)$. We have to prove that T_r terminates with some answer $Y = u$. We have the following possibilities:

1. The next goal is $\leftarrow primitive(t), solve(t, Y)$, and $primitive(t)$ succeeds. We are blocked, but we can unblock the situation by abducing that $PC(solve, S_{solve})$ holds and that:

$$I_r(t) \wedge O_{prim}(t) \rightarrow I_{solve}(t) \qquad (2)$$

2. The next goal is $\leftarrow primitive(t), \ldots$ or $\leftarrow \neg primitive(t), \ldots$, and the call to $primitive(t)$ does not terminate. We have to exclude this case, so we assume $TC(primitive, S_{primitive})$ and:

$$I_r(t) \rightarrow I_{prim}(t) \tag{3}$$

3. The next goal is $\leftarrow \neg primitive(t), \ldots$ and $primitive(t)$ finitely fails. Then we get the goal $\leftarrow decompose(t, HX, TX_1, TX_2), r(TX_1, TY_1), r(TX_2, TY_2)$, $compose(HX, TY_1, TY_2, Y)$. Again, we are blocked, but we can unblock the situation by assuming:

$$\begin{aligned} I_{dec}(t) \wedge O_{dec}(t, HX, TX_1, TX_2) \rightarrow I_r(TX_1) \wedge TX_1 \prec t \wedge \\ I_r(TX_2) \wedge TX_2 \prec t \end{aligned} \tag{4}$$

where $\prec$ is a well-founded relation.[2] By structural induction, we can see that, if $PC(decompose, S_{decompose})$, $PC(compose, S_{compose})$, and

$$I_r(t) \wedge \neg O_{prim}(t) \rightarrow I_{dec}(t) \tag{5}$$

$$\begin{aligned} &I_{dec}(t) \wedge O_{dec}(t, HX, TX_1, TX_2) \wedge O_r(TX_1, TY_1) \wedge O_r(TX_2, TY_2) \\ &\rightarrow I_{comp}(HX, TY_1, TY_2, Y) \end{aligned} \tag{6}$$

then the computation terminates with an answer for Y. Indeed, by the induction hypothesis, we can assume that, for $TX_1 \prec t$ and $TX_2 \prec t$, program T_r computes TY_1 and TY_2 such that $O_r(TX_1, TY_1) \wedge O_r(TX_2, TY_2)$ holds.

Thus, we have abduced:

$$\begin{aligned} PC(r, S_r) \Leftarrow\ & TC(primitive, S_{primitive}), PC(solve, S_{solve}), \\ & PC(decompose, S_{decompose}), PC(compose, S_{compose}) \end{aligned} \tag{7}$$

$PC(solve, S_{solve})$, $PC(decompose, S_{decompose})$, and $PC(compose, S_{compose})$ admit correct programs only if their specifications S_{solve}, S_{dec}, and S_{comp} are such that

$$\begin{aligned} &I_{dec}(t) &&\rightarrow \exists HX, TX_1, TX_2 \,.\, O_{dec}(t, HX, TX_1, TX_2) \\ &I_{comp}(HX, TY_1, TY_2) &&\rightarrow \exists Y \,.\, O_{comp}(HX, TY_1, TY_2, Y) \\ &I_{solve}(t) &&\rightarrow \exists Y \,.\, O_{solve}(t, Y) \end{aligned} \tag{8}$$

Now we have to prove that T_r is partially correct. For this, we assume:[3]

$$\begin{aligned} &r(x, y) &&\leftrightarrow \neg I_r(x) \vee O_r(x, y) \\ &primitive(x) &&\leftrightarrow \neg I_{prim}(x) \vee O_{prim}(x) \\ &solve(x, y) &&\leftrightarrow \neg I_{solve}(x) \vee O_{solve}(x, y) \\ &decompose(x, hx, tx_1, tx_2) &&\leftrightarrow \neg I_{dec}(x) \vee O_{dec}(x, hx, tx_1, tx_2) \\ &compose(hx, ty_1, ty_2, y) &&\leftrightarrow \neg I_{comp}(hx, ty_1, ty_2, y) \vee \\ & &&\quad\ O_{comp}(hx, ty_1, ty_2, y) \end{aligned} \tag{9}$$

[2] In the isoinitial model and, hence, in the Herbrand base of the closed version T'_r of T_r.

[3] Here we make use of the fact that if $\mathcal{F} \cup \{\forall x : \mathsf{X}, \forall y : \mathsf{Y} \,.\, r(x, y) \leftrightarrow \neg I_r(x) \vee O_r(x, y)\} \vdash T_r$, then T_r is partially correct wrt S_r. See [19].

We have to prove that $\mathcal{F} \cup (9) \vdash T_r$. Let us try to prove the first clause. We abduce:

$$\neg I_r(x) \vee O_r(x,y) \leftarrow (\neg I_{prim}(x) \vee O_{prim}(x)) \wedge (\neg I_{solve}(x) \vee O_{solve}(x,y))$$

This is logically equivalent to

$$O_r(x,y) \leftarrow I_r(x) \wedge (\neg I_{prim}(x) \vee O_{prim}(x)) \wedge (\neg I_{solve}(x) \vee O_{solve}(x,y))$$

Since any instance $\mathcal{F}$ must prove the p-axioms of $\mathcal{S}$ and since we have already abduced (2) and (3), we can simplify this to:

$$O_r(x,y) \leftarrow I_r(x) \wedge O_{prim}(x) \wedge O_{solve}(x,y) \tag{10}$$

By an analogous reasoning, from the attempt of proving the second clause, we obtain the simplified p-axiom:

$$\begin{aligned} O_r(x,y) \leftarrow I_r(x) \wedge \neg O_{prim}(x) \wedge O_{dec}(x, hx, tx_1, tx_2) \wedge \\ O_r(tx_1, ty_1) \wedge O_r(tx_2, ty_2) \wedge O_{comp}(hx, ty_1, ty_2, y) \end{aligned} \tag{11}$$

As before, the simplification of the input conditions is due to the p-axioms already abduced.

By the above proof, we have abduced a schema containing a suitable signature, our template, the termination requirements (7), and the p-axioms (2) ... (11).

This schema is correct, but it contains redundancies, due to constraints that make some parameters depend on others. We can try to simplify it as follows:

1. When we use the schema, we know the actual specification, which specifies in $\mathcal{F}$ a program P'_r such that $PC(r, S_r)$ holds, so we can instantiate I_r, O_r, X, and Y.
2. Then we instantiate $\prec$ by a well-founded relation on X.
3. Now the two constraints (10) and (11) contain four unknown output conditions. If we fix some of them, we can hope to deduce the other ones, and to simplify some constraints. In a divide-and-conquer strategy, it is reasonable to assume that we first choose the decomposition, i.e. I_{dec} and O_{dec}. We now have to infer I_{prim} and O_{prim} such that they satisfy the constraints (3) and (5). A possible reduction is based on the observation that (5) is logically equivalent to $I_r(x) \rightarrow (O_{prim}(x) \leftarrow \neg I_{dec}(x))$. We replace $\leftarrow$ by $\leftrightarrow$. By identifying I_{prim} and I_r, we satisfy (3) and can thus reduce S_{prim} to:

   $$I_r(x) \rightarrow (primitive(x) \leftrightarrow \neg I_{dec}(X))$$

   hence setting O_{prim} to $\neg I_{dec}$.
4. Now, by substitution and a simple logical manipulation, we transform (10) and (11) into:

   $$I_r(x) \wedge \neg I_{dec}(x) \rightarrow (O_r(x,y) \leftarrow O_{solve}(x,y))$$
   $$I_r(x) \wedge I_{dec}(x) \wedge O_{dec}(x, hx, tx_1, tx_2) \wedge O_r(tx_1, ty_1) \wedge O_r(tx_2, ty_2) \rightarrow$$
   $$(O_r(x,y) \leftarrow O_{comp}(hx, ty_1, ty_2, y))$$

where the unknown predicates O_{comp} and O_{solve} are defined, on the right-hand side of $\rightarrow$, by $\leftarrow$ instead of $\leftrightarrow$. We can assume stronger[4] constraints, by replacing $\leftarrow$ by $\leftrightarrow$. We get a conditional definition of O_{solve} and O_{comp}. Moreover, S_{solve} and S_{comp} can be reduced to:

$$\begin{aligned} I_r(x) \wedge \neg I_{dec}(x) &\rightarrow (solve(x,y) \leftrightarrow O_r(x,y)) \\ O_{dec}(x, hx, tx_1, tx_2) \wedge O_r(tx_1, ty_1) \wedge O_r(tx_2, ty_2) &\rightarrow (compose(hx, ty_1, ty_2, y) \\ &\qquad \leftrightarrow O_r(x,y)) \end{aligned}$$

Using the reduced specifications, we see that the constraints (2), (6), and the second and third constraints of (8) become proved.

Therefore we obtain the schema $\mathcal{DC}$ as defined in Example 5.

The above abduction process proves the following theorem:

Theorem 1. (Correctness of the divide-and-conquer schema)
The schema $\mathcal{DC}$ in Example 5, with the addition of the termination requirement (7), is correct, i.e. it contains a steadfast template.

This theorem is related to the one given by Smith [25] for a divide-and-conquer schema in functional programming. The innovations here are that we use specification frameworks and that we can thus also consider open programs. Moreover, we could also prove total correctness (and not just partial correctness as we have done here), because we are in a relational setting. Finally, we eliminated Smith's *Strong Problem Reduction Principle* by endeavouring to achieve these objectives.

Finally, we can specialise a schema to a data type. For example, we can incorporate the data type of lists with generic elements, by incorporating in $\mathcal{S}$ the framework $\mathcal{LIST}(\mathsf{X}, \lhd)$, or part of it. All the properties of $\mathcal{S}$ are inherited, and we can add further properties. For example, we can already know at the schema level that the relation defined by $A \prec B \leftrightarrow l(A) < l(B)$ is a well-founded relation in every instance of the schema, and therefore that it is one of the candidates to be used when instantiating the template.

4 Using Correct Schemas in Practice

Our characterisation of correct program schemas allows us to synthesise steadfast open programs. This is a significant step forwards in the field of synthesis, because the synthesised programs are then not only correct, but also *a priori* correctly reusable. This is achieved by means of steadfast templates together with their constraints. However, since we have identified correct templates with steadfast programs, there seems to be some circularity in our argument: how can we guide the synthesis of steadfast programs by steadfast programs? The answer is that some open programs are "more open" than others, and that such

[4] This reduces the search space, but, in general, it could cut some solutions. We do not discuss this issue here.

"more open" programs thus have more "guiding power," especially considering the specifications for their open relations. In [9], we discuss the synthesis of steadfast programs guided by correct schemas. To conclude this paper, in this section we briefly outline the main ideas.

Much of the program synthesis process can be pre-computed at the level of "completely open" schemas. The key to pre-computation is such a schema, especially its constraints. These specifications can be seen as an "overdetermined system of equations (in a number of unknowns)", which may be unsolvable as it stands (for instance, this is the case for the divide-and-conquer schema in Example 5). An arbitrary instantiation (through program extension), according to the informal semantics of the template, of one (or several) of its open relations may then provide a "jump-start", as the set of equations may then become solvable.

This leads us to the notion of *synthesis strategy* (cf. Smith's work [25]), as a pre-computed (finite) sequence of synthesis steps, for a given schema. A strategy has two phases, stating (i) which parameter(s) to arbitrarily instantiate first (by re-use), and (ii) which specifications to "set up" next, based on a pre-computed propagation of these instantiation(s). Once correct programs have been synthesised from these new specifications (using the synthesiser all over again, of course), they can be composed into a correct program for the original specified relation, according to the template. There can be several strategies for a given schema (e.g., Smith [25] gives three strategies for a divide-and-conquer schema), depending on which parameter(s) are instantiated first (e.g., *decompose* first, or *compose* first, or both at the same time).

Synthesis is thus a recursive problem reduction process followed by a recursive solution composition process, where the problems are specifications and the solutions are programs. Problem reduction stops when a "sufficiently simple" problem is reached, i.e. a specification that "reduces to" another specification for which a program is known and can thus be re-used. This is thus the "base case" of synthesis, and requires a formalisation of the process of re-use (see [9] for details).

Let us illustrate these ideas on the divide-and-conquer schema. In [8], we design the following strategy for it:

1. **Select an induction parameter** among x and y (such that it is of an inductively defined sort). Suppose, without loss of generality, that x is selected.
2. **Select (or construct) a well-founded order** over the sort of the induction parameter. Suppose that $\prec$ is selected (from a "knowledge base").
3. **Select (or construct) a decomposition operator** *decompose*. Suppose that the following specification is selected (from a "knowledge base"):

$$\begin{array}{l}\forall x, t_1, t_2 : \mathsf{X}, \forall h : \mathsf{H}\,. \\ I_{dec}(x) \rightarrow (decompose(x, h, t_1, t_2) \leftrightarrow Dec(x, h, t_1, t_2)).\end{array} \qquad (S'_{dec})$$

4. **Set up the specification of the discriminating operator** *primitive.* This amounts to first deriving a formula G such that

$$\begin{aligned}\mathcal{F} \models \forall x, tx_1, tx_2 : \mathsf{X}, \forall hx : \mathsf{H} \,.\, G(x) \wedge \\ Dec(x, hx, tx_1, tx_2) \leftrightarrow I_r(tx_1) \wedge I_r(tx_2) \wedge tx_1 \prec x \wedge tx_2 \prec x,\end{aligned}$$

and then setting up the following specification:

$$\forall x : \mathsf{X} \,.\, primitive(x) \leftrightarrow \neg(I_{dec}(x) \wedge G(x)). \qquad (S'_{prim})$$

5. **Set up the specification of the solving operator** *solve.* All place-holders of S_{solve} are known now, so we can set up a specification S'_{solve} by instantiating inside S_{solve}.
6. **Set up the specification of the composition operator** *compose.* Similarly, all place-holders of S_{comp} are known now, so we can set up a specification S'_{comp} by instantiating inside S_{comp}.

Four specifications (S'_{dec}, S'_{prim}, S'_{solve}, and S'_{comp}) have been set up now, so four auxiliary syntheses can be started from them, using the same overall synthesiser again, but not necessarily the (same) strategy for the (same) divide-and-conquer schema. The programs P_{dec}, P_{prim}, P_{solve}, and P_{comp} resulting from these auxiliary syntheses are then added to the open program P_r of the schema, which extension of P_r is guaranteed, by Theorem 1, to be steadfast.

Example 7. (A Sample Synthesis)
Suppose in $\mathcal{LIST}(Elem, \lhd)$ we want a steadfast sorting program with termination requirement $PC(sort, S_{sort})$.

First, we select the specification of a decomposition operator *part*, partitioning a list L into its first element h, the list A of its remaining elements that are smaller (according to $\lhd$) than h, and the list B of its remaining elements that are not smaller (according to $\lhd$) than h:

$$\begin{aligned}&\neg L = nil \rightarrow (part(L, h, A, B) \leftrightarrow \\ &L = h.T \wedge perm(A|B, T) \wedge A \sqsubset h \wedge B \sqsupset h)\end{aligned} \qquad (S_{part})$$

where the following axioms:

$$\begin{aligned}L \sqsubset e \leftrightarrow \forall x \,.\, mem(x, L) \rightarrow x \lhd e \\ L \sqsupset e \leftrightarrow \forall x \,.\, mem(x, L) \rightarrow \neg x \lhd e\end{aligned}$$

are added to $\mathcal{LIST}(Elem, \lhd)$.

In [9], we synthesise the following extension of the divide-and-conquer template by using the strategy outlined above:

$$
\begin{aligned}
sort(L,S) &\leftarrow primitive(L), solve(L,S)\\
sort(L,S) &\leftarrow \neg primitive(L), part(L,h,A,B),\\
&\qquad sort(A,C), sort(B,D), compose(h,C,D,S)\\
primitive(L) &\leftarrow L = nil\\
solve(L,S) &\leftarrow S = nil\\
part(L,h,A,B) &\leftarrow L = h.T, part(T,h,A,B)\\
part(L,p,A,B) &\leftarrow L = nil, A = nil, B = nil\\
part(L,p,A,B) &\leftarrow L = h.T, h \lhd p, part(T,p,TA,TB), A = h.TA, B = TB\\
part(L,p,A,B) &\leftarrow L = h.T, \neg h \lhd p, part(T,p,TA,TB), A = TA, B = h.TB\\
compose(e,C,D,S) &\leftarrow append(C, e.D, S)
\end{aligned}
$$

This is the classical Quicksort program. After a series of unfolding steps, this program can easily be transformed into the program of Example 4. Note that this is an open program, as there are no clauses yet for *append*, nor for $\lhd$.

5 Conclusion

We have shown that program schemas can be expressed as open (first-order) specification frameworks containing steadfast open programs, and we have outlined how correct and *a priori* correctly reusable (divide-and-conquer) programs can be synthesised, in a schema-guided way, from formal specifications expressed in the first-order language of a framework. These aspects of schema-guided synthesis are our new contribution.

Our work is very strongly influenced by Smith's pioneering work [25] in functional programming in the early 1980s. This is, in our opinion, inevitable, as this approach seems to be the only structured approach to synthesis. Our work is however *not* limited to simply transposing Smith's achievements to the logic programming paradigm: indeed, we have also enhanced the theoretical foundations by adding frameworks, enlarged the scope of synthesis by allowing the synthesis of a larger class of non-deterministic programs, and simplified (the formulation and proof of) the theorem on the correctness of the divide-and-conquer schema (Theorem 1).

Future work includes redoing the constraint abduction process for a more general (divide-and-conquer) template, namely where $nonPrimitive(x)$ is not necessarily $\neg primitive(x)$, and developing the corresponding strategies, in order to allow the synthesis of a larger class of non-deterministic programs.

Other strategies for the divide-and-conquer schema need to be elaborated, and other design methodologies need to be captured in program schemas and strategies.

Another important objective is the development of a proof system for deriving antecedents (as needed at Step 4 of the given strategy) and for obtaining simplifications of output conditions (the specifications S'_{solve} and S'_{comp} are often amenable to considerable simplifications). Eventually, a proof-of-concept implementation of the outlined synthesiser (and the adjunct proof system) is planned.

Acknowledgements

We wish to thank Doug Smith for his pioneering work that inspired us, John Gallagher for pointing out a technical error in our presentation at the workshop, and Yves Deville for his insightful comments which will help us in our future work. This work was partially supported by the European Union HCM Project on Logic Program Synthesis and Transformation, contract no. 93/414.

References

1. D. Barker-Plummer. Cliche Programming in Prolog. In M. Bruynooghe, editor, *Proc. META 90*, pages 246-256, 1992.
2. E. Chasseur and Y. Deville. Logic program schemas, semi-unification and constraints. This volume.
3. N. Dershowitz. *The Evolution of Programs*. Birkhäuser, 1983.
4. Y. Deville and J. Burnay. Generalization and program schemata: A step towards computer-aided construction of logic programs. In E.L. Lusk and R.A. Overbeek, editors, *Proc. NACLP'89*, pages 409–425. MIT Press, 1989.
5. Y. Deville. *Logic Programming: Systematic Program Development*. Addison-Wesley, 1990.
6. P. Flener. *Logic Program Synthesis from Incomplete Information*. Kluwer, 1995.
7. P. Flener and Y. Deville. Logic program transformation through generalization schemata. In M. Proietti, editor, *Proc. LOPSTR'95*, pages 171–173. *LNCS* 1048, Springer-Verlag, 1996.
8. P. Flener and K.-K. Lau. Program Schemas as Steadfast Programs and their Usage in Deductive Synthesis. Tech Rep BU-CEIS-9705, Bilkent University, Ankara, Turkey, 1997.
9. P. Flener, K.-K. Lau, and M. Ornaghi, Correct-schema-guided Synthesis of Steadfast Programs, *Proc. 12th IEEE International Automated Software Engineering Conference*, pages 153-160, IEEE Computer Society, 1997.
10. T.S. Gegg-Harrison. Representing logic program schemata in λ-Prolog. In L. Sterling, editor, *Proc. ICLP'95*, pages 467–481. MIT Press, 1995.
11. T.S. Gegg-Harrison. Extensible Logic Program Schemata. In J. Gallagher, editor, *Proc. LOPSTR'96*, *LNCS* 1207, pages 256-274, Springer-Verlag, 1997.
12. 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.
13. 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.
14. A. Hamfelt and J. Fischer-Nilsson. Inductive metalogic programming. In S. Wrobel, editor, *Proc. ILP'94*, pages 85–96. *GMD-Studien* Nr. 237, Sankt Augustin, Germany, 1994.
15. W. Hodges. Logical features of Horn clauses. In D.M. Gabbay, C.J. Hogger, and J.A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, *Volume 1: Logical Foundations*, pages 449-503, Oxford University Press, 1993.
16. A.-L. Johansson. Interactive program derivation using program schemata and incrementally generated strategies. In Y. Deville, editor, *Proc. LOPSTR'93*, pages 100–112. Springer-Verlag, 1994.

17. K.-K. Lau and M. Ornaghi. Forms of logic specifications: A preliminary study. In J. Gallagher, editor, *Proc. LOPSTR'96*, pages 295–312, *LNCS* 1207, Springer-Verlag, 1997.
18. K.-K. Lau, M. Ornaghi, and S.-Å. Tärnlund. The halting problem for deductive synthesis of logic programs. In P. van Hentenryck, editor, *Proc. ICLP'94*, pages 665–683. MIT Press, 1994.
19. K.-K. Lau, M. Ornaghi, and S.-Å. Tärnlund. Steadfast logic programs. *J. Logic Programming*, submitted.
20. J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2nd edition, 1987.
21. Z. Manna. *Mathematical Theory of Computation*. McGraw-Hill, 1974.
22. E. Marakakis and J.P. Gallagher. Schema-based top-down design of logic programs using abstract data types. In L. Fribourg and F. Turini, editors, *Proc. LOPSTR/META'94*, pages 138–153, *LNCS* 883, Springer-Verlag, 1994.
23. J. Richardson and N. Fuchs. Development of correct transformational schemata for Prolog programs. This volume.
24. D. Sannella and A. Tarlecki. Essential concepts of algebraic specification and program development. *Formal Aspects of Computer Science*, forthcoming.
25. D.R. Smith. Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence* 27(1):43–96, 1985.
26. D.R. Smith. KIDS: A semiautomatic program development system. *IEEE Trans. Software Engineering* 16(9):1024–1043, 1990.
27. L.S. Sterling and M. Kirschenbaum. Applying techniques to skeletons. In J.-M. Jacquet, editor, *Constructing Logic Programs*, pages 127–140. John Wiley, 1993.
28. W.W. Vasconcelos and N.E. Fuchs. An opportunistic approach for logic program analysis and optimisation using enhanced schema-based transformations. In M. Proietti, editor, *Proc. LOPSTR'95*, pages 174–188. *LNCS* 1048, Springer-Verlag, 1996.
29. M. Wirsing. Algebraic specification. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 675–788. Elsevier, 1990.