Program Schemas as Steadfast Programs and their Usage in Deductive Synthesis

Pierre Flener
Department of Computer Engineering and Information Science
Bilkent University, 06533 Bilkent, Ankara, Turkey
Email: pf@cs.bilkent.edu.tr

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

Abstract

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 nor (all) their actual data structures. We show that schemas can be expressed as first-order open programs, that is where some of the used relations are left undefined. Compared to higher-order representations, this considerably simplifies the semantics and manipulations of schemas. Actually, our schemas are steadfast open programs, expressed in the first-order sorted language of an axiomatisation (called framework) of the application domain. We give correctness and steadfastness (parametric correctness) criteria, the latter entailing a-priori-correct reusability. All this is illustrated by means of a schema capturing the divide-and-conquer methodology, and we derive the abstract conditions under which it is steadfast wrt an arbitrary specification in an arbitrary framework. Finally, we show how to use schemas for effectively guiding the deductive synthesis of steadfast programs from complete specifications (i.e. complete axiomatisations of the problem), and illustrate this by developing a strategy for synthesising divide-and-conquer programs as well as by actually synthesising a Quicksort program.

1 Introduction

Program schemas are a popular element of the programming folklore: 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 nor (all) their actual data structures. Program schemas have been shown useful in a number of applications, such as proving properties of programs, teaching programming to novices, guiding the manual construction of programs, debugging programs, transforming programs, and guiding the (semi-)automatic synthesis of programs. 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 (for an overview, see [6]).

In this paper, we take a different approach and show that schemas can also be expressed as first-order programs, but where some of the used relations are left undefined. This considerably simplifies the expression of the semantics of schemas (note that often none is given), not to mention their manipulations. Indeed, we show that program schemas can be expressed as steadfast open programs, thus linking this topic with the research on frameworks of the second author (see, e.g., [24]). Briefly, this means that we express specifications and programs in the first-order sorted language of an axiomatisation (called framework) of the application domain. We also give correctness and steadfastness criteria. The latter criterion entails a-priori-correct reusability, which is a very desirable feature of programs.

As of now, we mainly aim at using schemas for guiding the synthesis of steadfast programs from complete specifications (i.e. complete axiomatisations of the problem). Therefore, we combine some ideas of the previous work on deductive synthesis of the second author [18] with the ideas on schema-guided synthesis of the first author [6], and follow the approach taken by Smith in functional programming [30].

The rest of this paper is organised as follows. In Section 2, we define the concept of frameworks, namely as first-order axiomatisations of the considered problem domain. We distinguish between open and closed frameworks, depending on whether they are parameterised (on sorts, functions, and/or relations) or not. Specifications and programs are expressed in frameworks. Then, in Section 3, we first distinguish between open and closed programs, depending on whether they feature no or some undefined relations, and then define what it means for a closed program to be correct wrt a specification, in a given framework of course, as well as what it means for an open program to be steadfast wrt a specification, in a given framework. A steadfast program is parametrically correct, that is it is correctly reusable no matter how the parameter sorts, functions, and relations are instantiated. We can then define, in Section 4, program schemas as steadfast (first-order) programs, in an arbitrary framework. Next, in Section 5, we illustrate all definitions and concepts seen that far by means of a schema capturing the divide-and-conquer programming methodology, and show under what abstract conditions it is steadfast wrt an arbitrary specification in an arbitrary framework. Schemas can be used to effectively guide the synthesis of steadfast programs, as argued in Section 6, where we also develop a strategy for synthesising divide-and-conquer programs and illustrate it by synthesising the well-known Quicksort program. Finally, in Section 7, we conclude, examine related work, and line out future work.

2 Frameworks and Specifications

Our approach to 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 *framework* $\mathcal{F}$. Specifications are given in $\mathcal{F}$, i.e. written in the language of $\mathcal{F}$. We adopt a model-theoretic semantics for $\mathcal{F}$, and for specifications and programs[1] in $\mathcal{F}$. This declarative approach enables us to define program correctness wrt specifications not only for closed programs but also for open programs i.e. programs with parameters, in both closed and open frameworks. In this section, we briefly define frameworks and specifications and their model-theoretic semantics.

A framework $\mathcal{F}$ is a full first-order logical theory (with identity) with an intended model. The syntax of $\mathcal{F}$ is similar to that used in algebraic abstract data types (e.g. [13, 34, 28]). 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.

Definition 2.1 A model i^* is an *isoinitial model* of $\mathcal{F}$ iff, for every other model i of $\mathcal{F}$ there is a unique isomorphic embedding $h : \mathsf{i}^* \rightarrow \mathsf{i}$.

An isomorphic embedding $h : \mathsf{i}^* \rightarrow \mathsf{i}$ is a homomorphism with the additional property of preserving negation, i.e. for every relation symbol r, including identity, $(\alpha_1, \ldots, \alpha_n) \notin r^{\mathsf{i}^*}$ entails $(h(\alpha_1), \ldots, h(\alpha_n)) \notin r^{\mathsf{i}}$, where r^{i^*} and r^{i} are the interpretations of r in i^* and i respectively.

Less formally, if $\mathcal{F}$ has a *reachable* model, i.e. one where each element (of the domain) can be represented by a ground term, then:

> an *isoinitial model* i of $\mathcal{F}$ is a reachable model such that for any relation r defined in $\mathcal{F}$, ground instances $r(t)$ or $\neg r(t)$ are true in i iff they are true in all models of $\mathcal{F}$.

Such a (reachable) model is also an initial model:

Definition 2.2 An *initial model* j of $\mathcal{F}$ is a reachable model such that for any relation r defined in $\mathcal{F}$, ground instances $r(t)$ are true in j iff they are true in all models of $\mathcal{F}$.

Both initial and isoinitial theories enjoy the so-called 'no junk' and 'no confusion' properties [12]. 'No junk' means that the (initial or isoinitial) model is reachable (by ground terms), and 'no confusion' means that two ground terms of the domain of the model are identical iff they are equal according to the axioms. However, isoinitial theories handle negation properly, whereas initial theories can only do so via so-called 'final models'. Negation is an important property in reasoning about specifications and program correctness in general.

[1] That is, normal logic programs.

2.1 Closed and Open Frameworks

We distinguish between *closed* and *open* frameworks, depending on the absence or presence of *parameters*.

Definition 2.3 (Closed Frameworks)
A *closed framework* consists of:

- a *defined* (many-sorted) signature of
  - sort symbols;
  - function declarations, for declaring constant and function symbols;
  - relation declarations, for declaring relation symbols;
- a set of first-order axioms for the declared function and relation symbols, possibly containing induction schemas;
- a set of theorems, i.e. proven properties of the problem domain.

In general, a closed framework $\mathcal{F}$ typically completely defines a new abstract data type T. The (new) sort T is constructed from *constructors* declared as functions (though not labelled as such). Axioms are added to define the (new) functions and relations on T.

Example 2.1 (Closed Frameworks)
A typical closed framework is (first-order) Peano arithmetic $\mathcal{NAT}$: [2]

$$
\begin{array}{lllll}
\multicolumn{5}{l}{\textbf{Framework } \mathcal{NAT};} \\
\text{SORTS:} & Nat; & & & \\
\text{FUNCTIONS:} & 0 & : & \rightarrow Nat; & \\
 & s & : & Nat \rightarrow Nat; & \\
 & +, * & : & (Nat, Nat) \rightarrow Nat; & \\
\text{AXIOMS:} & \multicolumn{4}{l}{\neg 0 = s(x) \wedge s(a) = s(b) \rightarrow a = b;} \\
 & \multicolumn{4}{l}{x + 0 = x;} \\
 & \multicolumn{4}{l}{x + s(y) = s(x + y);} \\
 & \multicolumn{4}{l}{x * 0 = 0;} \\
 & \multicolumn{4}{l}{x * s(y) = x + x * y;} \\
 & \multicolumn{4}{l}{H(0) \wedge (\forall i \,.\, H(i) \rightarrow H(s(i)) \rightarrow \forall x \,.\, H(x).}
\end{array}
$$

This framework defines the abstract data type $\mathcal{NAT}$ as follows: the sort Nat of natural numbers is constructed *freely* from the constructors 0 (*zero*) and s (*successor*); the *freeness axiom* for these constructors is the first axiom; the functions $+$ (*sum*) and $*$ (*product*) on Nat are axiomatised by the next four axioms (in a primitive recursive manner).

It can be shown that an isoinitial model of $\mathcal{NAT}$ is the structure of natural numbers thus generated, i.e. the term model generated by the constructors 0 and s.

Note in particular that the last axiom in $\mathcal{NAT}$ is an induction schema. This 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.

Definition 2.4 (Open Frameworks)
An *open framework* consists of:

- a (many-sorted) signature of
  - both *defined* and *open* sort symbols;
  - function declarations, for declaring both *defined* and *open* constant and function symbols;
  - relation declarations, for declaring both *defined* and *open* relation symbols;

[2] We will omit the most external universal quantifiers.

- a set of first-order axioms each for the (declared) *defined* and *open* function and relation symbols, the former possibly containing induction schemas;
- a set of theorems.

Example 2.2 (Open Frameworks)
The following open framework axiomatises the (kernel of the) theory of lists with parametric element sort $Elem$ and partial ordering relation $\lhd$: [3]

Framework $\mathcal{LIST}(Elem, \lhd)$;

$$
\begin{array}{lll}
\text{IMPORT:} & \mathcal{NAT}; & \\
\text{SORTS:} & Nat, Elem, List;^4 & \\
\text{FUNCTIONS:} & nil & : \ \to List; \\
 & . & : \ (Elem, List) \to List; \\
 & nocc & : \ (Elem, List) \to Nat; \\
 & l & : \ List \to Nat; \\
 & | & : \ (List, List) \to List; \\
\text{RELATIONS:} & elemi & : \ (List, Nat, Elem); \\
 & \lhd & : \ (Elem, Elem); \\
 & mem & : \ (Elem, List); \\
 & len & : \ (List, Nat); \\
 & append & : \ (List, List, List); \\
 & perm & : \ (List, List); \\
 & ord & : \ (List); \\
\end{array}
$$

$$
\begin{array}{ll}
\text{AXIOMS:} & \neg nil = a.B \land (a_1.B_1 = a_2.B_2 \to a_1 = a_2 \land B_1 = B_2)); \\
 & H(nil) \land (\forall a, J \,.\, H(J) \to H(a.J)) \to \forall L \,.\, H(L); \\
 & nocc(x, nil) = 0; \\
 & a = b \to nocc(a, b.L) = nocc(a, L) + 1; \\
 & \neg a = b \to nocc(a, b.L) = nocc(a, L); \\
 & elemi(L, 0, a) \leftrightarrow \exists B \,.\, L = a.B; \\
 & elemi(L, s(i), a) \leftrightarrow \exists b, B \,.\, L = b.B \land elemi(B, i, a); \\
 & mem(e, L) \leftrightarrow \exists i \,.\, elemi(L, i, e); \\
 & len(L, n) \leftrightarrow \forall i \,.\, i < n \leftrightarrow \exists a \,.\, elemi(L, i, a); \\
 & n = l(L) \leftrightarrow len(L, n); \\
 & append(A, B, L) \leftrightarrow \ (\forall i, a \,.\, i < l(A) \to \\
 & \qquad\qquad (elemi(A, i, a) \leftrightarrow elemi(L, i, a))) \land \\
 & \qquad\qquad (\forall j, b \,.\, elemi(B, j, b) \leftrightarrow \\
 & \qquad\qquad elemi(L, j + l(A), b)); \\
 & perm(A, B) \leftrightarrow \forall e \,.\, nocc(e, A) = nocc(e, B); \\
 & C = A|B \leftrightarrow append(A, B, C); \\
 & ord(L) \leftrightarrow \forall i \,.\, elemi(L, i, e_1) \land elemi(L, s(i), e_2) \to e_1 \lhd e_2; \\
\text{P-AXIOMS:} & x \lhd y \land y \lhd x \leftrightarrow x = y; \\
 & x \lhd y \land y \lhd z \to x \lhd z. \\
\end{array}
$$

where the function $nocc(a, L)$ gives the number of occurrences of a in L, l and $|$ are the usual functions for length and catenation, $elemi(L, i, a)$ means a occurs at position i in L, and mem, len, $append$, $perm$, and ord are the usual 'membership', 'length', 'concatenation', 'permutation', and 'ordered' relations.

$\mathcal{LIST}(Elem, \lhd)$ imports the (closed) framework $\mathcal{NAT}$. The first axiom is the freeness axiom for the constructors $\cdot$ and nil. The second axiom is an induction schema for reasoning about all such predicates. The p-axioms are the parameter axioms for $\lhd$. In this case, they state that $\lhd$ must be a (strict) partial ordering.

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

[3] We shall write $\lhd$ in infix notation for clarity.
[4] Strictly speaking, we should write $List(Elem, \lhd)$, but to save space we simply write $List$.

Example 2.3 Consider the open framework $\mathcal{LIST}(Elem, \lhd)$. For every interpretation of the open sort $Elem$ and the ordering $\lhd$, we get a corresponding intended model of $\mathcal{LIST}(Elem, \lhd)$. For example, suppose $\mathcal{INT}$ is a closed framework axiomatising the set Int of integers with ordering $<$. Then $\mathcal{LIST}(Int, <)$ automatically imports $\mathcal{INT}$ and 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.

The class of intended models of an open framework can also be defined formally under isoinitial semantics, in a parametric manner. For simplicity, however, we shall not do so here, and instead we will consider an open framework just as a pair $\langle \Sigma, \mathcal{C} \rangle$, where Σ is the signature, and $\mathcal{C}$ is the class of intended interpretations. Note that a closed framework is a pair $\langle \Sigma, \mathcal{C} \rangle$ where $\mathcal{C}$ is a class of isomorphic interpretations.

Notation and Convention. We will also denote an open framework $\mathcal{F}$ as $\mathcal{F}(\Pi)$, where Π are the open symbols, or parameters, of $\mathcal{F}$. In the sequel, all frameworks will be considered open, as a closed framework is just an extreme case of an open one, namely where Π is empty.

Also, for simplicity, in definitions — but not necessarily in examples — we will restrict ourselves to binary relations.

2.2 Specifications

In a framework a specification S introduces new symbols by means of a set of axioms:

Definition 2.5 (Specifications)
A *specification* S of a new symbol s in a framework $\mathcal{F} = \langle \Sigma, \mathcal{C} \rangle$ is a $(\Sigma + s)$-axiom.[5]

Thus, from a semantic point of view, S is an expansion operator:

Definition 2.6 Let j be a Σ-interpretation, and i be an expansion of j to $\Sigma + s$. We say that i is an *expansion* of j determined by a specification S (of s) iff $\mathsf{i} \models S$.

S determines (one or more) interpretations of the specified symbol s, in terms of the old ones, by determining expansions of the intended interpretations of $\mathcal{F}$, i.e. Σ-interpretations.

We distinguish between specifications that determine only one, and those that determine more than one, interpretation of the specified symbols:

Definition 2.7 (Strict and Non-strict Specifications)
Let $\mathcal{F} = \langle \Sigma, \mathcal{C} \rangle$ be a framework, and S be a specification of a new symbol s.

S is said to be *strict* in $\mathcal{F}$ if, for every $\mathsf{j} \in \mathcal{C}$, it determines only one expansion of j, i.e. only one interpretation of s.

S is *non-strict* (or *loose*) in $\mathcal{F}$ if, for every $\mathsf{j} \in \mathcal{C}$, it determines more than one expansion of j, i.e. more than one interpretation of s.

A specification S in a framework $\mathcal{F}$ thus expands $\mathcal{F}$. We distinguish between *adequate* and *inadequate* expansions:

Definition 2.8 A closed framework $\mathcal{G}$ is an *adequate expansion* of a closed framework $\mathcal{F}$ if the signature and axioms of $\mathcal{G}$ contain those of $\mathcal{F}$, and the isoinitial model of $\mathcal{G}$ is an expansion[6] of that of $\mathcal{F}$.

Thus, adequate expansions of a framework $\mathcal{F}$ expand $\mathcal{F}$ by introducing new symbols and axioms, while preserving the intended models of $\mathcal{F}$. They therefore provide a means of constructing frameworks incrementally.

Non-strict specifications give rise to inadequate framework expansions. Symbols defined by such specifications may have many interpretations and thus destroy the existence of an isoinitial model for the expanded framework. However, non-strict specifications are very useful for program specification, since they enable us to avoid unnecessary details. We shall therefore use non-strict specifications only for program specifications, i.e. to introduce relations that are to be computed by programs. For this

[5] $\Sigma + s$ denotes the signature containing Σ and the new symbol s.
[6] An expansion of a model i, to a larger signature, is any model for the new signature that coincides with i for the old one.

purpose, they do not have to be adequate, although they must have a precise meaning in the isoinitial model of the framework.

Strict specifications on the other hand can be used to expand and build up the framework by adding new framework symbols. However, for this purpose they must be adequate.

Thus all our framework axioms are strict specifications, whilst all our non-strict specifications are program specifications. However, it should be noted that strict specifications can also be used to specify programs.

Convention. For uniformity, we shall only use 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 and Y are sorts of $\mathcal{F}$. Q is called the *input condition*, whereas R is called the *output condition* of the specification and may or may not contain the relation r.[7] When Q is true, then we drop it and speak of an *if-and-only-if* (*iff*) *specification*; otherwise, we speak of a *conditional specification*. Note that the former is strict, whilst the latter is non-strict.

In the sequel, we often drop the universal quantifications at the beginning of specifications. Also, all specifications will be considered conditional, as iff specifications are just an extreme case of conditional ones, namely where Q is true.

Example 2.4 (Specifications)
Let us give a specification, in the framework $\mathcal{LIST}(Elem, \lhd)$ introduced above, of the relation *sort*, which is informally specified as follows:

> $sort(L, S)$ iff S is an ordered (under strict partial ordering $\lhd$) permutation of L, where L and S are *Elem* lists.

Formally, $sort(L,S)$ can now be specified as follows:

$$sort(L,S) \leftrightarrow perm(L,S) \wedge ord(S)$$

(*perm* and *ord* are already defined in the framework.)

3 Correctness of Open Programs

Open programs arise in both closed and open frameworks. In a closed framework, the parameters of an open program may be relation symbols that are computed by other programs.

Definition 3.1 (Defined and Open Predicates)
In a framework $\mathcal{F} = \langle \Sigma, \mathcal{C} \rangle$, let P be a Σ-program, i.e. a normal logic program whose signature is a subsignature of Σ.

A predicate in P is *defined* (by P) if and only if it occurs in the head of at least one clause of P.

A predicate in P is *open* if it is not defined (by P). An open predicate in P is also called a *parameter* of P.

Notation.
In a signature Σ, we will write

$$P : \delta \Leftarrow \pi$$

for a Σ-program P with defined predicates δ, where π is the subsignature of Σ that does not contain δ. Thus the parameters of P are the set $\Sigma \setminus \pi$ of symbols.

For a program $P : \delta \Leftarrow \pi$, the meaning of π is considered to be pre-defined:

Definition 3.2 (Pre-interpretations)
Let $P : \delta \Leftarrow \pi$ be a Σ-program.

A π-interpretation will be called a *pre-interpretation* of P.

[7] For more on this issue, see the discussion in [22].

This definition of pre-interpretation is an extension of that in [25]. The two definitions become equivalent when P does not contain open predicates, and the signature of P coincides with Σ.

Definition 3.3 (Open and Closed Programs)
Let $P : \delta \Leftarrow \pi$ be a Σ-program.

- If P has at least one open predicate, then P is *open*.
- If P has no open predicates, then:
  - P is *closed* in a pre-interpretation j if (and only if) the Herbrand base generated by π is isomorphic to (a suitable restriction of) j;
  - P is *open* in a pre-interpretation j otherwise.

In the sequel, all programs will be considered open, as a closed program is just an extreme case of an open one, namely one without any parameters.

Example 3.1 (Open Programs)
A possible open program for $sort(L, S)$ in $\mathcal{LIST}(Elem, \lhd)$ is the following:

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

A model-theoretic definition of correctness of open programs, called *steadfastness*, is given in [21]. Here, we give an equivalent definition in proof-theoretic terms, which will turn out to be more "constructive" for our purposes (see Section 5.2).

Depending on whether a program is closed or open, we have two notions of correctness. For closed programs, we have the classical notion of (total) correctness:

Definition 3.4 (Total Correctness)
In a framework $\mathcal{F}(\Pi)$, a closed program P_r for relation r is (*totally*) *correct* wrt its specification

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

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

$$\mathcal{F} \cup \{r(x,y) \leftrightarrow O_r(x,y)\} \models r(t,u) \text{ iff } \mathcal{F} \cup P_r \vdash r(t,u) \tag{1}$$

Total correctness is the conjunction of *partial correctness* ('iff' replaced by 'if' in the above) and *totality* ('iff' replaced by 'implies').

In other words, and as intended, under input condition I_r, the program P_r is equivalent, in $\mathcal{F}$, to $r(x,y) \leftrightarrow O_r(x,y)$ for queries on r.

Note that (1) is equivalent to

$$\mathcal{F} \models O_r(t,u) \text{ iff } \mathcal{F} \cup P_r \vdash r(t,u) \tag{2}$$

and in the sequel this will play a crucial role in our correctness proofs.

This kind of correctness is not entirely satisfactory, for two reasons. First, it defines the correctness of P_r in terms of the programs for the relations in its body, 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 relations in its body is wasted. Thus, for open programs, we must bring in the specifications of at least their open relations, whereas for closed programs, it is preferable to bring in the specifications of at least some of their defined relations.

This leads us to the notion of steadfastness, which we only define here for the most interesting case, namely where all relations occurring in the body are also known by their specifications, whether they are open relations or the defined relation. Again, we do not give here a model-theoretic definition of steadfastness as in [21], but rather a proof-theoretic definition that will turn out to be more "constructive" for our later purposes (see Section 5.2).

Definition 3.5 (Steadfastness)
In a framework $\mathcal{F}(\Pi)$, an open program P_r for relation r (with parameters $p_1, \ldots, p_n$) is *steadfast* wrt a specification S_r of r and a set $\{S_1, \ldots, S_n\}$ of specifications of $p_1, \ldots, p_n$ iff, for any closed programs $P_1, \ldots, P_n$ that are correct wrt $S_1, \ldots, S_n$, respectively, we have that the (closed) program $P_r \cup P_1 \cup \ldots \cup P_n$ is correct wrt S_r in $\mathcal{F}(\Pi)$.

This is similar to Deville's notion of 'correctness in a set of specifications' [3, p.76], except that specifications and programs are not set within frameworks there. Moreover, we also (but not in this article, 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.

Definition 3.6 (Steadfast program)
In a framework $\mathcal{F}(\Pi)$, a *steadfast program* for a relation r consists of an open program P_r for r (with parameters $p_1, \ldots, p_n$) and a set $\{S_r, S_1, \ldots, S_n\}$ of specifications of $r, p_1, \ldots, p_n$, such that P_r is steadfast wrt S_r and $\{S_1, \ldots, S_n\}$.

This definition will be crucial in the rest of this paper, because we can now define program schemas as steadfast programs.

4 Program Schemas as Steadfast Programs

Program schemas are a popular element of the programming folklore: 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 nor (all) their actual data structures. One could for instance design a program schema capturing the class of divide-and-conquer programs, or a sub-class thereof (e.g., those featuring an input parameter of type list, and division of that list into two shorter lists).

Program schemas have been shown useful in a number of applications, such as proving properties of programs [26], teaching programming to novices [9], guiding the manual construction of programs [2, 32, 27], debugging programs [10], transforming programs [8, 11, 33], and guiding the (semi-)automatic synthesis [4] of programs, be it deductive synthesis [1, 16, 18, 19, 30, 31], constructive synthesis [nobody so far], or inductive synthesis [5, 7, 14]. For more details and more exhaustive references to related work, please refer to [6].

For representing schemas, there are essentially two approaches, the choice of any depending on the targeted manipulations of schemas.

First, most cited 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 [8], where some form of higher-order matching between actual programs and schemas is convenient to establish applicability of the start schema of a schematic transformation.

Second, Manna [26] 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 can also be represented as a (first-order) open program (in a possibly open framework, which is a class of interpretations), so that actual programs can be obtained by adding programs for some (but not necessarily all) of its open relations. So there is no need to invent a new (or higher-order) schema language, at least in a first approximation (but see Section 5.1 below).

We say that a schema *covers* a program iff it can be extended into that program, and that the program is an *instance* of that schema. In order to also consider a notion of correctness of a schema, we have to add to a schema the specifications of its open relations. This leads to the following definition:

Definition 4.1 (Program schema)
In a framework $\mathcal{F}(\Pi)$, a (*program*) *schema* is a steadfast program, whose open program is called the *template* of the schema, and whose specifications are called the *constraints* of the schema.

Most definitions of schemas, with the laudable exception of the one by Smith [30, 31], reduce this concept to what we here call the template. Such definitions are thus merely syntactic, providing only a pattern of place-holders, but they have no concerns 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 the system or person using the template. Despite the similarity, our definition even is an enhancement of 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 schemas as steadfast programs, rather than as entities different from programs.

5 Example: A Divide-and-Conquer Schema

We now illustrate all definitions and concepts seen so far by means of a schema capturing the divide-and-conquer programming methodology. First, in Section 5.1, we construct a divide-and-conquer template (i.e., open program) from that methodology. Then, in Section 5.2, we abduce the constraints (on the open relations) under which this template is a steadfast program wrt its specification.

5.1 A Divide-and-Conquer Template

A sub-class of the well-known class of divide-and-conquer programs can be captured by the following (open) program, or template:

$$\begin{array}{rcl} r(x,y) & \leftarrow & primitive(x), solve(x,y) \\ r(x,y) & \leftarrow & nonPrimitive(x), decompose(x, hx, tx_1, tx_2), \\ & & r(tx_1, ty_1), r(tx_2, ty_2), compose(hx, ty_1, ty_2, y) \end{array}$$

By itself, such an open program has no meaning, as it can be extended without necessarily obtaining a divide-and-conquer program. Taken to its extreme, in the absence of constraints, this divide-and-conquer template covers *every* program, which is obviously not what was wanted. Indeed, it would suffice to instantiate *primitive* by *true*, *nonPrimitive* by *false*, and *solve* by the given program (the instantiations of all other place-holders being arbitrary)! But 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 the *primitive* test holds) for which y can be easily directly computed (through *solve*), or x has a value (when the *nonPrimitive* test holds) for which y cannot be so easily directly computed.[8] In the latter case, the divide-and-conquer principle is applied by:

1. division (through *decompose*) of 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) in order to determine values of ty_1 and ty_2 corresponding to tx_1 and tx_2, respectively, and

3. combining (through *compose*) terms hx, ty_1, ty_2 in order to build y.

[8] Note that both cases may apply, as there may be values of y that it is easy to directly compute from a given x, as well as other values of y that it is not so easy to directly compute from that x. The classical program for *member* illustrates such non-determinism.

As, in general, the semantics of open programs is defined parametrically, we can also do so for this template. While doing this (in the next sub-section), we enforce the informal semantics above and supply the corresponding axioms (here called constraints) of the open relations.

Note that nothing, neither here nor elsewhere in this paper, prejudices the number of "heads" hx of x to be 1, or the number of "tails" tx_i of x to be 2 (i.e. the number of recursive calls to be 2). We have just chosen this version of the schema for illustration purposes, but nothing prevents particularization to other (fixed) numbers of heads and tails, nor parameterization to arbitrary numbers h of heads and t of tails. Also, the words "head" and "tail" should not be taken literally, as a head of a list (for instance) may well be its central element, if not a prefix, and a tail may well also be a prefix. Finally, this template is restricted to binary relations, so a parameterization to n-ary relations (possibly with *passive parameters*, which don't change through recursion) would thus also be interesting. See the proposal in [6] for more details on this.

For instance, in the $\mathcal{LIST}(Elem, \lhd)$ framework above, if $r(x, y)$ is replaced by $sort(L, S)$, then the open program above can be extended into a program by addition of the following clauses:

$$
\begin{array}{rcl}
primitive(L) & \leftarrow & L = nil \\
nonPrimitive(L) & \leftarrow & L = h.T \\
solve(L, S) & \leftarrow & S = nil \\
decompose(L, h, T_1, T_2) & \leftarrow & L = h.T, partition(T, h, T_1, T_2) \\
partition(L, p, S, B) & \leftarrow & L = nil, S = nil, B = nil \\
partition(L, p, S, B) & \leftarrow & L = h.T, h \lhd p, partition(T, p, TS, TB), S = h.TS, B = TB \\
partition(L, p, S, B) & \leftarrow & L = h.T, \neg h \lhd p, partition(T, p, TS, TB), S = TS, B = h.TB \\
compose(e, L_1, L_2, R) & \leftarrow & append(L_1, e.L_2, R)
\end{array}
$$

(All added clauses, except the ones for *partition*, can actually be unfolded, as they are non-recursive.) This is the classical Quicksort program, but it is still open as there is no program yet for deciding $\lhd$ nor *append*. However, the steadfastness of the overall program can be verified, as $\lhd$ is constrained by the p-axioms, and *append* is constrained by the (regular) axioms of $\mathcal{LIST}(Elem, \lhd)$.

Note that templates are thus composition operators, in the sense that they show how to compose individual programs into larger programs. As this is not a mere juxtaposition, this is a first step towards going beyond programming-in-the-small.

Also, the schemas mentioned here are design schemas (capturing a class of programs). Since we do not discuss *transformation schemas* (directed pairs of design schemas capturing a transformation process [8, 11, 33]) here, we will from now on simply talk about schemas.

5.2 Steadfastness of a Divide-and-Conquer Template

As we have observed earlier, the divide-and-conquer template above does not have a (formal) semantics by itself, so it is up to us to enforce that its extensions actually are programs of the divide-and-conquer class. This enforcement should result in the supply of axioms (here called constraints) on the open relations of the template. Also, as the main objective of schemas is the ability to pre-compile as much as possible of the manipulations on the programs covered by a template, it would be preferable to pre-compile this enforcement as much as possible.

We can do so by "proving," at an abstract level, that a template for an arbitrary relation r in an arbitrary framework is steadfast wrt the specification 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 reveal (or: abduce) the necessary relationships between r and the introduced open relations. These relationships, or axioms, are the *constraints* on the open relations of the template!

Let us illustrate these ideas on the divide-and-conquer template above, but simplified as follows for convenience (i.e., where $nonPrimitive(x) \leftrightarrow \neg primitive(x)$):

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

The simplification means that we do not cover some non-deterministic programs, namely those where $nonPrimitive(x)$ is not $\neg primitive(x)$.

Suppose the specification of r, in a framework $\mathcal{F}(\Pi)$, is:

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

The objective is to find specifications (in $\mathcal{F}$) S_{prim}, S_{solve}, S_{dec}, S_{comp} of *primitive*, *solve*, *decompose*, *compose*, respectively, such that the open program P_r is steadfast wrt S_r and $\{S_{prim}, S_{solve}, S_{dec}, S_{comp}\}$. To do so, we must apply the definition of steadfastness, but we will also manually enforce the informal semantics of the considered template.

By the definition of steadfastness, it suffices to show, by structural induction on X using some well-founded order $\prec$, that for all $t : \mathsf{X}$ and $u : \mathsf{Y}$ such that $I_r(t)$ we have:

$$\mathcal{F} \models O_r(t,u) \text{ iff } \mathcal{F} \cup P_r' \vdash r(t,u),$$

where P_r' is the union of P_r and any programs P_{prim}, P_{solve}, P_{dec}, P_{comp} that are correct wrt the yet unknown S_{prim}, S_{solve}, S_{dec}, S_{comp}, respectively, which are thus to be revealed by this proof.

The induction hypothesis is that for any $v : \mathsf{X}$ and $w : \mathsf{Y}$ such that $I_r(v)$ and $v \prec t$, we have:

$$\mathcal{F} \models O_r(v,w) \text{ iff } \mathcal{F} \cup P_r' \vdash r(v,w).$$

Let us first establish *partial correctness* of P_r' wrt S_r. Hypothesise thus that $I_r(t)$ and $\mathcal{F} \cup P_r' \vdash r(t,u)$ hold, for some arbitrary $t : \mathsf{X}$ and $u : \mathsf{Y}$. By the clauses in P_r' for r, there are two cases to consider, according to whether $primitive(t)$ holds or not.

1. Assume $primitive(t)$ holds. Then, by the hypothesis $\mathcal{F} \cup P_r' \vdash r(t,u)$ and by P_r, we have that $solve(t,u)$ holds. We are blocked now. But we can unblock the situation by postulating (or: abducing) the following two constraints:

   (a) The sub-program P_{prim} of P_r' is partially correct wrt the specification

   $$\forall x : \mathsf{X} \,.\, primitive(x) \leftrightarrow O_{prim}(x). \tag{S_{prim}}$$

   (b) The sub-program P_{solve} of P_r' is partially correct wrt the specification

   $$\forall x : \mathsf{X}, \forall y : \mathsf{Y} \,.\, I_r(x) \wedge O_{prim}(x) \rightarrow (solve(x,y) \leftrightarrow O_r(x,y)). \tag{S_{solve}}$$

   Now, by constraint (a) and the assumption $primitive(t)$, we have that $\mathcal{F} \models O_{prim}(t)$ holds.

   So, by constraint (b), the hypothesis $I_r(t)$, the just inferred $O_{prim}(t)$, and the previously inferred $solve(t,u)$, we have that $\mathcal{F} \models O_r(t,u)$ holds.

2. Now assume $\neg primitive(t)$ holds. Then, by the hypothesis $\mathcal{F} \cup P_r' \vdash r(t,u)$ and by P_r, we have that $decompose(t, ht, tt_1, tt_2)$, $r(tt_1, tu_1)$, $r(tt_2, tu_2)$, and $compose(ht, tu_1, tu_2, u)$ hold, for some $ht : \mathsf{H}$, some $tt_1, tt_2 : \mathsf{X}$, and some $tu_1, tu_2 : \mathsf{Y}$, where H is a (possibly new) sort (possibly, but not necessarily, designating the sort of the 'elements' of inductively defined sort X). We are blocked again. But we can unblock the situation by postulating (or: abducing) constraint (a) again, as well as the following two new constraints:

   (c) The sub-program P_{dec} of P_r' is partially correct wrt the specification

   $$\begin{aligned}&\forall x, tx_1, tx_2 : \mathsf{X}, \forall hx : \mathsf{H} \,.\, \neg O_{prim}(x) \rightarrow \\ &(decompose(x, hx, tx_1, tx_2) \leftrightarrow Dec(x, hx, tx_1, tx_2) \wedge I_r(tx_1) \wedge I_r(tx_2) \wedge tx_1 \prec x \wedge tx_2 \prec x).\end{aligned} \tag{S_{dec}}$$

   (d) The sub-program P_{comp} of P_r' is partially correct wrt the specification

   $$\begin{aligned}&\forall hx : \mathsf{H}, \forall ty_1, ty_2, y : \mathsf{Y}, \forall x, tx_1, tx_2 : \mathsf{X} \,.\, O_{dec}(x, hx, tx_1, tx_2) \wedge O_r(tx_1, ty_1) \wedge O_r(tx_2, ty_2) \rightarrow \\ &(compose(hx, ty_1, ty_2, y) \leftrightarrow O_r(x,y)).\end{aligned} \tag{S_{comp}}$$

   Note that the new sort H is shared by these specifications, but otherwise unspecified. In S_{comp}, and in the following, we refer to the output condition of *decompose* by $O_{dec}(x, hx, tx_1, tx_2)$.

   Now, by constraint (a) and the assumption $\neg primitive(t)$, we have that $\mathcal{F} \models \neg O_{prim}(t)$ holds.

Then, by constraint (c) and the inferred $\neg O_{prim}(t)$ and $decompose(t, ht, tt_1, tt_2)$, we have that $\mathcal{F} \models O_{dec}(t, ht, tt_1, tt_2)$ holds.

Next, by applying the (if part of the) induction hypothesis to the previously inferred $r(tt_1, tu_1)$ and $r(tt_2, tu_2)$, we have that $\mathcal{F} \models O_r(tt_1, tu_1$ and $\mathcal{F} \models O_r(tt_2, tu_2)$ hold, because $I_r(tt_1)$, $I_r(tt_2)$, $tt_1 \prec t$, and $tt_2 \prec t$ hold, as just inferred (as parts of O_{dec}).

So, by constraint (d), the inferred $\mathcal{F} \models Dec(t, ht, tt_1, tt_2)$, the just inferred $\mathcal{F} \models O_r(tt_1, tu_1)$ and $\mathcal{F} \models O_r(tt_2, tu_2)$, and the previously inferred $compose(ht, tu_1, tu_2, u)$, we have that $\mathcal{F} \models O_r(t, u)$ holds.

So, in both cases, we infer the desired $\mathcal{F} \models O_r(t, u)$, establishing thus that P'_r is partially correct wrt S_r.

Let us now establish *totality* of P'_r wrt S_r. Hypothesise thus that $I_r(t)$ and $\mathcal{F} \models O_r(t, u)$ hold, for some arbitrary $t : \mathsf{X}$ and $u : \mathsf{Y}$. There are again two cases to consider, according to whether $O_{prim}(t)$ holds or $\neg O_{prim}(t)$ holds.

1. Assume $O_{prim}(t)$ holds. We are blocked now. But we can unblock the situation by postulating (or: abducing) the following two constraints:

   (e) The sub-program P_{prim} of P'_r is total wrt S_{prim}.

   (f) The sub-program P_{solve} of P'_r is total wrt S_{solve}.

   Now, by constraint (e) and the assumption $O_{prim}(t)$, we infer that $\mathcal{F} \cup P_{prim} \vdash primitive(t)$ holds, i.e. that $\mathcal{F} \cup P'_r \vdash primitive(t)$ holds.

   Also, by constraint (f), the hypothesis $I_r(t)$, the hypothesis $\mathcal{F} \models O_r(t, u)$, and the assumption $O_{prim}(t)$, we have that $\mathcal{F} \cup P_{solve} \vdash solve(t, u)$ holds, i.e. that $\mathcal{F} \cup P'_r \vdash solve(t, u)$ holds.

   So, $\mathcal{F} \cup P'_r \vdash primitive(t), solve(t, u)$ holds, and, by modus ponens, $\mathcal{F} \cup P'_r \vdash r(t, u)$ holds.

2. Now assume $\neg O_{prim}(t)$ holds. Then, in order to be able to reason about the totality of P'_r, we also have to assume that *compose* is defined, i.e. that $O_{dec}(t, ht, tt_1, tt_2)$, $O_r(tt_1, tu_1)$, and $O_r(tt_2, tu_2)$ hold, for some $ht : \mathsf{H}$, some $tt_1, tt_2 : \mathsf{X}$, and some $tu_1, tu_2 : \mathsf{Y}$. We are blocked now. But we can unblock the situation by postulating (or: abducing) constraint (e) again, as well as the following two new constraints:

   (g) The sub-program P_{dec} of P'_r is total wrt S_{dec}.

   (h) The sub-program P_{comp} of P'_r is total wrt S_{comp}.

   Now, by constraint (e) and the assumption $\neg O_{prim}(t)$, we have that $\mathcal{F} \cup P_{prim} \vdash \neg primitive(t)$ holds, i.e. that $\mathcal{F} \cup P'_r \vdash \neg primitive(t)$ holds.

   Also, by constraint (g), the assumption $\neg O_{prim}(t)$, and the assumption $O_{dec}(t, ht, tt_1, tt_2)$, we have that $\mathcal{F} \cup P_{dec} \vdash decompose(t, ht, tt_1, tt_2)$, i.e. that $\mathcal{F} \cup P'_r \vdash decompose(t, ht, tt_1, tt_2)$ holds.

   Next, by applying the (implies part of the) induction hypothesis to the assumptions $O_r(tt_1, tu_1)$ and $O_r(tt_2, tu_2)$, we have that $\mathcal{F} \cup P_r \vdash r(tt_1, tu_1)$ and $\mathcal{F} \cup P_r \vdash r(tt_2, tu_2)$ hold, i.e. that $\mathcal{F} \cup P'_r \vdash r(tt_1, tu_1)$ and $\mathcal{F} \cup P'_r \vdash r(tt_2, tu_2)$ hold, because $I_r(tt_1)$, $I_r(tt_2)$, $tt_1 \prec t$, and $tt_2 \prec t$ hold, as they are parts of the assumption $O_{dec}(t, ht, tt_1, tt_2)$.

   And, by constraint (h), the hypothesis $O_r(t, u)$, and the assumptions $O_r(tt_1, tu_1)$ and $O_r(tt_2, tu_2)$, we have that $\mathcal{F} \cup P_{comp} \vdash compose(ht, tu_1, tu_2, u)$ holds, i.e. that $\mathcal{F} \cup P'_r \vdash compose(ht, tu_1, tu_2, u)$ holds.

   So, $\mathcal{F} \cup P'_r \vdash \neg primitive(t), decompose(t, ht, tt_1, tt_2), r(tt_1, tu_1), r(tt_2, tu_2), compose(ht, tu_1, tu_2, u)$ holds, and, by modus ponens, $\mathcal{F} \cup P'_r \vdash r(t, u)$ holds.

So, in both cases, we infer the desired $\mathcal{F} \cup P'_r \vdash r(t, u)$, establishing thus that P'_r is total wrt S_r.

We can thus now propose the following theorem:

Theorem 5.1 (Steadfastness of the divide-and-conquer schema)
In a framework $\mathcal{F}(\Pi)$, given a well-founded order $\prec$ on its sort X, the open program

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

is steadfast wrt the specification

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

and the specification set $\{S_{prim}, S_{solve}, S_{dec}, S_{comp}\}$ (as in the preceding pages).

Proof. Directly follows from the abduction process above. □

Note that this theorem is related to the one given by Smith [30] for a divide-and-conquer schema in the functional programming setting. The innovations here are that we use specification frameworks, that we thus can also consider open programs, and that we prove total correctness (and not just partial correctness), because we are in a relational setting. Moreover, we could eliminate Smith's *Strong Problem Reduction Principle* by endeavoring to achieve these objectives, thus giving the theorem a more elegant flavor (because all constraints are specifications).[9]

Finally, the specifications S_{solve} and S_{comp} deserve some special comments. Indeed, their output conditions are the same as those of S_r, so there seems to be no real problem reduction. We will get back to this issue at the end of Section 6.3.

6 Schema-Guided Synthesis of Steadfast Programs

We now define, in Section 6.1, the concept of schema-guided synthesis as the process of setting up specifications of sub-problems, whose programs can be correctly composed according to the chosen schema (it is thus sort-of the "step case" of synthesis). We then formalise the process of re-use (which is sort-of the "base case" of synthesis) in Section 6.2. Finally, we illustrate all this, first in Section 6.3 by developing a strategy of divide-and-conquer schema-guided synthesis, and then, in Section 6.4, by applying this strategy to the synthesis of a steadfast sorting program.

6.1 Introduction to Schema-Guided Synthesis

As mentioned earlier, schemas have been successfully used to guide the synthesis of programs. The benefit of such guidance is a reduced search space, because the synthesiser, at a given moment, only tries to construct a program that fits a given schema. This is feasible because a schema fixes the data-flow and restricts the relationships between its open relations. We establish the synthesisability of open programs, rather than only of closed ones, and even of 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 the means of steadfast schemas, i.e. correct program templates together with their steadfastness constraints. However, since we have identified schemas 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 "more open" programs thus have more "guiding power," especially considering the attached specifications for their open relations.

Let us now investigate how 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 attached specifications. 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 considered above). 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 [30]), 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 originally specified relation, according to the schema. There can be several strategies for a given schema (e.g., Smith [30] gives three strategies for a divide-and-conquer schema), depending

[9] This is of course a very subjective assessment.

on which parameter(s) are instantiated first (e.g. *decompose* first, or *compose* first, or both at the same time).

Note that the halting criterion of synthesis [23] can also be pre-computed here and hardwired into any strategy, for two reasons. First, we consider partial correctness and totality simultaneously. Second, at phase (*ii*) of a strategy, the specifications of all relations introduced by a schema are set up, and it can be guaranteed, by a theorem for the underlying schema (analogous to Theorem 5.1), that the composition, according to that schema, of any programs correct wrt these specifications yields a steadfast (and hence complete) program wrt the overall initial specification and the other specifications.

We may also introduce the notion of *synthesis tactic*, as a meta-program attempting synthesis by considering the available schemas in a fixed sequence and considering the strategies of each schema in a fixed sequence. This can be refined by allowing conditional and iterative/recursive composition in such a meta-program.

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.

6.2 Re-use in Synthesis

In order to formalise the process of re-use, we first need to capture the notion of what it means for a specification to reduce to another one, which must both be over the same sorts.

Definition 6.1 (Specification reduction)
In a framework $\mathcal{F}(\Pi)$, the specification

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

reduces to the specification

$$\forall x : \mathsf{X}, \forall y : \mathsf{Y}\,.\; I_k(x) \rightarrow (r(x,y) \leftrightarrow O_k(x,y)) \tag{S_k}$$

under conditions F and G iff the following two formulas hold:

(*i*) $\mathcal{F} \models \forall x : \mathsf{X}\,.\; F(x) \wedge I_r(x) \rightarrow I_k(x)$

(*ii*) $\mathcal{F} \models \forall x : \mathsf{X}, \forall y : \mathsf{Y}\,.\; G(x) \wedge O_k(x,y) \leftrightarrow O_r(x,y)$.

When F and G are both *true*, then we say that S_r *trivially reduces to* S_k.

Since nothing prevents F from being *false*, it is clear that, for practical purposes, one should look for the weakest possible F.

Now we can propose a theorem stating when and how it is possible to re-use a known program P that is correct wrt specification S_k for correctly implementing some other specification S_r.

Theorem 6.1 (Program re-use)
In a framework $\mathcal{F}(\Pi)$, given specifications S_k and S_r as in the previous definition, if a program P is correct wrt S_k, and if S_r reduces to S_k under conditions F and G, then P is also correct wrt the specification

$$\forall x : \mathsf{X}, \forall y : \mathsf{Y}\,.\; I_r(x) \wedge F(x) \wedge G(x) \rightarrow (r(x,y) \leftrightarrow O_r(x,y)). \tag{S'_r}$$

(Note that when F and G are both *true*, then S_r and S'_r are the same.)

Proof. In framework $\mathcal{F}(\Pi)$, let P be a program that is correct wrt S_k, i.e., for all $x : \mathsf{X}$ and $y : \mathsf{Y}$ such that $I_k(x)$, we have:

$$\mathcal{F} \models O_k(x,y) \text{ iff } \mathcal{F} \cup P \vdash r(x,y). \tag{3}$$

Also let S_r reduce to S_k under conditions F and G. Now, for an arbitrary $x : \mathsf{X}$, assume that

$$I_r(x) \wedge F(x) \wedge G(x) \tag{4}$$

holds, and that

$$\mathcal{F} \cup P \vdash r(x,y) \tag{5}$$

holds, for some $y : \mathsf{Y}$. From (4) and (i), we infer that $I_k(x)$ necessarily holds. Then, from (5) and (3), we infer that $\mathcal{F} \models O_k(x, y)$ necessarily and sufficiently holds. So, from (4) and (ii), we infer that $\mathcal{F} \models O_r(x, y)$ necessarily and sufficiently holds, which means that P is necessarily and sufficiently correct wrt S'_r. □

Note that this theorem is more general than the combination of Hoare's two consequence rules, in the sense that conditions F and G need not be *true* (as inspired by Smith [30]), and that we cover total correctness (rather than just partial correctness, as Hoare and Smith do). These features will turn out crucial for synthesis, namely when the input condition of a specification is only incompletely known. As we will see, this may happen during the synthesis of divide-and-conquer programs, namely for *decompose*. We'll of course also use the theorem in situations where F and G are *true*. Formula (ii) deserves some special comments: if it is turned into an implication, then only partial correctness of P wrt S'_r is guaranteed (which is acceptable if, for some reason, relation r is known to embody a function from X to Y). It is clear that finding G such that (ii) may be quite difficult (if not impossible); in this case, the following other theorem may come in handy. Basically, it says that some conjuncts (denoted V) of the input condition of a specification S_r may be "promoted" to its output condition, and others (denoted W) dropped, so as to form a new specification with the effect that any program correct wrt S'_r will also be correct wrt S_r.

Theorem 6.2 (Input condition promotion)
In a framework $\mathcal{F}(\Pi)$, a program P is correct wrt the specification

$$\forall x : \mathsf{X}, \forall y : \mathsf{Y} \,.\, I(x) \wedge V(x) \wedge W(x) \rightarrow (r(x, y) \leftrightarrow O(x, y)) \qquad (S)$$

if P is correct wrt the specification

$$\forall x : \mathsf{X}, \forall y : \mathsf{Y} \,.\, I(x) \rightarrow (r(x, y) \leftrightarrow O(x, y) \wedge V(x)). \qquad (S')$$

(The converse does not hold.)

Proof. In framework $\mathcal{F}(\Pi)$, let P be a program that is correct wrt S', i.e., for all $x : \mathsf{X}$ and $y : \mathsf{Y}$ such that $I(x)$, we have:

$$\mathcal{F} \models O(x, y) \wedge V(x) \text{ iff } \mathcal{F} \cup P \vdash r(x, y). \qquad (6)$$

Also, for an arbitrary $x : \mathsf{X}$, assume that

$$I(x) \wedge V(x) \wedge W(x) \qquad (7)$$

holds, and that

$$\mathcal{F} \cup P \vdash r(x, y) \qquad (8)$$

holds for some $y : \mathsf{Y}$. From (7), we infer that $I(x)$ necessarily holds. Then, from (8) and (6), we infer that $\mathcal{F} \models O(x, y) \wedge V(x)$ necessarily and sufficiently holds. So, using (7), we infer that $\mathcal{F} \models O(x, y)$ necessarily ***and sufficiently*** holds, which means that P is necessarily and sufficiently correct wrt S.
□

In order to smoothly integrate the re-use process into the schema-guided synthesis machinery, we propose a *re-use schema*, with the following template:

$$r(x, y) \leftarrow directlySolve(x, y)$$

and the following (unique) constraint attached to it:

$$I_r(x) \rightarrow (directlySolve(x, y) \leftrightarrow O_r(x, y)),$$

where I_r and O_r are the input and output conditions of the specification of r. There is a single strategy for this schema, namely: instantiate *directlySolve* by trying to re-use some program. The re-use schema must be considered first by every tactic.

6.3 A Divide-and-Conquer Synthesis Strategy

Let us illustrate all these ideas on the divide-and-conquer schema, for which we here repeat the steadfastness theorem (Theorem 5.1) for convenience:

In a framework $\mathcal{F}(\Pi)$, given a well-founded order $\prec$ on its sort X, the open program

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

is steadfast wrt its specification

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

and the following specifications:

$$\forall x:\mathsf{X}\,.\, primitive(x) \leftrightarrow O_{prim}(x) \qquad (S_{prim})$$

$$\forall x:\mathsf{X}, \forall y:\mathsf{Y}\,.\, I_r(x) \wedge O_{prim}(x) \rightarrow (solve(x,y) \leftrightarrow O_r(x,y)) \qquad (S_{solve})$$

$$\begin{array}{l} \forall x, tx_1, tx_2:\mathsf{X}, \forall hx:\mathsf{H}\,.\, \neg O_{prim}(x) \rightarrow \\ (decompose(x,hx,tx_1,tx_2) \leftrightarrow Dec(x,hx,tx_1,tx_2) \wedge I_r(tx_1) \wedge I_r(tx_2) \wedge tx_1 \prec x \wedge tx_2 \prec x) \end{array} \qquad (S_{dec})$$

$$\begin{array}{l} \forall hx:\mathsf{H}, \forall ty_1, ty_2, y:\mathsf{Y}, \forall x, tx_1, tx_2:\mathsf{X}\,.\, O_{dec}(x,hx,tx_1,tx_2) \wedge O_r(tx_1,ty_1) \wedge O_r(tx_2,ty_2) \rightarrow \\ (compose(hx,ty_1,ty_2,y) \leftrightarrow O_r(x,y)). \end{array} \qquad (S_{comp})$$

where $O_{dec}(x,hx,tx_1,tx_2)$ denotes the entire output condition of *decompose*.

A possible synthesis strategy is as follows:

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"):

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

   (Remember, here and in the next steps, that the purpose of synthesis is just to set up specifications, but not to directly implement them.)

4. **Set up the specification of the discriminating operator** *primitive*. Notice that S'_{dec} is different from S_{dec}, as they have different input conditions and the output condition of S_{dec} is stronger than the one of S'_{dec}, because it also requires the tx_i to satisfy the input condition of r and to be "smaller" (according to $\prec$) than x. Also note that O_{prim}, which occurs in the input condition of S_{dec}, is still unknown. This is precisely the scenario for which (the general version of) Theorem 6.1 has been introduced. Indeed, in order to show that some P_{dec} that is correct wrt S'_{dec} is also correct wrt (yet incompletely known) S_{dec}, we may first show that the following specification:

   $$\begin{array}{l} \forall x, tx_1, tx_2:\mathsf{X}, \forall hx:\mathsf{H}\,.\, true \rightarrow \\ (decompose(x,hx,tx_1,tx_2) \leftrightarrow Dec(x,hx,tx_1,tx_2) \wedge I_r(tx_1) \wedge I_r(tx_2) \wedge tx_1 \prec x \wedge tx_2 \prec x) \end{array} \qquad (S''_{dec})$$

   reduces to S'_{dec} under some conditions F and G, and then conclude that $\neg O_{prim}(x)$ is equivalent to $F(x) \wedge G(x)$, because $\neg O_{prim}(x)$ is precisely the "difference" between S_{dec} and S''_{dec}.

   Since the input condition of S''_{dec} is *true*, formula (i) of Definition 6.1 (specification reduction) turns for this problem into

   $$\mathcal{F} \models \forall x:\mathsf{X}\,.\, F(x) \wedge true \rightarrow I_{dec}(x),$$

so that we can pre-compute that the weakest $F(x)$ always is $I_{dec}(x)$, and thus dispense with this proof obligation!

Also, since $Dec(x, hx, tx_1, tx_2)$ occurs in the output conditions of both S_{dec} and S''_{dec}, formula (ii) of Definition 6.1 can be simplified for this problem from

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

into

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

because $(G \wedge D \leftrightarrow C) \rightarrow (G \wedge D \leftrightarrow D \wedge C)$, for any formulas G, D, and C. This does not affect the previous observation about $F(x)$ always being $I_{dec}(x)$.

Once formula G has been derived, and considering that $\neg O_{prim}(x)$ is equivalent to $F(x) \wedge G(x)$, we can set up the following specification:

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

hence setting $O_{prim}(x)$ to $\neg(I_{dec}(x) \wedge G(x))$.

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 synthesis tactic 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 5.1, to be steadfast.

The specifications S_{solve} and S_{comp} (and a fortiori the obtained specifications S'_{solve} and S'_{comp}) deserve some special comments. Indeed, as observed earlier, their output conditions are the same as those of S_r, so there seems to be no real problem reduction. Moreover, their input conditions are quite complex, but the here described synthesis strategy does not make much use of input conditions and even tends to build "lengthy" ones. So if the same divide-and-conquer strategy were to be used to synthesise programs from these specifications (and this is not unusual, especially for *compose*), then all known conditions would eventually disappear into input conditions and no problem reduction would ever occur in most output conditions! Fortunately, the theorem on input condition promotion (Theorem 6.2) provides us with an elegant solution to this (at first sight disturbing) phenomenon.

Indeed, suppose that $O_r(x, y)$ has a generate-and-test structure:

$$O_r(x, y) \leftrightarrow G_r(x, y) \wedge T_r(y),$$

where $G_r(x, y)$ generates a candidate y from a given x, and $T_r(y)$ tests whether a candidate y is "good." Such a specification structure is not unusual (see the specification of *sort* above). Also suppose that $Dec(x, h, t_1, t_2)$ has a generate-and-test structure:

$$Dec(x, h, t_1, t_2) \leftrightarrow G_{Dec}(x, h, t_1, t_2) \wedge T^1_{Dec}(h, t_1) \wedge T^2_{Dec}(h, t_2),$$

where $G_{Dec}(x, h, t_1, t_2)$ is the generator, and the $T^i_{Dec}(h, t_i)$ are the testers. Again, this is not unusual (consider the specification S_{part} below). Then, Theorem 6.2 motivates the following heuristic: it often suffices to synthesise a program P_{comp} that is correct wrt

$$\begin{array}{l}\forall hx : \mathsf{H}, \forall ty_1, ty_2, y : \mathsf{Y}\,.\, T_r(ty_1) \wedge T_r(ty_2) \rightarrow (compose(hx, ty_1, ty_2, y) \leftrightarrow\\ \exists x, tx_1, tx_2 : \mathsf{X}\,.\, O_r(x, y) \wedge G_{Dec}(x, hx, tx_1, tx_2) \wedge G_r(tx_1, ty_1) \wedge G_r(tx_2, ty_2))\end{array} \qquad (S''_{comp})$$

as well as a program P_{solve} that is correct wrt

$$\forall x : \mathsf{X}, \forall y : \mathsf{Y}\,.\, solve(x, y) \leftrightarrow O_{prim}(x) \wedge O_r(x, y) \qquad (S''_{solve})$$

The output conditions of these two new specifications can usually be dramatically simplified. Note that this is not the only way of applying Theorem 6.2 here. Indeed, a more general heuristic based on that theorem would be to first promote *all* input conditions, then to simplify the resulting output condition, and finally "demote" those (former) input conditions that have not been used in this simplification process and that only involve variables hx, ty_1, ty_2, and y (this is illustrated in the following sample synthesis).

Another strategy, where a specification for composition operator *compose* is selected (or constructed) at Step 3, can be elaborated analogously.

6.4 A Sample Synthesis

Let us now show how all these considerations can be put together in order to synthesise a program from the following formal specification, in the framework $\mathcal{LIST}(Elem, \lhd)$, of the *sort* relation:

$$\forall L, S : List \,.\, sort(L, S) \leftrightarrow perm(L, S) \wedge ord(S)$$

where *perm* and *ord* are defined in $\mathcal{LIST}(Elem, \lhd)$. Note, as pointed out above, that this specification has a generate-and-test structure, where $perm(L, S)$ is the generator and $ord(S)$ the tester.

At Step 1 of the strategy, suppose that we select L as induction parameter.

At Step 2, since the induction parameter L is of sort *List*, suppose that we select $\ll$ as well-founded order, where $A \ll B$ means that list A has fewer elements than list B, i.e., formally:

$$\forall A, B : List \,.\, A \ll B \leftrightarrow l(A) < l(B).$$

At Step 3, suppose that we select the following specification of a decomposition operator, embodying the idea of partitioning list L into its first element h, the list A of the remaining elements of L that are smaller (according to $\lhd$) than h, and the list B of the remaining elements of L that are not smaller (according to $\lhd$) than h:

$$\begin{array}{l} \forall L, A, B : List, \forall h : Elem \,.\, \neg L = nil \rightarrow \\ (part(L, h, A, B) \leftrightarrow L = h.T \wedge perm(A|B, T) \wedge A \sqsubset h \wedge B \sqsupset h), \end{array} \qquad (S_{part})$$

where the following axioms:

$$L \sqsubset e \leftrightarrow \forall x \,.\, mem(x, L) \rightarrow x \lhd e$$

$$L \sqsupset e \leftrightarrow \forall x \,.\, mem(x, L) \rightarrow \neg x \lhd e$$

have been added to the framework $\mathcal{LIST}(Elem, \lhd)$. Let $Dec(L, h, A, B)$ denote the entire output condition of this specification. Note that this output condition has a generate-and-test structure, as announced above, where the first two conjuncts are the generator, and the other two conjuncts are the tester. (Also note that the clauses for *partition* in Sections 3 and 5 do not constitute a program for S_{part}, because h is here the first element of L, whereas this is not the case for *partition*.) To summarise, so far the correspondence between the theory and the example is:

$$\begin{array}{rcl} \mathcal{F} & / & \mathcal{LIST}(Elem, \lhd) \\ r & / & sort \\ x : \mathsf{X} & / & L : List \\ y : \mathsf{Y} & / & S : List \\ I_r(x) & / & true \\ O_r(x, y) & / & perm(L, S) \wedge ord(S) \\ \prec & / & \ll \\ decompose(x, hx, tx_1, tx_2) & / & part(L, h, A, B) \\ I_{dec}(x) & / & \neg L = nil \\ Dec(x, hx, tx_1, tx_2) & / & L = h.T \wedge perm(A|B, T) \wedge A \sqsubset h \wedge B \sqsupset h \end{array}$$

At Step 4, we set up the specification of the discriminating operator *primitive*. According to the pre-computations of the strategy, we have to infer a formula G such that the following instance of (ii') holds in $\mathcal{LIST}(Elem, \lhd)$:

$$\forall L, A, B : List, \forall h : Elem \,.\, G(L) \wedge Dec(L, h, A, B) \leftrightarrow true \wedge true \wedge A \ll L \wedge B \ll L.$$

It should be obvious that G is $true$. So we can set up the following specification as an instance of S'_{prim}:

$$\forall L : List \,.\, primitive(L) \leftrightarrow \neg(\neg L = nil \wedge true),$$

which simplifies into

$$\forall L : List \,.\, primitive(L) \leftrightarrow L = nil. \qquad (S_{empty})$$

At Step 5, we set up the specification of the solving operator $solve$. Again, according to the suggested pre-computations of the strategy, it suffices to set up the following instance of S''_{solve}:

$$\forall L, S : List \,.\, solve(L, S) \leftrightarrow L = nil \wedge perm(L, S) \wedge ord(S)$$

which simplifies into:

$$\forall L, S : List \,.\, solve(L, S) \leftrightarrow S = nil, \qquad (S_{empty2})$$

based on the theorems $perm(nil, nil)$ and $ord(nil)$, which theorems are derivable from the axioms in $\mathcal{LIST}(Elem, \lhd)$.

Finally, at Step 6, we set up the specification of the composition operator $compose$. We follow the most general heuristic and initially promote *all* input conditions:

$$\begin{aligned}
&\forall h : Elem, \forall C, D, S : List \,.\, compose(h, C, D, S) \leftrightarrow \\
&\exists L, T, A, B : List \,.\, L = h.T \wedge perm(A|B, T) \wedge A \sqsubset h \wedge B \sqsupset h \wedge true \wedge true \wedge A \ll L \wedge B \ll L \\
&\wedge perm(A, C) \wedge ord(C) \wedge perm(B, D) \wedge ord(D) \wedge perm(L, S) \wedge ord(S)
\end{aligned}$$

The output condition of this specification can be simplified as follows. First, we infer $T = C|D$ by applying the following theorem:

$$perm(X|Y, Q|R) \leftarrow perm(X, Q) \wedge perm(Y, R)$$

to $perm(A|B, T)$, $perm(A, C)$, and $perm(B, D)$.
Then, we infer $perm(C|(h.D), S)$ by applying the following theorems:

$$perm(e.(X|Y), X|(e.Y))$$

$$perm(X', Y) \leftarrow perm(X, Y) \wedge perm(X, X')$$

to $perm(L, S)$, $L = h.T$, and $T = C|D$.
Next, we infer $C \sqsubset h$ and $D \sqsupset h$ by applying the following theorems:

$$Y \sqsubset e \leftarrow X \sqsubset e \wedge perm(X, Y)$$

$$Y \sqsupset e \leftarrow X \sqsupset e \wedge perm(X, Y)$$

to $A \sqsubset h$ and $perm(A, C)$, respectively $B \sqsupset h$ and $perm(B, D)$.
Now, we infer $ord(C|(h.D))$ by applying the following theorem:

$$ord(X|(e.Y)) \leftarrow ord(X) \wedge X \sqsubset e \wedge Y \sqsupset e \wedge ord(Y)$$

to $ord(C)$, $C \sqsubset h$, $D \sqsupset h$, and $ord(D)$.
Finally, we infer the final output condition $S = C|(h.D)$ by applying the following theorem:

$$perm(X, Y) \wedge ord(Y) \wedge ord(X) \rightarrow Y = X$$

to $perm(C|(h.D), S)$ (which was formerly $perm(L, S)$), $ord(S)$, and $ord(C|(h.D))$. Since all promoted input conditions either have been used in this simplification process or do not involve variables h, C, D, and S only, none of these conditions gets "demoted" to the final input condition, which is thus $true$. So we have set up the following specification:

$$\forall h : Elem, \forall C, D, S : List \,.\, compose(h, C, D, S) \leftrightarrow S = C|(h.D) \qquad (S_{catcons})$$

This simplification process (as well as the derivation of antecedent G above) leaves open the question about the origins of the used theorems, as well as the full description of the used proof system. As of now, these are open questions, but the objective of this paper is to show the feasibility of schema-guided synthesis of steadfast (open) programs, not to flesh out all the details on how to actually do it.

Four specifications (S_{part}, S_{empty}, S_{empty2}, and $S_{catcons}$) have been set up now, so four auxiliary syntheses are started from them. The latter three syntheses are trivial (and should succeed through the re-use schema and strategy, whereas the first one can be guided by the divide-and-conquer schema and strategy. We omit these syntheses here, but after extending the template with their results, one could for instance get the following program:

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

which is guaranteed, by Theorem 5.1, to be steadfast. Note that this is an open program, as there are no clauses yet for *append*, nor $\lhd$. This is the classical Quicksort program.

Other choices at Step 3 (instantiation of *decompose*) would lead to other sorting programs, such as insertion-sort, merge-sort, etc (as shown in [19] for instance). If we had, at Step 3, instantiated *compose* as above (using the not yet precomputed second strategy), then a specification for *decompose* would have been set up at Step 6, and a partitioning program synthesised from it. This choice is pursued in [30], but leads (there) to a very weird (and inefficient) partitioning program.

7 Conclusion, Related Work, and Future Work

We have shown 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.

Related work at the framework and steadfastness level is described in [20, 23, 24]. It focuses mainly on a model-theoretic characterisation of the semantics of frameworks (and specifications) and steadfastness. This provides a sound theoretical basis for modular program development by composing or reusing frameworks as well as steadfast programs. In terms of synthesis, it uses an incremental synthesis process (i.e. it synthesises one clause at a time) and gives halting criteria for determining when the synthesised program is steadfast (and hence for halting the synthesis process).

At the schema-guided synthesis level, this paper is very strongly influenced by Smith's pioneering work [30] 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 open programs, and simplified (the formulation and proof of) the steadfastness theorem.

Future work includes the development of a proof system for deriving antecedents (such as for formulas (i) and (ii')) and for obtaining simplifications of output conditions (such as those of *solve* and *compose*). As seen, to be efficient, this requires the pre-existence of a considerable set of theorems of the axiomatic theory in a framework, which theorems state the combined effects of the functions and relations of the framework. Such theorems could be either hand-crafted (and mechanically verified), or generated by forward reasoning. The work of Smith [29, 30] shows that deriving an antecedent A of a formula F (i.e., such that $A \rightarrow F$ is valid) is a generalization both of formula simplification (find a weakest antecedent of "optimal syntactic complexity") and of "conventional" theorem proving (find *true* as antecedent). In-between these (known) extremes lie other usages of antecedent derivation that are, as seen, crucial to schema-guided synthesis.

Another important objective is to redo the constraint abduction process for the more general template (i.e. where $nonPrimitive(x)$ is not necessarily $\neg primitive(x)$), and to develop the corresponding strategies, in order to allow the synthesis of non-deterministic programs.

We also need to show how well-founded orders on "complex" sorts (built from existing sorts by taking, e.g., their cross-products), as well as decomposition operators for such sorts can be constructed automatically, so that synthesis is not restricted to selecting among known orders and operators.

Other strategies for the divide-and-conquer schema need furthermore be elaborated.

Despite the ubiquity of divide-and-conquer programs, there are of course other design methodologies that need to be captured and codified in schemas (i.e. templates and constraints) and their strategies. Once again, Smith [31] has shown the way, namely by capturing a vast class of search methodologies in a global-search schema and seven corresponding strategies.

Eventually, a proof-of-concept implementation of the outlined synthesiser (and the adjunct proof system) is planned.

The schema-guided approach to synthesis is shown to involve a fair amount of theorem-proving-like tasks, so one may wonder whether this approach will not suffer from the same drawbacks as the deductive and constructive approaches to synthesis, which are also (somehow) based on automated theorem proving. We pretend that search spaces are much smaller in schema-guided synthesis (because very "targeted" proof obligations are set up), and that the resulting programs are better structured (because most of a design methodology can be hardwired in a schema and its strategies). The notion of proof plans [17] (and their use in directing synthesis) is not incompatible with our notion of schema guidance, and it would be exciting to explore the commonalities.

Acknowledgements

This work was partially supported by the European Union HCM Project on Logic Program Synthesis and Transformation, contract no. 93/414. We wish to thank Doug Smith for his pioneering work that inspired us.

References

[1] N. Dershowitz. *The Evolution of Programs*. Birkhäuser, 1983.

[2] 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.

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

[4] Y. Deville and K.-K. Lau. Logic program synthesis. *J. Logic Programming* 19,20:321–350, 1994. Special issue: 10 years of Logic Programming.

[5] P. Flener. *Logic Program Synthesis from Incomplete Information*. Kluwer, 1995.

[6] P. Flener. Logic Program Schemata: Synthesis and Analysis. Tech Rep BU-CEIS-9502, Bilkent University, Ankara, Turkey, 1995.

[7] P. Flener. Inductive logic program synthesis with DIALOGS. In S. Muggleton, editor, *Proc. ILP'96*. *LNAI*, Springer-Verlag, forthcoming.

[8] 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.

[9] T.S. Gegg-Harrison. Learning Prolog in a schema-based environment. *Instructional Science* 20:173–190, 1991.

[10] T.S. Gegg-Harrison. Exploiting program schemata in an automated program debugger. *J. Artificial Intelligence in Education* 5:255–278, 1994.

[11] T.S. Gegg-Harrison. Representing logic program schemata in λ-Prolog. In L. Sterling, editor, *Proc. ICLP'95*, pages 467–481. MIT Press, 1995.

[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] I. Kraan, D. Basin, and A. Bundy. Logic program synthesis via proof planning. In K.-K. Lau and T. Clement, editors, *Proc. LOPSTR'92*, pages 1–14. Springer-Verlag, 1993.

[18] 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. ICLP'90*, pages 667–684. MIT Press, 1990.

[19] K.-K. Lau and S. D. Prestwich. Synthesis of a family of recursive sorting procedures. In V. Saraswat and K. Ueda, editors, *Proc. ILPS'91*, pages 641–658. MIT Press, 1991.

[20] K.-K. Lau and M. Ornaghi. On specification frameworks and deductive synthesis of logic programs. In L. Fribourg and F. Turini, editors, *Proc. LOPSTR/META'94*, pages 104–121. *LNCS* 883, Springer-Verlag, 1994.

[21] K.-K. Lau and M. Ornaghi. The relationship between logic programs and specifications: The subset example revisited. *J. Logic Programming*, 30(3):239–257, 1997.

[22] 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.

[23] 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.

[24] K.-K. Lau, M. Ornaghi, and S.-Å. Tärnlund. Steadfast logic programs. *J. Logic Programming*, submitted.

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

[26] Z. Manna. *Mathematical Theory of Computation*. McGraw-Hill, 1974.

[27] 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.

[28] D. Sannella and A. Tarlecki. Essential concepts of algebraic specification and program development. *Formal Aspects of Computer Science*, forthcoming.

[29] D.R. Smith. Derived preconditions and their use in program synthesis. In D.W. Loveland, editor, *Proc. CADE'82*, pages 172–193.

[30] D.R. Smith. Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence* 27(1):43–96, 1985.

[31] D.R. Smith. KIDS: A semiautomatic program development system. *IEEE Trans. Software Engineering* 16(9):1024–1043, 1990.

[32] L.S. Sterling and M. Kirschenbaum. Applying techniques to skeletons. In J.-M. Jacquet, editor, *Constructing Logic Programs*, pages 127–140. John Wiley, 1993.

[33] 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.

[34] M. Wirsing. Algebraic specification. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 675–788. Elsevier, 1990.