Correct-schema-guided Synthesis of Steadfast Programs

Pierre Flener
Dept of Computer Science
Bilkent University
06533 Bilkent, Ankara, Turkey
pf@cs.bilkent.edu.tr

Kung-Kiu Lau
Dept of Computer Science
University of Manchester
Manchester M13 9PL, UK
kung-kiu@cs.man.ac.uk

Mario Ornaghi
DSI
Univ. degli studi di Milano
20135 Milano, Italy
ornaghi@dsi.unimi.it

Abstract

It can be argued that for (semi-)automated software development, program schemas are indispensable, since they capture not only structured program design principles, but also domain knowledge, both of which are of crucial importance for hierarchical program synthesis. Most researchers represent schemas purely syntactically (as higher-order expressions). This means that the knowledge captured by a schema is not formalised. We take a semantic approach and show that a schema can be formalised as an open (first-order) logical theory that contains an open logic program. By using a special kind of correctness for open programs, called steadfastness*, we can define and reason about the correctness of schemas. We also show how to use correct schemas to synthesise steadfast programs.*

1. Introduction

It can be argued that any systematic approach to software development must use some kind of schema-based strategies. In (semi-)automated software development, program schemas become indispensable, since they capture not only structured program design principles, but also domain knowledge, both of which are of crucial importance for hierarchical program synthesis. This is amply borne out by user-guided program development systems that have been successfully applied in practice, e.g. KIDS [17].

Informally, a program schema is an abstraction (in a given problem domain) 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. At a syntactic level, a schema is an open program, or a template, which can be instantiated to any concrete program of which it is an abstraction. Thus, most researchers represent schemas as higher-order syntactic expressions from which actual programs are obtained by higher-order substitutions. However, in such a purely syntactic approach, the knowledge that is captured by a schema is not formalised.

We take a semantic approach and show that a schema $\mathcal{S}$ consists of a syntactic component, viz. a template T, and a semantic component. T is formalised as an open (first-order) logic program in the context of the problem domain, characterised as a first-order axiomatisation called a *specification framework* $\mathcal{F}$ [10, 11]. $\mathcal{F}$ endows the schema $\mathcal{S}$ with a formal semantics, and enables us to define and reason about its correctness. In particular, we define a special kind of correctness for open programs such as templates, that we call *steadfastness*. A steadfast (open) program is always correct (wrt its specification) as long as its parameters are correctly computed (wrt their specifications). This means that a steadfast open program, though only partially defined, is always *a priori* correct when (re-)used in program composition, in the sense that its defined part is *a priori* correct (wrt its specification). A steadfast program is thus *a priori* correctly reusable, and such programs make ideal units in a library from which correct programs can be composed.

Thus we define a correct schema to be a specification framework containing a steadfast open program. Moreover, we show how to use correct schemas to synthesise steadfast open logic programs. The notion of correctness applied to schemas and the use of correct schemas in synthesising steadfast programs are the main novel themes of this paper.

Our general approach follows that of the pioneering work of Smith in functional programming [16]. Although we focus on the logic programming paradigm (see [13] for basic terminology), our ultimate goal is to extend it to a general paradigm with suitable logic semantics.

2. Defining Correct Schemas

Our approach to logic program synthesis is set in the context of a (fully) first-order axiomatisation of the problem domain in question, which we call a *specification framework* $\mathcal{F}$. Specifications and programs are given in the context 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.

Our notion of correctness for open programs is a special kind of correctness that we call *steadfastness*, and we define a correct program schema as a specification framework containing a steadfast (open) program. In this section, we discuss steadfastness and correct program schemas, but due to lack of space we can only give a brief summary (a more detailed account with examples can be found in [4]).

2.1. Specification Frameworks

Definition 2.1 A *specification framework* $\mathcal{F}(\Pi)$ with parameters Π consists of a (many-sorted) signature Σ and a set of first-order axioms for the symbols of Σ. The parameters Π belong to Σ. The axioms for the parameters are called *p-axioms*. We say that a (specification) framework $\mathcal{F}(\Pi)$ is *open* if Π is not empty; otherwise, we say that it is *closed* and we indicate it by $\mathcal{F}$.

A closed framework $\mathcal{F}$ axiomatizes one problem domain, as an intended model (unique up to isomorphism). In our approach, intended models are *reachable isoinitial* models. A model i is *reachable* if its elements can be represented by ground terms; a reachable model of $\mathcal{F}$ is *isoinitial* iff ground quantifier-free formulas are true in it whenever they are true in every model of $\mathcal{F}$.

Following the tradition of algebraic ADTs [14, 18], initial models have also been proposed for logic programs [5, 6]. We have preferred isoinitial models to properly deal with negation (see also the primal models proposed in [7]).

In general, a framework may have no isoinitial model. Hence the following adequacy condition:

Definition 2.2 A closed framework $\mathcal{F}$ is *adequate* if it has a reachable isoinitial model.

A typical closed framework is (first-order) Peano arithmetic $\mathcal{NAT}$, using the well-known axiomatisation, including the first-order induction schema. $\mathcal{NAT}$ has the standard structure of natural numbers as an intended (reachable isoinitial) model.

An open framework $\mathcal{F}(\Pi)$ has a non-empty set Π of parameters, which can be *instantiated* by a closed framework $\mathcal{G}$. The *instance*, denoted by $\mathcal{F}(\Pi)[\mathcal{G}]$, is the union of (the signatures and the axioms of) $\mathcal{F}(\Pi)$ and $\mathcal{G}$. It is defined only if Π is the intersection of the signatures of $\mathcal{F}(\Pi)$ and $\mathcal{G}$, and $\mathcal{G}$ proves the p-axioms.

Definition 2.3 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.

A more general notion of instance can be given, involving renamings (see also the pushout approach in algebraic ADTs [18]). However, it can be shown that $\mathcal{F}(\Pi)$ is adequate according to Definition 2.3 iff it is adequate considering the more general notion of instance. Therefore we can use our simpler definition, without loss of generality.

Example 2.1 The following open framework axiomatises the (kernel of the) theory of lists with parametric element sort $Elem$ and parametric total ordering relation $\lhd$:

Specification Framework $\mathcal{LIST}(Elem, \lhd)$;
IMPORT: $\mathcal{NAT}$;
SORTS: $Nat, Elem, List$;
FUNS: $nil : \rightarrow List$;
$\cdot : (Elem, List) \rightarrow List$;
$nocc : (Elem, List) \rightarrow Nat$;
RELS: $elemi : (List, Nat, Elem)$;
$\lhd : (Elem, Elem)$;
AXS: $\text{C-AXS}(nil, \cdot)$;
$elemi(L, i, a) \leftrightarrow \exists h, T, j \,.\, L = h \cdot T \wedge$
$(i = 0 \wedge a = h \vee i = s(j) \wedge elemi(T, j, a))$;
$nocc(x, nil) = 0$;
$a = b \rightarrow nocc(a, b \cdot L) = nocc(a, L) + 1$;
$\neg a = b \rightarrow nocc(a, b \cdot L) = nocc(a, L)$;
P-AXS: …total ordering axioms for $\lhd$ …

where $\text{C-AXS}(nil, \cdot)$ contains Clark's Equality Theory (see [13]) 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.

If $\mathcal{INT}$ is a closed framework axiomatising integers Int with total ordering $\leq$, then $\mathcal{LIST}(Int, \leq)[\mathcal{INT}]$ is a closed framework that axiomatises finite lists of integers. Note the renaming of $Elem$ by Int and $\lhd$ by $\leq$.

2.2. Specifications

Definition 2.4 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 every model of $\mathcal{F}$ a set of $(\Sigma + \delta)$-expansions, called *S_δ-expansions* (where $\Sigma + \delta$ is the signature Σ enriched by δ). An S_δ-expansion of a model m of $\mathcal{F}$ is any $(\Sigma + \delta)$-expansion m' of m, such that m' is a model of S_δ. A specification S_δ is *strict*, if, for every model m of $\mathcal{F}$, there is one S_δ-expansion. It is *non-strict* otherwise.

For uniformity, in this paper, we shall only use *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:Yare (possibly empty) lists of sorted variables, with sorts in $\mathcal{F}$. Q is called the *input condition*, whereas R is called the *output condition* of the specification.

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

In our approach, there is a clear distinction between frameworks and specifications. The latter introduce new symbols and assume their proper meaning only in the context of the framework.

Example 2.2 In $\mathcal{LIST}(Elem, \lhd)$, we can specify the usual length and concatenation functions l and $|$, and the usual 'membership', 'concatenation', 'permutation', 'ordered' and 'sort' relations mem, $append$, $perm$, ord and $sort$ as follows (we drop the universal quantifications at the beginning of specifications):

SPECS:
$mem(e, L) \leftrightarrow \exists i \,.\, elemi(L, i, e)$;
$n = l(L) \leftrightarrow \forall i \,.\, i < n \leftrightarrow \exists a \,.\, elemi(L, i, a)$;
$append(A, B, L) \leftrightarrow \forall i, a \,.$
$(elemi(A, i, a) \leftrightarrow elemi(L, i, a) \wedge i < l(A)) \wedge$
$(elemi(B, i, a) \leftrightarrow elemi(L, i + l(A), a))$;
$perm(A, B) \leftrightarrow \forall e \,.\, nocc(e, A) = nocc(e, B)$;
$C = A|B \leftrightarrow append(A, B, C)$;
P-SPECS:
$ord(L) \leftrightarrow$
$\forall i \,.\, elemi(L, i, e_1) \wedge elemi(L, s(i), e_2) \rightarrow e_1 \lhd e_2$;
$sort(L, S) \leftrightarrow perm(L, S) \wedge ord(S)$.

To distinguish the specified symbols from the signature of the framework, we will call them *s-symbols*. Also, specifications and axioms are clearly distinguished.

An s-symbol δ with specification S_δ can be used to expand the signature of the framework by δ and its axioms by S_δ. An expansion is *adequate* iff framework adequacy is preserved.

The expansions of $\mathcal{LIST}(Elem, \lhd)$ by l, $|$, mem, $append$, $perm$, ord and $sort$ can be shown to be adequate. In the following, we will consider $\mathcal{LIST}$ thus expanded. Note that in the expanded framework these symbols can be used both as s-symbols and as symbols of the language.

2.3. Correctness of Open Programs

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.

A model-theoretic definition of correctness of open programs in a framework, called *steadfastness*, is given in [11]. Here, we give a less abstract, but more conventional definition (for a comparison, see [1, 12]). 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 2.5 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:

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

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 either with 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 $(S_r, TC_t(r))$.

(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, and we say that P_r is *correct* wrt $(S_r, PC_t(r(x \rightarrow y)))$.

$TC_t(r)$ and $PC_t(r(x \rightarrow y))$ will be called *termination requirements*.

It is easy to see that total correctness is too weak for case (i), since a totally correct P_r could not 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, T_r), where T_r is a termination requirement; we will consider correctness wrt (S_r, T_r).

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 2.6 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 2.7 P_r is *steadfast* in an open framework $\mathcal{F}(\Pi)$ if it is steadfast in every instance $\mathcal{F}[\mathcal{G}]$.

2.4. Correct Schemas

Now we define a schema as an open framework containing a steadfast (open) program.

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 (it is worth recalling that programs are Σ-programs, and specifications are Σ-formulas, where Σ is the signature of $\mathcal{F}(\Pi)$):

Definition 2.8 A *correct* (*program*) *schema* for a relation r is an open framework $\mathcal{S}(\Pi)$ containing a steadfast program P_r for r. P_r is called the *template* of $\mathcal{S}(\Pi)$, whereas the p-axioms and the p-specifications are called the *constraints* of $\mathcal{S}(\Pi)$. A schema $\mathcal{S}$ *covers* a program P if ($\mathcal{S}$ and) its template can be instantiated into P.

Most researchers, with the laudable exception of Smith [16, 17], define a schema to be just a template. Such definitions are thus merely syntactic, providing only a pattern of place-holders, but not 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 synthesis, and the additional knowledge (corresponding to our constraints) somehow has to be hard-wired into 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 functional ones), uninstantiated schemas (rather than instantiated ones), and we set everything up 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 open frameworks.

Example 2.3 Figure 1 gives a divide-and-conquer schema $\mathcal{DC}$, for which one can prove the following theorem [4].

> **Theorem 2.1** The schema $\mathcal{DC}$ is correct, i.e. it contains a steadfast template.

This theorem is related to the one given by Smith [16] 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. Also, we could eliminate Smith's *Strong Problem Reduction Principle* by endeavouring to achieve these objectives.

3. Synthesis of Steadfast Programs

In the rest of the paper, we show how we can use correct schemas to guide the synthesis of steadfast programs.

Schemas have been successfully used to guide the synthesis of programs [16, 17, 2]. 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. In our approach, we use correct schemas, and 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.

We investigate how much of the synthesis process can be pre-computed at the level of "completely open" schemas. The key to pre-computation lies in the constraints. These can be seen as an "overdetermined system of equations (in a number of unknowns)," which may be unsolvable as it stands (as is the case for the divide-and-conquer schema above). An arbitrary instantiation, 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.

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

$$
\begin{array}{lll}
\text{SORTS:} & \mathsf{X},\mathsf{Y},\mathsf{H}; & \\
\text{RELS:} & I_r, I_{dec} \;:\; (\mathsf{X}); & \\
 & O_r \;:\; (\mathsf{X},\mathsf{Y}); & \\
 & O_{dec} \;:\; (\mathsf{X},\mathsf{H},\mathsf{X},\mathsf{X}); & \\
\text{P-AXS:} & I_{dec}(x) \wedge O_{dec}(x,h,x_1,x_2) \rightarrow I_r(x_1) \wedge x_1 \prec x \wedge I_r(x_2) \wedge x_2 \prec x; & (c_1) \\
 & I_{dec}(x) \rightarrow \exists h, x_1, x_2 \,.\, O_{dec}(x,h,x_1,x_2); & (c_2) \\
\text{P-SPECS:} & I_r(x,y) \rightarrow (r(x,y) \leftrightarrow O_r(x,y)) & (S_r) \\
 & I_r(x) \rightarrow (primitive(x) \leftrightarrow \neg I_{dec}(x)) & (S_{prim}) \\
 & I_{dec}(x) \rightarrow (decompose(x,h,x_1,x_2) \leftrightarrow O_{dec}(x,h,x_1,x_2)) & (S_{dec}) \\
 & I_r(x) \wedge \neg I_{dec}(x) \rightarrow (solve(x,y) \leftrightarrow O_r(x,y)) & (S_{solve}) \\
 & O_{dec}(x,h,x_1,x_2) \wedge O_r(x_1,y_1) \wedge O_r(x_2,y_2) \rightarrow (compose(h,y_1,y_2,y) \leftrightarrow O_r(x,y)) & (S_{comp}) \\
\text{T-REQS:} & PC_t(r(x \rightarrow y)) \;\Leftarrow\; TC_t(primitive), PC_t(solve(x \rightarrow y)), & \\
 & \qquad PC_t(decompose(x \rightarrow h,x_1,x_2)), PC_t(compose(h,y_1,y_2 \rightarrow y)) & \\
\text{TEMPL:} & r(x,y) \;\leftarrow\; primitive(x), solve(x,y) & \\
 & r(x,y) \;\leftarrow\; \neg primitive(x), decompose(x,hx,tx_1,tx_2), & (T_r) \\
 & \qquad r(tx_1,ty_1), r(tx_2,ty_2), compose(hx,ty_1,ty_2,y) &
\end{array}
$$

Figure 1. A correct divide-and-conquer schema.

This leads to the notion of *synthesis strategy* (cf. Smith's work [16]), as a pre-computed (finite) sequence of synthesis steps, for a given schema. A strategy has two phases, stating first which parameter(s) to arbitrarily instantiate, and next which specifications to "set up", 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), 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 [16] gives three strategies for a divide-and-conquer schema), depending on which parameter(s) are instantiated first.

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 [16]. Problem reduction (which is the "step case" of synthesis) 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 the "base case" of synthesis).

3.1. Re-use in Synthesis

To formalise the process of re-use, we need to capture what it means for a specification to reduce to another one.

Definition 3.1 In a framework $\mathcal{F}(\Pi)$ with isoinitial model i, the specification

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

reduces to the specification

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

under conditions F and G iff the following hold:

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

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

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 3.1 In a closed framework $\mathcal{F}$ with isoinitial model i, given specifications S_k and S_r (as above), if a program P is correct wrt S_k and termination requirement T, and if S_r reduces to S_k under conditions F and G, then P is also correct wrt the specification

$$I_r(x) \wedge F(x) \wedge G(x) \rightarrow (r(x,y) \leftrightarrow O_r(x,y)) \qquad (S'_r)$$

and the same termination requirement T.

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

$$\mathsf{i} \models O_k(x,y) \text{ iff } P \vdash r(x,y) \qquad (2)$$

Let S_r reduce to S_k under conditions F and G. For an arbitrary $x : \mathsf{X}$, assume

$$I_r(x) \wedge F(x) \wedge G(x) \tag{3}$$

$$P \vdash r(x, y) \tag{4}$$

for some $y : \mathsf{Y}$. From (3) and (i), we infer that $I_k(x)$ necessarily holds. From (4) and (2), we infer that $\mathsf{i} \models O_k(x, y)$ necessarily and sufficiently holds. From (3) and (ii), we infer that $\mathsf{i} \models O_r(x, y)$ necessarily and sufficiently holds. Since $TC_t(r)$ holds (the input condition is stronger), P is correct wrt $(S'_r, TC_t(r))$.
The proof for $T = PC_t(r(x \rightarrow y))$ goes similarly, considering partial instead of total correctness. □

This theorem is more general than the combination of Hoare's two consequence rules, since conditions F and G need not be $true$ (as inspired by Smith [16]), and since we cover total correctness (rather than just partial correctness, as Hoare and Smith do). This will turn out crucial for synthesis, namely when the input condition of a specification is incompletely known. Finding G such that (ii) holds may be quite difficult (if not impossible); the following theorem may then come in handy. It says that some conjuncts (denoted V) of the input condition of a specification S may be "promoted" to its output condition, so as to form a new specification S', with the effect that any call to a program that is correct wrt S can be replaced by a call to a program that is correct wrt S', provided V holds in the context of that call.

Theorem 3.2 In a closed framework $\mathcal{F}$ with isoinitial model i, any call to a program that is correct wrt the specification

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

and the termination requirement T can be replaced by a call to a program that is correct wrt

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

and the same requirement T, provided $V(x)$ holds in the context of that call.

Proof. Let T be $TC_t(r)$. In $\mathcal{F}$, let P be a program that is totally correct wrt S', i.e., for all $x : \mathsf{X}$ and $y : \mathsf{Y}$ such that $I(x)$, we have:

$$\mathsf{i} \models O(x, y) \wedge V(x) \;\text{ iff }\; P \vdash r(x, y) \tag{5}$$

For an arbitrary $x : \mathsf{X}$, assume

$$I(x) \wedge V(x) \tag{6}$$

$$P \vdash r(x, y) \tag{7}$$

for some $y : \mathsf{Y}$. From (6), we infer that $I(x)$ necessarily holds. From (7) and (5), we infer that $\mathcal{F} \models O(x, y) \wedge V(x)$ necessarily and sufficiently holds. Using (6), we infer that $\mathsf{i} \models O(x, y)$ necessarily *and sufficiently* holds. Since $TC_t(r)$ holds (the input condition is stronger), P is correct wrt $(S, TC_t(r))$.
The proof for $T = PC_t(r(x \rightarrow y))$ is similar. □

Theorems 3.1 and 3.2 can also be used in an open framework $\mathcal{F}(\Pi)$, to replace a specification with a better one, while preserving steadfastness (indeed, the proofs of (i) and (ii) in $\mathcal{F}(\Pi)$ are inherited by every instance). They can also be used in the single instances, allowing redefinition of program components.

3.2. A Divide-and-Conquer Synthesis Strategy

We illustrate all these ideas on the divide-and-conquer schema. Some reductions can be done directly at the level of the schema, which already contains, as a built-in, the following divide-and-conquer strategy. After the instantiation of I_r, O_r, $x : \mathsf{X}$, $y : \mathsf{Y}$, proceed as follows:
1. Select (or construct) a well-founded order (wfo) over the input sort X.
2. Select (or construct) a decomposition operator $decompose$, such that it satisfies (c_1) and (c_2). Suppose the following specification is obtained:

$$\begin{array}{l} I_{dec}(x) \rightarrow \\ (decompose(x, h, t_1, t_2) \leftrightarrow O_{dec}(x, h, t_1, t_2)) \end{array} \quad (S'_{dec})$$

3. Set up the specifications of the operators $primitive$, $solve$ and $compose$.
We can set up the specifications of the last step, because all their place-holders are known. In this way, four specifications $(S'_{dec}, S'_{prim}, S'_{solve}, S'_{comp})$ are set up, so four auxiliary syntheses can be started from them, using the same overall synthesis approach again, but not necessarily the (same) strategy for the (same) divide-and-conquer schema. The programs P_{dec}, P_{prim}, P_{solve}, P_{comp} resulting from these auxiliary syntheses are added to the template P_r of the schema, yielding a steadfast program, by Theorem 2.1.

The specifications S_{solve} and S_{comp} (and *a fortiori* 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. Moreover, their input conditions are quite complex, but the synthesis strategy described here does not make much use of input conditions and even tends to build "lengthy" ones. So if the same divide-and-conquer strategy were used to synthesise programs from these specifications (and this is not unusual, especially for $compose$), then all conditions would eventually disappear into input conditions and no problem reduction would ever occur in most output conditions! Fortunately, Theorem 3.2 provides an elegant solution to this (at first sight disturbing) phenomenon: since the input conditions of S_{solve} and S_{comp} are only made of the output conditions of (some of) their preceding computations (i.e. are guaranteed to hold in the calling context), one can promote these *entire* input conditions, then simplify the resulting output conditions, and call programs implementing these new specifications rather than the old ones.

3.3. A Sample Synthesis

We now show how all these considerations can be put together in order to synthesise a program from the $sort$ specification of Example 2.2. See [3] for more details.

We are in $\mathcal{LIST}(Elem, \lhd)$ and we want a steadfast sorting program with termination requirement $PC_t(sort(L \rightarrow S))$. Note that, since $sort$ is functional, this entails total correctness. We instantiate $I_r(L)$ by $true$, $O_r(L,S)$ by $perm(L,S) \wedge ord(S)$, $x : \mathsf{X}$ by $L : List$, and $y : \mathsf{Y}$ by $S : List$.

At Step 1, since L is of sort $List$, suppose we select $\ll$ as wfo, where $A \ll B$ means that A has fewer elements than B, i.e. $\forall A, B : List \, . \, A \ll B \leftrightarrow l(A) < l(B)$.

At Step 2, suppose we *select* the following specification of a decomposition operator, partitioning 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:

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

where the following axioms:

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

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

At Step 3, we set up the specifications of $primitive$, $solve$ and $compose$. For $primitive$ we get:

$$true \rightarrow (primitive(L) \leftrightarrow L = nil) \qquad (S_{empty})$$

For $solve$, after promoting the entire input condition of S_{solve}, we get:

$$solve(L,S) \leftrightarrow true \wedge L = nil \wedge perm(L,S) \wedge ord(S)$$

which simplifies into

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

Finally, we set up the specification of the composition operator $compose$. We promote the entire input condition:

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

which simplifies into

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

We leave open how these simplifications can be done (but see [4]). Our objective here is just to show the feasibility of schema-guided synthesis of steadfast (open) programs, not the details of how to actually do it.

Four specifications (S_{part}, S_{empty}, S_{empty2}, $S_{catcons}$) having been set up, four auxiliary syntheses are started from them. The latter three syntheses are trivial, whereas the first one can be guided by the divide-and-conquer schema and strategy. We omit them here, but after adding their result programs to the template, one could get the classical Quicksort program, which is steadfast, by Theorem 2.1.

Other choices at Step 3 would lead to other sorting programs, such as insertion-sort, merge-sort, etc (as shown in [9] for instance).

4. Conclusion

We have defined a notion of correctness for program schemas, and we have shown how we can use such schemas to guide the synthesis of steadfast, i.e. correct and *a priori* correctly reusable (divide-and-conquer) programs, from formal specifications expressed in the first-order language of a framework. In both these aspects, our work extends previous work in schema-guided synthesis. The synthesis of steadfast open programs is important from the point of view of constructing a library of correctly reusable program units for a chosen problem domain. However, here we have only laid the theoretical foundations; much more needs to be done in order to apply the results to the implementation of a practical system for (semi-)automated software development.

At the schema-guided synthesis level, our work is very strongly influenced by Smith's pioneering work [16] in functional programming in the early 1980s. Our work is however *not* limited to simply transposing this 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 theorem on the divide-and-conquer schema (Theorem 2.1).

Future work includes the development of a proof system for deriving antecedents and for obtaining simplifications of output conditions. To be efficient, this requires the preexistence 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 [15, 16] shows that deriving an antecedent A of a formula F (i.e., such that $A \rightarrow F$ is valid) is a generalisation both of formula simplification (find a weakest antecedent of "minimal syntactic complexity") and of "conventional" theorem proving (find $true$ as antecedent). In-between these (known) extremes lie other usages of antecedent derivation that are crucial to schema-guided synthesis.

We also need to abduce the constraints for a more general template (namely where $nonPrimitive(x)$ replaces $\neg primitive(x)$), and to develop the corresponding strategies, in order to allow the synthesis of larger classes of non-deterministic programs.

Another important objective is to identify templates and constraints for other design methodologies than divide-and-conquer, and to develop corresponding strategies. Once again, Smith [17] has shown the way, namely by capturing a vast class of search methodologies in a global-search schema and seven corresponding strategies. At the same time, other strategies for the divide-and-conquer schema also need to be developed.

Eventually, we plan a proof-of-concept implementation of the outlined synthesiser (and the adjunct proof system). Since schema-guided synthesis involves a fair amount of theorem-proving-like tasks, the notion of proof plans [8] and their use in directing synthesis will be worth investigating.

Acknowledgements

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

References

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

[2] P. Flener and Y. Deville. Logic program synthesis from incomplete specifications. *J. Symbolic Computation* 15(5–6):775–805, May/June 1993.

[3] P. Flener and K.-K. Lau. Program Schemas as Steadfast Programs and their Usage in Deductive Synthesis. Tech Rep BU-CEIS-9705, Bilkent University, 1997.

[4] P. Flener, K.-K. Lau, and M. Ornaghi. On correct program schemas. In N.E. Fuchs, editor, *Proc. LOPSTR'97*, Springer-Verlag, forthcoming.

[5] 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.

[6] 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.

[7] G. Jäger. Annotations on the consistency of the Closed World Assumption. *J. Logic Programming* 8(3):229–248, 1990.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

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

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

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

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

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

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