J. Symbolic Computation (1999) **11**, 1–35

An Abstract Formalisation of Correct Schemas for Program Synthesis

PIERRE FLENER[†], KUNG-KIU LAU[§], MARIO ORNAGHI[‡] and JULIAN RICHARDSON[¶]

[†] *Department of Information Science, Uppsala University, Box 513, S-751 20 Uppsala, Sweden*
pierre.flener@dis.uu.se
[§] *Department of Computer Science, University of Manchester, Manchester M13 9PL, U.K.*
kung-kiu@cs.man.ac.uk
[‡] *Dipartimento di Scienze dell'Informazione, Universita' degli studi di Milano, Via Comelico 39/41, 20135 Milano, Italy*
ornaghi@dsi.unimi.it
[¶] *Mathematical Reasoning Group, Division of Informatics The University of Edinburgh, 80 South Bridge, Edinburgh EH1 1HN, U.K.*
julianr@dai.ed.ac.uk

(Received 15 September 1998)

Program schemas should capture not only structured program design principles, but also domain knowledge, both of which are of crucial importance for hierarchical program synthesis. However, most researchers represent schemas as purely syntactic constructs, which can provide only a program template, but not the domain knowledge. In this paper, we take a semantic approach and show that a schema $\mathcal{S}$ consists of a syntactic part, viz. a template T, and a semantic part. Template 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* $\mathcal{F}$, which is the semantic part. $\mathcal{F}$ endows the schema $\mathcal{S}$ with a formal semantics, and enables us to define and reason about its correctness. Naturally, correct schemas can be used to guide the synthesis of correct 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 deployed in practice, e.g., KIDS (Smith, 1990; Smith, 1993; Smith, 1994), DESIGNWARE (Smith, 1996), PLANWARE (Blaine *et al.*, 1998).

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

0747–7171/90/000000 + 00 $03.00/0

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, with the notable exception of Smith (Smith, 1985; Smith, 1990), represent schemas as syntactic (logic) expressions, sometimes augmented by extra-logical features, from which actual programs are obtained by some form of textual substitutions. However, in such a purely syntactic approach, which provides only a pattern of place-holders, the knowledge that is captured by a schema is not formalised, such as the semantics of the template, the semantics of the programs it abstracts, or the interactions between these place-holders. So a template by itself has no guiding power for program synthesis, and the additional knowledge somehow has to be hardwired into the system or person using the template.

Therefore, we take a semantic approach and show that a schema $\mathcal{S}$ consists of a syntactic part, viz. a template T, and a semantic part. Template 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* $\mathcal{F}$ (Lau and Ornaghi, 1994; Lau and Ornaghi, 1997a), which is the semantic part. $\mathcal{F}$ 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 (with respect to its specification) as long as its parameters are correctly computed (with respect to 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 (with respect to its specification). A steadfast program is thus also *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 guide the synthesis of steadfast open logic programs. The benefit of such guidance is a reduced search space, because the synthesiser, at any given moment, only tries to construct a program that fits a chosen schema.

On a wider issue, program schemas have been shown to be useful in a variety of applications, such as proving properties of programs (Manna, 1974), teaching programming to novices (Gegg-Harrison, 1991), guiding manual synthesis (Barker-Plummer, 1992; Dershowitz, 1983; Deville, 1990; Deville and Burnay, 1989), inductive synthesis (Flener and Deville, 1993; Flener, 1995; Flener, 1997; Hamfelt and Fischer Nilsson, 1997; Kodratoff and Jouannaud, 1984; Sterling and Kirschenbaum, 1993; Summers, 1977), and deductive (semi-)automatic synthesis (Blaine *et al.*, 1998; Flener *et al.*, 1997; Flener *et al.*, 1998a; Flener *et al.*, 1998b; Flener and Richardson, 1999; Johansson, 1994; Marakakis and Gallagher, 1994; Smith, 1990; Smith, 1993; Smith, 1994; Smith, 1996) of programs, debugging programs (Gegg-Harrison, 1994), transforming/optimising programs (Büyükyıldız and Flener, 1998; Fuchs and Fromherz, 1992; Huet and Lang, 1978; Richardson and Fuchs, 1998; Vasconcelos and Fuchs, 1996), and so on. Further representation issues have been explored independently of applications in (Chasseur and Deville, 1998; Gegg-Harrison, 1995; Gegg-Harrison, 1997), and surveys have been made in (Flener and Yılmaz, 1999; Smith, 1984).

Whilst we have presented some of the ideas elsewhere, most of the technical details (and examples) in this paper are new. This paper thus gives a complete (though compact) account of our approach to formalising (correct) schemas.

The paper is organised as follows. In Section 2, we give the general picture and highlight the novelty of our approach, by informally defining the syntax, semantics, and correctness of schemas, and outlining how correct schemas can be used in program synthesis. In Section 3, we formalise specification frameworks, as well as their reuse through framework morphisms and framework composition. We do not define a precise system of operations for working with frameworks (such as composing them, and so on), but we just give the kind of semantics that such a system should have in order to apply the theory of steadfastness, and we show some examples. In Section 4, we consider specifications and introduce steadfast programs, i.e., correct programs in frameworks. Such programs can be correctly reused by composing frameworks and thus provide a second level of reuse. In Section 5, we introduce the notion of correct schemas and sketch a proof theory associated with our model-theoretic formalisation, so that we can prove schema correctness and use schemas for program synthesis. Finally, in Section 6, we conclude, discuss related work, and outline future work.

2. Overview

In this section, we give an overview of our approach to defining the syntax and semantics of schemas. We outline a notion of correctness for schemas, and briefly explain how correct schemas can be used in a program synthesis process. The material here will be informal, and largely based on examples. The aim is to give a general but more or less complete picture, and to highlight the novelty, of our approach, at an intuitive level.

2.1. DEFINING SCHEMAS

Our approach to defining schemas is based on a three-tier formalism (with model-theoretic semantics), illustrated in Figure 1.

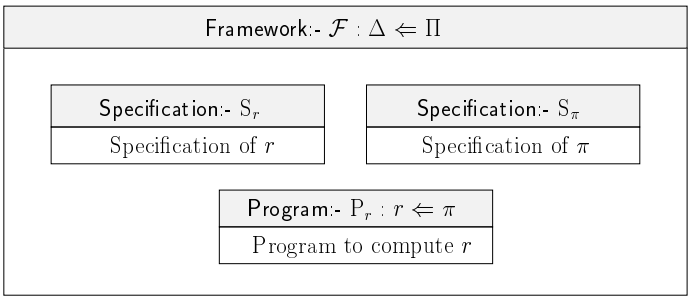

Figure 1. A three-tier formalism for schemas.

In this formalism, at the bottom level, we have *programs*, for computing (specified) relations. Programs are pure (standard or constraint) logic programs. The relations computed by logic programs are called (program) *predicates*. Some predicates may occur only in the body of the clauses of a program. We call such predicates *open predicates*,

and programs that contain such predicates *open programs*. In Figure 1, the program P_r computes a specified relation r in terms of open predicates π.

In the middle, we have *specifications* for defining or specifying new relations (and functions). All program predicates are introduced by specifications. In Figure 1, S_r and S_π are the respective specifications of r and π.

At the top, we have a *specification framework*, or just *framework* for short, that embodies an axiomatisation of (all the relevant knowledge of) the problem domain. The framework provides an unambiguous semantic underpinning for specifications and programs, as well as the correctness relationship between them. A framework may also be open or parametric. In Figure 1, the framework $\mathcal{F}$ has a signature that contains a set Δ of defined symbols (axiomatised in $\mathcal{F}$) and a set Π of parameters.

We define a *schema* to consist of a framework, an open program, called a *template*, and a set of specifications for the predicates of the template.† In the schema in Figure 1, the program P_r is the template.

We need to define a schema as a triple because all three ingredients are necessary for defining the semantics of a schema properly, in order that we can use a schema for the purpose for which it is intended, viz. synthesising programs that have the same computation pattern as the template. Such a semantic characterisation also provides guidelines for the synthesis process. Furthermore, it enables us to define, and reason about, the *correctness* of schemas and of their *reuse*.

EXAMPLE 2.1. Consider the simple template

$$r(x,y) \leftarrow d(x,h,a), rec(a,b), c(h,b,y) \qquad (\mathrm{T}_{dc})$$

for computing r. On its own, we would say this template is meaningless. Nevertheless, our intention is to use it as a generic representation of the typical steps of a divide-and-conquer algorithm: decomposition d, a (possibly empty) sequence *rec* of recursive calls, and composition c. For example, as we will show later in Example 2.2, T_{dc} can be specialised into the following more familiar form of divide-and-conquer:

$$\begin{array}{lcl} r(x,y) & \leftarrow & prim(x), solve(x,y) \\ r(x,y) & \leftarrow & \neg prim(x), dec(x,h,x1,x2), r(x1,y1), r(x2,y2), comp(h,y1,y2,y) \end{array}$$

where $prim(x)$ means that the input x is primitive, i.e. a base case; $dec(x,h,x1,x2)$ means that x can be decomposed into h, $x1$ and $x2$; $comp(h,y1,y2,y)$ means that h and the 'sub-solutions' $y1$ and $y2$ can be composed into the 'solution' y.

However, by itself T_{dc} does not represent any pattern of computation at all. To give it the above intended meaning, we need specifications for the predicates r, d, rec and c.

Specifications define new *specified symbols* in terms of other *specifying symbols*. For example, we could specify r, as follows:

$$\begin{array}{rcl} r & : & [\mathsf{I},\mathsf{O}]; \\ \mathrm{S}_r^c & : & I_r(x) \rightarrow (r(x,y) \leftrightarrow O_r(x,y)). \end{array}$$

Here, the specified symbol is r, with declaration $r : [\mathsf{I},\mathsf{O}]$,‡ while the sort symbols I and

† In this section, for simplicity but without loss of generality, we assume that a schema has only one template.

‡ This means that the arity of r is $\mathsf{I}\times\mathsf{O}$.

O, and the relation symbols I_r and O_r are the specifying symbols. I_r and O_r are called respectively the *input condition* and *output condition* of r.

S_r^c is an example of a form of specification called a *conditional specification* (Lau and Ornaghi, 1997a). Its meaning is the following: for every input x that satisfies the input condition $I_r(x)$, the specified relation $r(x, y)$ is to be true if and only if the output condition $O_r(x, y)$ is true. (We will discuss conditional specifications in Section 4.1.2.)

The meaning of S_r^c is not completely defined, since nothing is stated about the specifying symbols. Our intention is to use this to indicate that the template can be used for deriving programs for a generic, conditionally specified relation. This can be done at the framework level. Since our specifying symbols are generic, we define them as framework *parameters*, i.e., we assume the following fragment of a framework:

Framework $\mathcal{DC}(\mathsf{I}, \mathsf{O}, I_r, O_r)$;
DECLARATIONS:
I_r : $[\mathsf{I}]$;
O_r : $[\mathsf{I}, \mathsf{O}]$.

The sort symbols I and O, as well as the input and output conditions I_r and O_r, are open symbols, i.e. they are *parameters* of the framework $\mathcal{DC}$.

Now since the specifying symbols I_r and O_r in S_r^c are parameters, S_r^c begins to turn T_{dc} into a computation pattern, one for computing a generic (conditionally specified) relation r. S_r^c is also a guide for program synthesis, in the following sense. When we use this schema to synthesise a (correct) divide-and-conquer program from a specification S, S must be a conditional specification, so that we can instantiate the parameter I_r by the input condition of S, and O_r by the output condition of S.

We can further define the computation pattern that T_{dc} and S_r^c together represent, by specifying d. For example, we could specify d (with input condition I_r and output condition O_d) as follows:†

$$\begin{array}{rcl} d & : & [\mathsf{I}, \mathit{List}(\mathsf{I}), \mathit{List}(\mathsf{I})]; \\ \mathrm{S}_d^{sl} & : & I_r(x) \rightarrow (d(x,h,a) \rightarrow O_d(x,h,a)); \\ & & I_r(x) \rightarrow \exists h, a \,.\, d(x,h,a); \end{array}$$

by first expanding the above framework fragment $\mathcal{DC}$ to:

Framework $\mathcal{DC}(\mathsf{I}, \mathsf{O}, I_r, O_r, O_d)$;
IMPORT: $\mathcal{LIST}(\mathsf{I})$;
DECLARATIONS:
I_r : $[\mathsf{I}]$;
O_r : $[\mathsf{I}, \mathsf{O}]$;
O_d : $[\mathsf{I}, \mathit{List}(\mathsf{I}), \mathit{List}(\mathsf{I})]$;

in which $\mathcal{LIST}(\mathsf{I})$ is imported to give meaning to the sort $\mathit{List}(\mathsf{I})$.

The specification S_d^{sl} of d is an example of a *selector specification* (Lau and Ornaghi, 1997a). Its meaning is: for every input x that satisfies the input condition $I_r(x)$, the

† Note that h and a are *lists* of elements of sort I.

specified relation $d(x, h, a)$ is to be true for at least one output (h, a), such that the output condition $O_d(x, h, a)$ holds. (We will discuss selector specifications in Section 4.1.4.)

To ensure that this specification is satisfied, we need to add the following *constraint*† to the framework $\mathcal{DC}$:

$$I_r(x) \rightarrow \exists h, a \,.\, O_d(x, h, a).$$

The input condition for d coincides with that of r because, to compute $r(x, y)$, we first decompose (by d) the input x into two lists $h, a : [List(\mathsf{I})]$ of input values. The idea is that h (possibly) contains values to be used in a non-recursive manner, while r will be recursively applied to the elements of a. So we need to impose that all the elements of a satisfy the input condition I_r. To ensure termination, we also require them to be 'smaller than' x with respect to a well-founded ordering relation $\prec$. Therefore we add the following constraints to the framework $\mathcal{DC}$:‡

$$\begin{array}{l} I_r(x) \wedge O_d(x, h, a) \rightarrow (\forall y \,.\, mem(y, a) \rightarrow I_r(y) \wedge y \prec x); \\ WellFounded(\prec); \end{array}$$

where mem is the usual list membership relation, together with the declaration $\prec$: $[\mathsf{I}, \mathsf{I}]$.

In general, constraints are just axioms, but they play a specific role: we use them to restrict the possible interpretations of the parameters of the framework, in such a way that the template is correct with respect to the specifications. More importantly, they constrain framework composition and specialisation, so as to prevent unsound operations (see Section 3.3).

Now to continue defining the computation pattern represented by the template T_{dc} together with the specifications S_r^c and S_d^{sl}, we shall give S_{rec}^c. To do so, we shall make use of a relation $M(x, y)$, which is introduced by the following explicit definition:

$$M(a, b) \leftrightarrow l(a) = l(b) \wedge \forall x, y, i \,.\, elemi(a, i, x) \wedge elemi(b, i, y) \rightarrow O_r(x, y);$$

where the (overloaded) defining symbols l, $elemi(a, i, x)$ and $elemi(b, i, y)$ are defined in the composite abstract data type (ADT) $\mathcal{LIST}(\mathsf{I}) + \mathcal{LIST}(\mathsf{O})$.§ The function l defines list length, and $elemi(a, i, x)$ means that element x occurs at position i in the list a. Informally, M is similar to the map function of functional languages. For example, $M([x_1, x_2], b)$ holds if and only if $b = [y_1, y_2]$, and $O_r(x_1, y_1)$ and $O_r(x_2, y_2)$ hold.

Now we can specify *rec* as follows:

$$\begin{array}{rcl} rec & : & [List(\mathsf{I}), List(\mathsf{O})]; \\ \mathrm{S}_{rec}^c & : & (\forall x \,.\, mem(x, a) \rightarrow I_r(x)) \rightarrow (rec(a, b) \leftrightarrow M(a, b)). \end{array}$$

This specification says that it is correct to recursively apply r, to compute $M(a, b)$. For example, if $a = [x_1, x_2]$, we can correctly compute $b = [y_1, y_2]$ by the recursive calls $r(x_1, y_1)$ and $r(x_2, y_2)$.

Finally, we specify composition c as follows:

$$\begin{array}{rcl} c & : & [List(\mathsf{I}), List(\mathsf{O}), \mathsf{O}]; \\ \mathrm{S}_c^{gc} & : & I_r(x) \wedge O_d(x, h, a) \wedge M(a, b) \rightarrow (c(h, b, y) \leftrightarrow O_r(x, y)). \end{array}$$

† We mean it in the ordinary sense, not that of constraint programming.

‡ $WellFounded(\prec)$ is of course not first-order. It is the only kind of non-first-order axiom that we will use, and as we will show in Section 5.1, we do not have to prove such axioms anyway.

§ We will discuss composition of ADTs in Section 3.3.

The specification S_c^{gc} of c is a *generalised conditional specification*. It says that $c(h, b, y)$ takes the lists h (computed by $d(x, h, a)$) and b (computed by $rec(a, b)$) as inputs and composes them into a final result y that satisfies the desired output condition $O_r(x, y)$. (We will discuss generalised conditional specifications in Section 4.1.3.)

So, now we have a complete semantic characterisation of a divide-and-conquer schema in which the template is T_{dc}. The complete schema, made up of the framework $\mathcal{DC}$, the specifications we have discussed above, and the template T_{dc}, is:

Schema $\mathcal{DC}(\mathsf{I}, \mathsf{O}, I_r, O_r, O_d, \prec)$;

IMPORT: $\mathcal{LIST}(\mathsf{I}), \mathcal{LIST}(\mathsf{O})$;

DECLARATIONS:

I_r : $[\mathsf{I}]$;
O_r : $[\mathsf{I}, \mathsf{O}]$;
O_d : $[\mathsf{I}, List(\mathsf{I}), List(\mathsf{I})]$;
$\prec$: $[\mathsf{I}, \mathsf{I}]$;
M : $[List(\mathsf{I}), List(\mathsf{O})]$;

AXIOMS:

A1 : $M(a, b) \leftrightarrow l(a) = l(b) \wedge$
$\forall x, y, i \,.\, elemi(a, i, x) \wedge elemi(b, i, y) \rightarrow O_r(x, y)$;

CONSTRAINTS:

C1 : $I_r(x) \rightarrow \exists h, a \,.\, O_d(x, h, a)$;
C2 : $I_r(x) \wedge O_d(x, h, a) \rightarrow (\forall y \,.\, mem(y, a) \rightarrow I_r(y) \wedge y \prec x)$;
C3 : $WellFounded(\prec)$;
C4 : $I_r(x) \wedge O_d(x, h, a) \wedge O_r(x, y) \rightarrow \exists b \,.\, M(a, b)$;

SPECIFICATIONS:

r : $[\mathsf{I}, \mathsf{O}]$;
S_r^c : $I_r(x) \rightarrow (r(x, y) \leftrightarrow O_r(x, y))$;
d : $[\mathsf{I}, List(\mathsf{I}), List(\mathsf{I})]$;
S_d^{sl} : $I_r(x) \rightarrow (d(x, h, a) \rightarrow O_d(x, h, a))$;
$I_r(x) \rightarrow \exists h, a \,.\, d(x, h, a)$;
rec : $[List(\mathsf{I}), List(\mathsf{O})]$;
S_{rec}^c : $(\forall x \,.\, mem(x, a) \rightarrow I_r(x)) \rightarrow (rec(a, b) \leftrightarrow M(a, b))$;
c : $[List(\mathsf{I}), List(\mathsf{O}), \mathsf{O}]$;
S_c^{gc} : $I_r(x) \wedge O_d(x, h, a) \wedge M(a, b) \rightarrow (c(h, b, y) \leftrightarrow O_r(x, y))$;

TEMPLATE:

$r(x, y) \quad \leftarrow \quad d(x, h, a), rec(a, b), c(h, b, y).$

Constraints C1, C2 and C3 have been explained. C4 has been introduced to guarantee the correctness of the template. Correctness analysis can be performed by the proof methods introduced in (Flener *et al.*, 1998a; Lau *et al.*, 1999), and indeed in this case it reveals that C4 is required (we omit the details here). The intuitive meaning of C4 is the following: let x be an input that satisfies the input condition and has been correctly decomposed into h and a (i.e., $I_r(x) \wedge O_d(x, h, a)$ holds); then whenever an output y that satisfies the output condition $O_r(x, y)$ exists, the recursive map $M(a, b)$ must hold for at least one b, needed to compute y by the final composition $c(h, b, y)$.

This schema is a very generic one for divide-and-conquer. It can be specialised into divide-and-conquer schemas with arbitrary numbers of base cases, step cases and recursive calls. We will illustrate this with two examples.

A specialisation of $\mathcal{DC}$ is obtained by making some open symbols of the framework less generic, for instance O_d. For each specialisation, we need to supply a decomposition program for d, a recursion map for rec, and a composition program for c. These programs must be correct with respect to their specialised specifications in $\mathcal{DC}$, so that they correctly compose with the template T_{dc}. Of course for each specialisation of $\mathcal{DC}$, the new specifications will also provide a guide for the synthesis of programs that have the same computation pattern as the template.

EXAMPLE 2.2. Now we show a schema, which is an instance of $\mathcal{DC}$, with one base case and one step case.

Suppose we specialise the specification of decomposition d as follows:

$$\begin{array}{lcl} O_d(x,h,a) & \leftrightarrow & (O_{prim}(x) \wedge h = [x] \wedge a = [\,]) \vee \\ & & (\neg O_{prim}(x) \wedge a = [x_1, x_2] \wedge O_{dec}(x,h,x_1,x_2)); \end{array}$$

where $O_{prim}(x)$ and $O_{dec}(x,h,x_1,x_2)$ are the output conditions of $prim(x)$ and $dec(x,h,x_1,x_2)$. Informally, $prim(x)$ means that x is primitive, i.e. a base case; $dec(x,h,x_1,x_2)$ means that x can be decomposed into h, x_1 and x_2.

A program for $d(x,h,a)$ correct with respect to this specification is:

$$\begin{array}{rcl} d(x,[x],[\,]) & \leftarrow & prim(x) \\ d(x,h,[x1,x2]) & \leftarrow & \neg prim(x), dec(x,h,x1,x2) \end{array}$$

where $prim$ is correct with respect to the conditional specification:

$$I_r(x) \rightarrow (prim(x) \leftrightarrow O_{prim}(x));$$

and dec with respect to the selector specification:

$$\begin{array}{lcl} I_r(x) \wedge \neg O_{prim}(x) & \rightarrow & (dec(x,h,x_1,x_2) \rightarrow O_{dec}(x,h,x_1,x_2)); \\ I_r(x) \wedge \neg O_{prim}(x) & \rightarrow & \exists x_1, x_2, h \,.\, dec(x,h,x_1,x_2). \end{array}$$

If we compose this correct program for d with T_{dc}, we get:

$$\begin{array}{rcl} r(x,y) & \leftarrow & prim(x), rec([\,],b), c([x],b,y) \\ r(x,y) & \leftarrow & \neg prim(x), dec(x,h,x1,x2), rec([x1,x2],b), c(h,b,y). \end{array}$$

Using the specification for rec this becomes:

$$\begin{array}{rcl} r(x,y) & \leftarrow & prim(x), c([x],[\,],y) \\ r(x,y) & \leftarrow & \neg prim(x), dec(x,h,x1,x2), r(x1,y1), r(x2,y2), c(h,[y1,y2],y). \end{array}$$

It is easy to see that by using suitable specifications for $solve$ and $comp$, we can transform this program into the more familiar one for divide-and-conquer with one base case and one step case:

$$\begin{array}{rcl} r(x,y) & \leftarrow & prim(x), solve(x,y) \\ r(x,y) & \leftarrow & \neg prim(x), dec(x,h,x1,x2), r(x1,y1), r(x2,y2), comp(h,y1,y2,y) \end{array}$$

Note that in the step case, there are two recursive calls to r.

We can also get an instance of $\mathcal{DC}$ with one base case and two step cases.

EXAMPLE 2.3. By specialising the template over the data type of natural numbers,† we can obtain the template:

$$\begin{array}{lcl} r(0,y,z) & \leftarrow & c1(y,z) \\ r(s(x),y,z) & \leftarrow & sum(v,v,s(x)), r(v,y,w), c2(0,y,w,z) \\ r(s(x),y,z) & \leftarrow & sum(v,v,x), r(v,y,w), c2(s(0),y,w,z). \end{array}$$

The derivation of this template is given later, in Example 5.5.

As illustrated by Examples 2.2 and 2.3, the template T_{dc} is very generic. In its instances, the number of recursive calls (to r) is arbitrary, a being a list and *rec* being a map of the relation r. Equally, the number of base and step cases is arbitrary. This overcomes the rigidity normally associated with schemas that are purely syntactic structures, where the numbers of recursive calls, as well as base and step cases, are pre-determined.

Finally, it is worth reiterating that all three ingredients of a schema, viz. framework, specifications and template, are indispensable for defining the schema, as the above examples have illustrated.

2.2. CORRECTNESS OF SCHEMAS AND THEIR REUSE

In our three-tier formalism for schemas, *correctness* is the adhesive that glues framework, specifications and template together. It is defined model-theoretically, using the notion of *steadfastness* (Lau *et al.*, 1999). Steadfastness is a correctness property of open programs (e.g. templates) in classes of interpretations (those of the specifications, in the context of the framework) that can be both *composed* and *inherited*. It is thus suitable for defining the correctness of schema templates, and hence the correctness of schemas.

Having a notion of correctness for schemas allows us to reuse schemas at the three levels of frameworks, specifications and templates, and to reason about the correctness of such reuse.

As Example 2.1 suggests, frameworks are our first level of reuse. We can reuse frameworks by (a) specialising them, by adding new axioms and/or new symbols; and (b) composing them, according to their constraints. When the framework of a schema is specialised into a new one, the axioms, theorems and correct template of the schema are inherited, and hence reused. The same happens when we compose the frameworks of two schemas: the composed schema inherits from the component schemas.

Thus, after we have specialised a schema or composed it with another one, we get a schema with a new framework, containing new axioms and/or symbols. Using this new, richer framework, we can synthesise programs for some specifications of predicates that are open in the inherited template. In the synthesis process, we can reuse both the specifications and the template. To see this, let p be a predicate, with specification S_p, of a template T. There are two cases:

(a) In the richer framework, we already have a correct program P for S_p. In this case, we can correctly compose T and P, i.e., we have correct reuse at the template level (and this synthesis sub-task stops successfully).
(b) If (a) does not hold, then we can try to transform S_p into a new specification that is more suited to the new richer knowledge, i.e., we have reuse at the specification

† Constructed from 0 and the *successor* function s.

level if we succeed in finding such a transformation. The specification specialisations used in Examples 2.2 and 2.3 are the kinds of specification transformations that will be explained in Section 5. Once we have a satisfactory specification we continue the synthesis process iteratively.

Reuse at the framework level is based on operations on frameworks. At this level, our approach is similar to that of algebraic ADTs, and correctness is not meaningful, since building a framework is a modelling process, whereby an abstract model of a problem domain or the abstract data types involved in a computation pattern are set up, typically using predefined building blocks. In contrast, correctness of reuse is a key requirement at the specification and template levels. We believe that correct reuse at these levels, made possible by our notion of correct schemas, is new, and important, and yields a powerful mechanism for deriving correct programs.

2.3. USING CORRECT SCHEMAS FOR PROGRAM SYNTHESIS

Correct schemas can be used to synthesise correct programs for a given problem domain. We view the program synthesis task as problem solving (where the problem domain is formalised as a framework) and the program synthesis process as a problem reduction process whereby the synthesis task is successively sub-divided until the sub-tasks can be solved (the sub-solutions are then composed into a solution for the top-level synthesis task).

The program synthesis task is specified in the problem domain by a specification S_r of a relation r to be computed. The synthesis process starts by choosing a schema $\mathcal{S}'$ that contains a template $T_{r'}$ for computing some relation r', specified by $S_{r'}$, such that by renaming or specialising[†] the schema $\mathcal{S}'$ into $\mathcal{S}$ we can 'match' S_r and $S_{r'}$ (and the template $T_{r'}$ becomes a template T_r for r). The synthesis process then consists of iterative attempts to synthesise programs for the predicates in the body of the template T_r from their specifications in $\mathcal{S}$. As programs are synthesised, and as sub-tasks are generated, the template will be updated, so at any moment in time, there is a 'current' template that has evolved from the original template T_r. We shall denote the 'current' template simply by T, and the corresponding 'current' schema $\mathcal{S}(T)$.

In each iteration of the synthesis process, for a predicate p in the body of the template T, if we can find an existing program Q which is correct with respect to a specification S_q, such that S_q can be transformed into the specification S_p of p (through the operations explained in Section 5), then Q is also correct with respect to S_p, and we can (re)use the program Q for p, and the sub-task is solved.

Otherwise, we look for a predefined schema $\mathcal{S}''$ with a template T'' for computing a predicate q with a specification S_q. If S_q can be transformed into the specification S_p, then we import the schema $\mathcal{S}''$ into the 'current' schema $\mathcal{S}(T)$, and add to $\mathcal{S}(T)$ the specifications of any predicates in the body of the template T''. These new open predicates correspond to the sub-problems generated by the sub-solution that $\mathcal{S}''$ represents.

EXAMPLE 2.4. We can import into the basic schema $\mathcal{DC}$ the following schema for computing the map relation, after (possible) renamings and constraint checking:

[†] We will deal with such framework morphisms in Section 3.2.

Schema $\mathcal{MAP}(\mathsf{I}, \mathsf{O}, I_r : [\mathsf{I}], O_r : [\mathsf{I}, \mathsf{O}])$;

IMPORT: $\mathcal{LIST}(\mathsf{I}), \mathcal{LIST}(\mathsf{O})$;

AXIOMS:

$$\begin{array}{rcl} \text{A1} & : & M(a,b) \leftrightarrow l(a) = l(b) \wedge \\ & & \forall x, y, i\,.\ elemi(a,i,x) \wedge elemi(b,i,y) \rightarrow O_r(x,y); \end{array}$$

SPECIFICATIONS:

$$\begin{array}{rcl} r & : & [\mathsf{I}, \mathsf{O}]; \\ \mathrm{S}^c_r & : & I_r(x) \rightarrow (r(x,y) \leftrightarrow O_r(x,y)); \\ map & : & [List(\mathsf{I}), List(\mathsf{O})]; \\ \mathrm{S}^c_{map} & : & (\forall x\,.\ mem(x,a) \rightarrow I_r(x)) \rightarrow (map(a,b) \leftrightarrow M(a,b)); \end{array}$$

TEMPLATE:

$$\begin{array}{lcl} map([\,],[\,]) & \leftarrow & \\ map([x|a],[y|b]) & \leftarrow & r(x,y), map(a,b). \end{array}$$

In this case things are simple, since we immediately recognise that it is sufficient to rename *map* by *rec*, to get a correct open program for *rec*. The general case will be discussed in Section 5, where we will also show that correctness is preserved by schema composition. This simple example shows how schemas can be reused, together with their templates, by being imported into other schemas.

If no schema can be found, or if we prefer to choose a more specific pattern, then we can try to specialise the current template, in the way we specialise the basic schema $\mathcal{DC}$ (in Example 2.1) in Examples 2.2 and 2.3 (and later in Example 5.5).

The iterative synthesis process stops successfully if and when we have synthesised programs for all the predicates in the body of the template T_r, as well as such predicates in all templates imported during the sub-tasks. In the absence of success, we have to backtrack.

Finally, it is worth noting that we can apply the same process to transform a schema into a family of more specialised schemas. In this case we halt the process whenever we have reached a satisfactory specialisation. In Examples 2.2 and 2.3 (and 5.5), we have stopped the specialisation process after just one step. It may also happen that, during some synthesis process, some new interesting specialisation gets constructed. In this case it can be saved as a new predefined schema for future use.

3. Specification Frameworks

As we have shown in the previous section, a specification framework is the context where the specification language and the meaning of the specifying symbols are provided, together with the general laws for reasoning about specifications and program correctness. In this section, we formalise specification frameworks, as well as their reuse through framework morphisms and framework composition.

3.1. A FORMALISATION OF SPECIFICATION FRAMEWORKS

In our approach, the *specifying symbols* are symbols of a many-sorted first-order signature Σ, formalised as a pair $\Sigma = \langle S, D \rangle$ that contains a set S of *sort symbols* and a set

D of *function* and *relation declarations*. A function declaration has the form $f : a \to s$, where f is the declared function symbol, a its arity[†] and s its sort; and a relation declaration has the form $r : a$, where r is the declared relation symbol and a its arity. Function declarations with empty arity introduce constants. Arbitrary overloading is allowed, so that the union of two signatures can be defined as the signature containing the unions of the sorts and of the declarations, and (since we will work with first-order logic with identity) overloaded identity $= : [s, s]$ (for every sort s) will be always understood.

EXAMPLE 3.1. A signature for the domain of planar figures and their areas can be built by importing the signature of reals (which we omit here for conciseness):

Signature *FIGURES*;
IMPORT: *REALS*;
SORTS: *Figs*;
DECLS: $area : [Figs] \to Reals$;
$\cup : [Figs, Figs] \to Figs$;
$separated : [Figs, Figs]$.

Here, overloaded identities $= : [Reals, Reals]$ and $= : [Figs, Figs]$ are understood.

The meaning of specifying symbols is given by a chosen class $\mathcal{I}$ of Σ-*interpretations*. As usual, a Σ-interpretation maps every sort symbol s into a set s^{i}, each constant declaration $c : [\,] \to s$ into an element $(c : [\,] \to s)^{\mathsf{i}} \in s^{\mathsf{i}}$, each function declaration $f : a \to s$ into a function $(f : a \to s)^{\mathsf{i}} : a^{\mathsf{i}} \to s^{\mathsf{i}}$,[†] and each relation declaration $r : a$ into a relation $(r : a)^{\mathsf{i}} \subseteq a^{\mathsf{i}}$. We interpret declarations instead of symbols, because overloading is allowed.

EXAMPLE 3.2. We will consider the following interpretation fig of the signature *FIGURES*:

$Figs^{\mathsf{fig}}$	:	regions of the plane delimited by closed lines, or finite unions of such regions;
$(area : [Figs] \to Reals)^{\mathsf{fig}}$	:	area of a figure;
$(\cup : [Figs, Figs] \to Figs)^{\mathsf{fig}}$	:	union of two figures;
$(separated : [Figs, Figs])^{\mathsf{fig}}$	:	$separated(x, y)$ holds if the (possible) common points of figures x and y belong to their borders.

The interpretation of the imported reals is the usual one, and the (understood) overloaded $=$ is interpreted as the standard identity.

From the signature Σ, we generate the (first-order) *specification language* L_Σ. Σ-formulas are built and interpreted (in a Σ-interpretation) in the standard way. Some care is needed though, due to arbitrary overloading. If an overloaded function symbol has two declarations $f : a \to s_1$ and $f : a \to s_2$ with the same arity and different sorts, then to avoid confusion, we will use f_{s_1} to refer to the first declaration, and f_{s_2} to refer to the second one. In this way, we can associate one declaration with each occurrence of

[†] An arity a is a list $[s_1, \ldots, s_n]$ of sort symbols.
[†] If $a = [s_1, \ldots, s_n]$, then a^{i} is $s_1^{\mathsf{i}} \times \ldots \times s_n^{\mathsf{i}}$.

a function or relation symbol in a formula, and interpret it according to that declaration, in an unambiguous way.

Finally, in a specification framework, the laws for reasoning about specifications and program correctness are given by a Σ-*axiomatisation* Ax, i.e., a set of Σ-sentences, such that the chosen interpretations $\mathcal{I}$ are models of Ax, or $\mathcal{I} \models Ax$. In general $\mathcal{I}$ will be a subset of all the models of Ax. We call $\mathcal{I}$ the *intended interpretations* of the framework.

EXAMPLE 3.3. We will consider the following axioms $Ax(FIGURES)$ for $FIGURES$:

$$\begin{array}{lcl} idempotence & : & \forall x : Figs \,.\, x \cup x = x; \\ commutativity & : & \forall x, y : Figs \,.\, x \cup y = y \cup x; \\ associativity & : & \forall x, y, z : Figs \,.\, (x \cup y) \cup z = x \cup (y \cup z); \\ additivity & : & \forall x, y : Figs \,.\, separated(x, y) \rightarrow area(x \cup y) = area(x) + area(y). \end{array}$$

The intended interpretation is fig, which indeed is a model of the axioms, but there are other models that are completely unrelated to fig. For example, if we interpret *Figs* as any domain containing sets of reals and closed under union, $\cup$ as set union, $separated(x, y)$ as empty intersection, and *area* as the sum of the elements of a set, then we get another model of the axioms. In other words, we have a *loose axiomatisation* of the intended interpretations.

Now we can define specification frameworks formally as follows:†

DEFINITION 3.1. (SPECIFICATION FRAMEWORKS) A *specification framework* $\mathcal{F} = \langle \Sigma, \mathcal{I}, Ax \rangle$ is composed of a signature Σ, a set $\mathcal{I}$ of intended Σ-interpretations, and a set Ax of axioms, such that $\mathcal{I} \models Ax$. $\mathcal{F}$ is *closed* if $\mathcal{I}$ contains just one interpretation; it is *open* if $\mathcal{I}$ contains many interpretations.

An example of a closed framework (with a loose axiomatisation) is $\mathcal{FIG} = \langle FIGURES, \mathsf{fig}, Ax(FIGURES) \rangle$, where the signature $FIGURES$, the intended interpretation fig, and the axioms $Ax(FIGURES)$ are those in Examples 3.1, 3.2 and 3.3.

A particular kind of closed specification frameworks are closed *ADT-frameworks*, for axiomatising Abstract Data Types. The intended interpretation of a closed ADT-framework is a *reachable isoinitial model* (Bertoni *et al.*, 1983), or more precisely a (unique) isoinitial *term-model*.

EXAMPLE 3.4. The ADT-framework for natural numbers is $\mathcal{NAT} = \langle NAT, \mathcal{N}, Ax(NAT) \rangle$, where NAT is the following signature:

Signature NAT;
SORTS: Nat;
DECLS:
$$\begin{array}{rcl} 0 & : & [\,] \rightarrow Nat; \\ s & : & [Nat] \rightarrow Nat; \\ +, * & : & [Nat, Nat] \rightarrow Nat; \end{array}$$

† For conciseness, after this definition we shall refer to specification frameworks simply as frameworks.

and the axioms $Ax(NAT)$ are:

$$\begin{array}{lcl} sax & : & \neg s(x) = 0; \\ & & s(x) = s(y) \rightarrow x = y; \\ +ax & : & x + 0 = x; \\ & & x + s(y) = s(x + y); \\ *ax & : & x * 0 = 0; \\ & & x * s(y) = (x * y) + x. \end{array}$$

The standard structure of natural numbers is an isoinitial model reachable by 0 and s, and we choose the isoinitial term-model of $\mathcal{NAT}$ generated by 0 and s to be the intended interpretation $\mathcal{N}$.

Reachable isoinitial models are similar to the more popular initial models (Goguen and Meseguer, 1987; Goguen *et al.*, 1978), used in algebraic specifications (Sannella and Tarlecki, 1997; Wirsing, 1990). A difference is that, while initial models behave as any other model for positive ground quantifier-free formulas only, isoinitial models do so for any ground quantifier-free formulas, including negation. We choose isoinitial models as intended models, because negation is important for reasoning about specification and correctness.

As for *open* frameworks, we focus our attention on those that are *parametric*:

Definition 3.2. (Parametric Frameworks) A *parametric framework* is an open framework $\mathcal{F}(\Pi) = \langle \Sigma, \mathcal{I}, Ax \rangle$, where (i) Π is a set of symbols in Σ, called *parameters*; (ii) $\mathcal{I}$ is a class of Σ-interpretations, such that, for every pair i_1 and i_2 of interpretations (in $\mathcal{I}$), if the interpretation of the parameters Π is the same for both i_1 and i_2, then $\mathsf{i}_1 = \mathsf{i}_2$.

That is, the parameters can be interpreted in many ways, but any chosen interpretation of the parameters completely determines the interpretation of all the other symbols. For this reason, we call the latter *defined symbols*.

For example, consider the signature *FIGURES* in Example 3.1 enriched by a predicate $basic : [Figs]$, indicating some class of basic figures, for which we can compute the area. Now, for every interpretation of the parameter *basic*, we interpret the sort *Figs* as the subset of the figures that can be generated by finite unions of basic figures, and $\cup$, *area* and *separated* as before. By varying the interpretation of *basic*, we get a parametric framework $\mathcal{FIG}(basic)$ with a loose axiomatisation.

ADT-frameworks can also be parametric. For such frameworks, the intended interpretations are j-reachable j-isoinitial models (Lau and Ornaghi, 1999), where j is a (pre-)interpretation of the parameters Π.

Example 3.5. The ADT-framework for pairs is $\mathcal{PAIR}(X, Y) = \langle PAIR, \mathcal{P}, Ax(PAIR) \rangle$, where *PAIR* is the following signature:

Signature *PAIR*;
SORTS: $Pair(X, Y), X, Y$;
DECLS: $\langle\ \rangle \ : \ [X, Y] \rightarrow Pair(X, Y)$.

For clarity we use the notation $\langle x, y\rangle$ (instead of $\langle\,\rangle(x, y)$). The axioms $Ax(PAIR)$ are:†

$$\begin{array}{rcl} pair & : & \forall x : X, y : Y \,.\, \langle x, y\rangle = \langle x', y'\rangle \rightarrow x = x' \wedge y = y'; \\ inductivepair & : & (\forall x : X \,.\, \forall y : Y \,.\, H(\langle x, y\rangle)) \rightarrow (\forall p : Pair(X, Y) \,.\, H(p)). \end{array}$$

For every interpretation j of the parameters X and Y, the j-models of $Ax(PAIR)$ are the models of $Ax(PAIR)$ that coincide with j over X and Y. The intended (j-isoinitial) j-model corresponding to j is the interpretation $\mathcal{P}_{\mathsf{j}}$ where $Pair(X, Y)$ is the cartesian product $X^{\mathsf{j}} \times Y^{\mathsf{j}}$, and $\langle\,\rangle$ is the usual pairing function. The class $\mathcal{P}$ of intended interpretations is the class of $\mathcal{P}_{\mathsf{j}}$'s.

Finally, as we mentioned in Section 2.2, frameworks are our first level of reuse. The key to their reuse are framework *morphisms* and framework *composition*.

3.2. FRAMEWORK MORPHISMS

Framework morphisms are based on signature morphisms, which we briefly recall here.

For two signatures Σ and Δ, a signature morphism $h : \Sigma \rightarrow \Delta$ maps sorts of Σ into sorts of Δ, and declarations of Σ into declarations of Δ, while preserving arities and (for function declarations) sorts. Morphism h induces a translation $h : L_\Sigma \rightarrow L_\Delta$† and a reduct operation $|h : \Delta$-interpretations $\rightarrow \Sigma$-interpretations. Translation h has a straightforward recursive definition, while the reduct $\mathsf{j}\,|h$ of a Δ-interpretation j interprets each Σ-sort s as $h(s)^{\mathsf{j}}$ and each Σ-declaration d as $h(d)^{\mathsf{j}}$. If $\mathsf{j} = \mathsf{i}\,|h$, then j is called an *h-expansion* of i. In general, there are many h-expansions.

Injective signature morphisms are called *signature expansions*, and bijective signature morphisms are called *signature renamings*. If a signature expansion $h : \Sigma \rightarrow \Delta$ is such that, for every sort symbol or declaration σ of Σ, $h(\sigma) = \sigma$, then $\Sigma \subseteq \Delta$. In this case, h is left implicit, the h-reduct of a Δ-interpretation j is called a Σ-*reduct*, and the h-expansion of a Σ-interpretation i is called a Δ-*expansion*. The Σ-reduct of a Δ-interpretation j just forgets the (interpretation of the) new symbols, and is indicated by $\mathsf{j}\,|\Sigma$.

Finally, for every Σ-sentence F and Δ-interpretation i, the following satisfaction property holds (Goguen and Burstall, 1992):

$$\mathsf{i} \models h(F) \;\text{ iff }\; \mathsf{i}\,|h \models F. \tag{3.1}$$

Now we can define framework morphisms as follows:

DEFINITION 3.3. (FRAMEWORK MORPHISMS) Let $\mathcal{F} = \langle \Sigma, \mathcal{I}, Ax\rangle$ and $\mathcal{G} = \langle \Delta, \mathcal{I}', Ax'\rangle$ be two frameworks. A signature morphism $h : \Sigma \rightarrow \Delta$ is a *framework morphism* from $\mathcal{F}$ to $\mathcal{G}$ if and only if (i) for every $\mathsf{j} \in \mathcal{I}'$, the reduct $\mathsf{j}\,|h$ belongs to $\mathcal{I}$; (ii) for every axiom $A \in Ax$, we have that $Ax' \vdash h(A)$.

By (i), the h-reduct is a map from the intended interpretations of $\mathcal{G}$ to those of $\mathcal{F}$. By (3.1), (i) entails that, for every Σ-sentence F, if $\mathcal{I} \models F$, then $\mathcal{I}' \models h(F)$. In particular, $\mathcal{I}' \models h(A)$, for every $A \in Ax$, i.e., Ax are inherited (under translation). By (ii), we may

† Note that H is first-order because it represents a schema of first-order formulae.

† We use an overloaded h.

change axiomatisations, on condition that the inherited axioms are included or become theorems.

We distinguish three important cases of frameworks morphisms: *refinement*, *expansion* and *specialisation*.

3.2.1. FRAMEWORK REFINEMENT

In a *refinement*, h is injective and every interpretation $\mathsf{i} \in \mathcal{I}$ has at least one h-expansion.

EXAMPLE 3.6. The ADT-framework $\mathcal{PAIR}(X, Y)$ can be *refined* into the ADT-framework $\mathcal{TOPAIR}(X, Y, \leq : [X, X], \leq : [Y, Y])$ of totally ordered pairs by adding to the signature the declarations

$$\begin{array}{lcl} \leq & : & [X, X]; \\ \leq & : & [Y, Y]; \\ \leq & : & [Pair(X, Y), Pair(X, Y)]; \end{array}$$

and to $Ax(PAIR)$ the total ordering axioms for $\leq : [X, X]$ and $\leq : [Y, Y]$, and:

$$\forall a, b : X \,.\, \forall c, d : Y \,.\, \langle a, c\rangle \leq \langle b, d\rangle \leftrightarrow (\neg a = b \wedge a \leq b) \vee (a = b \wedge c \leq d).$$

3.2.2. FRAMEWORK EXPANSION

In an *expansion*, h is injective and $|h$ bijective. By the bijectivity of $|h$, every interpretation $\mathsf{i} \in \mathcal{I}$ has one h-expansion $\mathsf{j} \in \mathcal{I}'$, i.e., h-expansion becomes the inverse function of $|\, h$. In general, the expansion of a framework is defined through the corresponding h-expansion function.

Renaming is a special case of expansion, where h is a signature renaming, and $Ax' = h(Ax)$, i.e., nothing is changed, but the symbols. We can easily see that:

$$\mathcal{I}' = \mathcal{I}\,|\,h^{-1}. \tag{3.2}$$

EXAMPLE 3.7. Consider the ADT-framework $\mathcal{PAIR}(X, Y)$. We have that $\mathcal{PAIR}(A, B)$ is obtained by the signature renaming ρ, where:

$$\begin{array}{lcl} \rho(X) & = & A \\ \rho(Y) & = & B \\ \rho(Pair(X, Y)) & = & Pair(A, B) \\ \rho(\langle\,\rangle) & = & \langle\,\rangle\,. \end{array}$$

For every $\{X, Y\}$-interpretation i, the interpretation $\mathcal{P}_\mathsf{i}$ is mapped (by $|\,\rho^{-1}$) into the interpretation $\mathcal{P}'$ such that:

$$\begin{array}{l} A^{\mathcal{P}'} = X^\mathsf{i} \\ B^{\mathcal{P}'} = Y^\mathsf{i} \\ Pair(A, B)^{\mathcal{P}'} = X^\mathsf{i} \times Y^\mathsf{i} = A^{\mathcal{P}'} \times B^{\mathcal{P}'} \\ \langle\,\rangle^{\mathcal{P}'} = \text{pairing} \end{array}$$

i.e., we have just changed the alphabet of the signature.

Expansion by *explicit definitions* (Lau and Ornaghi, 1997a) of new relation and function declarations is an important ingredient in our approach.

An *explicit* Σ-*definition* of a *relation* $r : a$, where a contains only sorts from Σ, is a $(\Sigma \cup \{r : a\})$-sentence of the form:

$$\forall x \,.\, r(x) \leftrightarrow R(x) \tag{3.3}$$

where x is a tuple of distinct variables with sorts a, and $R(x)$ is a Σ-formula with free variables x.†

For every Σ-interpretation i, there is one $(\Sigma \cup \{r : a\})$-expansion i' of i, such that $\mathsf{i}' \models D_{r:a}$. We call i' the $D_{r:a}$-*expansion* of i.

An *explicit* Σ-*definition* of a *function* $f : a \to s$, where a and s contain only sorts from Σ, is a $(\Sigma \cup \{f : a \to s\})$-sentence of the form:

$$\forall x \,.\, F(x, f(x)) \tag{3.4}$$

where x is a tuple of distinct variables with sorts a, and $F(x, y)$ is a Σ-formula with free variables x and y.

Let i be a Σ-interpretation that satisfies the *obligation*:‡

$$\mathsf{i} \models \forall x \,.\, \exists! y \,.\, F(x, y). \tag{3.5}$$

Then there is one $(\Sigma \cup f : a \to s)$-expansion i' of i such that $\mathsf{i}' \models D_{f:a\to s}$. We call i' the $D_{f:a\to s}$-*expansion* of i.

An explicit Σ-definition D_d of a function or relation declaration d can be used to expand a framework $\mathcal{F} = \langle \Sigma, \mathcal{I}, Ax \rangle$ into the framework $\mathcal{G} = \langle \Sigma \cup \{d\}, \mathcal{I}', Ax \cup \{D_d\} \rangle$, such that $\mathcal{I}'$ is the set of D_d-expansions of the interpretations of $\mathcal{I}$. Of course, if d is a function declaration, we require that the corresponding obligation (3.5) is satisfied by $\mathcal{I}$. $\mathcal{G}$ will be called the D_d-*expansion* of $\mathcal{F}$, and such an expansion will be denoted by $Fxp(\mathcal{F}, D_d)$.

EXAMPLE 3.8. The following relations and functions can be explicitly defined in $\mathcal{NAT}$:

$$\begin{array}{lcl} D_{\leq:[Nat,Nat]} & : & x \leq y \leftrightarrow \exists z \,.\, x + z = y; \\ D_{<:[Nat,Nat]} & : & x < y \leftrightarrow x \leq y \wedge \neg x = y; \\ D_{sqrt:[Nat]\to Nat} & : & sqrt(x) * sqrt(x) \leq x \wedge x < s(sqrt(x)) * s(sqrt(x)). \end{array}$$

Similarly, in the parametric framework $\mathcal{PAIR}(X, Y)$, we can explicitly define projections as follows:

$$\begin{array}{lcl} D_{\pi_1:Pair(X,Y)\to X} & : & \forall p : Pair(X, Y) \,.\, \exists v : Y \,.\, p = \langle \pi_1(p), v \rangle; \\ D_{\pi_2:Pair(X,Y)\to X} & : & \forall p : Pair(X, Y) \,.\, \exists u : X \,.\, p = \langle u, \pi_2(p) \rangle. \end{array}$$

3.2.3. FRAMEWORK SPECIALISATION

Finally, in the third kind of framework morphism, a *specialisation*, h is surjective and | h injective. In this case we can also define an expansion operator, but it is a partial function, i.e., some Σ-interpretations may not have h-expansions. For this reason, we say that we have a specialisation.

† Thus explicit definitions are non-recursive.

‡ We omit sorts whenever no confusion can arise.

EXAMPLE 3.9. Consider the ADT-framework $\mathcal{PAIR}(X, Y)$. We define $\mathcal{PAIR}(A, A)$ as the target of the framework specialisation based on the following signature morphism h:

$$\begin{array}{lcl} h(X) & = & A \\ h(Y) & = & A \\ h(Pair(X, Y)) & = & Pair(A, A) \\ h(\langle\,\rangle) & = & \langle\,\rangle\,. \end{array}$$

For every $\{X, Y\}$-interpretation i, if $X^{\mathsf{i}} \neq Y^{\mathsf{i}}$, then i has no expansion. Otherwise, the interpretation $\mathcal{P}_{\mathsf{i}}$ is mapped into the interpretation $\mathcal{P}'$ such that:

$$\begin{array}{l} A^{\mathcal{P}'} = X^{\mathsf{i}} = Y^{\mathsf{i}} \\ Pair(A, A)^{\mathcal{P}'} = X^{\mathsf{i}} \times Y^{\mathsf{i}} = A^{\mathcal{P}'} \times A^{\mathcal{P}'} \\ \langle\,\rangle^{\mathcal{P}'} = \text{pairing} \end{array}$$

i.e., we have a specialisation to the case where X and Y are (interpreted as) the same domain.

3.3. FRAMEWORK COMPOSITION

Framework composition is performed through two separate operations, namely *union* and *internalisation*.

DEFINITION 3.4. (FRAMEWORK UNION) The *union* of two frameworks $\mathcal{F} = \langle \Sigma, \mathcal{I}, Ax \rangle$ and $\mathcal{G} = \langle \Delta, \mathcal{J}, Ax' \rangle$ is the framework $\mathcal{F} + \mathcal{G} = \langle \Sigma \cup \Delta, \mathcal{I} \bullet \mathcal{J}, Ax \cup Ax' \rangle$ where $\mathcal{I} \bullet \mathcal{J}$ is the set of $(\Sigma \cup \Delta)$-interpretations i such that $\mathsf{i}\,|\,\Sigma \in \mathcal{I}$ and $\mathsf{i}\,|\,\Delta \in \mathcal{J}$.

If the two signatures have common symbols, then $\mathcal{I} \bullet \mathcal{J}$ may be empty. In this case, we say that the union is *inconsistent*. We can easily see that, if the union is consistent, then $\mathcal{I} \bullet \mathcal{J} \models Ax \cup Ax'$, as required in a framework.

By union and renaming or specialisation, we can *compose* frameworks.

EXAMPLE 3.10. If we have a closed framework $\mathcal{INT}$ for integers, with the sort *Int* of integers, we can introduce pairs of integers by the specialisation $\mathcal{PAIR}(Int, Int)$ of $\mathcal{PAIR}(X, Y)$ and by the union $\mathcal{INT} + \mathcal{PAIR}(Int, Int)$.

It is important to give conditions for the consistency of union. To this end, we introduce *constraints*.

DEFINITION 3.5. (CONSTRAINTS) Let $\mathcal{F}(\Pi) = \langle \Sigma, \mathcal{I}, Ax \rangle$ be a framework, and let Γ be a subsignature of Σ containing Π. A Γ-*constraint* for $\mathcal{I}$ is any set *Constrs* of Γ-sentences, such that $\mathcal{I} \mid \Gamma$ are the models of *Constrs*. A Γ-*constrained framework* is a framework $\mathcal{F}(\Pi) = \langle \Sigma, \mathcal{I}, Ax \cup Constrs \rangle$, containing, as a distinguished subset of the axioms, a Γ-constraint *Constrs* for $\mathcal{I}$.

EXAMPLE 3.11. $\mathcal{TOPAIR}(X, Y, \leq : [X, X], \leq : [Y, Y])$ can be given in the form of a $(X, Y, \leq : [X, X], \leq : [Y, Y])$-constrained framework, by putting the total ordering axioms for $\leq : [X, X]$ and $\leq : [Y, Y]$ into the constraint.

To monitor constraint satisfaction, we perform framework composition through *internalisation* and *union*.

Let $\mathcal{F}(\Pi) = \langle \Sigma, \mathcal{I}, Ax \cup Constrs \rangle$ be a Γ-constrained framework, and $\mathcal{G} = \langle \Delta, \mathcal{J}, Ax' \rangle$ be a (possibly closed) framework, such that the common symbols of the two frameworks are only the sort symbols that occur in Γ.† Then we can perform internalisation of the declarations of Γ by $\mathcal{G}$, as follows:

(i) Let $r : a$ be a relation declaration of Γ. Its internalisation in $\mathcal{G}$ is an explicit Δ-definition $\forall x \,.\, r(x) \leftrightarrow R(x)$.
(ii) Let $f : a \to s$ be an open function declaration of Γ. Its internalisation in $\mathcal{G}$ is an explicit Δ-definition $\forall x \,.\, F(x, f(x))$, where $\mathcal{J} \models \forall x \,.\, \exists! y \,.\, F(x, y)$.

DEFINITION 3.6. (INTERNALISATION) Let $\mathcal{F}(\Pi) = \langle \Sigma, \mathcal{I}, Ax \cup Constrs \rangle$ and $\mathcal{G} = \langle \Delta, \mathcal{J}, Ax' \rangle$ be defined as before. If a set D of explicit definitions internalises all the function and relation declarations of Γ, then it is called a Γ-*internalisation*.

A Γ-internalisation D defines one D-expansion $Fxp(\mathcal{G}, D)$ of $\mathcal{G}$. We have the following consistency definition and result:

DEFINITION 3.7. (CONSISTENT Γ-INTERNALISATIONS) Let $\mathcal{F}(\Pi) = \langle \Sigma, \mathcal{I}, Ax \cup Constrs \rangle$ and $\mathcal{G} = \langle \Delta, \mathcal{J}, Ax' \rangle$ be defined as before. A Γ-internalisation D is *consistent* with respect to $\mathcal{F}(\Pi)$ if and only if every interpretation of the D-expansion $Fxp(\mathcal{G}, D)$ of $\mathcal{G}$ is a model of *Constrs*.

THEOREM 3.1. (CONSISTENCY RESULT) *Let* $\mathcal{F}(\Pi) = \langle \Sigma, \mathcal{I}, Ax \cup Constrs \rangle$ *and* $\mathcal{G} = \langle \Delta, \mathcal{J}, Ax' \rangle$ *be defined as before, and* D *be a* Δ*-internalisation. If* D *is consistent with respect to* $\mathcal{F}(\Pi)$*, then* $Fxp(\mathcal{G}, D) + \mathcal{F}(\Pi)$ *is an expansion of* $\mathcal{G}$*, as well as one of* $\mathcal{F}(\Pi)$.

PROOF. Consider an interpretation $\mathsf{j} \in \mathcal{J}$. Since (by the consistency of D) j is a model of *Constrs*, Δ contains Π, and $\mathcal{F}(\Pi)$ is parametric, there is one Σ-interpretation $\mathsf{i} \in \mathcal{I}$ that coincides with j over the symbols of Δ. Therefore, the union contains the interpretation $\mathsf{i} \bullet \mathsf{j}$, which is the unique expansion (in the union) of i, as well as of j. □

The advantage of performing internalisation before doing union is that, after the internalisation steps, we can check constraint satisfaction in $\mathcal{G}$, since we have an expansion of the language of $\mathcal{G}$ containing all the symbols involved in the constraints.

EXAMPLE 3.12. We can compose $\mathcal{TOPAIR}(X, Y, \leq : [X, X], \leq : [Y, Y])$ and the framework $\mathcal{INT}$, which formalises the standard integer type *Int*, by the following algorithm:

(1) Rename or specialise $\mathcal{TOPAIR}(X, Y, \leq : [X, X], \leq : [Y, Y])$, in such a way that the only common symbols are the (possibly renamed) sort symbols of the constraint signature $X, Y, \leq : [X, X], \leq : [Y, Y]$. Here, this is obtained by the specialisation $\mathcal{TOPAIR}(Int, \lhd : [Int, Int])$. Note that *Int* is open, i.e., we have only renamed (specialised) X and Y by *Int*. The translation of the constraint contains the total ordering axioms for $\lhd : [Int, Int]$.

† If this condition does not hold, then we rename $\mathcal{F}(\Pi)$ as appropriate.

(2) Internalise $\lhd : [Int, Int]$ in $\mathcal{INT}$, in such a way that the (translated) constraint, i.e. total ordering axioms for $\lhd$, becomes a theorem. We can obtain this by the internalisation $D_{\lhd} : \ \forall x, y \,.\, x \lhd y \leftrightarrow x \leq y$.
(3) Perform the union $\mathcal{TOPAIR}(Int, \lhd : [Int, Int]) + Fxp(\mathcal{INT}, D_{\lhd})$.

Looking at this example, we can see that we can use composition based on internalisation and union, to implement *parameter passing*. The example corresponds to the parameter passing where X and Y are replaced by Int, and $\leq : [X, X]$ and $\leq : [Y, Y]$ by $\leq : [Int, Int]$. The difference is that, to internalise properly, we have introduced an alias $\lhd$ of $\leq$. By the eliminability of explicit definitions, we could uniformly replace it by $\leq$, if we wished. On the other hand, internalisation can be used in a more flexible way, and, as we will see, the choice of the internalising definitions will also impinge on the synthesis process.

4. Specifications and Correctness

In this section, we consider specifications. They allow us to introduce *steadfast* programs, i.e., *correct programs in frameworks*. Such programs can be correctly reused by composing frameworks, as illustrated in Section 2, and thus they provide us with a second level of reuse.

4.1. SPECIFICATIONS

DEFINITION 4.1. (SPECIFICATIONS) Let $\mathcal{F} = \langle \Sigma, \mathcal{I}, Ax \rangle$ be a framework and δ be a set of relation symbols not in Σ. A Σ-*specification* S_δ of δ is a set of $(\Sigma \cup \delta)$-formulas.

A specification S_δ is interpreted as an expansion operator, in the following way:

DEFINITION 4.2. (S_δ-EXPANSIONS OF INTERPRETATIONS) Let Σ be a signature, and S_δ be a Σ-specification of δ. A S_δ-*expansion* of a class $\mathcal{I}$ of Σ-interpretations is a class $\mathcal{I}'$ of $(\Sigma \cup \delta)$-interpretations such that $\mathcal{I}' \models S_\delta$, and, for every interpretation $\mathsf{i} \in \mathcal{I}$, there is one $(\Sigma \cup \delta)$-expansion $\mathsf{i}' \in \mathcal{I}'$. The set of S_δ-expansions of $\mathcal{I}$ will be denoted by $Ixp(\mathcal{I}, S_\delta)$.

If $Ixp(\mathcal{I}, S_\delta)$ is empty, then S_δ is *inconsistent* with respect to the framework. If $Ixp(\mathcal{I}, S_\delta)$ contains just one expansion, then S_δ is *strict* with respect to the framework. If $Ixp(\mathcal{I}, S_\delta)$ contains more than one expansion, then S_δ is *non-strict*.

Specification symbols have the sole purpose of specifying programs, which are to be synthesised in a framework. To avoid confusion, we will call *framework symbols* the symbols that are defined in a framework and can be used to write down specifications, and *specification symbols* those that are used to specify programs. Thus, specification symbols will be considered to be disjoint from the framework language, and will be designated as *s-symbols*.

There are many kinds of specifications (see e.g. (Lau and Ornaghi, 1997a; Lau and Ornaghi, 1997b)). Here we briefly discuss the more important ones: *explicit definitions*, *super-sub specifications*, *conditional specifications* and *selector specifications*.

Explicit definitions have already been explained in Section 3.2 (see Example 3.8). In many cases, they can be used as specifications as well.

4.1.1. SUPER-SUB SPECIFICATIONS

In a framework $\mathcal{F}(\Pi) = \langle \Sigma, \mathcal{I}, Ax \rangle$, a *super-sub specification* S_r^{ss} of a new relation declaration $r : a$ is a Σ-definition of the form

$$\forall x \,.\, (R_{sub}(x) \rightarrow r(x)) \wedge (r(x) \rightarrow R_{super}(x))$$

where $R_{sub}(x)$ and $R_{super}(x)$ are Σ-formulas. It is consistent if the obligation

$$\mathcal{I} \models \forall x \,.\, R_{sub}(x) \rightarrow R_{super}(x)$$

holds. Its meaning is the following. Let i be a Σ-interpretation of $\mathcal{I}$. Let i_{sub} be the $\forall x \,.\, r(x) \leftrightarrow R_{sub}(x)$-expansion of i and i_{super} be the $\forall x \,.\, r(x) \leftrightarrow R_{super}(x)$-expansion of i. Clearly, every $(\Sigma \cup \{r : a\})$-expansion j of i such that

$$r^{\mathsf{i}_{sub}} \subseteq r^{\mathsf{j}} \subseteq r^{\mathsf{i}_{super}} \tag{4.1}$$

is a model of S_r^{ss}.

Super-sub specifications are very useful, because they have a proof theory (see (Lau and Ornaghi, 1997a)) and many cases can be reduced to them. For example, conditional and generalised conditional specifications are a particular case of super-sub specifications.

4.1.2. CONDITIONAL SPECIFICATIONS

A *conditional specification* S_r^c of a new relation declaration $r : a$, in a framework $\mathcal{F}(\Pi) = \langle \Sigma, \mathcal{I}, Ax \rangle$, is a Σ-definition of the form[†]

$$\forall (I \rightarrow (r(x) \leftrightarrow R))$$

where I and R are Σ-formulas and x is the union of the free variables of I and R. I is called the *input condition*, whereas R is called the *output condition* of the specification. S_r^c is equivalent to the super-sub specification:

$$\forall ((I \wedge R \rightarrow r(x)) \wedge (r(x) \rightarrow \neg I \vee R)).$$

Therefore it is always consistent, but, in general, it is non-strict.

In Example 2.1, S_r^c and S_{rec}^c are examples of conditional specifications.

4.1.3. GENERALISED CONDITIONAL SPECIFICATIONS

A *generalised conditional specification* S_r^{gc} of a new relation declaration $r : a$ is of the form

$$\forall (I \rightarrow (r(x) \leftrightarrow R))$$

where I and R are formulas in the language of $\mathcal{F}(\Pi)$ and their free variables are $x \cup y$, with y non-empty. S_r^{gc} is equivalent to the following super-sub specification:

$$\forall (((\exists y \,.\, I \wedge R) \rightarrow r(x)) \wedge (r(x) \rightarrow (\forall y \,.\, I \rightarrow R))).$$

Therefore, it is consistent if the following obligation holds:

$$\mathcal{I} \models \forall x \,.\, (\exists y \,.\, I \wedge R) \rightarrow (\forall y \,.\, I \rightarrow R).$$

In Example 2.1, S_c^{gc} is an example of a generalised conditional specification.

[†] $\forall(F)$ is the universal closure of F.

4.1.4. SELECTOR SPECIFICATIONS

A *selector specification* S_r^{sl} of a new relation declaration $r : a$ contains two formulas, of the form

$$\forall(I(x) \rightarrow (r(x,z) \rightarrow R));$$
$$\forall(I(x) \rightarrow \exists z\,.\,r(x,z));$$

where x and z are tuples of sorted variables, and the free variables of R belong to x and z.

Selector specifications are consistent under the obligation

$$\mathcal{I} \models \forall(I \rightarrow \exists z\,.\,R).$$

In general, selector specifications are non-strict. For every input x, there may be many (but more than one) outputs y such that $r(x,y)$ holds.

In Example 2.1, S_d^{sl} is an example of a selector specification.

Now we can define program correctness with respect to specifications. We shall use a model-theoretic definition of correctness, based on *steadfastness*.

4.2. STEADFASTNESS AND REUSABLE CORRECT PROGRAMS

If a predicate appears in the head of a clause of a program P, then we say that it is *defined* by P. If it is not defined by P, i.e. it appears only in the body of P's clauses, then we say that it is *open* (in P). The meaning of an open predicate in P is left open by P, along with the meaning of the sort, constant and function symbols in P. In contrast, the meaning of the defined predicates is determined by P in terms of that of the open symbols. To express this dependence more precisely, we introduce the *type* of a program, in the context of a framework $\mathcal{F}(\Pi)$, as follows:

DEFINITION 4.3. (TYPE) A program P has *type* $\delta \Leftarrow \Pi$, written $P : \delta \Leftarrow \Pi$, if δ are the defined predicates of P, and Π is a signature containing the open predicates π and the sort and (constant and) function symbols of P.

Apart from the open predicates, i.e. the (program) parameters π of P (as well as the sort and constant and function symbols of P), Π may also contain symbols that do not occur in P itself, that is, we consider P in the context of the framework $\mathcal{F}(\Pi)$.† For simplicity, the symbols of Π will be called *parameters*, and a Π-interpretation j will be called a *pre-interpretation*. For every pre-interpretation j, program $P : \delta \Leftarrow \Pi$ has a class of j-models, defined as follows:

DEFINITION 4.4. (J-MODELS) Let $P : \delta \Leftarrow \Pi$ be an open program, and j be a pre-interpretation. A j-*model* of P is a model m of P such that $\mathsf{m}\,|\,\Pi = \mathsf{j}$.

j-models have the complete partial ordering $\subseteq_\delta$ defined as follows:

DEFINITION 4.5. Let $P : \delta \Leftarrow \Pi$ be an open program, and i_1 and i_2 be two j-models. Then $\mathsf{i}_1 \subseteq_\delta \mathsf{i}_2$ if and only if, for every defined predicate $r \in \delta$, we have that $r^{\mathsf{i}_1} \subseteq r^{\mathsf{i}_2}$.

† For clarification, the reader may wish to refer to Figure 1 in Section 2.

An open program $P : \delta \Leftarrow \Pi$ has, for every pre-interpretation j, a corresponding *intended* j-model, written j^{P^+}, defined as follows:

DEFINITION 4.6. (MINIMUM J-MODELS) Let $P : \delta \Leftarrow \Pi$ be a program, and j be a Π-interpretation. The *minimum* j-*model* of P is the model j^{P^+} such that $\mathsf{j}^{P^+} \subseteq_\delta \mathsf{m}$, for every j-model m of P.

Now, consider a program $P : \delta \Leftarrow \Pi$, in the context of a class $\mathcal{I}$ of $(\Pi \cup \delta)$-interpretations. Any interpretation $\mathsf{i} \in \mathcal{I}$ contains a pre-interpretation $\mathsf{i}\,|\,\Pi$ of the parameters Π, i.e., $\mathsf{i}\,|\,\Pi$ acts as parameter passing. Thus the minimum $(\mathsf{i}\,|\,\Pi)$-model of P represents the interpretation of δ *defined by* P with parameter passing $\mathsf{i}\,|\,\Pi$. If this interpretation coincides with i, then we can say that P is correct with respect to i. If this happens for *every* $\mathsf{i} \in \mathcal{I}$, then we can say that P is correct with respect to $\mathcal{I}$. Steadfastness is just this kind of *model-theoretic correctness* in a class of interpretations.

DEFINITION 4.7. (STEADFAST LOGIC PROGRAMS) Let $P : \delta \Leftarrow \Pi$ be an open program, and $\mathcal{I}$ be a class of $(\Pi \cup \delta)$-interpretations. Then:

(i) P is *steadfast* in a $(\Pi \cup \delta)$-interpretation i if the minimum $(\mathsf{i}\,|\,\Pi)$-model of P coincides with i, i.e. $(\mathsf{i}\,|\,\Pi)^{P^+} = \mathsf{i}$.
(ii) P is *steadfast* in $\mathcal{I}$ iff it is steadfast in every interpretation $\mathsf{i} \in \mathcal{I}$.

Now we show how correctness with respect to a specification in a framework can be formalised in terms of steadfastness.†

In a framework $\mathcal{F}(\Pi) = \langle \Sigma, \mathcal{I}, Ax \rangle$, programs always satisfy the following requirements:

(i) The sort, constant and function symbols of P are symbols of the signature Σ.
(ii) The predicate symbols of a program P are s-symbols, i.e., they have been introduced by specifications and do not belong to Σ. We will distinguish the specifications S_π of the open predicates of P, and S_δ of the defined ones. Thus, a specification of P in $\mathcal{F}(\Pi)$ will be a pair (S_δ, S_π), and the type of P in this context will be $\delta \Leftarrow (\Sigma \cup \pi)$. Thus the type of P is determined by its specification, and so we need not state it explicitly.

If the specifications S_δ and S_π are strict, then the definition of correctness coincides with that of steadfastness in the unique (S_δ, S_π)-expansion of $\mathcal{F}(\Pi)$. For non-strict specifications, correctness is defined as follows:

DEFINITION 4.8. (CORRECTNESS) Let $\mathcal{F}(\Pi) = \langle \Sigma, \mathcal{I}, Ax \rangle$ be a framework. Let P be an open program with specification (S_δ, S_π). Then P is *correct* in $\mathcal{F}(\Pi)$ with respect to (S_δ, S_π) if and only if, for every S_π-expansion $\mathcal{I}_\pi$ of $\mathcal{I}$ there is an S_δ-expansion $\mathcal{I}_{\pi,\delta}$ of $\mathcal{I}_\pi$ such that P is steadfast in $\mathcal{I}_{\pi,\delta}$.

Intuitively, the meaning of the definition is the following. $P : \delta \Leftarrow (\Sigma \cup \pi)$ is a program to be completed by programs Q for computing the open predicates π. Since we are

† In (Lau and Ornaghi, 1997b; Lau *et al.*, 1999), it is shown that this formalisation is very similar to that in (Deville, 1990).

in an open framework, we may have different Q_{i}'s, for different interpretations $\mathsf{i} \in \mathcal{I}$. Each Q_{i} must be correct with respect to S_π, therefore it 'computes' an S_π-expansion i. Considering all the interpretations $\mathsf{i} \in \mathcal{I}$ and the corresponding expansions computed by the corresponding Q_{i}, we get an S_π-expansion $\mathcal{I}_\pi$. If P is steadfast in an S_δ-expansion $\mathcal{I}_{\delta,\pi}$ of $\mathcal{I}_\pi$, then it correctly composes with every Q_{i}, i.e., it can be correctly reused in the various interpretations of the framework. Correctness requires that this holds for every S_π-expansion $\mathcal{I}_\pi$, to get correct composition with any correct Q_{i}. The following theorem can be proven (Lau *et al.*, 1999):

THEOREM 4.1. (COMPOSITIONALITY OF CORRECTNESS) *Let $\mathcal{F} = \langle \Sigma, \mathcal{I}, Ax \rangle$ be a framework. Let P be correct in $\mathcal{F}$ with respect to $(S_{\delta_1}, S_{\pi_1} \cup S_{\delta_2})$, and Q in $\mathcal{F}$ correct with respect to $(S_{\delta_2}, S_{\pi_2})$. Then $P \cup Q$ is correct in $\mathcal{F}$ with respect to $(S_{\delta_1} \cup S_{\delta_2}, S_{\pi_1} \cup S_{\pi_2})$.*

PROOF. (Outline.) Let i be an interpretation of $\mathcal{I}$, and j be a $(S_{\pi_1} \cup S_{\pi_2})$-expansion of i. Since Q is correct, there is a S_{δ_2}-expansion j_{δ_2} of j, such that Q is steadfast in j_{δ_2}. Since P is correct, there is a S_{δ_1}-expansion $\mathsf{j}_{\delta_1,\delta_2}$ of j_{δ_2}, such that P is steadfast in $\mathsf{j}_{\delta_1,\delta_2}$. Q remains steadfast in the expansion $\mathsf{j}_{\delta_1,\delta_2}$ and, by Lemma 4.1 of (Lau *et al.*, 1999), $P \cup Q$ is steadfast in $\mathsf{j}_{\delta_1,\delta_2}$. Since the above reasoning holds for a generic $(S_{\pi_1} \cup S_{\pi_2})$-expansion j of a generic $\mathsf{i} \in \mathcal{I}$, we have proved the compositionality of correctness. □

This theorem is the basis of (correct) reusability at the level of specifications and (correct) programs. We can also prove the following theorem, which guarantees inheritance of correct programs at the level of framework composition:

THEOREM 4.2. (INHERITANCE OF CORRECTNESS) *Correctness is preserved by framework morphisms and union.*

PROOF. (Outline.) We prove our theorem for framework morphisms. The case of union follows as a corollary. Let $\mathcal{F}_1 = \langle \Sigma_1, \mathcal{I}_1, Ax_1 \rangle$ and $\mathcal{F}_2 = \langle \Sigma_2, \mathcal{I}_2, Ax_2 \rangle$ be two frameworks, $h : \Sigma_1 \to \Sigma_2$ be a framework morphism, and P be a program correct with respect to $(S_{r:a}, S_\pi)$ in $\mathcal{F}_1$.†

We have to prove that $h(P)$ is correct with respect to $(h(S_{r:a}), h(S_\pi))$ in $\mathcal{F}_2$. Let j be a $h(S_\pi)$-expansion of an $\mathsf{i} \in \mathcal{I}_2$. Then $\mathsf{j}\,|\,h \models S_\pi$, i.e., it is a S_π-expansion of $\mathsf{i}\,|\,h$. Since h is a framework morphism, we get $\mathsf{i}\,|\,h \in \mathcal{I}_1$ and, by the correctness of P, there is a $S_{r:a}$-expansion $(\mathsf{j}\,|\,h)_{r:a}$ of $\mathsf{j}\,|\,h$, such that P is steadfast in it. By interpreting $h(r:a)$ as $r:a$ in $(\mathsf{j}\,|\,h)_{r:a}$, we get an expansion $\mathsf{j}_{r:a}$ of j, such that $\mathsf{j}_{r:a}\,|\,h = (\mathsf{j}\,|\,h)_{r:a}$. Thus $\mathsf{j}_{r:a}$ is a $h(S_{r:a})$-expansion of j and a j-model of $h(P)$. We can see that it is also the minimum j-model, i.e., $h(P)$ is steadfast in it. Since the above reasoning holds for a generic $h(S_\pi)$-expansion j of a generic $\mathsf{i} \in \mathcal{I}_2$, we obtain the inheritance of correctness. □

Since the operations we have considered in Section 3 can be reduced to suitable combinations of framework morphisms and to framework union, we can expand, refine, rename, specialise and compose frameworks, while inheriting correct programs. This holds for any system of framework operations that can be explained in terms of morphisms and unions.

† We consider just one single defined predicate $r : a$. The extension to the general case is straightforward.

Inheritance, together with correct reusability at the level of specifications and programs, is the basis of our use of correct schemas, as discussed in the next section.

5. Correct Schemas for Program Synthesis

Using the results of the previous two sections, we now introduce correct schemas, as open frameworks containing a set of specifications together with (open) programs that are correct with respect to these specifications. These programs are called the *templates* of the schema, and correspond to syntactic structures often referred to as program schemas in the literature (see Section 1 for references). Our (correct) schemas are therefore more abstract than such program schemas, and yet they are also more suitable for synthesising (correct) programs.

Since our characterisation of schemas is based on our model-theoretic formalisations of frameworks and correctness, we shall sketch an associated proof theory (which is sound with respect to our schema semantics) in order that we can prove schema correctness, and more importantly, in order that we can use our schemas for program synthesis (which must be based on formal proofs).

5.1. Correct Schemas

Definition 5.1. (Correct Schemas) A *schema* $\mathcal{S} = \langle \mathcal{F}(\Pi), Spec, T \rangle$ is composed of an open constrained framework $\mathcal{F}(\Pi)$, a set $Spec$ of specifications, and a set T of logic programs $P : \delta \Leftarrow \pi$ with specifications (S_δ, S_π) in $Spec$. The programs of T are called the *templates* of the schema. A template $P : \delta \Leftarrow \pi$ is *correct* in $\mathcal{S}$ if it is correct in $\mathcal{F}(\Pi)$ with respect to its specification (S_δ, S_π). The schema $\mathcal{S} = \langle \mathcal{F}(\Pi), Spec, T \rangle$ is *correct* if all the templates of T are correct in $\mathcal{S}$.

We have already shown examples of correct schemas in Section 2 (see Examples 2.1 and 2.2), and we will consider others later (see Example 5.5).

The semantics of correct schemas given by Definition 5.1 is useful, because it allows us to devise suitable associated proof methods. Although these methods are not the main concern of this paper, we need to outline the main underlying ideas in order that we can deal with schema *correctness* and *specialisation*. Specialisation consists in deriving new schemas from a correct schema, by suitable transformations that preserve schema correctness. This generalises the idea of program transformation to schemas and is the basis for *schema reuse*: once we have a schema that has been proved correct, we can specialise it into a family of schemas, while preserving correctness. In the limiting case, specialisation can yield a correct closed program, i.e., we can apply the same methods to schema specialisation and program synthesis.

Furthermore, templates compose correctly, according to Theorem 4.1, and so we can consider the definition of a single relation in a composite template as a single component, whose correctness can be dealt with independently from the other components. Therefore, in a schema, each template is a component with type $P : (r : a) \Leftarrow \pi$, i.e., it contains one defined predicate $r : a$. This view is not restrictive, unless we have mutual recursion, which is not considered in Theorem 4.1. For lack of space, we will not deal with mutually recursive templates. They have essentially the same proof theory, but they require a deeper termination analysis.

Our correctness proofs are based on open completion and open termination (Lau *et*

al., 1999). The *open completion* of a program $P : (r : a) \Leftarrow \pi$ is the *completed definition* (Lloyd, 1987) of r in P. The *if*-part of the completed definition will be called the *positive open completion* of P, and denoted by $Ocomp^+(P)$. The *only-if*-part will be called the *negative open completion* of P, and denoted by $Ocomp^-(P)$.

Open termination is a property of $P : (r : a) \Leftarrow \pi$ in a class $\mathcal{J}$ of pre-interpretations. It has been defined in (Lau *et al.*, 1999), using SLD_E-derivations and SLD_E-failed trees. Intuitively, given a pre-interpretation $\mathsf{j} \in \mathcal{J}$, an SLD_E-derivation in j is a computation of an idealised j-interpreter that knows j.

Our correctness proofs for schemas will be based on the following theorems (which are corollaries of the results in (Lau *et al.*, 1999)):

THEOREM 5.1. *Let $\mathcal{S} = \langle \mathcal{F} = \langle \Sigma, \mathcal{I}, Ax \rangle, Spec, T \rangle$ be a schema, and $P : (r : a) \Leftarrow \pi$ be a template with specification (S_r^{ss}, S_π), where S_r^{ss} is the super-sub specification:*

$$\forall x \,.\, (R_{sub}(x) \rightarrow r(x)) \wedge (r(x) \rightarrow R_{super}(x)).$$

If

(a) $Ax \cup S_\pi \cup \{\forall x \,.\, r(x) \leftrightarrow R_{super}(x)\} \vdash Ocomp^+(P)$;
(b) $Ax \cup S_\pi \cup \{\forall x \,.\, r(x) \leftrightarrow R_{sub}(x)\} \vdash Ocomp^-(P)$;

and P existentially terminates[†] *in every S_π-expansion of $\mathcal{I}$, then P is correct in $\mathcal{S}$.*

PROOF. (Outline.) Since P existentially terminates (in every S_π-expansion of $\mathcal{I}$), it decides $r : a$ (see Theorem 5.7 of (Lau *et al.*, 1999)). Then, by Theorem 6.4 of (Lau *et al.*, 1999), we get the correctness result. □

THEOREM 5.2. *Let $\mathcal{S} = \langle \mathcal{F} = \langle \Sigma, \mathcal{I}, Ax \rangle, Spec, T \rangle$ be a schema, and $P : (r : a) \Leftarrow \pi$ be a template with specification (S_r^{sl}, S_π), where S_r^{sl} is the selector specification*

(sel_1) $\forall (I(x) \rightarrow (r(x, z) \rightarrow R(x, z)))$;
(sel_2) $\forall (I(x) \rightarrow \exists z \,.\, r(x, z))$.

If

(a) $Ax \cup S_\pi \cup \{\forall x, z \,.\, r(x, z) \leftrightarrow (\neg I(x) \vee R(x, z))\} \vdash Ocomp^+(P)$
(b) $Ax \cup S_\pi \cup Ocomp^+(P) \vdash \forall x \,.\, I(x) \rightarrow \exists z \,.\, r(x, z)$

then P is correct in $\mathcal{S}$.

PROOF. (Outline.) Let j be a S_π-expansion of an interpretation $\mathsf{i} \in \mathcal{I}$, j_r be the $(\forall x, z \,.\, r(x, z) \leftrightarrow \neg I(x) \vee R(x, z))$-expansion of j, and j^P be the minimum j-model of P. By (a), $\mathsf{j}^P \subseteq_r \mathsf{j}_r$. Since $r(x, y) \rightarrow (\neg I(x) \vee R(x, y)$ is logically equivalent to (sel_1), we get that $\mathsf{j}^P \models (sel_1)$. By (b), $\mathsf{j}^P \models (sel_2)$. Since this holds for a generic S_π-expansion j of a generic $\mathsf{i} \in \mathcal{I}$, we have proved the theorem. □

Thus, in our correctness proofs, we are interested in existential termination in a class of pre-interpretations. Here we give a sufficient condition for existential termination, that works for a large class of interesting schemas.

[†] In an interpretation, P existentially terminates if for every assignment $\mathbf{a}$ of x, either $r(x)$ is successful, or $r(x)$ is finitely failed for $\mathbf{a}$.

In order to simplify the definition, we consider program clauses where the arguments of predicates (except the equality predicate) are variables only. All clauses can be transformed into this form, by using equality. For example, $r(f(x), y) \leftarrow r(x, g(y)), h(x, u(y))$ can be written as $r(a, y) \leftarrow a = f(x), b = g(y), c = u(y), r(x, b), h(x, c)$.

DEFINITION 5.2. Let $P : (r : a) \Leftarrow \pi$ be an open program with clauses already transformed in the above manner. We say that P is *decreasing* in a pre-interpretation j with respect to argument positions $i_1, \ldots, i_n$ in r if, for every recursive clause $r(\mathbf{x}) \leftarrow B$ of r in P, every assignment $\mathbf{a}$ of the variables of the clause such that $\mathsf{j} \models_{\mathbf{a}} B \setminus r$, where $B \setminus r$ is the set of the equations and open predicates of B, and every recursive call $r(\mathbf{y})$ in the body B, $\langle \mathbf{a}(y_{i_1}), \ldots, \mathbf{a}(y_{i_n}) \rangle \prec \langle \mathbf{a}(x_{i_1}), \ldots, \mathbf{a}(x_{i_n}) \rangle$, where $\prec$ is well-founded in j. We say that P is *decreasing* in a class $\mathcal{J}$ of pre-interpretations with respect to (the argument positions) $i_1, \ldots, i_n$ if it is decreasing with respect to $i_1, \ldots, i_n$ in every $\mathsf{j} \in \mathcal{J}$.

EXAMPLE 5.1. The program:

$$\begin{array}{lll} r(x, a, x) & \leftarrow & a = 0 \\ r(x, a, b) & \leftarrow & a = s(y), b = s(z), r(x, y, z) \end{array}$$

is decreasing with respect to (the argument position) 2 in every interpretation where the relation explicitly defined by $y \prec a \leftrightarrow a = s(y)$ is well founded. In these interpretations, it is also decreasing with respect to 3.

The existence of the well-founded relation $\prec$ allows us to state the following sufficient condition:

THEOREM 5.3. *If a program $P : (r : a) \Leftarrow \pi$ is decreasing in a class $\mathcal{J}$ of pre-interpretations with respect to at least one (non-empty) set of argument positions, then it existentially terminates in $\mathcal{J}$.*

To get decreasing templates, we associate with each recursive template $P : (r : a) \Leftarrow \pi$ a relation $\prec$ such that P is decreasing with respect to some set of argument positions in all the pre-interpretations where $\prec$ is well-founded. If necessary, we force well-foundedness by the constraint $\mathit{WellFounded}(\prec)$. Such constraints will be the only non-first-order statements that we will use in the constraints. However, we will not have to prove them. As we will see, either $\mathit{WellFounded}(\prec)$ is inherited, or $\prec$ is internalised by a relation that is known to be well-founded. In the former case no proof is needed, since the statement belongs to the axioms. In the latter case, $\mathit{WellFounded}(\prec)$ is guaranteed by the internalisation. Finally, when a well-founded relation is declared in a framework, we can assume that the first-order instances of the corresponding induction and descending chain principles implicitly belong to the axioms. This allows, in particular, inductive reasoning over the recursive structure of templates.

By introducing well-founded relations as required in frameworks, we have that existential termination is always guaranteed, either by the constraints, or by their internalisation. Therefore, correctness proofs will be based on the provability of the open completion, according to Theorems 5.1 and 5.2.

EXAMPLE 5.2. We can prove the correctness of the schema $\mathcal{DC}$ in Example 2.1. The constraint $\mathit{WellFounded}(\prec)$ is not strictly needed here, since the template is not recursive.

However, $\mathit{WellFounded}(\prec)$ will be necessary for the existential termination of recursive specialisations. If we do not have this constraint here, we will have to introduce it when we use recursive clauses.

The other constraints are needed to prove the completion. In this case, $\mathit{Ocomp}^+(\mathrm{T}_{dc})$ is (logically equivalent to):

$$\forall x, y, h, a, b \,.\, r(x, y) \leftarrow d(x, h, a) \wedge rec(a, b) \wedge c(h, b, y)$$

and $\mathit{Ocomp}^-(\mathrm{T}_{dc})$ is:

$$\forall x, y \,.\, r(x, y) \rightarrow \exists h, a, b \,.\, d(x, h, a) \wedge rec(a, b) \wedge c(h, b, y).$$

$\mathit{Ocomp}^+(\mathrm{T}_{dc})$ is to be proved using the axioms, the constraints, the specifications of the predicates in the body, and the definition:

$$\forall x, y \,.\, r(x, y) \leftrightarrow (\neg I_r(x) \vee O_r(x, y)).$$

This definition is to be replaced by:

$$\forall x, y \,.\, r(x, y) \leftrightarrow (I_r(x) \wedge O_r(x, y))$$

to prove $\mathit{Ocomp}^-(\mathrm{T}_{dc})$.

Finally, we consider specialisation methods that preserve correctness. To prove correctness preservation, we use our results so far, together with unfolding or correct folding. Of course, the idea is to give general transformation rules that have been proved correct once and for all. Here, we cite just two of them.

EXAMPLE 5.3. The first transformation rule allows us to replace single calls by sequences of calls, in the body of a template. The rule is:

> Let $q(t)$ be a call occurring in the body of a template $P : (r : a) \Leftarrow \pi$, and let $I_q \rightarrow (q(x) \leftrightarrow O_q(x))$. If the internalisation $O_q(x) \leftrightarrow A(x) \wedge B(x)$ satisfies the constraints for O_q, then the call $q(t)$ can be replaced by calls to $a(t)$ and $b(t)$, where a and b are two new predicates specified as follows:
>
> $$\begin{array}{lcl} I_r & \rightarrow & (a(x) \leftrightarrow A(x)); \\ I_r \wedge A(x) & \rightarrow & (b(x) \leftrightarrow B(x)). \end{array}$$

For the correctness of the transformation, we use Theorem 5.1 to prove that the (non-recursive) clause $r(x) \leftarrow a(x), b(x)$ is correct with respect to its specifications. Then, the result follows from correct composability of correct templates and from the fact that unfolding preserves the minimum model semantics.

A similar result holds if r has a selector specification.

EXAMPLE 5.4. The second transformation rule allows us to replace variables with open sorts by tuples, if suitable conditions are satisfied. The definition of the rule requires a detailed recursive definition of a suitable translation, so we omit it here for lack of space and just give an example that shows how the translation works and how it can be proved correct, in a particular case.

Assume that we want to replace x by u and v in the predicate $r(x, y)$ of the $\mathcal{DC}$ schema. To this end, we rename I by $\mathit{Pair}(U, V)$ and we consider the union of the renamed framework with $\mathcal{PAIR}(U, V)$. Then we introduce a new declaration $r' : [U, V, \mathsf{O}]$ and the

specification $I_r(\langle u, v\rangle) \rightarrow (r'(u, v, y) \leftrightarrow O_r(\langle u, v\rangle, y))$. We can easily prove that the template $r'(u, v, y) \leftarrow x = \langle u, v\rangle, r(x, y)$ is correct. By template composition and unfolding, we get $r'(u, v, y) \leftarrow x = \langle u, v\rangle, d(x, h, a), rec(a, b), c(h, b, y)$. By a similar transformation on $d(x, h, a)$, we get the template:

$$r'(u, v, y) \leftarrow d'(u, v, h, a), rec(a, b), c(h, b, y).$$

Of course, the steps that have been performed here manually, should be performed automatically by the rule. When applied to a variable of sort s, the rule checks that s can be consistently instantiated by $Pair(U, V)$, with U and V new sorts. This is guaranteed for any open sort s,† like I in the example.

5.2. USING CORRECT SCHEMAS FOR PROGRAM SYNTHESIS

In this section we briefly explain how schemas can be (re)used in program derivation. As we mentioned earlier, we can use the same methods for specialising a schema to a specific ADT and for deriving a program for solving a given task. In the first case, the result is another, more specific correct schema, whilst in the latter it is a (closed) program.

As we said in Section 2.3, we view the program synthesis process as problem solving by successive problem reduction until the sub-tasks can be solved. Therefore, in program synthesis, we start with a (*problem*) *specification* S_r^{prob} of $r : a$, in the context of a framework $\mathcal{G}$ representing an ADT or a problem domain. Suppose $\mathcal{S}$ is a schema containing a template $P : (r : a) \Leftarrow \pi$ with specification (S_r, S_π), and a framework $\mathcal{F}(\Pi)$. We shall assume that $r : a$ is the same in S_r and in S_r^{prob}, and the sort symbols in the arity a are the only common symbols of $\mathcal{G}$ and $\mathcal{S}$. If this does not hold, then we have to first perform a suitable renaming and (possible) specialisation of the schema. We also assume, for conciseness, that S_r is a conditional specification with input condition I_r and output condition O_r. A program derivation step then has the following form:

(i) We internalise I_r, O_r and the other open symbols of the schema, in the composed framework $\mathcal{F}(\Pi) + \mathcal{G}$. The internalisation should allow us to prove that (S_r, S_π) reduces to (S_r^{prob}, S_π), that is, correctness (in $\mathcal{F}(\Pi) + \mathcal{G}$) with respect to (S_r, S_π) entails correctness with respect to (S_r^{prob}, S_π). A sufficient condition is that $S_r \leftrightarrow S_r^{prob}$ can be proved, but there are other useful sufficient conditions (see (Flener *et al.*, 1997)).

(ii) We try to prove the constraints involving I_r and O_r. The result is that either we can prove a constraint, or we (possibly) simplify some parts of it. In the first case, we can delete the constraint, whilst in the latter, we inherit the simplified constraint.

(iii) We (possibly) transform the template $P : (r : a) \Leftarrow \delta$, to get a better specialised template. The transformation may involve the internalisation of other open symbols, as well as the analysis of appropriate constraints.

Program synthesis may halt with a specialised schema, or with a closed program, i.e., a set of templates where the predicates in the body of a template occur in the head of some other clause. For the latter case, we require that each constraint has been proved.

† For an open sort, any interpretation is allowed, while constrained sorts are open sorts that can be interpreted in a constrained way.

The whole synthesis process is such that, if at each step the framework union $\mathcal{F}(\Pi)+\mathcal{G}$ is consistent (i.e., it has a non-empty class of intended interpretations), then the final framework is consistent, and the program contained in it is correct with respect to the specifications.

Now we close this section with two examples. The first one shows a specialisation of the $\mathcal{DC}$ schema (in Example 2.1) in the ADT of natural numbers, while the second one uses this specialisation to synthesise a closed program.

EXAMPLE 5.5. We can get a specialisation of $\mathcal{DC}$ to natural numbers as follows. We rename I by *Nat*, and then we build the union $\mathcal{NAT}+\mathcal{DC}(Nat, 0, I_r, O_r, O_d, \prec)$. Since we are using natural numbers, we replace the specification of the decomposition predicate $O_d(x,h,a)$ by:

$$\begin{aligned} O_d^{prob}(x,h,a) \quad : \quad & x=0 \wedge h=[0] \wedge a=[\,] \vee \\ & (\exists y,i,v\,.\; x=s(y) \wedge h=[i] \wedge a=[v] \wedge i \leq s(0) \wedge i+v+v=x). \end{aligned}$$

For example, $O_d^{prob}(9,[1],[4])$ holds, because 1+4+4=9. The decomposition implicit in the specification is to compute the integer half of x, and then to apply recursion to it.

Now we proceed to the internalisation phase. To reduce the specification containing O_d to the one containing O_d^{prob}, it suffices to internalise O_d by:

$$O_d(x,h,a) \leftrightarrow O_d^{prob}(x,h,a).$$

Also we internalise $\prec$ by:

$$x \prec y \leftrightarrow (x+x \leq y \wedge \neg y=0).$$

This relation is known to be well-founded, and the length of a chain starting from y is logarithmic in y. Therefore, constraint C3: *WellFounded*$(\prec)$ is satisfied.

We also have to prove constraints involving O_d. We can easily see that C1 is satisfied, for every interpretation of I_r, i.e., with I_r open. We can simplify C2 to:

$$I_r(s(x)) \wedge i \leq s(0) \wedge i+v+v=s(x) \rightarrow I_r(v);$$

and C4 to:

$$I_r(s(x)) \wedge i \leq s(0) \wedge i+v+v=s(x) \wedge O_r(s(x),y) \rightarrow \exists w\,.\; O_r(v,w);$$

and then inherit these simplified constraints.

Now we specialise our templates. We can synthesise in the framework for natural numbers the following correct decomposition program:

$$\begin{aligned} d(0,[0],[\,]) \quad &\leftarrow \\ d(s(y),[0],[v]) \quad &\leftarrow \quad sum(v,v,s(y)) \\ d(s(y),[s(0)],[v]) \quad &\leftarrow \quad sum(v,v,y) \end{aligned}$$

where the predicate *sum* is specified by:†

$$sum(x,y,z) \leftrightarrow z=x+y.$$

So, 0 is the primitive case, and $s(x)$ the non-primitive one. The latter is decomposed

† This is a conditional specification with no input condition.

into a list of one simpler value, to which the recursor *rec* is to be applied. Using the specification of *rec*, we can get:

$$\begin{array}{rcl} r(0,y) & \leftarrow & c([0],[\,],y) \\ r(s(x),y) & \leftarrow & sum(v,v,s(x)), r(v,w), c([0],[w],y) \\ r(s(x),y) & \leftarrow & sum(v,v,x), r(v,w), c([s(0)],[w],y). \end{array}$$

Now, if we introduce the specifications:

$c1(y) \leftrightarrow O_r(0,y)$;
$I_r(x) \wedge \neg x = 0 \wedge i + v + v = x \wedge i \leq s(0) \wedge O_r(v,w) \rightarrow (c2(i,w,y) \leftrightarrow O_r(x,y))$;

we can derive the correct program:

$$\begin{array}{rcl} c([0],[\,],y) & \leftarrow & c1(y) \\ c([i],[w],y) & \leftarrow & c2(i,w,y) \end{array}$$

and we get our final specialised template:

$$\begin{array}{rcl} r(0,y) & \leftarrow & c1(y) \\ r(s(x),y) & \leftarrow & sum(v,v,s(x)), r(v,w), c2(0,w,y) \\ r(s(x),y) & \leftarrow & sum(v,v,x), r(v,w), c2(s(0),w,y). \end{array}$$

In this template, lists have disappeared, and we have obtained a schema for divide-and-conquer for the structure of natural numbers.

Now, the reusability of the schema obtained in this example has the limitation that the input variable $x : Nat$ cannot be replaced by tuples, because the specialisation used in Example 5.4 cannot be applied to the closed sort *Nat*. To get a more general schema, we can specialise $\mathcal{DC}$, by replacing I by a pair (Nat, I), and then apply the specialisation used in Example 5.5. Thus we get the schema:

Schema $\mathcal{DCNAT}(\mathsf{I}, \mathsf{O}, I_r, O_r)$;

IMPORT: $\mathcal{NAT}$;

DECLARATIONS:

I_r : $[Nat, \mathsf{I}]$;
O_r : $[Nat, \mathsf{I}, \mathsf{O}]$;

CONSTRAINTS:

C1 : $I_r(s(x), y) \wedge i \leq s(0) \wedge i + v + v = s(x) \rightarrow I_r(v, y)$;
C2 : $I_r(s(x), y) \wedge i \leq s(0) \wedge i + v + v = s(x) \wedge O_r(s(x), y, z) \rightarrow \exists w \,.\, O_r(v, y, w)$;

SPECIFICATIONS:

r : $[Nat, \mathsf{I}, \mathsf{O}]$;
S^c_r : $I_r(x,y) \rightarrow (r(x,y,z) \leftrightarrow O_r(x,y,z))$;
sum : $[Nat, Nat, Nat]$;
S^c_{sum} : $sum(x,y,z) \leftrightarrow z = x + y$;
$c1$: $[\mathsf{I}, \mathsf{O}]$;
S^c_{c1} : $c1(y,z) \leftrightarrow O_r(0,y,z)$;
$c2$: $[Nat, \mathsf{I}, \mathsf{O}, \mathsf{O}]$;
S^{gc}_{c2} : $I_r(x,y) \wedge \neg x = 0 \wedge i + v + v = x \wedge i \leq s(0) \wedge O_r(v,y,w) \rightarrow (c2(i,y,w,z) \leftrightarrow O_r(x,y,z))$;

TEMPLATE:

$$\begin{array}{lcl} r(0,y,z) & \leftarrow & c1(y,z) \\ r(s(x),y,z) & \leftarrow & sum(v,v,s(x)), r(v,y,w), c2(0,y,w,z) \\ r(s(x),y,z) & \leftarrow & sum(v,v,x), r(v,y,w), c2(s(0),y,w,z). \end{array}$$

Finally, as an example of a synthesis of a closed program, we use this schema to synthesise a program for the product of natural numbers.

EXAMPLE 5.6. We start from the following problem specification:

$$prod(x,y,z) \leftrightarrow z = x * y.$$

To apply the schema $\mathcal{DCNAT}$, we replace I and O by Nat, and we rename r by $prod$, I_r by I_{prod}, and O_r by O_{prod}. Then we internalise O_{prod} by $O_{prod}(x,y,z) \leftrightarrow z = x * y$ and I_{prod} by $I_{prod}(x,y) \leftrightarrow true$. In this way, the problem specification becomes equivalent to that in the schema.

Now we have to check the constraints. They are satisfied, as we can easily see.

As a final step, we eliminate the explicit definition of O_{prod} in the specifications of $c1$ and $c2$. We get

$$\begin{array}{l} c1(y,z) \leftrightarrow z = 0 * y; \\ \neg x = 0 \wedge i + v + v = x \wedge i \leq s(0) \wedge w = v * y \rightarrow (c2(i,y,w,z) \leftrightarrow z = x * y); \end{array}$$

that is, $c1(y,z) \leftrightarrow z = 0$ and $c2(i,y,w,z) \leftrightarrow (i = 0 \wedge z = w+w) \vee (i = s(0) \wedge z = w+w+y)$. From the specifications, we can get the final template:

$$\begin{array}{lcl} prod(0,y,0) & \leftarrow & \\ prod(s(x),y,z) & \leftarrow & sum(v,v,s(x)), prod(v,y,w), sum(w,w,z) \\ prod(s(x),y,z) & \leftarrow & sum(v,v,x), prod(v,y,w), sum(w,w,u), sum(y,u,z). \end{array}$$

To get a final closed program, we need to synthesise a program for sum. We could also transform the program, by specifying the predicate $half(w,x) \leftrightarrow w + w = x$, and synthesising a program for $half$.

6. Conclusion

In our work, we use specification frameworks (i.e. open first-order axiomatisations) to formalise program schemas (containing templates) in order to ensure that the templates (and the programs that are instances thereof) do indeed have the behaviour we intend them to have. In this paper, we have shown an abstract formalisation of a correct program schema as a specification framework containing specifications (of predicates), and templates which are open logic programs (containing the specified predicates) that are correct with respect to the specifications. We have also outlined how we can use such correct schemas to synthesise correct logic programs. Our work is very strongly influenced by Smith's pioneering work (Smith, 1985; Smith, 1990; Smith, 1996) in applicative programming since the mid 1980s.

In contrast to most approaches (with the exception of Smith's) to schemas in the literature, which regard schemas as purely syntactic constructs (which therefore do not capture domain knowledge), our approach defines schemas as semantic entities with both a model theory and a proof theory. The model theory allows us to define a suitable notion of *schema correctness*, which in turn provides a formalisation of *correct schema reuse*.

The proof theory enables us not only to prove schema correctness for any given schema but, more importantly, also to use schemas to guide program synthesis (which must be based on formal proofs).

Schema correctness is based on the consistency of frameworks and the correctness of the templates with respect to the specifications of the predicates in the templates. This correctness leads to correct schema reuse at all three levels of frameworks, specifications and templates. Framework reuse occurs via framework operations such as specialisation and composition. Specification reuse mainly takes the form of specification transformation. Template reuse can be effected by importing schemas into other schemas. It is the correct reuse at these three levels that makes our semantic characterisation of schemas unique. More importantly, it defines correct schema reuse (for program synthesis).

Specifications also play an important role in guiding program synthesis. Different forms of specifications have different associated proof methods, and these methods provide guidance on how to proceed at various stages during synthesis.

For frameworks, we have shown examples of closed and open parametric frameworks, with loose axiomatisations (with many diverse models) and ADT-axiomatisations (with unique isoinitial term-models). However, we have not explained the details of isoinitial models, which are their intended models, because this is not the central issue of this paper (although we have explained that we use these models in order to deal with negation). Indeed, correctness can be based on any class of intended interpretations. This is important, since it allows the use of loose axiomatisations, which in our opinion cannot be avoided if we want to deal with real programs.

Since we concentrate on the semantics of schemas, we do not define a precise system of operations on frameworks, but we just give the kind of semantics needed to apply the theory of correctness (based on steadfastness). Any such system for frameworks (i.e. theories) that can be interpreted according to this semantics (in particular, by framework morphisms) will work. In particular, we can apply the metatheory developed in the field of algebraic ADTs (Marti-Oliet and Meseguer, 1996).

So far we have only the basis for a proof theory, and our future work will include the development of a specialised proof theory for schema correctness and schema specialisation. This will in turn provide the foundations for implementing a system based on our ideas.

Another important future objective is to identify templates and constraints for other design methodologies than divide-and-conquer. Once again, Smith (Smith, 1990) has shown the way, namely by capturing a vast class of search methodologies in a global-search schema. This was adapted to constraint logic programming in (Flener *et al.*, 1998b).

Finally, a few words about the consistency of frameworks. We use consistency preserving operations and formal correctness proofs, starting from predefined and well understood ADT frameworks and schemas. This guarantees consistency and formal correctness. Of course, the correctness of formal specifications with respect to informal requirements cannot be guaranteed. To partially remedy this, we allow frameworks with loose axiomatisations of their intended interpretations. Although we cannot automatically check mistakes and inconsistencies, we can have some partial checks, and this is surely a better alternative than any completely informal approach of translating (informal) requirements into (formal) specification by ADT frameworks and schemas.

ACKNOWLEDGEMENTS

We wish to thank the referees and the editor in charge of the reviews, for their helpful comments and suggestions.

References

D. Barker-Plummer (1992). Cliché programming in Prolog. In M. Bruynooghe, editor, *Proc. META'90*, pages 246–256.

A. Bertoni, G. Mauri, and P. Miglioli (1983). On the power of model theory in specifying abstract data types and in capturing their recursiveness. *Fundamenta Informaticae* **VI**(2), 127–170.

L. Blaine, L. Gilham, J. Liu, D.R. Smith, and S. Westfold (1998). PLANWARE: Domain-specific synthesis of high-performance schedulers. In D.F. Redmiles and B. Nuseibeh, editors, *Proc. ASE'98*, pages 270–279. IEEE Computer Society Press.

H. Büyükyıldız and P. Flener (1998). Generalised logic program transformation schemas. In N.E. Fuchs, editor, *Proc. LOPSTR'97*, *LNCS* **1463**, 46–65. Springer-Verlag.

E. Chasseur and Y. Deville (1998). Logic program schemas, semi-unification and constraints. In N.E. Fuchs, editor, *Proc. LOPSTR'97*, *LNCS* **1463**, 69–89. Springer-Verlag.

N. Dershowitz (1983). *The Evolution of Programs*. Birkhäuser.

Y. Deville (1990). *Logic Programming: Systematic Program Development*. Addison-Wesley.

Y. Deville and J. Burnay (1989). 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.

P. Flener (1995). *Logic Program Synthesis from Incomplete Information*. Kluwer.

P. Flener (1997). Inductive logic program synthesis with DIALOGS. In S. Muggleton, editor, *Proc. ILP'96*, *LNAI* **1314**, 175–198. Springer-Verlag.

P. Flener and Y. Deville (1993). Logic program synthesis from incomplete specifications. *J. Symbolic Computation* **15**(5–6), 775–805.

P. Flener, K.-K. Lau, and M. Ornaghi (1997). Correct-schema-guided synthesis of steadfast programs. In Y. Ledru and M. Lowry, editors, *Proc. ASE'97*, pages 153–160. IEEE Computer Society Press.

P. Flener, K.-K. Lau, and M. Ornaghi (1998a). On correct program schemas. In N.E. Fuchs, editor, *Proc. LOPSTR'97*, *LNCS* **1463**, 124–143. Springer-Verlag.

P. Flener, H. Zidoum, and B. Hnich (1998b). Schema-guided synthesis of constraint logic programs. In D.F. Redmiles and B. Nuseibeh, editors, *Proc. ASE'98*, pages 168–176. IEEE Computer Society Press.

P. Flener and J. Richardson (1999). A unified view of programming schemas and proof methods. In A. Bossi, editor, *Pre-Proc. LOPSTR'99*. TR, University of Venice.

P. Flener and S. Yılmaz (1999). Inductive synthesis of recursive logic programs: Achievements and prospects. *J. Logic Programming* **41**(2–3), 141–195.

N.E. Fuchs and M.P.J. Fromherz (1992). Schema-based transformation of logic programs. In T. Clement and K.-K. Lau, editors, *Proc. LOPSTR'91*, pages 111–125. Springer-Verlag.

T.S. Gegg-Harrison (1991). Learning Prolog in a schema-based environment. *Instructional Science* **20**, 173–190.

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

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

T.S. Gegg-Harrison (1997). Extensible logic program schemata. In J. Gallagher, editor, *Proc. LOPSTR'96*, *LNCS* **1207**, 256–274. Springer-Verlag.

J.A. Goguen and R.M. Burstall (1992). Institutions: Abstract model theory for specification and programming. *J. ACM* **39**(1), 95–146.

J.A. Goguen and J. Meseguer (1987). 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.

J.A. Goguen, J.W. Thatcher, and E. Wagner (1978). An initial algebra approach to specification, correctness and implementation. In R. Yeh, editor, *Current Trends in Programming Methodology, IV*, pages 80–149. Prentice-Hall.

A. Hamfelt and J. Fischer Nilsson (1997). Towards a logic programming methodology based on higher-order predicates. *New Generation Computing* **15**(4), 421–448.

G. Huet and B. Lang (1978). Proving and applying program transformations expressed with second-order patterns. *Acta Informatica* **11**, 31–55.

A.-L. Johansson (1994). Interactive program derivation using program schemata and incrementally generated strategies. In Y. Deville, editor, *Proc. LOPSTR'93*, pages 100–112. Springer-Verlag.

Y. Kodratoff and J.-P. Jouannaud (1984). Synthesizing LISP programs working on the list level of embedding. In A.W. Biermann, G. Guiho, and Y. Kodratoff, editors, *Automatic Program Construction Techniques*, pages 325–374. Macmillan.

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

K.-K. Lau and M. Ornaghi (1997a). Forms of logic specifications: A preliminary study. In J. Gallagher, editor, *Proc. LOPSTR'96*, *LNCS* **1207**, 295–312. Springer-Verlag.

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

K.-K. Lau and M. Ornaghi. OOD frameworks in component-based software development in computational logic. In P. Flener, editor, *Proc. LOPSTR'98*, *LNCS* **1559**, 101–123, Springer-Verlag, 1999.

K.-K. Lau, M. Ornaghi, and S.-Å. Tärnlund (1994). The halting problem for deductive synthesis of logic programs. In P. van Hentenryck, editor, *Proc. ICLP'94*, pages 665–683. MIT Press.

K.-K. Lau, M. Ornaghi, and S.-Å. Tärnlund (1999). Steadfast logic programs. *J. Logic Programming* **38**(3), 259–294, 1999.

J.W. Lloyd (1987). *Foundations of Logic Programming*. Springer-Verlag, 2nd edition.

Z. Manna (1974). *Mathematical Theory of Computation*. McGraw-Hill.

E. Marakakis and J.P. Gallagher (1994). Schema-based top-down design of logic programs using abstract data types. In L. Fribourg and F. Turini, editors, *Proc. LOPSTR/META'94*, *LNCS* **883**, 138–153. Springer-Verlag.

N. Marti-Oliet and J. Meseguer (1996). Inclusions and subtypes II: Higher-order case. *J. Logic Comput.* **6**(4), 541–572.

J. Richardson and N.E. Fuchs (1998). Development of correct transformational schemata for Prolog programs. In N.E. Fuchs, editor, *Proc. LOPSTR'97*, *LNCS* **1463**, 263–281. Springer-Verlag.

D. Sannella and A. Tarlecki (1997). Essential concepts of algebraic specification and program development. *Formal Aspects of Computing* **9**, 229–269.

D.R. Smith (1984). The synthesis of LISP programs from examples: A survey. In A.W. Biermann, G. Guiho, and Y. Kodratoff, editors, *Automatic Program Construction Techniques*, pages 307–324. Macmillan.

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

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

D.R. Smith (1993). Constructing specification morphisms. *J. Symbolic Computation* **15**(5–6), 571–606.

D.R. Smith (1994). Towards the synthesis of constraint propagation algorithms. In Y. Deville, editor, *Proc. LOPSTR'93*, pages 1–9. Springer-Verlag.

D.R. Smith (1996). Toward a classification approach to design. *Proc. AMAST'96*, *LNCS* **1101**, 62–84. Springer-Verlag.

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

P.D. Summers (1977). A methodology for LISP program construction from examples. *J. ACM* **24**(1), 161–175.

W.W. Vasconcelos and N.E. Fuchs (1996). An opportunistic approach for logic program analysis and optimisation using enhanced schema-based transformations. In M. Proietti, editor, *Proc. LOPSTR'95*, *LNCS* **1048**, 174–188. Springer-Verlag.

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