Formal Development of Correct Classes in Computational Logic

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

Chris Moss
Department of Computing, Imperial College
180 Queen's Gate, London SW7 2BZ, United Kingdom
cdsm@doc.ic.ac.uk

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

Abstract

Quality software must be reusable, extensible and reliable. In computational logic , we have developed an approach to constructing programs that are formally correct. Our approach can provide a basis for constructing software that is (formally) reusable and extensible, and not just reliable but formally correct. In this paper, we explain our notion of correct classes, and how to develop them.

1 Introduction

Quality software must be reusable, extensible (or maintainable), and reliable. Object-oriented programming is seen as a suitable programming paradigm for constructing such software. It achieves reusability and extensibility by the use of classes (of objects) with inheritance. Reliability can be gained by verifying assertions which correspond to invariants on classes and pre-post-conditions on their methods.

In computational logic, we have developed an approach to constructing programs that are formally correct (wrt their specifications). Our approach is declarative in the sense that both specifications and programs have declarative (logical) semantics. So we do not have classes of objects with internal states. However, we have classes that are parameterised modules, i.e. generic classes. In addition, we can define and reason about the correctness and reusability of methods (programs) as well as classes. Thus our approach can provide a basis for constructing software that is (formally) reusable and extensible, and not just reliable but formally correct. In this paper, we explain our notion of correct classes, and how to develop them.

For lack of space, we shall assume some familiarity with basic logic programming terminology, as in e.g. [10].

2 Overview of Our Approach to Logic Program Synthesis

Our approach to program synthesis in computational logic has the following key components (more details can be found in [6, 8]):

*Part of this work was done during the first author's visit to Computing Science Department, Uppsala University, Sweden, supported by the European Union HCM project on Logic Program Synthesis and Transformation, contract no. 93/414. He wishes to thank Jonas Barklund for his invitation and hospitality.

- A (full) first-order theory that axiomatises the problem domain, called a *framework* $\mathcal{F}$.

  $\mathcal{F}$ has a model-theoretic semantics, i.e. it has an intended model I. (We will discuss frameworks in Section 3.)

- Two kinds of specifications:

  – *explicit specifications* of new relations and functions in $\mathcal{F}$ (see Example 3.1);

  – *partial specifications* of relations for synthesis purposes only (see Example 4.10).

  These specifications are first-order formulas, and they also have a model-theoretic semantics, i.e. they are interpreted in the intended model I of $\mathcal{F}$. An explicit specification has a unique interpretation, whereas a partial specification has many possible interpretations. The former is used to add new symbols to $\mathcal{F}$, as well as for synthesis of course; whilst the latter is more flexible for synthesis purposes, but it may not be used to introduce new symbols in $\mathcal{F}$.

- *Correctness criteria* for programs wrt their specifications, i.e. specifications of the relations they compute, in $\mathcal{F}$.

  These criteria are also model-theoretic. In a framework $\mathcal{F}$, a logic program P (that computes a specified relation r) has an intended model, viz. its minimum Herbrand model H; and we say P is correct wrt an explicit specification S (of r) iff the interpretation of r in the intended model H of P coincides with (or is at least isomorphic to) its interpretation in the intended model I of $\mathcal{F}$.

  If S is a partial specification, then r's interpretation in H must coincide with one of its interpretations in I.

  Of course for proving correctness, we use proof-theoretic versions of these correctness criteria.

- A *synthesis process* that employs synthesis methods to derive correct programs from their specifications. (A survey of such methods can be found in [1].)

Typically an explicit specification of a new relation in $\mathcal{F}$ takes the form:[1]

$$\forall\, (r(x) \leftrightarrow R(x))$$

where $R(x)$ is a formula of the language of $\mathcal{F}$. The synthesis process employs some synthesis method to iteratively generate a (possibly infinite) sequence of programs

$$P_1 \subseteq P_2 \subseteq \cdots \subseteq P_k \subseteq \cdots$$

where, for every P_j, there is a corresponding set S_j of specifications of the new relation symbols of P_j.
The synthesis method must ensure that each P_j is *partially correct*, i.e.:

$$\mathcal{F} \cup S_j \vdash Comp^+(P_j)$$

where $Comp^+$ is (logically equivalent to) the *if*-part of the completion $Comp(P)$ of P, and that P_j terminates.
The synthesis process halts successfully, when P_j becomes *totally correct*, i.e.:

$$\mathcal{F} \cup S_j \vdash Comp(P_j)$$

and that P_j terminates.[2] Note that we shall use $Comp^-(P)$ to denote the *only-if*-part of $Comp(P)$. Note also that since partial correctness is guaranteed in each synthesis step, total correctness can be proved in a synthesis step if

$$\mathcal{F} \cup S_j \vdash Comp^-(P_j)$$

This provides a useful criterion for determining when to halt the synthesis process with a totally correct program.
To illustrate our approach, here is a very simple example in $\mathcal{NAT}$, the framework of Peano arithmetic (which is defined in Example 3.1):

[1] $\forall(F)$ denotes F's universal closure.
[2] Our current synthesis process does not guarantee termination of P_j.

Example 2.1 Consider the specification

$S_{prod}: \ \forall\,(prod(x,y,z) \leftrightarrow x \cdot y = z)$

A synthesis process (using a completely manual synthesis method) would be the following:

1. Synthesise a clause for $prod(x,0,z)$. In $\mathcal{NAT} \cup S_{prod}$ we can prove:

   $$prod(x,0,z) \leftrightarrow x \cdot 0 = z \leftrightarrow z = 0$$

   Thus $\mathcal{NAT} \cup S_{prod} \vdash Comp^{+}(prod(x,0,z) \leftarrow z = 0)$ and we get the program P_1:

   $$prod(x,0,0) \leftarrow$$

2. Next synthesise a clause for $prod(x,s(y),z)$. In $\mathcal{NAT} \cup S_{prod}$ we can prove:

   $$prod(x,s(y),z) \leftrightarrow x \cdot s(y) = z \leftrightarrow x \cdot y + x = z \leftrightarrow \exists w\,(prod(x,y,w) \wedge w + x = z)$$

   From this we could synthesise the clause $prod(x,s(y),z) \leftarrow prod(x,y,w), w + x = z$. Next introduce a new relation sum:

   $$S_{sum}: \ \forall(sum(x,y,z) \leftrightarrow x + y = z)$$

   and we get $S_2 = \{S_{prod}, S_{sum}\}$ and we have

   $$\mathcal{NAT} \cup S_2 \vdash Comp^{+}(prod(x,s(y),z) \leftarrow prod(x,y,w), sum(w,x,z))$$

   That is our new program P_2 is:

   $$\begin{array}{rcl} prod(x,0,0) & \leftarrow & \\ prod(x,s(y),z) & \leftarrow & prod(x,y,w), sum(w,x,z) \end{array} \tag{1}$$

   Now P_2 is not totally correct: we can prove $Comp^{-}(prod,P_2)$ (i.e. the only-if part of the completed definition of $prod$ wrt P_2), but not $Comp^{-}(sum,P_2)$. This means that we can stop synthesis for $prod$, but not for sum.

3. $\mathcal{NAT} \cup S_2 \vdash Comp^{+}(sum(x,0,x))$, so synthesise P_3 containing this new clause, i.e. P_3 is:

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

   Continue synthesis because $\mathcal{NAT} \cup S_2 \vdash \neg Comp^{-}(sum,P_3)$.

4. $\mathcal{NAT} \cup S_2 \vdash Comp^{+}(sum(x,s(y),s(w)) \leftarrow sum(x,y,w))$, so synthesise P_4 containing this new clause, i.e. P_4 is:

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

   Now the synthesis process successfully halts because $\mathcal{NAT} \cup S_2 \vdash Comp^{-}(sum,P_4)$, i.e. P_4 (terminates and) is totally correct.

The key result of our work so far is that our approach extends to *open*, or *parameterised*, frameworks: in such frameworks we can synthesise programs that are *steadfast*, i.e. *open* programs that are always totally correct whenever their parameters are instantiated correctly. We will elaborate on this in Section 4.2. Steadfastness thus provides a basis for modularity, reusability, as well as formal correctness.

3 Parameterised Modules as Generic Classes

As we have just seen, in our approach to logic program synthesis, correct programs are developed in the context of a framework $\mathcal{F}$ which is a first-order theory. The use of $\mathcal{F}$ marks the main difference between our approach and others. $\mathcal{F}$ not only provides an unambiguous underpinning of the semantics of specifications and programs (and hence the latter's correctness wrt the former), but, as we will show in this section, $\mathcal{F}$ itself can also be an integral part of a parameterised module, or a generic class in object-oriented terminology, that can be used for modular program development.

We define a parameterised module as follows:

Definition 3.1 A *parameterised module* M consists of:

- a framework $\mathcal{F}$ with parameters Π;
- specifications spec_1, spec_2, ..., (possibly with parameters from Π);
- programs prog_1, prog_2, ..., with parameters $\pi_1, \pi_2, \ldots$

$\mathcal{F}$ is a full first-order logical theory. Specifications are certain kinds of formulas in the language of $\mathcal{F}$. Programs are either standard or constraint logic programs that have been synthesised from their specifications in $\mathcal{F}$.

A parameterised module M can be visualised as follows:[3]

Framework:- $\mathcal{F} : \Delta \Leftarrow \Pi$

Program:- $\mathrm{prog}_1 : \delta_1 \Leftarrow \pi_1$	Program:- $\mathrm{prog}_2 : \delta_2 \Leftarrow \pi_2$	...
Program 1	Program 2	

The outer box denotes the framework $\mathcal{F}$, while the inner boxes denote logic programs that have been synthesised in $\mathcal{F}$.

A framework $\mathcal{F}$ basically axiomatises a problem domain, and typically contains ADT's as well as other properties (e.g. theorems) that hold in the problem domain. $\mathcal{F}$ may be open or parameterised, i.e. it may contain parameters Π, and it defines a set Δ of symbols in terms of Π. Π are called *framework parameters*. These parameters may be instantiated by sort, function or relation symbols that have already been defined elsewhere (in another framework). In each program prog_i, δ_i stands for the set of symbols that are defined in prog_i (i.e. they appear in the clause heads of prog_i) in terms of the parameters π_i. π_i are called *program parameters*. Program parameters may be just any relation symbols in $\mathcal{F}$ that are not yet (completely) defined, or of course they may be relation symbols from Π.

To be more precise, a *closed* framework $\mathcal{F}$ contains

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

  For example, in Example 3.1, $Comp(sum, P_4)$ and $Comp(prod, P_4)$ are theorems, and to prove them we have (implicitly) used well-known theorems of Peano arithmetic.

An *open* framework $\mathcal{F}$ with parameters Π will contain in addition:

- parameter axioms, that state assumptions about the framework parameters Π.

[3] We omit the specifications for simplicity.

Frameworks are usually constructed incrementally, possibly by using previously defined frameworks. Typically a framework $\mathcal{F}$ axiomatises a new abstract data type T by making use of, or *importing*, pre-defined types. The signature of $\mathcal{F}$ contains the old sorts and a sort symbol for the new data type T. The type T is constructed from *constructors*, which are declared as functions. The axioms of the pre-defined data types are imported into $\mathcal{F}$, and new axioms are added to define new functions and relations on T.

Example 3.1 The following open framework axiomatises the theory of lists with parametric element type $Elem$ and partial ordering relation $\lhd$:[4]

$$
\begin{array}{lll}
\textbf{Framework } \mathcal{LIST}(Elem, \lhd); & & \\
\text{IMPORT:} & \mathcal{NAT}\ ; & \\
\text{SORTS:} & Nat, Elem, List; & \\
\text{FUNCTIONS:} & nil \quad : \quad \to List; & \\
 & \quad . \quad\ \ : \quad (Elem, List) \to List; & \\
 & nocc \ \ : \quad (Elem, List) \to Nat; & \\
\text{RELATIONS:} & elemi \ \ : \quad (List, Nat, Elem); & \\
 & \quad \ldots & \\
\text{AXIOMS:} & \neg nil = a.B \wedge (a_1.B_1 = a_2.B_2 \to a_1 = a_2 \wedge B_1 = B_2)); & \\
 & nocc(x, nil) = 0; & \\
 & a = b \to nocc(a, b.L) = nocc(a, L) + 1; & \\
 & \neg a = b \to nocc(a, b.L) = nocc(a, L); & \\
 & elemi(L, 0, a) \leftrightarrow \exists B\,(L = a.B); & \\
 & elemi(L, s(i), a) \leftrightarrow \exists b, B\,(L = b.B \wedge elemi(B, i, a)); & \\
 & \ldots & \\
\text{P-AXIOMS:} & x \lhd y \wedge y \lhd x \leftrightarrow x = y; & \\
 & x \lhd y \wedge y \lhd z \to x \lhd z &
\end{array}
$$

where $\mathcal{NAT}$ is the following framework for Peano arithmetic:

$$
\begin{array}{ll}
\textbf{Framework } \mathcal{NAT}; & \\
\text{SORTS:} & Nat; \\
\text{FUNCTIONS:} & 0 \quad : \quad \to Nat; \\
 & s \quad : \quad Nat \to Nat; \\
 & +, * \ : \quad (Nat, Nat) \to Nat; \\
\text{AXIOMS:} & \forall x\,(\neg 0 = s(x)) \wedge \forall x, y\,(s(x) = s(y) \to x = y); \\
 & \forall x\,(x + 0 = x); \\
 & \forall x, y\,(x + s(y) = s(x + y)); \\
 & \forall x\,(x * 0 = 0); \\
 & \forall x, y\,(x * s(y) = x + x * y); \\
 & \forall (H(0) \wedge \forall i\,(H(i) \to H(s(i))) \to \forall x\, H(x));
\end{array}
$$

In $\mathcal{LIST}$, external universal quantifiers have been omitted, variables of sort $Elem$ are in lower-case, and variables of sort $List$ are in upper-case. $\mathcal{LIST}$ is essentially the standard axiomatisation of lists. It makes use of, i.e. it imports, the (closed) framework $\mathcal{NAT}$ for Peano arithmetic. The function $nocc(a, L)$ gives the number of occurrences of a in L, and the relation $elemi(L, i, a)$ means a occurs at position i in L. The p-axioms are the parameter axioms. In this case, the p-axioms state that $\lhd$ must be a partial ordering.

Here we have given only part of the complete axiomatisation of $\mathcal{LIST}$. Other relations and functions can be defined in terms of symbols already defined here. For example, we can define the usual relations len, $concat$, $perm$, and ord,

[4] We will write $\mathcal{LIST}(Elem, \lhd)$ instead of $\mathcal{LIST} : \Delta \Leftarrow Elem, \lhd$ whenever Δ is not relevant.

for *length*, *concatenation*, *permutation*, and *ordered* respectively, as well as the length and catenation functions l and $|$ as follows:

$$
\begin{array}{lcl}
len(L,n) & \leftrightarrow & \forall i(i < n \leftrightarrow \exists a(elemi(L,i,a))) \\
n = l(L) & \leftrightarrow & len(L,n) \\
concat(A,B,L) & \leftrightarrow & \forall i,a\,(i < l(A) \rightarrow (elemi(A,i,a) \leftrightarrow elemi(L,i,a)))\wedge \\
 & & \forall j,b\,(elemi(B,j,b) \leftrightarrow elemi(L,j+l(A),b)) \\
L = A|B & \leftrightarrow & concat(A,B,L) \\
perm(A,B) & \leftrightarrow & \forall e(nocc(e,A) = nocc(e,B)) \\
ord(L) & \leftrightarrow & \forall i(elemi(L,i,e_1) \wedge elemi(L,s(i),e_2) \rightarrow e_1 \leq e_2)
\end{array}
\tag{2}
$$

Note also that the last axiom in $\mathcal{NAT}$ is an induction schema. This is useful for reasoning about properties of $+$ and $*$ which cannot be derived from the other axioms, e.g. associativity and commutativity. In $\mathcal{LIST}$, we can also use induction on lists. This illustrates the fact that in a framework we may have more than just ADT definitions.

The syntax of a framework is thus similar to that of an algebraic ADT (e.g. [3, 16]). However, whereas an algebraic ADT is an *initial model* of its specification, the intended model of a closed framework is an *isoinitial model*, and an open framework has a class of isoinitial models.

Definition 3.2 An *isoinitial model* I of $\mathcal{F}$ is a reachable model, i.e. a model where each element (of the domain) can be represented by a ground term, such that for any relation r defined in $\mathcal{F}$, ground instances $r(t)$ or $\neg r(t)$ are true in I iff they are true in all models of $\mathcal{F}$.

Like initial semantics (e.g. [3]), isoinitial semantics also guarantees 'no junk' and 'no confusion', but has the additional capability to handle negation, which is important in database applications.

Example 3.2 The isoinitial model of the closed framework $\mathcal{LIST}(Int,<)$, i.e. the framework $\mathcal{LIST}(Elem,\lhd)$ with the parameters $Elem$ and $\lhd$ instantiated to the integer type Int and $<$ respectively, is the usual list structure (viz. the term model generated by the constructors $\cdot$ and nil of $List$) for integers, with the ordering $<$.[5]

Now a parameterised module can be regarded as a generic class of ADT's. As illustrated in Figure 1, such a module

$$
\begin{array}{rcccc}
\text{(Parameterised) Module} & = & \text{Framework} & + & \text{Programs} \\
\downarrow & & \downarrow & & \downarrow \\
\text{(Generic) Class} & = & \text{ADT} & + & \text{Methods}
\end{array}
$$

Figure 1: Parameterised modules as generic classes

contains ADT's as well as programs or methods that work on these ADT's, and is therefore a generic class in object-oriented terminology (e.g. [15, 12]). (It is, however, not a class of objects with internal states.) However, in contrast to classes in object-oriented programming, which usually have a type-theoretic characterisation, our classes have a model-theoretic semantics.

Furthermore, in a class M, the programs also have intended models, viz. their minimum Herbrand models. Therefore, we can define program correctness in a model-theoretic manner in a class. As a result, we can use model-theoretic semantics to reason about the correctness of classes.

4 Correct Classes

Reasoning about the correctness of a parameterised module M has to be done at two levels: *framework* and *program* levels. M can be composed with other modules, to yield *instances* of M containing instances of the framework $\mathcal{F}$ and

[5] $<$ is a (strict) total ordering, hence it is also a partial ordering, as required by the p-axioms.

the programs prog_i. In each instance of M, at the framework level, $\mathcal{F}$ is composed with another framework that defines Π; whilst at the program level, each program prog_i (and its specification spec_i) inherits the new Π. Correctness of M means the *consistency* of (each instance of) $\mathcal{F}$, and the *correctness* of (each instance of) each prog_i wrt its specification spec_i in (each instance of) $\mathcal{F}$, i.e. the consistency of $\mathcal{F}$ and the correctness of each prog_i in each instance of M.

We will use only parameterised modules that are correct in this sense, and we will show how to construct such modules.

Definition 4.1 A *correct* parameterised module consists of:[6]

- an *adequate* framework $\mathcal{F}$ with parameters Π;
- *steadfast* (logic) programs prog_1, prog_2, with parameters $\pi_1, \pi_2, \ldots$

We will define adequate frameworks and steadfast programs more formally in Sections 4.1 and 4.2 respectively. For the moment, informally, an adequate framework $\mathcal{F}$ is one that remains consistent and a steadfast program one that remains correct in each instance of M.

Example 4.1 We can show that the following module is correct:

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

Program:- ms : $sort \Leftarrow merge, split$	Program:- ps : $sort \Leftarrow p, concat$	$\ldots$
merge sort program	partition sort program	

(3)

where $\mathcal{LIST}(Elem, \lhd)$ is the (open) list framework of Example 3.1, and the programs are sorting programs derived from a specification of list sorting.

For example, the program ms for merge sort is the following:

Program:- ms : $sort \Leftarrow merge, split$

$$\begin{array}{rcl} sort(nil, nil) & \leftarrow & \\ sort(x.nil, x.nil) & \leftarrow & \\ sort(x.y.A, W) & \leftarrow & split(x.y.A, I, J) \wedge sort(I, U) \wedge sort(J, V) \wedge \\ & & merge(U, V, W) \end{array} \quad (4)$$

and the partition sort program ps is:

Program:- ps : $sort \Leftarrow p, concat$

$$\begin{array}{rcl} sort(nil, nil) & \leftarrow & \\ sort(x.nil, x.nil) & \leftarrow & \\ sort(a.A, Z) & \leftarrow & p(a.A, I, J, K) \wedge sort(I, I') \wedge sort(J, J') \wedge sort(K, K') \wedge \\ & & concat(I', J', V) \wedge concat(V, K', Z) \end{array} \quad (5)$$

where $split, merge, p$, and $concat$ are defined in some suitable ways.

As we will show later, this module is correct because the framework $\mathcal{LIST}$ is *adequate* (see Section 4.1), and the programs ms and ps, as well as others which we cannot list here but will see later, are *steadfast* (see Section 4.2).

4.1 Adequate Frameworks

An adequate framework $\mathcal{F}$ is one for which a(n intended) model always exists. Adequacy can be achieved by making sure that if $\mathcal{F}$ is closed, then $\mathcal{F}$ has an isoinitial model; and if $\mathcal{F}$ is open, then every closed instance of $\mathcal{F}$ has an isoinitial model. We shall discuss adequate closed frameworks first.

[6] Again we omit spec_i for brevity.

Definition 4.2 A closed framework is *adequate* iff it has a reachable isoinitial model.

Example 4.2 For example, the closed framework $\mathcal{LIST}(Int, <)$ in Example 3.2 is adequate since it has a reachable isoinitial model.

In general, the existence of an isoinitial model is of course not guaranteed. However, if a framework $\mathcal{F}$ has a reachable model I, then it can be shown that I is an isoinitial model of $\mathcal{F}$ if and only if every closed atomic formula A (in the language of $\mathcal{F}$) is decided by $\mathcal{F}$, i.e. either $\mathcal{F} \vdash A$ or $\mathcal{F} \vdash \neg A$. This is a useful *existence condition* for isoinitial models. Furthermore, it can be shown that if $\mathcal{F}$ is a (possibly infinite) recursively enumerable axiomatisation, then there exists an isoinitial model in which relation symbols are interpreted by decidable relations, and function symbols are interpreted by total computable functions. Indeed, we only ever use such frameworks, with (computable) isoinitial models.

To construct such a framework $\mathcal{F}_n$ for axiomatising a type T, we proceed incrementally. We start with a small framework $\mathcal{F}_0$ with an obvious isoinitial model I_0. For instance, if the freeness axioms ([14]) hold in T, then we use as $\mathcal{F}_0$ the framework consisting of just the constructor symbols in $\mathcal{F}_n$ together with their freeness axioms. The term model generated by the constructors is of course just the set of ground terms of T. It is an isoinitial model of their freeness axioms, i.e. it is an isoinitial model of $\mathcal{F}_0$.

Then we successively expand $\mathcal{F}_i$ into $\mathcal{F}_{i+1}$ by adding new function and relation symbols, together with their axioms, in such a way that I_i can be expanded into an isoinitial model I_{i+1} of $\mathcal{F}_{i+1}$.

For example, the closed framework $\mathcal{LIST}(Int, <)$ can be constructed in this way from the term model generated by constructors $\cdot$ and nil of $List$, which is in turn constructed from the term model generated by the constructors 0 (zero) and s (successor) of $\mathcal{NAT}$.

Framework expansions by definitions of new symbols that preserve isoinitiality in this way are called *adequate* expansions, and such definitions are also said to be *adequate*. So we say a framework is *adequate* if it is constructed only from adequate expansions. A useful criterion for the adequacy of an expansion is the existence of a steadfast program to compute the relation defined by the expansion (see [5]). Thus program synthesis itself provides a means of adequately expanding a framework.

To extend this discussion to open frameworks, we need to address closed instances of an open framework $\mathcal{F}$, i.e. compositions of $\mathcal{F}$ with closed frameworks $\mathcal{C}$'s that are closed frameworks themselves. So we will now briefly define framework composition in general, i.e. between open frameworks.

Definition 4.3 Consider two (open) frameworks $\mathcal{F}_1(\Pi_1)$ and $\mathcal{F}_2(\Pi_2)$ with signatures Σ_1 and Σ_2. The *composition* of $\mathcal{F}_1$ with $\mathcal{F}_2$ is performed through a Π_1-renaming ρ. The composite framework is denoted by $\mathcal{F}_1[\rho, \mathcal{F}_2]$. $\rho(\Sigma_1)$ is a Π_1-renaming for Σ_1 and Σ_2 if for every σ used in Π_1, $\rho(\sigma)$ belongs to Σ_2, whilst the sort symbols and declarations not used in Π_1 are mapped into names that do not occur in Σ_2.

$\mathcal{F}_1[\rho, \mathcal{F}_2]$ is *defined* if the p-axioms of $\mathcal{F}_1$ become axioms or theorems of $\mathcal{F}_2$,[7] and in this case, in $\mathcal{F}_1[\rho, \mathcal{F}_2]$:

- the parameters are the (possible) ones of $\mathcal{F}_2$;
- the signature is $\rho(\Sigma_1) \cup \Sigma_2$;
- the axioms and theorems are the unions of those in $\mathcal{F}_1$ and $\mathcal{F}_2$.

Our definition of composition essentially coincides with that of institutions [2]. However, the theory of institutions is not useful for our purposes, since it does not deal with the problem of preserving consistency.[8]

Example 4.3 In our approach, framework composition is effected via parameter instantiation. For example, the closed instance $\mathcal{LIST}(Int, <)$ of the framework $\mathcal{LIST}(Elem, \lhd)$ is the composition of the latter with the framework that defines the integer type Int with the total ordering $<$.

[7] The p-axioms of $\mathcal{F}_1$ are therefore like proof obligations, or requirement theories in [3].

[8] Equational or strict Horn axioms are guaranteed to be consistent.

Now we can define adequate open frameworks:

Definition 4.4 An open framework $\mathcal{F}(\Pi)$ is *adequate* iff, for every closed framework $\mathcal{C}$, if $\mathcal{F}[\rho,\mathcal{C}]$ is defined, then $\mathcal{F}[\rho,\mathcal{C}]$ has a reachable isoinitial model $\mathcal{I}$, i.e. it is closed, and the reduct $\mathcal{I}|\rho$ is an isoinitial model of $\mathcal{C}$.

We can show that framework composition preserves adequacy:

Theorem 4.1 If $\mathcal{F}_1(\Pi_1)$ and $\mathcal{F}_2(\Pi_2)$ are adequate, and $\mathcal{F}_1[\rho,\mathcal{F}_2]$ is defined, then $\mathcal{F}_1[\rho,\mathcal{F}_2]$ is adequate.

(A proof is given in [4].)

Theorem 4.1 provides the basis for our strategy to start from small adequate (closed or open) frameworks, and incrementally construct larger adequate frameworks. The following basic adequate frameworks form a sufficient set of such building blocks:

Example 4.4 (Term models)

Consider a set K of constant and function symbols. K may be open or closed, depending on whether it contains open sort symbols or not. (A sort s is open (wrt K) iff no declaration of the form $f:a\rightarrow s$ belongs to K.)

If K is closed, then $free(K)$ is an adequate closed framework, and the structure freely generated by K is an isoinitial model of $free(K)$.

If K contains open sorts $s_1,\ldots,s_n$, then $free(K(s_1,\ldots,s_n))$ is an adequate open framework.

Example 4.5 (Explicit function definitions)

The (open) framework $\mathcal{F}(\Pi)$ consisting of

- the signature containing
  - sort symbols a and b;
  - function declaration $f : a \rightarrow b$;
  - relation declaration $r : (a,b)$;
- the axioms $\forall x\, r(x,f(x))$ and $\forall x \exists! y\, r(x,y)$

where $\Pi = \{r,a,b\}$, is an adequate open framework.

Example 4.6 (Explicit relation definitions)

The (open) framework $\mathcal{F}(\Pi)$ consisting of

- the signature Σ of a specification $r(x) \leftrightarrow R(x)$, where $R(x)$ is a quantifier-free formula that does not contain r;
- the axiom $\forall x\,(r(x) \leftrightarrow R(x))$

where Π is the signature of $R(x)$ (i.e. Π does not contain r), is an adequate open framework.

Example 4.7 (Logic programs)

Let P be a closed logic program that terminates for every ground atomic goal $\leftarrow A$. Then the theory

$$\mathcal{T} = free(P) \cup Ocomp(P)$$

where $free(P)$ is the set of freeness axioms for the constant and function symbols of P, and $Ocomp(P)$ is the set of completed definitions of the predicates defined by P, is an adequate closed framework, and the minimum Herbrand model of P is an isoinitial model of $\mathcal{T}$.

Using the above results, we can construct adequate (closed and open) frameworks incrementally. We can also show that the framework $\mathcal{LIST}(Elem,\lhd)$ in Example 4.1 (defined in Example 3.1) is an adequate open framework.

4.2 Steadfast Programs

In a (parameterised) module M, an open program is steadfast iff it is correct wrt its specification in *all* instances of M. Thus steadfastness means 'correct reusability', i.e. a steadfast program is a correctly reusable unit in all instances of M.

Typically, in our approach, from a given specification in a module M we would synthesise a hierarchy of interrelated steadfast programs. The programs in this hierarchy can be composed with each other to form any desired collection of steadfast programs. Then in any instance of M, we have an instance of the same hierarchy of steadfast programs, which can be composed in different combinations.

Correct reusability of steadfast programs means *a priori* correctness of their composition. This is a very important feature of steadfastness, and contrasts favourably with *a posteriori* correctness normally employed in verification-based algebraic approaches to program construction.

Now we define steadfastness. Note that since a program may have (program) parameters that are distinct from the framework parameters, open programs arise in both closed and open frameworks.

In a closed framework $\mathcal{C}$, steadfastness can be defined as follows:

Definition 4.5 The (open) program $P : \delta_1 \Leftarrow \pi_1$ is *steadfast* in $\mathcal{C}$ iff, for every correct (closed) program $\{Q : \delta_2 \Leftarrow\}$ for computing δ_2, with $\delta_2 \supseteq \pi_1$ and $\delta_2 \cap \delta_1 = \emptyset$, $\{P \cup Q : \delta_1 \cup \delta_2 \Leftarrow\}$ is a correct program for computing $\delta_1 \cup \delta_2$.

Thus programs that are steadfast within a closed framework behave like correct units that can be composed in a correct way.

Example 4.8 In $\mathcal{NAT}$, from the specification S_{prod} in Example 2.1 we can derive the open program oprod (which is just P_2 (1) in Example 2.1 of course):

Program:- oprod : $prod \Leftarrow sum$		
$prod(x, 0, 0)$	$\leftarrow$	
$prod(x, s(y), z)$	$\leftarrow$	$prod(x, y, w), sum(w, x, z)$

The program oprod is open, since it does not contain clauses for computing the parameter sum. Therefore, it cannot be used for computing. However, it is steadfast in $\mathcal{NAT}$. This means that, for every (closed) program sum for computing sum, if sum is correct (in $\mathcal{NAT}$), then {prod $\cup$ sum} is a (closed) correct program for computing $prod$. Thus in Example 2.1 we could stop synthesis for $prod$ after Step 2 and get the open steadfast program oprod.

In an open framework $\mathcal{F}(\Pi)$, the meaning of the defined symbols Δ is characterised in terms of the parameters Π. Since the latter are supplied by a closed framework, steadfastness can be defined as follows (a more abstract, model-theoretic characterisation, can be found in [9]):

Definition 4.6 Let $\mathcal{F}(\Pi)$ be an open framework, and $P : \delta \Leftarrow \pi$ be an open program. P is *steadfast* in $\mathcal{F}(\Pi)$ iff it is steadfast in every closed instance $\mathcal{F}(\Pi) \cup \mathcal{C}$, where $\mathcal{C}$ is a closed framework.

This means that $P : \delta \Leftarrow \pi$ is (closed and) correct wrt to its specification whenever $\mathcal{F}(\Pi)$ becomes closed, and correct closed programs for computing π are supplied. In other words, an open program is steadfast iff its clauses that define symbols in δ are all correct, assuming that the clauses or programs for π are all correct.

Note that programs for computing framework parameters can be supplied only after the instantiation $\mathcal{F}(\Pi) \cup \mathcal{C}$, while those for the program parameters can be supplied either before or after such an instantiation.

Example 4.9 The programs ms (4) and ps (5) in Example 4.1 are examples of open programs in an open framework. We can show that they are steadfast in $\mathcal{LIST}(Elem, \lhd)$ by showing that ms and ps define correct clauses for sorting based on merging and partitioning, assuming correct clauses for $split$ and $merge$, and for P and $concat$, respectively.

To synthesise a steadfast program $P : \delta \Leftarrow \pi$ in an open framework, we can employ the same synthesis process that is exemplified in Example 2.1. The criterion for halting synthesis successfully (with a steadfast P) is also the same, except that we need only consider predicate symbols in δ:

For every predicate $p \in \delta$ in the current program P_j:

(i) if $\mathcal{F} \cup S_j \vdash \neg Comp^{-}(P_j)$, search for other clauses with p in their head;

(ii) if $\mathcal{F} \cup S_j \vdash Comp^{-}(P_j)$, do not search for any more clauses defining p.

If (ii) holds for every defined predicate p of P_j, then synthesis successfully halts, and P_j is steadfast.

Now we will show a hierarchy of steadfast programs that can be synthesised in an open framework in a module M, to illustrate how steadfast programs can be used as correct program units in M and all its instances.

Example 4.10 In the framework $\mathcal{LIST}(Elem, \lhd)$, from the specification

$$sort(x,y) \leftrightarrow perm(x,y) \wedge ord(y) \tag{6}$$

(where $perm$ and ord are as defined in (2)) we can derive different steadfast programs Ps, and from these programs we can in turn derive different (steadfast) programs to compute the parameters of Ps. Thus we can synthesise a hierarchy of (steadfast) sorting programs in a modular way. (Note that in the p-axioms we require that $\lhd$ is a partial (not necessarily total) ordering. This means that $sort(x,y)$ is a partial map from x to y. Indeed, a sorted y exists only for those x whose elements are totally ordered by $\lhd$. The programs that we exhibit below are steadfast even in this weaker situation, and they behave as the usual sorting programs in the case where $\lhd$ is total.)

For example, we can synthesise the steadfast program (4) in Example 4.1 for merge sort, where the program parameters $split$ and $merge$ can be defined by the *partial specification*:

$$\begin{array}{l} split_{max}(x,i,j) \leftrightarrow (perm(x,i|j) \wedge i \prec x \wedge j \prec x) \\ \forall x,y,a \exists i,j\, split(x.y.a,i,j) \wedge \forall x,i,j\, (split(x,i,j) \rightarrow split_{max}(x,i,j)) \\ ord(x) \wedge ord(y) \rightarrow (merge(x,y,z) \leftrightarrow ord(z) \wedge perm(x|y,z)). \end{array} \tag{7}$$

where $\prec$ is the well-founded relation $x \prec y \leftrightarrow l(x) \leq 1 \vee l(x) < l(y)$. Now (7), like any partial specification, allows many possible interpretations. (Intuitively (7) reflects a design decision to split the input list, then sort the sublists and merge the sorted sublists.) Leaving $split$ and $merge$ open, we can prove that the program (4) is steadfast (in $\mathcal{LIST}(Elem, \lhd)$) wrt the specification (6), that is, it is correct in every closed instance of $\mathcal{LIST}(Elem, \lhd)$, i.e. for any element type $Elem$ and any (total) ordering relation $\lhd$, whenever $merge$ and $split$ compute relations that satisfy the partial specification (7). Thus it can be composed with different programs for $merge$ and $split$ to yield different sorting programs based on merging.

For instance, consider the (adequate) explicit definition

$$split1(x,i,j) \leftrightarrow \exists a,A\, (x = a.A \wedge i = a.nil \wedge j = A \wedge split_{max}(a.A, a.nil, A))$$

Now $split1$ satisfies (7), and we can derive the (trivial) closed program:

Program:- split1 : $split$
$split(a.A, a.nil, A) \quad \leftarrow$

Composing the programs ms and split1 (and simplifying) will give the following steadfast program for insertion sort:

Program:- insertion sort : $sort \Leftarrow merge$		
$sort(nil, nil)$	$\leftarrow$	
$sort(x.A, W)$	$\leftarrow$	$sort(A,V) \wedge merge(x.nil, V, W)$

Alternatively, from the top-level specification (6), we can synthesise the steadfast program (5) for partition sort, where p is a *partition* relation partially specified by:

$$\begin{array}{l} p_{max}(X,I,J,K) \leftrightarrow perm(X, I|J|K) \wedge less(I,J) \wedge less(J,K) \wedge \\ I \prec X \wedge J \prec X \wedge K \prec X, \\ \forall x,X\, \exists I,J,K (p(x.X,I,J,K) \wedge \forall X,I,J,K (p(X,I,J,K) \rightarrow p_{max}(X,I,J,K) \end{array}$$

where $less(A, B) \leftrightarrow \forall a, b\,(a \in A \wedge b \in B \rightarrow a \lhd b)$ and the well-founded relation $\prec$ is defined by $A \prec B \leftrightarrow l(A) \leq 1 \vee l(A) < l(B)$.

From the program ps (5) we can derive a family of programs by specialising p in different ways. For instance, from the specialisation

$$p_1(X, I, J, K) \leftrightarrow \exists a, A(X = a.A \wedge J = a.nil \wedge p_{max}(a.A, I, a.nil, K))$$

we can derive the open program

Program:- p1 : $p1 \Leftarrow\ \lhd$		
$p1(nil, nil, nil, nil)$	$\leftarrow$	
$p1(u.nil, nil, u.nil, nil)$	$\leftarrow$	
$p1(a.X, a.Y, u.nil, W)$	$\leftarrow$	$a \lhd u \wedge p1(X, Y, u.nil, W)$
$p1(a.X, Y, u.nil, a.W)$	$\leftarrow$	$u \lhd a \wedge p1(X, Y, u.nil, W)$

Composing the programs ps and p1 would give us the following program for quick sort:

Program:- quick sort : $sort \Leftarrow p1, concat$		
$sort(nil, nil)$	$\leftarrow$	
$sort(a.A, Z)$	$\leftarrow$	$p1(a.A, I, a.nil, K) \wedge sort(I, I') \wedge sort(K, K') \wedge$ $concat(I', a.nil, V) \wedge concat(V, K', Z)$

Similarly we could derive the steadfast programs p2 and p3:

Program:- p2 : $p2 \Leftarrow\ \lhd$		
$p2(nil, nil, nil)$	$\leftarrow$	
$p2(u.nil, u.nil, nil)$	$\leftarrow$	
$p2(a.X, a.nil, u.W)$	$\leftarrow$	$a \lhd u \wedge p2(X, u.nil, W)$
$p2(a.X, u.nil, a.W)$	$\leftarrow$	$u \lhd a \wedge p2(X, u.nil, W)$

Program:- p3 : $p3 \Leftarrow\ \lhd$		
$p3(nil, nil, nil)$	$\leftarrow$	
$p3(u.nil, u.nil, nil)$	$\leftarrow$	
$p3(a.X, a.nil, X)$	$\leftarrow$	$a \lhd u \wedge p3(X, u.nil, W)$
$p3(a.X, u.nil, a.W)$	$\leftarrow$	$u \lhd a \wedge p3(X, u.nil, W)$

which can be composed with ps to get the following programs for bubble sort and selection sort:

Program:- bubble sort : $sort \Leftarrow p2$		
$sort(nil, nil)$	$\leftarrow$	
$sort(a.A, z.Z)$	$\leftarrow$	$p2(a.A, z.nil, W) \wedge sort(W, Z)$

Program:- selection sort : $sort \Leftarrow p3$		
$sort(nil, nil)$	$\leftarrow$	
$sort(a.A, z.Z)$	$\leftarrow$	$p3(a.A, z.nil, W) \wedge sort(W, Z)$

These sorting program units are all steadfast. Each open steadfast program can be used as a correct schema for program composition, over different instances of the module. Thus our family of steadfast programs can be reused for generating code for different sorting algorithms in different problem domains.

Returning to Example 4.1, having shown in Section 4.1 that the open framework $\mathcal{LIST}(Elem, \lhd)$ is *adequate*, and that all the programs in Example 4.10 in the framework are steadfast, we can now say that the module (3) is a *correct*.

Thus a correct parameterised module is a *correct class*. As illustrated in Figure 2, an adequate framework is equivalent to a class in which the class invariants on the ADT are always satisfied because of the way the framework has been constructed; and steadfast programs are like methods that always satisfy their pre-post-condition specifications because of the *a priori* nature of correctness that steadfastness embodies.

Correct parameterised module	=	Adequate framework	+	Steadfast programs
↓		↓		↓
Correct class	=	ADT	+	Methods
		(+ class invariants)		(+ pre-post-conditions)

Figure 2: Correct parameterised modules as correct classes

5 Conclusion

We have presented a notion of a correct class and its formal development in computational logic. A class is a parameterised module M consisting of a full first-order theory, or a framework $\mathcal{F}$, and logic programs prog_i that are specified and derived in $\mathcal{F}$. The correctness of a class is defined as the adequacy of $\mathcal{F}$ and the correctness of prog_i in all instances of M. We have discussed the notions of adequate frameworks and steadfast programs, and shown that such frameworks and programs make up a correct class.

This notion of class is different from that of a module in the algebraic approach in that the latter is usually a structure or a sub-program, whereas in our approach, a class includes a framework and many programs. The presence of the framework allows us to define not only the correctness of the class, i.e. adequacy of the framework and steadfastness of the programs in the framework, but also the correctness of class compositions or instances. Moreover, both notions of correctness are *a priori* correctness because of adequacy and steadfastness, in contrast to the *a posteriori* correctness that is usually established by verification in the algebraic approach.

Compared to classes in object-oriented programming, our class also has the advantage of correctness, as shown in Figure 2. However, our class is a class of polymorphic ADT's, rather than a class of state-based objects as in object-oriented programming. To be more precise, our class can be depicted in Figure 3, which is a refinement of Figure 1 in Section 3. The kind of polymorphism in the ADT here is *parametric polymorphism*, as opposed to *inheritance polymorphism* ([15]) that is the basis of inheritance in object-oriented programming.

(Parameterised) Module	=	(Open) Framework	+	(Open) Programs
↓		↓		↓
(Generic) Class	=	(Polymorphic) ADT	+	(Open) Methods

Figure 3: Generic classes

Thus our class is a semantic characterisation of a generic class in object-oriented programming, with the additional property of correctness, but without the 'efficiency' of state-based objects or the flexibility of sub-type inheritance. It would be interesting to study how the advantages of both approaches could be combined, perhaps through the vehicle of object-oriented logic programming languages such as [11, 13].

Acknowledgements

We wish to thank the referees for pointing out a number of minor errors in an earlier version of this paper.

References

[1] Y. Deville and K.K. Lau. Logic program synthesis. *J. Logic Programming* 19,20:321–350, 1994. Special issue: Ten years of logic programming.

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

[3] J. Goguen and J. Meseguer. Unifying functional, object-oriented and relational programming with logical semantics. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 417–477, MIT Press, 1987.

[4] C. Kreitz, K.K. Lau, and M. Ornaghi. Formal reasoning about modules, reuse and their correctness. In D.M. Gabbay and H.J. Ohlbach, editors, *Proc. Int. Conf. on Formal and Applied Practical Reasoning*, *LNAI* 1085:384–399, Springer-Verlag 1996.

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

[6] K.K. Lau, M. Ornaghi and S.-Å. Tärnlund. The halting problem for deductive synthesis of logic programs. In P. van Hentenryck, editor, *Proc.* 11^{th} *Int. Conf. on Logic Programming*, pages 665–683, MIT Press, 1994.

[7] K.K. Lau and M. Ornaghi. Towards an object-oriented methodology for deductive synthesis of logic programs. In M. Proietti, editor, *Proc. LOPSTR 95*, *LNCS* 1048:152–169, Springer-Verlag, 1996.

[8] K.K. Lau and M. Ornaghi. A formal approach to deductive synthesis of constraint logic programs. In J.W. Lloyd, editor, *Proc. 1995 Int. Logic Programming Symp.*, pages 543–557, MIT Press, 1995.

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

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

[11] F.G. McCabe. *L& O: Logic and Objects*. Prentice-Hall, 1992.

[12] B. Meyer. *Eiffel the Language*. Prentice Hall, 1992.

[13] C.D.S. Moss. *Prolog++: The Power of Object-Oriented and Logic Programming*. Addison Wesley, 1994.

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

[15] P. Wegner. The object-oriented classification paradigm. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 479–560, MIT Press, 1987.

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