Towards An Object-Oriented Methodology for Deductive Synthesis of Logic Programs*

Kung-Kiu Lau[1] and Mario Ornaghi[2]

[1] Fachgebiet Intellektik, Fachbereich Informatik
Technische Hochschule Darmstadt
Alexanderstr. 10, D-64283 Darmstadt, Germany
lau@intellektik.informatik.th-darmstadt.de (or kung-kiu@cs.man.ac.uk)
[2] Dipartimento di Scienze dell'Informazione
Universita' degli studi di Milano, Via Comelico 39/41, Milano, Italy
ornaghi@hermes.mc.dsi.unimi.it

Abstract. Quality software must be reusable, extensible, and reliable. Object-oriented programming purports to achieve these attributes by the use of classes and inheritance (informally). In this paper, we show how our existing approach to deductive synthesis of logic programs can serve as the basis for an object-oriented methodology for formal program development that achieves reusability, extensibility and correctness (formally).

1 Introduction

Object-oriented programming is very much seen as the standard bearer of the software industry nowadays, because it purports to achieve the key attributes of any quality software: *reusability*, *extensibility*, and *reliability* (see e.g. [12, 13]). Object-oriented programming achieves reusability and extensibility by using a *class* as the basic modular unit, and a mechanism called *inheritance* to define new classes from existing ones. Reliability results from the ability to monitor *assertions* and *invariants* contained in classes. However, at present, there is no single standard (formal) characterisation of object-oriented programming (see [7]), so we shall adopt the common view of object-oriented programming, as expressed in e.g. [13, 16] (based on imperative programming).

In an object-oriented system, the basic modular unit is a *class*, which describes an implementation of an abstract data type (ADT), including a set of procedures, called *methods*, that implement operations on the ADT. A class can be composed with other classes to define new classes, through *inheritance*, to form larger, composite classes. If a class C_2 inherits class C_1, then C_1 is a *superclass* of C_2; or conversely, C_2 is a *subclass* of C_1. At execution time, instances of a class are created dynamically. Such instances are called *objects*. An object contains actual *data* (of the type of its ADT). Its methods, when invoked by *messages* sent by other objects, are used to manipulate its data. An object therefore has an internal state. Thus a class is a (compile-time) template for generating a family of (run-time) objects with the same data type and methods.

* The first author is on leave from the Department of Computer Science, University of Manchester, UK, and is supported by the European Union HCM project on Logic Program Synthesis and Transformation, contract no. 93/414.

Code reuse arises through inheritance as well as through *dynamic binding*. Inheritance allows code reuse, since the code for the methods in a class is generated only once, and 'inherited' by all the objects of any subclass. If a method has different implementations in different subclasses, then dynamic binding allows any common code for different implementations of the method to be placed in the superclass, and be reused with different subclasses.

However, apart from monitoring assertions and invariants at run-time, current object-oriented programming technology does not support formal program development, i.e. the development of object-oriented programs that are guaranteed to be (formally) correct (wrt their specifications). In this paper, we outline (the beginnings of) such an approach for logic programs, based on our existing approach to deductive synthesis of logic programs. In order to avoid confusion, we shall use *classes* and *objects* only in the sense of conventional object-oriented programming, as described above, and we will not employ these terms when describing elements or features of our approach. However, we will of course relate our approach to object-oriented programming.

Our starting point is a well-defined and well-understood (global) context, that we call a *framework*. A framework is a theory that embodies our knowledge of the problem domain in question. Within this theory, we define semantically notions similar to classes and inheritance in object-oriented programming, and thus characterise modularity and reusability at a semantic level, rather than at code level.

More importantly, *correctness*, both of programs and of their composition, can be defined in a framework. Moreover, the framework itself can be used to state precise conditions under which program composition and reuse preserve correctness. Thus our approach not only offers modularity and reusability, but it also has the additional property of formal correctness, not just of modules and module composition, but also of reuse. This we believe is an advancement over the current object-oriented technology, and in this paper we will discuss how we can use our approach as a basis for an object-oriented methodology for logic program synthesis.

The paper is organised as follows. In Section 2 we give a brief overview of our existing approach. In particular, we describe frameworks (corresponding to ADT's), and steadfast logic programs (corresponding to methods). Then, in Section 3, we discuss how our existing approach can be the basis of a fully object-oriented program synthesis methodology.

2 An Overview of Our Existing Approach

In our existing approach to logic program synthesis a specification S is given in the context of a (first-order) theory $\mathcal{F}$, which we call a *specification framework* (or just *framework* for short). The purpose of a framework is to provide a well-defined context for program specifications, and for reasoning about program synthesis.

Synthesis is *deductive*, i.e. it iteratively derives (using some synthesis method)[3] from a specification S of a relation r a sequence of programs $P_0 \subseteq \cdots \subseteq P_n$, such that P_n, when the synthesis process halts, is *totally correct* wrt S.

A detailed description of our approach, including criteria for correctness and for determining when to halt the synthesis process (with a totally correct program), can be found in [9]. Here we give a brief overview of its key elements that are relevant to this paper.

2.1 Specification Frameworks

A framework is a first-order theory that embodies all the relevant knowledge of the problem domain of interest. Such a theory could be regarded as an axiomatisation of a collection of ADTs. However, it has more than the usual ADTs, since it can also contain *any* relevant axioms for reasoning about ADTs (e.g. induction axioms), as well as other axioms for reasoning about the domain (e.g. stating that in a library each book has a title, a list of authors, and so on).

We endow frameworks with *isoinitial semantics* [1, 8, 14]. It is closely related to initial semantics used in algebraic ADTs [17]. In initial semantics the *intended models* are initial models, whereas in isoinitial semantics they are isoinitial models. We choose isoinitial semantics because the existence of an isoinitial model corresponds to the existence of steadfast programs.

A framework may be *closed*, i.e. completely defined, or it may be *open*, i.e. it may contain *parameters*. A closed framework has only one intended model (unique up to isomorphism), whereas an open framework has a class of intended models.

Example 1. The following *closed* framework formalises Peano Arithmetic, denoted by $\mathcal{PA}$,

Framework $\mathcal{PA}$;

SORTS: Nat;

FUNCTIONS:
$$0 \; : \to Nat;$$
$$s \; : Nat \to Nat;$$
$$+, \cdot : (Nat, Nat) \to Nat;$$

AXIOMS:
$$\forall x(\neg 0 = s(x)) \wedge \forall x, y(s(x) = s(y) \to x = y);$$
$$\forall (H(0) \wedge \forall i(H(i) \to H(s(i))) \to \forall x H(x));$$
$$\forall x(x + 0 = x);$$
$$\forall x, y(x + s(y) = s(x + y));$$
$$\forall x(x \cdot 0 = 0);$$
$$\forall x, y(x \cdot s(y) = x + x \cdot y)$$

The intended model of $\mathcal{PA}$ (under isoinitial semantics) is the standard structure of natural numbers.

---
[3] Synthesis methods are not relevant to this paper. A survey can be found in [6].

Example 2. The following *open* framework formalises a theory of lists, with list element type and ordering as *parameters:*[4]

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

The intended models of $\mathcal{LIST}(Elem, \lhd)$ are the usual list structures with a partial ordering $\lhd$ on the (parametric) element type.

Natural numbers, the function $nocc(x, L)$ (number of occurrences of x in L), and the relation $elemi(L, i, a)$ (a occurs in L at position i) have been introduced to get an *expressive* specification framework, as we will see later.

Frameworks can be built incrementally through the operations of *framework composition* and *framework extension*. The *composition operation* behaves properly, if the frameworks involved are *adequate*.

Definition 1. A closed framework $\mathcal{C}$ is *adequate* iff it has an isoinitial model $\mathcal{I}$.

It turns out that, for every ground literal L, $\mathcal{I} \models L$ iff $\mathcal{C} \vdash L$. Thus, in an adequate closed framework $\mathcal{C}$, the axioms completely define the intended meaning of the symbols of its signature.

We shall denote an open framework by $\mathcal{F}(\Pi) = \mathcal{D} \cup \mathcal{P}(\Pi)$, where Π are the parameters, $\mathcal{D}$ the d-axioms, and $\mathcal{P}(\Pi)$ the p-axioms. The latter state properties that are assumed to be verified by the parameters, and may contain *only* parameters. In Example 2 we assume (as the p-axioms state) that $\lhd$ is a partial ordering. The d-axioms characterise the other symbols, that we will call *defined symbols* of the framework.

[4] In the axioms, external universal quantifiers have been omitted, variables of sort *Elem* are in lower-case, and variables of sort *List* are in upper-case.

Definition 2. An open framework $\mathcal{F}(\Pi)$ is *adequate* iff, for every adequate closed framework $\mathcal{C}$ that satisfies the proof obligation $\mathcal{C} \vdash \mathcal{P}(\Pi)$,[5] $\mathcal{F}(\Pi) \cup \mathcal{C}$ is an adequate closed framework. We call $\mathcal{F}(\Pi) \cup \mathcal{C}$ a *closed instance* of $\mathcal{F}(\Pi)$.

Adequacy of $\mathcal{F}(\Pi) = \mathcal{D} \cup \mathcal{P}(\Pi)$ means that, whenever we have the information on the parameters Π (given by $\mathcal{C}$), we get a closed framework, i.e. that the axioms of $\mathcal{D}$ completely define the meaning of the defined symbols (in terms of the meaning of the parameters).

An *extension operation* introduces a new defined symbol r through a set D_r of axioms, that we call a *definition* of r. Since adequacy is essential for framework composition, we allow only definitions that preserve adequacy, and we also call such definitions *adequate*.

Definition 3. Let $\mathcal{C}$ be a closed framework with isoinitial model $\mathcal{I}$. A definition D_r of a relation r is *adequate* in $\mathcal{C}$ iff $\mathcal{C} \cup D_r$ is an adequate closed framework, and it has an intended model $\mathcal{I}_r$ that is an expansion[6] of $\mathcal{I}$ (by r).

Let $\mathcal{F}(\Pi)$ be an open framework. A definition D_r is *adequate* in $\mathcal{F}(\Pi)$ iff it is adequate in every closed instance $\mathcal{F}(\Pi) \cup \mathcal{C}$ of $\mathcal{F}(\Pi)$.

We consider *recursive definitions* and *explicit definitions*. A recursive definition of a function or a relation is adequate if it satisfies suitable sufficient conditions. Essentially, we require that it is like a primitive recursive definition, like the axioms introducing $+$ and $\cdot$ in $\mathcal{PA}$.

An explicit definition of a relation r has the form $\forall x(r(x) \leftrightarrow R(x))$, where R is a formula of the old language, that we call the *defining formula*. If $R(x)$ is *quantifier-free* or contains only *bounded quantifiers*, then the definition is adequate. Otherwise, a sufficient condition for adequacy is the derivability of a steadfast program for computing the defined relation. For example, in $\mathcal{LIST}(Elem, \lhd)$ we can explicitly define length as follows:

$$\begin{array}{l} len(L,n) \leftrightarrow \forall i(i < n \leftrightarrow \exists a(elemi(L,i,a))) \\ \forall L, n\,(n = l(L) \leftrightarrow len(L,n)) \end{array}$$

In $len(L,n)$ the length n of a list L is defined as its first empty position. The derivability of a program for computing len proves adequacy. Moreover, since we can prove $\forall L \exists! n\, len(L,n)$, we can explicitly define the function l.

Similarly, we can introduce the most useful relations and functions on lists, like $L|M$ (the *concatenation* function), $x \in L$ (x is a member of list L), $perm(L,M)$ (L is a *permutation* of M), $ord(L)$ (L is *ordered*), and so on. We say that $\mathcal{LIST}(Elem, \lhd)$ is *expressive*.

2.2 Steadfast Logic Programs

The standard notion of *correctness* in logic program synthesis is defined for *closed* programs only. It can be stated in terms of specification frameworks as

[5] Here, for the sake of simplicity, we assume that the intersection of the signatures of $\mathcal{C}$ and $\mathcal{F}(\Pi)$ coincides with Π.

[6] See e.g. [5].

follows: in a closed framework $\mathcal{C}$, a program P for computing r, specified by (the adequate definition) $\forall(r(x) \leftrightarrow R(x))$, is correct wrt the specification iff P's minimum Herbrand model is isomorphic to the intended model of $\mathcal{C}$ (restricted to the signature of P).

Steadfastness extends this to *open* programs. An open program, denoted by $P : \delta \Leftarrow \pi$, contains the definitions of a set δ of relation symbols in terms of a set π of parameters; to be more precise, the *defined relations* δ occur in the heads of the clauses of P, whilst the *parameters* π occur only in the bodies. Open programs can arise in both *closed* and *open* frameworks.

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

Definition 4. $P : \delta_1 \Leftarrow \pi_1$ 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 3. In $\mathcal{PA}$ we can specify the relations $\forall(prod(x, y, z) \leftrightarrow z = x \cdot y)$ and $\forall(sum(x, y, z) \leftrightarrow z = x+y)$. Since the defining formulas are quantifier-free, these definitions are adequate. We can derive the open program

Framework:- $\mathcal{PA}$
Program:- prod : $prod \Leftarrow sum$
$prod(x, 0, 0) \leftarrow$ $prod(x, s(y), z) \leftarrow prod(x \cdot y, w), sum(w, x, z)$

The program prod 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 $\mathcal{PA}$. This means that, for every (closed) program sum for computing *sum*, if sum is correct (in $\mathcal{PA}$), then $\{$prod $\cup$ sum$\}$ is a (closed) correct program for computing *prod*.

In an open framework $\mathcal{F}(\Pi)$, the meaning of the defined symbols is characterised in terms of the parameters Π. Since the latter are supplied by a closed framework, steadfastness can be defined as follows:[7]

Definition 5. 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 $\mathcal{F}(\Pi) \cup \mathcal{C}$.

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. The parameters of π that belong to Π will be called *framework-parameters*, to distinguish them from those that do not belong to Π, which we will call

[7] A more abstract, model-theoretic characterisation, can be found in [10].

program-parameters. The programs for computing framework-parameters can be supplied only after the instantiation $\mathcal{F}(\Pi) \cup \mathcal{C}$ by a closed framework $\mathcal{C}$, while those for the program-parameters can be supplied either before or after such an instantiation. Note that, in a closed framework, an open program has only program-parameters. These parameters may be completely specified, e.g. *sum* in the program prod in Example 3, or partially defined, as we will see in the next example.

Example 4. In $\mathcal{LIST}(Elem, \lhd)$ sorting can be defined in the usual way, by the adequate explicit definition:

$$sort(x, y) \leftrightarrow perm(x, y) \wedge ord(y) \tag{1}$$

From (1) we can synthesise the following open program for *sort*:

Framework:- $\mathcal{LIST}(Elem, \lhd)$	
Program:- merge sort : $sort \Leftarrow merge, split$	
$sort(nil, nil) \leftarrow$	
$sort(x.nil, x.nil) \leftarrow$	
$sort(x.y.A, W) \leftarrow$	$split(x.y.A, I, J) \wedge sort(I, U) \wedge sort(J, V) \wedge$
	$merge(U, V, W)$

(2)

The program-parameters *split* and *merge* can be specified by the *partial definition*:

$$\begin{array}{l} split_{max}(x, i, j) \leftrightarrow (perm(x, i|j) \wedge i \prec x \wedge j \prec x) \\ \forall x, y, a \exists i, j \; split(x.y.a, i, j) \wedge \forall x, i, j \; (split(x, i, j) \rightarrow split_{max}(x, i, j)) \\ ord(x) \wedge ord(y) \rightarrow (merge(x, y, z) \leftrightarrow ord(z) \wedge perm(x|y, z)). \end{array} \tag{3}$$

where $\prec$ is the well founded relation $x \prec y \leftrightarrow l(x) \leq 1 \vee l(x) < l(y)$. (3) allows many possible interpretations. It reflects a design decision to split the input list, then sort the sublists and merge the sorted sublists. Even if the meaning of the program-parameters *split* and *merge* is left open, we can prove that the program (2) is steadfast (in $\mathcal{LIST}(Elem, \lhd)$) wrt the specification (1), that is, it is correct in every closed instance of $\mathcal{LIST}(Elem, \lhd)$, i.e. for any element type *Elem* and any (partial) ordering relation $\lhd$, whenever *merge* and *split* compute relations that satisfy the partial definitions (3). Thus it can be composed with different programs for *merge* and *split* to yield different sorting algorithms based on merging, e.g. insertion sort (as we will see in Section 3.2).

Intuitively, steadfastness can be seen as a kind of 'two-dimensional' parametric correctness, viz. with respect to both program-parameters and framework-parameters.

3 Towards an Object-Oriented Methodology

Using frameworks and steadfastness, we can synthesise correct programs in an object-oriented manner. In this section, we shall discuss the elements of such a methodology.

3.1 Classes and Inheritance

In object-oriented programming, a class is an implementation of an ADT, with methods that implement operations on the ADT. A framework contains axioms that formalise an ADT (or more generally a problem domain), and adequate definitions which serve as program specifications. Thus a framework can be considered to be a class, in which adequate definitions are declarative specifications of methods. Moreover, like classes, we can construct a hierarchy of frameworks with an inheritance mechanism. In this section, we explain how to construct frameworks by *framework extension* and *framework composition*, and how inheritance works between super- and sub-frameworks.

In a *framework extension*, new axioms and, possibly, new symbols are added, either to strengthen properties of the parameters, or to axiomatise new defined symbols, or to make stronger assumptions.

An *extension* $\mathcal{G}(\Delta)$ of a framework $\mathcal{F}(\Pi)$ (where Δ is a possibly empty list of new parameters) will be of the following form:

Framework $\mathcal{G}(\Delta)$;
IMPORT: $\mathcal{F}(\Pi)$;
SORTS: ...
FUNCTIONS: ...
RELATIONS: ...
D-AXIOMS: ...
P-AXIOMS: ...
THEOREMS: ...

where the new symbols of $\mathcal{G}$ (namely, the signature introduced after the import statement) must not occur in $\mathcal{F}(\Pi)$.

When adding new axioms, we must ensure that adequacy is preserved, i.e. that the extension is *adequate*. Once adequacy has been established, the new framework $\mathcal{G}(\Delta)$ inherits all the axioms, theorems, definitions and steadfast programs that have been developed in $\mathcal{F}(\Pi)$.

Since, in general, it may be not easy to establish adequacy, extensions by arbitrary axioms are rarely used.[8] As we said in Section 2.1, we use *recursive definitions* and *explicit definitions*.

Recursive definitions are used to build the core of a framework, that is to introduce a *few* basic operations and relations, needed to get an *expressive* initial framework. Let us consider for example Presbürger Arithmetic, i.e. $\mathcal{PA}$ without $\cdot$.

[8] We shall discuss this at the end of the section.

It is a complete theory. However, its completeness is not a symptom of deductive power, but of lack of expressiveness. We cannot explicitly define $\cdot$, for example, and therefore we have to add it by a recursive definition.

Once we have $\cdot$, we get the expressive framework $\mathcal{PA}$, where all the computable functions and relations can be explicitly defined. Adequate explicit definitions are our main tool to introduce new defined relations and functions, that can be used as specifications of new programs or simply as useful abbreviations. In $\mathcal{LIST}(Elem, \lhd)$, we have seen the example of *perm*, *ord*, used to define *sort*, and of *sort* itself, used to derive a family of sorting algorithms.

We also allow extension by new parameters. In this case, if the (possible) new p-axioms are consistent, then adequacy is (trivially) preserved.

In a *framework composition*, two frameworks are put together. For example, let us assume that we have a closed framework $\mathcal{C}$ containing the (closed) sort $Elem_{\mathcal{C}}$ and an ordering relation $\lhd_{\mathcal{C}}$. It can be composed with the open framework $\mathcal{LIST}(Elem, \lhd)$, and we obtain a closed framework. Moreover, if $\mathcal{C}$ contains a program for deciding $\lhd_{\mathcal{C}}$, we can compose it with the open sorting program already contained in $\mathcal{LIST}$, and we get a closed sorting program.

We recall that an open framework is of the form $\mathcal{F}(\Pi) = \mathcal{D} \cup \mathcal{P}(\Pi))$, where Π are the parameters, $\mathcal{D}$ the d-axioms, and $\mathcal{P}(\Pi)$ the p-axioms. The operation of composition is defined as follows.

Definition 6. Let $\mathcal{F}_1(\Pi_1) = \mathcal{D}_1 \cup \mathcal{P}_1(\Pi_1)$ and $\mathcal{F}_2(\Pi_2) = \mathcal{D}_2 \cup \mathcal{P}(\Pi_2)$ be two open frameworks. Let us assume that:

(a) the intersection of the signatures of $\mathcal{F}_1$ and $\mathcal{F}_2$ is contained in Π_1;
(b) the proof-obligation $\mathcal{F}_2(\Pi_2) \vdash \mathcal{P}_1(\Pi_1)$ holds.

Then the composition operation $\mathcal{F}_1[\mathcal{F}_2]$ is defined and yields the framework $\mathcal{G}(\Delta)$ such that Δ contains Π_2 and the (possible) symbols of Π_1 that are not defined by $\mathcal{F}_2$, the signature is the union of the signatures of $\mathcal{F}_1$ and $\mathcal{F}_2$, and the axioms are the union of those of $\mathcal{F}_1$ and $\mathcal{F}_2$.

If (a) and (b) are not satisfied, then $\mathcal{F}_1[\mathcal{F}_2]$ is not defined. The obligation (a) is rather restrictive; it can be avoided by using signature morphisms. We do not have the space to present them here. They have been introduced in the algebraic approach (see e.g. [17]) and can be used in our approach, essentially in the same way.

Now, let $\mathcal{F}(\Pi) = \mathcal{D} \cup \mathcal{P}(\Pi)$ be an adequate open framework and $\mathcal{C}$ be an adequate closed framework that defines all the parameters Π. Then $\mathcal{G} = \mathcal{F}[\mathcal{C}]$ is a closed instance of $\mathcal{F}$. Since $\mathcal{F}$ and $\mathcal{C}$ are adequate, $\mathcal{G}$ is an adequate closed framework.

Moreover we can easily see that $\mathcal{F}_1[\mathcal{F}_2[\mathcal{C}]] = (\mathcal{F}_1[\mathcal{F}_2])[\mathcal{C}]$. Therefore we have:

Theorem 7. *If $\mathcal{F}_1$ and $\mathcal{F}_2$ are adequate, and $\mathcal{F}_1[\mathcal{F}_2]$ is defined, then $\mathcal{F}_1[\mathcal{F}_2]$ yields an adequate framework.*

This theorem shows the operation $\mathcal{F}_1[\mathcal{F}_2]$ preserves the adequacy property. It is obvious that all the theorems that have been proved (separately) in $\mathcal{F}_1$

and $\mathcal{F}_2$ are inherited. Moreover, it can be shown that composition preserves the steadfastness of the programs that have been derived (separately) in $\mathcal{F}_1$ and $\mathcal{F}_2$. Thus adequate definitions and steadfast programs are also inherited.

The operations of framework extension and framework composition can be combined for building frameworks in an object-oriented manner. In object-oriented terminology, a framework is a class, and its steadfast programs are the methods of the class. An adequate extension $\mathcal{G}(\Delta)$ of $\mathcal{F}(\Pi)$ is a *subclass* of $\mathcal{F}(\Pi)$. Subclasses inherit from their parent classes. For example, the framework $\mathcal{LIST}(Elem, \lhd)$ has been built up as an extension of $\mathcal{PA}$. It inherits $\mathcal{PA}$ (by importing it), and introduces the new parameters $Elem, \lhd$ and the new defined symbols nil, :, $nocc$, $elemi$.

As another example, we can define the purely parametric class of partial orderings $\mathcal{PO}(Elem, \lhd)$, with signature containing only the two parameters $Elem$ and $\lhd : (Elem, Elem)$, and the axioms that state that $\lhd$ is a partial ordering. Then we can build the subclass $\mathcal{TO}(Elem, \lhd)$ by adding the axiom $\forall x, y : Elem(x \lhd y \vee y \lhd x)$, that states that the ordering $\lhd$ is also total.

Framework composition works like *class composition*, and the associated inheritance is a kind of *multiple inheritance*: $\mathcal{F}_1[\mathcal{F}_2]$ inherits from both $\mathcal{F}_1$ and $\mathcal{F}_2$. For example, let $\mathcal{LIST}(Elem)$ be the framework for lists without $\lhd$, then $\mathcal{LIST}(Elem, \lhd)$ can be generated by the composition:

$$\mathcal{LIST}(Elem, \lhd) = \mathcal{LIST}(Elem)[\mathcal{PO}(Elem, \lhd)].$$

It inherits from both $\mathcal{LIST}(Elem)$ and $\mathcal{PO}(Elem, \lhd)$.

An interesting fact is that subclass inheritance and composition inheritance are orthogonal: if $\mathcal{G}_1$ is an (adequate) extension of $\mathcal{F}_1$, and $\mathcal{G}_2$ is an (adequate) extension of $\mathcal{F}_2$, then, if $\mathcal{G}_1[\mathcal{G}_2]$ is defined, it is an adequate extension of $\mathcal{F}_1[\mathcal{F}_2]$. For example, since we know that $\mathcal{TO}(Elem, \lhd)$ is an extension of $\mathcal{PO}(Elem, \lhd)$, we automatically know that $\mathcal{LIST}(Elem)[\mathcal{TO}(Elem, \lhd)]$ is an extension of $\mathcal{LIST}(Elem)[\mathcal{PO}(Elem, \lhd)]$ (from which it therefore inherits). This allows propagation of extensions to various frameworks.

The above discussion shows how reusability is supported by the inheritance property of the framework operations we have considered. Now we briefly comment on reliability. There are three kinds of correctness problems:

- Informal correctness wrt the intuitive definition of a data type or an informal problem domain.
- Formal correctness of framework extensions and compositions.
- Formal correctness of program composition.

Formal correctness of framework extension and composition is guaranteed by the adequacy of the operations, as we have already discussed. Formal correctness of program composition will be (further) addressed in Section 3.3. Here, we briefly comment on informal correctness. Since this cannot be treated formally, we should build a possibly small, intuitively correct formalisation, that we call a *top-level* framework. When we are convinced that the top-level framework meets our informal idea of the problem domain, and we have a sufficiently expressive and adequate framework, then we extend it by adequate definitions. Adequate definitions are safe and completely modular. Indeed each definition is completely independent of the others, since it is eliminable and gives rise to a conservative extension. This means that, even if we discover that it does not meet our intuitive specification, we can modify it (and the related steadfast programs) without affecting any other part of the framework. This cannot be done, in general, with other kinds of definitions, since they may be non-eliminable (they increase the expressive power and cannot be eliminated, if they have been used elsewhere), or even non-conservative (they allow us to prove new theorems, that may have been used in other parts of the framework).

3.2 Synthesising Methods

Having shown how to construct classes (frameworks), we now show how in a given class, we can synthesise methods (steadfast programs), in an object-oriented style, i.e. in a hierarchical, modular manner.

Example 5. In the framework $\mathcal{LIST}(Elem, \lhd)$, from the specification (1) of sorting in Example 4 (in Section 2.2), 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. Let us see some examples.

As we saw in Example 4, from (1) we can synthesise the steadfast program (2) for merge sort. By using different specialisations of *split* to obtain different implementations, we can get different sorting programs. For instance, let us consider the adequate explicit definition

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

Now *split1* satisfies (3). Therefore it can replace *split* in (2), and we can derive the following (open) steadfast program, where *split1* disappears and the second clause of (2) is subsumed by the second one of (4):

Framework:- $\mathcal{LIST}(Elem, \lhd)$
Program:- insertion sort : $sort \Leftarrow merge$
$sort(nil, nil)$. $sort(x.A, W) \leftarrow sort(A, V) \wedge merge([x], V, W)$

(4)

Now we can rewrite the method *merge* to obtain insertion sort. Alternatively, from (1), we can synthesise the following steadfast program for partition sort:

$$\begin{array}{|c|}\hline \textsf{Framework:-}\ \mathcal{LIST}(Elem, \lhd) \\ \hline \\ \begin{array}{|l|}\hline \quad\textsf{Program:-}\ \text{partition sort}: sort \Leftarrow P, concat \\ \hline \quad sort(nil, nil). \\ sort(x.nil, x.nil) \leftarrow \\ \quad sort(a.A, Z) \leftarrow P(a.A, I, J, K) \wedge sort(I, I') \wedge sort(J, J') \wedge \\ \qquad\qquad sort(K, K') \wedge concat(I', J', V) \wedge concat(V, K', Z) \\ \hline \end{array} \\ \\ \hline \end{array} \tag{5}$$

where P is a *partition* relation partially defined 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 (5) we can derive a family of programs by specialising P in different ways. For instance, by the (adequate) specialisation $P_1(X, I, J, K) \leftrightarrow \exists a, A(X = a.A \wedge J = [a] \wedge P_{max}(a.A, I, [a], K))$ we can get the following program for quick sort:

$$\begin{array}{|c|}\hline \textsf{Framework:-}\ \mathcal{LIST}(Elem, \lhd) \\ \hline \\ \begin{array}{|l|}\hline \quad\textsf{Program:-}\ \text{quick sort}: sort \Leftarrow part1, concat \\ \hline sort(nil, nil) \leftarrow \\ sort(a.A, Z) \leftarrow P_1(a.A, I, [a], K) \wedge sort(I, I') \wedge sort(K, K') \wedge \\ \qquad\qquad concat(I', [a], V) \wedge concat(V, K', Z) \\ \hline \end{array} \\ \\ \hline \end{array}$$

Now $P_1(a.A, I, [a], K)$ can be rewritten by the usual partition relation for quick sort, defined by $part1(A, I, a, K) \leftrightarrow P_1(a.A, I, [a], K)$. Finally, we can derive a steadfast program for computing such a relation.

These sorting programs are steadfast. Each open steadfast program can be used as a correct schema for program composition, over different instances of the parameters (both program- and framework-parameters). Thus our family of steadfast programs can be reused for generating code for different sorting algorithms in different problem domains. In object-oriented programming terms, they provide a basis for constructing libraries of 'correct' classes for sorting.

3.3 Reusability of Methods

A distinguishing feature of the methods (steadfast programs) in our classes (frameworks) is their *reusability* wrt both program-parameters and framework-parameters, as demonstrated by the example in the previous section. However, in general, such reusability is not always as straightforward as the example suggests. In this section, we take a closer look at this issue.

A steadfast program is a unit $P : \delta \Leftarrow \pi$, and it is defined in the context of a framework $\mathcal{F}(\Pi)$. We require that the symbols of δ have been introduced by adequate (explicit) definitions. If a parameter q of π is a framework-parameter (i.e. q belongs to Π), then the axioms for q are the parameter axioms of the framework. Otherwise q is a program-parameter, and is a new symbol introduced in (an extension to) the framework either by an adequate definition or by a partial definition, as we have seen in Section 2.2.

The composition of two programs $P_1 : \delta_1 \Leftarrow \pi_1$ and $P_2 : \delta_2 \Leftarrow \pi_2$, denoted by $P_1 + P_2$, is defined if $\delta_1 \cap \pi_2 = \emptyset$. We have:

$$P_1 + P_2 : \delta_1 \cup \delta_2 \Leftarrow (\pi_1 - \delta_2) \cup \pi_2$$

Here the framework has an important role, since every symbol used in a program must be defined by the framework. We have two cases. In the first case, the meaning of a symbol q is *completely* characterised by an explicit definition, and we can use q in many different programs, without any further constraint. In the second case, a program-parameter q (of a program P) has been introduced by a partial definition (i.e. set of axioms) D_q. In this case q *cannot* be used as a parameter in other programs, and we have to synthesise programs for q using a synthesis method that also works correctly for partial definitions.[9]

If a set of symbols δ has been introduced in $\mathcal{F}(\Pi)$ by adequate or partial definitions, then a program $P : \delta \Leftarrow \pi$ can be derived directly in $\mathcal{F}(\Pi)$. Another program $Q : \delta \Leftarrow \pi$ could be derived later, in a subclass of $\mathcal{F}(\Pi)$ obtained by framework extension or composition. Both P and Q can be used in the subclass. More generally, in a given class we may have many programs with the behaviour $\delta \Leftarrow \pi$, and we can use any one of them. Correctness follows from the fact that the meaning of the defined symbol has been stated by the framework *once and for all*, and the programs are steadfast with respect to this meaning.

Example 6. Let $\mathcal{F}(Elem, \lhd, \ldots)$ be a framework that inherits $\mathcal{LIST}(Elem, \lhd)$ and uses the merge sort program to perform some other computation. Now, let $\mathcal{F}[\mathcal{C}]$ be a closed instance, where the elements have been instantiated by binary numbers. Then we can restart from the adequate explicit definition of *sort* and derive a new sort program, e.g. a distribution sort program. Since we have derived it starting from the same definition of sort, we can overwrite the old method by this new one.

If δ belong to the framework-parameters Π, then the derivation of P must be postponed, until we have an instantiation of the framework parameters by framework composition (using another open or closed framework).

Example 7. Let $\mathcal{LIST}_1(Elem_1, \lhd_1) = \mu(\mathcal{LIST}(Elem, \lhd))$, where μ is a signature morphism renaming the symbols of $\mathcal{LIST}$. In $\mathcal{LIST}_1(Elem_1, \lhd_1)$ we can

[9] For instance, we have considered this in [10].

define:

$$
\begin{aligned}
X \lhd Y \leftrightarrow {} & \exists B(X|B = Y) \vee \\
& \exists I, a, A, b, B(X = I|a.A \wedge Y = I|b.A \wedge a \lhd_1 b \wedge \neg a = b)
\end{aligned}
$$

We can prove that $\lhd$ is a partial ordering (by using the p-axioms). Therefore $\mathcal{LIST}[\mathcal{LIST}_1]$ satisfies the required proof obligation, and is a framework that inherits the sorting programs. Now we can develop programs to decide $\lhd$, or we can inherit them, if they had been derived in $\mathcal{F}_1$ before the composition.

This discussion shows the central rôle of frameworks in (reasoning about) the correctness of program composition. We believe that this use of frameworks is a new contribution to reasoning about modularity, reusability, and composition.

3.4 Objects

Classes introduced by frameworks in Section 3.1 can only be instantiated in their parameters, i.e. they are like *generic* classes in object-oriented programming, and as such they cannot be used directly to create instances called objects in object-oriented programming. Clearly, in order to implement our methodology as (part of) an object-oriented programming system, we need to have objects. To round off this paper, we make a tentative proposal for objects.

To introduce objects, we distinguish two kinds of frameworks: *adt*-frameworks and *obj*-frameworks. We use *adt*-frameworks to define the basic data types. They may be closed, like natural numbers, reals with a given precision, ASCII characters, and so on; or open, like lists with generic elements $\mathcal{LIST}(Elem)$. The ADTs defined by *adt*-frameworks are typically used (imported) by other frameworks.

On the other hand, we use *obj*-frameworks to define classes of objects. An *obj*-framework is open and the information needed to completely specify its open relations, which we will call *internal relations*, is stored in the objects. Objects are generated (and updated) at run-time, by supplying the missing information.

Example 8. An *obj*-framework looks like:

Framework $\mathcal{SYM}$;

INTERNAL:	$ssym, fsym, rsym$;
SORTS:	$ssym, fsym, rsym$;
FUNCTIONS:	$\ldots : \to ssym$;
	$\ldots : \to fsym$;
	$=, \ldots : \to rsym$;

Framework $\mathcal{SIGN}$;

INTERNAL: $arity, kind$;

IMPORT: $\mathcal{LIST}(Elem = ssym, List = arity)[\mathcal{SYM}]$;

RELATIONS: $kind : (fsym, arity, ssym)$;
$kind : (rsym, arity)$;

I-AXIOMS: $\forall(\exists a, s\, kind(f, a, s))$
$\forall(kind(f, a, s_1) \wedge kind(f, a, s_2) \wedge \neg s_1 = s_2 \rightarrow false)$
$\forall r \exists a\, kind(r, a)$
$\forall A(kind(=, A) \leftrightarrow \exists s\, A = [s, s])$

The above is only a possible syntax to implement our ideas. $\mathcal{LIST}(Elem = ssym, List = arity)$ indicates the result of the signature morphism that renames *Elem* by *ssym* and *List* by *arity*. This *obj*-framework is a template for generating objects of class $\mathcal{SIGN}$. Except for those imported from lists, all the symbols are *internal*: they are like parameters, but the information on them will be held in the objects of the class. Moreover, part of the signature (indicated by ...) is left open and will be completed by the objects. Thus an object of this class will look like:

Object $\mathcal{PA}$-$\mathcal{SIGN} : \mathcal{SIGN}$;

OBJ-SIGNATURE: $Nat \;:\rightarrow ssym$;
$0, s, +, \cdot : \rightarrow fsym$;

OBJ-AXIOMS: $kind(0, A, S) \leftrightarrow A = nil \wedge S = Nat$
$kind(s, A, S) \leftrightarrow A = [Nat] \wedge S = Nat$
$kind(+, A, S) \leftrightarrow A = [Nat, Nat] \wedge S = Nat$
$kind(\cdot, A, S) \leftrightarrow A = [Nat, Nat] \wedge S = Nat$

We assume here the hypothesis of unique names, i.e. $\neg s = +$, $\neg + = \cdot$, and so on.[10] It is easy to see that the *obj*-axioms are the completion of a set of facts, that contain the information on the relation *kind*. The completion has the isoinitial model where the domains correspond to the constants, and the relations to the listed facts. We can also see that the intended model of this object satisfies the *i*-axioms.

This object can be composed with an open *class*-framework $\mathcal{AF}$, that contains the steadfast programs to decide if a given expression is an atomic formula of the signature stored in $\mathcal{PA}$-$\mathcal{SIGN}$; in this way we obtain an object $\mathcal{PA}$-$\mathcal{AF}$ to recognise the atomic formulas of $\mathcal{PA}$. Alternatively, the open *obj*-framework $\mathcal{SIGN}$ can be composed with $\mathcal{AF}$, and then we can build the object $\mathcal{PA}$-$\mathcal{AF}$ exactly in the way shown before, since the internal open symbols are the same.

In an object, the internal information is supplied at run-time and is stored in its internal state. Thus it can be updated. For example, we can add new symbols to the signature of $\mathcal{PA}$, by storing them in the corresponding object.

Thus an *obj*-framework $\mathcal{F}(\text{INTERNAL} : \Pi)$ is an open framework with internal parameters Π, and an object is a closed instance of $\mathcal{F}$, that contains some pieces

[10] The problem of distinguishing e.g. $0 :\rightarrow fsym$ from $0 :\rightarrow Nat$ will be addressed below.

of internal information (lists of facts) that are generated and updated at run-time.

Adequacy of *obj*-frameworks is defined as we have done for open frameworks in Definition 2 (in Section 2.1). There is only the (obvious) difference that now the open symbols Π are the internal open symbols.

Finally, we also have to distinguish two kinds of methods: *class*-methods and *obj*-methods. *Class*-methods correspond to steadfast programs, as we have seen, e.g. steadfast open programs to recognise if an expression is a formula over an (open) signature. Note that it does not matter whether an open symbol is supplied as an external parameter or as an internal state. Therefore we can apply the same deductive synthesis approach, to derive class-methods both in *adt*-classes and in *obj*-classes.

Obj-methods allow us to create and update an object at run-time. Therefore they cannot be compared with steadfast programs. Rather, they correspond to a kind of parameter passing, and consequently we should verify that the i-axioms are satisfied by the actual parameters, i.e. by the information that is introduced in the object. In contrast to the case of framework composition, where the proof obligation on the p-axioms is tested at compile-time, here we have to endow *obj*-methods with programs to test at run-time that the updates give rise to intended models that still satisfy the i-axioms.

For example, updates could be done by using a mechanism like Prolog's *assert* and *retract*. Errors could be tested by suitable programs, with clauses like $error \leftarrow kind(f, a, s_1) \wedge kind(f, a, s_2) \wedge \neg s_1 = s_2$, and updates should be aborted when an error occurs. Moreover, if we allow the introduction of new symbols in the signature at object-level, we need to be able to distinguish between the object-level-signature and the framework-level-signature. We have yet to address these problems fully, and we intend to do so in future.

4 Related Work and Concluding Remarks

In this paper, we have made a preliminary proposal for a methodology for deductive synthesis of logic programs, that is object-oriented in the sense of the common view of object-oriented programming. Our motivation is not merely to jump on the 'object-oriented programming' bandwagon, but to borrow ideas from object-oriented programming for a methodology to construct quality software that is *correctly* reusable and extensible, and not only reliable but formally *correct* wrt specifications. Although it is good for constructing reusable and extensible software, current object-oriented programming technology does not support such formal program development. Our proposed methodology would hopefully be a first step towards this goal, which is of paramount importance in safety-critical applications in real-world software engineering.

To attack the correctness problem, our chief weapon is steadfastness. This is *a priori* correctness, which therefore guarantees correctness of program composition. It is thus the opposite of *a posteriori* correctness that is normally used in module systems based on type theories, where correctness of composition must be verified after composition.

Our notion of classes and inheritance is based on frameworks. In the terminology of conventional object-oriented programming, a steadfast logic program P in a framework $\mathcal{F}$ would be regarded as a *method* in the *class* $\mathcal{F}$. However, a framework captures much more than ADT definitions, as we already mentioned. In fact, a framework essentially characterises a family of classes, or a *generic* class, and in addition contains all the knowledge needed to reason about these classes and steadfastness. Consequently all that can be said and proved in a class can be said and proved in the framework.

Moreover, open frameworks can be composed with closed frameworks, giving rise to a hierarchy of frameworks. We believe such a hierarchy is a better semantic characterisation of type inheritance than the usual inheritance (and multiple inheritance) mechanisms in object-oriented programming which can be very unwieldy (see e.g. [3]).

Our methodology is similar in spirit to, but different in substance from, existing work on modular or object-oriented program construction. Basically, we have raised modularity and reusability to the semantic level. Our characterisation of object-oriented programming is different from that used in object-oriented logic programming, e.g. [11, 15], where the conventional notions of object-oriented programming are imported virtually wholesale, at the syntactic level.

Our characterisation of modularity and reusability is not only different from that in conventional object-oriented programming, but also from that in modular logic programming, e.g. [2, 4]. In the latter, composition is not performed in the context of a framework, and there is no notion of parametric correctness corresponding to steadfastness. Consequently, the notion of correct reusability is also missing there.

However, our work is only in its initial stage, and much more remains to be done. Details of the semantic issues involved need to be worked out, and we have not even begun to investigate the feasibility of implementing the methodology, although we do believe that some object-oriented programming language would provide a suitable vehicle.

Acknowledgements

We wish to thank Laurent Fribourg and Alberto Pettorossi for their insightful comments and constructive suggestions which have helped to improve this paper greatly.

References

1. A. Bertoni, G. Mauri and P. Miglioli. On the power of model theory in specifying abstract data types and in capturing their recursiveness. *Fundamenta Informaticae* VI(2):127–170, 1983.
2. A. Brogi, P. Mancarella, D. Pedreschi and F. Turini. Modular logic programming. *ACM TOPLAS* 16(4):1361-1398, 1994.

3. K.M. Bruce. A paradigmatic object-oriented programming language: Design, static typing and semantics. *J. Functional Programming* 4(2):127–206, 1994. Special issue: Type systems for object-oriented programming.
4. M. Bugliesi, E. Lamma and P. Mello. Modularity in logic programming. *J. Logic Programming* 19,20:443–502, 1994. Special issue: Ten years of logic programming.
5. C.C. Chang and H.J. Keisler. *Model Theory*. North-Holland, 1973.
6. Y. Deville and K.K. Lau. Logic program synthesis. *J. Logic Programming* 19,20:321–350, 1994. Special issue: Ten years of logic programming.
7. *J. Functional Programming* 4(2), 1994. Special issue: Type systems for object-oriented programming.
8. 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*, *Lecture Notes in Computer Science* 883, pages 104–121, Springer-Verlag, 1994.
9. 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.
10. K.K. Lau, M. Ornaghi and S.-Å. Tärnlund. Steadfast logic programs. Submitted to *J. Logic Programming*.
11. F.G. McCabe. *L&O: Logic and Objects*. Prentice-Hall, 1992.
12. B. Meyer. *Object-oriented Software Construction*. Prentice Hall, 1988.
13. B. Meyer. *Eiffel the Language*. Prentice Hall, 1992.
14. P. Miglioli, U. Moscato and M. Ornaghi. Abstract parametric classes and abstract data types defined by classical and constructive logical methods. *J. Symb. Comp.* 18:41-81, 1994.
15. C.D.S. Moss. *Prolog++ The Power of Object-Oriented and Logic Programming*. Addison Wesley, 1994.
16. J. Palsberg and M.I. Schwartzbach. *Object-Oriented Type Systems*. Wiley, 1994.
17. M. Wirsing. Algebraic specification. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 675–788. Elsevier, 1990.

This article was processed using the LaTeX macro package with LLNCS style