From OBJ to ML to Coq

Jacek Chrząszcz[1] * and Jean-Pierre Jouannaud[2] **

[1] Institute of Informatics, Warsaw University, ul. Banacha 2, Warsaw
http://www.mimuw.edu.pl/~chrzaszc
[2] École Polytechnique, LIX, CNRS UMR 7161, F-91400 Palaiseau
http://www.lix.polytechnique.fr/Labo/Jean-Pierre.Jouannaud/

This work is dedicated to our colleague Joseph Goguen, who was extremely influential in the design of modern programming languages.

1 Introduction

Rigorous program development is notoriously difficult because it involves many aspects, among which specification, programming, verification, code reuse, maintenance, and version management. Besides, these various tasks are interdependent, requiring going back and forth between them. In this paper, we are interested in certain language features and in languages which help make the user's life easier for developing programs satisfying their specifications.

Our interest focuses on three implemented specification/programming languages, OBJ [14,18], ML[27] and Coq [10], which have played an important historical role in the process of coming up with better languages. And indeed, both OBJ and ML had many successors or dialects, among which OBJ3 [20], Cafe-OBJ [28], Maude [9] and ELAN [2] for OBJ, and SML [23], CAML [30] and OCaml [29] among others for ML. Coq has evolved with many different versions keeping the same name, following the evolution of type theory from the calculus of constructions [11] to the extended calculus of constructions [22] and the development of the theory of inductive types from Martin-Löf's type theory [25,26] to the calculus of inductive constructions [12,31]. Other proof assistants based on a similar historical development include Lego [21], Alf [24] and Agda/Alfa [1]. Coq remains the most mature and widely used of them all.

We explain briefly in the introduction what important properties are shared by these three languages, and how OBJ has been influential in such a way that many important characteristics of ML and Coq were already present in OBJ, sometimes in disguise. In what sense can these three languages be considered as specification languages, or programming languages, or proof development systems is another important aspect we are interested in.

The user does not like doing things twice. Writing a specification in one language before coding it in another language is more than a challenge: it is helpless. The coding part must be automated as is the case in all three languages we are interested in. This automation obeys the same principle: forgetting the non-executable subpart of the specification or of its proof.

* Partly supported by Polish KBN Grant 3 T11C 002 27
** Project LogiCal, Pôle Commun de Recherche en Informatique du Plateau de Saclay, CNRS, École Polytechnique, INRIA, Université Paris-Sud.

K. Futatsugi et al. (Eds.): Goguen Festschrift, LNCS 4060, pp. 216–234, 2006.

A specification is nothing but a logical property of the form $\forall \overline{x}.P(\overline{x}) \rightarrow Q(\overline{x})$, where $\overline{x}$ is the vector of data, $P(\overline{x})$ is the assumption, and $Q(\overline{x})$ is the conclusion. Therefore, the specification/programming language must contain (possibly via an encoding) a mechanism for expressing properties, as well as one for expressing computations and, possibly, a last one for expressing proofs. In ML, the specification part is simple enough to be inferred automatically by the system from the user's functional program: this is called type inference. The program then satisfies this (extremely poor) specification without requiring any further proof. In OBJ, specifications are algebraic, that is, conditional equations giving meaning to the various functions and predicates introduced by the user, and are executable via rewriting. Showing that the rewrite program implements the specification requires several checks (confluence and termination) left to the user. Proving properties of an OBJ specification can be done in the language itself by using reflexion, this has been done in Maude and Elan, as well as in CafeOBJ — to a limited extent. Coq uses higher-order intuitionistic logic as a specification language, and includes the possibility to carry out the development of a (constructive) proof of the specification by using a tactic language which generates a Coq term representing the proof. A functional program meeting the specification can then be extracted automatically from that proof by erasing all subepressions without computational content which differ from the others by their type.

The old paradigm that the same program piece can be used several times in a bigger program with different data has led to a first notion of abstraction, giving rise to the notion of function, or subprogram. The idea that a program operating upon certain data should not depend upon the way they are actually represented has led to the notion of abstract data type. The same paradigm applied to groups of functions or subprograms achieving some well-defined task, processing some well-defined data, has led to the notion of module. Abstracting over modules themselves has lead to the notion of functor. All three languages have pioneered the design of modules and functors in their respective areas, not to speak about abstract types, and OBJ has been very influential in this matter.

Object-orientation is a different, important abstraction mechanism that is not part of our three languages, and indeed, the first two have been extended so as to include object-oriented features. We will not say more about object orientation, although OCaml, a dialect of ML, played an important role in popularizing object orientation among the community of functional programmers.

Among the programming tasks that should be eased by a good language choice, only the last one, version management, is not taken care of at all by our three languages. Some of the others tasks are better taken care of by OBJ or by ML or by Coq. In particular, the verification principles behind these languages differ in the expressivity of their underlying specification language. In OBJ, typing looks very elementary, since OBJ static types are checked in linear time by a bottom-up tree automaton. But OBJ types are not all static, requiring some runtime type-checking as well. In ML, static typing is more advanced, with a polymorphic type discipline for which types can be inferred by an exponential (but practically linear) algorithm. In Coq, types are arbitrary formulas of higher-order (intuitionistic) logic which can be checked in finite (but indefinite) time, and cannot be inferred in general. This typing system generalizes both OBJ's and ML's

typing as we will see. Verification can also be achieved by model checking or testing. Both are lacking in OBJ, ML and Coq, but can of course be made available as tactics in OBJ's successors and Coq.

The quest for the ideal programming language will continue until a satisfactory language is designed that internalizes features still taken care of by the user or by the programming environment.

2 The Three Languages

2.1 OBJ

In their first landmark paper on CLEAR, Rod Burstall and Joseph Goguen introduced the brand new bright idea that specifying a program required a specific language able to reflect the structure of the problem itself [6]. Following the ADJ group [16,15], they advocated for an algebraic specification language based on equationnal logic, together with a module system in which logical theories could be specified. This was the birth of CLEAR, later developped more formally in [7]. To our knowledge, CLEAR was the very first specification language. CLEAR was algebraic, using many-sorted algebras with error-sorts, an approach later revised to yield OBJ's order-sorted algebras. CLEAR had parameterized modules and theories, but no functors and was not implemented, although one can consider that the first implementation of OBJ by Joseph Tardo [17], a student of Joseph Goguen at UCLA, was indeed an implementation of CLEAR. A second more advanced implementation was then written by David Plaisted when visiting Joseph Goguen at SRI in 1982, which included associative-commutative rewriting.

OBJ2 was the third implementation of OBJ. It was developped in 1984, when Kokichi Futatsugi and the second author visited Joseph Goguen and and José Meseguer at SRI for one year. OBJ2 was the first algebraic specification language based on a fragment of a Horn logic built on the equality predicate and finitely many membership predicates called subsorts [14]. The many novel features of OBJ2 included a flexible user-defined syntax, defining subsorts by Horn sentences, rapid prototyping via rewriting modulo associativity, commutativity, identity and their combinations, parameterized modules and functors. OBJ2 was followed by OBJ3 [20], an improved implementation developped by Claude and Hélène Kirchner whose postdoctoral visit closely followed their advisor's. Full Horn logic is available in the Maude language [9,3], one of OBJ's successors developped by José Meseguer and his collaborators.

An OBJ program is a collection of modules followed by queries. A module is either an *object* or a *theory*. A module has a name, which we always write with capital letters. Objects are made of two parts: a *signature* made of basic types called sorts, and of constructors and (defined) operators for these sorts; the meaning of the operators and of the subsorts is given by (executable) Horn clauses (called *equalities* or *sort constraints* depending on the predicate heading the positive atom). We will also use the name of *membership* for sort constraint, as in Maude. In general, the *principal sort* of a module bears the same name as the module itself, but the first letter only is capitalized. Semantically, objects are initial algebras, implemented via the computation of normal forms: the meaning of the defined operators must be given by a convergent set of conditional

rewrite rules (possibly modulo associativity, commutativity and the like). A theory is much like an object except that it is not executable: its (*loose*) semantics is given by the class of all algebras that satisfy the arbitrary first-order logical sentences specifying its properties. The definition of an object or theory can use other objects or theories. The keywords: *using* allows to import a module without ensuring any property of the imported module which must therefore be copied; *protecting* ensures that the imported module is not modified, making copying unnecessary; *extending* stands in-between, since new values can be added in sorts, but old values cannot be made equal unless they were equal beforehand. *Parameterization* is one more way for importing a module. If T is a theory, parameterizing a module M by an abstract module X satisfying T will allow using the symbols defined in T in order to build M, possibly by using qualification as a disambiguation mechanism. The parameterized module M can later be instantiated by an actual A provided A satisfies the axioms of T. Asserting a module property is done by a *view*, which is the third kind of entity in OBJ. The construction of the instantiated module may also involve some copying.

OBJ has a much more powerful mechanism for defining types than it appears. Besides its basic types called *sorts*, like $\mathbb{N}$ and $List$, it also has type constructors: if the module $LIST$ is parameterized by an abstract module X assumed to satisfy the theory T, then any type $List(Elt)$ exists potentially, provided Elt is the sort of a module satisfying T. This allows to build the types $List(\mathbb{N})$ as well as $List(List(\mathbb{N}))$, therefore providing with some form of polymorphism. However, these types can only be used if the corresponding module instances $LIST[NAT]$ and $LIST[LIST[NAT]]$ are explicitly constructed. The same mechanism provides with dependent types like bounded lists of length n, where n can be a parameter of sort $\mathbb{N}$ defined via a theory. It also has arbitrary first-order Horn sentences as types, written $t : s'$ if A, where A is an arbitrary conjunction of equations and memberships built from the variables in t. OBJ's subsort declaration is a static restriction of this mechanism. So, OBJ's type system was quite strong at the time OBJ was implemented, and has even some Curry-Howard flavour. In retrospect, theories themselves can be seen as types for modules, and a view becomes then an assertion that a module has some theory as type.

OBJ's types, however, only serve specification purposes. Unlike modern functional programming languages like ML, typing is not really internalized in OBJ: property checking is left to the user's responsibility. Still, a limited amount of type-checking is done. For example, the left-hand and right-hand side of an equality must have the same sort. And the expression occurring in the head of a membership must have a sort whose asserted sort must be a subsort.

OBJ specifications are assumed to satisfy a few other properties, all left to the user. For example, the set of rules in a module is supposed terminating and confluent, and the operators should be completely defined. Maude provides support for checking these properties.

2.2 ML

ML was the first functional programming language in which specifications were given (actually, inferred) as types, another novel bright idea from the late seventies due to

Robin Milner [27]. ML has a powerful higher-order module system, an efficient execution model via separate compilation, and a primitive verification mechanism via type inference.

An ML program is a collection of *modules*. A module is either a *structure*, which corresponds to an OBJ non-parametric object, or a *functor* which corresponds to a parametric object. Contrary to the latter, ML functors can be higher order, i.e. they can be parametrized by a module which itself is parametrized. Specification of a functor parameter is given by a *module type*. This can either be a *signature*, corresponding to an OBJ theory, or a functor type. Contrary to OBJ theories, values cannot be specified by equations, but types can.

Another difference is the lack of views in the ML module system. Since subtyping is implicit, a functor *F*, expecting an argument of type *SIG*, can be applied to all modules *M*, whose principal module type *MSIG* is a subtype of *SIG*. Using type inference, the principal module type can be computed efficiently and since subtyping is an extension of inclusion, views are not necessary. On the other hand, the OBJ views can also be used to rename components of an object, which in ML can only be done via a functor.

The important feature of OBJ that is missing in ML is theory extension via keywords *extending* and *using*. Because equational specification of values is lacking in ML, signature inclusion, present in most ML implementations, is much weaker than its OBJ counterpart, hence cannot be seen as a substitute. Indeed, theory extension can be used as another means of parametrisation: assume one declares a function f of some type in a theory A and one then uses it in a subsequent equational specification of some function g; in a theory B extending A, one can then provide equations defining f, therefore completing the specifications of g at the same time. In fact, the specification of g is parametrized by f. Similar ideas are currently being investigate by the ML community with the so called mixins [4,19].

2.3 Coq

In the mid-eighties, following the path initiated by Curry, Howard, Girard and De Bruijn, Thierry Coquand and Gérard Huet made another important step with the beautiful Calculus of Constructions [11], in which types are arbitrary sentences of higher-order intuitionistic logic. This calculus was the start of the language Coq, a proof assistant including a full functional programming language as an executable subset. Coq has a powerful higher-order module system with cut elimination semantics studied and implemented by the first author [8], at that time a phd-student of the second author, a primitive execution model via rewriting and an efficient execution model via compilation. It also includes a sophisticated proof search engine via tactics (and a tactic language), a secure proof checker based on type checking, and an extraction mechanism towards modular ML code. Here, it must be stressed that the module system is used to structure first specifications, then proofs, and finally the programs extracted from proofs. The latter is of course facilited by the fact that the module systems of Coq and ML are essentially the same.

The logical formalism implemented in Coq is based on the calculus of inductive constructions [12,31]. The terms in Coq are of two sorts: calculable `Set` and logical

`Prop`[3]. Values are typed by types, which are typed by the sort `Set` (for example `0 : nat` and `nat : Set`). The second sort, `Prop`, is a type of logical formulas, which in turn are types of their proofs (formula, whose proof is e.g. `fun x ⇒ x`). In type theory with dependent types these two worlds interleave, but it is nevertheless possible to use this dichotomy in order to extract the computable content of a proof, by deleting all its (logical) subterms of sort `Prop`.

The general structure of a Coq development is the same as that of an ML program. The main difference lies in logical parts: axioms in specifications and theorems in implementations. While in ML code precise specifications are usually written informally as comments and correctness is based on trusting the programmer, in Coq one can write specifications as logical formulas, and then carry out the proof that the specification is satisfied.

3 Example

To compare the modular features of the three languages, we shall study a simple sorting algorithm using an abstract priority queue. We also provide a naive implementation of the priority queue and show how the abstract algorithm can be composed with the given implementation. The obtained algorithm and data structure remain parameterized with respect to the element ordering, which can itself be instantiated later on.

Priority queues are data structures implementing the following functionalities: creation of an empty queue, insertion of an element into the queue and extraction of the minimal element from the queue. They can be realized very efficiently imperatively (Fibonacci heaps, binomial heaps, etc) but efficient functional implementations also exist (see e.g. [5]).

Using a priority queue, one can implement the following sorting algorithm: insert all element into the queue and then extract them one by one. Several apparently different sorting algorithms can be seen as instances of this abstract schema using a particular implementation of a priority queue: selection sort uses unsorted lists, insertion sort uses sorted lists and heapsort uses heaps.

This example, despite being so small and simple, illustrates quite well the modular features of our three languages and how they evolved from OBJ to ML and Coq. We show how a specification and an implementation of a data structure look like, how an implementation of the data structure can be composed with an abstract algorithm, and how the resulting concrete but parametric algorithm can be instantiated and used in a program.

Our example shows the advantages of each approach: in OBJ one can write very concise equational specifications, in ML specifications are very brief (and imprecise) but implementations are very efficient, and Coq allows one to formally specify and prove correctness of a data structure or algorithm. The comparison between ML and Coq further shows how much work is needed to formally specify and verify a piece of code.

We will give the actual code of the example in the presentation.

[3] There are other sorts in Coq, namely the predicative hierarchy of $\texttt{Type}_i$, $i \in \mathbb{N}$, called universes [22], but we do not use them in this paper.

4 Priority Queues in OBJ

We will take the liberty to exploit the full power of Maude and use its syntax when appropriate, to ease the understanding. Using OBJ instead would sometimes require some irrelevant detour.

Specification of an ordered type, pairs, queues and priority queues.

We define successively trivial theories with a distinguished sort, pairs, totally ordered sets, queues and priority queues. Being part of any OBJ specification, the predefined module `BOOL` has one sort, `Bool`, two (truth) values, `true` and `false`, and the usual Boolean connectives as operations. In all examples, italics are used to identify OBJ keywords. All sentences are terminated by a dot for parsing purposes. Underscores are used to indicate arguments of operators which use a mixfix syntax.

```
th   TRIV is
sort Elt .
endt
```

The theory TRIV requires the existence of (at least) one sort, named `Elt`.

```
obj  PAIR[X :: TRIV, Y :: TRIV] is
sort Pair .
op   pair : Elt.X Elt.Y -> Pair .
op   1st : Pair -> Elt.X .
op   2nd : Pair -> Elt.Y .
var  E : Elt.X .
var  E' : Elt.Y .
eq   1st(pair(E, E')) == E .
eq   2nd(pair(E, E')) == E' .
endo
```

The parameterized object `PAIR` builds upon two formal objects `X` and `Y` satisfying `TRIV`, which acts as a binder for the sort names `Elt.X` and `Elt.Y`, therefore providing for the polymorphic sort constructor `pair`. Note the use of qualification for disambiguating between the two instances of `TRIV`. The symbol `==` is used for equations in theories and for rules in objects. It is also used for the built-in equality available at all sorts. Similarly, `: s` is the built-in membership predicate available at sort s. In the equations, the variables `E`, `E'` and `E''` are universally quantified by the binding declaration *var*.

```
th   TOSET[X :: TRIV] is protecting BOOL .
op   _ ≤ _: Elt Elt -> Bool .
var  E E' E'' : Elt .
E    E ≤ E == true .
eq   E == E' if E ≤ E' and E' ≤ E .
eq   E ≤ E'' == true if E ≤ E' and E' ≤ E'' .
eq   E ≤ E' or E' ≤ E == true .
endt
```

The theory TOSET uses the module BOOL with the keyword *protecting* implying two important properties: no new element of sort Bool can exist in the semantics (for any two elements e, e' of sort X, e≤e' must be equal to either true or false), and no two elements of sort Bool that were semantically different in BOOL can be equated in TOSET.

```
th       QUEUE[X :: TRIV] is protecting BOOL .
sorts    NeQueue Queue .
subsorts Elt < NeQueue < Queue .
op       empty : Queue .
op       get : NeQueue -> Elt .
op       rest : NeQueue -> Queue .
op       insert : Elt Queue -> NeQueue .
op       eq : Queue Queue -> Bool .
var      Q : NeQueue .
eq       eq(empty, empty) == true .
eq       eq(insert(E, Q), empty) == false .
eq       eq(insert(E, Q), insert(E', Q') ==
         (E == E') and eq(Q, Q') .
eq       eq(insert(get(Q), rest(Q)), Q) == true .
endt
```

In the theory of queues, the declaration NeQueue < Queue implies that get and rest are total on their domain. An alternative is

```
var Q : NeQueue .
mb Q : Queue .
```

```
th   PRIOQUE[X :: TRIV, Y :: POSET[X]] is extending
     PAIR[X, QUEUE[X]] .
op   extract : NeQueue -> Pair .
op   ≤ : Elt Queue -> Bool .
var  Q : NeQueue .
var  E, E' : Elt .
eq   E ≤ nil == true .
eq   E ≤ insert(E', Q) == E ≤.Y E' and E ≤ Q .
eq   extract(insert(E, Q)) == pair(E, Q) if E ≤ Q .
eq   extract(insert(E, Q)) == pair(1st(extract(Q)),
     insert(E, 2nd(extract(Q)))) if E ≤ Q == false .
endt
```

Note how models of PRIOQUE alternate loose interpretations (of TRIV, QUEUE and PRIOQUE) with initial interpretations (of PAIR and BOOL). The role of the PAIR is to provide a polymorphic pairing construct.

Specification of an abstract sorting algorithm based on priority queues.

```
th       LIST[X :: TRIV] is protecting BOOL .
sorts    NeList List .
subsorts Elt < NeList < List .
op       nil : List .
op       _ _ : List List -> List [assoc id : nil] .
op       head : NeList -> Elt .
op       tail : NeList -> List .
var      E E' : Elt .
var      L L' : List .
eq       head(E L) == E .
eq       tail(E L) == L .
mb       L L' : NeList if L : NeList or L' : NeList .
endt
```

```
th       ORDLIST[X :: TRIV, Y :: POSET[X],
         Z :: LIST[X]] is
sorts    NeOList OList .
subsorts NeOlist < OList < List .
subsorts NeOlist < NeList .
op       sorted : List -> Bool .
op       sort : List -> OList .
var      L L' L'' : List .
var      E E' : Elt .
eq       sorted(nil) == true .
eq       sorted(E) == true .
eq       sorted(E E' L) == E ≤ E' and sorted(E' L) .
mb       nil : OList .
mb       L : NeOList if sorted(L) and L : NeList .
eq       sort(L E L' E' L'') == sort(L E' L' E L'') .
eq       sort(L) == L if sorted(L) .
endt
```

Note the subtle use of associativity and identity of concatenation in specifying `sort` and `sorted`.

```
obj   SORT[X :: TRIV, Y :: POSET[X], Z :: PRIOQUE[X, Y]] is
op   sort : Queue -> OList .
var  Q : NeQueue .
eq   sort(empty) == nil .
eq   sort(Q) == 1st(extract(Q)) sort(2nd(extract(Q))) .
endo
```

Concrete algorithms for sorting elements of an ordered set.

```
view QLIST[X :: TRIV] of LIST[X] as QUEUE[X] .
sort Queue to List .
sort NeQueue to NeList .
op   empty to nil
op   get to head .
op   rest to tail .
op   insert to _ _ .
endv
```

This kind of typing assertion implies proof obligations to be checked by the user. Here, the equation given for `insert`, `get` and `rest` must be verified for their interpretation in `LIST`. We now construct specific priority queues as views to instantiate the abstract sorting algorithm.

```
view PRIOQUE1[X :: TRIV, Y :: POSET[X]] of
     PAIR[X, QLIST[X]] as PRIOQUE[X, Y] .
var  L L' : Queue .
var  E : Elt .
op   extract(L E L') to pair(E, L L')
     if  E ≤ L and E ≤ L' .
op   insert(E, L) to E L .
endv
```

```
view PRIOQUE2[X :: TRIV, Y :: POSET[X]] of
     PAIR[X, ORDLIST[X, QLIST[X]]] as PRIOQUE[X, Y] .
var  L : NeOList .
var  L' : OList .
var  E E' E'' : Elt .
op   extract(L) to pair(head(L), tail(L)) .
op   insert : Elt List -> NeList .
eq   insert(E, nil) == E .
eq   insert(E, E') == E E' if E ≤ E' .
eq   insert(E, L E' E'' L') == L E' E E'' L'
     if E' ≤ E and E ≤ E'' .
endv
```

The module `SORT[X, Y, PRIOQUE1[X, Y]]` and the module `SORT[X, Y, PRIOQUE2[X, Y]]` both inherit a sorting algorithm still parameterized by `X`, a set, and `Y`, an order on that set. Applying further to, for example, the built-in module `NAT` of natural numbers having the usual ordering on natural numbers, will generate objects in which we can run the obtained sorting algorithms.

5 Priority Queues in ML

The ML version of our example is given in the Caml [29] dialect. It is divided into four parts: the definition of all needed signatures, a simple implementation of priority

queues as unsorted lists `ListPQ`, an implementation of sorting by an abstract priority queue `PQSort` and composition of both implementations into a sorting module `Sort`.

The first file contains the signatures of an ordered type (consisting of a type and an ordering function), a priority queue and a sorting algorithm. The latter two declare a submodule `E` defining the ordering.

```
module type OrderedType =
  sig
   type t
       (* The type of elements *)
   val compare : t → t → int
       (* compare a b is smaller than 0 if a is smaller than b, 0 if a=b, and is
       larger than 0 if a is larger than b *)
  end
module type PrioQueSig =
  sig
   module E : OrderedType
       (* The type and ordering of the elements of the queue *)
   type t
       (* The type of priority queues *)
   (* Operations: *)
   val create : t
   val insert : E.t → t → t
   val extract : t → t * E.t
       (* raises Not_found if the queue is empty *)
  end
module type SortSig =
  sig
   module E : OrderedType
       (* The type and ordering of the elements to sort *)
   val sort : E.t list → E.t list
       (* The sorting function *)
  end
```

The second file contains the definition of a priority queue based on unordered lists. We skip the (straightforward) implementation here, the only interesting thing is the functor's header:

```
module ListPQ (O: OrderedType)
  : PrioQueSig with module E=O
```

which says that the module `ListPQ` is a functor, taking an order `O` as parameter and returning a priority queue where the ordering is the same as in `O`. Note that since the output signature of this functor is given, its users will only have access to types and functions specified in this signature. Other types and functions are treated as local and implementation specific and therefore they will be inaccessible.

The third element is the abstract algorithm, whose implementation is also trivial. Again the interesting part is the functor's header, which can have two possible forms. The first one is the following:

```
module PQSort1 (O: OrderedType)
               (PQ: PrioQueSig with module E=O)
  : SortSig with module E=O
```

Now, in order to obtain the final sorting algorithm one can do it in OCaml in the following way:

```
module Sort1 (O: OrderedType)
  : SortSig with module E=O
  = PQSort1(O)(ListPQ(O))
```

The module's output signature is the signature of sorting with respect to the argument ordering. Its implementation is simply the composition of existing algorithms, all this under the abstraction with respect to the argument ordering.

There is also a second way of writing the header of the abstract priority queue sorting algorithm:

```
module type PQFunctSig
  = functor (O': OrderedType)
            → PrioQueSig with module E=O'

module PQSort2 (O: OrderedType) (PQF: PQFunctSig)
  : SortSig with module E=O
```

The above code fragment consists of two parts: first the functor type is defined, which corresponds exactly to the specification of `ListPQ`. Then the sorting algorithm is presented as a higher-order functor, i.e. a functor which itself takes a functor as a parameter. Of course, the first line of `PQSort2` is the application of `PQF` to `O` in order to get the priority queue `PQ`, and from this point on the code of both functors is identical.

Higher-order functors are not available in OBJ.

In order to obtain the final sorting algorithm, one applies `PQSort2` to `ListPQ`:

```
module Sort2 (O: OrderedType)
  : SortSig with module E=O
  = PQSort2(O)(ListPQ)
```

The first approach to composing modules is more general than the second, because one does not necessarily have to use a generic priority queue functor. Consequently the use of data structures specialized to a given type is possible (e.g. if a set of values is finite a priority queue can be based on counting elements).

On the other hand, the higher-order functor may correspond better to the intended way the programmer wishes to use a given part of code in the whole program. This is exactly our case, since we want to compose `PQSort` with the generic `ListPQ` *functor*.

Of course it is possible to get the advantages of both approaches: write the most general specification, as in `PQSort1`, and then wrap it in a higher-order functor, presenting the intentions of the programmer:

```
module PQSort2' (O: OrderedType) (PQF: PQFunctSig)
  : SortSig with module E=O
  = PQSort1(O)(PQF(O)).
```

6 Priority Queues in Coq

The structure of the Coq development is the same as in ML, but the signatures now contain formal specifications, and structures contain proofs of desired properties.

The first file, as in ML, contains the definition of all needed signatures. The signatures are preceded by the definition of the type of a three-value proof-carrying comparison: the type `comparison t < = a b` is for example inhabited by `Lt p`, where `p` is a proof of the property `a < b`.

Inductive *comparison* (*X* : *Set*) (*lt eq* : *X* → *X* → *Prop*) (*x y* : *X*) : *Set* :=
| *Lt* : *lt x y* → *comparison X lt eq x y*
| *Eq* : *eq x y* → *comparison X lt eq x y*
| *Gt* : *lt y x* → *comparison X lt eq x y*.

Module Type *OrderedType*.

Parameter *t* : *Set*.

Parameter *eq* : *t* → *t* → *Prop*.
Parameter *lt* : *t* → *t* → *Prop*.

Parameter *compare* : ∀ *x y* : *t*, *comparison t lt eq x y*.

Axiom *eq_refl* : ∀ *x* : *t*, *eq x x*.
Axiom *eq_sym* : ∀ *x y* : *t*, *eq x y* → *eq y x*.
Axiom *eq_trans* : ∀ *x y z* : *t*, *eq x y* → *eq y z* → *eq x z*.

Axiom *lt_trans* : ∀ *x y z* : *t*, *lt x y* → *lt y z* → *lt x z*.
Axiom *lt_not_eq* : ∀ *x y* : *t*, *lt x y* → ¬ *eq x y*.

Hint Immediate *eq_sym*.
Hint *Resolve eq_refl eq_trans lt_not_eq lt_trans*.

End *OrderedType*.

Module Type *PrioQueSig*.

(* Declarations *)
Declare Module *E* : *OrderedType*.

Parameter *t* : *Set*.

Parameter *create* : *t*.
Parameter *insert* : *t* → *E.t* → *t*.
Parameter *extract* : *t* → *option* (*t* × *E.t*).

(* Specification - auxiliary functions and predicates *)

```
  Parameter number : t → E.t → nat .
  Definition empty q : Prop := ∀ x, number q x = 0.
    (* Queues are similar iff q1 = q2 + {x} *)
  Definition similar (q1 q2 : t) (x : E.t) : Prop :=
    (∀ y : E.t, ¬ E.eq x y → number q1 y = number q2 y)
    ∧ (∀ y : E.t, E.eq x y → number q1 y = S (number q2 y)).
  (* Specification of operations *)
  Axiom create_empty : empty create.
  Axiom insert_similar :
    ∀ (q : t) (x : E.t), similar (insert q x) q x.
  Axiom extract_similar :
    ∀ (q q2 : t) (x : E.t),
      extract q = Some (q2, x) → similar q q2 x.
  Axiom extract_minimal :
    ∀ (q q2 : t) (x y : E.t),
      extract q = Some (q2, x) → E.lt y x → number q y = 0.
  Axiom extract_empty_none :
    ∀ q : t, extract q = None → empty q.
End PrioQueSig.

Module Type SortSig.
  Declare Module E : OrderedType.
  Parameter sort : list E.t → list E.t.
  Definition le e1 e2 := E.lt e1 e2 ∨ E.eq e1 e2.
  Axiom sort_sorted : ∀ l : list E.t, Sorting.sort le (sort l).
  Axiom eq_dec : ∀ e1 e2 : E.t, {E.eq e1 e2} + { ¬ E.eq e1 e2}.
  Axiom sort_permut :
    ∀ l : list E.t, Permutation.permutation E.eq eq_dec l (sort l).
End SortSig.
```

The signature `OrderedType`, taken from [13], contains the same calculable elements as its ML counterpart, but is constructed differently. Its main elements are the type and the equality and ordering predicates (i.e. logical elements). The function `compare` is only an addition to the predicates. Instead of an `int`, the `compare` function returns an element of the `comparison` type defined earlier, i.e. the ordering decision together with the proof that the decision is right.

Apart from this, the `OrderedType` signature contains axioms specifying the properties of ordering and equality and hints to instrument automatic tactics, trying to prove properties concerned with the order. The latter element is of course not part of the type theory.

The priority queue signature is also divided into two parts: declarations and specifications. The declarations contain the same elements as in ML with the only exception of the `extract` function, which returns an option type, i.e. `Some` value if the queue is not empty and `None` otherwise (instead of raising an exception). Note, however, that in

order to specify the queue operations one must declare additional functions, counting the `number` of occurrences of a given element in the queue. Based on this function, two predicates `empty` and `similar` can easily be defined in order to write the purely logical axioms specifying how `create`, `insert` and `extract` work.

The signature of a sorting algorithm is simply an extension of its ML counterpart by the logical axioms, saying that the list resulting from sorting is sorted and is a permutation of the input list. The `Sorting.sort` and `Permutation.permutation` predicates from the Coq standard library need additional elements such as less than or equal predicate `le` or equality decidability property `eq_dec`.

In the second file, the header of `ListPQ` is the following:

```
Module ListPQ (O: OrderedType) <: (PrioQueSig with Module E:=O).
```

The difference between the ML and Coq versions of this functor is the way the resulting module type is declared. The Coq syntax Module *M* <: *SIG* means that the type checker should check that the principal signature of *M* is included in *SIG* and the users of *M* are allowed to use all the information inferred in its principal signature. We say that this module type annotation is transparent as opposed to the opaque one that was used in the ML version. The fact the transparent annotation is used is only important for evaluation of programs inside Coq, such as Eval compute in *(sort l)*, see below. Thanks to transparency the reduction mechanism can *see* the definitions of all functions and evaluate them. For typechecking reasons the opaque module type annotations would be equally good.

In Coq, we also have two possibilities of writing the *PQSort* functor. The header of the first order one is as follows:

```
Module PQSort1 (O: OrderedType)
                (PQ: PrioQueSig with Module E := O)
  <: SortSig with Module E := O.
```

Unfortunately, due to the requirement that functors are applied only to *names* of modules, and the lack of local module bindings, the composition of *PQSort1* and *ListPQ* is somewhat lengthy:

```
Module Sort1 (O: OrderedType) <: (SortSig with Module E:=O).
  Module ListPQ_O := ListPQ O.
  Module PQSort_O := PQSort1 O ListPQ_O.
  (* Include PQSort_O. *)
  Module E := PQSort_O.E.
  Definition sort := PQSort_O.sort.
  Definition le := PQSort_O.le.
  Definition sort_sorted := PQSort_O.sort_sorted.
  Definition eq_dec := PQSort_O.eq_dec.
  Definition sort_permut := PQSort_O.sort_permut.
End Sort1.
```

Now we can apply the functor to an example module NatOrder and test the sorting!

```
Module NatSort1 <: (SortSig with Module E:=NatOrder)
  := Sort1 NatOrder.
Eval compute in (NatSort1.sort (4::5::1::2::nil)).
```

The higher-order way of writing *PQSort*

```
Module Type PQFunctSig (O' : OrderedType)
  := PrioQueSig with Module E := O'.

Module PQSort2 (O: OrderedType) (PQF: PQFunctSig)
  <: SortSig with Module E := O.
```

starting with the creation of the priority queue for *O*:

```
  Module PQ := PQF O.
```

leads to a much simpler composition code:

```
Module Sort2 (O: OrderedType)
  <: SortSig with Module E:=O
  := PQSort2 O ListPQ.
```

Unfortunately, due to a certain weakness of the Coq module system with respect to transparency of higher-order functors, the instances of the *PQSort2* functor cannot be evaluated inside Coq. However, the ML code extracted from both functors can of course be evaluated without any problems.

To summarize, it is interesting to compare the size of ML and Coq code. It follows that Coq signatures with specifications by logical formulas are about 2-3 times longer than their commented ML counterparts. Unfortunately, the implementations, which in Coq contain proofs of required properties, are about 10-20 times longer than the corresponding ML code.

7 Conclusion

We have presented three languages which integrate specification and implementation. With the simple example of an abstract sorting algorithm based on a priority queue, we demonstrate how each of the three languages can be used for programming in the large by writing specifications, implementations and by composing abstract components. In particular, we want to stress that parameterization should be available for all kinds of modules.

We have seen that the most important concepts of the OBJ modules are still present in more recent systems such as ML and Coq. Indeed, OBJ objects correspond to structures, parametric objects to functors and OBJ theories to signatures. Only the parametric OBJ theories do not have direct representatives in the ML and Coq module systems, but abstract signatures can easily be refined to concrete ones using the "with" notation. On the other hand, higher-order modules are lacking in OBJ. Although they are not much used in practice, our example shows their adequacy to describing dependencies on other parametric components.

Concerning the ability of these languages to specify and implement software components, OBJ lies somewhere between ML and Coq. In ML, specifications are simply given as types for functions, and execution is based on an efficient call-by-value evaluation strategy. In OBJ, one can write first-order equational and membership specifications that are executable via an efficient built-in associative-commutative rewriting mechanism guided by user-defined strategies. In Coq, the specification language is higher-order predicate logic, which is by far the most expressive of the three. This makes it possible to write a specification, implement it, prove that the implementation is correct, run the implementation inside Coq and even extract the program into an executable ML code. Some of these steps may of course involve complex, lengthy machine computations.

The question arises of which language is best suited for fast prototyping. If no verification is needed, the answer would probably be ML. Separating signatures from their actual implementation is just very neat, and allows a two steps development methodology which does not require much interaction between these two phases unless there are major design errors. Because OBJ modules provide at the same time with an interface and logical requirements for the interface, specification and coding are no more clearly separated. The development process becomes more complicated, going back and forth between different pieces of the code. A comparison with Coq is more difficult, since Coq gives you a lot more: while it is possible in OBJ to forget about the proof obligations generated when typing modules, this is not the case with Coq. A consequence is that every change requires tedious adjustments of the proofs.

Acknowledgments: We thank Andrzej Gąsienica-Samek and Tomasz Stachowicz for their help with the Coq development, Pierre-Yves Strub for checking preliminary versions of the OBJ development in Maude, and the referee for many valuable comments.

References

1. The Agda proof assistant. http://www.cs.chalmers.se/~catarina/agda/.
2. Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Marian Vittek. ELAN: A logical framework based on computational systems. In J. Meseguer, editor, *1st International Workshop on Rewriting Logic and its Applications, Electronic Notes in Theoretical Computer Science 4*, 1996.
3. Adel Bouhoula, Jean-Pierre Jouannaud, and José Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236:35–132, 1999.
4. Gilad Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. PhD thesis, Dept. of Computer Science, University of Utah, 1992.
5. Gerth Stolting Brodal and Chris Okasaki. Optimal purely functional priority queues. *Journal of Functional Programming*, 6(6):839–857, 1996.
6. Rod M. Burstall and Joseph A. Goguen. Putting theories together to make specifications. In *Proc. 5th International Joint Conference of Artificial Intelligence, Cambridge Massachusetts*, pages 1045–1058, Edinburgh University, 1977.
7. Rod M. Burstall and Joseph A. Goguen. The semantics of CLEAR, a specification language. In *1979 Copenhagen Winter School on Abstract Software Specification*, volume 86 of *LNCS*. Springer-Verlag, 1980.

8. Jacek Chrząszcz. Modules in Coq are and will be correct. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *Types for Proofs and Programs, International Workshop, TYPES 2003, Torino, Italy, April 30 - May 4, 2003, Revised Selected Papers*, volume 3085 of *LNCS*, pages 130–146. Springer, 2004.
9. Manuel Clavel, Steven Eker, Patrick Lincoln, and José Meseguer. Principles of Maude. In J. Meseguer, editor, *1st International Workshop on Rewriting Logic and its Applications, Electronic Notes in Theoretical Computer Science 4*, 1996.
10. The Coq proof assistant. http://coq.inria.fr/.
11. Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76:95–120, February 1988.
12. Thierry Coquand and Christine Paulin-Mohring. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *COLOG-88: International conference on computer logic*, volume 417 of *LNCS*. Springer-Verlag, 1990.
13. Jean-Christophe Filliâtre and Pierre Letouzey. Functors for Proofs and Programs. In *European Symposium on Programming*, volume 2986 of *LNCS*, pages 370–384, Barcelona, Spain, April 2004. Springer-Verlag.
14. Kokichi Futatsugi, Joseph A. Goguen, Jean-Pierre Jouannaud, and José Meseguer. Principles of OBJ2. In *Proc. 12th ACM Symp. on Principles of Programming Languages, New Orleans*, 1985.
15. J. A. Goguen, J. W. Thatcher, and E. G. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In *Current Trends in Programming Methodology, vol. 4*, pages 80–149. Prentice Hall, 1978.
16. J. A. Goguen, J. W. Thatcher, E. W. Wagner, and J. B. Wright. Initial algebra semantics and continuous algebra. *Journal of the ACM*, 24(1):68–95, January 1977.
17. Joseph A. Goguen and Joseph J. Tardo. An introduction to obj, a language for writing and testing formal algebraic specifications. In *Specification of Reliable Software Conference*, pages 170–189, April 1979.
18. Joseph A. Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. *Applications of Algebraic Specifications Using OBJ*, chapter Introducing OBJ*. Cambridge University Press, 1993. D. Coleman, R. Gallimore and J. A. Goguen, eds.
19. Tom Hirschowitz and Xavier Leroy. Mixin modules in a call-by-value setting. In D. Le Métayer, editor, *Programming Languages and Systems, ESOP'2002*, volume 2305 of *LNCS*, pages 6–20. Springer-Verlag, 2002.
20. Claude Kirchner, Hélène Kirchner, and José Meseguer. Operational semantics of OBJ3. In *15th International Conference on Automata, Languages and Programming*, volume 317 of *LNCS*, pages 287–301. Springer-Verlag, 1988.
21. The LEGO proof assistant. http://www.dcs.ed.ac.uk/home/lego/.
22. Zhaohui Luo. ECC an Extended Calculus of Constructions. In *4th Symposium on Logic in Computer Science*, Pacific Grove, California, 1989.
23. David MacQueen. Theory and practice of higher-order type systems or the Standard ML type system. Copy of Transparencies.
24. Lena Magnusson and Bengt Nordström. The alf proof editor and its proof engine. In H. Barendregt and T. Nipkow, editors, *Types for Proofs and Programs*, volume 806 of *LNCS*, pages 213–237. Springer-Verlag, 1993.
25. Per Martin-Löf. An intuitionistic theory of types: Predicative part. In H. E. Rose and J. C. Sheperdson, editors, *Logic Colloquium '73*, volume 80 of *Studies in Logic*, pages 73–118. North-Holland, 1975.
26. Per Martin-Löf. *Intuitionistic Type Theory*. Biblioplois, Napoli, 1984. Notes of Giowanni Sambin on a series of lectues given in Padova.
27. Robert Milner. A theory of type polymorphism programming. *Journal of Computer and System Sciences*, 17, 1978.

28. Shin Nakajima and Kokichi Futatsugi. An object-oriented modeling method for algebraic specifications in Cafe OBJ. In *19th International Conference on Software Engineering*, pages 34–44. ACM Press, 1997.
29. The Objective Caml language. http://caml.inria.fr/.
30. Pierre Weis et al. The CAML reference manual. Rapport de Recherche 121, INRIA, 1990.
31. Benjamin Werner. *Méta-théorie du Calcul des Constructions Inductives*. PhD thesis, Univ. Paris VII, 1994.