CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE
LABORATOIRE d'ANALYSE et d'ARCHITECTURE des SYSTÈMES
7, Avenue du Colonel Roche, 31077 Toulouse Cedex 4, France
Tel: 05 61 33 62 00 Fax: 05 61 55 35 77 Telex: LAASTSE.520930F

OO Programming styles in ML

Bernard Berthomieu

email: Bernard.Berthomieu@laas.fr

LAAS Report # 2000111
March 2000

Abstract: This report discusses encoding of objects and OOP idioms into ML, or more generally into programming languages equipped with parametric polymorphism, recursive type operators, and abstract types. It is shown that the essential OO concepts and idioms, including inheritance and dynamic dispatch, can be encoded in this well understood framework, without requiring any operational or typing extensions of ML, that is in much simpler operational and typing frameworks than previously proposed. These encodings are unusual in essentially two aspects: They do not rely on records, neither simple nor extensible, but upon tuples instead, and they do not rely on subtyping and subsumption, but on an encoding of inheritance polymorphism into parametric polymorphism. Classes are encoded as abstract types through SML's structures. The encodings can readily be used for OO programming in ML, or can serve as reference interpretations of object features in dialects of ML extended with OO features

Contents

1 Introduction

1.1 OOP in an ML framework

This report discusses encoding of objects and OOP idioms into ML. It is shown that it is indeed possible to encode most of object orientation features in SML, without any additional features, and with a quite tractable complexity, though SML does not offer any specific support for this programming style.

There have been several proposals to add OOP features to ML, with or without specific extensions, notably [6] [19] [18] [8]. Compared to these proposals, our encodings are unusual in several aspects:

- Like the encoding of Thorup/Tofte [19], and Ocaml [18] (if object coercions are omitted), but unlike Moby [8], our encoding does not require subtyping to handle inheritance polymorphism. Instead, inheritance polymorphism is encoded into ML's parametric polymorphism. This is conceptually close to the approach of Ocaml [18]; their treatment, based on extensible records and row types, takes advantage of the fact that field selection is polymorphic.
- To represent objects, unlike all of the above notations, our encoding does not make use of records, neither simple, nor extensible, but simply of ML tuples and ML's recursive type operators. The required operational and typing frameworks are thus much simpler than those previously proposed, avoiding in particular row types and recursive types. Method invocation is expressed as function calls (messages are functions). In addition, for the simple dynamic dispatch case equivalent to that provided by [19] or [18], the wrappings of [19] are totally avoided.
- Our notation is class-based; classes are encoded as abstract types through SML structures with opaque signatures. Encoding classes does not require any extension to SML's module system either, and these features interact naturally with SML's modularity mechanisms.

The key of our family of object models is a hierarchical encoding of objects into tuples that allows in turn encoding inheritance polymorphism into ML's parametric polymorphism. Basing inheritance polymorphism on parametric polymorphism rather than on subtyping allows to avoid in most programming situations the problems resulting from non-coincidence between subtyping and subclassing.

The encodings can readily be used for OO programming in ML (unextended) or can serve as a reference interpretation of object features in dialects of ML extended with OO features. They also have some pedagogical virtues for those familiar with ML but not with OOP, as well as for those with opposite background.

1.2 Plan

The report is organized in 7 sections following this introduction, and an Appendix. Some prior basic knowledge of ML and OOP concepts and idioms are assumed.

Our object models naturally fall into two families, according to whether methods are embedded or not within objects. These two families do not exclude one another, but rather constitute the two ends of a spectrum of encodings. The simpler "nonembedded methods" encoding is developed in Sections 2 and 3, the "embedded methods" version follows in Section 4. For each encoding, we review the essential OOP concepts, and interpret them in our framework. Some extensions of ML are discussed too, that would ease encoding some features. We use as running example a sample class hierarchy presented in Appendix A1, with Ocaml and Java code. Sample ML code illustrating the various encodings for our example hierarchy constitute Appendix A2.

Sections 5 to 7 survey a number of issues not developed in the preceding sections. Section 5 addresses the problem of multiple inheritance, which is not canonically supported in our models. A variant of our encodings making use of subtyping is discussed in Section 6. Additional issues related to OOP are briefly reviewed in Section 7, such as virtual classes, instance variables, and alike. Section 7 also discusses possible support from an extended ML environment to put these encodings to work. Section 8 is the conclusion.

2 Objects, Classes, Inheritance

2.1 Incremental development

2.1.1 OO development

OOP is about defining hierarchies of data structures called objects, and primitives for their manipulation, called methods. The objects and methods at some level in the hierarchy are typically defined in modularization units called classes. Classes are organized hierarchically. Objects defined in derived classes enrich objects built from ancestor classes in the sense that they have more structure. Enriched objects may be manipulated via the methods of their ancestor classes (those they inherited), and also by specific methods taking advantage of their specific components. Derived classes may specify new methods as well as specialized versions of existing, inherited, methods. Further, adding new layers should not require recoding existing layers.

How does this differ from the usual modular design practice as applicable in e.g. SML? The differences lie in the way data structures are enriched. The natural means in ML of enriching a data structure is instantiating it e.g. integer lists (of type `int list`) enrich polymorphic lists (of type `'a list`); functions defined on polymorphic lists are applicable to integer lists as well. In OOP, one typically enriches a data structure by adding new constituents to it. E.g. a colored point enriches a point by adding to it a "color" attribute; points and colored points are typically monomorphic.

So, objects at some level in an object hierarchy should be seen are "partially-specified", or "enrichable", values, that is values of which only part of the components, or attributes, are known when writing the methods that manipulate them, but some other components may be missing. Methods at some level of specification take only advantage of the object components known at that level of specification. Later refinements of specifications may concern the structure of objects themselves (by addition of new components), and/or the set of methods that operate upon them (by addition of new methods and/or specialization of inherited methods).

2.1.2 Object hierarchies

Let us try to encode in ML some enrichable structures, in the above sense. That is, structures to which adjunction of information would preserve the position of existing components.

The simplest form one can give to such an enrichable structure is that of a pair, with its left-hand side containing the currently specified information, and its right-hand side reserved for future enrichments. At some level of specification, future enrichments are unknown, but it must still be possible to build objects of that level. For this, we introduce the following datatype:

```
datatype closed = %;
```

Values of type `closed` may be understood as "unspecified". An example of value not totally unspecified is a point, defined with a single "position" attribute, represented by an integer:

```
type point = int * closed;
val p = (456,%) : point;
```

Later enrichment of points will make precise a content for point extensions. One may define for instance colored points and shaped points as follows:

```
type cpoint = int * (color * closed)
val cp = (456,(red,%)) : cpoint;

type spoint = int * (shape * closed)
val sp = (456,(square,%)) : spoint;
```

Colored points and shaped points are enrichments of points in the sense that they make precise the "unknown" component of points. Note that, in these extended point types, we still keep explicit an "extension" component, to be able to later enrich these objects themselves. Let us rewrite our examples with some enlightening type abbreviations:

A root object is a totally unspecified structure (without attributes):

```
type 'a subroot = 'a;
type root = closed subroot;
```

A *point* extends a root object with a *position* attribute represented by an integer:

```
type 'a subpoint = (int * 'a) subroot;
type point = closed subpoint;
```

A *colored point* enriches a point with an extra *color* attribute, represented by a color value (of type `color`, left undefined for the moment):

```
type 'a subcpoint = (color * 'a) subpoint;
type cpoint = closed subcpoint;
```

Parametrically, a cpoint (for instance) can be seen as either a `subcpoint` with a closed extension, a `subpoint` with an extension of type `(color * closed)`, or as a `subroot` with extension of type `(int * (color * closed))` since these three types are equal. But it cannot be seen as a `point`, nor as a `root` object, nor a `point` can be seen as a `subcpoint`.

To make precise relations between objects, we could introduce a "more-specified-than" relation $\subset$, defined as the least reflexive and transitive relation including $\tau \subset$ `closed`. Alternatively, if ML supported existential types, we could define objects in terms of their possible enrichments by:

```
type root = ∃ 'a. 'a subroot
type point = ∃ 'a. 'a subpoint
etc,
```

2.1.3 Methods as extension-parametric functions

In that model of objects, the methods at some description level are just functions operating on the representation of objects as made precise at that level of description (aware only of attributes specified at that level and above).

Now, if we want methods of some level to be applicable to enriched objects as well, then they ought to be polymorphic on all possible object extensions, rather than applicable to "closed" objects only.

For example, here is how pos and slide methods for points can be defined; note that they operate polymorphically on subpoints:

```
fun pos ((p,_) : 'a subpoint) = p;
fun slide ((p, ext):'a subpoint) n :'a subpoint = (p+n,ext);
```

Similarly, methods col and paint below operate polymorphically on subcpoints. Method col returns its color, and paint builds a new colored point from an existing point and a color, borrowing all attributes from its point argument, except its color.:

```
fun col ((_,(c,_)) : 'a subcpoint) = c;
fun paint ((p,(_,e)) : 'a subcpoint) c : 'a subcpoint = (p,(c,e));
```

Defined this way, functions pos and slide are applicable to colored points as well as to points. Similarly, col and paint are applicable to any enrichments of cpoint (i.e. to all instances of 'a subcpoint), but they are not applicable to points. Note that pos would not be applicable to colored points if its domain was restricted to point (if it had type point -> int).

Obviously, nothing prevents us defining methods that operate on several object parameters rather than a single one. A typical example of such methods is an equality binary method. We define here two versions of it, for points and cpoints (overloading, or grouping these three functions into a single eq method applicable to any point, will be discussed later):

```
fun peq (p:'a subpoint) (q:'b subpoint) = pos p = pos q;
fun cpeq (p:'a subcpoint) (q:'b subcpoint) = peq p q andalso col p = col q;
```

Note that none of these equality methods requires access to the internal states of objects since all information to compute equality is available through existing methods. This is not true in general, however. In terms of types, the two equality methods receive types as general as possible, in particular, their object arguments need not have the same type.

2.1.4 Classes as Structures, Subclassing, Inheritance

The standard view of classes is that of groups of declarations keeping together routines for creating and manipulating objects at the same specification level. Extension of specifications is done by defining subclasses from existing classes, from which one can create objects sharing their methods (inheriting them) with objects of the ancestor classes (also called super-classes).

Classes are conveniently encoded as ML structures. For regularity, we use the same type names in each class (self, subself) for designating the type of objects defined in that class and the (polymorphic) type of their descendant objects rather than specializing the names to reflect the class (root, subroot, point, subpoint, etc.). Conventionally, the structures encoding classes define an object constructor new, types self and subself, and, in addition, may introduce methods.

E.g. class Root specifies no methods, it just provides generator new and types self and subself:

```
structure Root = struct
   type 'a subself = 'a;
   type self = closed subself;
   fun new () : self = %;
end;
```

Class `Point` is a subclass of class `Root`, in addition to `new`, `self` and `subself`, it specifies `pos` and `slide` methods. Inheritance is made explicit by defining type `Point.subself` from type `Root.subself` (points enrich root objects).

```
structure Point = struct
   type 'a subself = (int * 'a) Root.subself;
   type self = closed subself;
   fun pos ((p,_) : 'a subself) = p;
   fun slide ((p, ext):'a subself) n :'a subself = (p+n,ext);
   fun new p : self = (p,%);
end;
```

Before refining our notion of class, let us briefly discuss an unusual aspect of our encoding. The fact that our encoding does not rely on records is a noticeable difference with previous encoding of objects in ML (e.g. that of Thorup/Tofte) or dialects of ML supporting objects (Ocaml, Moby). An obvious consequence is that messages are functions, rather than labels in a specific syntactic class. This difference can be considered minor, having messages as functions just forces the user to choose some naming discipline for messages to avoid hiding messages by other values; SML structures and qualified identifiers are perfectly adequate for that purpose.

The notion of inheritance we used so far is "simple" inheritance, by which a class can inherit from at most one superclass. Some notations allow multiple-inheritance, in which a class is allowed to inherit messages and methods from ancestor classes themselves unrelated by subclassing, what our notation does not permit. The issue of multiple-inheritance will be addressed in a specific section.

2.2 Encapsulation, Classes as abstract types

This naive model of classes exposed in Section 2.1.4 suffers from some drawbacks that we solve here by packaging classes into abstract types. Several devices are available in ML for achieving this encapsulation, including abstypes and structures with opaque signature ascription. Though more verbose, the latter is retained for its flexibility. Encapsulation of objects is explained in two steps: packaging of attributes, then object encapsulation.

2.2.1 Hiding attributes of objects

A first drawback of the naive model of classes of Section 2.1.4 is that the internal state of objects is revealed. This allows derived classes to manipulate directly the state of higher-level classes, which is not good programming practice. Worse, this may make objects unrelated by subclassing appear related since the types representing local states are public. E.g. if we defined type color as type int, and a "shaped points" class derived from points with shapes represented by integers too, then shaped points would not be structurally distinct from colored points!

Both these problems are solved by encapsulation of derived classes into structures with opaque signature ascription hiding the type of objects attributes. E.g. let us rewrite type `subself` in class `Point` as follows, making explicit a type function `ext`:

```
type 'a ext = int * 'a;
type 'a subself = 'a ext Root.subself;
```

Then encapsulation of the declarations of the updated Point class, hiding representation of component ext, prevents derived classes from manipulating the position of points directly (since its exact representation is not visible), and solves the "parasitic-subclassing" problem (since each class introduces a unique ext type operator).

There is a price to pay though: since the partial state at some level n is only known inside the structure defining level-n objects, creating an object of level n with some attributes to be introduced at levels above (towards root) cannot be done entirely at level n. E.g. no information on the actual representation of position information is available to the Cpoint class, so the new constructor of class Cpoint (enriching Point) has to invoke some routine of class Point to install position information.

To solve this packing problem, we could define derived objects directly from existing objects. This would work for our current encoding, though it would not for all encodings to come. A more flexible alternative is to introduce in each class a dedicated packing function behaving as follows:

- Given a tuple $(a_1, \ldots, a_n)$ of attributes for all specification levels from root (1) to current (n), and an extension ext, pack $(a_1, \ldots, a_n)$ ext builds an object with these attributes and extension. For this, the pack routine invokes the packing routines found in higher layers. Generator new at some level is easily implemented from pack of the same level by passing it the relevant parameters and the "closed" extension, as will be seen.

2.2.2 Hiding objects structure

A second drawback of the model of classes of Section 2.1.4 is that the structure of objects is visible. Hiding the details of representation of partial states does not forbid one to build a fake object by borrowing state components from several objects of the same type, that is without invoking the new generator of the class. To solve that particular problem, we need to hide the overall tree structure of objects.

Since, using the pack routines just introduced, all objects are ultimately built with the Root.pack function, it suffices for this to hide the representation of type Root.subself in class Root. But this has an annoying side-effect: Since the structure of objects is now unknown to derived classes, and the most recently introduced attributes are the innermost in object representations, derived classes have no means to access their own state information!

So, at this point, we are facing a dilemma: we can't totally hide object representation, as it would be hidden to derived classes as well (preventing to define their methods), but if that representation is not hidden then one may build fake objects. This is indeed a well known problem in OO development. One should distinguish two interfaces for classes: one for using them, visible to everyone and including the class generator and methods, and one for extending them, visible only within derived classes, and used under the control of the compiler.

As long as classes are encoded at user-level rather than at compiler level, as is the case here, it is impossible to hide from users the "system interface" needed to build derived classes. So, rather than trying to solve the "fake object" problem, we will just make precise the required system interface for derived classes and isolate it from other components of classes.

Assuming objects' representations are hidden by the above means, what do we need to allow derived classes to access state of their own objects? We introduce two other packing related functions for this: each class is now equipped with an unpack routine and, for convenience, with a repack routine. Function unpack must be visible to subclasses, but repack need not be.

- Applied to an object, unpack returns a pair (f,ext) in which ext is the pair constituted of the partial state and the extension relevant at the current level of specification, and f is the function that rebuilds the original object when applied to ext.

- Given a function g that modifies the partial state of an object, and an object p of the relevant level, repack g ob builds an object with its partial state s replaced by state (g s), and other partial states (at higher or lower levels) copied from object p. At level n, repack makes use of the unpack routine of level (n-1). repack results in a new object being built, rather than a physical modification of an existing object, but it preserves the partial states of upper and lower layers.

The unpack and repack routines, together with the previous pack routine, constitute the *system interface* of classes, as opposed to the *user interface* constituted of generator new and methods. Ideally, system interfaces should be only used by a compiler, to produce the code below from class specifications in some dedicated notation, this particular issue will be briefly discussed in Section 7.

2.2.3 Classes as Abstract Types

Using encapsulation, together with the packing routines, our object hierarchy is now built as follows.

Datatype closed holds the single value %, as before. We also make use of two auxiliary functions defined as follows.

```
datatype closed = %;
fun explode (f,(s,ext)) = (fn e => f (s,e), ext);
fun implode s (f,(_,ext)) = (f (s,ext));
```

The following code shows the definition of the improved Root class. The representation of type 'a subself is hidden. As root objects have no attributes, the repack function is omitted in class Root, and the attribute parameters of Root.pack and Root.new are empty. Function unpack returns the identity function paired with the object itself:

```
signature ROOT = sig
   type 'a subself
   type self = closed subself
   val unpack : 'a subself -> ('a -> 'a subself) * 'a
   val pack : unit -> 'a -> 'a subself
   val new : unit -> self
end;

structure Root :> ROOT = struct
(* types *)
   type 'a subself = 'a
   type self = closed subself
(* packing *)
   fun unpack ob = (I,ob);
   fun pack () ob = ob;
(* new *)
   fun new () = pack () %;
end;
```

Next, this is how points are encapsulated. Point.unpack and Point.repack make use of Root.unpack; Point.pack invokes Root.pack. Type ext links types Point.subself to type Root.subself, its representation is hidden by the signature. Note how repack is used in method `slide` to create a new subpoint with an altered state from an existing subpoint. Methods `slide` does not rely on `new` for this, since `new` produces a closed point while `slide` must preserve the type of its self parameter:

```
signature POINT = sig
   type 'a ext
   type 'a subself = 'a ext Root.subself;
   type self = closed subself
   val unpack : int -> ('a -> 'a subself) * 'a
   val pack : int -> 'a -> 'a subself
   val pos : 'a subself -> int
   val slide : 'a subself -> int -> 'a subself
   val eq : 'a subself -> 'b subself -> bool
   val dist : 'a subself -> 'b subself -> real
   val new : int -> self
end;

structure Point :> POINT = struct
(* types *)
   type 'a ext = int * 'a;
   type 'a subself = 'a ext Root.subself;
   type self = Root.closed subself;
(* packing *)
   fun unpack p = explode (Root.unpack p);
   fun repack f p = implode f (Root.unpack p);
   fun pack n xt = Root.pack () (n,xt);
(* messages *)
   fun pos p = let val (_,(n,_)) = Root.unpack p in n end;
   fun slide p n = repack (fn x => x+n) p;
   fun eq p q = pos p = pos q;
   fun dist p q = real (pos q = pos p);
(* new *)
   fun new n = pack n Root.%;
end;
```

Abstract type packaging of classes induces a clear distinction between external functions and methods: method are functions that have access to the internal state of objects (at a single level of description), e.g. function `pos` could not be defined as an external function. Function `Point.eq`, on the other hand, could be moved outside the structure `Point` since it can be implemented in terms of the publicly available `Point` methods.

Another benefit of the packing routines is that they make defining a class from its superclass a more regular process. This regularity even suggests a functorial construction of classes, in which one would define classes as functors parameterized by their superclass. This particular issue will be investigated in depth in Section 5.

In terms of typing, hiding object representations, together with messages as functions, leads to strong differences with record-oriented notations. Consider for instance the function `f= fn p => Point.pos p`. Then, in our treatment, `f` may only be applied to objects of class `Point` or one of its descendant classes. In record-based notations, e.g. in Ocaml, the type of `f = fun p -> p#pos` is more general; `f` accepts as argument any object having a method labelled `pos`, whether or not the class of that object is a descendant of `Point`.

Though our concern so far is not about performance, it is clear that using the packing routines is more expensive than using pattern matching on object parameters as was done in Section 2.1.4. There are a number of possible solutions to improve efficiency (such as, where applicable, using a simplified version of `unpack` that does not compute the object reconstruction function). For ease of explanation, we choose to minimize the number of such auxiliary functions rather than trying to improve efficiency.

2.3 Specialization

Method specialization refers to the fact that a method may have different behaviors according to the type of self and, possibly, according to the type of other object arguments of the method as well.

There are several obvious candidate methods for such specializations in our example: `Point.slide` moves a point along a single direction. In a class `D2point` of bidimensional points, derived from `Point`, a specialized version of `slide` could move bidirectional points along both directions simultaneously. Binary methods `eq` and `dist` are other candidates. `Point.eq` would compare points for positions, `Cpoint.eq` would in addition compare them for colors, etc.

Structures and qualified identifiers provide a first means to manage such specialized methods. Specialized methods are just redefined in the relevant structures, possibly using the inherited versions they specialize in their definition. Qualified identifiers allow to make precise the intended version to be used in each context. With this technique, the method invoked at some point is entirely determined from the static information available at that point, this is *static dispatch*.

This is, for instance, how a class `D2point` of bidimensional points can be defined, inheriting of the `Point` class. It specializes (redefines) methods `slide`, `eq` and `dist`, also found in class `Point`:

```
signature D2POINT = sig
   type 'a ext
   type 'a subself = 'a ext Point.subself
   type self = closed subself
   val unpack : 'a subself -> ('a -> 'a subself) * 'a
   val pack : int * int -> 'a -> 'a subself
   val slide : 'a subself -> int -> 'a subself
   val eq : 'a subself -> 'b subself -> bool
   val dist : 'a subself -> 'b subself -> real
   val ypos : 'a subself -> int
   val new : int * int -> self
end;

structure D2point :> D2POINT = struct
(* types *)
   type 'a ext = int * 'a;
   type 'a subself = 'a ext Point.subself;
   type self = closed subself;
(* packing *)
   fun unpack p = explode (Point.unpack p);
   fun repack f p = implode f (Point.unpack p);
```

```
   fun pack (y,x) ext = Point.pack x (y, ext);
(* messages *)
   fun ypos p = let val (_,(y,_)) = Point.unpack p in y end;
   fun slide p n = repack (fn y => y+n) (Point.slide p n);
   fun eq p q = Point.eq p q andalso ypos p = ypos q;
   fun dist p q = sqrt (sq (real (ypos q - ypos p) + sq (Point.dist p q));
(* new *)
   fun new (x,y) = pack (y,x) %;
end;
```

Most OO notations provide an alternative view of specialization: instead of having as many messages as specialized methods, as we do, one has a single message dynamically bound to some method according to the type of self (a technique known as *simple dynamic dispatch*) or, possibly, according to the type of other object arguments of the message as well (refered to as *multiple dynamic dispatch*). In our example, for instance, there would be a single `eq` message defined in class `Point`, and that message would fetch a method that behaves like `Cpoint.eq` when applied to two subcpoints, like `D2point.eq` when applied to two subd2points, or like `Point.eq` otherwise.

The particular issues of dynamic dispatch will be addressed in the next two chapters. It requires significant enhancements of our current model of objects that we wish to defer. Briefly, implementing dynamic dispatch requires some indirection in method lookup. If methods are kept in the execution environment, then dynamic dispatch amounts to *dynamically resolved overloading*; keeping them within objects yields a class of *embedded-methods* models. Section 3 discusses the first approach (overloading), and Section 4 addresses embedding of methods within objects.

Though not in the mainstream of OO notations, we believe that the "static dispatch" approach may still be valuable, so let us try to do without dynamic dispatch for the moment. The next section discusses one of the consequences of this choice.

2.4 Parameterized Classes as Functors

When trying to encode our full sample hierarchy of Appendix A1 with our technique, we are faced with a difficulty for encoding class `Line`: class generator `Line.new` is intended to build a line from a pair of points of class `Point` or any of its descendant classes. All methods of class `Line` depend on the methods of the particular point subclass to which belong line ends, and that point subclass is not statically known (it is only known that it is a subclass of `Point`). E.g. the `shift` method of class `Line` uses the `slide` method of the particular points involved.

This is indeed a typical case where dynamic dispatch is useful. With overloading and dynamic dispatch, method `shift` would move a line using `Point.slide` when built from a pair of objects of class `Point`, or using `D2point.slide` if built from points of class `D2point`, without requiring the user to specify the right version of `slide` to use. But let us see if there are static dispatch alternatives.

In absence of overloading and dynamic dispatch, as mentioned, specializing a method is obtained by redefining methods in classes. To achieve this for Lines, we may see class `Line` as a parameterized structure, or functor, rather than simply as a structure. Class `Line` can be defined as a functor that takes as argument some subclass of `Point`. This way, `Line` methods can be defined in terms of the methods of the parameter structure rather than from the generic `Point` class methods. Generator `Line.new` is still polymorphic, but functor application freezes the versions of `slide`, `eq` and `dist` methods used to define `Line` methods. Of course, several `Line` classes can be defined by functor application.

To define functor Line, we first need to make some minimal assumptions about its class parameter. These are summarized by the following signature:

```
signature MINIPOINT = sig
   type 'a subself
   type self = closed subself
   val slide : 'a subself -> int -> 'a subself
   val eq : 'a subself -> 'b subself -> bool
   val dist : 'a subself -> 'b subself -> real
end;
```

Next, class Line has the following interface, in which structure Param is the particular subclass of Point passed as argument of functor Line.

```
signature LINE = sig
   structure Param : MINIPOINT
   type ('a,'b) ext
   type ('a,'b) subself = ('a,'b) ext Root.subself
   type 'a self = ('a,closed) subself
   val unpack : ('a,'b) subself -> ('b -> ('a,'b) subself) * 'b
   val repack : (('a Param.subself * 'a Param.subself) ->
                 ('a Param.subself * 'a Param.subself)) ->
                ('a,'b) subself -> ('a,'b) subself
   val pack : ('a Param.subself * 'a Param.subself) -> 'b -> ('a,'b) subself
   val from   : ('a,'b) subself -> 'a Param.subself
   val to     : ('a,'b) subself -> 'a Param.subself
   val shift  : ('a,'b) subself -> int -> ('a,'b) subself
   val eq     : ('a,'b) subself -> ('c,'d) subself -> bool
   val length : ('a,'b) subself -> real
   val new    : 'a Param.subself * 'a Param.subself -> 'a self
end;
```

Finally, this is how functor Line is defined:

```
functor Line (structure P : MINIPOINT) :> LINE where Param = P = struct
   structure Param = P;
(* types *)
   type ('a,'b) ext = ('a Param.subself * 'a Param.subself) * 'b;
   type ('a,'b) subself = ('a,'b) ext Root.subself;
   type 'a self = ('a,closed) subself;
(* packing *)
   fun unpack p = explode (Root.unpack p);
   fun repack f p = implode f (Root.unpack p);
   fun pack ft ext = Root.pack () (ft, ext);
(* messages *)
   fun from l = let val (_,((f,_),_)) = Root.unpack l in f end;
   fun to l = let val (_,((_,t),_)) = Root.unpack l in t end;
```

```
    fun shift l n = repack (fn (f,t) => (Param.slide f n,Param.slide t n)) l;
    fun eq p q = Param.eq (from p) (from q) andalso Param.eq (to p) (to q);
    fun length l = Param.dist (from l) (to l);
 (* new *)
    fun new (f,t) = pack (f,t) %;
end;
```

Particular line classes are obtained by applying functor `Line`. The argument class of `Line` must provide methods `eq`, `dist` and `slide` (those which are used within functor `Line`). If the particular point subclass used as argument of `Line` does not redefine one of them, then it must be copied from the `Point` class (e.g. the `Cpoint` class does not redefine `dist` nor `slide`).

```
structure PointLine =
      Line (structure P = Point);
structure CpointLine =
      Line (structure P = struct open Point; open Cpoint; end);
structure D2pointLine =
      Line (structure P = D2point);
```

Class `PointLine` allows one to build pairs of objects which are points or descendants of point. `PointLine.new` is polymorphic, but the `slide`, `eq`, and `dist` methods captured in the methods of `PointLine` are those of class `Point`. So, when building a `PointLine` from two points of class `D2point`, then its length is simply the 1-dimensional distance between its extremities (through `Point.dist`). On another hand, class `D2pointLine` allows one to build pairs of objects which are points or descendants of `D2point`, and its length method computes the 2D distance between the line ends (through `D2point.dist`).

2.5 Summary

This concludes our first family of object encodings, not relying upon embedded methods nor dynamic dispatch. Our full sample hierarchy packed with structures with opaque signatures, and with static dispatch, can be found in Appendix A2.2.1.

Let us summarize the important steps:

- Objects are represented by particular trees. The leaves of the tree carry components of the state; one particular leaf is left “unspecified”, and materializes the place where extra attributes are found in objects refining the structure of the current one. Objects enrich other objects higher in the hierarchy by having more attributes, but still carry an “unspecified” field, to preserve extension capabilities. One will notice that object representations are non-recursive. Whether the impossibility of defining global state recursively is a limitation of the encoding is left unanswered at the time; we conjecture it is not. This does not prevent in any case to build objects with recursively defined partial states, as some examples will show in time.
- Methods are arranged to be polymorphic at the place of extensions in their arguments. Methods apply to “open-ended” types, as opposed to “closed” ones, they are polymorphic in object enrichments. This allows them to be applied to objects at all sublevels of the level at which methods are defined. As shown, we could achieve the sharing of methods in a hierarchy of classes without recourse to subtyping and subsumption, simply by encoding inheritance by parametricity. Another benefit of

parametric classes is that we do not suffer the problem of loss of type information; methods preserve self's type. In essence, our treatment of inheritance is similar to that of Ocaml: Ocaml row variables in classes play a role similar to that of our extension variables in subobject types.

- Object types and methods are encapsulated in ML structures with opaque signature ascriptions hiding the details of implementation of object state. Structures and qualified identifiers also allow some form of method specialization, but dispatch is static, the particular version of a method used in any context is made explicit by the user. Though some would say this is not OO, we still believe the approach is of interest.
- An advantage of methods as "external" functions (as opposed to embedded) is that there is no polymorphism loss for binary methods (e.g. `Point.eq`, `Point.dist`, etc. have types as general as possible).

Finally, we could achieve so far a modest account of OO, with implementation inheritance, inheritance polymorphism, but without dynamic dispatch, this without any extension to the type system or operational semantics of SML. The solution obviously interacts well with SML features. The next chapter discusses in length a first technique for achieving dynamic dispatch, namely the "dynamically resolved overloading" technique. The following will discuss the alternative "embedded-methods" models.

3 Nonembedded Methods and Overloading

3.1 Overloading

Overloading, or adhoc polymorphism, allows one binding several values (called versions in the sequel) to the same name, selected from their types. Various forms of overloading are found in programming languages Hope, SML, Haskell, ADA and many others. At first sight, this is just what we need to add to the "nonembedded-methods" model of the previous section to provide dynamic dispatch. Unfortunately, none of the forms mentioned is adequate in our context. Let us check the specific requirements in details:

- First, to preserve the incremental aspect of specifications, overloading should be *open*. Users must be allowed to add versions to an overloaded operator at any time (or place in the class hierarchy).
- Next, the overloaded versions should be allowed to be *polymorphic* and have their types overlap, since we will typically overload methods (which are polymorphic) with new versions whose types instantiate the type of some existing version. Since version types overlap, *resolution,* the process of binding a version of an overloaded function to an occurrence of an overloaded identifier, should obey some "best-match" strategy.
- Finally, resolution should be *delayed* at run-time, as opposed to performed at compile-time. E.g. if function `f = (fn p => fn q => eq p q)` is bound at a place where class `Cpoint` is not in the environment yet, then, after declaring this latter class and specializing `eq` for colored points, the call `(f cp cq)` should return true if colored points `cp` and `cq` are equal by colored point equality, and not simply by point equality which was the only version of `eq` bound when f was declared.

Clearly, none of the mentioned forms of overloading satisfy these conditions. In its simplest form, as found e.g. in SML, it satisfies none of these. The last two conditions amount to some form of type directed dynamic binding. Operationally, this requires some indirection in method lookup, either within the values themselves, or within the execution environment. This also requires some encoding of type information within the objects themselves.

The work of Catagna et al. [6] precisely addresses the form of overloading we need. Their work results in a new foundational calculus called λ&-calculus, in which overloading is in some sense built-in, including dynamic resolution and best-match version selection. An ML-style language equipped with that form of overloading is being developed [2].

Our purpose here is more modest, it is simply to encode OO features in existing ML, as much as possible. We show in the next sections that, in our particular context, and to some extent, this form of overloading can be simulated in ML.

3.2 Simulating dynamic overloading by assignable method dictionaries

The overall idea to simulate dynamic resolution overloading is to maintain method dictionaries into ML references. These methods would operate on objects packed with some (possibly partial) type information to allow type directed dynamic version selection. Directories are updated when methods are specialized.

In addition, we must be able to wrap objects packed with their type information into some monotype, simply to allow storing versions in ML references (which must be monomorphic). The stored methods must operate on wrapped or monotype arguments and must return wrapped or monotype results. All overloaded versions of some method must have the same type, which is the type of the most general version.

Method dictionaries described this way are not different from methods performing pattern matching on type information (or on constructions reflecting the types of values). The wrapping and packing operations on arguments, and then the opposite operations on results, can be performed by the message functions, prior to invoking methods.

Techniques for packing type information with objects, wrapping them into some monotypes, and encoding objects to take advantage of these, are discussed in the next sections. Let us assume for the moment that some techniques to do this are available. More precisely, let us assume available for each object a `wrap` function (of type `self -> wrap`, where `self` is the type of the object and `wrap` is a monotype), and for each class a `self'wrap` function (of type `wrap -> self`, where `self` is the type of objects created in this class). Function `self'wrap` is assumed to raise exception `Wrap` if its argument cannot be unwrapped to an object of that class.

Method dictionary are created with the `new_method` function below. `new_method` is applied to the first version f of a method (which should be the least specific version, in terms of requirements from its object arguments).

```
fun new_method f =
    let val M = ref f
        fun get () = !M
        fun over f = M := f (get ())
    in (over,get)
    end;
```

Version f must have a monotype (its arguments and results are wrapped/unwrapped by the corresponding message, if necessary), it constitutes the initial dictionary. `new_method f` returns two functions: `get` and `over`: `get()` returns the method (dictionary) currently stored, and `(over g)`, where `g` is some function, replaces the current dictionary `f` by `(g f)`.

Let us illustrate overloading by the `eq` example in our point hierarchy. The dictionary for `eq` is created in class `Point` by a call to `new_method` applied to the initial eq method (that checks position equality). This call returns a `get_eq` access function and an `overload_eq` update function. Message `eq` fetches the eq dictionary and applies it to its wrapped arguments, function `position` extracts the position information from an unwrapped point:

```
structure Point :> POINT = struct
   ...
   val (overload_eq, get_eq) = (* creates overloadable eq method *)
      let fun peq (p : wrap) (q : wrap) : bool =
             position (point'wrap p) = position (point'wrap q)
      in new_method peq
      end;
   fun eq p q = get_eq () (wrap p) (wrap q);
   ...
end;
```

Next, this is how method `eq` is specialized in e.g. class `Cpoint`. One first defines a specialized equality dictionary for cpoints, then asks to replace the current dictionary by the specialized one. Function `cpeq` takes the current dictionary `peq` as first parameter and returns a new dictionary. Passing the current dictionary as parameter of `cpeq` allows specialized methods to invoke the current (less specialized) methods in their definition (cpeq is defined in terms of the current equality peq and color equality). The most specialized version must be tried first. If it fails (which is checked by handling the Wrap exception), then the previous dictionary is invoked.

```
structure Cpoint :> CPOINT = struct
   ...
   val _ = (* overloads peq with cpeq *)
        let fun cpeq peq p q =
             (col (cpoint'wrap p) = col (cpoint'wrap q) andalso peq p q)
              handle Root.Wrap => peq p q
        in Point.overload_eq cpeq
        end;
   ...
end;
```

The next section discussed wrapping techniques.

3.3 Wrapping techniques

We are seeking type-safe methods allowing to cast an object together with some encoding of its type into a monomorphic value, say of type wrap, and to later recover the original object from that wrapped value, or part of the information in it.

Wrapping is a general technique that is used for many purposes in the literature. Several means are possible to achieve wrapping, depending on the exact requirements. We will discuss three of them which are suitable for our purposes: folding, existential wrapping, and subtype wrapping. The last two require extensions of ML, the first does not.

3.3.1 Folding

Folding is close to the technique simply called "wrapping" in [19]. We tried to give it a more specific name. The idea of folding is to wrap objects into objects of their superclass; this is a form of coercion. If this is achieved for each class, then any object can be ultimately wrapped into a root object, so type wrap is simply type root.

Actually, it is quite simple to define functions to, say, fold a cpoint into a point or to a root object, and back. From our notations and declarations of Section 2.1, let's redefine type closed , and introduce type wrap, as follows:

```
datatype closed =
      %
    | Point of int * closed
    | Cpoint of color * closed
    | Spoint of shape * closed
    | ... ;
type wrap = root;
```

Now, consider a cpoint cp:

```
val cp = (123,(red,%));
```

Then the value `(123,Cpoint(red,%))` obtained from `cp` by packing its second constituent with constructor `Cpoint`, clearly has type `point`, and the value `Point(123,Cpoint(red,%)))` has type `root` (it has type `wrap` too). The following declarations are well typed:

```
val w : wrap = Point (cp, Cpoint (cp, %));
let val Point (n, Cpoint ce) = w in col (n,ce) end;
let val Point ne = w in pos ne end;
```

With our notations of Section 2.2 this time, assuming that class states are monomorphic, then each class can be equipped with a wrapper `Self` for its local state and extension. Using these wrappers, each class can be further equipped with two functions: `wrap'self` and `self'wrap`. These functions have types:

```
Self : closed ext
wrap'self : self -> wrap
self'wrap : wrap -> self
```

For each class `C`, the combination `C.self'wrap o C.wrap'self` is the identity function on subobjects of this class. In addition, for any pair of objects, say of classes `Thisobj` and `Thatobj`, one can also define function `thisobj'thatobj` as `Thisobj.self'wrap o Thatobj.wrap'self`. That function should succeeds if `Thisobj` is an ancestor class of `Thatobj`, but it should fail otherwise. We assume a `Wrap` exception is raised in that case.

Folding in the nonembedded model of Section 2 is as simple as above, as long as object states are monomorphic. To be able to use the packing routines to fold and unfold objects, it is just necessary to slightly relax the type of unpack to `'a subself -> ('b -> 'b subself) * 'a`, disconnecting in its result type the type of the extension from that of the argument of the reconstruction function. This is always possible if the type of class extensions does not occur in the type of object states. As will be seen later, folding in the embedded model of Section 4 is a bit more complicated as object states capture methods, but it can still be done.

So, using folding one can coerce an object of some class into an object of any other class (and not only to a particular `wrap` type). That operation succeeds only if the destination class is an ancestor of the originating class (exception `Wrap` is raised otherwise). Note that, conversely to type casts, or to subsumption in type systems with subtyping, coercions here do generate some code.

An important interest of folding is that it does not require any extension of ML, neither of its type system, nor operational. Another important interest is that it is reversible, without loss of type information. Its drawbacks are its operational cost and the constraint that class states must be monomorphic (one could think of using a polymorphic `wrap` type to handle objects with polymorphic states, but that solution does not appears scalable).

3.3.2 Existential Wrapping

Compared to folding, existential wrapping, in both the versions discussed here, does require extensions of ML, though these have been thoroughly investigated and may be considered minor extensions.

The first technique, existential value wrappers, follows from the existential datatypes of [10]. Some implementations of ML offered existential types in that setting [12]. In that setting, type wrap can be defined as follows (variables `'e` are treated as existential type variables):

```
datatype wrap =
     %%
    | Point of 'e subpoint * wrap
    | Cpoint of 'e subcpoint * wrap
    | Spoint of 'e subspoint * wrap
    | ... ;
```

Consider now the same cpoint cp as before. Then the following declarations are well typed:

```
val w : wrap = Point (cp, Cpoint (cp, %%));
let val Point (_, Cpoint (p, _)) = w in col p end;
let val Point (p, _) = w in pos p end;
```

Objects in wrapped values occur at several places, with their packed types increasing in preciseness with their depth in the value. Pattern matching on wrapped values allows one recovering part of the type of objects, but not all that type since wrappers are polymorphic. Compared to folding, existential value wrapping is thus not reversible. However, though scope rules for existential type variables prevent to extract a point from its wrapped value, they do not prevent extracting part of the information found in the point, as the above examples show (see [10] or [12] for details on scoping of existential variables).

The second existential wrapping technique is closely related to the one above, it is based on dynamic values, which are values packed together with a type. Details on dynamics in ML can be found in e.g. [11]. Dynamics have been provided in some versions of ML too.

The particular version of dynamics we are using here could be called “existential” dynamics; it is the particular case of the general version discussed in [11], in which the type variables in type tags are treated existentially. For our purposes, that restriction makes it very close to the wrapping technique just explained: it allows one to match a dynamic value against a dynamic pattern (value and type) with its type possibly holding type variables.

Briefly (the notations below are ours), one builds a dynamic value by encapsulating a value and a type (called the tag of the value) with the **at** constructor. The type variables occurring in the tag, if any, have local scope and are considered existentially quantified. The construction is well typed if the type of the value packed matches the type packed (so, in general, the packed type is less precise than the exact type of the packed value).

For using it as a wrapping technique, type wrap is simply bound to type `dynamic`:

```
type wrap = dynamic;
```

Considering the same cpoint cp as before, the following declarations are well-typed:

```
val w : wrap = cp at 'e Cpoint.subself;
let val Point (p at 'e subcpoint) = w in col p end;
let val Point (p at 'e subpoint) = w in pos p end;
```

Dynamics in this setting have similar wrapping and unwrapping capabilities than the previous existential value wrapping technique. A significant advantage over both folding and existential value wrapping, however, is that wrapping functions are much simpler since wrapped values are obtained by a single (**at**) construction, as opposed to the previous iterated wrapping processes. The effects of the previous embedded wrappers (ultimately aimed at reconstructing the type of objects) are here obtained by the "natural" matching relation on types.

So, we will favor the use of dynamics in our examples, rather than that of existential constructors. Finally, to prevent any confusion, it must be stressed that, in spite of the name "dynamic", the type information packed within dynamic values is determined statically; it is not the run-time type of the value that is packed with the value, but simply the type mentioned in the code! In this sense, dynamic typing is simply a particular instance of static typing.

3.3.3 Subtype Wrapping

Subtyping will be discussed in details in a later chapter. We mention it here as it provides yet another wrapping technique.

Type wrap is defined as follows:

```
datatype wrap =
      %%
    | Point of Point.self * wrap
    | Cpoint of Cpoint.self * wrap
    | ... ;
```

Considering the same cpoint cp as before. Then, assuming subclasses yield subtypes, the following declarations are well-typed:

```
val w = Point (cp, Cpoint (cp, %%)) : wrap
let val Point (_, Cpoint (p, _)) = w in col p end;
let val Point (p, _) = w in pos p end;
```

Subtype wrapping is used exactly as existential value wrapping was above. It may replace wrapping with dynamics when, for some other reasons, subtyping is assumed available (we will use it in Section 6).

3.3.4 Preserving incremental development

With any of the wrapping techniques we discussed (except wrapping with dynamics, notably), a wrapper constructor is needed for each class defined. Since we want to preserve the incremental development method essential to OOP, these wrappers cannot be declared in a single datatype declaration, as done above for readability. Wrappers should be declared in their relevant classes.

To do this, we need extensible datatypes. Though some variants of ML supported extensible types (Caml, at some point, it is also a scheduled feature of ML2000 [13]), these are not generally provided in ML, except for the type exn of exceptions. Technically, introduction of extensible datatypes in ML is a minor extension. For our purpose, however, and since wrappers must be monomorphic, we could safely implement the required extensible types in terms of the sole extensible datatype available in ML: the type of exceptions.

Using exceptions as object wrappers is rather inelegant though. A convenient alternative, improving the clarity of specifications, is to define extensible datatypes distinct from the type of exceptions by taking "copies" of the exn type, as follows.

Assuming the usual option type operator (defined by **datatype** 'a option = Some **of** 'a | None), consider the following signature and structure:

```
structure Extensible = struct
   type extensible = exn;
   fun create _ =
      let exception X of 'a
          fun matches (X y) = Some y | matches _ = None
      in (X, matches)
      end;
end;

signature EXTENSIBLE = sig
   type extensible;
   val create : 'a option -> (('a -> extensible) * (extensible -> 'a option))
end;
```

Then every opaque ascription of signature EXTENSIBLE to structure Extensible introduces a new extensible datatype, disjoint from all existing datatypes, e.g. :

```
structure Extensible1 = Extensible :> EXTENSIBLE;
type extensible1 = Extensible1.extensible;
structure Extensible2 = Extensible :> EXTENSIBLE;
type extensible2 = Extensible2.extensible;
```

etc,

Function create in these structures introduce a new constructor for the relevant extensible datatype and returns functions to build values with this constructor and to destructure them. Function create is typically called with argument (None:ty option) in which ty is the type of the arguments of the constructor created.

This construction is adequate for our purposes, it will be used in our encodings whenever we need some extensible datatype. Note however that, as presented above, the construction does not allow one defining 0-ary constructors. For this purpose one could add a specific creation function in structure Extensible; an alternative is to encode 0-ary constructors as 1-ary constructors with some constant argument (e.g. "()"). More importantly, the construction does not allow building parameterized extensible types, but our encodings do not require them.

3.4 Dynamic Dispatch in the Nonembedded Model of Objects

We now have all the necessary ingredients to propose a working version of specialization with dynamic dispatch in the nonembedded model. For this purpose, we need to combine one of the wrapping techniques of Section 3.3 with the encoding of dynamically resolved overloading of Section 3.2.

3.4.1 Using folding

Starting from the specifications of Section 2.2, classes are first extended with incremental folding as explained in Sections 3.3.1 and 3.3.4. Each class is assumed equipped with functions `wrap'self` and `self'wrap`; the `Self` wrappers themselves are local to classes. `Self` wrappers and constant `%` belong to an extensible datatype `closed`, defined as follows using the technique explained in Section 3.3.4:

```
signature XCLOSED = sig
   type closed;
   val create : `a option -> ((`a -> closed) * (closed -> `a option))
end;
structure ExtensibleClosed = Extensible :> XCLOSED;

open ExtensibleClosed;
val % = let val (x,_) = create (None : unit option) in x () end;
```

Objects are replaced by *object packets*. An object packet is a triple `(v,w,u)` of type `self packet` in which `v` is the object itself, of type `self`, `w` is its folding function (of type `self -> wrap`) and `u` is its unfolding function (of type `wrap -> self`).

For convenience a `packet` type function is defined, as follows:

```
type 'a packet = 'a * ('a -> wrap) * (wrap -> 'a);
```

Methods operate on object packets, rather than on objects, e.g. method `Point.pos` has type `'a Point.subself packet -> int`. Generating functions `new` produce object packets as well, their definitions typically look like:

```
fun new args : self packet = (pack args  %, wrap'self, self'wrap);
```

Class `Root` is defined as follows. The type of `unpack` is updated as explained Page 22.

```
signature ROOT = sig
   type wrap
   exception Wrap
   type 'a subself
   type self = closed subself
   type 'a packet = 'a * ('a -> wrap) * (wrap -> 'a);
   val wrap'self : self -> wrap
   val self'wrap : wrap -> self
   val unpack : 'a subself -> ('b -> 'b subself) * 'a
   val pack : unit -> 'a -> 'a subself
   val new : unit -> self packet
end;

structure Root :> ROOT = struct
(* types *)
   type 'a subself = 'a;
   type self = closed subself;
   type wrap = self;
```

```
    type 'a packet = 'a * ('a -> wrap) * (wrap -> 'a);
  (* packing *)
    fun unpack ext = (I,ext);
    fun pack () ext = ext;
  (* wrapping *)
    exception Wrap;
    fun wrap'self p = p;
    fun self'wrap p = p;
  (* new *)
    fun new () = (pack () %, wrap'self, self'wrap);
  end;
```

Class `Point` extends class `Root`. Beside standard contents (type abbreviations, packing, wrapping, new), class Point declares the specializable `eq` method and initializes it with (simple) point equality. The other `Point` methods may be specialized too but, for readability, only the code relevant to wrapping and method eq is shown below. To allow specialization of method `eq`, function `overload_eq` must be visible in the `Point` class interface.

```
  signature POINT = sig
    ...
    val wrap'self : self -> Root.wrap
    val self'wrap : Root.wrap -> self
    val unpack : 'a subself -> ('b -> 'b subself) * 'a
    ...
    val overload_eq : ((Root.wrap -> Root.wrap -> bool) ->
                       (Root.wrap -> Root.wrap -> bool)) -> unit
    val eq : 'a subself Root.packet -> 'b subself Root.packet -> bool
    ...
    val new : int -> self Root.packet
  end;

  structure Point :> POINT = struct
    ...
  (* wrapping *)
    val (Self,matches) = create (None : (int * closed) option);
    fun wrap'self p =
      let val (f,ext) = Root.unpack p in Root.wrap'self (f (Self ext)) end;
    fun self'wrap w =
      let val (f,ww) = Root.unpack (Root.self'wrap w)
      in case matches ww of Some ext => f ext | _ => raise Root.Wrap
      end;
  (* messages *)
    fun wpos p = let val (_, (x,_)) = Root.unpack p in x end;
    val (overload_eq, get_eq) = (* declaring overloadable eq *)
      let fun peq (p : wrap) (q : wrap) : bool =
               wpos (self'wrap p) = wpos (self'wrap q)
      in new_method peq
      end;
```

```
    fun eq (p,wp,_) (q,wq,_) = get_eq () (wp p) (wq q);
    ...
  (* new *)
    fun new () = (pack () %, wrap'self, self'wrap);
end;
```

Class Cpoint specializes method eq. Again, only the code relevant to specialization of `eq` is shown:

```
structure Cpoint :> CPOINT = struct
    ...
  (* messages *)
    fun wcol p = let val (_,(c,_)) = Point.unpack p in c end;
    val _ = (* overloads eq *)
        let fun cpeq peq p q =
                (wcol (self 'wrap p) = wcol (self'wrap q) andalso peq p q)
                 handle Root.Wrap => peq p q
        in Point.overload_eq cpeq
        end;
    ....
end;
```

Implementing wrapping by folding is convenient as long as object states are not polymorphic. Folding functions are more tedious to implement that some alternatives for acheiving wrapping, but it has two important advantages: it does not require any extensions of ML, and it is reversible. Reversibility implies that methods returning objects, like `Point.slide` or `Line.shift`, can be specialized: the versions of these methods return wrapped objects, which are unwrapped within the message functions. Now, the constraint that object states be monomorphic has some annoying consequences; in our examples, it prevents to fold objects of class `Line` as its generator should be polymorphic (as in Section 2.4). Indeed, the only way left to be able to fold lines is to define them monomorphic (that is, the state of a line is a pair of “closed” objects of type `Param.self packet`, rather than a pair of subobjects of type `'a Param.subself packet`). We can still use a functorial construction similar to that of Section 2.4, however, to reintroduce some genericity, but some is necessarily lost.

Our full hierarchy encoded with the nonembedded model and overloading based on folding is shown in Appendix A2.2.2. It slightly differs from the above presentation by introducing an intermediate `Eqobj` class, ancestor of all classes using message eq. This is indeed necessary if we want `Points` and `Lines` share message `eq`. Also, for simplicity, the methods returning object attributes, like `pos` or `col`, are not declared “overloadable”. The reason is not technical, but simply that it would be pedantic to allow overloading of, say, `pos`. In standard OO practice, these methods would be considered instance variables rather than actual methods.

3.4.2 Using Alternative wrapping techniques

Wrapping with dynamics offers an interesting alternative that greatly simplifies the previous specification by removing all needs to maintain wrapping utilities in each class. This requires an extension of ML, as mentioned, but the idea was taken further for its simplifying benefits. Compared to folding, it suffers an important defect though, as already discussed: wrapping with dynamics is not reversible. This implies that methods returning objects, like `Point.slide` for instance, cannot be specialized, as their results cannot be unwrapped to the type of `self`. On another side, existential or dynamics wrapping allow wrapping objects with polymorphic state, conversely to folding.

An analogue of the previous folding-based specification, but based on dynamics, can be found in Appendix A2.2.3. Concerning objects representation, the object packets used in this version do not carry any "unwrapping function" component. Furthermore, it does not overload methods returning objects, but uses static dispatch to handle these, with method redefinitions in the relevant classes, as in Section 2.3.

Subtype based wrapping, on another hand, is simply not applicable here, as the wrapping functions packed with objects prevent subclasses to yield subtypes.

3.5 Summary

At this point, we could implement dynamically dispatched specialized methods without specific support from the compiling environment. The technique relies on associating with objects some information that reveal their type, and wrapping them into some monotype. Several wrapping techniques were investigated, each has some merits and pitfalls. A number of points have been just overlooked and may need precisions, including:

- Version ordering: The order in which versions are tried is not arbitrary, versions are ordered from most specialized to most general. This ordering is only partial however, there may be several "most specialized versions", which can be tried in sequence, but there must be a single "most general" version, tried last. To avoid any `Wrap` exception raised by a method, the most general version must be able to handle without error the most general case allowed by the type of the corresponding message. e.g. since message eq has type
  ```
  'a Point.subself Root.packet -> 'b Point.subself  Root.packet -> bool,
  ```
  the most general eq version stored must not rely on other object attributes than those present in objects of class `Point`. Best-match selection is discussed in length by Castagna [6] in the framework of their λ&-calculus.
- As one may have noticed, wrapping could also be used to solve the typing problem for inheritance polymorphism by using their wrapped forms as the sole representation of objects. Doing so would obviously prevent to detect as compile time many specification errors like objects invoking methods they do not possess. These checks would be performed at run time instead.
- It may be noticed too that the overloading technique does not make a special case for simple dispatch, what some embedded-methods encodings do (this will be seen in Section 4). There is here a uniform treatment of simple and multiple dispatch.

Overall, encoding method specialization by dynamically resolved overloading is complex. Though the encoding proposed would be adequate in most cases, some support from the programming environment is certainly desirable. The next section discusses a handy alternative, that of embedding methods within objects. Indeed, specialization with dynamic dispatch justifies the model of objects with embedded methods discussed in Section 4. This alternative also has some pitfalls, as will be seen.

4 Embedded methods and Dynamic Dispatch

We explore in this section another model of objects, in which methods are embedded within the objects rather than defined as “external” functions. This is indeed a classical approach, except that our variant does not rely on records. The tuples of methods introduced in some class constitute now the state of objects in that class.

The “parametric inheritance” notion introduced in Section 2 is kept unchanged. Technically, the level of indirection gained by embedding methods will permit to easily implement method specialization and simple dynamic dispatch by method update.

4.1 Object states as method suites, self-application

Concerning method invocation, and using the terminology of [1], we develop the “self-application” approach, as opposed to the “recursive-record” or the “state-application” approach (these two variants can be expressed in our framework as well). At that point, this is mostly an aesthetic decision. Among these views, the self application is certainly the easiest to comprehend and use; the complexity of the two others are more difficult to justify in absence of subtyping (subtyping will be addressed later).

In the self-application approach, methods are implemented from embedded functions, called pre-methods, that take as first parameter the object in which they are found (conventionally called *self*). In our setting, messages are functions that fetch pre-methods in the object they are applied to (self), and apply self to these pre-methods to obtain a method.

As in Sections 2 and 3, we stick to functional objects. This is purely for ease of explanation and generality; imperative objects are less demanding than functional ones in terms of type checking.

As a starting point, an “embedded methods” version of our example hierarchy of Appendix A2.2.1 is given in Appendix A2.3.1. That version does not make use of dynamic dispatch; specialized methods are redefined in the relevant classes.

Datatype `closed` and class `Root` are defined exactly as in Appendix A2.2.1. Class `Point` is defined as follows:

```
signature POINT = sig
   type 'a ext
   type 'a subself = 'a ext Root.subself
   type self = closed subself
   val unpack : 'a subself -> ('a -> 'a subself) * 'a
   val pack : int -> 'a -> 'a subself
   val pos : 'a subself -> int
   val slide : 'a subself -> int -> 'a subself
   val eq : 'a subself -> 'a subself -> bool
   val dist : 'a subself -> 'a subself -> real
   val new : int -> self
end;

structure Point :> POINT = struct
(* types *)
   datatype 'a state = S of
            ('a subself -> int) *
```

```
               ('a subself -> int -> 'a subself) *
               ('a subself -> 'a subself -> bool) *
               ('a subself -> 'a subself -> real)
   withtype 'a ext = 'a state * 'a
   and 'a subself = 'a ext Root.subself;
   type self = closed subself;
(* messages *)
   fun pos p     = let val (_,(S (n,_,_,_),_)) = Root.unpack p in n p end;
   fun slide p n = let val (_,(S (_,s,_,_),_)) = Root.unpack p in s p n end;
   fun eq p q    = let val (_,(S (_,_,e,_),_)) = Root.unpack p in e p q end;
   fun dist p q  = let val (_,(S (_,_,_,d),_)) = Root.unpack p in dp q end;
(* packing *)
   fun unpack p = explode (Root.unpack p);
   fun repack f p = implode (fn S m => S (f m)) (Root.unpack p);
   fun pack n ext =
      let fun methods n =
             let fun mpos p = n
                 fun mslide p n' = repack (fn _ => methods (n + n')) p
                 fun meq p q = pos p = pos q
                 fun mdist p q = real (pos q - pos p)
             in (mpos,mslide,meq,mdist)
             end
      in Root.pack () (S (methods n),ext)
      end;
(* new *)
   fun new n = pack n %;
end;
```

In the *types* section, the state of a point object is now constituted of four pre-methods, corresponding to messages `pos`, `slide`, `eq` and `dist`. The `state` type operator is recursive since pre-methods take the self object as first argument. Thanks to the explicit extension component of objects, we do not need plain recursive types to define type `subself`, but simply ML's recursive type operators. Note also that the type of self in pre-methods is polymorphic in its extension (`'a subself`), so that on one hand methods will be applicable to any descendant of points, rather than just "closed" points, and on another hand the type of self will propagate unchanged when used as result type of methods.

The *messages* section declares the available messages for point objects. Each message takes self as first argument, possibly followed by extra parameters. Messages first fetch the relevant pre-method from the self parameter, then apply that pre-method to self (which produces a method), and finally apply the extra parameters to the resulting method. Not the use of the `unpack` routine of the father class to gain access to the state of objects relevant in the current class.

The *packing* section provides `unpack`, `repack` and `pack` routines for points. These serve exactly the same purposes as in Section 2.2.

The *new* section defines a generator for the class. A point is created from scratch from its position value.

The `POINT` signature is the same as that in Appendix A2.2.1 except for the types of messages `eq` and `dist`: conversely to the nonembedded specification of A2.2.1 the binary methods are here homogeneous, that is they require both their parameters to be of the same type. Of course, we could still define messages `eq` and `dist` as “external” functions, as before, rather than as embedded methods. The advantage of embedding binary methods will appear when specialization comes into play. Their restricted type can be seen as the price to pay for simple dynamic dispatch.

4.2 Method update and simple dynamic dispatch

We have seen in Section 3.4 that dynamic dispatch requires some indirection in method lookup. There, this indirection was implemented in the “external” environment, through method dictionaries kept in references. In this section, we take advantage of the indirection provided by embedding the methods: by packing different pre-methods in objects of different classes, messages will fetch different methods, according to the class of self. In terms of types, a specialized version must have the same type as the non specialized one, but in which the type of self is instantiated to the current self type (that is `'a subself`, for the class being defined).

In class definitions, packing different versions of some pre-methods according to the type of self will be implemented by method update. To create, say, a cpoint with a specialized eq method taking advantage of colors, we first build a cpoint with the eq pre-method of a point, then replace that eq pre-method by one suitable for cpoints. We use the `Point.repack` routine for this, so method update builds a new object rather than modifying an existing one. Thus, specialization first requires the `repack` routines to be visible to descendant classes.

Now, if `Point` method suites are defined as in the previous specification (Page 32), that is capturing the position of points in their environment, then methods returning objects (such as `Point.slide`) cause a specific problem: the effect of `Point.slide` in Page 32 is to replace the whole method suite with new versions, capturing in their environment the new value of the point position. This is fine if the actual class of the point moved is `Point`, but if that actual class is some descendant class of `Point` that specializes `slide` or any other point method, then that specialization is lost in the process!

This particular problem requires a slightly different representation of state information to allow method update without loosing specializations. The state information of objects need be split into two components:

- The part of the state that may be changed on method invocation (e.g. the position information of points). That information should not be captured by pre-methods;
- The part which is invariant by method invocation (typically the pre-methods themselves, plus possibly some extra invariant information).

Our updated specification now looks as follows. Class `Root` is unchanged. Signature `POINT` lets function `repack` visible to its subclasses. Compared to Section 4.1, point states are now pairs, with the point position as first component and the pre-method suite as the second component. The messages and **`repack`** functions are updated accordingly (`repack` only updates the pre-methods component). The pre-methods to be packed within objects are defined so that they do not capture in their closure the position information. To implement the `slide` method, an extra `repacks` utility is defined, that returns a point with an updated position information (but does not affect pre-methods). Method `pos` extracts from self the position information.

```
structure Point :> POINT = struct
(* types *)
   datatype 'a state = S of
          int *                                        (* position *)
          (('a subself -> int) *                       (* pos    *)
```

```
            ('a subself -> int -> 'a subself) *          (* slide *)
            ('a subself -> 'a subself -> real) *         (* dist  *)
            ('a subself -> 'a subself -> bool)           (* eq    *)
   withtype 'a ext = 'a state * 'a
   and 'a subself = 'a ext Root.subself;
   type self = closed subself;
(* messages *)
   fun pos p    = let val (_,(S (_,(n,_,_)),_)) = Root.unpack p in n pend;
   fun slide p n = let val (_,(S (_,(_,s,_)),_)) = Root.unpack p in s p n
end;
   fun dist p q  = let val (_,(S (_,(_,_,d)),_)) = Root.unpack p in d p q end;
(* packing *)
   fun unpack p    = explode (Eqobj.unpack p);
   fun repack f p   = implode (fn S (s,m) => S (s,f m)) (Root.unpack p);
   fun repacks f p  = implode (fn S (s,m) => S (f s, m)) (Root.unpack p);
   fun mpos p      = let val (_,(S (s,_),_)) = Root.unpack p in s end;
   fun mslide p n' = repacks (fn n => n + n') p;
   fun mdist p q   = real (pos q - pos p);
   fun meq p q     = pos p = pos q;
   fun pack n ext = Root.pack () (S (n,(mpos,mslide,mdist,meq)),ext);
(* new *)
   fun new n = pack n %;
end;
```

As for class Point, Cpoint.repack must be visible to the subclasses of Cpoint. Class Cpoint specializes method eq. Note how Cpoint equality is computed from Point equality (and not from scratch) and color comparison. It is important in general to be able to gain access and use previous versions of methods since it is not generally possible to re-implement them in descendant classes.

```
structure Cpoint :> CPOINT = struct
(* types *)
   datatype 'a state = S of
          color *
          (('a subself -> color) *                    (* col *)
           ('a subself -> color -> 'a subself))       (* paint *)
   ...
(* packing *)
   ...
   fun mcol p = let val (_,(S (s,_),_)) = Point.unpack p in s end;
   fun mpaint p c' = repacks (fn c => c') p;
   fun cpeq peq p q = peq p q andalso col p = col q;
   fun pack (c,n) ext =
          Point.repack (fn (po,sl,di,peq) => (po,sl,di,cpeq peq))
                 (Point.pack n (S (c,(mcol,mpaint)),ext));
(* new *)
   fun new (n,c) = pack (c,n) %;
end;
```

The code for our sample hierarchy is found in Appendix A2.3.2, complete with specialized equality and distance methods, and line objects. As for the specification in Appendix A2.2.2, the hierarchy includes a separate `Eqobj` class, ancestor of all classes that make use of equality (this is necessary as we want to specialize eq for lines too). Class `Line` can be defined with a polymorphic generator. Thanks to dynamic dispatch, its methods can be correctly defined without making precise the exact class of line ends.

So, by embedding methods, we have been able to implement rather easily simple dynamic dispatch, that is selection of a method according to the type of self. Messages act as "selectors", that fetch a pre-method into their self parameter. Messages and pre-methods are polymorphic in object extensions. Applied to self, messages return methods, which have at that point the type of their associated pre-methods, with the type of self instantiated to its "actual" type. This explains that message `eq` (say) is typable in ML, while the overloading of methods `eq p` and `eq cp`, in which `p` is point and `cp` is a cpoint, would not be. It is interesting to note that we have obtained simple dispatch without dynamic typing, encapsulation of types within values, or subtyping. It is the parametricity of pre-methods and the indirection provided by embedding methods that do the trick.

Compared to some approaches based on subtyping, we avoided the problem of loss of type information for methods returning self, and that of specialization of objects with contravariant interfaces: both covariant and contravariant specialization work here. However, as already noted, and compared to the "external methods" version of Section 3.4, the binary methods (n-ary methods, in general) have a more restricted type: all of their object parameters must have the same type as that of self. Consequently, one cannot compare for equality e.g. a point with a cpoint, or a cpoint with an point, though one may compare two points or two cpoints. We implemented so far "simple" dynamic dispatch, or dispatch on self's type, as opposed to multiple-dispatch (on self's type and other object parameters of methods as well). Solutions to recover this lost polymorphism and implement multiple dispatch are discussed in the next section.

4.3 Multiple dispatch

4.3.1 Achieving multiple dispatch

We are interested in solutions to provide multiple dispatch that preserves all of the benefits of the OOP approach, that is in solutions notably preserving incremental class development. Two mostly separated issues are involved:

(1) A typing issue: how to restore the lost polymorphism of binary methods (compared to Section 2). We must find some way of disconnecting the extension parameters of the possibly many non-self object parameters of methods with that of self and between themselves.

(2) An operational issue: How to implement dispatch on other objects than self.

A third issue to consider is interaction of the solution retained with the type system of ML.

An obvious technique to solve (1), for instance, is to add parameters to object types that stand for the types of objects other than self involved in binary methods. That solution is not satisfying for a number of reasons including its bad interaction with ML typing (e.g. the function **`fun`** eq2 x y = eq x y **`andalso`** eq y x would be ill-typed). In any case, it would solve problem (1), but does not help solving (2).

Ingalls proposed in [9] a technique to simulate multiple dispatch with simple dispatch. The technique, relying on a double indirection in method lookup, would solve (2), but not (1). Furthermore, it can't be made incremental (or with assignable method dictionaries in the vein of Section 3).

Other possible techniques for handling binary methods are discussed in [3]. The pragmatic solution developed in the next section relies on the wrapping techniques explained in Section 3.3.

4.3.2 Multiple dispatch by wrapping

Let us consider first the version of wrapping referred to as folding introduced in Section 3.3.1. Let us recall that this wrapping technique is reversible (objects can be unwrapped with their original type), but that it cannot be used to wrap parametric objects.

The self argument of pre-methods is handled as in Section 4.2 (unwrapped), but other object arguments of messages are folded to type `wrap` (bound to `Root.self` here) before invoking the embedded methods. Binary pre-method perform pattern matching on their wrapped argument(s) to simulate a type-case.

Other differences with the folding technique used in Section 3.4 to simulate overloading include:

- First, the state of objects typically involve self's type here, conversely to Section 3.4. Folding an object into an object of its father class, and back, is still possible, but it requires in addition conversion of parameters and results of the embedded methods. That task is devoted to a new `coerce` function added to class interfaces. It is interesting to note too that, though coerced to an arbitrary object type, objects retain their original pre-methods.
- Instead of considering objects as triples: (contents, wrap function, unwrap function) as in Section 3.4.1, functions `wrap` is handled as a method and embedded within objects, and unwrapping is not necessary. Having `wrap` as a method slightly complicates object packing, but is more consistent with the "embedded-methods" idea. For uniformity of interfaces, method wrap is provided by the Root class (an alternative is to introduce a "`Wrappable`" class inheriting class `Root` and ancestor of all classes using wrapping).

Class `Root` is defined as follows. Compared to the version in Section 4.2, it uses the extensible version of datatype `closed` defined in Section 3.4.1, it introduces method `wrap`, wrapping utilities `wrap'self`, `self'wrap` (which are identity functions since the wrapping type is `root`), and `coerce`. These utilities must be visible to descendant classes. Function `pack` takes an extra parameter which is the folding function to be embedded within the object built.

```
signature ROOT = sig
   type wrap
   exception Wrap
   type 'a subself
   type self = closed subself
   type 'a ext = 'a subself
   val pack   : unit -> ('a subself -> wrap) -> 'a -> 'a subself
   val unpack : 'a subself -> ('a -> 'a subself) * 'a
   val coerce : ('a subself -> 'b subself) ->
                ('b subself -> 'a subself) ->
                'a subself -> 'b -> 'b subself
   val wrap'self : self -> wrap
   val self'wrap : wrap -> self
   val wrap : 'a subself -> wrap
   val new : unit -> self
end;

structure Root :> ROOT = struct
(* types *)
   datatype 'a state = S of 'a subself -> wrap
```

```
   withtype 'a subself = 'a state * 'a
   and self = closed subself
   and wrap = self;
(* wrapping *)
   exception Wrap;
   fun wrap'self p = p;
   fun self'wrap p = p;
   fun coerce c c' (S w,_) = (S (w o c'),e');
(* messages *)
   fun wrap (p as (S w,_)) = w p;
(* packing *)
   fun unpack wob = explode (I,wob);
   fun pack () wrap'curr ext = (S wrap'curr, ext);
(* new *)
   fun new () = pack () wrap'self %;
end;
```

Class Point is shown below, only the code for folding/unfolding and for implementation of method `eq` is shown. Signature `POINT` is similar to the previous, except for the new `coerce`, `wrap'self`, and `self'wrap` functions. Also, binary messages have a “fully polymorphic” type, rather than an homogeneous one. In point states, the second argument of binary pre-method have type `wrap`. We could have specified it as `Point.self` instead, but specifying it as `wrap` eases unwrapping; the constraint that the second argument of message `eq` is actually a subpoint is conveniently enforced by the type assigned to message `eq` in the signature:

```
signature POINT = sig
   ...
   val eq : 'a subself -> 'b subself -> bool
end;

structure Point :> POINT = struct
(* types *)
   datatype 'a state = S of
         ... *
         ('a subself -> Root.wrap -> bool)           (* eq    *)
   ...
(* wrapping *)
   val (Self,matches) = create (None : closed ext option);
   fun super'self p =
      let val (_,ext) = Root.unpack p
      in Root.coerce super'self self'super p (Self ext)
      end
   and self'super (w : Eqobj.self) : self =
      let val (_,ww) = Eqobj.unpack w
      in case matches ww of
        Some ext => Eqobj.coerce self'super super'self w ext
      | _ => raise Root.Wrap
      end;
```

```
    val wrap'self = Root.wrap'self o super'self;
    val self'wrap = self'super o Root.self'wrap;
    fun coerce c c' p e' =
        let val (_,(S (x,(n,s,d,q)),e)) = Root.unpack p
        in Root.coerce c c' p (S (x,(n o c',fn n => c o (s o c') n,d o c',q o c')),e')
        end;
(* messages *)
    ...
    fun eq p q = let val (_,(S (_,_,_,e),_)) = Root.unpack p in e p (Root.wrap q) end;
(* packing *)
    ...
    fun repacks f p = implode (fn S (s,m) => S (f s,m)) (Root.unpack p);
    fun mpos p = let val (_,(S (s,_),_)) = Root.unpack p in s end;
    fun mslide p n' = repacks (fn n => n + n') p;
    fun mdist p q = real (pos (self'wrap q) - pos p);
    fun meq p q = pos p = pos (self'wrap q);
    fun pack n wrap'curr ext =
         Root.pack () wrap'curr (S (n,(mpos, mslide, mdist,meq)),ext);
(* new *)
    fun new n = pack n wrap'self %;
end;
```

Class Cpoint is defined along the same lines. The interesting part is the definition of cpoint equality within function pack and the update of point equality by cpoint equality. Unwrapping may fail in `cpeq`: it does so when the third `cpeq` argument is not a cpoint of a descendant of a cpoint (e.g. if it is simply a point). In that case, the `Wrap` exception raised by `self'wrap` (when trying to coerce the alternate point into a cpoint) is trapped and color comparison is abandoned.

```
structure Cpoint :> CPOINT = struct
    ...
(* packing *)
    ...
    fun cpeq peq p q =
        (col p = col (self'wrap q) andalso peq p q) handle Root.Wrap => peq p q;
    fun pack (c,n) wrap'curr ext =
         Point.repack (fn (po,sl,di,peq) => (po,sl,di,cpeq peq))
                (Point.pack n wrap'curr (S (c,(mcol,mpaint)),ext))
(* new *)
    fun new (n,c) = pack (c,n) wrap'self Root.%;
end;
```

The full code for our sample hierarchy is given in Appendix A2.3.3. Compared to the specification in Appendix A2.3.2, we have recovered the most general types expected for binary messages, at the price of maintaining folding/unfolding utilities in each class, and of some loss of polymorphism for class `Line` (since wrappers cannot be polymorphic here). The size overhead of specifications is significant, but the folding utilities are highly regular. The essential difficulties we had, as for the specification in Section 3.4, was for folding line objects, as lines are polymorphic in their point extremities, and so the exact wrap and unwrap functions to use for lines cannot be known until some line is actually built. For this reason, class `Line` in Appendix A2.3.3 is defined as a functor producing monomorphic lines (generator of the resulting line class is monomorphic).

An alternate version using wrapping with dynamics is shown in Appendix A2.3.4. It results in a significantly more compact specification as wrapping with dynamics does not require most of the above wrapping apparatus. More importantly, since dynamics may wrap values with partially specified types, wrapping with dynamics allows generator of class `Line` to be polymorphic, as it should be. Obviously, an extension of ML is required, but this specification shows that the minor extension required is worth including if multiple dispatch is desired.

4.4 Summary

4.4.1 About the “embedded-methods” encoding

At this point, we have been able to encode satisfactorily the essentials of OOP into ML, without any extension. The most preeminent missing feature is certainly multiple inheritance, though many established OO notations do not support it.

With multiple dispatch omitted, the fragment encoded is quite close to that provided by OCaml, for instance, except for multiple inheritance and subtype-based coercions, but clearly in much simpler typing and operational frameworks. With wrapping and multiple dispatch, we come close to the object modeling capabilities of production OO languages, with all the benefits of ML in addition. Technically, it is deceptive that multiple dispatch can’t be encoded more simply but we believe the “embedded-methods” encodings are worth even without multiple dispatch.

4.4.2 To embed or not to embed?

We now have three views of objects to play with:

- an “external-methods” view without dynamic dispatch (Section 2);
- an “external-methods” view with overloading (Section 3);
- an “embedded-methods” view, with (possibly multiple) dynamic dispatch (Section 4).

How to choose one?

As long as dynamic dispatch is not needed, it makes few sense to embed methods: the specifications are more complex and embedded binary methods suffer some loss of polymorphism. The encoding of Section 2 is thus a good choice in that case; it supports inheritance and method specialization, but not dynamic dispatch.

If dynamic dispatch is needed, but one can live with simple dispatch, then the “embedded-methods” encoding of Section 4.2 seems the best, in spite of some loss of polymorphism for binary methods. It is simple enough to be used without support from the programming environment, and offers most of the features of OOP.

If multiple dispatch is occasionally required, then the simplest solution is certainly to use again the “embedded-methods” encoding, together with dynamics-based wrapping for simulating multiple dispatch. Adding the required form of dynamics to ML can be considered a simple task, and it is useful for many other purposes than OO programming.

If multiple dispatch was to be used heavily, then some support from the programming environment must definitely be added to ML. With the non-embedded model of objects, it could conveniently take the form of some built-in support for dynamically resolved overloading (see e.g. [2]); methods would just be overloaded functions then. For the embedded-methods models, whether a better support than that provided by dynamics can be found is unclear yet (to the author).

Mixed-models encodings may also be of interest. Consider for instance the methods that are only used to access object attributes (`pos`, or `col`, etc.). In many languages, object attributes would not be implemented as methods, but rather as "instance variables". In our framework, these attributes could be simply part of the state of objects, without having an embedded method associated with them.

Alternative "embedded" encodings, including analogues in our framework of the "imperative", the "recursive record", and the "state-application" approaches do not offer significant advantages over the "self-application" approach developed. The first variant is limited in expressiveness, and the latter two are hard to justify in absence of subtyping, as already mentioned.

4.4.3 What do we miss?

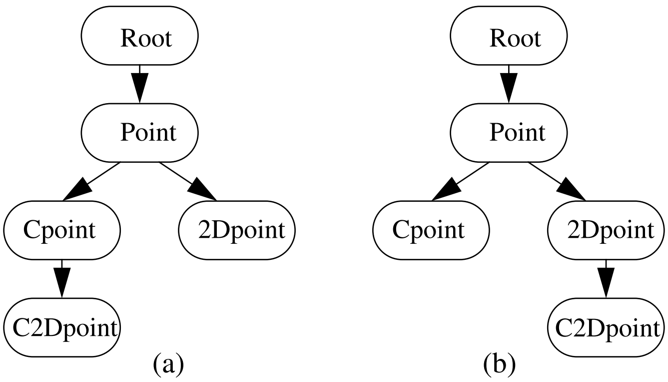
A number of issues remain to be explored. Multiple inheritance and subtyping are the most important, they are discussed in the next two sections, respectively. A few others, less important, issues are briefly discussed in Section 7.

5 Multiple Inheritance

5.1 Multiple Inheritance

The models of objects and classes developed in Sections 2 to 4 do not support multiple inheritance. Our encoding of inheritance forces the class hierarchy to form a tree, while notations supporting multiple inheritance allow this hierarchy to form a dag.

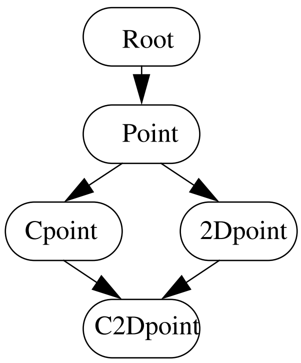
Consider for instance a hierarchy including points, 2D-points, colored points, and colored 2D-points. Using single inheritance, possible inheritance trees are:



(a) (b)

In (a), classes C2Dpoint and 2Dpoint do not share the 2D methods, while in (b), Cpoint and C2Dpoint do not share the color methods.

Using multiple inheritance, we could use the following inheritance dag, in which class C2Dpoint inherits of both Cpoint (for color methods) and 2Dpoint (for 2D methods):



The benefits of multiple inheritance are also its pitfalls, unfortunately. In the above dag, class C2Dpoint inherits the methods of class Point twice, through two different paths! Languages supporting multiple inheritance typically handle this by privileging one of these paths (with some help from the programmer), so that changes to a point object are unambiguously interpreted in descendant classes. Such state sharing problems, along with severe technical difficulties concerning typing and implementation, motivate rejection of multiple inheritance in many designs.

Are there means to allow optimal sharing of method suites while avoiding the difficulties of multiple inheritance ? What we want to say here indeed, is that class C2Dpoint has the point methods, the color methods, and the 2D methods, without having to duplicate either series of methods in the hierarchy. If we allow type safe instantiation of the code of methods (rather than duplication, as macros would do), then there is in many cases an alternative in ML and languages supporting modules: Since classes are structures, method suites could be provided as functors[1] parameterized by their "super" class. Building a class with some method suites would just amount then to a series of functor applications, each of them introducing a particular method suite.

The technique clearly leads to an OO development method different from the class-based method discussed so far, however. To allow incremental development, the basic building blocks of specifications should be the functors generating method suites rather than the classes one can build from them; and classes should be systematically developed from these functors. In spite of its specific requirements, we believe the technique has some merits. It is not a substitute form multiple inheritance in the general case, but rather a handy alternative applicable in some cases. The idea is developed in the sequel.

5.2 Pre-classes as Functors

How to define classes from functors that just specify the methods and attributes added when the functor is applied to some class? To be of interest, these functors should be generic enough in their requirements from their superclass parameter so that they are usable to define more than a single class! Not all declarations in the classes discussed in the previous sections satisfy to this genericity requirement. In particular the `pack` and `new` functions do not, since some of their parameters are not specific to the layer being added. However, with a minor restructuring of `pack` functions, and elimination of `new` from functors, a mixed functors/structures solution can be reached in which the generic part of classes of embodied in functors (that can be seen as Pre-classes, or Meta-classes), and classes reduce to structures built by functor application and generator specification.

To illustrate the approach, let us assume we obtained an `Eqobj` class, descendant of Root, and providing an `eq` method. We are interested in building a hierarchy of classes rooted at `Eqobj` by adding to an existing class in the hierarchy either a dimension, or a color, in all cases with specialization of the `eq` method.

With the approaches developed in Sections 2 to 4, we would fully define a structure for each class in the hierarchy, replicating e.g. the definition of `col` and `paint` methods each time we add a color attribute. Using functorial packaging, we will still have to define a structure for each class, but with most of their contents obtained by application of one of the two different functors:

- A functor `Place` that adds a dimension to a class; specializing `eq`;
- A functor `Paint` that adds a color to a class; specializing `eq` too.

Functorial packaging is applicable with few differences to a variety of encodings, including the "non-embedded" encoding of Section 2 and the embedded self-application one of Section 4. We illustrate the idea on the later, in the simpler version with simple dynamic dispatch introduced in Section 4.2.

Functors `Place` and `Paint` need some assumption on their `Super` class parameter. It is assumed that this `Super` class is a descendant of class `Eqobj`, and so that it allows one updating method `eq`. These assumptions are summarized by the following signature for parameter `Super`:

```
signature EQOBJECT = sig
```

1. Structures parameterized by other structures.

```
   type 'a subself
   type self = closed subself
   val unpack : 'a subself -> ('a -> 'a subself) * 'a
   val repack_eq : (('a subself -> 'a subself -> bool) ->
                    ('a subself -> 'a subself -> bool)) ->
                   'a subself -> 'a subself
end;
```

Functor `Place` adds `pos` and `slide` methods and specializes `eq`. The `slide` method applies to the particular dimension added (it is not itself specialized when adding a dimension). The class it defines inherits from class `Super` (the functor's parameter) assumed to obey signature `EQOBJECT. The` result class is ascribed the following `PLACED` signature. The type of `pack` in this signature will be justified soon.

```
signature PLACED = sig
   structure Super : EQOBJECT
   type 'a ext
   type 'a subself = 'a ext Super.subself
   type self = Root.closed subself
   val pack : int -> ('a subself -> 'a subself) * ('a -> 'a ext)
   val unpack : 'a subself -> ('a -> 'a subself) * 'a
   val repack : (('a subself -> int) * ('a subself -> int -> 'a subself) ->
                 ('a subself -> int) * ('a subself -> int -> 'a subself)) ->
                'a subself -> 'a subself
   val pos : 'a subself -> int
   val slide : 'a subself -> int -> 'a subself
end;
```

Compared to the contents of the `Point` structure in Section 4.2, and beside omission of the `dist` method, the body of functor `Place` differs by not providing generator `new` and by a specific version of the `pack` utility. As function `pack` cannot be made generic enough to appear in `Super`'s signatures, functions for packing objects are not part of pre-classes definitions. Instead, functors just include a partial packing function (also called `pack` though) that returns two functions: the method specialization to be applied to the resulting object, and a function expecting an object extension and producing an object extension including position methods. These two functions will be combined later with those of the `Super` class to produce a working packing function for objects. Functor `Place` is defined as follows.

```
functor Place (structure S : EQOBJECT) :> PLACED where Super = S = struct
   structure Super = S;
(* types *)
   datatype 'a state = S of
            int *
            (('a subself -> int) *
             ('a subself -> 'a subself -> real)
   withtype 'a ext = 'a state * 'a
   and 'a subself  = 'a ext Super.subself;
   type self = closed subself;
(* messages *)
```

```
      fun pos p    = let val (_,(S (_,(n,_)),_)) = Super.unpack p in n p end;
      fun slide p n = let val (_,(S (_,(_,s)),_)) = Super.unpack p in s p n end;
   (* packing *)
      fun unpack p = explode (Super.unpack p);
      fun repack f p = implode (fn S (s,m) => S (s,f m)) (Super.unpack p);
      fun repacks f p = implode (fn S (s,m) => S (f s, m)) (Super.unpack p);
      fun mpos p = let val (_,(S (s,_),_)) = Super.unpack p in s end;
      fun mslide p n' = repacks (fn n => n + n') p;
      fun peq _ p q = pos p = pos q;
      fun pack n = (Super.repack_eq peq, fn ext => (S (n,(mpos,mslide)),ext));
   end;
```

Similarly, functor `Paint` adds `col` and `paint` methods to its parameter class, and specializes `eq`:

```
   signature PAINTED = sig
      structure Super : EQOBJECT
      type 'a ext
      type 'a subself = 'a ext Super.subself
      type self = closed subself
      val pack : color -> ('a subself -> 'a subself) * ('a -> 'a ext)
      val unpack : 'a subself -> ('a -> 'a subself) * 'a
      val repack : (('a subself -> color) * ('a subself -> color -> 'a subself) ->
                    ('a subself -> color) * ('a subself -> color -> 'a subself))
->
                   'a subself -> 'a subself
      val col : 'a subself -> color
      val paint : 'a subself -> color -> 'a subself
   end;

   structure Paint (structure S : EQOBJECT) :> PAINTED where Super = S = struct
      structure Super = S;
   (* types *)
      datatype 'a state = S of
               color *
               (('a subself -> color) *
                ('a subself -> color -> 'a subself))
      withtype 'a ext = 'a state * 'a
      and 'a subself = 'a ext Super.subself;
      type self = closed subself;
   (* messages *)
      fun col p     = let val (_,(S (_,(c,_)),_)) = Super.unpack p in c p end;
      fun paint p c = let val (_,(S (_,(_,pa)),_)) = Super.unpack p in pa p c
end;
   (* packing *)
      fun unpack p = explode (Super.unpack p);
      fun repack f p = implode (fn S (s,m) => S (s,f m)) (Super.unpack p);
      fun repacks f p = implode (fn S (s,m) => S (f s,m)) (Super.unpack p);
      fun mcol p = let val (_,(S (s,_),_)) = Super.unpack p in s end;
```

```
    fun mpaint p c' = repacks (fn c => c') p;
    fun cpeq peq p q = peq p q andalso col p = col q;
    fun pack c = (Super.repack_eq cpeq, fn ext => (S (c,(mcol,mpaint)),ext));
end;
```

Let us build now a sample hierarchy from class `Eqobj` and these functors. Each class invokes a functor, and adds to the structure obtained a generator `new`, a `pack` function for objects of the class and utilities for specialization of methods (the `repack_*` functions):

Assuming the following declarations (`oo` is used in infix notation):

```
fun (f,g) oo (F,G) = (f o F, G o g);
fun make (f,g) = f (g %);
```

Then class `Point` is created from `Eqobj` by adding a dimension using functor `Place`, as follows. Structure `Point` also defines `pack` and `new` functions for point objects, `Point.pack` combines the pack function for `Eqobj` and that obtained after functor application.

```
structure Point = struct
    local
    structure P = Place (structure S = struct
                                           open Eqobj;
                                           val repack_eq = Eqobj.repack;
                                       end);
    in
    open P;
    fun pack n = P.pack n oo Eqobj.pack ();
    fun new n : self = make (pack n);
    end;
end;
```

Class `D2point` is created from `Point` by another application of functor `Place`:

```
structure D2point = struct
    local
    structure P = Place (structure S = struct
                                           open Point;
                                           val repack_eq = Eqobj.repack;
                                       end);
    in
    open P;
    fun pack (x,y) = P.pack y oo Point.pack x;
    fun new (x,y) : self = make (pack (x,y));
    end;
 end;
```

Both classes `Point` and `D2point` provide `pos` and `slide` methods, but these are different messages, that operate on different dimensions. One may wish to add a few aliases like:

```
val xpos = Point.pos and xslide = Point.slide;
val ypos = D2point.pos and yslide = D2point.slide;
```

`Point` or `D2point` objects may be colored by application of functor `Paint`, e.g.

```
structure Cd2point = struct
   local
   structure P = Paint (structure S = struct
                                          open D2point;
                                          val repack_eq = Eqobj.repack;
                                      end);
   in
   open P;
   fun pack (x,y,c) = P.pack c oo D2point.pack (x,y);
   fun new (x,y,c) : self = make (pack (x,y,c));
   end;
end;
```

So, though classes still need be built, the code to built them is much smaller than that shown in Section 4.2. More importantly, no part of the code need be replicated and functors provide some amount of implementation inheritance. Using functors allows more code reuse than allowed by simple inheritance, but the class hierarchy still forms a tree, rather than a dag. So, a `Cpoint` class built by an application of `Paint` from class `Point` would not be an ancestor class of `Cd2point` (built as above).

Our running example hierarchy is provided with functorial packaging in Appendices A2.2.4 and A2.3.5, for two different models of objects.

5.3 Summary

Expressing class generators, or pre-classes, as functors parameterized by a superclass, then deriving classes from these functors allows restoring most of the sharing of code lost by not allowing multiple inheritance. The code produced by functors need be instantiated though, but this is done in a type safe manner and still allows some amount of implementation inheritance.

Whether this is a good enough substitute for multiple inheritance is left to the opinion of the reader. In general, multiple inheritance causes severe scoping problems for instance variables and methods, that typically prompt as many unfortunate adhoc restrictions. Associated with an adequate OO development method, functorial construction offer some important benefits of multiple inheritance while avoiding these pitfalls.

6 Subtyping

6.1 Subtyping and Typed OOP

Subtyping is an ordering on types, we write $\tau <: \tau'$ to mean that τ is a *subtype* of τ', or that τ' is a *supertype* of τ. That ordering can take many forms, depending on the particular set of types being ordered. The form we will use requires ML types, with constants `top` and `bottom` added. Omitting user-defined type operators, the subtyping relation is then the closure of relation <: by reflexivity, transitivity and the following axioms (for any types τ, τ_i, τ_i'):

$$\tau <: \texttt{top}$$

$$\texttt{bottom} <: \tau$$

$$\frac{\tau_1 <: \tau_1' \qquad \tau_2' <: \tau_2}{\tau_1' \to \tau_2' <: \tau_1 \to \tau_2}$$

$$\frac{\tau_1' <: \tau_1 \qquad \tau_2' <: \tau_2}{\tau_1' * \tau_2' <: \tau_1 * \tau_2}$$

Reading the last two rules backward, note that subtype constraints for products are propagated with unchanged directions to the components of products, the product type operator is said for this reason *covariant* in both its parameters. The function type operator, on another hand, is covariant on its second parameter (the result type) and *contravariant* in the first.

Subtyping systems have been introduced for many purposes. Some may not require top and bottom types, some add axioms concerning base types like `int` <: `real`, or `bool` <: `int`, others add axiom sets concerning records like e.g. `{pos:int, slide: int -> unit}` <: `{pos:int}`.

The interests of subtyping in general, and for OOP in particular, stems from the following rule, called *subsumption*. The rule says that, in any context, we may weaken the type of some expression to any of its supertypes, if it admits some:

$$\frac{\tau <: \tau' \qquad \texttt{e} : \tau}{\texttt{e} : \tau'}$$

Assuming that some function `f` has some type $\tau_1 \to \tau_2$, subsumption implies that `f` may safely accept as argument a value with any subtype of τ_1, and may return a value which has any supertype of τ_2. The relevance to OOP comes from the analogy between inheritance and subtyping. Assuming that subclassing implies subtyping of the corresponding object types, the subsumption rule says that objects in subclasses may safely use the methods they inherited from their superclasses. Subsumption has been traditionally seen as a way of interpreting inheritance at typing level.

Unfortunately, the situation is a bit more complex as subclassing does not generally imply subtyping [7]. Some OO notations enforce a "subtyping as subclassing" view by adhoc syntactic or typing restrictions, but this severely limits their expressiveness. Other notations just assume that view without enforcing it, trading type safety for expressiveness.

In our framework, we could successfully handle subclassing and inheritance without recourse to subtyping, so why should we bother to introduce it? Typical cases where subtyping allows typing more expressions that we currently do are the following, consider the declarations:

```
val p = Point.new 123;
val cp = Cpoint.new (123,red);
fun f (p : Point.self) = pos p + 10;

val x = f cp;
val l = [p,cp];
```

The last two are ill-typed in our framework (without subtyping), while, with subtyping, and if type `Cpoint.self` is a subtype of `Point.self`, they are well typed.

The benefits of subtyping are not restricted to OO features (e.g. it makes sense of the ML expression `[(2,2),(3,true)]`, for instance, that receives type `(int * top) list`). Concerning OO, does subtyping help to solve the difficulties we had with the standard ML type system for encoding some features? It seems it does not, while adding typing power as the above examples show. In particular, it does not help solving the multiple dispatch problem. Referring to our discussion in Section 4.3.1, it helps solving the typing problem (1), but, alone, does not help solving the operational problem (2). Furthermore, the concept of folding discussed in Section 3.3 provides rough solutions for typing the above expressions: the following alternative declarations of `x` and `l`, making use of folding, are well typed and behave as expected:

```
val point'cpoint = Point.self'wrap o Cpoint.wrap'self
val x = f (point'cpoint cp);
val l = [p,point'cpoint cp];
```

So, whether the gain in typing power is worth the complication of a plain implementation of subtyping in a polymorphic language like ML is open to debate. We carried out an experiment anyway, that we describe in the remaining of this section.

6.2 SML with subtyping, S*ub*ML

Which form of subtyping suits best our needs? Remember that we do not make use of records, and that we already solved the typing problem for inheritance by encoding it into parametricity. So, using an explicit subtyping relation on object type constructors as in e.g. Moby [8] makes little sense here, as it would be probably redundant. For this reason, we choose to experiment with “pure” subtyping, just adding to the set of types top and bottom elements. A recent account of the issues involved in implementing such a typing system, focusing on simplification of type constraints, appears in [15] [16].

To experiment with that subtyping system, we developed S*ub*ML. S*ub*ML implements Core SML, omitting references, imperative types, polymorphic equality and equality types. Exceptions are required monomorphic and structures are supported, but signatures and functors are currently not. The subtyping and constraint simplification systems are based on that of [15]. ML types are augmented with types `top` (noted `1`) and `bottom` (noted `0`). Type `1` is the type with no information, of which any type is a subtype; `0` is the type with all information, a subtype of any type. There are significant differences however between S*ub*ML and Pottier’s *Wallace* experimental implementation of its typing and simplification system:

- Pottier's type system assumes prior lambda lifting to remove free variables from let-bound expressions, while our does not. Our solution is currently very expensive as it duplicates constraint sets to handle let-polymorphism. Efficiency has just not been addressed yet. Nevertheless, at the occasional price of some code rewriting (manual lambda lifting), it is efficient enough to handle our object specifications.
- To the subtyping axioms for the arrow and product type operators (axioms that propagate subtyping constraints), S*ub*ML adds specific axioms for each user defined type operator. For each type operator declared by the user, the system computes a "variance" information that associates with each type parameter of the operator an indication of the way subtype constraints are to be propagated.

  For the following declaration of type operator T, for instance, the system infers that operator `T` is invariant in its first parameter, covariant in the second, contravariant in the third, and neutral in the fourth:

  ```
  datatype ('a,'b,'c,'d) T = F of 'c -> 'a | G of 'a -> 'b
  ```

- The type constraints simplifier is heavily based on Pottier's simplifier, but with adhoc enhancements mostly aimed at supporting the so-called "bipolar" variables, e.g. those found in invariant parameters of type operators.

The experimental S*ub*ML tool is available "as is" to those who would like to try subtyping in SML, check the author's web page for details[1].

6.3 Taking advantage of subtyping

Now that we have got an adequate version of ML with subtyping, the next step is to update our object encodings to take advantage of subtyping.

Let us first consider the "non-embedded" encodings of Section 2 and, to start with, the version without overloading discussed in Section 2. Take advantage of subtyping in that model is simply done by encoding the extension component of objects by an "undefined" value, of type `1` (*top*), instead of making use of a constant of type `closed`. It is perfectly safe to have S*ub*ML provide a constant of type `1` since no information can be gained from that constant. So, assuming the environment holds a constant `% : 1`, adapting the specification of Section 2 for subtyping is simply done by binding type `closed` to `1` rather than defining it as a datatype. Beside this, the specification is exactly as in Appendix A2.2.1.

As running the examples with S*ub*ML would confirm, subclasses yield subtypes in this model. Assuming `p` is a point object, `cp` is a colored point, and `dp` is a 2dpoint, the following declarations are now admitted:

```
let fun f (p:point) = 2 * pos p in f cp + f dp end;
val l = [p,cp,dp];
```

List `l` has a type homogeneous with `Point.self list` since, in this type system, `Point.self` is the least upper bound of types `Cpoint.self` and `D2point.self`.

The treatment cannot be extended to the specification of Section 3 (nonembedded model with overloading) and AppendicesA2.3.3 and A2.3.4, however. In these, the wrapping functions packed with objects prevent subclasses to yield subtypes.

1. http://www.laas.fr/~bernard

The “self-application” embedded-methods encodings of Section 4 fail to produce derived objects which have subtypes of their ancestors too. The reason there is that the type of self occurs in contravariant position in the types of the embedded pre-methods. This is indeed a well known pitfall of the self-application encoding. It is precisely for that reason that alternative models for objects with embedded methods have been addressed for a long time, several of them are compared in [1] [4].

An alternative embedded-methods model of objects known to fit better with subtyping is the so-called “recursive-record” model of [5][21]. It is adapted to our framework in the next section. Other alternatives that could be adapted include the “state-application” model of [14]; this otherwise elegant encoding offers poor support for methods returning self (like our `slide` method), which is why we do not develop it here.

6.4 The recursive-self encoding

We introduce in this section an analogue of the “recursive-record” model of [5] [21], cast into our framework, and without bothering about subtyping yet. Since our encoding does not make use of records, we will refer to it as the *recursive-self* model instead of using the original record-centric name.

The idea of the recursive-record model, starting from the self-application record model, is to abstract pre-methods from their self parameter in the definition of classes. For this, all methods at some level of specification capture self in their closures. In the record notation, this typically requires recursive types and some delayed evaluation strategy for self. These are not needed in our framework.

When adapting this model into our framework, the main difficulty is to define `self`, a function that returns the current object. All methods must have self in their closure, but the attributes (self function’s arguments) that will be added in derived classes are not known at the current level of specification. For this reasons, self is defined as a curried function, with one parameter added at each level (the innermost parameter is the last added). To allow sharing self between levels, the self function produced at some level (capturing the arguments required at that level and below) is passed as parameter to the `pack` function of level above. Function self is defined in generator `new` at each level, and passed to function `pack`. This is better explained on an example. Let us consider our hierarchy of points of Section 4.2.

Class `Root` is similar to that in Section 4.2, except for the type of function `pack`. For regularity, `Root.pack` is ascribed a similar type than the `pack` functions in derived classes, though it only makes use of its second argument.

```
signature ROOT = sig
    ...
    val pack : unit -> 'a -> (unit -> 'a subself) -> 'a subself
end;

structure Root :> ROOT = struct
    ...
(* packing *)
    ...
    fun pack () ob self = ob;
(* new *)
    fun new () = let fun self () = pack () % self in self () end;
end;
```

Class `Point` is defined as follows. The recursive-self encoding does not require to split object states into an invariant part and method suite, conversely to the self-application model. In structure `Point`, the types section is similar to that of Section 4.2, except for the missing point "position" and self arguments in the types of embedded methods. For simplicity, method `dist` is omitted. The message functions and the packing functions are updated accordingly. Note that functions `self` and `pack` are mutually recursive. The `slide` method does not use `repack` to update the object's methods, but builds instead a new object from scratch, using the `self` function. Function `repack` will still be needed to specialize methods.

```
signature POINT = sig
    ...
    val pack : int -> 'a -> (int -> unit -> 'a subself) -> 'a subself
end;

structure Point :> POINT = struct
(* types *)
   datatype 'a state = S of
                int *                             (* pos   *)
                (int -> 'a subself) *             (* slide *)
                ('a subself -> bool)              (* eq    *)
   withtype 'a ext = 'a state * 'a
   and 'a subself = 'a ext Root.subself;
   type self = closed subself;
(* messages *)
   fun pos p     = let val (_,(S (n,_,_),_)) = Root.unpack p in n end;
   fun slide p n = let val (_,(S (_,s,_)),_)) = Root.unpack p in s n end;
   fun eq p q    = let val (_,(S (_,_,e)),_)) = Root.unpack p in e q end;
(* packing *)
   fun unpack p = explode (Root.unpack p);
   fun repack f p = implode (fn S m => S (f m)) (Root.unpack p);
   fun pack n ext self =
      let fun mslide k = self (n + k) ()
           fun mpeq q = n = pos q
      in Root.pack () (S (n,mslide,mpeq), ext) (self n)
      end;
(* new *)
   fun new n = let fun self n () = pack n % self in self n () end;
end;
```

To illustrate method specialization, let us show how class `D2point` is built from class `Point`:

```
signature D2POINT = sig
    ...
    val pack : int * int -> 'a -> (int -> int -> unit -> 'a subself) -> 'a subself
end;

structure D2point :> D2POINT = struct
```

```
(* types *)
   datatype state = S of int
   withtype 'a ext = state * 'a
   and 'a subself = 'a ext Point.subself;
   type self = closed subself;
(* messages *)
   fun ypos p = let val (_,(S n,_)) = Point.unpack p in n end;
(* packing *)
   ...
   fun pack (y,x) ext self =
      let val mypos = y
           fun mslide _ k = self (y+k) (x+k) ()
           fun d2peq peq q = peq q andalso (y = ypos q)
      in Point.repack (fn (p,s,peq) => (p,mslide s,d2peq peq))
                 (Point.pack x (S mypos, ext) (self y))
      end;
(* new *)
   fun new (x,y) =
      let fun self y x () = pack (y,x) % self in self y x () end;
end;
```

Class `D2point` specializes methods `slide` and `eq`. The effects of `mslide` are not defined from those of `Point.slide` (as in Section 4.2), but from function `self;` `mslide` builds a new object from scratch. This seems unavoidable for methods that return self.

The full code for our sample hierarchy is shown in Appendix A2.4.1.

6.5 The recursive-self encoding with subtyping

Removing binary methods in the previous specification, then it would be sufficient to interpret type `closed` as type `1`, and constructor `%` as a constant of type `1` to take advantage of subtyping, as we did for the nonembedded model without overloading.

If binary methods are desired, then one must use in addition one of the wrapping techniques of Section 3.3. Tough all of the wrapping techniques explained there are applicable, and since we assume subtyping available, a sensible choice is that of the "subtype based" wrapping technique of Section 3.3.3. It provides the same benefits as existential wrapping techniques, but it relies on subtyping rather than existential types. The resulting code for our sample hierarchy (with binary methods and subtyping) is shown in Appendix A2.4.2; this code can be run with S*ub*ML[1].

Our attempts to rewrite the recursive-self specifications with the functorial packaging technique of Section 5 failed so far. We are not sure at the moment whether a suitable concept of pre-class can be built or not for the recursive-self model. The problem is essentially the non-regularity of the type of pack (resulting from the non-regularity of the type of function `self`).

---

1. Well, not yet at the time this report was written as S*ub*ML does not currently support signatures and functors. Abstype-based alternative specifications, runable in S*ub*ML, are provided in the author's web pages.

6.6 Summary

Added to our encodings of Sections 2, or in conjunction with the recursive-self encoding, subtyping allows typing more expressions, including some relevant to OOP. The implementation of subtyping is quite complex however; constraint simplification is expensive. So, whether typing these extra expressions are worth the complication of adding subtyping to ML needs more investigations.

Two alternatives to full subtyping come close in benefits without the full expense:

- The first is to provide subtype-based coercions, as Ocaml does. These coercions allows one weakening the type of some object, provided its static type is a subtype of the target type. This does not require full implementation of subtyping, but merely means of being able to check at compile time that some given pairs of types are in the subtyping relation.
- A second is to use folding as introduced in this report to achieve a similar effect. Folding an object into another is guaranteed not to cause dynamic failures if the source type is a descendant (rather than a subtype) of the target type, but it does generate some code, conversely to subtype-based coercions.

7 Miscellaneous Issues

We briefly investigate in this section a number of side issues not discussed in previous sections.

7.1 Other issues

7.1.1 Virtual classes, virtual methods, final classes

In OO development, it often happens that one has to provide a set of methods for objects in some hierarchy while this hierarchy has no obvious root class for which these methods have some meaning. E.g. method `eq` in our example hierarchy has totally disjoint implementations for lines and points.

Up to now, we handled that problem by providing a default eq method in class `Eqobj`; that default method raises exception `Eq` whatever its parameters. A more classical alternative, used in many OO notations, is to handle class `Eqobj` as a virtual class. Virtual classes have no generator (no `new`), they act as interface declarations for descending classes, but cannot produce objects. Any descendant class of `Eqobj` that should produce objects is constrained to define method `eq`; descendant class of `Eqobj` that do not define `eq` are considered virtual classes themselves.

In the embedded-method models, we could easily encode virtual classes as classes in which the missing methods are passed by the generator of the class. E.g. `Point.pack` would call `Eqobj.pack` with the point equality method as parameter.

Some OO languages also provide final classes. Final classes are classes that do not admit descendant classes. Final classes are easily interpreted in our framework as classes that do not provide an extension variable in their (object's) type.

7.1.2 Instance variables

Many OO notations distinguish methods from so-called instance variables. Instance variables typically hold objects attributes in imperative objects. They may be seen as methods that do not require passing self as first argument. They may be encoded using a blend of embedded (for methods) and non-embedded (for variables) encodings as we did for the "noninvariant" part of object states in the self-application model. If assignable, these variables should be defined as ML references.

7.1.3 Scope of methods, System and User interfaces

One may use ML local scoping facilities (**`local`**, **`let`**, signatures ascription) to control the scope of methods and other values. One can easily define this way methods local to some class (not visible to subclasses). As seen in Section 2.2, however, as long as classes are fully defined at user level one cannot hide to users the *system interface* of classes to users, while making it visible to subclasses (this system interface is a data-abstraction hole as it does allow building fake objects).

7.1.4 Object extensions, Object-based vs. Class-based notations

As was briefly mentioned in Section 2.2, it is possible with some of the models developed to build new objects from existing objects, rather than building objects from scratch. This is done by copying object attributes and replacing their extension components (constant `%`) by extra attributes and a new extension. The objects produced in the nonembedded models can readily be enriched this way (provided the type of `unpack` is slightly relaxed, as in Section 3.4.1). This is harder to achieve for objects with embedded methods, unless some form of folding of objects (cf. Section 3.3.1) is provided so that the existing object's methods can be coerced to methods operating on the enriched object.

7.1.5 Interface restrictions

Though not found in available OO programming languages, the capability of restricting the interface of an existing class to produce another class with, say, a subset of methods of the former, is considered an important feature by some OO language designers. To some extent, our encodings allow such "interface restriction" operations.

Assume for instance that one wants to produce a class sharing all messages of class `Point` (as found in Section 4.2, that is with self-application encoding and simple dynamic dispatch), but without its `slide` message. To build such a class, one would proceed as follows:

One first declares a signature for the restricted class (signatures specify class interfaces). The signature `RPOINT` below is obtained from signature `POINT` of Section 4.2 by removing its `slide` component:

```
signature RPOINT = sig
   type 'a ext
   type 'a subself = 'a ext Eqobj.subself
   type self = closed subself
   val unpack : 'a subself -> ('a -> 'a subself) * 'a
   val repack : (omitted)
   val pack : int -> 'a -> 'a subself
   val pos : 'a subself -> int
   val dist : 'a subself -> 'a subself -> real
   val new : int -> self
end;
```

Then one simply takes a copy of the `Point` class restricted with signature `RPOINT`:

```
structure RPoint = Point :> RPOINT;
```

Since the representation of type `ext` is hidden by the `POINT` and `RPOINT` signatures, the last declaration produces a class `RPoint` distinct from `Point`, that shares its implementation with it as well as its ancestor classes, but that does not provide a `slide` message. Note that the class introduces its own `pos` and `dist` messages, distinct from those of class `Point` in terms of typing.

Interface restriction may only concern here the messages introduced by the class to which the restriction is applied; it cannot concern inherited methods.

7.2 Implementations

What can be done with these object models?

7.2.1 Use them "as is" for OO programming in ML

This is fine for seldom use:

- Specifications are more verbose than, say, with Ocaml or Java, but highly regular, and size is reasonable if multiple dispatch is not required. Less verbose specifications may often be expressed with the (deprecated) **abstype** construction of ML together with type constraints, rather than with structures and signatures.

- Specifications are reasonably efficient if the hierarchy is not too deep (each level adds an extra layer of indirection). The version of `unpack` provided in the examples is expensive, but it could be replaced in most instances by a faster version not computing the object reconstruction function, as already said.

A good point is that object features used this way do not require any extension of ML, and that they interact well with other ML features.

7.2.2 Use them as implementation models, with some programming environment support

An alternative is to use these models as implementation models, the real textual specifications being written in a dedicated notation then translated into those implementations (or proven equivalent alternatives). The benefits would include:

- More compact class specifications;
- Hiding of system interfaces of classes, restoring data abstraction. These administrative interfaces should be entirely managed by the compiler.
- More efficient code generation, since we could separate typing issues from operational ones. E.g. classes could be typed as if they were translated as in our sample code, but implemented in a more efficient way.

Three important decisions would shape the concrete OO notation:

- Whether or not methods are embedded;
- Whether multiple dispatch is a feature or not;
- Whether subtyping, or a weak form of it, is a feature or not.

8 Conclusion

Technical conclusions were drawn at the end of each section. We developed in this report a number of object models and encoded them into ML. Most of the OO features and idioms could be encoded in “straightforward” ML, that is in much simpler typing and operational frameworks than previously proposed. The two essential differences between our models and most existing notations are that:

- Operationally, our models are not based upon records.
- They do not rely on subtyping and subsumption for interpreting inheritance polymorphism, but instead on an encoding of inheritance polymorphism into ML’s parametric polymorphism.

Not using records enables a simple encoding of inheritance, as shown, but fetching methods has an higher cost than in record-based encodings. On another hand, simple record-based encodings require expensive techniques to handle inheritance, like the wrappers of [19]. Implementations of objects relying upon extensible records, like Ocaml’s, allow both an efficient encoding of inheritance and fast method access, but they require considerably more complex typing and operational frameworks.

Our object models can readily be used for OO programming in ML, as shown in the Appendix, or as implementation models. Some features (e.g. multiple dispatch) would certainly benefit of some dedicated support, however, or at least of some minor extensions to ML (built-in extensible data types, existential dynamics).

The next step is to develop a notation for OO features, and a code generator, the models in this reports serving as “reference implementations”.

Acknowledgments

I wish to thank Didier Rémy for his comments and suggestions, François Pottier for a number of e-mail discussions about subtype constraints simplification, and Khalil Drira for many debates about OOP.

References

[1] Abadi M., Cardelli L., *A Theory of Objects*, Monographs in Computer Science, Springer Verlag, 1996.

[2] Bourdoncle F., Merz S., *Type Checking Higher-Order Polymorphic Multi-Methods*. 24th ACM Conf. on Principles Of Programming Languages, 1997.

[3] Bruce K. B., Cardelli L., Castagna G., The Hopkins Group, Leavens G. T. and Pierce B., *On binary methods*. Theory and Practice of Objects Systems 1(3); 217-238, 1995.

[4] Bruce K. B., Cardelli L., Pierce B., *Comparing Object Encodings*. Information and Computation.Vol. 155, No. 1/2, pp. 108-133, Dec 1999.

[5] Cardelli L., *A semantics for multiple inheritance*. In G. Khan, D. MacQueen, and G. Plotkin, editors, Semantics of Data Types, LNCS 173, Springer Verlag, 1988.

[6] Castagna G., *Covariance and Contravariance: Conflict without a cause*. ACM Transactions on Programming Languages and Systems 17(3); 431-447, 1995.

[7] Cook W. R., Hill W. L., Canning P. S., *Inheritance Is Not Subtyping*. 17th ACM Conf. on Principles Of Progrtamming Languages, 1990.

[8] Fisher K., Reppy J., *The design of a class mechanism for Moby*. ACM Conferenc on Programming Language Design and Implementation, 1999.

[9] Ingalls D. H., *A simple technique for handling multiple polymorphism*. OOPSLA'86 Conference Proceedings, ACM, 1986.

[10] Laufer K., Odersky M., *An Extension of ML with First-Class Abstract Types*. ACM Workshop on ML and its Applications, ACM, 1992.

[11] Leroy X., Mauny M., *Dynamics in ML*. Jounal of Functional Programming 3(4), 1993.

[12] Mauny M., Pottier F., *An implementation of Caml-Light with Existential Types*. Technical Report, INRIA, 1993.

[13] ML2000 Working Group, *Principles and a Preliminary design for ML2000*. Draft document, March 1999.

[14] Pierce B., Turner D., *Simple type-theoretic foundations for object-oriented programming*. Journal of Functional Programming 4(2); 207-247, 1994.

[15] Pottier F., *Simplifying subtyping constraints*. ACM International Conference on Functional Programming, ACM, 1996.

[16] Pottier F., *Type inference in presence of subtyping: from theory to practice*. Ph.D. Thesis. INRIA Research Report 3483, Sept. 1998.

[17] Rémy D., *Programming Objects with ML-ART - An extension to ML with Abstract and Record Types*, TACS 94.

[18] Rémy D., Vouillon J., *Objective ML: An effective object-oriented extension of ML*. Theory and Practice of Object Systems 4(1); 27-50, 1998.

[19] Thorup L., Tofte M., *Object-Oriented Programming and Standard ML*. ACM Workshop on ML and its Applications, ACM, 1994.

[20] Thorup L., *A comparative Analysis of Concepts in Object-oriented Programming*, M.Sc. thesis 93-7-11, Dept. of Computer Science, Univ. of Copenhagen, Denmark (in Danish).

[21] Uday S. Reddy, *Objects as Closures: Abstract semantics of object oriented languages*. 1988 ACM Symposium on Lisp and Functional Programming, ACM, 1988.

APPENDIX

Contents

A1 Example hierarchy

A1.1 Hierarchy

Our encodings are illustrated with a simple hierarchy of point and line classes. That hierarchy includes three classes of points, named `Point`, `Cpoint` and `D2point`, and a `Line` class; a line is an homogeneous pair of points of any class in the points sub-hierarchy.

- Objects of class `Point` have a *position* attribute (an integer), and methods `pos`, `slide`, `dist`, and `eq`. Method `pos` returns the position of an object, method `slide` shifts its position, the binary method `eq` implements position equality of two points, and `dist` returns the distance between two points.
- Class `Cpoint` (for colored points) is a subclass of `point`. Colored points have in addition a *color* attribute, with `col` (color) and `paint` methods, and specialize method `eq`. Method `paint` changes the color of a colored point to that passed as argument of the method. Method `eq` is specialized to sum up position and color equality.
- Class `D2point` (for two-dimensional points) is another subclass of `point` (but not a subclass of `cpoint`). D2points have an extra `ypos` method, returning the position of the point in the second dimension. Methods `eq`, `dist`, and `slide`, are specialized.
- Class `Line` is not a subclass of `Point`. It implements pairs of points (of any kind). It has a binary `eq` method that returns true if two lines have their point ends equal by `eq`. It has also a `length` method (the distance between its end point and starting point) and a `shift` method (that slides both ends of the line by some integer amount). The `eq`, `slide`, and `dist` methods on points used to implement Line methods should be the most specific versions of these methods applicable to the points constituting line ends.

A1.2 Ocaml version (Ocaml 2.0)

A1.2.1 Code

```
type color = Red | Green | Blue;;
let sq r = r *. r;;

exception Eq;;
class eqobj () =
object (self : 'a)
    method eq (p : 'a) = (raise Eq : bool)
end;;

class point n =
object (self : 'a)
    inherit eqobj () as super
    val x = n
    method pos = x
    method slide m = {<x= self#pos + m>}
    method eq (p : 'a) = self#pos = p#pos
    method dist (p : 'a) = float (p#pos - self#pos)
end;;

class cpoint (n,c) =
object (self : 'a)
    inherit point n as super
```

```
    val k = c
    method col = k
    method paint (c':color) = {<y= c'>}
    method eq (p : 'a) = super#eq p && (self#col = p#col)
end;;

class d2point (n,(m:int)) =
object (self : 'a)
    inherit point n as super
    val y = m
    method ypos = y
    method slide k = {<x= self#pos + k; y= self#ypos + k>}
    method eq (p : 'a) = super#eq p && (self#ypos = p#ypos)
    method dist (p : 'a) = sqrt (sq (super#dist p) +. sq (float (p#ypos - self#ypos)))
end;;

class ['b] line ((f:'b),(t:'b)) =
object (self : 'a)
    inherit eqobj () as super
    val ff = f
    val tt = t
    method lfrom = ff
    method lto = tt
    method shift n = {<ff= self#lfrom#slide n; tt= self#lto#slide n>}
    method length = ff#dist tt
    method eq (p : 'a) = self#lfrom#eq (p#lfrom) && self#lto#eq (p#lto);
end;;
```

A1.2.2 Sample evaluations

```
let p = new point 456;;
let p' = new point 678;;
let cp = new cpoint (456,Red);;
let cp' = new cpoint (678,Green);;
let dp = new d2point (456,712);;
let dp' = new d2point (678,1024);;
let l = new line (p,p');;
let cl = new line (cp,cp');;

p#pos;;
cp#pos;;
dp#pos;;
((cp#slide 5)#switch )#pos;;
((cp#slide 5)#switch )#col;;

p#col;;                                 (* ill-typed *)
cp#col;;
dp#col;;                                (* ill-typed *)

let f x = x#pos;;
f cp;;

let positions l = List.map (fun x -> x#pos) l;;
positions [p;p'];;
positions [cp;cp'];;
positions [dp;dp'];;
positions [p;cp];;                      (* ill-typed *)
```

```
positions [p;(cp :> point)];;           (* ill-typed *)

p#eq p';;
p#eq cp;;                               (* ill-typed *)
p#eq dp;;                               (* ill-typed *)
cp#eq p;;                               (* ill-typed *)
cp#eq cp;;
cp#eq dp;;                              (* ill-typed *)

l#length;;
l#eq (l#shift 5);;
l#eq cl;;                               (* ill-typed *)
```

A1.2.3 Comments

- Binary methods must be homogeneous (simple dynamic dispatch on self's type). No support for multiple dispatch.
- Supports virtual classes, instance variables, parameterized classes, multiple inheritance, subtyping based coercions.
- Type constraints on objects are "structural". E.g. any object that has variable `x` and methods `eq`, `dist`, `pos`, with types compatible with those expected in class `line` are acceptable as line ends, whether or not generated by a subclass of class `point`.

A1.3 Java version

A1.3.1 Code

```
/* declarations of colors, sq (squaring), and sqrt omitted */

class eqobj {
    public boolean eq(eqobj e)  {throw new Error();}
};

class point extends eqobj {
    public int pos;
    public point (int p) {pos = p;}
    public point slide (int n) {pos = pos+n; return this;}
    public boolean eq (point p) {return (pos == p.pos);}
    public float dist (point p) {return (p.pos - pos);}
};

class cpoint extends point {
    public color col;
    public cpoint (int n,color c) {super(n); col = c;}
    public cpoint paint c' {col = c'; return this;}
    public boolean eq (cpoint p) {return (super.eq(p) && (col == p.col));}
};

class d2point extends point {
    public int ypos;
    public d2point (int x,int y) {super(x); ypos = y;}
/*  public d2point slide (int n) {pos = pos+n; ypos = ypos+n; return this;} */
    public boolean eq (d2point p) {return (super.eq(p) && (ypos == p.ypos));}
    public float dist (d2point p) {return (sqrt (sq (super.dist(p)) + sq (p.ypos - ypos)));}
```

```
};

class line extends eqobj {
    public point from;
    public point to;
    public line(point f, point t) {from=f; to=t;}
    public line shift(int n) {from = from.slide(n); to = to.slide(n); return this;}
    public boolean eq(line l) {return (from.eq(l.from) && to.eq(l.to));}
    public float length(line l) {return (l.from.dist(l.to));}
};
```

A1.3.2 Sample evaluations

```
class test {
    public static void main(String argv[]) {
    point p1 = new point(123);
    point p11 = new point(123);
    point p2 = new point(456);
    cpoint cp1 = new cpoint(123,red);
    cpoint cp11 = new cpoint(123,red);
    cpoint cp2 = new cpoint(456,green);
    cpoint cp3 = new cpoint(123,blue);
    line pl1 = new line(p1,p2);
    line cpl1 = new line(cp1,cp2);
    line pl2 = new line(p2,p1);
    line cpl2 = new line(cp3,cp2);

    p11.slide (6)).pos;                                 /* 129 */
    cp11.col;                                           /* 1    */

    p1.eq(p1);                                          /* true */
    p1.eq(p2);                                          /* false */
    p1.eq(cp1);                                         /* true */
    cp1.eq(p1);                                         /* true */
    cp1.eq(cp1);                                        /* true */
    cp1.eq(cp3);                                        /* false */

    pl1.eq(pl2);                                        /* false */
    pl1.eq(cpl1);                                       /* true */
    cpl1.eq(cpl2);                                      /* true */
    }
}
```

A1.3.3 Comments

- Loss of polymorphism for methods returning self. E.g. method `slide` can't be expressed so that returns an object of the "actual" type of self; invoking `slide` on a `cpoint` returns a `point`. For the same reasons, method `slide` can't be specialized in class `d2point` (or method `slide` in class `point` has to be updated so that it operates imperatively).
- Class `line` can't be defined polymorphic. So, `line` methods invoke the methods of class `point`, rather than those of the exact class of line ends.
- No support for multiple inheritance.

A2 ML code for the various models

All code examples in this appendix can be downloaded from the web page http:://www.laas.fr/~bernard/ooml.html. That page also provides versions of ML equipped with existential dynamics (`dynml`) and subtyping (`subml`).

Except where mentioned, the examples may be run in any SML'97 compliant environment (but not in SML'90, as opaque signature ascription is required). Encodings of the hierarchy making use of existential dynamics may be run with the `dynml` environment, those making use of subtyping may be run with the `subml` environment[1].

A2.1 Prelude code

Encodings load one of the prelude codes in Sections A2.1.3, A2.1.4, or A2.1.5, depending on the features used. These preludes all use library "basic" in A2.1.1, some use in addition library "extensible" in A2.1.2.

A2.1.1 library basic

```
fun explode (f,(s,ext)) = (fn e => f (s,e), ext);
fun implode g (f,(s,ext)) = (f (g s,ext));
datatype color = red | green | blue;
datatype 'a option = None | Some of 'a;
fun I x = x;
fun sq (r : real) = r * r;
val sqrt = Math.sqrt;
```

A2.1.2 library extensible

```
structure Extensible = struct
   type extensible = exn;
   fun create _ =
      let exception X of 'a
          fun matches (X y) = Some y | matches _ = None
      in (X, matches)
      end;
end;
```

A2.1.3 Prelude code for encodings requiring sml'97

```
use "basic";
use "extensible";

structure Closed = struct
  datatype closed = %;
end;

signature XCLOSED = sig
```

1. This is not quite true at the time this report is written, as neither `dynml` nor `subml` currently supports signatures and functors. In this printed version of the appendix, examples making use of dynamics or subtyping are still packaged with signatures and functors, to ease comparison with other examples. The electronic version of the appendix provides in addition versions of these examples packaged with abstypes and type constraints instead, that can be actually run with `dynml` or `subml`.

```
  type closed;
  val % : closed;
  val create : 'a option -> (('a -> closed) * (closed -> 'a option))
end;
structure ExtensibleClosed :> XCLOSED = struct
  open Extensible;
  type closed = extensible;
  val % = let val (x,_) = create (None : unit option) in x () end;
end;
```

A2.1.4 Prelude code for encodings requiring dynamics

```
use "basic";

structure Closed = struct
  datatype closed = %;
end;
```

A2.1.5 Prelude code for encodings requiring subtyping

```
(* subml assumed to provide type 1 (top) and constant % : 1 *)
use "basic";
use "extensible";

structure Closed = struct
  type closed = 1;
  val % : closed = %;
end;

(* for examples that require subtyping and wrapping *)
signature WRAPPED = sig
  type wrapped;
  val create : 'a option -> (('a -> wrapped) * (wrapped -> 'a option))
end;
structure Wrapped :> WRAPPED = struct
  open Extensible
  type wrapped = extensible;
end;
```

A2.2 Non-embedded methods encodings

A2.2.1 Structures, static dispatch

```
(* include smlprelude and run with SML'97 compliant ML or *)
(* include subprelude and run with SubML *)

open Closed;

(* root objects ------------------------------------------------------*)

signature ROOT = sig
   type 'a subself
   type self = closed subself
   val unpack : 'a subself -> ('a -> 'a subself) * 'a
   val pack : unit -> 'a -> 'a subself
   val new : unit -> self
end;

structure Root :> ROOT = struct
(* types *)
   type 'a subself = 'a;
   type self = closed subself;
(* packing *)
   fun unpack ob = (I,ob);
   fun pack () ob = ob;
(* new *)
   fun new () = pack () %;
end;

(* points -------------------------------------------------------------*)

signature POINT = sig
   type 'a ext
   type 'a subself = 'a ext Root.subself
   type self = closed subself
   val unpack : 'a subself -> ('a -> 'a subself) * 'a
   val pack : int -> 'a -> 'a subself
   val pos : 'a subself -> int
   val slide : 'a subself -> int -> 'a subself
   val eq : 'a subself -> 'b subself -> bool
   val dist : 'a subself -> 'b subself -> real
   val new : int -> self
end;

structure Point :> POINT = struct
(* types *)
   type 'a ext = int * 'a;
   type 'a subself = 'a ext Root.subself;
   type self = closed subself;
(* packing *)
   fun unpack p = explode (Root.unpack p);
   fun repack f p = implode f (Root.unpack p);
   fun pack n ext = Root.pack () (n, ext);
(* messages *)
   fun pos p = let val (_,(x,_)) = Root.unpack p in x end;
```

```
   fun slide p n = repack (fn x => x+n) p;
   fun eq p q = pos p = pos q;
   fun dist p q = real (pos q - pos p);
(* new *)
   fun new n = pack n %;
end;

(* colored points  ---------------------------------------------------*)

signature CPOINT = sig
   type 'a ext
   type 'a subself = 'a ext Point.subself;
   type self = closed subself
   val unpack : 'a subself -> ('a -> 'a subself) * 'a
   val pack : color * int -> 'a -> 'a subself
   val eq : 'a subself -> 'b subself -> bool
   val col : 'a subself -> color
   val paint : 'a subself -> color -> 'a subself
   val new : int * color -> self
end;

structure Cpoint :> CPOINT = struct
(* types *)
   type 'a ext = color * 'a;
   type 'a subself = 'a ext Point.subself;
   type self = closed subself;
(* packing *)
   fun unpack p = explode (Point.unpack p);
   fun repack f p = implode f (Point.unpack p);
   fun pack (c,n) ext = Point.pack n (c, ext);
(* messages *)
   fun col p = let val (_,(c,_)) = Point.unpack p in c end;
   fun paint p c = repack (fn _ => c) p;
   fun eq p q = col p = col q andalso Point.eq p q;
(* new *)
   fun new (n,c) : self = pack (c,n) %;
end;

(* 2d points -------------------------------------------------------- *)

signature D2POINT = sig
   type 'a ext
   type 'a subself = 'a ext Point.subself
   type self = closed subself
   val unpack : 'a subself -> ('a -> 'a subself) * 'a
   val pack : int * int -> 'a -> 'a subself
   val xpos : 'a subself -> int
   val ypos : 'a subself -> int
   val slide : 'a subself -> int -> 'a subself
   val eq : 'a subself -> 'b subself -> bool
   val dist : 'a subself -> 'b subself -> real
   val new : int * int -> self
end;

structure D2point :> D2POINT = struct
(* types *)
```

```
   type 'a ext = int * 'a;
   type 'a subself = 'a ext Point.subself;
   type self = closed subself;
(* packing *)
   fun unpack p = explode (Point.unpack p);
   fun repack f p = implode f (Point.unpack p);
   fun pack (y,x) ext = Point.pack x (y, ext);
(* messages *)
   val xpos = Point.pos;
   fun ypos p = let val (_,(y,_)) = Point.unpack p in y end;
   fun slide p n = repack (fn y => y+n) (Point.slide p n);
   fun eq p q = Point.eq p q andalso ypos p = ypos q;
   fun dist p q = sqrt (Real.+ (sq (real (ypos q - ypos p)), sq (Point.dist p q)));
(* new *)
   fun new (x,y) : self = pack (y,x) %;
end;

(* lines -------------------------------------------------------- *)

signature MINIPOINT = sig
   type 'a subself
   type self = closed subself
   val pos : 'a subself -> int
   val slide : 'a subself -> int -> 'a subself
   val eq : 'a subself -> 'b subself -> bool
   val dist : 'a subself -> 'b subself -> real
end;

signature LINE = sig
   structure Param : MINIPOINT
   type ('a,'b) ext
   type ('a,'b) subself = ('a,'b) ext Root.subself
   type 'a self = ('a,closed) subself
   val unpack : ('a,'b) subself -> ('b -> ('a,'b) subself) * 'b
   val pack   : ('a Param.subself * 'a Param.subself) -> 'b -> ('a,'b) subself
   val from   : ('a,'b) subself -> 'a Param.subself
   val to : ('a,'b) subself -> 'a Param.subself
   val shift  : ('a,'b) subself -> int -> ('a,'b) subself
   val eq : ('a,'b) subself -> ('c,'d) subself -> bool
   val length : ('a,'b) subself -> real
   val new: 'a Param.subself * 'a Param.subself -> 'a self
end;

functor Line (structure P : MINIPOINT) :> LINE where Param = P = struct
   structure Param = P;
(* types *)
   type ('a,'b) ext = ('a Param.subself * 'a Param.subself) * 'b;
   type ('a,'b) subself = ('a,'b) ext Root.subself;
   type 'a self = ('a,closed) subself;
(* packing *)
   fun unpack p = explode (Root.unpack p);
   fun repack f p = implode f (Root.unpack p);
   fun pack ft ext = Root.pack () (ft, ext);
(* messages *)
   fun from l = let val (_,((f,_),_)) = Root.unpack l in f end;
   fun to l = let val (_,((_,t),_)) = Root.unpack l in t end;
```

```
   fun shift l n = repack (fn (f,t) => (Param.slide f n,Param.slide t n)) l;
   fun eq p q = Param.eq (from p) (from q) andalso Param.eq (to p) (to q);
   fun length l = Param.dist (from l) (to l);
(* new *)
   fun new (f,t) = pack (f,t) %;
end;

(* as static dispatch is used, must define a line structure for each point class: *)

structure PointLine = Line (structure P = Point);
structure CpointLine = Line (structure P = struct open Point; open Cpoint; end);
structure D2pointLine = Line (structure P = struct open Point; open D2point; end);
```

A2.2.2 Structures, overloading, folding

```
(* include smlprelude and run with SML'97 compliant ML *)

open ExtensibleClosed;

fun new_method f =
  let val M = ref f
      fun get () = !M
      fun over f = M := f (get ())
  in (over,get)
  end;

(* root objects -------------------------------------------------------*)

signature ROOT = sig
   exception Wrap
   type 'a subself
   type self = closed subself
   type wrap
   type 'a packet = 'a * ('a -> wrap) * (wrap -> 'a);
   val wrap'self : self -> wrap
   val self'wrap : wrap -> self
   val unpack : 'a subself -> ('b -> 'b subself) * 'a
   val pack : unit -> 'a -> 'a subself
   val new : unit -> self packet
end;

structure Root :> ROOT = struct
(* types *)
   type 'a subself = 'a;
   type self = closed subself;
   type wrap = self;
   type 'a packet = 'a * ('a -> wrap) * (wrap -> 'a);
(* packing *)
   fun unpack ext = (I,ext);
   fun pack () ext = ext;
(* wrapping *)
   exception Wrap;
   fun wrap'self p = p;
   fun self'wrap p = p;
(* new *)
   fun new () = (pack () %, wrap'self, self'wrap);
end;

(* eqobj ------------------------------------------------------------- *)

signature EQOBJ = sig
   type 'a ext
   type 'a subself = 'a ext Root.subself
   type self = closed subself
   val wrap'self : self -> Root.wrap
   val self'wrap : Root.wrap -> self
   val unpack : 'a subself -> ('b -> 'b subself) * 'a
   val pack : unit -> 'a -> 'a subself
   exception Eq
```

```
   val eq : 'a subself Root.packet -> 'b subself Root.packet -> bool
   val overload_eq : ((Root.wrap -> Root.wrap -> bool) -> (Root.wrap -> Root.wrap -> bool)) -> unit
   val new : unit -> self Root.packet
end;

structure Eqobj :> EQOBJ = struct
(* types *)
   type 'a ext = unit * 'a;
   type 'a subself = 'a ext Root.subself;
   type self = closed subself;
(* packing *)
   fun unpack p = explode (Root.unpack p);
   fun repack f p = implode f (Root.unpack p);
   fun pack () ext = Root.pack () ((), ext);
(* wrapping *)
   val (Self,matches) = create (None : closed ext option);
   fun wrap'self p =
      let val (f,ext) = Root.unpack p in Root.wrap'self (f (Self ext)) end;
   fun self'wrap w =
      let val (f,ww) = Root.unpack (Root.self'wrap w)
      in case matches ww of Some ext => f ext | _ => raise Root.Wrap
      end;
(* messages *)
   exception Eq;
   val (overload_eq, get_eq) =
      let fun eqeq (p : Root.wrap) (q : Root.wrap) : bool = raise Eq in new_method eqeq end;
   fun eq (p,wp,_) (q,wq,_) = get_eq () (wp p) (wq q);
(* new *)
   fun new () = (pack () %, wrap'self, self'wrap);
end;

(* points --------------------------------------------------------------*)

signature POINT = sig
   type 'a ext
   type 'a subself = 'a ext Eqobj.subself
   type self = closed subself
   val wrap'self : self -> Root.wrap
   val self'wrap : Root.wrap -> self
   val unpack : 'a subself -> ('b -> 'b subself) * 'a
   val pack : int -> 'a -> 'a subself
   val pos : 'a subself Root.packet -> int
   val slide : 'a subself Root.packet -> int -> 'a subself Root.packet
   val overload_slide : ((Root.wrap -> int -> Root.wrap) -> (Root.wrap -> int -> Root.wrap)) -> unit
   val dist : 'a subself Root.packet -> 'b subself Root.packet -> real
   val overload_dist : ((Root.wrap -> Root.wrap -> real) -> (Root.wrap -> Root.wrap -> real)) ->
unit
   val new : int -> self Root.packet
end;

structure Point :> POINT = struct
(* types *)
   type 'a ext = int * 'a;
   type 'a subself = 'a ext Eqobj.subself;
   type self = closed subself;
(* packing *)
   fun unpack p = explode (Eqobj.unpack p);
```

```
   fun repack f p = implode f (Eqobj.unpack p);
   fun pack n ext = Eqobj.pack () (n, ext);
(* wrapping *)
   val (Self,matches) = create (None : closed ext option);
   fun wrap'self p =
      let val (f,ext) = Eqobj.unpack p in Eqobj.wrap'self (f (Self ext)) end;
   fun self'wrap w =
      let val (f,ww) = Eqobj.unpack (Eqobj.self'wrap w)
      in case matches ww of Some ext => f ext | _ => raise Root.Wrap
      end;
(* messages *)
   fun wpos p = let val (_, (x,_)) = Eqobj.unpack p in x end;
   fun pos (p,_,_) = wpos p;
   val (overload_slide,get_slide) =
      let fun wslide (p : Root.wrap) (n : int) : Root.wrap =
              wrap'self (repack (fn x => x+n) (self'wrap p))
      in new_method wslide
      end;
   fun slide (p,w,u) n = (u (get_slide () (w p) n),w,u);
   val (overload_dist,get_dist) =
      let fun pdist (p : Root.wrap) (q : Root.wrap) : real =
              real (wpos (self'wrap q) - wpos (self'wrap p));
      in new_method pdist
      end;
   fun dist (p,wp,_) (q,wq,_) = get_dist () (wp p) (wq q);
   val _ = (* specializing eq: *)
      let fun peq eqeq p q =
              wpos (self'wrap p) = wpos (self'wrap q) handle Root.Wrap => eqeq p q
      in Eqobj.overload_eq peq
      end;
(* new *)
   fun new n = (pack n %, wrap'self, self'wrap);
end;

(* colored points  ----------------------------------------------------*)

signature CPOINT = sig
   type 'a ext
   type 'a subself = 'a ext Point.subself
   type self = closed subself
   val wrap'self : self -> Root.wrap
   val self'wrap : Root.wrap -> self
   val unpack : 'a subself -> ('b -> 'b subself) * 'a
   val pack : color * int -> 'a -> 'a subself
   val col : 'a subself Root.packet -> color
   val paint : 'a subself Root.packet -> color -> 'a subself Root.packet
   val overload_paint : ((Root.wrap -> color -> Root.wrap) ->
                         (Root.wrap -> color -> Root.wrap)) -> unit
   val new : int * color -> self Root.packet
end;

structure Cpoint :> CPOINT = struct
(* types *)
   type 'a ext = color * 'a;
   type 'a subself = 'a ext Point.subself;
   type self = closed subself;
```

```
(* packing *)
   fun unpack p = explode (Point.unpack p);
   fun repack f p = implode f (Point.unpack p);
   fun pack (c,n) ext = Point.pack n (c, ext);
(* wrapping *)
   val (Self,matches) = create (None : closed ext option);
   fun wrap'self p =
      let val (f,ext) = Point.unpack p in Point.wrap'self (f (Self ext)) end;
   fun self'wrap w =
      let val (f,ww) = Point.unpack (Point.self'wrap w)
      in case matches ww of Some ext => f ext | _ => raise Root.Wrap
      end;
(* messages *)
   fun wcol p = let val (_,(c,_)) = Point.unpack p in c end;
   fun col (p,w,u) = wcol p;
   val (overload_paint,get_paint) =
      let fun wpaint (p : Root.wrap) (c:color) : Root.wrap =
             wrap'self (repack (fn _ => c) (self'wrap p));
      in new_method wpaint
      end;
   fun paint (p,w,u) c = (u (get_paint () (w p) c),w,u);
   val _ = (* overloads eq *)
      let fun cpeq peq p q =
             (wcol (self'wrap p) = wcol (self'wrap q) andalso peq p q)handle Root.Wrap => peq p q
      in Eqobj.overload_eq cpeq
      end;
(* new *)
   fun new (n,c) = (pack (c,n) %,wrap'self,self'wrap);
end;

(* 2d points ---------------------------------------------------------- *)

signature D2POINT = sig
   type 'a ext
   type 'a subself = 'a ext Point.subself
   type self = closed subself
   val wrap'self : self -> Root.wrap
   val self'wrap : Root.wrap -> self
   val unpack : 'a subself -> ('b -> 'b subself) * 'a
   val pack : int * int -> 'a -> 'a subself
   val xpos : 'a subself Root.packet -> int
   val ypos : 'a subself Root.packet -> int
   val new : int * int -> self Root.packet
end;

structure D2point :> D2POINT = struct
(* types *)
   type 'a ext = int * 'a;
   type 'a subself = 'a ext Point.subself;
   type self = closed subself;
(* packing *)
   fun unpack p = explode (Point.unpack p);
   fun repack f p = implode f (Point.unpack p);
   fun pack (y,x) ext = Point.pack x (y, ext);
(* wrapping *)
   val (Self,matches) = create (None : closed ext option);
```

```
   fun wrap'self p =
      let val (f,ext) = Point.unpack p in Point.wrap'self (f (Self ext)) end;
   fun self'wrap w =
      let val (f,ww) = Point.unpack (Point.self'wrap w)
      in case matches ww of Some ext => f ext | _ => raise Root.Wrap
      end;
(* messages *)
   fun wypos p = let val (_,(y,_)) = Point.unpack p in y end;
   val xpos = Point.pos; (* alias *)
   fun ypos (p,w,u) = wypos p;
   val _ = (* overloads slide *)
      let fun d2slide pslide p n =
              pslide (wrap'self (repack (fn y => y+n) (self'wrap p))) n
              handle Root.Wrap => pslide p n
      in Point.overload_slide d2slide
      end;
   val _ = (* overloads dist *)
      let fun d2dist pdist p q =
              sqrt (sq (real (wypos (self'wrap q) - wypos (self'wrap p))) + sq (pdist p q))
              handle Root.Wrap => pdist p q
      in Point.overload_dist d2dist
      end;
   val _ = (* overloads eq *)
      let fun d2eq peq p q =
              (wypos (self'wrap p) = wypos (self'wrap q) andalso peq p q)
              handle Root.Wrap => peq p q
      in Eqobj.overload_eq d2eq
      end;
(* new *)
   fun new (x,y) = (pack (y,x) %, wrap'self, self'wrap);
end;

(* lines ---------------------------------------------------------------- *)

signature MINIPOINT = sig
   type 'a ext
   type 'a subself
   type self = closed subself
   val slide : 'a subself Root.packet -> int -> 'a subself Root.packet
   val dist : 'a subself Root.packet -> 'b subself Root.packet -> real
   val eq : 'a subself Root.packet -> 'b subself Root.packet -> bool
end;

signature LINE = sig
   structure Param : MINIPOINT
   type 'a ext
   type 'a subself = 'a ext Eqobj.subself
   type self = closed subself
   val wrap'self : self -> Root.wrap
   val self'wrap : Root.wrap -> self
   val unpack : 'a subself -> ('b -> 'b subself) * 'a
   val pack : (Param.self Root.packet * Param.self Root.packet) -> 'a -> 'a subself
   val from : 'a subself Root.packet -> Param.self Root.packet
   val to   : 'a subself Root.packet -> Param.self Root.packet
   val shift : 'a subself Root.packet -> int -> 'a subself Root.packet
   val length : 'a subself Root.packet -> real
```

```
   val overload_length : ((Root.wrap -> real) -> (Root.wrap -> real)) -> unit
   val overload_shift : ((Root.wrap -> int -> Root.wrap) ->
                         (Root.wrap -> int -> Root.wrap)) -> unit
   val new : Param.self Root.packet * Param.self Root.packet -> self Root.packet
end;

functor Line (structure P : MINIPOINT) :> LINE where Param = P = struct
   structure Param = P;
(* types *)
   type 'a ext = (Param.self Root.packet * Param.self Root.packet) * 'a;
   type 'a subself = 'a ext Eqobj.subself;
   type self = closed subself;
(* packing *)
   fun unpack p = explode (Eqobj.unpack p);
   fun repack f p = implode f (Eqobj.unpack p);
   fun pack ft ext = Eqobj.pack () (ft, ext);
(* wrapping *)
   val (Self,matches) = create (None : closed ext option);
   fun wrap'self p =
      let val (f,ext) = Eqobj.unpack p in Eqobj.wrap'self (f (Self ext)) end;
   fun self'wrap w =
      let val (f,ww) = Eqobj.unpack (Eqobj.self'wrap w)
      in case matches ww of Some ext => f ext | _ => raise Root.Wrap
      end;
(* messages *)
   fun wfrom l = let val (_,((f,_),_)) = Eqobj.unpack l in f end;
   fun wto   l = let val (_,((_,t),_)) = Eqobj.unpack l in t end;
   fun from (l,_,_) = wfrom l;
   fun to   (l,_,_) = wto l;
   val (overload_shift,get_shift) =
      let fun wshift (l : Root.wrap) (n:int) : Root.wrap =
              wrap'self (repack (fn (f,t) =>(Param.slide f n, Param.slide t n)) (self'wrap l));
      in new_method wshift
      end;
   fun shift (l,w,u) n = (u (get_shift () (w l) n),w,u);
   val (overload_length,get_length) =
      let fun wlength (l : Root.wrap) : real = Param.dist (wfrom (self'wrap l)) (wto (self'wrap l));
      in new_method wlength
      end;
   fun length (l,w,u) = get_length () (w l);
   val _ = (* overloads eq *)
      let fun leq eqeq p q =
          let val dp = self'wrap p and dq = self'wrap q
          in Param.eq (wfrom dp) (wfrom dq) andalso Param.eq (wto dp) (wto dq)
          end
          handle Root.Wrap => eqeq p q
      in Eqobj.overload_eq leq
      end;
(* new *)
   fun new (f,t) = (pack (f,t) %, wrap'self, self'wrap)
end;

(* as lines are monomorphic, must define a line structure for each point class: *)

structure PointLine =
          Line (structure P = struct
```

```
                                open Point;
                                val eq = Eqobj.eq;
                            end);

structure CpointLine =
          Line (structure P = struct
                                open Cpoint;
                                val slide = Point.slide;
                                val dist = Point.dist;
                                val eq = Eqobj.eq;
                            end);

structure D2pointLine =
          Line (structure P = struct
                                open D2point;
                                val dist = Point.dist;
                                val eq = Eqobj.eq;
                            end);
```

A2.2.3 Structures, overloading, dynamics

```
(* include dynprelude and run with DynML *)

open Closed;

fun new_method f =
   let val M = ref f
       fun get () = !M
       fun over f = M := f (get ())
   in (over,get)
   end;

(* root objects -------------------------------------------------------*)

signature ROOT = sig
   type 'a subself
   type self = closed subself
   type 'a packet = 'a * ('a -> dynamic)
   val unpack : 'a subself -> ('a -> 'a subself) * 'a
   val pack : unit -> 'a -> 'a subself
   val new : unit -> self packet
end;

structure Root :> ROOT = struct
(* types *)
   type 'a subself = 'a;
   type self = closed subself;
   type 'a packet = 'a * ('a -> dynamic);
(* packing *)
   fun unpack ext = (I,ext);
   fun pack () ext = ext;
(* new *)
   fun new () = (pack () %, fn p => p at 'e subself);
end;

(* eqobj -------------------------------------------------------------- *)

signature EQOBJ = sig
   type 'a ext
   type 'a subself = 'a ext Root.subself
   type self = closed subself
   val unpack : 'a subself -> ('a -> 'a subself) * 'a
   val pack : unit -> 'a -> 'a subself
   exception Eq
   val eq : 'a subself Root.packet -> 'b subself Root.packet -> bool
   val overload_eq : ((dynamic -> dynamic -> bool) -> (dynamic -> dynamic -> bool)) -> unit
   val new : unit -> self Root.packet
end;

structure Eqobj :> EQOBJ = struct
(* types *)
   type 'a ext = unit * 'a;
   type 'a subself = 'a ext Root.subself;
   type self = closed subself;
(* packing *)
```

```
   fun unpack p = explode (Root.unpack p);
   fun repack f p = implode f (Root.unpack p);
   fun pack () ext = Root.pack () ((), ext);
(* messages *)
   exception Eq;
   val (overload_eq, get_eq) =
      let fun eqeq (p : dynamic) (q : dynamic) : bool = raise Eq in new_method eqeq end;
   fun eq (p,wp) (q,wq) = get_eq () (wp p) (wq q);
(* new *)
   fun new () = (pack () %, fn p => p at 'e subself);
end;

(* points -------------------------------------------------------------*)

signature POINT = sig
   type 'a ext
   type 'a subself = 'a ext Eqobj.subself
   type self = closed subself
   val unpack : 'a subself -> ('a -> 'a subself) * 'a
   val pack : int -> 'a -> 'a subself
   val pos : 'a subself Root.packet -> int
   val slide : 'a subself Root.packet -> int -> 'a subself Root.packet
   val dist : 'a subself Root.packet -> 'b subself Root.packet -> real
   val overload_dist : ((dynamic -> dynamic -> real) -> (dynamic -> dynamic -> real)) -> unit
   val new : int -> self Root.packet
end;

structure Point :> POINT = struct
(* types *)
   type 'a ext = int * 'a;
   type 'a subself = 'a ext Eqobj.subself;
   type self = closed subself;
(* packing *)
   fun unpack p = explode (Eqobj.unpack p);
   fun repack f p = implode f (Eqobj.unpack p);
   fun pack n ext = Eqobj.pack () (n, ext);
(* messages *)
   fun wpos p = let val (_, (x,_)) = Eqobj.unpack p in x end;
   fun pos (p,_) = wpos p;
   (* slide can't be overloaded *)
   fun slide (p,w) (n:int) = (repack (fn x => x+n) p,w);
   val (overload_dist,get_dist) =
      let fun pdist (p at 'e subself) (q at 'e subself) : real = real (wpos q - wpos p);
      in new_method pdist
      end;
   fun dist (p,wp) (q,wq) = get_dist () (wp p) (wq q);
   val _ = (* specializing eq: *)
      let fun peq eqeq (p at 'e subself) (q at 'e subself) = wpos p = wpos q
            | peq eqeq x y = eqeq x y
      in Eqobj.overload_eq peq
      end;
(* new *)
   fun new n = (pack n %, fn p => p at 'e subself);
end;

(* colored points  ----------------------------------------------------*)
```

```
signature CPOINT = sig
   type 'a ext
   type 'a subself = 'a ext Point.subself
   type self = closed subself
   val unpack : 'a subself -> ('a -> 'a subself) * 'a
   val pack : color * int -> 'a -> 'a subself
   val col : 'a subself Root.packet -> color
   val paint : 'a subself Root.packet -> color -> 'a subself Root.packet
   val new : int * color -> self Root.packet
end;

structure Cpoint :> CPOINT = struct
(* types *)
   type 'a ext = color * 'a;
   type 'a subself = 'a ext Point.subself;
   type self = closed subself;
(* packing *)
   fun unpack p = explode (Point.unpack p);
   fun repack f p = implode f (Point.unpack p);
   fun pack (c,n) ext = Point.pack n (c, ext);
(* messages *)
   fun wcol p = let val (_,(c,_)) = Point.unpack p in c end;
   fun col (p,w) = wcol p;
   (* paint can't be overloaded *)
   fun paint (p,w) c = (repack (fn _ => c) p,w);
   val _ = (* overloads eq *)
      let fun cpeq peq (x as (p at 'e subself)) (y as (q at 'e subself)) =
                       peq x y andalso (wcol p = wcol q)
            | cpeq peq x y = peq x y
      in Eqobj.overload_eq cpeq
      end;
(* new *)
   fun new (n,c) = (pack (c,n) %, fn p => p at 'e subself);
end;

(* 2d points ---------------------------------------------------------- *)

signature D2POINT = sig
   type 'a ext
   type 'a subself = 'a ext Point.subself
   type self = closed subself
   val unpack : 'a subself -> ('a -> 'a subself) * 'a
   val pack : int * int -> 'a -> 'a subself
   val xpos : 'a subself Root.packet -> int
   val ypos : 'a subself Root.packet -> int
   val slide : 'a subself Root.packet -> int ->  'a subself Root.packet
   val new : int * int -> self Root.packet
end;

structure D2point :> D2POINT = struct
(* types *)
   type 'a ext = int * 'a;
   type 'a subself = 'a ext Point.subself;
   type self = closed subself;
(* packing *)
```

```
   fun unpack p = explode (Point.unpack p);
   fun repack f p = implode f (Point.unpack p);
   fun pack (y,x) ext = Point.pack x (y, ext);
(* messages *)
   fun wypos p = let val (_,(y,_)) = Point.unpack p in y end;
   val xpos = Point.pos; (* alias *)
   fun ypos (p,w) = wypos p;
   (* redefining slide: *)
   fun slide (p,w) n = Point.slide (repack (fn y => y+n) p, w) n;
   val _ = (* overloads dist *)
      let fun d2dist pdist (x as (p at `e subself)) (y as (q at `e subself)) =
                  sqrt (sq (pdist x y) + sq (real (wypos q - wypos p)))
            | d2dist pdist x y = pdist x y
      in Point.overload_dist d2dist
      end;
   val _ = (* overloads eq *)
      let fun d2eq peq (x as (p at `e subself)) (y as (q at `e subself)) =
                  peq x y andalso (wypos p = wypos q)
            | d2eq peq x y = peq x y
      in Eqobj.overload_eq d2eq
      end;
(* new *)
   fun new (x,y) = (pack (y,x) %, fn p => p at `e subself);
end;

(* lines ---------------------------------------------------------------- *)
(* Line is a functor since slide can't be overloaded *)

signature MINIPOINT = sig
   type `a ext
   type `a subself
   type self = closed subself
   val slide : `a subself Root.packet -> int -> `a subself Root.packet
   val dist : `a subself Root.packet -> `b subself Root.packet -> real
   val eq : `a subself Root.packet -> `b subself Root.packet -> bool
end;

signature LINE = sig
   structure Param : MINIPOINT
   type (`a,'b) ext
   type (`a,'b) subself = (`a,'b) ext Eqobj.subself
   type `a self = (`a, closed) subself
   val unpack : (`a,'b) subself -> (`b -> (`a,'b) subself) * `b
   val pack : (`a Param.subself Root.packet * `a Param.subself Root.packet) -> `b -> (`a,'b) subself
   val from : (`a,'b) subself Root.packet -> `a Param.subself Root.packet
   val to   : (`a,'b) subself Root.packet -> `a Param.subself Root.packet
   val shift : (`a,'b) subself Root.packet -> int -> (`a,'b) subself Root.packet
   val length : (`a,'b) subself Root.packet -> real
   val new : `a Param.subself Root.packet * `a Param.subself Root.packet -> `a self Root.packet
   val overload_length : ((dynamic -> real) -> (dynamic -> real)) -> unit
end;

functor Line (structure P : MINIPOINT) :> LINE where Param = P = struct
   structure Param = P;
(* types *)
   type (`a,'b) ext = (`a Param.subself Root.packet * `a Param.subself Root.packet) * `b;
```

```
   type ('a,'b) subself = ('a,'b) ext Eqobj.subself;
   type 'a self = ('a, closed) subself;
(* packing *)
   fun unpack p = explode (Eqobj.unpack p);
   fun repack f p = implode f (Eqobj.unpack p);
   fun pack ft ext = Eqobj.pack () (ft, ext);
(* messages *)
   fun wfrom l = let val (_,((f,_),_)) = Eqobj.unpack l in f end;
   fun wto   l = let val (_,((_,t),_)) = Eqobj.unpack l in t end;
   fun from (l,_) = wfrom l;
   fun to   (l,_) = wto l;
   (* shift can't be overloaded *)
   fun shift (l,w) n = (repack (fn (f,t) =>(Param.slide f n, Param.slide t n)) l, w);
   val (overload_length,get_length) =
      let fun wlength (l at ('e,'f) subself) : real = Param.dist (wfrom l) (wto l);
      in new_method wlength
      end;
   fun length (l,w) = get_length () (w l);
   val _ = (* overloads eq *)
      let fun leq _ (p at ('e,'f) subself) (q at ('e,'f) subself) =
                      Param.eq (wfrom p) (wfrom q) andalso Param.eq (wto p) (wto q)
            | leq eqeq x y = eqeq x y
      in Eqobj.overload_eq leq
      end;
(* new *)
   fun new (f,t) = (pack (f,t) %, fn p => p at ('e,'f) subself);
end;

(* as slide can't be overloaded, must define a line structure for each point class: *)

structure PointLine =
          Line (structure P = struct
                                  val eq = Eqobj.eq;
                                  open Point;
                              end);

structure CpointLine =
          Line (structure P = struct
                                  val eq = Eqobj.eq;
                                  val dist = Point.dist;
                                  val slide = Point.slide;
                                  open Cpoint;
                              end);

structure D2pointLine =
          Line (structure P = struct
                                  val eq = Eqobj.eq;
                                  val dist = Point.dist;
                                  open D2point;
                             end);
```

A2.2.4 Functors, static dispatch

```
(* include smlprelude and run with SML'97 compliant ML or *)
(* include subprelude and run with SubML *)

open Closed;

(* root objects -------------------------------------------------------*)

signature ROOT = sig
   type 'a subself
   type self = closed subself
   type 'a ext = 'a subself
   val unpack : 'a subself -> ('a -> 'a subself) * 'a
   val pack : unit -> 'a -> 'a ext
   val new : unit -> self
end;

structure Root :> ROOT = struct
(* types *)
   type 'a subself = 'a
   type 'a ext = 'a subself;
   type self = closed subself;
(* packing *)
   fun unpack ob = (I,ob);
   fun pack () ext = I ext;
(* new *)
   fun new () = pack () %;
end;

(* first coordinate ---------------------------------------------------*)

signature OBJECT = sig
   type 'a ext
   type 'a subself
   type self = closed subself
   val unpack : 'a subself -> ('a -> 'a subself) * 'a
end;

signature PLACED = sig
   structure Super : OBJECT
   type 'a ext
   type 'a subself = 'a ext Super.subself
   type self = closed subself
   val unpack : 'a subself -> ('a -> 'a subself) * 'a
   val pack : int -> 'a -> 'a ext
   val pos : 'a subself -> int
   val slide : 'a subself -> int -> 'a subself
   val eq : 'a subself -> 'b subself -> bool
   val dist : 'a subself -> 'b subself -> real
end;

functor  Place (structure S : OBJECT) :> PLACED where Super = S = struct
   structure Super = S;
(* types *)
   type 'a ext = int * 'a;
```

```
   type 'a subself = 'a ext Super.subself;
   type self = closed subself;
(* packing *)
   fun unpack p = explode (Super.unpack p);
   fun repack f p = implode f (Super.unpack p);
   fun pack n ext = (n, ext);
(* messages *)
   fun pos p = let val (_,(x,_)) = Super.unpack p in x end;
   fun slide p n =repack (fn x => x+n) p;
   fun eq p q = pos p = pos q;
   fun dist p q = real (pos q - pos p);
end;

(* extra coordinates ---------------------------------------------------*)

signature PLACEDOBJ = sig
   type 'a ext
   type 'a subself
   type self = closed subself
   val unpack : 'a subself -> ('a -> 'a subself) * 'a
   val pos : 'a subself -> int
   val slide : 'a subself -> int -> 'a subself
   val eq : 'a subself -> 'b subself -> bool
   val dist : 'a subself -> 'b subself -> real
end;

functor  XPlace (structure S : PLACEDOBJ) :> PLACED where Super = S = struct
   structure Super = S;
(* types *)
   type 'a ext = int * 'a;
   type 'a subself = 'a ext Super.subself;
   type self = closed subself;
(* packing *)
   fun unpack p = explode (Super.unpack p);
   fun repack f p = implode f (Super.unpack p);
   fun pack n ext = (n, ext);
(* messages *)
   fun pos p = let val (_,(x,_)) = Super.unpack p in x end;
   fun slide p n =repack (fn x => x+n) (Super.slide p n);
   fun eq p q = Super.eq p q andalso pos p = pos q;
   fun dist p q = sqrt (Real.+ (sq (Super.dist p q), sq (real (pos q - pos p))))
end;

(* color --------------------------------------------------------------*)

signature PAINTED = sig
   structure Super : OBJECT
   type 'a ext
   type 'a subself = 'a ext Super.subself
   type self = closed subself
   val pack : color -> 'a -> 'a ext
   val unpack : 'a subself -> ('a -> 'a subself) * 'a
   val col : 'a subself -> color
   val paint : 'a subself -> color -> 'a subself
   val eq : 'a subself -> 'b subself -> bool
end;
```

```
functor Paint (structure S : PLACEDOBJ) :> PAINTED where Super = S = struct
   structure Super = S;
(* types *)
   type 'a ext = color * 'a;
   type 'a subself = 'a ext Super.subself;
   type self = closed subself;
(* packing *)
   fun unpack p = explode (Super.unpack p);
   fun repack f p = implode f (Super.unpack p);
   fun pack c ext = (c, ext);
(* messages *)
   fun col cp = let val (_,(c,_)) = Super.unpack cp in c end;
   fun paint cp c = repack (fn _ => c) cp;
   fun eq cp cq = col cp = col cq andalso Super.eq cp cq;
end;

(* line --------------------------------------------------------------- *)

signature MINIPOINT = sig
   type 'a ext
   type 'a subself
   type self = closed subself
   val slide : 'a subself -> int -> 'a subself
   val eq : 'a subself -> 'b subself -> bool
   val dist : 'a subself -> 'b subself -> real
end;

signature LINE = sig
   structure Param : MINIPOINT
   structure Super : OBJECT
   type ('a,'b) ext
   type ('a,'b) subself = ('a,'b) ext Super.subself
   type 'a self = ('a,closed) subself
   val unpack : ('a,'b) subself -> ('b -> ('a,'b) subself) * 'b
   val pack : ('a Param.subself * 'a Param.subself) -> 'b -> ('a,'b) ext
   val from : ('a,'b) subself -> 'a Param.subself
   val to : ('a,'b) subself -> 'a Param.subself
   val shift : ('a,'b) subself -> int -> ('a,'b) subself
   val eq : ('a,'b) subself -> ('c,'d) subself -> bool
   val length : ('a,'b) subself -> real
end;

functor Join (structure P : MINIPOINT and S : OBJECT) :>
                              LINE where Super = S and Param = P = struct
   structure Param = P;
   structure Super = S;
(* types *)
   type ('a,'b) ext = ('a Param.subself * 'a Param.subself) * 'b;
   type ('a,'b) subself = ('a,'b) ext Super.subself;
   type 'a self = ('a,closed) subself;
(* packing *)
   fun unpack p = explode (Super.unpack p);
   fun repack f p = implode f (Super.unpack p);
   fun pack ft ext = (ft, ext);
(* messages *)
```

```
    fun from l = let val (_,((f,_),_)) = Super.unpack l in f end;
    fun to l = let val (_,((_,t),_)) = Super.unpack l in t end;
    fun shift l n =repack (fn (f,t) => (Param.slide f n, Param.slide t n)) l;
    fun eq p q = Param.eq (from p) (from q) andalso Param.eq (to p) (to q);
    fun length l = Param.dist (from l) (to l);
end;

(* class hierarchy. ---------------------------------------------------- *)

structure Point = struct
   local
   structure P = Place (structure S = Root);
   in
   open P;
   fun pack n = Root.pack () o P.pack n;
   fun new n : self = pack n %;
   end;
end;

structure Cpoint = struct
   local
   structure P = Paint (structure S = Point);
   in
   open P;
   fun pack (n,c) = Point.pack n o P.pack c;
   fun new (n,c) : self = pack (n,c) %;
   end;
end;

structure D2point = struct
   local
   structure P = XPlace (structure S = Point);
   in
   open P;
   fun pack (x,y) = Point.pack x o P.pack y;
   fun new (x,y) : self = pack (x,y) %;
   end;
end;

(* as static dispatch is used, must define a line structure for each point class: *)

structure PointLine = struct
   local
   structure L = Join (structure S = Root and P = Point);
   in
   open L;
   fun pack ft = Root.pack () o L.pack ft;
   fun new ft : 'a self = pack ft %;
   end;
end;

structure CpointLine = struct
   local
   structure L = Join (structure S = Root
                       and P = struct
                                  open Cpoint;
```

```
                                    val slide = Point.slide;
                                    val dist = Point.dist;
                               end)
  in
  open L;
  fun pack ft = Root.pack () o L.pack ft;
  fun new ft : 'a self = pack ft %;
  end;
end;

structure D2pointLine = struct
  local
  structure L = Join (structure S = Root and P = D2point);
  in
  open L;
  fun pack ft = Root.pack () o L.pack ft;
  fun new ft : 'a self = pack ft %;
  end;
end;
```

A2.3 Self-application encodings

A2.3.1 Structures, static dispatch

```
(* include smlprelude and run with an SML'97 compliant ML *)

open Closed;

(* root objects -------------------------------------------------------*)

signature ROOT = sig
   type 'a subself
   type self = closed subself
   val unpack : 'a subself -> ('a -> 'a subself) * 'a
   val pack : unit -> 'a -> 'a subself
   val new : unit -> self
end;

structure Root :> ROOT = struct
(* types *)
   type 'a subself = 'a
   type self = closed subself
(* packing *)
   fun unpack ob = (I,ob);
   fun pack () ob = ob;
(* new *)
   fun new () = pack () %;
end;


(* points -------------------------------------------------------------*)

signature POINT = sig
   type 'a ext
   type 'a subself = 'a ext Root.subself
   type self = closed subself
   val unpack : 'a subself -> ('a -> 'a subself) * 'a
   val pack : int -> 'a -> 'a subself
   val pos : 'a subself -> int
   val slide : 'a subself -> int -> 'a subself
   val dist : 'a subself -> 'a subself -> real
   val eq : 'a subself -> 'a subself -> bool
   val new : int -> self
end;

structure Point :> POINT = struct
(* types *)
   datatype 'a state = S of
           ('a subself -> int) *                    (* pos   *)
           ('a subself -> int -> 'a subself) *      (* slide *)
           ('a subself -> 'a subself -> real) *     (* dist  *)
           ('a subself -> 'a subself -> bool)       (* eq    *)
   withtype 'a ext = 'a state * 'a
   and 'a subself = 'a ext Root.subself;
   type self = closed subself;
(* messages *)
```

```
   fun pos p =     let val (_,(S (n,_,_,_),_)) = Root.unpack p in n p end;
   fun slide p n = let val (_,(S (_,s,_,_),_)) = Root.unpack p in s p n end;
   fun dist p q =  let val (_,(S (_,_,d,_),_)) = Root.unpack p in d p q end;
   fun eq p q =    let val (_,(S (_,_,_,e),_)) = Root.unpack p in e p q end;
(* packing *)
   fun unpack p = explode (Root.unpack p);
   fun repack f p = implode (fn S m => S (f m)) (Root.unpack p);
   fun pack n ext =
   let fun peq _ p q = pos p = pos q
       fun methods n =
          let fun mpos p = n
              fun mslide p n' = repack (fn _ => methods (n + n')) p
              fun mdist p q = real (pos q - pos p)
              fun meq p q = pos p = pos q
          in (mpos,mslide,mdist,meq)
          end
      in Root.pack () (S (methods n),ext)
      end;
(* new *)
   fun new n = pack n %;
end;


(* colored points  ----------------------------------------------------*)

signature CPOINT = sig
   type 'a ext
   type 'a subself = 'a ext Point.subself;
   type self = closed subself
   val unpack : 'a subself -> ('a -> 'a subself) * 'a
   val pack : color * int -> 'a -> 'a subself
   val col : 'a subself -> color
   val paint : 'a subself -> color -> 'a subself
   val eq : 'a subself -> 'a subself -> bool
   val new : int * color -> self
end;

structure Cpoint :> CPOINT = struct
(* types *)
   datatype 'a state = S of
          ('a subself -> color) *                  (* col   *)
          ('a subself -> color -> 'a subself) *    (* paint *)
          ('a subself -> 'a subself -> bool)       (* eq    *)
   withtype 'a ext = 'a state * 'a
   and 'a subself = 'a ext Point.subsel;
   type self = closed subself;
(* messages *)
   fun col p =     let val (_,(S (c,_,_),_)) = Point.unpack p in c p end;
   fun paint p c = let val (_,(S (_,t,_),_)) = Point.unpack p in t p c end;
   fun eq p q =    let val (_,(S (_,_,e),_)) = Point.unpack p in e p q end;
(* packing *)
   fun unpack p = explode (Point.unpack p);
   fun repack f p = implode (fn S m => S (f m)) (Point.unpack p);
   fun pack (c,n) ext =
      let fun methods c =
          let fun mcol cp = c
```

```
                fun mpaint cp c' = repack (fn _ => methods c') cp
                fun meq p q = Point.eq p q andalso (col p = col q)
            in (mcol,mpaint,meq)
            end
        in Point.pack n (S (methods c),ext)
        end;
(* new *)
    fun new (n,c) = pack (c,n) %;
end;


(* d2point ----------------------------------------------------*)

signature D2POINT = sig
    type 'a ext
    type 'a subself = 'a ext Point.subself
    type self = closed subself
    val unpack : 'a subself -> ('a -> 'a subself) * 'a
    val pack : int * int -> 'a -> 'a subself
    val xpos : 'a Point.subself -> int
    val ypos : 'a subself -> int
    val slide : 'a subself -> int -> 'a subself
    val dist : 'a subself -> 'a subself -> real
    val eq : 'a subself -> 'a subself -> bool
    val new : int * int -> self
end;

structure D2point :> D2POINT = struct
(* types *)
    datatype 'a state = S of
            ('a subself -> int) *                    (* ypos  *)
            ('a subself -> int -> 'a subself) *      (* slide *)
            ('a subself -> 'a subself -> real) *     (* dist  *)
            ('a subself -> 'a subself -> bool)       (* eq    *)
    withtype 'a ext = 'a state * 'a
    and 'a subself = 'a ext Point.subself;
    type self = closed subself;
(* messages *)
    val xpos = Point.pos; (* alias *)
    fun ypos p =    let val (_,(S (y,_,_,_),_)) = Point.unpack p in y p end;
    fun slide p n = let val (_,(S (_,s,_,_),_)) = Point.unpack p in s p n end;
    fun dist p q =  let val (_,(S (_,_,d,_),_)) = Point.unpack p in d p q end;
    fun eq p q =    let val (_,(S (_,_,_,e),_)) = Point.unpack p in e p q end;
(* packing *)
    fun unpack p = explode (Point.unpack p);
    fun repack f p = implode (fn S m => S (f m)) (Point.unpack p);
    fun pack (x,y) ext =
        let fun methods y =
            let fun mypos p = y
                fun mslide p n = repack (fn _ => methods (y+n)) (Point.slide p n)
                fun mdist p q = sqrt (sq (real (ypos q - ypos p)) + sq (Point.dist p q))
                fun meq p q = Point.eq p q andalso (ypos p = ypos q)
            in (mypos,mslide,mdist,meq)
            end
        in Point.pack x (S (methods y),ext)
        end;
```

```
(* new *)
   fun new (x,y) = pack (x,y) %;
end;

(* line ------------------------------------------------------*)

signature MINIPOINT = sig
   type 'a subself
   type self = closed subself
   val slide : 'a subself -> int -> 'a subself
   val eq : 'a subself -> 'a subself -> bool
   val dist : 'a subself -> 'a subself -> real
end;

signature LINE = sig
   structure Param : MINIPOINT
   type ('a,'b) ext
   type ('a,'b) subself = ('a,'b) ext Root.subself
   type 'a self = ('a,closed) subself
   val unpack : ('a,'b) subself -> ('b -> ('a,'b) subself) * 'b
   val pack : ('a Param.subself * 'a Param.subself) -> 'b -> ('a,'b) subself
   val from : ('a,'b) subself -> 'a Param.subself
   val to : ('a,'b) subself -> 'a Param.subself
   val shift : ('a,'b) subself -> int -> ('a,'b) subself
   val length : ('a,'b) subself -> real
   val eq : ('a,'b) subself -> ('a,'b) subself -> bool
   val new : 'a Param.subself * 'a Param.subself -> 'a self
end;

functor Line (structure P : MINIPOINT) :> LINE where Param=P = struct
   structure Param = P;
(* types *)
   datatype ('a,'b) state = S of
          (('a,'b) subself -> 'a Param.subself) *          (* from   *)
          (('a,'b) subself -> 'a Param.subself) *          (* to     *)
          (('a,'b) subself -> int -> ('a,'b) subself) *    (* shift  *)
          (('a,'b) subself -> real) *                      (* length *)
          (('a,'b) subself -> ('a,'b) subself -> bool)     (* eq     *)
   withtype ('a,'b) ext = ('a,'b) state * 'b
   and ('a,'b) subself = ('a,'b) ext Root.subself;
   type 'a self = ('a,closed) subself;
(* messages *)
   fun from l =    let val (_,(S (f,_,_,_,_),_)) = Root.unpack l in f l end;
   fun to l =      let val (_,(S (_,t,_,_,_),_)) = Root.unpack l in t l end;
   fun shift l n = let val (_,(S (_,_,s,_,_),_)) = Root.unpack l in s l n end;
   fun length l =  let val (_,(S (_,_,_,e,_),_)) = Root.unpack l in e l end;
   fun eq l l' =   let val (_,(S (_,_,_,_,e),_)) = Root.unpack l in e l l' end;
(* packing *)
   fun unpack p = explode (Root.unpack p);
   fun repack f p = implode (fn S m => S (f m)) (Root.unpack p);
   fun pack (f,t) ext =
      let fun methods (f,t) =
          let fun mfr l = f
              fun mto l = t
              fun msh l n = repack (fn _ => methods (Param.slide f n, Param.slide t n)) l
              fun mlen l = Param.dist (from l) (to l)
```

```
                fun meq p q = Param.eq (from p)(from q) andalso Param.eq (to p)(to q)
            in (mfr,mto,msh,mlen,meq)
            end
        in Root.pack () (S (methods (f,t)),ext)
        end;
(* new *)
    fun new (f,t) = pack (f,t) %;
end;

(* as static dispatch is used, must define a line structure for each point class: *)

structure PointLine = Line (structure P = Point);
structure CpointLine = Line (structure P = struct open Point; open Cpoint; end);
structure D2pointLine = Line (structure P = D2point);
```

A2.3.2 Structures, simple dynamic dispatch

```
(* include smlprelude and run with an SML'97 compliant ML *)

open Closed;

(* root objects ------------------------------------------------------*)

signature ROOT = sig
   type 'a subself
   type self = closed subself
   val unpack : 'a subself -> ('a -> 'a subself) * 'a
   val pack : unit -> 'a -> 'a subself
   val new : unit -> self
end;

structure Root :> ROOT = struct
(* types *)
   type 'a subself = 'a
   type self = closed subself
(* packing *)
   fun unpack ob = (I,ob);
   fun pack () ob = ob;
(* new *)
   fun new () = pack () %;
end;

(* eqobjs ------------------------------------------------------------*)

signature EQOBJ = sig
   type 'a ext
   type 'a subself = 'a ext Root.subself
   type self = closed subself
   val unpack : 'a subself -> ('a -> 'a subself) * 'a
   val repack : (('a subself -> 'a subself -> bool) -> ('a subself -> 'a subself -> bool)) ->
                'a subself -> 'a subself
   val pack : unit -> 'a -> 'a subself
   exception Eq;
   val eq : 'a subself -> 'a subself -> bool
   val new : unit -> self
end;

structure Eqobj :> EQOBJ = struct
(* types *)
   datatype 'a state = S of
          ('a subself -> 'a subself -> bool)               (* eq    *)
   withtype 'a ext = 'a state * 'a
   and 'a subself = 'a ext Root.subself;
   type self = closed subself;
(* messages *)
   exception Eq;
   fun eq p q = let val (_,(S e,_)) = Root.unpack p in e p q end;
(* packing *)
   fun unpack p = explode (Root.unpack p);
   fun repack f p = implode (fn S m => S (f m)) (Root.unpack p);
   fun oeq (p : 'a subself) (q : 'a subself) : bool = raise Eq;
```

```
   fun pack () ext = Root.pack () (S oeq,ext);
(* new *)
   fun new () = pack () %;
end;

(* points -------------------------------------------------------------*)

signature POINT = sig
   type 'a ext
   type 'a subself = 'a ext Eqobj.subself
   type self = closed subself
   val unpack : 'a subself -> ('a -> 'a subself) * 'a
   val repack : (('a subself -> int) *
                 ('a subself -> int -> 'a subself) *
                 ('a subself -> 'a subself -> real) ->
                 ('a subself -> int) *
                 ('a subself -> int -> 'a subself) *
                 ('a subself -> 'a subself -> real)) ->
                'a subself -> 'a subself
   val pack : int -> 'a -> 'a subself
   val pos : 'a subself -> int
   val slide : 'a subself -> int -> 'a subself
   val dist : 'a subself -> 'a subself -> real
   val new : int -> self
end;

structure Point :> POINT = struct
(* types *)
   datatype 'a state = S of
          int *
          (('a subself -> int) *                        (* pos *)
           ('a subself -> int -> 'a subself) *          (* slide *)
           ('a subself -> 'a subself -> real))          (* dist  *)
   withtype 'a ext = 'a state * 'a
   and 'a subself = 'a ext Eqobj.subself;
   type self = closed subself;
(* messages *)
   fun pos p     = let val (_,(S (_,(n,_,_)),_)) = Eqobj.unpack p in n p end;
   fun slide p n = let val (_,(S (_,(_,s,_)),_)) = Eqobj.unpack p in s p n end;
   fun dist p q  = let val (_,(S (_,(_,_,d)),_)) = Eqobj.unpack p in d p q end;
(* packing *)
   fun unpack p = explode (Eqobj.unpack p);
   fun repack f p = implode (fn S (s,m) => S (s,f m)) (Eqobj.unpack p);
   fun repacks f p = implode (fn S (s,m) => S (f s, m)) (Eqobj.unpack p);
   fun mpos p = let val (_,(S (s,_),_)) = Eqobj.unpack p in s end;
   fun mslide p n' = repacks (fn n => n + n') p;
   fun mdist p q = real (pos q - pos p);
   fun peq _ p q = pos p = pos q;
   fun pack n ext = Eqobj.repack peq (Eqobj.pack () (S (n,(mpos,mslide,mdist)),ext));
(* new *)
   fun new n = pack n %;
end;

(* colored points  ----------------------------------------------------*)

signature CPOINT = sig
```

```
   type 'a ext
   type 'a subself = 'a ext Point.subself;
   type self = closed subself
   val unpack : 'a subself -> ('a -> 'a subself) * 'a
   val repack : (('a subself -> color) * ('a subself -> color -> 'a subself) ->
                 ('a subself -> color) * ('a subself -> color -> 'a subself)) ->
                'a subself -> 'a subself
   val pack : color * int -> 'a -> 'a subself
   val col : 'a subself -> color
   val paint : 'a subself -> color -> 'a subself
   val new : int * color -> self
end;

structure Cpoint :> CPOINT = struct
(* types *)
   datatype 'a state = S of
          color *
          (('a subself -> color) *                     (* col *)
           ('a subself -> color -> 'a subself))        (* paint *)
   withtype 'a ext = 'a state * 'a
   and 'a subself = 'a ext Point.subself;
   type self = closed subself;
(* messages *)
   fun col p     = let val (_,(S (_,(c,_)),_)) = Point.unpack p in c p end;
   fun paint p c = let val (_,(S (_,(_,t)),_)) = Point.unpack p in t p c end;
(* packing *)
   fun unpack p = explode (Point.unpack p);
   fun repack f p = implode (fn S (s,m) => S (s,f m)) (Point.unpack p);
   fun repacks f p = implode (fn S (s,m) => S (f s,m)) (Point.unpack p);
   fun mcol p = let val (_,(S (s,_),_)) = Point.unpack p in s end;
   fun mpaint p c' = repacks (fn c => c') p;
   fun cpeq peq p q = peq p q andalso col p = col q;
   fun pack (c,n) ext = Eqobj.repack cpeq (Point.pack n (S (c,(mcol,mpaint)),ext));
(* new *)
   fun new (n,c) = pack (c,n) %;
end;

(* d2point ----------------------------------------------------*)

signature D2POINT = sig
   type 'a ext
   type 'a subself = 'a ext Point.subself
   type self = closed subself
   val unpack : 'a subself -> ('a -> 'a subself) * 'a
   val repack : (('a subself -> int) -> ('a subself -> int)) -> 'a subself -> 'a subself
   val pack : int * int -> 'a -> 'a subself
   val xpos : 'a Point.subself -> int
   val ypos : 'a subself -> int
   val new : int * int -> self
end;

structure D2point :> D2POINT = struct
(* types *)
   datatype 'a state = S of
          int *
          ('a subself -> int)                     (* ypos  *)
```

```
   withtype 'a ext = 'a state * 'a
   and 'a subself = 'a ext Point.subself;
   type self = closed subself;
(* messages *)
   val xpos = Point.pos;                          (* alias *)
   fun ypos p = let val (_,(S (_,y),_)) = Point.unpack p in y p end;
(* packing *)
   fun unpack p = explode (Point.unpack p);
   fun repack f p = implode (fn S (s,m) => S (s,f m)) (Point.unpack p);
   fun repacks f p = implode (fn S (s,m) => S (f s,m)) (Point.unpack p);
   fun mypos p = let val (_,(S (s,_),_)) = Point.unpack p in s end;
   fun d2pslide pslide p n = repacks (fn y => y+n) (pslide p n);
   fun d2pdist pdist p q =sqrt (sq (real (ypos q - ypos p)) + sq (pdist p q));
   fun d2peq peq p q = peq p q andalso ypos p = ypos q;
   fun pack (x,y) ext =
      Eqobj.repack d2peq
          (Point.repack (fn (mp,sp,dp) => (mp,d2pslide sp,d2pdist dp))
              (Point.pack x (S (y,mypos),ext)));
(* new *)
   fun new (x,y) = pack (x,y) %;
end;

(* line -------------------------------------------------------*)

signature LINE = sig
   type ('a,'b) ext
   type ('a,'b) subself = ('a,'b) ext Eqobj.subself
   type 'a self = ('a,closed) subself
   val unpack : ('a,'b) subself -> ('b -> ('a,'b) subself) * 'b
   val repack : ((('a,'b) subself -> 'a Point.subself) *
                 (('a,'b) subself -> 'a Point.subself) *
                 (('a,'b) subself -> int -> ('a,'b) subself) *
                 (('a,'b) subself -> real) ->
                 (('a,'b) subself -> 'a Point.subself) *
                 (('a,'b) subself -> 'a Point.subself) *
                 (('a,'b) subself -> int -> ('a,'b) subself) *
                 (('a,'b) subself -> real)) ->
                ('a,'b) subself -> ('a,'b) subself
   val pack : ('a Point.subself * 'a Point.subself) -> 'b -> ('a,'b) subself
   val from : ('a,'b) subself -> 'a Point.subself
   val to : ('a,'b) subself -> 'a Point.subself
   val shift : ('a,'b) subself -> int -> ('a,'b) subself
   val length : ('a,'b) subself -> real
   val new : 'a Point.subself * 'a Point.subself -> 'a self
end;

structure Line :> LINE = struct
   datatype ('a,'b) state = S of
          ('a Point.subself * 'a Point.subself) *
          ((('a,'b) subself -> 'a Point.subself) *          (* from *)
           (('a,'b) subself -> 'a Point.subself) *          (* to   *)
           (('a,'b) subself -> int -> ('a,'b) subself) *    (* shift *)
           (('a,'b) subself -> real))                       (* length *)
   withtype ('a,'b) ext = ('a,'b) state * 'b
   and ('a,'b) subself = ('a,'b) ext Eqobj.subself;
   type 'a self = ('a,closed) subself;
```

```
(* messages *)
   fun from l    = let val (_,(S (_,(f,_,_,_)),_)) = Eqobj.unpack l in f l end;
   fun to l      = let val (_,(S (_,(_,t,_,_)),_)) = Eqobj.unpack l in t l end;
   fun shift l n = let val (_,(S (_,(_,_,s,_)),_)) = Eqobj.unpack l in s l n end;
   fun length l  = let val (_,(S (_,(_,_,_,e)),_)) = Eqobj.unpack l in e l end;
(* packing *)
   fun unpack p = explode (Eqobj.unpack p);
   fun repack f p = implode (fn S (s,m) => S (s,f m)) (Eqobj.unpack p);
   fun repacks f p = implode (fn S (s,m) => S (f s,m)) (Eqobj.unpack p);
   fun state p = let val (_,(S (s,_),_)) = Eqobj.unpack p in s end;
   fun state p = let val (_,(S (s,_),_)) = Eqobj.unpack p in s end;
   fun mfr l = let val (f,_) = state l in f end;
   fun mto l = let val (_,t) = state l in t end;
   fun msh l n = repacks (fn (f,t) => (Point.slide f n, Point.slide t n)) l;
   fun mlen l = Point.dist (from l) (to l);
   fun leq _ p q =Eqobj.eq (from p) (from q) andalso Eqobj.eq (to p) (to q);
   fun pack (f,t) ext = Eqobj.repack leq (Eqobj.pack () (S ((f,t),(mfr,mto,msh,mlen)),ext))
(* new *)
   fun new (f,t) = pack (f,t) %;
end;

(* lines are polymorphic and dynamic dispatch is provided, so a single line class is needed *)

structure PointLine = Line;
```

A2.3.3 Structures, multiple dynamic dispatch, folding

```
(* include smlprelude and run with an SML'97 compliant ML *)

open ExtensibleClosed;

(* root objects -------------------------------------------------------*)

signature ROOT = sig
   type 'a subself
   type self = closed subself
   type wrap
   val pack : unit -> ('a subself -> wrap) -> 'a -> 'a subself
   val unpack : 'a subself -> ('a -> 'a subself) * 'a
   val coerce : ('a subself -> 'b subself) -> ('b subself -> 'a subself) ->
                'a subself -> 'b -> 'b subself
   exception Wrap
   val wrap'self : self -> wrap
   val self'wrap : wrap -> self
   val wrap : 'a subself -> wrap
   val new : unit -> self
end;

structure Root :> ROOT = struct
(* types *)
   datatype 'a state = S of
          'a subself -> wrap                   (* wrap   *)
   withtype 'a subself = 'a state * 'a
   and self = closed subself
   and wrap = self;
(* wrapping *)
   exception Wrap;
   fun wrap'self (p : self) : wrap = p;
   fun self'wrap (p : wrap) : self = p;
   fun coerce c c' ((S w,_) :'a subself) e' : 'b subself = (S (w o c'),e');
(* messages *)
   fun wrap (p as (S w,_)) = w p;
(* packing *)
   fun unpack wob = explode (I,wob);
   fun pack () wrap'curr ext = (S wrap'curr, ext);
(* new *)
   fun new () = pack () wrap'self %;
end;

(* eqobjs -------------------------------------------------------------*)

signature EQOBJ = sig
   type 'a ext
   type 'a subself = 'a ext Root.subself
   type self = closed subself
   val pack   : unit -> ('a subself -> Root.wrap) -> 'a -> 'a subself
   val unpack : 'a subself -> ('a -> 'a subself) * 'a
   val repack : (('a subself -> Root.wrap -> bool) -> ('a subself -> Root.wrap -> bool)) ->
               'a subself -> 'a subself
   val coerce : ('a subself -> 'b subself) -> ('b subself -> 'a subself) ->
                'a subself -> 'b -> 'b subself
```

```
   val wrap'self : self -> Root.wrap
   val self'wrap : Root.wrap -> self
   exception Eq
   val eq : 'a subself -> 'b subself -> bool
   val new : unit -> self
end;

structure Eqobj :> EQOBJ = struct
(* types *)
   datatype 'a state = S of
          'a subself -> Root.wrap -> bool              (* eq    *)
   withtype 'a ext = 'a state * 'a
   and 'a subself = 'a ext Root.subself;
   type self = closed subself;
(* wrapping *)
   val (Self,matches) = create (None : closed ext option);
   fun super'self (p : self) : Root.self =
      let val (_,ext) = Root.unpack p in Root.coerce super'self self'super p (Self ext) end
   and self'super (w : Root.self) : self =
      let val (_,ww) = Root.unpack w
      in case matches ww of
          Some ext => Root.coerce self'super super'self w ext | _ => raise Root.Wrap
      end;
   val wrap'self = Root.wrap'self o super'self;
   val self'wrap = self'super o Root.self'wrap;
   fun coerce c c' (p:'a subself) e' : 'b subself =
      let val (_,(S q,e)) = Root.unpack p in Root.coerce c c' p (S (q o c'),e') end;
(* messages *)
   exception Eq;
   fun eq p q = let val (_,(S e,_)) = Root.unpack p in e p (Root.wrap q) end;
(* packing *)
   fun unpack p = explode (Root.unpack p);
   fun repack f p = implode (fn S m => S (f m)) (Root.unpack p);
   fun oeq p q = raise Eq;
   fun pack () wrap'curr ext = Root.pack () wrap'curr (S oeq,ext);
   fun new () = pack () wrap'self %;
end;

(* points ------------------------------------------------------------*)

signature POINT = sig
   type 'a ext
   type 'a subself = 'a ext Eqobj.subself
   type self = closed subself
   val pack   : int -> ('a subself -> Root.wrap) -> 'a -> 'a subself
   val unpack : 'a subself -> ('a -> 'a subself) * 'a
   val repack : (('a subself -> int) *
                 ('a subself -> int -> 'a subself) *
                 ('a subself -> Root.wrap -> real) ->
                 ('a subself -> int) *
                 ('a subself -> int -> 'a subself) *
                 ('a subself -> Root.wrap -> real)) ->
                'a subself -> 'a subself
   val coerce : ('a subself -> 'b subself) -> ('b subself -> 'a subself) ->
                'a subself -> 'b -> 'b subself
   val wrap'self : self -> Root.wrap
```

```
   val self'wrap : Root.wrap -> self
   val pos : 'a subself -> int
   val slide : 'a subself -> int -> 'a subself
   val dist : 'a subself -> 'b subself -> real
   val new : int -> self
end;

structure Point :> POINT = struct
(* types *)
   datatype 'a state = S of
          int *
          (('a subself -> int) *                       (* pos *)
           ('a subself -> int -> 'a subself) *         (* slide *)
           ('a subself -> Root.wrap -> real))          (* dist  *)
   withtype 'a ext = 'a state * 'a
   and 'a subself = 'a ext Eqobj.subself;
   type self = closed subself;
(* wrapping *)
   val (Self,matches) = create (None : closed ext option);
   fun super'self (p : self) : Eqobj.self =
      let val (_,ext) = Eqobj.unpack p in Eqobj.coerce super'self self'super p (Self ext) end
   and self'super (w : Eqobj.self) : self =
      let val (_,ww) = Eqobj.unpack w
      in case matches ww of
          Some ext => Eqobj.coerce self'super super'self w ext | _ => raise Root.Wrap
      end;
   val wrap'self = Eqobj.wrap'self o super'self;
   val self'wrap = self'super o Eqobj.self'wrap;
   fun coerce c c' (p:'a subself) e' : 'b subself =
      let val (_,(S (x,(n,s,d)),e)) = Eqobj.unpack p
      in Eqobj.coerce c c' p (S (x,(n o c',fn n => c o (s o c') n,d o c')),e')
      end;
(* messages *)
   fun pos p     = let val (_,(S (_,(n,_,_)),_)) = Eqobj.unpack p in n p end;
   fun slide p n = let val (_,(S (_,(_,s,_)),_)) = Eqobj.unpack p in s p n end;
   fun dist p q  = let val (_,(S (_,(_,_,d)),_)) = Eqobj.unpack p in d p (Root.wrap q) end;
(* packing *)
   fun unpack p = explode (Eqobj.unpack p);
   fun repack f p = implode (fn S (s,m) => S (s,f m)) (Eqobj.unpack p);
   fun repacks f p = implode (fn S (s,m) => S (f s,m)) (Eqobj.unpack p);
   fun mpos p = let val (_,(S (s,_),_)) = Eqobj.unpack p in s end;
   fun mslide p n' = repacks (fn n => n + n') p;
   fun mdist p q = real (pos (self'wrap q) - pos p);
   fun peq eqeq p q = pos p = pos (self'wrap q) handle Root.Wrap => eqeq p q;
   fun pack n wrap'curr ext =
      Eqobj.repack peq (Eqobj.pack () wrap'curr (S (n,(mpos, mslide, mdist)),ext));
(* new *)
   fun new n = pack n wrap'self %;
end;

(* colored points  ----------------------------------------------------*)

signature CPOINT = sig
   type 'a ext
   type 'a subself = 'a ext Point.subself
   type self = closed subself
```

```
   val pack   : color * int -> ('a subself -> Root.wrap) -> 'a -> 'a subself
   val unpack : 'a subself -> ('a -> 'a subself) * 'a
   val repack : (('a subself -> color) *
                 ('a subself -> color -> 'a subself) ->
                 ('a subself -> color) *
                 ('a subself -> color -> 'a subself)) ->
                'a subself -> 'a subself
   val coerce : ('a subself -> 'b subself) -> ('b subself -> 'a subself) ->
                'a subself -> 'b -> 'b subself
   val wrap'self : self -> Root.wrap
   val self'wrap : Root.wrap -> self
   val col : 'a subself -> color
   val paint : 'a subself -> color -> 'a subself
   val new : int * color -> self
end;

structure Cpoint :> CPOINT = struct
(* types *)
   datatype 'a state = S of
          color *
          (('a subself -> color) *                    (* col *)
           ('a subself -> color -> 'a subself))        (* paint *)
   withtype 'a ext = 'a state * 'a
   and 'a subself = 'a ext Point.subself;
   type self = closed subself;
(* wrapping *)
   val (Self,matches) = create (None : closed ext option);
   fun super'self (p : self) : Point.self =
      let val (_,ext) = Point.unpack p in Point.coerce super'self self'super p (Self ext) end
   and self'super (w : Point.self) : self =
      let val (_,ww) = Point.unpack w
      in case matches ww of
          Some ext => Point.coerce self'super super'self w ext | _ => raise Root.Wrap
      end;
   val self'wrap = self'super o Point.self'wrap;
   val wrap'self = Point.wrap'self o super'self;
   fun coerce c c' (p:'a subself) e' : 'b subself =
      let val (_,(S (x,(cl,pa)),e)) = Point.unpack p
      in Point.coerce c c' p (S (x,(cl o c',fn co => c o (pa o c') co)),e')
      end;
(* messages *)
   fun col p     = let val (_,(S (_,(c,_)),_)) = Point.unpack p in c p end;
   fun paint p c =let val (_,(S (_,(_,t)),_)) = Point.unpack p in t p c end;
(* packing *)
   fun unpack p = explode (Point.unpack p);
   fun repack f p = implode (fn S (s,m) => S (s,f m)) (Point.unpack p);
   fun repacks f p = implode (fn S (s,m) => S (f s,m)) (Point.unpack p);
   fun mcol p = let val (_,(S (s,_),_)) = Point.unpack p in s end;
   fun mpaint p c' = repacks (fn c => c') p;
   fun cpeq peq p q = (col p = col (self'wrap q) andalso peq p q) handle Root.Wrap => peq p q;
   fun pack (c,n) wrap'curr ext =
      Eqobj.repack cpeq (Point.pack n wrap'curr (S (c,(mcol,mpaint)),ext));
(* new *)
   fun new (n,c) = pack (c,n) wrap'self %;
end;
```

```
(* d2point ---------------------------------------------------*)

signature D2POINT = sig
   type 'a ext
   type 'a subself = 'a ext Point.subself
   type self = closed subself
   val pack   : int * int -> ('a subself -> Root.wrap) -> 'a -> 'a subself
   val unpack : 'a subself -> ('a -> 'a subself) * 'a
   val repack : (('a subself -> int) -> ('a subself -> int)) -> 'a subself -> 'a subself
   val coerce : ('a subself -> 'b subself) -> ('b subself -> 'a subself) ->
                'a subself -> 'b -> 'b subself
   val wrap'self : self -> Root.wrap
   val self'wrap : Root.wrap -> self
   val xpos : 'a subself -> int
   val ypos : 'a subself -> int
   val new : int * int -> self
end;

structure D2point :> D2POINT = struct
(* types *)
   datatype 'a state = S of
          int *
          ('a subself -> int)                      (* ypos  *)
   withtype 'a ext = 'a state * 'a
   and 'a subself = 'a ext Point.subself;
   type self = closed subself;
(* wrapping *)
   val (Self,matches) = create (None : closed ext option);
   fun super'self (p : self) : Point.self =
      let val (_,ext) = Point.unpack p in Point.coerce super'self self'super p (Self ext) end
   and self'super (w : Point.self) : self =
      let val (_,ww) = Point.unpack w
      in case matches ww of
          Some ext => Point.coerce self'super super'self w ext | _ => raise Root.Wrap
      end;
   val wrap'self = Point.wrap'self o super'self;
   val self'wrap = self'super o Point.self'wrap;
   fun coerce c c' (p:'a subself) e' : 'b subself =
      let val (_,(S (x,po),e)) = Point.unpack p in Point.coerce c c' p (S (x,po o c'),e') end;
(* messages *)
   val xpos = Point.pos; (* alias *)
   fun ypos p = let val (_,(S (_,y),_)) = Point.unpack p in y p end;
(* packing *)
   fun unpack p = explode (Point.unpack p);
   fun repack f p = implode (fn S (s,m) => S (s,f m)) (Point.unpack p);
   fun repacks f p = implode (fn S (s,m) => S (f s,m)) (Point.unpack p);
   fun mypos p = let val (_,(S (s,_),_)) = Point.unpack p in s end;
   fun d2pslide pslide p n = repacks (fn s => s+n) (pslide p n);
   fun d2pdist pdist p q =
      sqrt (sq (real (ypos (self'wrap q) - ypos p)) + sq (pdist p q)) handle Root.Wrap => pdist p q;
   fun d2peq peq p q = (ypos p = ypos (self'wrap q) andalso peq p q) handle Root.Wrap => peq p q;
   fun pack (x,y) wrap'curr ext =
      Eqobj.repack d2peq
          (Point.repack (fn (mp,sp,dp) =>(mp,d2pslide sp,d2pdist dp))
              (Point.pack x wrap'curr (S (y,mypos),ext)));
(* new *)
```

```
   fun new (x,y) = pack (x,y) wrap'self %;
end;

(* line ------------------------------------------------------*)

signature MINIPOINT = sig
   type 'a ext
   type 'a subself
   type self = closed subself
   val slide : 'a subself -> int -> 'a subself
   val dist : 'a subself -> 'b subself -> real
   val eq : 'a subself -> 'b subself -> bool
end;

signature LINE = sig
   structure Param : MINIPOINT
   type 'a ext
   type 'a subself = 'a ext Eqobj.subself
   type self = closed subself
   val pack   : Param.self * Param.self -> ('a subself -> Root.wrap) -> 'a -> 'a subself
   val unpack : 'a subself -> ('a -> 'a subself) * 'a
   val repack : (('a subself -> Param.self) *
                 ('a subself -> Param.self) *
                 ('a subself -> int -> 'a subself) *
                 ('a subself -> real) ->
                 ('a subself -> Param.self) *
                 ('a subself -> Param.self) *
                 ('a subself -> int -> 'a subself) *
                 ('a subself -> real)) ->
                'a subself -> 'a subself
   val wrap'self : self -> Root.wrap
   val self'wrap : Root.wrap -> self
   val coerce : ('a subself -> 'b subself) -> ('b subself -> 'a subself) ->
                'a subself -> 'b -> 'b subself
   val from : 'a subself -> Param.self
   val to : 'a subself -> Param.self
   val shift : 'a subself -> int -> 'a subself
   val length : 'a subself -> real
   val new : Param.self * Param.self -> self
end;

functor Line (structure P : MINIPOINT) :> LINE where Param = P = struct
   structure Param = P;
(* types *)
   datatype 'a state = S of
          (Param.self * Param.self) *
          (('a subself -> Param.self) *                (* from *)
           ('a subself -> Param.self) *                (* to   *)
           ('a subself -> int -> 'a subself) *         (* shift *)
           ('a subself -> real))                       (* length *)
   withtype 'a ext = 'a state * 'a
   and 'a subself = 'a ext Eqobj.subself;
   type self = closed subself;
(* wrapping *)
   val (Self,matches) = create (None : closed ext option);
   fun super'self (p : self) : Eqobj.self =
```

```
      let val (_,ext) = Eqobj.unpack p in Eqobj.coerce super'self self'super p (Self ext) end
   and self'super (w : Eqobj.self) : self =
      let val (_,ww) = Eqobj.unpack w
      in case matches ww of
          Some ext => Eqobj.coerce self'super super'self w ext | _ => raise Root.Wrap
      end;
   val self'wrap = self'super o Eqobj.self'wrap;
   val wrap'self = Eqobj.wrap'self o super'self;
   fun coerce c c' (l : 'a subself) e' : 'b subself =
      let val (_,(S (x,(fr,to,sh,le)),e)) = Eqobj.unpack l
      in Eqobj.coerce c c' l (S (x,(fr o c',to o c',fn n => c o (sh o c') n,le o c')),e')
      end;
(* messages *)
   fun from l    = let val (_,(S (_,(f,_,_,_)),_)) = Eqobj.unpack l in f l end;
   fun to l      = let val (_,(S (_,(_,t,_,_)),_)) = Eqobj.unpack l in t l end;
   fun shift l n = let val (_,(S (_,(_,_,s,_)),_)) = Eqobj.unpack l in s l n end;
   fun length l  = let val (_,(S (_,(_,_,_,e)),_)) = Eqobj.unpack l in e l end;
(* packing *)
   fun unpack p = explode (Eqobj.unpack p);
   fun repack f p = implode (fn S (s,m) => S (s,f m)) (Eqobj.unpack p);
   fun repacks f p = implode (fn S (s,m) => S (f s,m)) (Eqobj.unpack p);
   fun state p = let val (_,(S (s,_),_)) = Eqobj.unpack p in s end;
   fun mfr l = let val (f,_) = state l in f end;
   fun mto l = let val (_,t) = state l in t end;
   fun msh l n = repacks (fn (f,t) => (Param.slide f n, Param.slide t n)) l;
   fun mlen l = Param.dist (from l) (to l);
   fun leq eqeq p q =
      (Param.eq (from p) (from (self'wrap q)) andalso Param.eq (to p) (to (self'wrap q)))
      handle Root.Wrap => eqeq p q;
   fun pack (f,t) wrap'curr ext =
      Eqobj.repack leq (Eqobj.pack () wrap'curr (S ((f,t),(mfr,mto,msh,mlen)),ext));
(* new *)
   fun new (f,t) = pack (f,t) wrap'self %;
end;

(* as lines are monomorphic, must provide aline class for each point class *)

structure PointLine =
          Line (structure P = struct open Point; val eq = Eqobj.eq end);

structure CpointLine =
          Line (structure P = struct
                                open Cpoint;
                                val dist = Point.dist;
                                val slide = Point.slide;
                                val eq = Eqobj.eq
                            end);
```

A2.3.4 Structures, multiple dynamic dispatch, dynamics

```
(* include dynprelude and run with DynML *)

open Closed;

(* root objects -------------------------------------------------------*)

signature ROOT = sig
   type 'a subself
   type self = closed subself
   type wrap = dynamic
   val pack   : unit -> ('a subself -> wrap) -> 'a -> 'a subself
   val unpack : 'a subself -> ('a -> 'a subself) * 'a
   val wrap : 'a subself -> wrap
   val new : unit -> self
end;

structure Root :> ROOT = struct
(* types *)
   datatype 'a state = S of
           'a subself -> wrap                   (* wrap   *)
   withtype 'a subself = 'a state * 'a
   and self = closed subself
   and wrap = dynamic;
(* messages *)
   fun wrap (p as (S w,_)) = w p;
(* packing *)
   fun unpack wob = explode (I,wob);
   fun pack () wrap'curr ext = (S wrap'curr, ext);
(* new *)
   fun new () = pack () (fn p => p at 'e subself) %;
end;

(* eqobjs -------------------------------------------------------------*)

signature EQOBJ = sig
   type 'a ext
   type 'a subself = 'a ext Root.subself
   type self = closed subself
   val pack   : unit -> ('a subself -> Root.wrap) -> 'a -> 'a subself
   val unpack : 'a subself -> ('a -> 'a subself) * 'a
   val repack : (('a subself -> Root.wrap -> bool) -> ('a subself -> Root.wrap -> bool)) ->
                'a subself -> 'a subself
   exception Eq
   val eq : 'a subself -> 'b subself -> bool
   val new : unit -> self
end;

structure Eqobj :> EQOBJ = struct
(* types *)
   datatype 'a state = S of
           'a subself -> Root.wrap -> bool      (* eq    *)
   withtype 'a ext = 'a state * 'a
   and 'a subself = 'a ext Root.subself;
   type self = closed subself;
```

```
(* messages *)
   exception Eq;
   fun eq p q = let val (_,(S e,_)) = Root.unpack p in e p (Root.wrap q) end;
(* packing *)
   fun unpack p = explode (Root.unpack p);
   fun repack f p = implode (fn S m => S (f m)) (Root.unpack p);
   fun oeq p w = raise Eq;
   fun pack () wrap'curr ext  = Root.pack () wrap'curr (S oeq,ext);
(* new *)
   fun new () = pack () (fn p => p at 'e subself) %;
end;

(* points -----------------------------------------------------------*)

signature POINT = sig
   type 'a ext
   type 'a subself = 'a ext Eqobj.subself
   type self = closed subself
   val pack   : int -> ('a subself -> Root.wrap) -> 'a -> 'a subself
   val unpack : 'a subself -> ('a -> 'a subself) * 'a
   val repack : (('a subself -> int) *
                 ('a subself -> int -> 'a subself) *
                 ('a subself -> Root.wrap -> real) ->
                 ('a subself -> int) *
                 ('a subself -> int -> 'a subself) *
                 ('a subself -> Root.wrap -> real)) ->
                'a subself -> 'a subself
   val pos : 'a subself -> int
   val slide : 'a subself -> int -> 'a subself
   val dist : 'a subself -> 'b subself -> real
   val new : int -> self
end;

structure Point :> POINT = struct
(* types *)
   datatype 'a state = S of
          int *
          (('a subself -> int) *                    (* pos *)
           ('a subself -> int -> 'a subself) *      (* slide *)
           ('a subself -> Root.wrap -> real))       (* dist  *)
   withtype 'a ext = 'a state * 'a
   and 'a subself = 'a ext Eqobj.subself;
   type self = closed subself;
(* messages *)
   fun pos p     = let val (_,(S (_,(n,_,_)),_)) = Eqobj.unpack p in n p end;
   fun slide p n = let val (_,(S (_,(_,s,_)),_)) = Eqobj.unpack p in s p n end;
   fun dist p q  = let val (_,(S (_,(_,_,d)),_)) = Eqobj.unpack p in d p (Root.wrap q) end;
(* packing *)
   fun unpack p = explode (Eqobj.unpack p);
   fun repack f p = implode (fn S (s,m) => S (s,f m)) (Eqobj.unpack p);
   fun repacks f p = implode (fn S (s,m) => S (f s,m)) (Eqobj.unpack p);
   fun mpos p = let val (_,(S (s,_),_)) = Eqobj.unpack p in s end;
   fun mslide p n' = repacks (fn n => n + n') p;
   fun mdist p (q at 'e subself) = real (pos q - pos p);
   fun peq eqeq p (q at 'e subself) = pos p = pos q
     | peq eqeq p w = eqeq p w;
```

```
   fun pack n wrap'curr ext =
      Eqobj.repack peq (Eqobj.pack () wrap'curr (S (n,(mpos, mslide, mdist)),ext));
(* new *)
   fun new n = pack n (fn p => p at 'e subself) %;
end;

(* colored points  ---------------------------------------------------*)

signature CPOINT = sig
   type 'a ext
   type 'a subself = 'a ext Point.subself
   type self = closed subself
   val pack   : color * int -> ('a subself -> Root.wrap) -> 'a -> 'a subself
   val unpack : 'a subself -> ('a -> 'a subself) * 'a
   val repack : (('a subself -> color) * ('a subself -> color -> 'a subself) ->
                 ('a subself -> color) * ('a subself -> color -> 'a subself)) ->
                'a subself -> 'a subself
   val col : 'a subself -> color
   val paint : 'a subself -> color -> 'a subself
   val new : int * color -> self
end;

structure Cpoint :> CPOINT = struct
(* types *)
   datatype 'a state = S of
           color *
           (('a subself -> color) *                  (* col *)
            ('a subself -> color -> 'a subself))     (* paint *)
   withtype 'a ext = 'a state * 'a
   and 'a subself = 'a ext Point.subself;
   type self = closed subself;
(* messages *)
   fun col p     = let val (_,(S (_,(c,_)),_)) = Point.unpack p in c p end;
   fun paint p c =let val (_,(S (_,(_,t)),_)) = Point.unpack p in t p c end;
(* packing *)
   fun unpack p = explode (Point.unpack p);
   fun repack f p = implode (fn S (s,m) => S (s,f m)) (Point.unpack p);
   fun repacks f p = implode (fn S (s,m) => S (f s,m)) (Point.unpack p);
   fun mcol p = let val (_,(S (s,_),_)) = Point.unpack p in s end;
   fun mpaint p c' = repacks (fn c => c') p;
   fun cpeq peq p (w as (q at 'e subself)) = col p = col q andalso peq p w
     | cpeq peq p w = peq p w;
   fun pack (c,n) wrap'curr ext =
      Eqobj.repack cpeq (Point.pack n wrap'curr (S (c,(mcol,mpaint)),ext));
(* new *)
   fun new (n,c) = pack (c,n) (fn p => p at 'e subself) %;
end;

(* d2point ---------------------------------------------------*)

signature D2POINT = sig
   type 'a ext
   type 'a subself = 'a ext Point.subself
   type self = closed subself
   val pack   : int * int -> ('a subself -> Root.wrap) -> 'a -> 'a subself
   val unpack : 'a subself -> ('a -> 'a subself) * 'a
```

```
   val repack : (('a subself -> int) -> ('a subself -> int)) -> 'a subself -> 'a subself
   val xpos : 'a subself -> int
   val ypos : 'a subself -> int
   val new : int * int -> self
end;

structure D2point :> D2POINT = struct
(* types *)
   datatype 'a state = S of
          int *
          ('a subself -> int)                    (* ypos  *)
   withtype 'a ext = 'a state * 'a
   and 'a subself = 'a ext Point.subself;
   type self = closed subself;
(* messages *)
   val xpos = Point.pos; (* alias *)
   fun ypos p = let val (_,(S (_,y),_)) = Point.unpack p in y p end;
(* packing *)
   fun unpack p = explode (Point.unpack p);
   fun repack f p = implode (fn S (s,m) => S (s,f m)) (Point.unpack p);
   fun repacks f p = implode (fn S (s,m) => S (f s,m)) (Point.unpack p);
   fun mypos p = let val (_,(S (s,_),_)) = Point.unpack p in s end;
   fun d2pslide pslide p n = repacks (fn s => s+n) (pslide p n);
   fun d2pdist pdist p (w as (q at 'e subself)) =
          sqrt (sq (real (ypos q - ypos p)) + sq (pdist p w))
     | d2pdist pdist p w = pdist p w;
   fun d2peq peq p (w as (q at 'e subself)) = ypos p = ypos q andalso peq p w
     | d2peq peq p w = peq p w;
   fun pack (x,y) wrap'curr ext =
      Eqobj.repack d2peq
          (Point.repack (fn (mp,sp,dp) =>(mp,d2pslide sp,d2pdist dp))
              (Point.pack x wrap'curr (S (y,mypos),ext)));
(* new *)
   fun new (x,y) = pack (x,y) (fn p => p at 'e subself) %;
end;

(* line -------------------------------------------------------*)

signature LINE = sig
   type ('a,'b) ext
   type ('a,'b) subself = ('a,'b) ext Eqobj.subself
   type 'a self = ('a, closed) subself
   val pack : 'a Point.subself * 'a Point.subself -> (('a,'b) subself -> Root.wrap) ->
              'b -> ('a,'b) subself
   val unpack : ('a,'b) subself -> ('b -> ('a,'b) subself) * 'b
   val repack : ((('a,'b) subself -> 'a Point.subself) *
                (('a,'b) subself -> 'a Point.subself) *
                (('a,'b) subself -> int -> ('a,'b) subself) *
                (('a,'b) subself -> real) ->
                (('a,'b) subself -> 'a Point.subself) *
                (('a,'b) subself -> 'a Point.subself) *
                (('a,'b) subself -> int -> ('a,'b) subself) *
                (('a,'b) subself -> real)) ->
               ('a,'b) subself -> ('a,'b) subself
   val from : ('a,'b) subself -> 'a Point.subself
   val to : ('a,'b) subself -> 'a Point.subself
```

```
   val shift : ('a,'b) subself -> int -> ('a,'b) subself
   val length : ('a,'b) subself -> real
   val new : 'a Point.subself * 'a Point.subself -> 'a self
end;

structure Line :> LINE = struct
(* types *)
   datatype ('a,'b) state = S of
           ('a Point.subself * 'a Point.subself) *
           ((('a,'b) subself -> 'a Point.subself) *         (* from *)
            (('a,'b) subself -> 'a Point.subself) *         (* to   *)
            (('a,'b) subself -> int -> ('a,'b) subself)*    (* shift *)
            (('a,'b) subself -> real))                      (* length *)
   withtype ('a,'b) ext = ('a,'b) state * 'b
   and ('a,'b) subself = ('a,'b) ext Eqobj.subself;
   type 'a self = ('a, closed) subself;
(* messages *)
   fun from l    = let val (_,(S (_,(f,_,_,_)),_)) = Eqobj.unpack l in f l end;
   fun to l      = let val (_,(S (_,(_,t,_,_)),_)) = Eqobj.unpack l in t l end;
   fun shift l n = let val (_,(S (_,(_,_,s,_)),_)) = Eqobj.unpack l in s l n end;
   fun length l  = let val (_,(S (_,(_,_,_,e)),_)) = Eqobj.unpack l in e l end;
(* packing *)
   fun unpack p = explode (Eqobj.unpack p);
   fun repack f p = implode (fn S (s,m) => S (s,f m)) (Eqobj.unpack p);
   fun repacks f p = implode (fn S (s,m) => S (f s,m)) (Eqobj.unpack p);
   fun state p = let val (_,(S (s,_),_)) = Eqobj.unpack p in s end;
   fun mfr l = let val (f,_) = state l in f end;
   fun mto l = let val (_,t) = state l in t end;
   fun msh l n = repacks (fn (f,t) => (Point.slide f n, Point.slide t n)) l;
   fun mlen l = Point.dist (from l) (to l);
   fun leq eqeq p (w as (q at ('e,'f) subself)) =
              Eqobj.eq (from p) (from q) andalso Eqobj.eq (to p) (to q)
     | leq eqeq p w = eqeq p w;
   fun pack (f,t) wrap'curr ext =
      Eqobj.repack leq (Eqobj.pack () wrap'curr (S ((f,t),(mfr,mto,msh,mlen)),ext));
(* new *)
   fun new (f,t) = pack (f,t) (fn l => l at ('e,'f) subself) %;
end;

(* lines are polymorphic and dynamic dispatch is provided, so a single line class is needed *)

structure PointLine = Line;
```

A2.3.5 Functors, simple dynamic dispatch

```
(* include smlprelude and run with an SML'97 compliant ML *)

open Closed;

(* root objects ------------------------------------------------------- *)

signature ROOT = sig
   type 'a subself
   type 'a ext = 'a subself
   type self = closed subself
   val pack : unit -> ('a subself -> 'a subself) * ('a -> 'a ext)
   val unpack : 'a subself -> ('a -> 'a subself) * 'a
   val new : unit -> self
end;

structure Root :> ROOT = struct
(* types *)
   type 'a subself = 'a;
   type self = closed subself;
   type 'a ext = 'a subself;
(* packing *)
   fun unpack ob = (I,ob);
   fun pack () = (I,I);
(* new *)
   fun new () = %;
end;

signature OBJECT = sig
   type 'a ext
   type 'a subself
   type self = closed subself
   val unpack : 'a subself -> ('a -> 'a subself) * 'a
end;


(* equality -------------------------------------------------------------*)

signature EQUAL = sig
   structure Super : OBJECT
   type 'a ext
   type 'a subself = 'a ext Super.subself
   type self = closed subself
   val pack : unit -> ('a subself -> 'a subself) * ('a -> 'a ext)
   val unpack : 'a subself -> ('a -> 'a subself) * 'a
   val repack : (('a subself -> 'a subself -> bool) ->
                 ('a subself -> 'a subself -> bool)) ->
                'a subself -> 'a subself
   val eq : 'a subself -> 'a subself -> bool
end;

functor Equality (structure S : OBJECT) :> EQUAL where Super = S = struct
   structure Super = S;
(* types *)
   datatype 'a state = S of
```

```
          'a subself -> 'a subself -> bool         (* eq *)
   withtype 'a ext = 'a state * 'a
   and 'a subself  = 'a ext Super.subself;
   type self = closed subself;
(* messages *)
   exception Eq;
   fun eq p q = let val (_,(S e,_)) = Super.unpack p in e p q end;
(* packing *)
   fun unpack p = explode (Super.unpack p);
   fun repack f p = implode (fn S m => S (f m)) (Super.unpack p);
   fun oeq p q = raise Eq;
   fun pack () = (I, fn ext => (S oeq,ext));
end;

signature EQOBJECT = sig
   type 'a subself
   type self = closed subself
   val unpack : 'a subself -> ('a -> 'a subself) * 'a
   val repack_eq : (('a subself -> 'a subself -> bool) ->
                    ('a subself -> 'a subself -> bool)) ->
                   'a subself -> 'a subself
end;


(* first coordinate ----------------------------------------------------------*)

signature PLACED = sig
   structure Super : EQOBJECT
   type 'a ext
   type 'a subself = 'a ext Super.subself
   type self = closed subself
   val pack : int -> ('a subself -> 'a subself) * ('a -> 'a ext)
   val unpack : 'a subself -> ('a -> 'a subself) * 'a
   val repack : (('a subself -> int) *
                 ('a subself -> 'a subself -> real) *
                 ('a subself -> int -> 'a subself) ->
                 ('a subself -> int) *
                 ('a subself -> 'a subself -> real) *
                 ('a subself -> int -> 'a subself)) ->
                'a subself -> 'a subself
   val pos : 'a subself -> int
   val slide : 'a subself -> int -> 'a subself
   val dist : 'a subself -> 'a subself -> real
end;

functor Place (structure S : EQOBJECT) :> PLACED where Super = S = struct
   structure Super = S;
(* types *)
   datatype 'a state = S of
          int *
          (('a subself -> int) *(* pos   *)
           ('a subself -> 'a subself -> real) *        (* dist *)
           ('a subself -> int -> 'a subself))          (* slide *)
   withtype 'a ext = 'a state * 'a
   and 'a subself  = 'a ext Super.subself;
   type self = closed subself;
```

```
(* messages *)
   fun pos p     = let val (_,(S (_,(n,_,_)),_)) = Super.unpack p in n p end;
   fun dist p q  = let val (_,(S (_,(_,d,_)),_)) = Super.unpack p in d p q end;
   fun slide p n = let val (_,(S (_,(_,_,s)),_)) = Super.unpack p in s p n end;
(* packing *)
   fun unpack p = explode (Super.unpack p);
   fun repack f p = implode (fn S (s,m) => S (s,f m)) (Super.unpack p);
   fun repacks f p = implode (fn S (s,m) => S (f s, m)) (Super.unpack p);
   fun mpos p = let val (_,(S (s,_),_)) = Super.unpack p in s end;
   fun mslide p n' = repacks (fn n => n + n') p;
   fun mdist p q = real (pos q - pos p);
   fun peq _ p q = pos p = pos q;
   fun pack n = (Super.repack_eq peq, fn ext => (S (n,(mpos,mdist,mslide)),ext));
end;

signature PLACEDOBJ = sig
   type 'a subself
   type self = closed subself
   val unpack : 'a subself -> ('a -> 'a subself) * 'a
   val repack_eq : (('a subself -> 'a subself -> bool) ->
                    ('a subself -> 'a subself -> bool)) ->
                  'a subself -> 'a subself
   val repack_pos : (('a subself -> int) *
                     ('a subself -> 'a subself -> real) *
                     ('a subself -> int -> 'a subself) ->
                     ('a subself -> int) *
                     ('a subself -> 'a subself -> real) *
                     ('a subself -> int -> 'a subself)) ->
                    'a subself -> 'a subself
end;


(* extra coordinates ----------------------------------------------------*)

signature XPLACED = sig
   structure Super : PLACEDOBJ
   type 'a ext
   type 'a subself = 'a ext Super.subself
   type self = closed subself
   val pack : int -> ('a subself -> 'a subself) * ('a -> 'a ext)
   val unpack : 'a subself -> ('a -> 'a subself) * 'a
   val repack : (('a subself -> int) -> ('a subself -> int)) -> 'a subself -> 'a subself
   val pos : 'a subself -> int
end;

functor XPlace (structure S : PLACEDOBJ) :> XPLACED where Super = S = struct
   structure Super = S;
(* types *)
   datatype 'a state = S of
           int *
           ('a subself -> int)                     (* pos *)
   withtype 'a ext = 'a state * 'a
   and 'a subself  = 'a ext Super.subself;
   type self = closed subself;
(* messages *)
   fun pos p = let val (_,(S (_,n),_)) = Super.unpack p in n p end;
```

```
(* packing *)
   fun unpack p = explode (Super.unpack p);
   fun repack f p = implode (fn S (s,m) => S (s,f m)) (Super.unpack p);
   fun repacks f p = implode (fn S (s,m) => S (f s,m)) (Super.unpack p);
   fun mpos p = let val (_,(S (s,_),_)) = Super.unpack p in s end;
   fun pslide dslide p n = repacks (fn y => y+n) (dslide p n);
   fun pdist ddist p q =sqrt (sq (real (pos q - pos p)) + sq (ddist p q));
   fun peq dpeq p q = dpeq p q andalso pos p = pos q;
   fun pack n =
      (Super.repack_eq peq o Super.repack_pos (fn (mp,dp,sp) => (mp,pdist dp,pslide sp)),
       fn ext => (S (n,mpos),ext));
end;

(* color -------------------------------------------------------------*)

signature PAINTED = sig
   structure Super : EQOBJECT
   type 'a ext
   type 'a subself = 'a ext Super.subself
   type self = closed subself
   val pack : color -> ('a subself -> 'a subself) * ('a -> 'a ext)
   val unpack : 'a subself -> ('a -> 'a subself) * 'a
   val repack : (('a subself -> color) *
                 ('a subself -> color -> 'a subself) ->
                 ('a subself -> color) *
                 ('a subself -> color -> 'a subself)) ->
                'a subself -> 'a subself
   val col : 'a subself -> color
   val paint : 'a subself -> color -> 'a subself
end;

functor Paint (structure S : EQOBJECT) :> PAINTED where Super = S = struct
   structure Super = S;
(* types *)
   datatype 'a state = S of
          color *
          (('a subself -> color) *                (* col   *)
           ('a subself -> color -> 'a subself))   (* paint *)
   withtype 'a ext = 'a state * 'a
   and 'a subself = 'a ext Super.subself;
   type self = closed subself;
(* messages *)
   fun col p     = let val (_,(S (_,(c,_)),_)) = Super.unpack p in c p end;
   fun paint p c = let val (_,(S (_,(_,pa)),_)) = Super.unpack p in pa p c end;
(* packing *)
   fun unpack p = explode (Super.unpack p);
   fun repack f p = implode (fn S (s,m) => S (s,f m)) (Super.unpack p);
   fun repacks f p = implode (fn S (s,m) => S (f s,m)) (Super.unpack p);
   fun mcol p = let val (_,(S (s,_),_)) = Super.unpack p in s end;
   fun mpaint p c' = repacks (fn c => c') p;
   fun cpeq peq p q = peq p q andalso col p = col q;
   fun pack c = (Super.repack_eq cpeq, fn ext => (S (c,(mcol,mpaint)),ext));
end;

(* line ---------------------------------------------------------*)
```

```
signature MINIPOINT = sig
   type 'a subself
   type self = closed subself
   val slide : 'a subself -> int -> 'a subself
   val dist : 'a subself -> 'a subself -> real
   val eq : 'a subself -> 'a subself -> bool
end;

signature LINE = sig
   structure Param : MINIPOINT
   structure Super : EQOBJECT
   type ('a,'b) ext
   type ('a,'b) subself = ('a,'b) ext Super.subself
   type 'a self = ('a, closed) subself
   val pack : 'a Param.subself * 'a Param.subself ->
              (('a,'b) subself -> ('a,'b) subself) * ('b -> ('a,'b) ext)
   val unpack : ('a,'b) subself -> ('b -> ('a,'b) subself) * 'b
   val repack : ((('a,'b) subself -> 'a Param.subself) *
                 (('a,'b) subself -> 'a Param.subself) *
                 (('a,'b) subself -> int -> ('a,'b) subself) *
                 (('a,'b) subself -> real) ->
                 (('a,'b) subself -> 'a Param.subself) *
                 (('a,'b) subself -> 'a Param.subself) *
                 (('a,'b) subself -> int -> ('a,'b) subself) *
                 (('a,'b) subself -> real)) ->
                ('a,'b) subself -> ('a,'b) subself
   val from : ('a,'b) subself -> 'a Param.subself
   val to : ('a,'b) subself -> 'a Param.subself
   val shift : ('a,'b) subself -> int -> ('a,'b) subself
   val length : ('a,'b) subself -> real
end;

functor Join (structure P : MINIPOINT and S : EQOBJECT) :>
                              LINE where Super = S and Param = P = struct
   structure Param = P;
   structure Super = S;
(* types *)
   datatype ('a,'b) state = S of
          ('a Param.subself * 'a Param.subself) *
          ((('a,'b) subself -> 'a Param.subself) *         (* from *)
           (('a,'b) subself -> 'a Param.subself) *         (* to   *)
           (('a,'b) subself -> int -> ('a,'b) subself) *   (* shift *)
           (('a,'b) subself -> real))                      (* length *)
   withtype ('a,'b) ext = ('a,'b) state * 'b
   and ('a,'b) subself = ('a,'b) ext Super.subself;
   type 'a self = ('a,closed) subself;
(* messages *)
   fun from l    = let val (_,(S (_,(f,_,_,_)),_)) = Super.unpack l in f l end;
   fun to l      = let val (_,(S (_,(_,t,_,_)),_)) = Super.unpack l in t l end;
   fun shift l n = let val (_,(S (_,(_,_,s,_)),_)) = Super.unpack l in s l n end;
   fun length l  = let val (_,(S (_,(_,_,_,le)),_)) = Super.unpack l in le l end;
(* packing *)
   fun unpack p = explode (Super.unpack p);
   fun repack f p = implode (fn S (s,m) => S (s,f m)) (Super.unpack p);
   fun repacks f p = implode (fn S (s,m) => S (f s,m)) (Super.unpack p);
   fun state p = let val (_,(S (s,_),_)) = Super.unpack p in s end;
```

```
    fun state p = let val (_,(S (s,_),_)) = Super.unpack p in s end;
    fun mfr l = let val (f,_) = state l in f end;
    fun mto l = let val (_,t) = state l in t end;
    fun msh l n = repacks (fn (f,t) => (Param.slide f n, Param.slide t n)) l;
    fun mlen l = Param.dist (from l) (to l);
    fun leq _ p q =Param.eq (from p) (from q) andalso Param.eq (to p) (to q);
    fun pack (f,t) = (Super.repack_eq leq, fn ext => (S ((f,t),(mfr,mto,msh,mlen)), ext));
end;


(* sample class hierarchy ----------------------------------------------------- *)

infix 7 oo;
fun (f,g) oo (F,G) = (f o F, G o g);
fun make (f,g) = f (g %);

structure Eqobj = struct
   local
   structure P = Equality (structure S = Root);
   in
   open P;
   fun pack () = P.pack () oo Root.pack ();
   fun new () : self = make (pack ());
   end;
end;

structure Point = struct
   local
   structure P = Place (structure S = struct open Eqobj; val repack_eq = Eqobj.repack; end);
   in
   open P;
   fun pack n = P.pack n oo Eqobj.pack ();
   fun new n : self = make (pack n);
   end;
end;

structure Cpoint = struct
   local
   structure P = Paint (structure S = struct open Point; val repack_eq = Eqobj.repack; end);
   in
   open P;
   fun pack (n,c) = P.pack c oo Point.pack n;
   fun new (n,c) : self = make (pack (n,c));
   end;
end;

structure D2point = struct
   local
   structure P = XPlace (structure S = struct
                                          open Point;
                                          val repack_eq = Eqobj.repack;
                                          val repack_pos =  Point.repack;
                                      end);
   in
   open P;
   fun pack (x,y) = P.pack y oo Point.pack x;
```

```
   fun new (x,y) : self = make (pack (x,y));
   end;
end;

structure D3point = struct
   local
   structure P = XPlace (structure S = struct
                                          open D2point;
                                          val repack_eq = Eqobj.repack;
                                          val repack_pos =  Point.repack;
                                      end);
   in
   open P;
   fun pack (x,y,z) = P.pack z oo D2point.pack (x,y);
   fun new (x,y,z) : self = make (pack (x,y,z));
   end;
end;

structure Cd3point = struct
   local
   structure P = Paint (structure S = struct open D3point; val repack_eq = Eqobj.repack; end);
   in
   open P;
   fun pack (x,y,z,c) = P.pack c oo D3point.pack (x,y,z);
   fun new (x,y,z,c) : self = make (pack (x,y,z,c));
   end;
end;

structure Line = struct
   local
   structure L = Join (structure P = struct open Point; val eq = Eqobj.eq; end
                       and S = struct open Eqobj; val repack_eq = Eqobj.repack; end);
   in
   open L;
   fun pack (f,t) = L.pack (f,t) oo Eqobj.pack ();
   fun new (f,t) : 'a self = make (pack (f,t));
   end;
end;

(* lines are polymorphic and dynamic dispatch is provided, so a single line class is needed *)

structure PointLine = line;
```

A2.4 Recursive-self encodings

A2.4.1 Structures, simple dynamic dispatch

```
(* include smlprelude and run with an SML'97 compliant ML *)

open Closed;

(* root objects ------------------------------------------------------*)

signature ROOT = sig
   type 'a subself
   type self = closed subself
   val pack : unit -> 'a -> (unit -> 'a subself) -> 'a subself
   val unpack : 'a subself -> ('a -> 'a subself) * 'a
   val new : unit -> self
end;

structure Root :> ROOT = struct
(* types *)
   type 'a subself = 'a;
   type self = closed subself;
(* packing *)
   fun unpack wob = (I,wob);
   fun pack () ob self = ob;
(* new *)
   fun new () = let fun self () = pack () % self in self () end;
end;

(* eqobjs ------------------------------------------------------------ *)

signature EQOBJ = sig
   type 'a ext
   type 'a subself = 'a ext Root.subself
   type self = closed subself
   val pack : unit -> 'a -> (unit -> unit -> 'a subself) -> 'a subself
   val unpack : 'a subself -> ('a -> 'a subself) * 'a
   val repack : (('a subself -> bool) -> ('a subself -> bool)) -> 'a subself -> 'a subself
   exception Eq;
   val eq : 'a subself -> 'a subself -> bool
   val new : unit -> self
end;

structure Eqobj :> EQOBJ = struct
(* types *)
   datatype 'a state = S of 'a subself -> bool          (* eq    *)
   withtype 'a ext = 'a state * 'a
   and 'a subself = 'a ext Root.subself;
   type self = closed subself;
(* messages *)
   exception Eq;
   fun eq p q = let val (_,(S e,_)) = Root.unpack p in e q end;
(* packing *)
   fun unpack p = explode (Root.unpack p);
   fun repack f p = implode (fn S m => S (f m)) (Root.unpack p);
   fun pack () ext self = let fun eqeq w = raise Eq in Root.pack () (S eqeq, ext) (self ()) end;
```

```
(* new *)
   fun new () = let fun self () () = pack () % self in self () () end;
end;

(* points -------------------------------------------------------------*)

signature POINT = sig
   type 'a ext
   type 'a subself = 'a ext Eqobj.subself
   type self = closed subself
   val pack : int -> 'a -> (int -> unit -> unit -> 'a subself) -> 'a subself
   val unpack : 'a subself -> ('a -> 'a subself) * 'a
   val repack : (int * (int -> 'a subself) * ('a subself -> real) ->
                 int * (int -> 'a subself) * ('a subself -> real)) ->
                'a subself -> 'a subself
   val pos : 'a subself -> int
   val slide : 'a subself -> int -> 'a subself
   val dist : 'a subself -> 'a subself -> real
   val new : int -> self
end;

structure Point :> POINT = struct
(* types *)
   datatype 'a state = S of
           int *                                  (* pos   *)
           (int -> 'a subself) *                  (* slide *)
           ('a subself -> real)                   (* dist  *)
   withtype 'a ext = 'a state * 'a
   and 'a subself = 'a ext Eqobj.subself;
   type self = closed subself;
(* messages *)
   fun pos p = let val (_,(S (n,_,_),_)) = Eqobj.unpack p in n end;
   fun slide p n =let val (_,(S (_,s,_),_)) = Eqobj.unpack p in s n end;
   fun dist p q = let val (_,(S (_,_,d),_)) = Eqobj.unpack p in d q end;
(* packing *)
   fun unpack p = explode (Eqobj.unpack p);
   fun repack f p = implode (fn S m => S (f m)) (Eqobj.unpack p);
   fun pack n ext self =
      let fun mslide k = self (n + k) () ()
          fun mdist q = real (pos q - n)
          fun peq eqeq q = n = pos q
      in Eqobj.repack peq (Eqobj.pack () (S (n,mslide,mdist), ext) (self n))
      end;
(* new *)
   fun new n = let fun self n () () = pack n % self in self n () () end;
end;

(* colored points  ----------------------------------------------------*)

signature CPOINT = sig
   type 'a ext
   type 'a subself = 'a ext Point.subself
   type self = closed subself
   val pack : color * int -> 'a -> (color -> int -> unit -> unit -> 'a subself) -> 'a subself
   val unpack : 'a subself -> ('a -> 'a subself) * 'a
   val repack : (color * (color -> 'a subself) -> color * (color -> 'a subself)) ->
```

```
                'a subself -> 'a subself
   val col : 'a subself -> color
   val paint : 'a subself -> color -> 'a subself
   val new : int * color -> self
end;

structure Cpoint :> CPOINT = struct
(* types *)
   datatype 'a state = S of
          color *                               (* col *)
          (color -> 'a subself)                 (* paint *)
   withtype 'a ext = 'a state * 'a
   and 'a subself = 'a ext Point.subself;
   type self = closed subself;
(* messages *)
   fun col p = let val (_,(S (c,_),_)) = Point.unpack p in c end;
   fun paint p c =let val (_,(S (_,t),_)) = Point.unpack p in t c end;
(* packing *)
   fun unpack p = explode (Point.unpack p);
   fun repack f p = implode (fn S m => S (f m)) (Point.unpack p);
   fun pack (c,n) ext self =
      let fun mpaint c' = self c' n () ()
          fun cpeq peq q = peq q andalso (c = col q)
      in Eqobj.repack cpeq (Point.pack n (S (c,mpaint), ext) (self c))
      end;
(* new *)
   fun new (n,c) = let fun self c n () () = pack (c,n) % selfin self c n () () end;
end;

(* 2d points  --------------------------------------------------------*)

signature D2POINT = sig
   type 'a ext
   type 'a subself = 'a ext Point.subself
   type self = closed subself
   val pack   : int * int -> 'a -> (int -> int -> unit -> unit -> 'a subself) -> 'a subself
   val unpack : 'a subself -> ('a -> 'a subself) * 'a
   val repack : (int -> int) -> 'a subself -> 'a subself
   val xpos : 'a subself -> int
   val ypos : 'a subself -> int
   val new : int * int -> self
end;

structure D2point :> D2POINT = struct
(* types *)
   datatype state = S of int                   (* ypos  *)
   withtype 'a ext = state * 'a
   and 'a subself = 'a ext Point.subself;
   type self = closed subself;
(* messages *)
   val xpos = Point.pos;                       (* alias *)
   fun ypos p = let val (_,(S n,_)) = Point.unpack p in n end;
   (* packing *)
   fun unpack p = explode (Point.unpack p);
   fun repack f p = implode (fn S m => S (f m)) (Point.unpack p);
   fun pack (y,x) (ext:'a) self =
```

```
        let val mypos = y
            fun mslide pslide k = self (y+k) (x+k) () ()
            fun d2dist pdist q = sqrt (sq (real (ypos q - y)) + sq (pdist q))
            fun d2peq peq q = peq q andalso (y = ypos q)
        in Eqobj.repack d2peq
                (Point.repack (fn (p,s,d) => (p,mslide s,d2dist d))
                      (Point.pack x (S mypos, ext) (self y)))
        end;
(* new *)
   fun new (x,y) = let fun self y x () () = pack (y,x) % self in self y x () () end;
end;

(* line -------------------------------------------------------------- *)

signature LINE = sig
   type ('a,'b) ext
   type ('a,'b) subself = ('a,'b) ext Eqobj.subself
   type 'a self = ('a, closed) subself
   val pack   : 'a Point.subself * 'a Point.subself -> 'b ->
               ('a Point.subself * 'a Point.subself -> unit -> unit -> ('a,'b) subself) ->
               ('a,'b) subself
   val unpack : ('a,'b) subself -> ('b -> ('a,'b) subself) * 'b
   val repack : ('a Point.subself * 'a Point.subself * (int -> ('a,'b) subself) * (unit -> real) ->
                'a Point.subself * 'a Point.subself * (int -> ('a,'b) subself) * (unit -> real)) ->
               ('a,'b) subself -> ('a,'b) subself
   val from : ('a,'b) subself -> 'a Point.subself
   val to : ('a,'b) subself -> 'a Point.subself
   val shift : ('a,'b) subself -> int -> ('a,'b) subself
   val length : ('a,'b) subself -> real
   val new : 'a Point.subself * 'a Point.subself -> 'a self
end;

structure Line :> LINE = struct
(* types *)
   datatype ('a,'b) state = S of
          'a Point.subself *                       (* from *)
          'a Point.subself *                       (* to   *)
          (int -> ('a,'b) subself)*(* shift *)
          (unit -> real)                (* length *)
   withtype ('a,'b) ext = ('a,'b) state * 'b
   and ('a,'b) subself = ('a,'b) ext Eqobj.subself;
   type 'a self = ('a,closed) subself;
(* messages *)
   fun from l = let val (_,(S (f,_,_,_),_)) = Eqobj.unpack l in f end;
   fun to l = let val (_,(S (_,t,_,_),_)) = Eqobj.unpack l in t end;
   fun shift l n =let val (_,(S (_,_,s,_),_)) = Eqobj.unpack l in s n end;
   fun length l = let val (_,(S (_,_,_,le),_)) = Eqobj.unpack l in le () end;
(* packing *)
   fun unpack p = explode (Eqobj.unpack p);
   fun repack f p = implode (fn S m => S (f m)) (Eqobj.unpack p);
   fun pack (f,t) ext self =
      let val mfr = f
          val mto = t
          fun msh n = self (Point.slide f n, Point.slide t n) () ()
          fun mlen () = Point.dist f t
          fun leq eqeq q = Eqobj.eq f (from q) andalso Eqobj.eq t (to q)
```

```
      in Eqobj.repack leq (Eqobj.pack () (S (mfr,mto,msh,mlen),ext) (self (f,t)))
      end;
(* new *)
   fun new (f : 'a Point.subself, t : 'a Point.subself) : 'a self =
      let fun self ft () () = pack ft % self in self (f,t) () ()end;
end;

(* lines are polymorphic and dynamic dispatch is provided, so a single line class is needed *)

structure PointLine = Line;
```

A2.4.2 Structures, multiple dynamic dispatch, subtyping

```
(* include subprelude and run with SubML *)

open Closed;
open Wrapped;

(* root objects -------------------------------------------------------*)

signature ROOT = sig
   type 'a subself
   type self = closed subself
   type wrap = wrapped option
   val pack : unit -> wrap option ref -> 'a -> (unit -> 'a subself) -> 'a subself
   val unpack : 'a subself -> ('a -> 'a subself) * 'a
   val dowrap : wrap -> self -> wrap
   val unwrap : wrap -> wrap
   val wrap : 'a subself -> wrap
   val new : unit -> self
end;

structure Root :> ROOT = struct
(* types *)
   datatype state = S of wrap option ref                (* wrap   *)
   withtype 'a subself = state * 'a
   and self = closed subself
   and wrap = wrapped option;
(* wrapping *)
   val (Self,matches) = create (None : (self * wrap) option);
   fun dowrap w p = Some (Self (p,w));
   fun unwrap (w as Some x) = case matches x of Some (_,w) => w | _ => w | unwrap w = w;
(* messages *)
   fun wrap (S (ref (Some w)),_) = w;
(* packing *)
   fun unpack wob = explode (I,wob);
   fun pack () wrapped ob self = (S wrapped, ob);
(* new *)
   fun new () =
      let fun self () =
              let val r = ref None
                  val p = pack () r % self
              in r := Some (dowrap None p); p
              end;
      in self ()
      end;
end;

(* eqobjs ------------------------------------------------------------ *)

signature EQOBJ = sig
   type 'a ext
   type 'a subself = 'a ext Root.subself
   type self = closed subself
   val pack : unit -> Root.wrap option ref -> 'a -> (unit -> unit -> 'a subself) -> 'a subself
   val unpack : 'a subself -> ('a -> 'a subself) * 'a
   val repack : ((Root.wrap -> bool) -> (Root.wrap -> bool)) -> 'a subself -> 'a subself
```

```
   val dowrap : Root.wrap -> self -> Root.wrap
   val unwrap : Root.wrap -> Root.wrap
   exception Eq
   val eq : 'a subself -> 'b subself -> bool
   val new : unit -> self
end;

structure Eqobj :> EQOBJ = struct
(* types *)
   datatype state = S of Root.wrap -> bool              (* eq    *)
   withtype 'a ext = state * 'a
   and 'a subself = 'a ext Root.subself;
   type self = closed subself;
(* wrapping *)
   val (Self,matches) = create (None : (self * Root.wrap) option);
   fun dowrap w p = Root.dowrap (Some (Self (p,w))) p;
   fun unwrap w = case Root.unwrap w of Some x => (case matches x of Some (_,w) => w | _ => w) | w => w;
(* messages *)
   exception Eq;
   fun eq p q = let val (_,(S e,_)) = Root.unpack p in e (Root.wrap q) end;
   (* packing *)
   fun unpack p = explode (Root.unpack p);
   fun repack f p = implode (fn S m => S (f m)) (Root.unpack p);
   fun pack () wrapped ext self =
      let fun eqeq w = raise Eq in Root.pack () wrapped (S eqeq, ext) (self ()) end;
(* new *)
   fun new () =
      let fun self () () =
              let val r = ref None
                  val p = pack () r % self
              in r := Some (dowrap None p); p
              end
      in self () ()
      end;
end;

(* points -------------------------------------------------------------*)

signature POINT = sig
   type 'a ext
   type 'a subself = 'a ext Eqobj.subself
   type self = closed subself
   val pack : int -> Root.wrap option ref -> 'a -> (int -> unit -> unit -> 'a subself) -> 'a subself
   val unpack : 'a subself -> ('a -> 'a subself) * 'a
   val repack : (int * (int -> 'a subself) * (Root.wrap -> real) ->
                 int * (int -> 'a subself) * (Root.wrap -> real)) ->
                'a subself -> 'a subself
   val dowrap : Root.wrap -> self -> Root.wrap
   val unwrap : Root.wrap -> Root.wrap
   val pos : 'a subself -> int
   val slide : 'a subself -> int -> 'a subself
   val dist : 'a subself -> 'b subself -> real
   val new : int -> self
end;

structure Point :> POINT = struct
```

```
(* types *)
   datatype 'a state = S of
          int *                                (* pos *)
          (int -> 'a subself) *                (* slide *)
          (Root.wrap -> real)                  (* dist  *)
   withtype 'a ext = 'a state * 'a
   and 'a subself = 'a ext Eqobj.subself;
   type self = closed subself;
(* wrapping *)
   val (Self,matches) = create (None : (self * Root.wrap) option);
   fun dowrap w p = Eqobj.dowrap (Some (Self (p,w))) p;
   fun unwrap w = case Eqobj.unwrap w of Some x => (case matches x of Some (_,w) => w | _ => w) | w => w;
(* messages *)
   fun pos p = let val (_,(S (n,_,_),_)) = Eqobj.unpack p in n end;
   fun slide p n = let val (_,(S (_,s,_),_)) = Eqobj.unpack p in s n end;
   fun dist p q = let val (_,(S (_,_,d),_)) = Eqobj.unpack p in d (Root.wrap q) end;
(* packing *)
   fun unpack p = explode (Eqobj.unpack p);
   fun repack f p = implode (fn S m => S (f m)) (Eqobj.unpack p);
   fun pack n wrapped ext self =
      let fun mslide k = self (n + k) () ()
          fun mdist Q =
              let val Some x = Eqobj.unwrap Q
                  val Some (q,_) = matches x
              in Eqobj.unwrap Q in real (pos q - n)
              end
          fun peq eqeq Q = case Eqobj.unwrap Q of
                Some x => (case matches x of Some (q,_) => n = pos q | _ => eqeq Q)
              | _ => eqeq Q
      in Eqobj.repack peq (Eqobj.pack () wrapped (S (n,mslide,mdist), ext) (self n))
      end;
(* new *)
   fun new n =
      let fun self n () () =
              let val r = ref None
                  val p = pack n r % self
              in r := Some (dowrap None p); p
              end
      in self n () ()
      end;
end;

(* colored points  ---------------------------------------------------*)

signature CPOINT = sig
   type 'a ext
   type 'a subself = 'a ext Point.subself
   type self = closed subself
   val pack : color * int -> Root.wrap option ref ->
             'a -> (color -> int -> unit -> unit -> 'a subself) -> 'a subself
   val unpack : 'a subself -> ('a -> 'a subself) * 'a
   val repack : (color * (color -> 'a subself) -> color * (color -> 'a subself)) ->
               'a subself -> 'a subself
   val dowrap : Root.wrap -> self -> Root.wrap
   val unwrap : Root.wrap -> Root.wrap
   val col : 'a subself -> color
```

```
   val paint : 'a subself -> color -> 'a subself
   val new : int * color -> self
end;

structure Cpoint :> CPOINT = struct
(* types *)
   datatype 'a state = S of
          color *                                   (* col *)
          (color -> 'a subself)                     (* paint *)
   withtype 'a ext = 'a state * 'a
   and 'a subself = 'a ext Point.subself;
   type self = closed subself;
(* wrapping *)
   val (Self,matches) = create (None : (self * Root.wrap) option);
   fun dowrap w p = Point.dowrap (Some (Self (p,w))) p;
   fun unwrap w = case Point.unwrap w of Some x => (case matches x of Some (_,w) => w | _ => w) | w => w;
(* messages *)
   fun col p = let val (_,(S (c,_),_)) = Point.unpack p in c end;
   fun paint p c = let val (_,(S (_,t),_)) = Point.unpack p in t c end;
(* packing *)
   fun unpack p = explode (Point.unpack p);
   fun repack f p = implode (fn S m => S (f m)) (Point.unpack p);
   fun pack (c,n) wrapped ext self =
      let fun mpaint c' = self c' n () ()
          fun cpeq peq Q = case Point.unwrap Q of
                  Some x => (case matches x of Some (q,_) => peq Q andalso (c = col q) | _ => peq Q)
                | _ => _ => peq Q
      in Eqobj.repack cpeq (Point.pack n wrapped (S (c,mpaint), ext) (self c))
      end;
(* new *)
   fun new (n,c) =
      let fun self c n () () =
              let val r = ref None
                  val p = pack (c,n) r % self
              in r := Some (dowrap None p); p
              end
      in self c n () ()
      end;
end;

(* 2d points  --------------------------------------------------------*)

signature D2POINT = sig
   type 'a ext
   type 'a subself = 'a ext Point.subself
   type self = closed subself
   val pack   : int * int -> Root.wrap option ref ->
               'a -> (int -> int -> unit -> unit -> 'a subself) -> 'a subself
   val unpack : 'a subself -> ('a -> 'a subself) * 'a
   val repack : (int -> int) -> 'a subself -> 'a subself
   val dowrap : Root.wrap -> self -> Root.wrap
   val unwrap : Root.wrap -> Root.wrap
   val xpos : 'a subself -> int
   val ypos : 'a subself -> int
   val new : int * int -> self
end;
```

```
structure D2point :> D2POINT = struct
(* types *)
   datatype state = S of int                    (* ypos  *)
   withtype 'a ext = state * 'a
   and 'a subself = 'a ext Point.subself;
   type self = closed subself;
(* wrapping *)
   val (Self,matches) = create (None : (self * Root.wrap) option);
   fun dowrap w p = Point.dowrap (Some (Self (p,w))) p;
   fun unwrap w = case Point.unwrap w of Some x => (case matches x of Some (_,w) => w | _ => w) | w => w;
(* messages *)
   val xpos = Point.pos;                        (* alias *)
   fun ypos p = let val (_,(S n,_)) = Point.unpack p in n end;
(* packing *)
   fun unpack p = explode (Point.unpack p);
   fun repack f p = implode (fn S m => S (f m)) (Point.unpack p);
   fun pack (y,x) wrapped ext self =
      let val mypos = y
          fun mslide pslide k = self (y+k) (x+k) () ()
          fun d2dist pdist Q = case Point.unwrap Q of
                Some x => (case matches x of
                        Some (q,_) => sqrt (Real.+ (sq (real (ypos q - y)), sq (pdist Q)))
                       | _ => pdist Q)
               | _ => pdist Q
          fun d2peq peq Q = case Point.unwrap Q of
                Some x => (case matches x of Some (q,_) => peq Q andalso (y = ypos q) | _ => peq Q)
               | _ => peq Q
      in Eqobj.repack d2peq
              (Point.repack (fn (p,s,d) => (p,mslide s,d2dist d))
                   (Point.pack x wrapped (S mypos, ext) (self y)))
      end;
(* new *)
   fun new (x,y) =
      let fun self y x () () =
              let val r = ref None
                  val p = pack (y,x) r % self
              in r := Some (dowrap None p); p
              end
      in self y x () ()
      end;
end;

(* line --------------------------------------------------------------- *)

signature LINE = sig
   type ('a,'b) ext
   type ('a,'b) subself = ('a,'b) ext Eqobj.subself
   type 'a self = ('a,closed) subself
   val pack   : 'a Point.subself * 'a Point.subself -> Root.wrap option ref -> 'b ->
               ('a Point.subself * 'a Point.subself -> unit -> unit -> ('a,'b) subself) ->
               ('a,'b) subself
   val unpack : ('a,'b) subself -> ('b -> ('a,'b) subself) * 'b
   val repack : ('a Point.subself * 'a Point.subself * (int -> ('a,'b) subself) * (unit -> real) ->
                'a Point.subself * 'a Point.subself * (int -> ('a,'b) subself) * (unit -> real)) ->
               ('a,'b) subself -> ('a,'b) subself
```

```
   val dowrap : Root.wrap -> 'a self -> Root.wrap
   val unwrap : Root.wrap -> Root.wrap
   val from : ('a,'b) subself -> 'a Point.subself
   val to : ('a,'b) subself -> 'a Point.subself
   val shift : ('a,'b) subself -> int -> ('a,'b) subself
   val length : ('a,'b) subself -> real
   val new : 'a Point.subself * 'a Point.subself -> 'a self
end;

structure Line :> LINE = struct
(* types *)
   datatype ('a,'b) state = S of
          'a Point.subself *                        (* from *)
          'a Point.subself *                        (* to   *)
          (int -> ('a,'b) subself) *                (* shift *)
          (unit -> real)                            (* length *)
   withtype ('a,'b) ext = ('a,'b) state * 'b
   and ('a,'b) subself = ('a,'b) ext Eqobj.subself;
   type 'a self = ('a,closed) subself;
(* wrapping *)
   val (Self,matches) = create (None : (self * Root.wrap) option);
   fun dowrap w p = Eqobj.dowrap (Some (Self (p,w))) p;
   fun unwrap w = case Eqobj.unwrap w of Some x => (case matches x of Some (_,w) => w | _ => w) | w => w;
(* messages *)
   fun from l = let val (_,(S (f,_,_,_),_)) = Eqobj.unpack l in f end;
   fun to l = let val (_,(S (_,t,_,_),_)) = Eqobj.unpack l in t end;
   fun shift l n = let val (_,(S (_,_,s,_),_)) = Eqobj.unpack l in s n end;
   fun length l = let val (_,(S (_,_,_,le),_)) = Eqobj.unpack l in le () end;
(* packing *)
   fun unpack p = explode (Eqobj.unpack p);
   fun repack f p = implode (fn S m => S (f m)) (Eqobj.unpack p);
   fun pack (f,t) wrapped ext self =
      let val mfr = f
          val mto = t
          fun msh n = self (Point.slide f n, Point.slide t n) () ()
          fun mlen () = Point.dist f t
          fun leq eqeq Q = case Eqobj.unwrap Q of
                Some x => (case matches x of
                      Some (q, _) => Eqobj.eq f (from q) andalso Eqobj.eq t (to q) | _ => eqeq Q)
              | _ => eqeq Q
      in Eqobj.repack leq (Eqobj.pack () wrapped (S (mfr,mto,msh,mlen),ext) (self (f,t)))
      end;
(* new *)
   fun new (f,t) =
      let fun self ft () () =
              let val r = ref None
                  val l = pack ft r % self
              in r := Some (dowrap None l); l
              end
      in self (f,t) () ()
      end;
end;

(* lines are polymorphic and dynamic dispatch is provided, so a single line class is needed *)

structure PointLine = Line;
```

A2.5 Sample evaluations

A2.5.1 Basic

```
(* for use with encoding 232 to 242 *)

(* binds messages at top level: *)

open Root;
open Eqobj;
open Point;
open Cpoint;
open D2point;
open PointLine;

(* some objects: *)

val p = Point.new 456;
val p' = Point.new 678;
val cp = Cpoint.new (456,red);
val cp' = Cpoint.new (678,green);
val cp'' = Cpoint.new (678,blue);
val dp = D2point.new (123,456);
val dp' = D2point.new (312,724);
val l = PointLine.new (p,p');
val l' = PointLine.new (dp,dp');(* ill-typed if PointLine monomorphic *)
val l'' = PointLine.new (p,dp);(* ill-typed *)

(* using methods: *)

pos p;
pos cp;
pos dp;

col p;     (* ill-typed *)
col cp;
col dp;    (* ill-typed *)

pos (paint (slide cp 5) green) = pos (slide (paint cp green) 5);
col (paint (slide cp 5) green) = col (slide (paint cp green) 5);

length l;
length l';

(* binary methods must be homogeneous: *)

eq p p';
eq cp cp';
eq dp dp';
eq p p';
eq dp dp';
eq l (shift l 10);

eq p cp;   (* ill-typed *)
eq cp p;   (* ill-typed *)
eq dp l;   (* ill-typed *)
```

```
eq l l';   (* ill-typed *)

dist p p';
dist dp dp';

dist p dp';(* ill-typed *)

(* exercising polymorphism: *)

fun map f [] = [] | map f (h::t) = f h :: map f t;
fun positions l = map pos l;
positions [p,p'];
positions [cp,cp'];
positions [dp,dp'];

positions [p,cp,dp];(* ill-typed *)

fun f1 p = 2 * pos p;
f1 p;
f1 cp;
f1 dp;

fun f3 p = (col p, ypos p);(* ill-typed *)

fun f4 (p:Point.self) = slide p (2 * pos p);
f4 p;
f4 cp;        (* ill-typed *)
f4 dp;        (* ill-typed *)

fun f5 p q =(* ill-typed *)
    if red = col p andalso 7 = ypos q
    then p else q;

fun breakit p =
    let val q = Point.new 123 in eq p q end;
breakit p;
breakit cp;(* ill-typed *)

fun f6 p = (* ill-typed *)
   let val q = Point.new 123
   in if eq p q then col p else blue
   end;
```

A2.5.2 Folding and multiple dispatch

```
(* for use with encoding 233 *)

(* binary methods are now fully polymorphic: *)

eq p l;(* raises Eq *)
eq l p;(* raises Eq *)

eq p p';
eq p cp;
eq p dp;
eq cp p;
```

```
eq cp cp';
eq cp dp;
eq dp p;
eq dp cp;
eq dp dp';
eq l (shift l 10);

dist p p';
dist p dp';
dist dp' p;
dist dp dp';

(* use of coercions: *)

fun point'cpoint p = (Point.self'wrap o wrap) p;
fun cpoint'point p = (Cpoint.self'wrap o wrap) p;
fun point'd2point p = (Point.self'wrap o wrap) p;
fun d2point'point p = (D2point.self'wrap o wrap) p;

positions [p,point'cpoint cp];
positions [p,point'd2point dp];
positions [point'cpoint cp,point'd2point dp];
positions [p,point'cpoint cp,point'd2point dp];

f4 (point'cpoint cp);
f4 (point'd2point dp);

fun f6 p =
  let val q = Point.new 123
  in if eq p q then col p else blue
  end;
f6 p;      (* ill-typed *)
f6 (cpoint'point p);(* raises Wrap *)
```

A2.5.3 Versions with subtyping

```
(* for use with encoding 242 *)

val l'' = PointLine.new (p,dp);
length l'';

positions [p,cp];
positions [p,dp];
positions [cp,dp];
positions [p,cp,dp];

f4 p;
f4 cp;
f4 dp;

fun f5 p q =
    if red = col p andalso 7 = ypos q
    then p else q;
f5 p dp;   (* ill-typed *)
val x = f5 cp dp;(* well-typed *)
pos x;
```

```
col x;         (* ill-typed *)
   (* ability to apply pos to x is kept, but not those
      of applying col or ypos *)

f6 p;      (* ill-typed *)
f6 cp;

breakit cp;
```