Wide classes

Manuel Serrano
Manuel.Serrano@unice.fr
http://kaolin.unice.fr/~serrano

Université de Nice Sophia-Antipolis
650, route des Colles, B.P. 145 F-06903 Sophia-Antipolis, CEDEX

Abstract. This paper introduces the concepts of *wide classes* and *widening* as extensions to the object model of class-based languages such as Java and Smalltalk. Widening allows an object to be temporarily *widened*, that is transformed into an instance of a subclass, a *wide class*, and, later on, to be *shrunk*, that is reshaped to its original class. Wide classes share the main properties of plain classes: they have a name, a superclass, they may be instantiated, they have an associated class predicate and an associated type that may be used to override function definitions.
Widening is also useful to implement transient data storage for long-lasting computations. In particular, it helps reducing *software data retention*. This phenomenon arises when the actual data structures used in a program fail to reflect time-dependent properties of values and can cause excessive memory consumption during the execution.
Wide classes may be implemented for any dynamically-typed class-based programming language with very few modifications to the existing run-time system. We describe the simple and efficient implementation strategy used in the Bigloo runtime system.

Keywords: language implementation, dynamic inheritance, dynamic type checking, instance modification.

1 Introduction

Few object-oriented programming languages permit objects to change their nature during execution. If an object `obj` is an instance of a class $\mathcal{C}$ at some point of the execution, then `obj` remains an instance of $\mathcal{C}$ for the whole computation. In other words, the *type* of an object may not evolve over time. Most Object-oriented languages only provide a mechanism to associate behaviors to types by the means of *methods*. In consequence, object-oriented programming languages do not allow the behavior to evolve over time. For instance, consider window objects in an object-oriented window manager. We may imagine several kind of windows such as windows with a menu bar and windows without menu bar. This can be easily implemented in a class-based object-oriented language. We may build a bare window class and a subclass for windows with menu bar. This is easy because if a window has a menu bar, it will keep it for its entire life time.

The trouble comes when one wants to express that a window is either visible or iconified. A window may be visible at some point and iconified at another point. The property of being visible or iconified is dynamic; it changes over time. In consequence, the behavior associated to the window should also change over time. For instance, refreshing a visible window or refreshing an iconified window requires different operations. In general, object-oriented languages do not provide mechanisms that allow a straightforward representation of this phenomenon.

Wide classes extend the standard object-oriented programming model with a disciplined type conversion operation called *widening*. This allows an object to be temporarily *widened* (that is transformed into an object of a subclass, a *wide class*) and then to be *shrunk* (that is reshaped to its original class). Widening is a runtime operation. During the execution, a program explicitly widens and shrinks objects. Widening and shrinking are tools of choice to implement transient states. In our window manager example, we may widen a visible window to an instance of the iconified window class when the program requests iconification and we may shrink the iconified window to a plain visible window when the program requests de-iconification. Widening and shrinking change the type of the window from visible to iconified, so we may write methods that will apply to visible windows and methods that will apply to iconified windows. Furthermore widening may add data attributes needed by the new behavior while shrinking sheds data that is not needed anymore.

Both Smalltalk and Clos propose a construction that allows unrestricted changes to the runtime type of objects. These type-change operations may turn any object into an instance of any class. Widening is more disciplined as type-changes are restricted to (wide) subclasses. In other words, widening preserves subtyping relationships (after widening the new runtime type of an object is a subtype of its former runtime type). This limitation is important for program correctness and essential for obtaining an efficient implementation of wide classes. We show in this paper that wide classes have a simple and fast implementation with two important properties:

- Wide classes have no impact on the implementation of the rest of the runtime system.
- A program that does not make use of widening does not pay any extra cost for that feature.

Neither of these properties hold for the equivalent mechanisms offered in Smalltalk and Clos.

From a programmer's point of view, widening is very useful to implement transient data storage and thus helps facing the problem of *software data retention*.

1.1 Software data retention

During the execution of a program, it might happen that some data items have a limited life time but that the actual data structure in which they are contained is too rigid and will waste memory because it will retain space longer than necessary. Softwares that use a *pipelined* architecture, that is softwares that cascade several distinct stages, must frequently face these retention problems.

As an example, let's suppose that in a compiler one would like to introduce a stage that removes useless variables. For the purpose of that stage, the number of occurrences of each variable is needed and we suppose that this information is never needed elsewhere. The problem is "where to store the number of occurrences in the abstract syntax tree". A data structure allocating an extra field holding the number of occurrences for each node that implements variables wastes memory as soon as the stage is completed. (Of course, many stages of a compiler use more than one temporary information.) We say that a program making use of such data structure faces *data retention*.

1.2 Fighting software data retention

Current programming languages do not propose mechanisms to reduce software data retention and thus programmers are used to one of the following *ad-hoc* strategies.

1. *Global data*: this strategy consists in allocating a global data structure that provides room for all the data of all the stages of the execution. This solution is inefficient in space because the structure is much larger than the actual data that it contains. Many fields of the data structure will be empty at any given time. Moreover with a programming language that enables automatic memory management, the program must take provisions to explicitly free the unused fields; otherwise the algorithm to reclaim memory will fail at re-using the memory that has been allocated for a completed stage. In other words, the program must explicitly free the dead data structures; this greatly minimizes the pros of automatic memory management.
   Further, the *global data* approach fails to enforce strict barriers between stages, and since the information from the previous stages is still available there is a temptation of using it. This is in contradiction to the principles of clear separation between stages of the pipelined architecture.
2. *Copy on entrance*: when entering a new stage the global data structure is copied into a new one that provides rooms for the local data that belongs to that stage. This solution is safe to avoid confusion between distinct stages. It is efficient in space because the size of the largest structure live at a time is the size of the largest structure allocated by each stage. Unfortunately, this framework is inefficient in execution time as it implies a large amount of memory allocation/deallocation. Thus, it is likely that the time spent in the memory manager will be significant. Moreover the time required to copy the structure is important because sharing properties must be preserved from one copy to another.
3. *User field:* with this strategy, a so-called *user field* is added to each structure of the global data type and then each stage hooks its own local structures using this extra field. This solution is frequently used: for example, the C

structure `fieldnode` of the `form` library declares a field `usrptr` that can be used by the clients of that library. If this technique has the advantage of being space efficient (only one word is wasted by structure to hold the extra field) it has two severe disadvantages:

- It forfeits static type checking. The type of the user field has to be the type of the most generic user type (e.g., `void *` in C). As a consequence, the *user field* framework leads to a very common error. The stage S_i reads the user field that is still holding a local data of the stage S_{i-1}. Then an error (or an exception for languages with higher level runtime support) is raised and the execution stops.
- More pragmatically, it is simply inconvenient to use, as each access to the user slot requires an extra indirection (and possibly an extra coercion for statically type checked languages). This all makes the code dealing with user fields unnecessarily verbose.

Wide classes and widening represent a solution to software data retention. Objects may be widened to hold transient data and shrunk when that data appears to be useless. Wide classes use inheritance. That is, widening affects the dynamic binding of methods. Widening is used extensively in the Bigloo compiler for the Scheme programming language. Bigloo compilation passes are implemented using the *widening/shrinking* mechanism. Upon entrance to a pass, objects are widened with pass-specific fields and, on pass exit, they are shrunk back in order to forget the information related to that pass. Our experience has been that using widening has greatly simplified the organization of the code of the compiler; from a software engineering point of view this has been beneficial, and from an efficiency point of view the current release of the compiler uses memory more sparingly than its predecessors.

1.3 Organization

Section 2 presents a general overview of the Bigloo object model. Readers familiar with the Clos-like style may skip that section. Section 3 introduces the design of wide classes. The new language constructions are presented here. Examples of small programs using widening are given. This section concludes with a discussion of the limitations of wide classes. Section 4 presents widening used in the context of a large program, our Scheme compiler, Bigloo. Section 5 discusses the implementation of wide classes and widening operations. Section 6 compares wide classes with other object oriented features. Section 7 presents some possible extensions to widening and wide classes.

2 Bigloo

Bigloo is an open implementation of the Scheme language. From the beginning, the aim was to propose a realistic and pragmatic alternative to the strict Scheme implementation defined by [7]. Bigloo does not implement "all" of Scheme: for example, the execution of tail-recursions may allocate memory and very few arithmetics are implemented (Bigloo only supports integers and reals). However, Bigloo implements numerous extensions: support for lexical and syntactic analysis, pattern matching, an exception mechanism, a foreign interface and a module language. The recent Bigloo versions offers a new extension with an object layer, to a great extent inspired by Meroon [8] of C. Queinnec.

The term "object-oriented" is a rather loose fitting description that has been prefixed to a number of very different languages and systems over the years. We will restrict our attention to two object models: the Smalltalk model [5] where methods are associated to classes and the Clos model [1] where methods are associated to generic functions. Each of those models has many incarnations: with static or dynamic type checking, with single or multiple inheritance, and single or multiple dispatch. In this paper we assume a Clos-like object model with single inheritance and single dispatch.

To introduce the Clos object model, let us present a small example. Consider a class *point* implementing 2 dimensional points and a class *point-3d* implementing 3-d points and suppose that we want to implement a function that computes the norm of these points. Instead of placing methods in the declaration of *point* and *point-3d* that implement this functionality as we would do in Smalltalk or Java, we define a *generic function* that will apply to the *point* and *point-3d* classes:

```
(define-generic (norm::real p::point))
```

In a first step a generic function may be considered as a declaration. No user code is associated to it. A generic function is a placeholder for method definitions. The above declaration of `norm` specifies that methods called `norm` can be defined, taking exactly one argument which is a subtype of *point*. Methods are said to *override* the generic function.

```
(define-method (norm::real p::point)
   (sqrt (sqr (point-x p)) (sqr (point-y p))))

(define-method (norm::real p::point-3d)
   (sqrt (sqr (point-3d-x p)) (sqr (point-3d-y p)) (sqr (point-3d-z p))))
```

The functions `point-x`, `point-y`, `point-3d-x`, ... are accessors, automatically generated by the compiler. Methods can be used as:

```
(let ((p   (instantiate::point (x 3.3) (y 4.1)))
      (p3d (instantiate::point (x 8.9) (y 4) (z 2))))
   (print (+ (norm p) (norm p3d))))
```

The Clos model fits well with the traditional functional programming style. A function can be thought as a generic function overridden with exactly one method. Another reason to choose the Clos model is the nature and the architecture of the Bigloo compiler that we wanted to rewrite with objects.

Former versions of the Bigloo compiler were written in plain Scheme. The compiler is a program of 40,000 code lines. It reads a program to be compiled and builds the abstract syntax tree used to represent the program. This tree contains a structure of 23 different node types. There is a node for constants, variable assignments, conditionals, function calls, etc. The compiler is made up of stages, each of which can be seen as a process that modifies the abstract syntax tree. The driver is a Scheme function that looks like the following one:

```
(define (compiler src)
   (let ((ast (build-ast src)))
      (macro-expand! ast)         ;; 1st stage
      (function-inline! ast)      ;; 2nd stage
      ...
      (code-generate! ast)))      ;; 20th stage
```

Each stage is implemented in a single file, except for the most complex ones that can be split over various files. For the new releases making use of object-oriented programming, we wanted to keep this decomposition as we felt it was the most natural way to write pipelined systems.

This encouraged us to adopt a generic function object model rather than the traditional Smalltalk model as this last model would have forced us to give up the stage-driven structure in favor of a structure driven by the abstract syntax tree. Each compiler stage would have been split in all classes implementing the tree nodes. A file, implementing the `function-inline!` stage for example, would not exist anymore and its code would have been split among classes.

Adding new nodes to the syntax tree, however, would have been easier for our compiler with the Smalltalk model. Using this model, adding a stage requires to modify and to recompile all the existing code (because we need to add the methods defining the new stage for each class). But, adding a node does not require changing the existing code. Adding a node means that we are adding a feature to the target language (which is uncommon), while adding a stage usually means that we are adding a compiler optimization (very common for compiler writers). Our experience has shown that it is important to ease the task of writing new stages as these have a local impact, while language additions are difficult anyway because they may impact many features.

2.1 Class declarations

Classes can be declared static or exported. It is then possible to make a declaration accessible from another module or to limit its scope to one module. The abbreviated syntax of a class declaration is:

```
(class class-id::super-class-id <field>*)

<field> ::= field-id::type-id
  | (field-id::type-id <option>*)
  | (* field-id::type-id <option>*)

<option> ::= read-only
  | (default value)
```

A class can inherit from a single super class. Classes with no specified super class inherit from the `object` class. The type associated with a subclass is a subtype of the type of the super class.

A field is typed (with the annotation `::`*`type-id`*), might be immutable (`read-only`), and may have a default value (`default`). Fields with clauses that begin with `*` are said to be indexed: those fields do not contain a single value but a sequence of values. Indexed fields (directly inspired by MEROON) can be used, for example, to implement strings of characters. Here are possible declarations for the traditionals `point` and `point-3d`:

```
(module small-points
   (export (class point
              (x::double read-only (default 0.0))
              (y::double read-only (default 0.0)))
           (class point-3d::point
              (z::double read-only (default 0.0)))))
```

2.2 Generic functions

Generic function declarations are function declarations annotated by the `generic` keyword. They can be exported: this means that they can be used from within other modules and that methods can be added to those functions from other modules. They can be static, that is not accessible from within other modules. In this case, no method can be added to the ones introduced by the module that defines the generic function. The syntax to define a generic function is very similar to an usual function definition:

```
(define-generic (fun-id::type-id arg-id::class-id ...) optional-body)
```

Generic functions should have at least one argument as it is the first argument that is used to solve the *dynamic dispatch* of methods. This argument is of type T that must be associated to a class. It is impossible to override generic functions with methods whose first argument is not of a subtype of T. Generic functions can have a body. This body will be evaluated if, when calling a generic function, no method is defined for the type of the first argument. If this situation occurs and the generic function has no body, an error is raised. Here is an example of a generic function definition whose distinguishing argument must be a subtype of `point`:

```
(define-generic (display-point::obj p::point)
   (print "<???>"))
```

2.3 Methods

Methods are declared by the following syntactic form:

```
(define-method (fun-id::type-id arg-id::class-id ...) body)
```

Defining a method if there is no defined generic function with a compatible prototype (arguments and result types in a sub-typing relation with those of the generic function) is an error. A method can be written as follows:

```
(define-method (display-point::obj p::point)
   (with-access::point p (x y)
      (print "<point:" x ", " y ">")))
```

Methods override generic function definitions. When executing the following code:

```
(let ((p (instantiate::point)))
   (display-point p))
```

because there is a method defined over the class *point* the code of that method will be executed instead of the default code of the generic function.

2.4 Instances

When declaring a class *cla*, Bigloo automatically generates the predicate `cla?`, an allocator `instantiate::cla`, a cloner `duplicate::cla`, accessors (e.g., `cla-x` for a field `x`), modifiers (e.g., `cla-x-set!` for a field `x`), and, an abbreviated access form, `with-access::cla`, to allow accessing and writing fields by referencing them by their name. Here is how to allocate and access an instance:

```
(let ((p (instantiate::point-3d (y -3.4) (x 1.0))))
   ;; The initialization value of a field can be omitted from the arguments list if the
   ;; field has a default value; this is the case for the field z of class point-3d.
   (with-access::point-3d p (x y z)
      (sqrt (+ (sqr x) (sqr y) (sqr z)))))
```

3 Wide classes

The key point to *wide classes* is that objects may become (may be widened to) instances of wide classes and thus that the type of an object may change dynamically over the execution.

3.1 Wide class declarations

The declarations of *wide classes* are similar to the declarations of plain classes:

```
(wide-class class-id::super-class-id <field>*)
```

The super class of a wide class may be a plain class or a wide class. By contrast the super class of a plain class cannot be a wide class. This point will be discussed in Section 3.6. As an example, here are the declarations of "named points" and "named 3-d points".

```
(module named-point
   (export (wide-class named-point::point
              name::string)
           (wide-class named-point-3d::named-point
              (z::double (default 0.0)))))
```

3.2 Wide instances

Wide objects (i.e. objects that are instances of wide classes) may be used in the exact same way as plain objects. Wide objects are instantiated from wide classes. Wide objects are used to solve generic function run time dispatches and fields of wide objects are fetched from the instances. The types of the wide objects are checked by automatically generated class predicates. Thus, we may write:

```
(define-method (display-point::obj p::named-point)
   (with-access::named-point p (x y name)
      (print "<point(" name "): x:" x ", y:" y ">")))

(define (display-instance x)
   (if (point? x)
       (display-point x)
       (display x)))

(let ((p (instantiate::named-point (name "org") (x 0) (y 0))))
   (display-instance x))
```

3.3 Widening

The new feature introduced by wide classes is that during its life-time, an instance of a given class may be *widened* to an instance of one of its wide subclasses. Later on, the same instance may be *shrunk*, that is reshaped to its original class. The mandatory point here is that the widening and shrinking operations do not introduce copies. A widened instance stays the same, being merely extended with new slots. In addition, if an object `obj` satisfies a class predicate $\mathcal{P}$, then after widening, `obj` *still satisfies* $\mathcal{P}$.

The syntax of the widening operator resembles the syntax of the instance allocator:

```
(widen!::class-id instance (field-id0 val0) (field-id1 val1) ...)
```

`Class-id` is the name of a wide class, the name of the wide class the *instance* is widened to. It is an error if *`class-id`* is the class identifier of a plain class. The *field-id*$_i$ identifiers are the name of the fields of the wide class *class-id*. The values *val*$_i$ are initial expressions for the fields of the wide class. As for object

instantiation, fields declared with default values may be omitted within a `widen!` expression. Here is an example of widening from the class *point* to the class *named-point*.

```
(let ((pt (instantiate::point (x 0.0) (y 5.3))))
   (widen!::named-point pt (name "dummy"))
   (display-point pt))
```

Instantiation: Instantiating a wide object is semantically equivalent to creating a plain object and then applying successive widening until the desired wide class is obtained. That is

```
(instantiate::named-point-3d (x 0) (y 0) (z 0) (name "gee"))
```

is a shorten for:

```
(let ((p (instantiate::point (x 0) (y 0))))
   (widen!::named-point p (name "gee"))
   (widen!::named-point-3d p (z 0))
   p)
```

Identity: The semantics of widening preserves object identity, in Scheme terminology this means that a widened instance is still `eq?` to what it was before widening. Thus,

```
(let* ((p     (instantiate::point))
       (new-p (widen!::named-point p (name "new"))))
   (eq? new-p p))
```

evaluates to true. This feature is very important because it allows some parts of a whole data structure to be widened while preserving sharing relationships. We have tried hard to design widening so as to limit the runtime cost. Section 5 presents an implementation strategy which allows us to get widening at no extra implementation cost.

3.4 Shrinking

Shrinking is simpler. Instances get shrunk one class at a time by the means of the form:

```
(shrink! instance)
```

It shrinks an *instance* by one level from its widest level. Thus,

```
(let ((p (intantiate::point)))
   (widen!::named-point p (name "dummy2"))
   (widen!::named-point-3d p (z 3.0))
   (display-point p)
   (shrink! p)
   (display-point p)
   (shrink! p)
   (display-point p))
```

when executed, prints out:

```
<point(dummy2): x: 0.0 y: 0.0 z: 3.0>
<point(dummy2): x: 0.0 y: 0.0>
<point: x: 0.0 y: 0.0>
```

It is an error to shrink an object that is not a wide object. Wide objects may be tested with the predicate:

```
(wide-object? obj)
```

As an example, here is the code of a polymorphic function that takes any object and returns the same object with all wide layers removed (if there were any):

```
(define (shrink-all! obj)
   (if (wide-object? obj)
       (begin (shrink! obj) (shrink-all! obj))
       obj))
```

In presence of multiple widening, it could be useful to specify a destination class when shrinking. The `shrink!` operator accepts an optional argument that denotes that class. The syntax of this form is:

```
(shrink!::class-id instance)
```

Shrinking with a destination class enables several shrink operations to be automatically performed. Its implementation is close to `shrink-all!`. A restriction exists for *class-id*. It must denote a super class of the actual class of *instance* that is a wide class or the first plain class that *instance* belongs to. The following example is a legal shrink:

```
(let ((p (intantiate::named-point-3d (name "dummy2") (z 3.0))))
   (shrink!::point p)
   (display-point p))
```

3.5 Example

Widening may be used to implement transient object properties. For instance, suppose we would like to model marital status. The class implementing persons would be:

```
(class person
   name::string
   fname::string
   (sex::symbol read-only))
```

As we can see, people are allowed to change their name but not their sex. The identity of a person can be printed as follows:

```
(define-method (object-display p::person)
   (with-access::person p (name fname sex)
      (print "firstname : " fname)
      (print "name      : " name)
      (print "sex       : " sex)
      p))
```

Married men and women are implemented by the means of two wide classes:

```
(wide-class married-man::person
   mate::person)
(wide-class married-woman::person
   maiden-name::string
   mate::person)
```

A married woman's identity is printed by the following method (we suppose an equivalent method definition for married-man):

```
(define-method (object-display p::married-woman)
  (with-access::married-woman p (name fname sex mate)
     (call-next-method)
     (print "married to: " (person-fname mate) " " (person-name mate))
     p))
```

The form `(call-next-method)` invokes the method that would have been used if no method for *married-woman* was overriding the generic function. Here, the `(call-next-method)` expression will act as a call to the method `object-display` defined over the *person* class. A wedding is celebrated using the `get-married!` function:

```
(define (get-married! woman::person man::person)
   (if (not (and (eq? (person-sex woman) 'female)
                 (eq? (person-sex man) 'male)))
       (error "get-married" "Illegal wedding" (cons woman man))
       (let* ((mname (person-name woman))
              (wife  (widen!::married-woman woman
                        (maiden-name mname)
                        (mate man))))
          (person-name-set! wife (person-name man))
          (widen!::married-man man (mate woman)))))
```

We can check if two persons are married by:

```
(define (married? woman::person man::person)
   (and (married-woman? woman)
        (married-man? man)
        (eq? (married-woman-mate woman) man)
        (eq? (married-man-mate man) woman)))
```

We may now study what happens during a wedding:

```
(define *junior* (instantiate::person
                    (name "Jones") (fname "Junior") (sex 'male)))
(define *pamela* (instantiate::person
                    (name "Smith") (fname "Pamela") (sex 'female)))

(define *old-boy-junior* *junior*)
(define *old-girl-pamela* *pamela*)
(get-married! *pamela* *junior*)
```

We may check the wedding:

```
(married? *pamela* *junior*)
   ⇒ #t
(display-object *pamela*)
   ⇒ name       : Jones
      firstname  : Pamela
      sex        : FEMALE
      married to : Junior Jones
```

Note that both `*pamela*` and `*junior*` are still the same persons:

```
(eq? *old-boy-junior* *junior*)  ⇒ #t
(eq? *old-girl-pamela* *pamela*) ⇒ #t
```

Finally, divorce is implemented as:

```
(define (divorce! woman::person man::person)
   (if (not (couple? woman man))
       (error "divorce!" "Illegal divorce" (cons woman man))
       (let ((mname (married-woman-maiden-name woman)))
          (begin
             (shrink! man)
             (shrink! woman)
             (person-name-set! woman mname))
```

And thus:

```
(divorce! *pamela* *junior*)
(display-object *pamela*)
   → name       : Smith
      firstname  : Pamela
      sex        : FEMALE
(eq? *old-boy-junior* *junior*)  ⇒ #t
(eq? *old-girl-pamela* *pamela*) ⇒ #t
```

3.6 Restrictions

We present here several rules that restrict widening and class declarations. We conclude with a description of the features a language must provide to be able to host wide classes.

Widening and plain classes: An object cannot be widened to a plain class. This is not possible because as described in Section 5 accessing fields of wide classes requires different operations than accessing fields of plain classes. We enforce this separation between plain and wide classes for efficiency reasons: we want to be able to generate efficient code for programs which do not use wide classes; this would not be the case if arbitrary widening along the inheritance tree would be allowed.

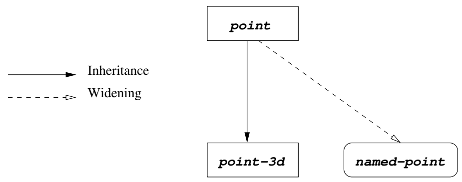
In consequence, wide classes may inherit from plain classes or from wide classes. But plain classes cannot inherit from wide classes. There is no technical reason for that. The only reason is that it is useless to declare a class that inherits from a wide class because no object may be widened to a plain class!

Single subtyping and widening: As we mentioned in Section 2, the object model of the Bigloo's source language is based on single subtyping. That is a class has one single superclass. To be consistent with this choice we have chosen to restrict widening accordingly. Thus, an object can only be widened to a wide class that directly inherits from the current class of the object. In other terms, it is illegal to widen an instance whose actual class is $\mathcal{C}_1$ into an instance of class $\mathcal{C}_2$ if $\mathcal{C}_2$ does not inherit from $\mathcal{C}_1$. Thus

```
(module named-point-3d
   (export
    (class point ...)
    (class point-3d::point ...)
    (wide-class named-point::point ...)))

(let ((p (intantiate::point-3d)))
   (widen!::named-point p ...))
```

is illegal because *named-point* does not inherit from *point-3d* (the actual class of p). If simultaneous widening were permitted for different branches of inheritance, then we could build a tree like the following one:

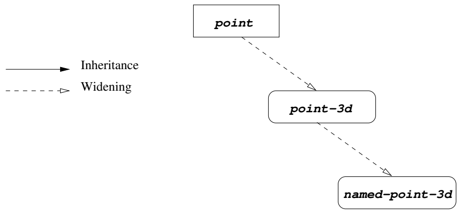


This obviously breaks the single subtyping rule. An object is simultaneously a *point-3d* and a *named-point* and neither the rule *point-3d* $<$ *named-point* nor *point-3d* $>$ *named-point* stands. On the other hand:

```
(module named-point-3d
   (export
    (class point ...)
    (wide-class point-3d::point ...)
    (wide-class named-point-3d::point-3d ...)))

(let ((p (intantiate::point)))
   (widen!::point-3d p ...)
   (widen!::named-point-3d p ...))
```

is legal because *point-3d* inherits from *point* (the actual class of `p`) and because at the time of the widening to *named-point-3d*, `p` is an instance of *point-3d* and *named-point-3d* inherits from *point-3d*. The restriction that applies to widening is justified to avoid multiple simultaneous widening. A plain object may be widened only once at a time. It builds the following inheritance tree:



which preserves single subtyping property.

Type checking: Widening may be implemented for any language that supports dynamic type checking. Languages belonging to the Lisp family such as Scheme or Smalltalk fulfill that criteria. For statically typed languages wide classes cannot be used as such. The problem comes from the nature of shrinking. Shrinking, like widening, preserves the sharing structure of programs (that is, it respects `eq?`). So for example, consider the case where `obj` is shrunk from class `B` to `A`. The semantics of wide classes and shrinking guarantees that `B` is a subtype of `A`. The problem is that if any other object in the system keeps a reference to `obj` as an instance of `B`, then there is a risk of accessing fields that have been erased. That situation is very frequent for languages using "`self`" pseudo-variables. Widening is able to change the type of "`self`"! The only solution is to enforce a dynamic type test when accessing *any* wide object. This solution could be implemented within Eiffel or Java runtime system because they provide enough run-time type informations. Obviously this solution is equivalent to (partially) giving up on static type checking.

Concurrent programming: Because they operate physical updates, it is mandatory that in presence of multi-threading, widening and shrinking be atomic operations. Otherwise, it might be possible that a thread accesses the field of one wide object before that field is created. Such a situation may arise when two threads share an object and that one of the threads accesses the object while the other thread is widening it.

3.7 Possible extensions

It may be useful to propose a `shrink-then-widen!` operator. This would first shrink a wide object to the lowest common ancestor class of the current and desired wide class, then widen. This new form would allow wide classes to be defined as subclasses of an abstract class $\mathcal{C}$ because the `shrink-then-widen!` operator eliminates the need for explicit constructions of instances of the class $\mathcal{C}$.

4 Widening in a real context

As we mentioned in section 2 the compiler for the Scheme programming language Bigloo has been re-implemented with wide classes. Bigloo tries hard at optimizing the source code and thus contains many different analyses and optimization phases. For instance, Bigloo *inlines* user functions, reduces the number of instructions executed with data flow analyses such as *common sub-expression elimination* and *copy propagation*, minimizes the amount of memory needed to execute the program (*closure analysis* that improves the compilation of first class closures, *control flow analysis* that isolates polymorphic part of the programs that can be compiled has monomorphic programs), etc. All these analyses and optimizations are implemented in separate stages; in total there are twenty stages, that comes one after the other.

The whole compilation uses one unique global abstract syntax tree (AST henceforth). This data structure contains the minimal common information required by every stage. For instance, during the whole compilation nodes that implement variables are required to have a field holding the name of the variable. The AST changes from one stage to the next. Some nodes are removed while new nodes are introduced (for instance, *dead code elimination* optimization obviously prunes the AST and *loop unrolling* introduces code duplications). Nevertheless, the global structure of the AST barely changes during the compilation. Each compilation stage accepts as input the same data structure.

The only communication channel between stages is the AST. This strict policy makes each stage acts as a stand alone program. This property is very useful for a compiler because it helps the debugging process, it makes possible to change the order in which the stages are applied, and multiple applications of one stage (for instance, *copy propagation* enlarges the set of candidates to *common sub-expression elimination* and *common sub-expression elimination* enlarges the number of candidates to *copy propagation*: thus it is a convenient solution to apply the *copy propagation*, the *common sub-expression elimination* and then to re-apply the *copy propagation*).

Despite the global sharing of the AST some stages need local informations. For instance, the stage that makes the closures allocation (the stage that turns Scheme closures into C functions) needs to access the closure free variables. Once closures are allocated there are no more free variables. Freeness property is thus strictly local to that stage. Another example is the control flow analysis stage.

This compilation pass computes static approximations to actual dynamic values. Once these approximations are computed the optimizer searches for polymorphic definitions that are used with only one single type of argument. These definitions may be compiled using an efficient monomorphic framework. Once this optimization is completed there is no further need for approximations. Approximations are then local to the control flow analysis. The common framework for a stage is:

1. Widen the nodes of the abstract syntax tree that are of some use for the stage.
2. Process the computation of the stage on the wide tree.
3. Shrink the nodes of the abstract syntax tree.

This framework reduces memory consumption for the AST because only the informations relevant to one pass are live at a time. Moreover this framework helps the development of the compiler because it strictly enforces the separation between stages. If a piece of information is needed by more than one pass, it has to be held by the global tree making apparent that the information is global.

Former Bigloo versions (prior to our implementation of wide classes) relied on the *user field* strategy. With this strategy, the AST used to be extended with one "free" field acting as a hook for each stage that needed local informations. The new version of Bigloo has eliminated all the *user fields* in favor of wide instances. Widening helps at the design and the implementation of programs that conform to the compiler's framework. The improvement is fivefold:

- The source code is easier to write because we benefit from the object library and object facilities. For instance, with traditional languages access to a value held by a user field looks like:

  ```
  (define (add-approx! funcall ap)
     (let* ((uf (funcall-user-field funcall))
            (approx (cfa-approx uf)))
        (cfa-approx-set! uf (cons ap approx))))
  ```

  Using widening and the `with-access` construction the same code is turned into:

  ```
  (define (add-approx! funcall ap)
     (with-access::wide-funcall funcall (approx)
        (set! approx (cons ap approx))))
  ```

  The second version is shorter to write and we think easier to read. It is still possible to write sets of macros or functions that simulate the look and feel of the wide instance but it is in charge of the programmer. Wide classes mechanisms help the programmer with convenient ways to access objects and we think that it is the role of a programming language to provide with such help.

- In addition to accesses, wide instances benefit from the full features of the object-oriented model. Wide classes use inheritance and subtyping effectively: methods may override generic function definition for wide classes.
- The code of the compiler can be easily extended. The extensions may be implemented by the means of supplementary widening. Adding local informations to the AST or adding new compiler stages do not require a change in the whole program because the common AST can be left unchanged.
- Wide classes are safer because they benefit from the type checking of the compiler and the runtime system. As we discussed in the introduction, *user fields* may not be of another type than the most generic type (`void *` for C). In the context of a programming language that supports full polymorphism as Scheme does, this gives no opportunities for the compiler to produce good code by eliminating dynamic type tests and to help the programmer by signaling, at compile time, type errors. Wide classes is of great help in this context. For instance, in t
  The consequence of that extra safety is that within the context of our compiler, wide classes help enforcing strict separation between compiler passes. Pass separation is implemented as type tests and these tests are *automatically* introduced by the compiler.
- Wide instances implementation is more efficient than *user fields*. This is discussed in the following Section 5.

5 Wide classes implementation

This section describes an implementation strategy for wide classes. It is very close to the implementation of the current Bigloo release.

5.1 Instances

Bigloo compiles Scheme modules into C programs. The natural mapping for a Bigloo object is a C data structure. In addition to the fields found in the class declaration, two extra fields are added to the C structure. The first one, `class_num`, holds the runtime type information used for type predicates and generic functions. The second one is used as a `mark` for object marshaling. As an example, the class:

```
(class a-class x::long y::char)
```

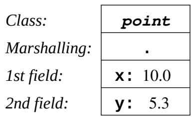
is compiled into:

```
typedef struct a_class_125 {
   class_t  class;  // The class pointer.
   void    *mark;   // The object marshaling mark.
   long     x;      // The 1st field
   char     y;      // The 2nd field
} *a_class_125_t;
```

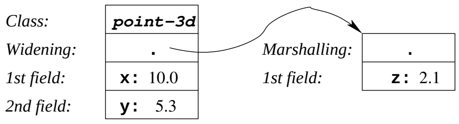
Accessing a field of an instance is thus implemented as accessing a C structure slot.

5.2 Widening and shrinking

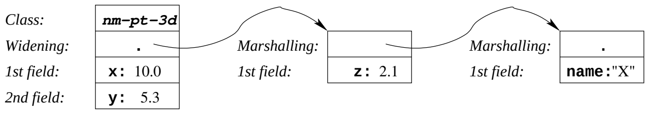
Widening implementation is straightforward: widening means allocating rooms for the new fields of the wide class; establishing a link between the instance and that new memory chunk (henceforth *wide chunk*); changing the header field of the object to reflect the operation. The wide chunk contains one extra field to implement the instance marshaling. The hooking between the instance and wide chunk uses the slot previously devoted to marshaling. Let us consider the memory implementation of an instance of class *point*.



After widening to the *point-3d* class, the instance becomes



Multiple widening repeat that framework and thus the memory layout for the widening of the instance of *point-3d* to the *named-point-3d* wide class is:



With our implementation framework, shrinking is pruning the last *widening* pointer and restoring class pointers. There is no need to store in the wide chunk the former class of the instance because, when shrinking, that class is statically known.

5.3 Accessing wide instances

Accessing the field of a plain instance is implemented by accessing the slot of a C structure. Accessing the field of a wide class requires several accesses to several C structures. For instance, the C sequence for fetching the `z` field of an instance `p` of the class *point-3d* is:

```
(point_3d_t)(p->mark)->z;
```

Fetching the `name` field of an instance of class *named-point-3d* is:

```
(named_point_3d_t)((point_3d_t)(p->mark)->mark)->name;
```

The wider the object is, the more expensive is fetching a slot of its wider chunk. Fetching a wide instance slot has the same performances as the *user field* framework. On the other hand, because of the dynamic type checking nature of Scheme, even a smart compiler is not always able to prove at compile time that the access implied in a fetch is of the correct type. In consequence, dynamic type checks have to be executed at runtime. In the case of wide instances, this requires exactly one check: is the object an instance of a given class? This operation is constant in execution time. With the *user field* approach, this does not apply anymore. The object has to be checked for its class and each accessed user field has to be checked itself. Accessing the `name` field of an instance of the wide class *`named-point-3d`* costs exactly one test. Implemented without wide classes the same operation would have required 3 tests.

6 Related work

6.1 Smalltalk `become:` and Clos `change-class`

The `become:` mechanism of Smalltalk-80™ allows the values of two objects to be swapped. The Clos library function `change-class` allows an object of a class $\mathcal{C}_1$ to be reshaped into an instance of any other class $\mathcal{C}_2$. Both Smalltalk's `become:` and Clos' `change-class` are some kind of unrestricted widening+shrinking operations. Using the Smalltalk construction two objects of unrelated classes may be swapped. Using the Clos generic function an object may be reshaped into an instance of any other class. This is not possible with widening. Widening is restricted to subclasses. When an object is widened, its actual type is turned into a subtype of its previous actual type. Such a property does not hold for `become:` or `change-class`.

We believe that preserving subtyping relationships is very important. It helps the system at statically detecting type errors and it makes the semantics of widening more intuitive and simpler.

The other important advantage of widening is that it is easy to implement. Widening may be efficiently implemented because it does not change the memory layout of an object (see Section 5). This is not true for `become:` [13] nor `change-class`. The CommonLisp `change-class`, as the widening operator, preserves `eq` equivalence. If this has no consequence for widening it has a consequence for `change-class` where a kind of double indirection is required to access object slots. If an object stays `eq` to what it used to be before a `change-class` and if `change-class` may have totally changed the shape of that object, it means that the object slots are not encoded inside the object but outside the object. This is why a double indirection is needed to access one object field.

Another important property of widening is that a program that does not use widening does not pay any extra cost for that construction. This is not true for `change-class` nor `become:` as they impact the whole run-time design.

6.2 Class predicates and instance classification

The programming language Cecil [2] extends traditional object-oriented languages with *Predicate classes* [3]. Wide classes may look like predicate classes because a predicate class is also designed to reify transient states or behavior modes of objects. A predicate class is a class extended with one predicate. An object belongs to that class if the predicate applied to that object is satisfied. The promotion from a class to a predicate class is automatic as soon as the predicate is satisfied. This is the main difference with wide classes. An object is explicitly widen and shrunk in the program.

Traditional languages cannot dispatch on general object property such as "is a window iconified?". Predicate classes are designed to deal with such situations. Wide classes could be used too because the iconification action is explicitly required and then a widening operation may take place at that time. Wide classes cannot be used when there is no location in the source file associated to the object state modification. It is not easy to model changes in object states on some global property such as "more than 10000 instances".

Predicate classes are not designed to help at reducing software retention which is the main goal of wide classes. "Cecil predicate objects (instances of predicate classes) reverse space for any fields that might be inherited from a predicate object, i.e., those fields inherited by an object assuming all predicate expressions evaluate to true. The value stored in a field of an object persists even when the controlling predicate evaluates to false and the field is inaccessible".

Moreover, it is likely that predicate classes do not enable efficient implementation. Method lookup is augmented with additional evaluations of predicate expressions for methods attached to predicate objects. Type checking might be expensive because it requires user code evaluation and the time complexity of the type checking depends on the number of predicate classes.

The Kea programming language is a functional object-oriented language [6] that proposes *classifiers*. These are similar to Cecil's predicate classes. Both support property-based classification. Being a strict functional language Kea is not designed to help at implementing object which types vary over the execution time. Objects in Kea can't be modified.

6.3 Dynamic slots

The paper [11] presents dynamic slots that enable runtime extensions of objects. A dynamic slot is an attribute that exists in all objects, an for which storage is only allocated if the dynamic slot is used, i.e., if a value is assigned to it. Dynamic slots are statically declared in the source code, hence statically typed. The main difference between dynamic slots and wide classes is that dynamic slots are available on all objects, i.e., they are orthogonal to the type system. When an instance is widen, its type changes and thus, its behavior may change. Because dynamic slots are not related to the type system, they cannot be used to change the behavior of an object. Contrarily to widening, defining dynamic slots for an object does not allow that object to react to new methods.

Dynamic slots are statically declared but the way objects can be dynamically extended is left undeclared. Thus, dynamic slots are implemented by the means of linked lists. Fetching the value of a dynamic slot requires a lookup in an association table. It is linear in time. Has it has been presented in Section 5, because widening restricts the way objects may evolve over the execution, accessing the slot of a wide object is constant in execution time, i.e., the offsets associated to the wide object slots are statically computed.

Conceptually dynamic slots exist for all object. Thus, it is unclear if dynamic slots can be erased from an object. In consequence, we do not think that dynamic slots can help at solving the software data retention problem.

The *adoption* presented in [10] is close to dynamic slots. In that model inspired from Smalltalk's `become:`, an instance may be *adopted* by another class. The adoption may enlarge the number of fields implemented by an object. Fields are thus held by lists as for dynamic slots. In opposition to dynamic slots, an instance may be adopted several times. Thus it is possible to simulate widening and shrinking with class adoptions.

6.4 Object-oriented programming with modes

The paper [12] presents an extension to the class-based object-oriented model in order to support explicit definition of logical states, named *modes*. Modes can be added to class by allowing the definition of multiple sets of operations within class definitions and modifying the late binding algorithm to automatically select between these sets of operations on the basis of the current mode of the current receiver object. In addition, a notion of transition is introduced to enable mode determination to be performed without explicit mode changing statements.

Modes helps at implementing specific behaviors with respect to some object states. For instance, if we consider the example of the window of the introduction: a window has two states, open or iconified. The behavior of a window differs if it is open or iconified. For instance, clicking on an icon opens a window but clicking on an open window simply selects it. Modes proposes an elegant way to implement such situations. Two different implementations for the `click` message will be given in the window `class`: one for each states. The transition function for windows simply records if the window is open or iconified.

The primary goal for widening is not to express states for objects. It is to help the implementation of transient data storage. Modes does not address this problem. Modes can be simulated with wide classes. To change the behavior of an instance `o` of a class $\mathcal{C}$, it is sufficient to add a wide class $\mathcal{C}'$ with no additional slot. Widening `o` from $\mathcal{C}$ to $\mathcal{C}'$ or shrinking from $\mathcal{C}'$ to $\mathcal{C}$ may change the behavior of `o` because methods can be added in $\mathcal{C}'$. However, because widening uses inheritance, it is restricted by subtyping relationships. Modes implement type unions rather that subtypes. Type unions are not easily mapped to inheritance trees, specially with our system that makes use of single inheritance.

7 Extensions

7.1 Multiple widening

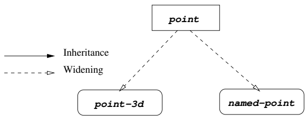
A possible extension to wide classes could be to allow multiple widening of a same object. This could be useful when unrelated extra informations have to be added to an object. To get back to the point example, it could be a useful feature to allow a plain `point` to be widened into a `point-3d` but also to be widened next into a `named-point`. The example below illustrates that possibility:

```
(module points
   (export (class point ...)
           (wide-class point-3d::point ...)
           (wide-class named-point::point ...)))
```

The same component of an instance may be widened more than once:

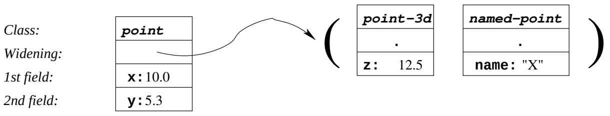
```
1: (let ((p (instantiate::point)))
2:    (widen!::point-3d p)
3:    (widen!::named-point p))
```

The inheritance tree is:



This is forbidden with the model previously presented because at line *3* `named-point` is not a subtype of the actual type of `p` (i.e. `point-3d`). The multiple widening would require to change the `shrink!` operation. It is now mandatory to decide which component of a wide object must be shrunk.

We have decided not to implement the multiple widening because we have found no way to implement it as efficiently as simple widening. In the presence of multiple widening an object could contain several wide chunks (see Section 5). The new memory layout with multiple widening may look like:

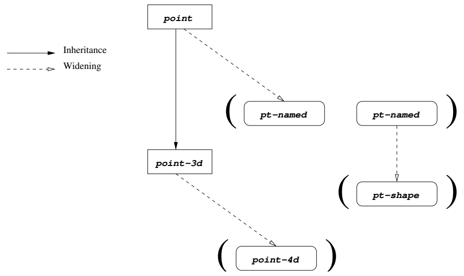


Testing for a typing property (e.g. is the object `p` an instance of the class $\mathcal{C}$?) is harder with multiple widening because an object may have several simultaneous actual types that are not in a subtyping relationship. It is likely that there is no way to perform type checking without checking in sequence all actual types of a wide object.

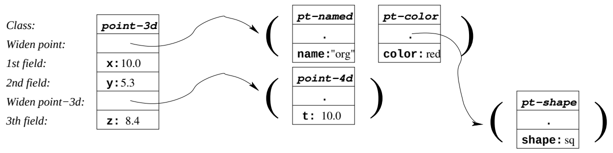
Accessing the slots of a wide object in presence of multiple widening is a more complex operation than with single widening because it is first necessary to fetch the correct wide chunk. The complexity of the worst case of that operation is $\mathcal{O}(c)$ where c is the number of classes.

7.2 Multiple widening and multiple inheritance

The second step to widening is to meet the multiple inheritance paradigm. With that programming model it would be coherent to allow a plain object to be widened more than once for each of the class it inherits from. Thus, it would be also possible to widen several times the same component of an instance. That is, it would be possible to first widen the `point` part into, for instance, a `point-named` and then to widen one more time the `point` part into a `point-colored`, etc. An instance must be provided with as many slots for widening as the number of classes it inherits from. For our example, a `point-3d` instance must be provisioned with 2 slots for widening, one for the `point` component, one for the `point-3d` component. This is not enough: a slot for widening must contain the list of wide chunks instead of pointer to a single wide chunk. The inheritance tree would look like:



The data layout for such an object would look like:



With that naive framework accessing the slot of a wide chunk and checking for a wide type is expensive. Most likely the clever representations that are used within implementation for languages that support multiple inheritance would applied here.

Conclusion

Wide classes extend traditional object-oriented programming languages with a construction that models transient properties. Such a construction does not exist in regular languages. Thanks to wide classes and wide inheritance the shape of an object may change during its life-time. An object can be, at a certain point of the execution, *widen* to another class, that is, an object may be extended with additional fields. Later on, this same object can be *shrunk* to its original shape. We think these features may help the programming task because we think programs using data structures that vary over execution time are frequent.

Wide classes are efficient at solving software retention. This problem arises when an actual data structure is not able to reflect time properties in order to minimize the memory consumption of an execution. We have shown how wide classes apply to the construction of our Scheme compiler and how they succeed in improving the programming style.

Wide classes may also be used to model object states. If an object goes from one state to another this transformation can be implemented by the means of wide classes. Wide classes provide a mechanism to change the type of an object and thus to change the way generic functions will react when applied to it.

Wide classes could probably be easily implemented in other object-oriented programming languages although dynamic type checking or, at least runtime type informations, are likely to be required. We don't think languages that make use of type inference like O'Caml [9] can benefit from wide classes. W [4] based type inference algorithms are not able to detect errors in programs like the one shown below where the last `point-3d-z` access is illegal:

```
(let ((p (instantiate::point)))
   (widen!::point-3d p (z 45))
   (+ (point-3d-z p) (begin (shrink! p) (point-3d-z p))))
```

The reason is that current type inference algorithms of that trend cannot associate several types to a same object. An object has to be either a ***point*** or a ***point-3d*** but it can't be at some point in time a ***point***, becomes a widen ***point-3d*** and returns back to a plain ***point***. On the other hand, languages that propose coercion operations like Java could probably be extended with wide classes although at the cost of extra dynamic type checks.

Acknowledgments

Many thanks to Christian Queinnec, Jan Vitek and to Céline for their helpful feedbacks on this work.

References

1. D. Bobrow, L. DeMichiel, R. Gabriel, S. Keene, G. Kiczales, and D. Moon. Common lisp object system specification. In *special issue*, number 23 in SIGPLAN Notices, September 1988.
2. C. Chambers. The Cecil Language: Specification and Rationale. Technical Report 93-03-05, University of Washington, Department of computer Science and Engineering, March 1993.
3. C. Chambers. Predicate classes. In O. Nierstrasz, editor, *Proceedings ECOOP'93*, LNCS 707, pages 268–296, Kaiserslautern, Germany, July 1993. Springer-Verlag.
4. L. Damas and R. Milner. Principle Type Inference for Functional Programs (extended abstract). In *9th ACM Symposium on Principle of Programming Langugages*, pages 207–212, 1982.
5. A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1983.

6. J. Hamer. Un-mixing inheritance with classifiers. In *Multiple Inheritance and Multiple Subtyping: Position papers of the ECOOP'92 Workshop*, LNCS 707, pages 6–9, Utrect, Netherlands, July 1992. Springer-Verlag.
7. IEEE Std 1178-1990. *IEEE Standard for the Scheme Programming Language*. Institute of Electrical and Electronic Engineers, Inc., New York, NY, 1991.
8. C. Queinnec. Designing MEROON v3. In *Workshop on Object-Oriented Programming in Lisp*, 1993.
9. D. Rémy and J. Vouillon. Obective ML: A simple object-oriented extension of ML. In *Symposium on Principles of Programming Languages*, pages 40–53, 1997.
10. F. Rivard. *Évolution du comportement des objets dans les langages à classes réflexifs*. 98-1-info, École des Mines de Nantes, 1998.
11. R. Schmidt. Dynamically Extensible Objects in a Class-Based Language. In *TOOLS USA*, July 1997.
12. A. Taivalsaari. Object-oriented programming with modes. *Journal of Object-oriented programming*, pages 25–32, June 1993.
13. D. Ungar. *The Design and Evaluation of a High Performance Smalltalk System*. MIT Press, Cambridge, Mass., USA, 1986.