Predicate Classes

Craig Chambers
Department of Computer Science and Engineering
University of Washington

Abstract. Predicate classes are a new linguistic construct designed to complement normal classes in object-oriented languages. Like a normal class, a predicate class has a set of superclasses, methods, and instance variables. However, unlike a normal class, an object is automatically an instance of a predicate class whenever it satisfies a predicate expression associated with the predicate class. The predicate expression can test the value or state of the object, thus supporting a form of implicit property-based classification that augments the explicit type-based classification provided by normal classes. By associating methods with predicate classes, method lookup can depend not only on the dynamic class of an argument but also on its dynamic value or state. If an object is modified, the property-based classification of an object can change over time, implementing shifts in major behavior modes of the object. A version of predicate classes has been designed and implemented in the context of the Cecil language.

1 Introduction

One of the chief strengths of object-oriented languages is the ability of methods to describe the circumstances for which they are intended to be used. In singly-dispatched (receiver-based) object-oriented languages, methods are placed within a class, and only apply for objects that inherit from that class. In multiply-dispatched languages, a multi-method's argument specializers describe the kinds of arguments for which it should be used. Method lookup uses the dynamic type of the actual arguments of a message to select the right method to invoke. From another standpoint, this dynamic dispatching mechanism is an important piece of infrastructure that supports the improved modelling capabilities of object-oriented languages: classes represent entities in the application domain, and methods attached to classes implement the operations on the entities. Classes and inheritance help to model natural specialization hierarchies in the application domain and support better factoring of the implementation of the application-domain entities.

Traditional object-oriented languages can model and implement various sorts of static type-based classifications of objects using classes and inheritance. However, some kinds of classifications escape these linguistic constructs. For example, few object-oriented languages can reify the concept of an empty collection in such a way that whenever a collection is empty, any methods attached to the empty-collection concept would apply, but when the collection is mutated to become non-empty, the empty-collection behavior would no longer apply. Few object-oriented languages allow methods to specialize on the identity or state of an argument, in addition to its dynamic type.

Predicate classes extend the standard object-oriented modelling constructs by reifying transient states or behavior modes of objects. A predicate class has all the properties of a normal class, including a name, a set of superclasses, a set of methods, and a set of instance variables. Additionally, a predicate class has an associated predicate expression. A predicate class represents the subset of the instances of its superclass(es) that also satisfy the predicate. Whenever an object is an instance of the superclasses of the predicate class, and the predicate expression evaluates to true when invoked on the object, the object will automatically be considered to inherit from the predicate class as well. While the object inherits from a predicate class, it inherits all the methods and instance variables of the predicate class. If the object's state later changes and the predicate expression no longer evaluates to true, the inheritance of the object will be revised to exclude the predicate class. Predicate classes thus support a form of automatic, dynamic classification of objects, based on their run-time value, state, or other user-defined properties. To the extent that these transient states are important in the application domain, predicate classes can help in modelling and implementing them.

Predicate classes are a relatively language-independent idea. For concreteness, however, we have been exploring them in the context of the Cecil language [Chambers 92b, Chambers 93]. The next section of this paper presents a brief overview of Cecil. Section 3 then describes in more detail the semantics of predicate classes as included in Cecil, and section 4 presents several examples of predicate objects at work. Section 5 discusses related work.

2 Cecil

Cecil is a purely object-oriented language based on multi-methods. Static type declarations are optional in Cecil. Where present, types are checked statically; otherwise, type checking is done dynamically as messages are sent to objects. For most of this paper, we will concentrate on the dynamically-typed core of Cecil.

The following Cecil example (not using predicate classes) implements simple linked lists:

```
object list isa collection;

object nil isa list;
method length(n@nil) { 0 }
method do(n@nil, closure) {}

object cons isa list;
field head(c@cons);
field tail(c@cons) := nil;
method length(c@cons) { 1 + c.tail.length }
method do(c@cons, closure) {
    eval(closure, c.head); do(c.tail, closure); }

method prepend(x, l@list) {
    object isa cons { head := x, tail := l } }
```

```
method print(c@collection) {
    print("[");
    do(c, &(elem){
        print("\t"); print(elem); print("\n");
    });
    print("]"); }
```

The constructs in this example are explained briefly in the following subsections. More information on the Cecil language is available in other papers [Chambers 92b, Chambers 93].

2.1 Objects

Cecil is classless, associating methods directly with objects and allowing objects to inherit directly from other objects. In general, new named objects (akin to classes or one-of-a-kind global objects) are created using the general form:

object *name* **isa** $parent_1$, ..., $parent_n$;

where the $parent_i$ name the object's parents. An object's parents act roughly like superclasses: the object inherits methods and fields (instance and class variables) from its parents. Zero or more parents are allowed. New anonymous objects, such as those created at run-time, use a similar syntax but omit the object name, as with the object created in the `prepend` method.

2.2 Methods

Methods are defined using the general form:

method *name* ($formal_1$@obj_1, ..., $formal_n$@obj_n) { *statements* }

Any of the @obj_i may be omitted. Where present, these *argument specializers* indicate that the method is defined only for message arguments that are descendants[1] of the object named obj_i. Unspecialized formals are treated as specialized to a "top" object that is implicitly an ancestor of all other objects, allowing unspecialized formals to accept any argument. By specializing on exactly the first argument, traditional singly-dispatched methods can be simulated. Specializing on no arguments allows normal procedures or default routines to be implemented. Argument specializers are viewed as attaching the multi-method to the specializing object(s), much as methods are defined inside a class in a singly-dispatched language.

To select the method invoked by a message send, the system first finds all methods that have the same name and number of arguments as the message and whose argument specializers are (improper) ancestors of the corresponding actuals; these are the *applicable methods* for the message. The system then orders the applicable methods according to specificity: one method M is more specific than another method N exactly when each of M's argument specializers is an (improper) descendant N's corresponding argument specializer and at least one of M's argument specializers is a proper descendant of N's corresponding argument specializer. Finally, the system selects the

1. The *descendant* relation is the reflexive, transitive closure of the *child* relation; an object is considered a descendant of itself. The *ancestor* relation is the inverse of the *descendant* relation. To reinforce the fact that the descendant and ancestor relations are reflexive, we will sometimes describe the relation as "improper."

single most specific method as the target of the message. If the system finds no applicable methods, it reports a "message not understood" error. If the system finds several applicable methods, but no single method is more specific than all other applicable methods, then it reports a "message ambiguous" error. Otherwise, the system has successfully located the single most applicable method for the message. The method is invoked and the result of its last statement is returned as the result of the message.

Unlike other languages with multi-methods, no ordering of parents or arguments is used to automatically resolve ambiguities. Cecil includes a resend mechanism, inspired by SELF's resend mechanism and similar to Smalltalk's `super`, CLOS's `call-next-method`, and C++'s qualified messages, that allows a method to invoke the method it is overriding and to explicitly resolve ambiguities among several inherited methods.

Argument specializers are not type declarations. Argument specializers are used to determine the outcome of method lookup. After method lookup is resolved, any type declarations attached to the formal parameters are checked (this checking is done statically before the program is run). Type declarations do not influence method lookup.

Methods can be encapsulated within an object despite the presence of multi-methods. If a method is prefixed with the `private` keyword, access to the method is restricted to methods associated with its argument specializers. Details on encapsulation and the intended programming model for Cecil may be found in an earlier paper [Chambers 92b].

2.3 Fields

Instance variables and class variables are realized in Cecil using the `field` construct. A field is defined using a notation similar to that used to define a method:

field *field-name* (*formal@obj*) := *expr;*

where *obj* names the object containing the new instance variable and *expr*, if present, provides an initial value for the field. Objects inheriting from *obj* receive their own copies of the *field-name* instance variable. If the field declaration is prefixed with the `shared` keyword, all inheriting objects share a single memory location, much like a class variable. Non-shared fields of newly-created objects may be provided an initial value as part of the object creation operation, by suffixing the object creation expression with initialization code of the following form:

... {*field-name*$_1$:= *expr*$_1$, ..., *field-name*$_n$:= *expr*$_n$}

A field may be restricted to be immutable. Fields prefixed with the `read_only` annotation are shared fields that cannot be modified. Fields prefixed with the `init_only` annotation are object-specific fields that cannot be modified after the containing object is created.

Fields are accessed solely through message sends, enabling fields to be overridden with methods and vice versa. To make accessing fields syntactically convenient, dot-notation syntactic sugar exists for messages of the following forms:

- *expr*`.`*name* is sugar for *name* (*expr*)
- *expr*`.`*name* := *expr2* is sugar for `set_`*name* (*expr*, *expr2*)

Any message of either of the above forms may be sugared, irrespective of whether it invokes a field accessor method or a normal method. Fields may be encapsulated within an abstraction in the same way that methods are encapsulated.

2.4 Closures

Closure objects are analogous to blocks in Smalltalk and first-class functions in other languages. Closures are heavily used in Cecil programs as arguments to user-defined control structures and to handle exceptions. A closure object is created with an expression of the following form:

& (*formal*$_1$, ..., *formal*$_n$) { *statements* }

Such an expression constructs a new object that inherits from the built-in `closure` object (upon which operations such as `loop` and `while` are defined). The new closure object also has an attached method named `eval` of the following form:

method `eval(`<*anon*>`@`<*the_closure*>, *formal*$_1$, ..., *formal*$_n$) { *statements* }

The body of the `eval` method executes in a context that is nested within the closure's lexically-enclosing context. Thus, to "invoke" a closure, the `eval` message is sent to the closure along with any additional arguments expected by the closure. Closures are first-class and may be returned upwards out of their enclosing scope. An example of a closure constructor expression appears in the `print` method defined earlier.

When invoked, a closure may either return normally to the sender of the `eval` message or force a *non-local return* from the closure's lexically-enclosing method. Such a non-local return is analogous to a non-local return from a block in Smalltalk or a `return` statement in a traditional language.

3 Predicate Objects

Because Cecil is object-based rather than class-based, the adaptation of the general idea of predicate classes to Cecil's object model is called *predicate objects*. The next several subsections describe how predicate objects are declared in Cecil and how they interact with normal objects, methods, and fields. Subsection 3.7 describes support for static type checking of predicate objects, and subsection 3.8 sketches some implementation strategies.

3.1 Predicate Objects

A predicate object is defined much like a normal object, except that a predicate object is introduced with the keyword `pred` and may have an additional `when` clause:

pred *name* **isa** *parent*$_1$, ..., *parent*$_n$ **when** *predicate-expr*;

For normal objects, one object is a child of another object exactly when the relationship is declared explicitly through `isa` declarations by the programmer. Predicate objects, on the other hand, support a form of automatic property-based classification: an object *O* is automatically considered a child of a predicate object *P* exactly when the following two conditions are satisfied:

- the object *O* is an (improper) descendant of each of the parents of the predicate object *P*, and

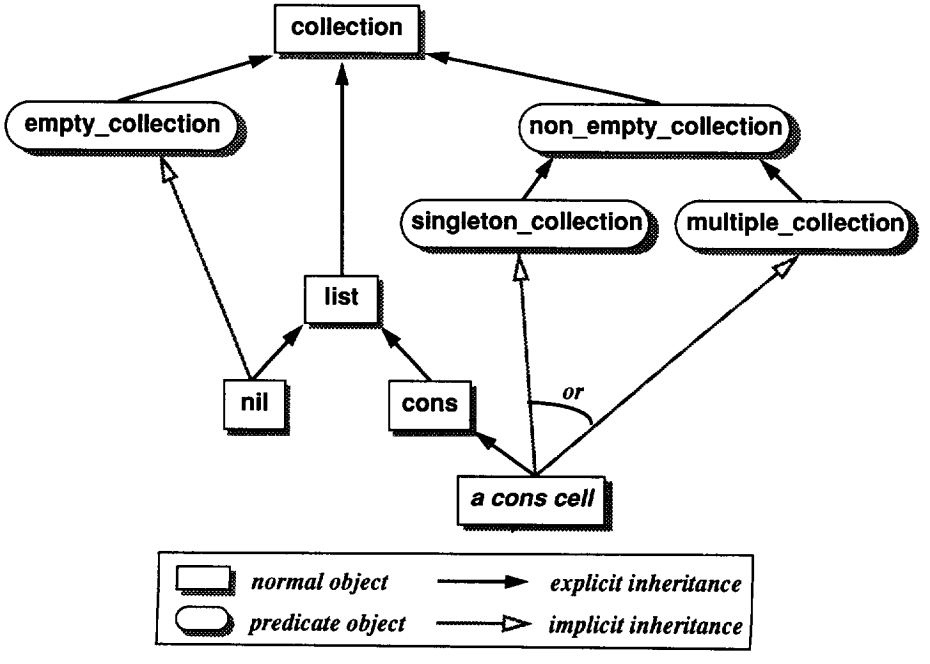
- the predicate expression of the predicate object P evaluates to true, when evaluated in a scope where each of the $parent_i$ names is bound to the object O.

By evaluating the predicate expression in a context where the parent names refer to the object being tested, the predicate expression can query the value or state of the object.

For example, the following predicate objects describe various important conditions of collections:

```
pred empty_collection isa collection
    when collection.length = 0;

pred non_empty_collection isa collection
    when collection.length > 0;

pred singleton_collection isa non_empty_collection
    when non_empty_collection.length = 1;

pred multiple_collection isa non_empty_collection
    when non_empty_collection.length > 1;
```

The object `nil` defined earlier in section 2 would be implicitly a child of the `empty_collection` predicate object, since `nil` is also a descendant of `empty_collection`'s parent (`collection`) and evaluating the expression "`collection.length = 0`" in a context where the name `collection` is bound to `nil` returns true. Similarly, a `cons` object would be considered a child of `non_empty_collection`. A particular `cons` object would also be a child of either `singleton_collection` or `multiple_collection`, depending on the length of the list at run-time. The following diagram illustrates the inheritance graph of the list example extended with these predicate object classifications:



An object may inherit explicitly from a predicate object, with the implication that the predicate expression will always evaluate to true for the child object; the system verifies this assertion dynamically. In the above example, `singleton_collection` is declared to inherit from the predicate object `non_empty_collection`. This implies that any object that is a `singleton_collection` is also a `non_empty_collection`. For this simple case, one might expect the Cecil system to deduce automatically that the predicate "`non_empty_collection.length = 1`" implies the predicate "`collection.length > 0`," and so infer the inheritance link from `singleton_collection` to `non_empty_collection` automatically. However, in Cecil, nearly all operations, including basic operations such as comparisons, are user-defined, and the system cannot reason about the semantics of these operations in general. Consequently, the programmer must explicitly declare any such implications among predicate classes. Section 5 describes some other systems that restrict predicates to using only built-in operations in order to infer these inheritance relations automatically.

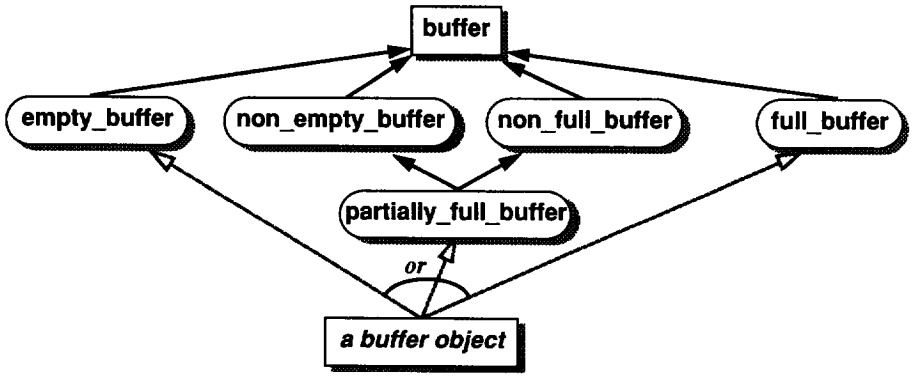
3.2 Predicate Objects and Methods

Predicate objects become more useful once methods and fields are associated with them. Predicate objects can be argument specializers just like normal objects. The method lookup rules remain the same: a method is applicable for a message when each actual argument object is a descendant of the corresponding argument specializer (independent of whether the specializer is a normal object or a predicate object), and one method is considered more specific than another when its argument specializers are descendants of the corresponding specializers of the other method, whether or not those specializers are normal or predicate objects.

For example, the following code implements a bounded buffer object with state-dependent behavior modes:

```
object buffer isa collection;

field elements(b@buffer); -- a queue of elements
field max_size(b@buffer); -- an integer

method length(b@buffer) { b.elements.length }
method is_empty(b@buffer) { b.length = 0 }
method is_full(b@buffer)  { b.length = b.max_size }

pred empty_buffer isa buffer when buffer.is_empty;
method get(b@empty_buffer) { ... } -- raise error or block caller

pred non_empty_buffer isa buffer when not(buffer.is_empty);
method get(b@non_empty_buffer) {
    remove_from_front(b.elements) }

pred full_buffer isa buffer when buffer.is_full;
method put(b@full_buffer, x) { ... } -- raise error or block caller

pred non_full_buffer isa buffer when not(buffer.is_full);
method put(b@non_full_buffer, x) {
    add_to_back(b.elements, x); }

pred partially_full_buffer isa
    non_empty_buffer, non_full_buffer;
```

The following diagram illustrates the inheritance hierarchy created by this example (the explicit inheritance link from the buffer object to `buffer` is omitted):



Predicate objects increase expressiveness for this example in two ways. First, important states of bounded buffers, e.g., empty and full states, are explicitly identified in the program and named. Besides documenting the important conditions of a bounded buffer, the predicate objects remind the programmer of the special situations that code must handle. This can be particularly useful during maintenance phases as code is later extended with new functionality. Second, attaching methods directly to states supports better factoring of code and eliminates `if` and `case` statements, much as does distributing methods among classes in a traditional object-oriented language. In the absence of predicate objects, a method whose behavior depended on the state of an argument object would include an `if` or `case` statement to identify and branch to the appropriate case; predicate objects eliminate the clutter of these tests and clearly separate the code for each case. In a more complete example, several methods might be associated with each special state of the buffer. By factoring the code, separating out all the code associated with a particular state or behavior mode, we hope to improve the readability and maintainability of the code.

The `partially_full_buffer` predicate object defined above illustrates that a predicate object declaration need not specify its own predicate expression. Such a predicate object may still depend on a condition if at least one of its ancestors is a predicate object. In the above example, the `partially_full_buffer` predicate object has no explicit predicate expression, yet since an object only inherits from `partially_full_buffer` whenever it already inherits from both `non_empty_buffer` and `non_full_buffer`, the `partially_full_buffer` predicate object effectively repeats the conjunction of the predicate expressions of its parents, in this case that the buffer be neither empty nor full.

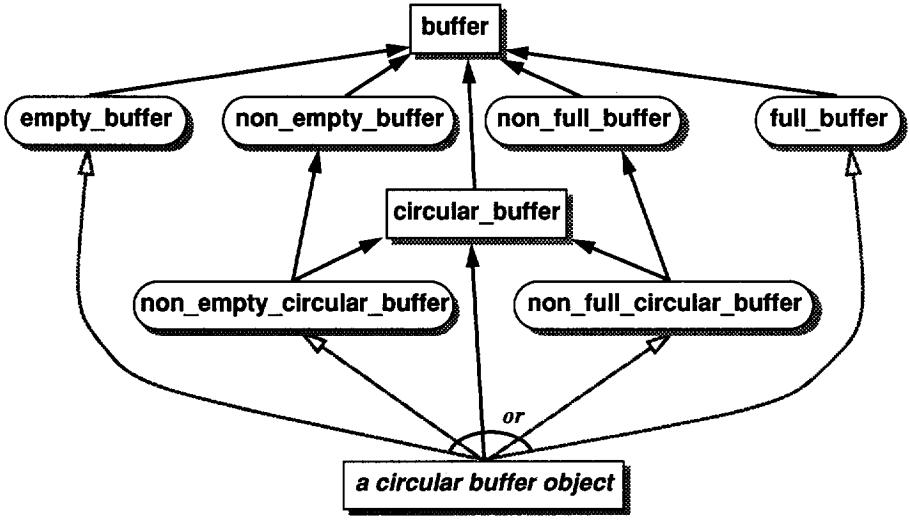
3.3 Predicate Objects and Inheritance

Predicate objects are intended to interact well with normal inheritance. If an abstraction is implemented by inheriting from some other implementation, any predicate objects that specialize the parent implementation will automatically specialize the child implementation whenever it is in the appropriate state. For example, a new

implementation of bounded buffers could be built that used a fixed-length array with insert and remove positions that cycle around the array:

```
object circular_buffer isa buffer;
field array(b@circular_buffer); -- a fixed-length array of elements
field insert_pos(b@circular_buffer); -- an index into the array
field remove_pos(b@circular_buffer); -- another integer index
method max_size(b@circular_buffer) { b.array.length }
method length(b@circular_buffer) {
    -- % is modulus operator
    (b.insert_pos - b.remove_pos) % b.array.length }

pred non_empty_circular_buffer isa
          circular_buffer, non_empty_buffer;
method get(b@non_empty_circular_buffer) {
    var x := fetch(b.array, b.remove_pos);
    b.remove_pos := (b.remove_pos + 1) % b.array.length;
    x }

pred non_full_circular_buffer isa
          circular_buffer, non_full_buffer;
method put(b@non_full_circular_buffer, x) {
    store(b.array, b.insert_pos, x);
    b.insert_pos := (b.insert_pos + 1) % b.array.length; }
```

The following diagram illustrates the extended inheritance graph for bounded and circular buffers (the `partially_full_buffer` predicate object is omitted):



Since the `circular_buffer` implementation inherits from the original `buffer` object, a `circular_buffer` object will automatically inherit from the `empty_buffer` or `full_buffer` predicate object whenever the `circular_buffer` happens to be in one of those states. No `empty_circular_buffer` or `full_circular_buffer` objects need to be implemented if specialized behavior is not needed. The `non_empty_circular_buffer` and `non_full_circular_buffer`

predicate objects are needed to override the default `get` and `put` methods in the non-blocking states. Any object that inherits from `circular_buffer` and that also satisfies the predicate associated with `non_empty_buffer` will automatically be classified as a `non_empty_circular_buffer`.

The specification of when an object inherits from a predicate object implicitly places a predicate object just below its immediate parents and after all other normal children of the parents. For example, consider an empty circular buffer object. Both the buffer object and its parent, the `circular_buffer` object, will be considered to inherit from the `empty_buffer` predicate object. Because `circular_buffer` is considered to inherit from `empty_buffer`, any methods attached to `circular_buffer` will override methods attached to `empty_buffer`. Often this is the desired behavior, but at other times it might be preferable for methods attached to predicate objects to override methods attached to "cousin" normal objects.[1] If this were the case, then the buffer code could be simplified somewhat, as follows:

```
object buffer isa collection;
... -- elements, length, etc.
method get(b@buffer) { remove_from_front(b.elements) }
method put(b@buffer, x) { add_to_back(b.elements, x); }

pred empty_buffer isa buffer when buffer.is_empty;
method get(b@empty_buffer) { ... } -- raise error or block caller

pred full_buffer isa buffer when buffer.is_full;
method put(b@full_buffer, x) { ... } -- raise error or block caller

object circular_buffer isa buffer;
... -- array, insert_pos, length, etc.
method get(b@circular_buffer) {
    var x := fetch(b.array, b.remove_pos);
    b.remove_pos := (b.remove_pos + 1) % b.array.length;
    x }
method put(b@circular_buffer, x) {
    store(b.array, b.insert_pos, x);
    b.insert_pos := (b.insert_pos + 1) % b.array.length; }
```

The non-blocking versions of `get` and `put` would be associated with the `buffer` object directly, and the `non_empty_buffer`, `non_full_buffer`, and `partially_full_buffer` predicate objects could be removed (if desired). The non-blocking `get` and `put` routines for circular buffers would similarly be moved up to the `circular_buffer` object itself, with the `non_empty_circular_buffer` and `non_full_circular_buffer` predicate objects being removed also. If the methods attached to the `empty_buffer` object were considered to override those of the `circular_buffer` object, then sending `get` to a circular buffer that was empty would (correctly) invoke the `empty_buffer` implementation. In the current semantics of predicate objects in Cecil, however, the `circular_buffer`'s implementation of `get` would be invoked, leading to an error.

1. One object is a cousin of another if they share a common ancestor but are otherwise unrelated.

A third potential semantics would be to consider the predicate object to be unordered with respect to "cousin" objects, and methods defined on two cousins to be mutually ambiguous. An important area of continuing work is determining whether one semantics is most helpful or if the programmer needs to use different rules in different circumstances.

3.4 Dynamic Reclassification of Objects

Since the state of an object can change over time (fields can be mutable), the results of predicate expressions evaluated on the object can change. If this happens, the system will automatically reclassify the object, recomputing its implicit inheritance links. For example, when a buffer object becomes full, the predicates associated with the `non_full_buffer` and `full_buffer` predicate objects both change, and the inheritance graph of the buffer object is updated. As a result, different methods may be used to respond to messages, such as the `put` message in the filled buffer example.

Semantically, predicate expressions are evaluated lazily as part of method lookup, rather than eagerly as the state of an object changes. Only when the value of some predicate expression is needed to determine the outcome of method lookup is the predicate evaluated. Since predicate expressions are expected to be pure functions, the exact time of evaluation of predicate expressions can usually be ignored. In any case, implementations are free to evaluate predicate expressions at other times, as described in section 3.8, as long as the externally-visible semantics is unchanged.

3.5 Predicate Objects and Fields

Fields may be associated with a predicate object. The semantics of accessing a field attached to a predicate object has already been specified: fields are accessed solely through message sends, and method lookup in the presence of predicate objects has been defined. However, the *contents* of a field inherited from a predicate object is less obvious. Several questions arise: does the field exist only when the controlling predicate evaluates to true? Does its value persist while the predicate evaluates to false? How does such a field get initialized?

In our version of predicate objects in Cecil, objects reserve 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. When the field becomes accessible again, its value will be the same as when it was last visible. At object-creation time, an initial value may be provided for fields inherited from predicate objects, even if those fields may not be visible in the newly-created object.

The following example exploits this semantics to implement a graphical window object that can be either expanded or iconified. Each of the two important states of the window remembers its own screen location (using a field named `position` in both cases), plus some other mode-specific information such as the text in the window and the bitmap of the icon, and this data persists across openings and closings of the window:

```
object window isa interactive_graphical_object;
field iconified(@window) := false;
method display(w@window) {
    -- draw window using w.position
    ... }
method erase(w@window) {
    -- clear space where window is
    ... }
method move(w@window, new_position) {
    -- works for both expanded and iconified windows!
    w.erase; w.position := new_position; w.display; }

pred expanded_window isa window when not(window.iconified);
field position(@expanded_window) := upper_left;
field text(@expanded_window);
method iconify(w@expanded_window) {
    w.erase; w.iconified := true; w.display; }

pred iconified_window isa window when window.iconified;
field position(@iconfied_window) := lower_right;
field icon(@iconified_window);
method open(w@iconified_window) {
    w.erase; w.iconified := false; w.display; }

method create_window(open_position, iconified_position,
                     text, icon) {
    object isa window { iconified := false,
        position@open_window := open_position,
        position@iconified_window := iconified_position,
        text := text, icon := icon } }
```

A window object has two `position` fields, but only one is visible at a time. This allows the `display`, `erase`, and `move` routines to send the message `position` as part of their implementation, without needing to know whether the window is open or closed. The `create_window` method initializes both `position` fields when the window is created, even though the position of the icon is not visible initially. The `position@`*object* notation used in the field initialization resolves the ambiguity between the two `position` fields.

3.6 Predicates on Methods

A predicate object characterizes the value or state of a single object. By using a predicate object as an argument specializer, a method can restrict its applicability to arguments in a particular state. In some cases, however, a method's applicability might be conditional on some predicate defined over all of its arguments as a group. For example, one early motivation for predicates in Cecil was to be able to write code like the following, which implements iterating through two lists in parallel:

```
method pair_do(l1@cons, l2@cons, closure) {
    eval(closure, l1.head, l2.head);
    pair_do(l1.tail, l2.tail, closure); }
method pair_do(l1@list, l2@list, closure)
    when l1@nil | l2@nil {}
```

The predicate restricts the second `pair_do` method to those cases where either or both of the list arguments are `nil`. Without the predicate expression, the code would be less robust to future programming extensions. If a new implementation of lists were added later, such as a special representation for singleton lists, but appropriate `pair_do` methods for singleton lists were accidentally omitted, the system would silently use the second "default" `pair_do` method when iterating through a singleton list, rather than signalling a "message not understood" error. To achieve this level of robustness without using a method predicate expression, the second `pair_do` method would need to be written using three separate methods:

```
method pair_do(l1@nil,  l2@list, closure) {}
method pair_do(l1@list, l2@nil,  closure) {}
method pair_do(l1@nil,  l2@nil,  closure) {}
```

The third method is needed to resolve the ambiguity between the first two methods when iterating through two `nil` objects; method lookup in Cecil does not prioritize arguments based on position.

One open issue with method predicate expressions is how to order predicated methods according to specificity. Method lookup depends on being able to order the applicable methods by specificity, raising an "ambiguous message" error if a single most specific method cannot be identified. For predicate objects, explicit inheritance declarations between two predicate objects can reflect when one predicate expression implies another, but methods cannot be named so easily in order to express an ordering.

At present, our extension of Cecil does not include predicated methods. Predicate objects already handle many practical cases simply and clearly. We prefer to gain experience with predicate objects before considering extensions such as predicated methods.

3.7 Static Type Checking

Cecil supports a static type system that can guarantee at program definition time that no "message not understood," "message ambiguous," "private method accessed," or "uninitialized field accessed" error messages can occur at run-time. When extended with predicate objects, these same guarantees should be preserved. The central type-checking problem introduced by predicate objects is that an object's inheritance graph, and consequently the set of methods inherited by an object, can change at run-time. To guarantee type safety, the type checker must verify that for each message declared in the interface of some object *O*:

- at all times there is an implementation of the message inherited by the object *O*, and
- at no time are there several mutually ambiguous implementations of the message inherited by the object *O*.

The set of methods inherited by the object *O* from normal objects is fixed at program-definition time and can be type-checked in the standard way. Methods inherited from predicate objects pose more of a problem. If two predicate objects might be inherited simultaneously by an object, either one predicate object must be known to override the other or they must have disjoint method names. For example, in the bounded buffer implementation, since an object can inherit from both the `non_empty_buffer` and

the non_full_buffer predicate objects, they cannot implement methods with the same name. Similarly, if the only implementations of some message are in some set of predicate objects, then one of the predicate objects must always be inherited for the message to be guaranteed to be understood. In other words, the checker needs to know when one predicate object *implies* another, when two predicate objects are *mutually exclusive*, and when a group of predicate objects is *exhaustive*. Once these relationships among predicate objects are determined, the rest of type-checking becomes straightforward.

Ideally, the system would be able to determine all these relationships automatically by examining the predicate expressions attached to the various predicate objects. However, as described earlier in section 3.1, predicate expressions in Cecil can run arbitrary user-defined code, and consequently the system would have a hard time automatically inferring implication, mutual exclusion, and exhaustiveness. Consequently, we rely on explicit user declarations to determine the relationships among predicate objects; the system can verify dynamically that these declarations are correct. Section 5 describes some other systems that can infer some of the relationships automatically by restricting the form of the predicate expressions.

A declaration already exists to describe when one predicate object implies another: the isa declaration. If one predicate object explicitly inherits from another, then the first object's predicate is assumed to imply the second object's predicate. Any methods in the child predicate object override those in the ancestor, resolving any ambiguities between them. For example, a method associated with non_empty_circular_buffer overrides a method associated with non_empty_buffer, since non_empty_circular_buffer inherits explicitly from non_empty_buffer.

Mutual exclusion among a group of predicate objects is declared using the following notation:

disjoint $object_1$, ..., $object_n$;

The predicate objects named by each of the $object_i$ are assumed by the static type checker to never be inherited simultaneously, i.e., that at most one of their predicate expressions will evaluate to true at any given time. Mutual exclusion of two predicate objects implies that the type checker should not be concerned if both predicate objects define methods with the same name, since they cannot both be inherited by an object. To illustrate, the following declarations extend earlier predicate objects with mutual exclusion information:

```
disjoint empty_collection, non_empty_collection;
disjoint singleton_collection, multiple_collection;
disjoint empty_buffer, non_empty_buffer;
disjoint full_buffer, non_full_buffer;
```

The system can infer that empty_collection is mutually exclusive with singleton_collection and multiple_collection, since singleton_collection and multiple_collection both inherit from non_empty_collection. A similar inference determines that empty_buffer and full_buffer are mutually exclusive with partially_full_buffer. Note that empty_buffer and full_buffer are not necessarily exclusive.

A final declaration asserts that a group of predicate objects exhaustively cover the possible states of some other object, using the following notation:

cover *object* **by** $object_1$, ..., $object_n$;

This declaration implies that whenever an object *O* descends from *object*, the object *O* will also descend from at least one of the $object_i$ predicate objects; each of the $object_i$ are expected to descend from *object* already. Exhaustiveness implies that if all of the $object_i$ implement some message, then any object inheriting from *object* will understand the message. For example, the following coverage declarations extend the earlier predicate objects:

```
cover collection by empty_collection, non_empty_collection;
cover non_empty_collection by singleton_collection,
                              multiple_collection;
cover buffer by empty_buffer,
                partially_full_buffer,
                full_buffer;
```

Often a group of predicate objects divide an abstraction into a set of exhaustive, mutually-exclusive subcases. To make specifying such situations easier, the following declaration is syntactic sugar for a `cover` declaration and a `disjoint` declaration:

divide *object* **into** $object_1$, ..., $object_n$;

Using this declaration, we can compress the above `disjoint` and `cover` declarations as follows:

```
divide collection into empty_collection,
                       non_empty_collection;
divide non_empty_collection into singleton_collection,
                                 multiple_collection;
divide buffer into empty_buffer,
                   partially_full_buffer,
                   full_buffer;
disjoint empty_buffer, non_empty_buffer;
disjoint full_buffer, non_full_buffer;
```

We believe that adding the extra `disjoint`, `cover`, and `divide` declarations will not be too burdensome for the programmer. In addition to supporting static type checking of predicate objects with arbitrary predicate expressions, the declarations help to document the code. Furthermore, type declarations and type checking are optional in Cecil, helping to support both exploratory and production programming within the same language, and `disjoint`, `cover`, and `divide` declarations may similarly be omitted during exploratory programming.

Since fields are accessed solely through accessor methods, checking accesses to fields in predicate objects reduces to checking legality of messages in the presence of predicate objects, as described above. To ensure that fields are always initialized before being accessed, the type checker simply checks that the values of all fields potentially inherited by an object are initialized either at the declaration of the field or at the creation of the object. While this check is overly conservative (it does not take into account assignments to a field immediately after it comes into scope), it is sufficient, simple, and, we hope, not too restrictive.

Static type checking as described above ensures that a uniform interface is provided by an object, no matter what its state happens to be at run-time. An interesting, less

restrictive approach to type checking would allow different states of an object to have different interfaces. For example, a stack object might be subdivided into `empty_stack` and `non_empty_stack` predicate objects, but only the `non_empty_stack` predicate object would support a `pop` operation. Type checking with such state-dependent interfaces might require some more interesting analysis akin to typestate checking in the Hermes language [Strom & Yemini 86, Strom *et al.* 91] or would fall back on run-time checking for state-dependent operations. Lea is exploring a similar idea which he calls fine-grained types [Lea 92].

3.8 Implementation Strategies

A straightforward implementation of predicate objects is not difficult. Each object allocates enough space for any field it might inherit from a predicate object, and method lookup is augmented with additional evaluations of predicate expressions for methods attached to predicate objects. This approach has comparable performance to what the programmer would likely have implemented in the absence of predicate objects: methods would have extra tests and case statements in them to evaluate the requisite predicate expressions, and objects would include all instance variables that might end up being needed. We followed this strategy in our initial implementation of predicate objects in our Cecil interpreter. As of this writing, the dynamically-typed portion of predicate objects has been running for several months, but the extensions to the static type system have not yet been implemented.

Several techniques can be used to improve the performance of predicate objects:

- The system can attempt to evaluate predicate expressions eagerly and to cache the results of these evaluations. For most simple predicates, the system can determine that they have no side-effects and therefore will not need to be reevaluated upon each method lookup. Instead, the result of the predicate can be stored with the object. A straightforward way of caching the result of a predicate is in a hidden instance variable, but a more efficient way is by replacing the class[1] of the object with a special internal subclass that represents the outcome of evaluating the predicate. Method lookup for one of these internal subclasses would bypass evaluation of the predicate, consequently running just as fast as normal method lookup. To record the outcomes of multiple independent predicates, internal combination subclasses can be constructed lazily as needed. For attributes that may vary during the lifetime of an object, assignments to the attribute would change the object's class to a generic class. Method lookup on the generic class would first evaluate all necessary predicate expressions, change the class of the object to record the predicates' results, and then resend the message to the new class to invoke the proper method. Newly-created objects would start out as instances of the generic class.

  In the absence of predicate objects, programmers sometimes generate state-specific subclasses by hand. However, hand-written simulations become difficult to maintain with multiple independent predicates, and time-varying predicates are not amenable

1. In a classless language like Cecil, the implementation can maintain internal data structures that act like classes, as is done in the SELF implementation [Chambers *et al.* 89, Chambers 92a].

to this approach (since programmers normally cannot change the class of an object at run-time).

- Space for fields used solely as boolean or enumerated values governing selection of one of several possible predicate classes, such as the `iconified` attribute defined for `window` objects, can be reclaimed if the value of the attribute is encoded in the internal class of the object as described above. Similarly, if a predicate expression is constant for a particular object, space for fields associated with mutually exclusive (and so unreachable) predicates can be reclaimed.

These optimizations could make predicate classes *more* efficient than hand-written code not using predicate classes, since the system can perform optimizations, such as changing the class of an object dynamically, that the programmer could not emulate in most object-oriented languages.

4 Additional Examples

This section contains additional examples illustrating the usefulness of predicate objects in Cecil. Each example highlights some strengths and/or weaknesses of predicate objects not previously addressed.

4.1 Pattern Matching-Style Functions

Predicate objects can be used to emulate some of the functionality of pattern matching-based function definitions, as found in functional languages such as Standard ML [Milner *et al.* 90] and Haskell [Hudak *et al.* 92]. With pattern matching, a programmer can write multiple versions of a function, with the system automatically selecting the proper version of the function to call based on the dynamic value of the function arguments. Patterns can range from constants through partial descriptions of structured types through variable names which match any actual. For example, the following Haskell examples define some standard operations on numbers and lists:

```
-- return the sign of the argument
sign x | x <  0 = -1
       | x == 0 =  0
       | x >  0 =  1
```

```
-- map the unary function f over the argument list, returning a list of results
map f []     = []
map f (x:xs) = f x : map f xs
```

```
-- take a pair of lists of equal length and return a list of pairs
zip []     []     = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys
zip xs     ys     = error "cannot zip lists of unequal length"
```

```
-- reduce the non-empty argument list using the binary function f
reduce f []       = error "cannot reduce an empty list"
reduce f [x]      = x
reduce f (x:y:zs) = f x (reduce f (y:zs))
```

The ability of pattern matching to select the function to call based on the dynamic value or state of the argument is absent from most object-oriented languages. Using predicate objects, however, we are able to capture this finer sort of dispatching:

```
pred negative isa number when number < 0;[1]
pred zero     isa number when number = 0;
pred positive isa number when number > 0;
method sign(n@negative) { -1 }
method sign(n@zero)     {  0 }
method sign(n@positive) {  1 }

method map(f, c@empty_collection) { nil }
method map(f, c@non_empty_collection) {
    prepend(eval(f, c.first), map(f, c.rest)) }

method zip(c1@empty_collection, c2@empty_collection) { nil }
method zip(c1@non_empty_collection,
           c2@non_empty_collection) {
    prepend([c1.first, c2.first],   -- an array constructor
            zip(c1.rest, c2.rest)) }
method zip(c1@collection, c2@collection) {
    error("cannot zip lists of unequal length"); }

method reduce(f, c@empty_collection) {
    error("cannot reduce an empty list"); }
method reduce(f, c@singleton_collection) { c.first }
method reduce(f, c@multiple_collection) {
    eval(f, c.first, reduce(f, c.rest)) }
```

Pattern matching provides additional facilities not supported by predicate objects. For example, pattern matching allows names to be bound to subcomponents of an object, such as in the pattern `(x:xs)`, which binds `x` to the head of the list and `xs` to the tail of the list. Also, patterns are very concise and readable syntactically, while predicate objects are somewhat more verbose.

Predicate objects have several advantages over pattern matching, however:

- The methods written using predicate objects are more general than those written using concrete patterns. Any representation of collections satisfying the predicate associated with `empty_collection` can invoke the corresponding `map` method, for instance, not just the one empty list object. Similarly, patterns can only range over concrete data types, not abstract data types, although Wadler has proposed an extension of pattern matching to support abstract data types [Wadler 87].
- Predicate objects can be ordered in terms of specificity, and the system with use this information to determine the most appropriate implementation of a method to call. Functional languages typically try functions in the order they were defined, invoking the first version whose patterns match.
- Predicate objects give a name to the interesting condition and can be used to group functions that operate on the same condition, potentially increasing the readability of the resulting code.

---

1. Recall that in the predicate "`number < 0`", the name `number` is bound to the descendant of `number` being considered, i.e., the number itself.

- With predicate objects, dispatching based on value or state is integrated with dispatching based on dynamic type. Pattern matching cannot select the version of the function based on the dynamic type of the argument.
- Predicate objects can be extended with new specialized cases as the system evolves. The versions of a function defined by pattern matching often must be defined as a group, and they cannot be extended later with new alternatives without editing the original function definition. Pattern matching in dynamically-typed logic languages usually does not suffer from this limitation.

In part, these examples are easy to write because Cecil provides the ability to add methods to an object without needing to modify the object's definition. Traditional singly-dispatched languages cannot achieve this kind of easy extension because the methods of a class are part of the class definition, which would need to be edited to add new methods to the class. To compensate for the extra separation between objects and methods, Cecil relies on the programming environment to show a view of the program in which methods are directly associated with their specializing objects [Chambers 92b].

4.2 Attributes of People

Predicate objects can help organize code for an abstraction that can be considered to have multiple independent attributes and behavior that depends on the state of the attributes. For example, a `person` object might have several independent fields such as `sex` and `age`, and some of the methods on `person` might depend on the values of the fields:

```
object person;
field sex(@person);
field age(@person);
method bedtime(@person) { "10pm" }
method long_lived(p@person) { p.age > p.expected_lifespan }
method have_birthday(p@person) { p.age := p.age + 1; }

pred male isa person when person.sex = "male";
method expected_lifespan(@male) { 70 }

pred female isa person when person.sex = "female";
method expected_lifespan(@female) { 74 }

pred child isa person when person.age <= 12;
method bedtime(@child) { "8pm" }

pred teenager isa person
         when person.age >= 13 & person.age <= 19;
method bedtime(@teenager) { "12am" }

pred boy isa male, child;
pred girl isa female, child;

method make_person(sex, age) {
    object isa person { sex := sex, age := age } }
```

Predicate objects provide a direct way of associating behavior with particular values of the object's attributes. In a traditional language without predicate objects, the programmer must choose between two different ways of implementing an attribute and its connected behavior:

- If the attribute is constant, it can be represented by instantiating a specialized subclass of the `person` class. For example, the `sex` attribute could be replaced with `male` and `female` subclasses. The programmer then could factor state-specific methods into the appropriate subclass. Unfortunately, in the presence of multiple independent attributes, this strategy suffers from a combinatorial explosion of combining subclasses. Also, object creations must name the appropriate subclass statically, which can be awkward if the corresponding attribute is a computed expression rather than a constant.
- The attribute could be represented as an instance variable. Behavior dependent on the attribute must be written in one place and be sprinkled with tests of the attribute value. Programmers usually have no choice but to implement time-varying attributes this way.

With predicate objects, attributes can be implemented using the first approach without fear of combinatorial explosions and without excluding time-varying attributes. Attributes still can be manipulated like instance variables when convenient: new people are created as children of the generic `person` object irrespective of their sex or age, and attributes can be queried and modified directly.

A similar example is found in several papers on mixins and object-oriented programming: windows that may have titles and/or borders. Conventional approaches implement titled-window and bordered-window subclasses, plus their combination class. In Cecil with predicate objects, a single `window` object would be defined with `has_title` and `has_border` fields. Predicate objects inheriting from `window` and conditional on the presence or absence of titles and/or borders would contain the code responsible for the two independent extensions. One complexity with this design is that in Cecil there is no automatic method combination: if a window has both a title and a border, and both predicate objects define a method such as `display`, the programmer must explicitly provide a `titled_bordered_window` predicate object that overrides the two others and resolves the ambiguity, perhaps by calling both `display` methods sequentially. The difficulty arises because the titled and bordered attributes are not truly independent; they interact for displaying behavior.

4.3 Squares, Rectangles, and Polygons

A classic example of object-oriented programming is a hierarchy of geometric shapes, such as would appear in a drawing editor. For example, the following code implements a fragment of a standard hierarchy of graphical shapes:

```
object polygon isa shape;
field vertices(@polygon);
method draw(p@polygon) { ... }
method add_vertex(p@polygon, vertex) { ... }

object rectangle isa polygon;
method length(r@rectangle) { ... } -- compute from vertices
method set_length(r@rectangle, new_length) { ... }
method draw(r@rectangle) { ... }
method widen(r@rectangle, factor) {
    r.length := r.length * factor; }

object square isa rectangle;
method draw(s@square) { ... }
```

According to mathematical definitions, all squares are rectangles, and all rectangles are polygons, so this inheritance hierarchy is desirable from a modelling viewpoint. However, if the user invokes the `add_vertex` method (which modifies the polygon in place) on a rectangle, the object will no longer be a rectangle. Similarly, invoking the `widen` operation on a square will violate the specification of the square.

In the Eiffel community, the recommended solution is to *undefine* the `add_vertex` operation in the `rectangle` class and to undefine the `widen` operation in the `square` class, thus disallowing illegal modifications [Meyer 91]. However, this has two undesirable consequences. First, static type checking of operations in the presence of the undefine construct is quite difficult [Cook 89], leading to a complex, two-phase typing algorithm [Meyer 92]. Second, the drawing editor application either must realize that certain kinds of polygons cannot have vertices added to them (as must users of the application), or the editor must construct only polygons, forgoing any functionality and performance advantages of the more specialized subclasses.

With predicate objects, this example can be reimplemented with `rectangle` and `square` treated as predicate objects:

```
object polygon isa shape;
method is_rectangle(p@polygon) { ... }
...

pred rectangle isa polygon when polygon.is_rectangle;
method is_square(r@rectangle) { r.length = r.width }
...

pred square isa rectangle when rectangle.is_square;
...
```

Whenever a polygon satisfies the restrictions of rectangles or squares, the specialized implementations of the operations suitable to those kinds of objects are used. If a vertex is added to an object classified as a `rectangle`, it will be automatically reclassified as a general `polygon`. Non-predicate versions of `rectangle` and `square` are not needed. If an object inherits directly from the `square` predicate object, for instance, this informs the system that the object will always remain a square, and consequently the object will act just as if it inherited from a non-predicate version of `square`. The

implementation strategies described in section 3.8 can make the implementation just as efficient as if `square` were a normal non-predicate object.

4.4 Mutable Binary Trees

Predicate objects can be used to represent distinct behavior modes of an object. The window example from section 3.5 illustrates this application, where the two distinct behavior modes are expanded and iconified windows. As a second example, the code below implements mutable binary trees, where the distinct behavior modes are empty and non-empty trees:

```
object tree isa collection;
field is_empty(@tree) := true;

pred empty_tree isa tree when tree.is_empty;
method insert(t@empty_tree, x) {
    t.is_empty := false;
    t.left  := object isa tree; -- create a new, empty tree
    t.right := object isa tree;
    t.contents := x; }
method do(t@empty_tree, closure) {}

pred non_empty_tree isa tree when not(tree.is_empty);
field left(@non_empty_tree);
field right(@non_empty_tree);
field contents(@non_empty_tree);
method insert(t@non_empty_tree, x) {
    if(x < t.contents,    -- if(,,) is a user-defined control structure
        { insert(t.left, x); },
        { insert(t.right, x); }); }
method do(t@non_empty_tree, closure) {
    do(t.left, closure);
    eval(closure, t.contents);
    do(t.right, closure); }
```

All trees understand the `insert` and `do` messages, but the implementation of these two messages is completely different for the two behavior modes, and predicate objects allow the two modes to be factored apart. Additionally, the state specific to non-empty trees is associated only with the `non_empty_tree` object. Optimizations described in section 3.8 can eliminate the storage space for the `is_empty` field by creating two internal subclasses of `tree` and changing the internal "class pointer" of a tree instance to implement assignment to the `is_empty` field.

Much of this example could be implemented without predicate objects. One approach would make `empty_tree` and `non_empty_tree` normal subclasses of the `tree` class. However, this approach would preclude adding in place to an empty tree, since the class of the tree cannot change. Alternatively, a single `tree` class could be defined without state-specific subclasses, but this approach would sacrifice the factoring of code, require `is_empty` checks in the implementation of `insert` and `do`, and expose the `left`, `right`, and `contents` fields even in empty trees. The solution with predicate objects supports both state-based factoring of code and mutating trees in place from one state to another.

5 Related Work

5.1 Value-Based Dispatching in Other Object-Oriented Languages

Object-oriented languages support one kind of dynamic binding of messages to methods, where the method to run can depend on the run-time class or type of the message receiver (for singly-dispatched languages) or for some subset of the message arguments (for multiply-dispatched languages). A few object-oriented languages, such as CLOS [Bobrow *et al.* 88] and Dylan [Apple 92], can dispatch on the identity of an argument, but cannot easily dispatch on a more general condition of an argument, such as being a negative number or an iconified window; prototype-based languages are similar in this regard.

5.2 Sets and Polymethods in LAURE

The LAURE language is an unusual hybrid language with object-oriented, rule-based, and constraint-based features [Caseau 91, Caseau & Silverstein 92, Caseau & Perron 93]. Of particular interest is LAURE's ability to define sets of objects and to associate methods (called *polymethods*) with all members of a set. For example, the following two polymethods define LAURE's fibonacci function:

```
[define fib(x:{0,1}) polymethod => 1]
[define fib(x:{integer & {sign as +}}) polymethod =>
   fib(x - 1) + fib(x - 2)]
```

A set in LAURE can describe a fixed list of objects, all the members of a particular class, or all the objects having a particular attribute with a particular value. Sets can be combined using intersection to form new sets, much as multiple inheritance is used to combine classes, but LAURE sets also may be combined using the union and power-set operators. Methods can be associated with arbitrary set specifications, not just classes, as in the `fib` example. LAURE uses the specifications of the sets to automatically construct a lattice over the sets, ordered by set inclusion, and this lattice is used like an inheritance graph to resolve conflicts among methods whenever more than one method applies. A set specification is reevaluated whenever necessary to determine whether some object is currently a member of the specified set.

Sets in LAURE share many of the characteristics of predicate classes. Both describe the objects contained by (descended from) them, and this collection of objects can vary dynamically. Methods are attached to sets directly, as methods are attached to predicate classes. LAURE uses special kinds of inheritance operators to describe exhaustive or mutually-exclusive sets: a closed union implies that its subclasses are exhaustive, while a closed intersection somewhat counter-intuitively specifies that its superclasses are mutually exclusive.

LAURE's sets and predicate classes have some differences. Sets in LAURE may be specified using a fixed group of set construction operations and base sets, while predicate classes can be defined with arbitrary predicates. In LAURE, some of the specificity relationships among sets (the subsumption relation) is inferred automatically based on the structure of the set specifications, while all inheritance relationships among predicate classes must be specified explicitly.

5.3 Classifiers in Kea

The Kea language is a functional object-oriented language based on multiple dispatching [Mugridge *et al.* 91, Hamer 92]. Kea supports a notion of *dynamic classification* of objects. A class may be explicitly divided into a group of mutually-exclusive subclasses, and instances of the class can be classified into one of the disjoint subclasses. For example, a `List` class may be classified into `EmptyList` and `NonEmptyList` subclasses. Multiple independent classifiers may be used for any class. For example, a `Person` class may be classified into `Male` and `Female` subclasses as well as independent `Young`, `MiddleAged`, and `Old` subclasses. This approach avoids the need for creating a combinatorially-exploding number of combining classes (e.g., a `YoungMale` class, an `OldFemale` class, etc.), as these combination subclasses become implicit. The example in section 4.2 was inspired by a similar example presented in Kea.

Classifiers in Kea are similar to predicate classes. Both support automatic attribute-based classification of objects, and operations can be associated with the classified subclasses. Classifiers, however, appear to subdivide a class into a set of exhaustive, mutually-exclusive subclasses, with the particular subclass for an object determined either by the value of a single attribute (whose type must be some enumerated type) of the object or by explicit instantiation of a particular subclass. Predicate classes support arbitrary predicates and non-exhaustive and overlapping classifications, as illustrated by the `buffer` example in section 3.2. Since Kea is a functional language, it does not address the issue of an object whose classification varies over time.

5.4 Term Classification

Yelland developed an experimental extension of Smalltalk that supported *term classification* [Yelland 92]. Yelland introduced two new kinds of class-like constructs into Smalltalk: *primitive concepts* and *defined concepts*:

- Primitive concepts are used for explicit classification of objects. An object is a member of a primitive concept only when explicitly stated.
- Defined concepts are used for implicit property-based classification. An object is a member of a defined concept whenever its attributes (called *roles*) satisfy certain *role restrictions*. Only a few kinds of role restrictions are allowed, such as checking for an attribute being an instance of a particular class or concept, being within some integer range, or being an element of some fixed set. In return, Yelland's system will automatically compute the subsumption relationships among concepts (i.e., when one concept "inherits" from another) based on the structure of the role restrictions.

Methods and instance variables may be attached to both kinds of concepts just as with regular classes.

An object in Yelland's system may be a member of several independent defined concepts. Yelland's experimental system creates internal combination subclasses, and uses a single combination subclass to record that an object is a member of several independent concepts simultaneously. Since Smalltalk is imperative, an object's properties can change at run-time, and thus the object's classification can become out-

of-date. Yelland describes problems that can occur if an object is eagerly reclassified immediately when its state changes, such as when an object temporarily violates its role restrictions while its state is being updated. Consequently, in Yelland's system, objects are reclassified only when explicitly requested by the program.

Yelland's system is similar to LAURE and Kea in that the system is responsible for automatically determining the "inheritance" relationships among concepts, at the cost of limiting the form of the role restrictions. The predicate expressions of predicate classes can be any boolean-valued expression, at the cost of requiring explicit programmer declaration of the inheritance relationships. To avoid problems with eager reclassification of objects while keeping automatic reclassification in the system, Cecil re-evaluates predicate expressions lazily as needed to resolve method lookup. As described in section 3.8, an optimizing implementation might choose to track inheritance from predicate classes in other ways for faster method lookup.

5.5 Dynamic Inheritance in SELF and Garnet

SELF is a prototype-based language with a simple and uniform object model [Ungar & Smith 87, Hölzle *et al.* 91]. One consequence of SELF's uniformity is that an object's parent slots, like other data slots, may be assigned new values at run-time. An assignment to a parent slot effectively changes an object's inheritance at run-time. Consequently, the object can inherit different methods and exhibit different behavior. This *dynamic inheritance* allows part of an object's implementation to change at run-time. Dynamic inheritance has been used in SELF to implement mutable objects with several distinct behavior modes, such as binary trees with empty and non-empty states [Ungar *et al.* 91]. The example in section 4.4 was inspired by this use of dynamic inheritance in SELF.

The Garnet system [Myers *et al.* 92] includes a similar mechanism, also called dynamic inheritance but implemented differently, to effect wholesale changes in the implementation of an object's behavior. This feature has been used in Garnet to capture the significant changes in a user-interface object's behavior when switching between build mode and test mode in an application builder tool.

Predicate classes can emulate some of the functionality of dynamic inheritance as found in SELF or Garnet. Where a SELF program would have an assignable parent slot and a group of parent objects that could be swapped in and out of the parent slot, Cecil with predicate objects would have an assignable field and a group of predicate objects whose predicates test the value of the field. However, dynamic inheritance is more powerful than are predicate objects in Cecil. Predicate objects support associating state and behavior with possibly time-varying behavior modes of an object. Dynamic inheritance can do the same, but dynamic inheritance also allows an object to inherit from other run-time objects with their own run-time state; Cecil today only supports inheritance from statically-defined objects. Dynamic inheritance is rather unstructured, and often it is difficult to determine the behavior of an object with assignable parents, since any object conceivably could be assigned as a parent. The set of potential predicate descendants of an object, in contrast, are statically determined (at link-time), and we hope are easier to reason about. In those situations where predicate objects

provide sufficient functionality, we believe they are preferable to dynamic inheritance since the purpose and dynamic behavior of predicate objects is clearer.

A related mechanism is the `become:` primitive in Smalltalk-80[1] [Goldberg & Robson 83]. This operation allows the identities of two objects to be swapped, and so is more than powerful enough to change the representation and implementation of an object. The `become:` operation thus is even more powerful and unstructured than dynamic inheritance, and is likely to be at least as difficult to reason about if used extensively. Additionally, `become:` is difficult to implement efficiently without slowing down other basic operations of the system.

5.6 Other Related Work

Several other systems have constructs similar to aspects of predicate classes. Boolean classes [McAllester & Zabih 86], Exemplars [LaLonde *et al.* 86], and Clovers [Stein 91] all address the issue of forming automatic combination or union subclasses to avoid combinatorial explosion and better organize methods; in none of these systems is the classification based on an object's state, however. Some knowledge representation systems address many of the same issues as predicate classes, though usually more from a representation or modelling viewpoint than from a linguistic viewpoint. Many specification systems restrict the applicability of operations using preconditions and many concurrent systems allow operations to be conditional on guard expressions. Exception handling mechanisms share predicate classes' goal of factoring cases, although from an entirely different vantage point.

6 Conclusion

Predicate classes provide a descriptive and modelling capability absent from most object-oriented languages. Predicate classes identify and name important states or behavior modes of objects, describe relations among these behavior modes, and associate state and behavior with these modes. By factoring the implementation of a class into a group of state-specific subclasses, we hope to make code clearer and easier to modify and extend. Predicate classes complement normal classes, providing a form of automatic property-based classification that is compatible with the explicit classification supported by normal classes.

Predicate classes enable programmers to resolve the tension between representing state as data and representing state through subclasses. Programmers can factor state-dependent behavior into specialized subclasses without incurring the maintenance headaches caused by a combinatorial explosion of multiple, independent subclasses and without restricting the state represented by the subclass to be immutable or creating difficult type-checking problems. Predicate classes support clean solutions to existing "benchmark" problems such as representing multiple attributes of people and representing hierarchies of mutable geometric shapes.

Predicate objects are an adaptation of the general idea of predicate classes to the Cecil language. Predicate objects can be associated with arbitrary time-varying

1. Smalltalk-80 is a trademark of ParcPlace Systems, Inc.

predicates defined over the state of an object. The relationships among predicate objects are specified explicitly by the programmer through inheritance declarations and `disjoint`, `cover`, and `divide` declarations; these declarations help method lookup find the most specific method and help the static type checker suppress spurious type errors. Predicate objects can have methods and fields associated with them just like normal objects, helping to integrate predicate objects into the rest of the language.

Several areas of predicate objects in Cecil need further study. Interesting interactions between predicate objects and inheritance and fields were described earlier in the paper. Predicated methods appear to generalize the idea of predicate objects to groups of objects. Strategies for efficient implementation of predicate objects need to be implemented and measured. Finally, predicate objects encourage a new kind of type checking to be investigated where the interface exported by an object depends on its current state.

Predicate classes would probably be easy to incorporate into other object-oriented languages in a similar fashion, although multiple inheritance appears to be required and the ability to add methods to a previously-declared predicate class would be helpful. We believe that the potential increased expressiveness of predicate classes and their easy integration within other object-oriented programming models merits further experimentation and study.

Acknowledgments

We thank Alan Borning, Miles Ohlrich, Jeff Dean, Kevin Sullivan, Stuart Williams, Christine Ahrens, Doug Lea, and the anonymous reviewers for their helpful comments on earlier drafts of this paper. This research has been generously supported by a National Science Foundation Research Initiation Award (contract number CCR-9210990), a University of Washington Graduate School Research Fund grant, and several gifts from Sun Microsystems, Inc.

References

[Apple 92] *Dylan, an Object-Oriented Dynamic Language*. Apple Computer, April, 1992.

[Bobrow *et al.* 88] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, D. A. Moon. Common Lisp Object System Specification X3J13. In *SIGPLAN Notices 23(Special Issue)*, September, 1988.

[Caseau 91] Yves Caseau. An Object-Oriented Language for Advanced Applications. In *Proceedings of TOOLS USA '91*, 1991.

[Caseau & Silverstein 92] Yves Caseau and Glenn Silverstein. Some Original Features of the LAURE Language. In *Proceedings of the OOPSLA '92 Workshop on Object-Oriented Programming Languages: The Next Generation*, pp. 35-43, Vancouver, Canada, October, 1992.

[Caseau & Perron 93] Yves Caseau and Laurent Perron. Attaching Second-Order Types to Methods in an Object-Oriented Language. In In *ECOOP '93 Conference Proceedings*, Kaiserslautern, Germany, July, 1993.

[Chambers *et al.* 89] Craig Chambers, David Ungar, and Elgin Lee. An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes. In *OOPSLA '89 Conference Proceedings*, pp. 49-70, New Orleans, LA, October, 1989. Published as *SIGPLAN Notices 24(10)*, October, 1989. Also published in *Lisp and Symbolic Computation 4(3)*, Kluwer Academic Publishers, June, 1991.

[Chambers & Ungar 91] Craig Chambers and David Ungar. Making Pure Object-Oriented Languages Practical. In *OOPSLA '91 Conference Proceedings*, pp. 1-15, Phoenix, AZ, October, 1991. Published as *SIGPLAN Notices 26(10)*, October, 1991.

[Chambers 92a] Craig Chambers. *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages.* Ph.D. thesis, Department of Computer Science, Stanford University, report STAN-CS-92-1420, March, 1992.

[Chambers 92b] Craig Chambers. Object-Oriented Multi-Methods in Cecil. In *ECOOP '92 Conference Proceedings*, pp. 33-56, Utrecht, the Netherlands, June/July, 1992. Published as *Lecture Notes in Computer Science 615*, Springer-Verlag, Berlin, 1992.

[Chambers 93] Craig Chambers. The Cecil Language: Specification and Rationale. Technical report #93-03-05, Department of Computer Science and Engineering, University of Washington, March, 1993.

[Cook 89] W. R. Cook. A Proposal for Making Eiffel Type-Safe. In *ECOOP '89 Conference Proceedings*, pp. 57-70, Cambridge University Press, July, 1989.

[Goldberg & Robson 83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, Reading, MA, 1983.

[Hamer 92] John Hamer. Un-Mixing Inheritance with Classifiers. In *Multiple Inheritance and Multiple Subtyping: Position Papers of the ECOOP '92 Workshop W1*, pp. 6-9, Utrect, the Netherlands, June/July, 1992. Also available as working paper WP-23, Markku Sakkinen, ed., Dept. of Computer Science and Information Systems, University of Jyväskylä, Finland, May, 1992.

[Hölzle *et al.* 91] Urs Hölzle, Bay-Wei Chang, Craig Chambers, Ole Agesen, and David Ungar. *The SELF Manual, Version 1.1.* Unpublished manual, February, 1991.

[Hudak *et al.* 92] Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. *Report on the Programming Language Haskell, Version 1.2.* In *SIGPLAN Notices 27(5)*, May, 1992.

[LaLonde *et al.* 86] Wilf R. LaLonde, Dave A. Thomas, and John R. Pugh. An Exemplar Based Smalltalk. In *OOPSLA '86 Conference Proceedings*, pp. 322-330, Portland, OR, September, 1986. Published as *SIGPLAN Notices 21(11)*, November, 1986.

[Lea 92] Doug Lea. Personal communication. December, 1992.

[McAllester & Zabih 86] David McAllester and Ramin Zabih. Boolean Classes. In *OOPSLA '86 Conference Proceedings*, pp. 417-428, Portland, OR, September, 1986. Published as *SIGPLAN Notices 21(11)*, November, 1986.

[Meyer 91] Bertrand Meyer. Static Typing for Eiffel. In *An Eiffel Collection.* Technical report #TR-EI-20/EC, Interactive Software Engineering, Goleta, California, 1991.

[Meyer 92] Bertrand Meyer. *Eiffel: The Language.* Prentice Hall, New York, 1992.

[Milner *et al.* 90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, Cambridge, MA, 1990.

[Mugridge *et al.* 91] W. B. Mugridge, J. G. Hosking, and J. Hamer. Multi-Methods in a Statically-Typed Programming Language. Technical report #50, Department of Computer Science, University of Auckland, 1991. A later version published in *ECOOP '91 Conference Proceedings*, Geneva, Switzerland, July, 1991.

[Myers *et al.* 92] Brad A. Myers, Dario A. Giuse, and Brad Vander Zanden. Declarative Programming in a Prototype-Instance System: Object-Oriented Programming Without Writing Methods. In *OOPSLA '92 Conference Proceedings*, pp. 184-200, Vancouver, Canada, October, 1992. Published as *SIGPLAN Notices 27(10)*, October, 1992.

[Stein 91] Lynn A. Stein. A Unified Methodology for Object-Oriented Programming. In *Inheritance Hierarchies in Knowledge Representation and Programming Languages*, John Wiley & Sons, 1991.

[Strom & Yemini 86] Robert E. Strom and Shaula Alexander Yemini. Typestate: A Programming Language Concept for Enhancing Software Reliability. In *IEEE Transactions on Software Engineering 12(1)*, pp. 157-171, January, 1986.

[Strom *et al.* 91] Robert E. Strom, David F. Bacon, Arthur P. Goldberg, Andy Lowry, Daniel M. Yellin, Shaula Alexander Yemini. *Hermes, A Language for Distributed Computing.* Prentice Hall, Englewood Cliffs, NJ, 1991.

[Touretzky 86] D. Touretzky. *The Mathematics of Inheritance Systems*. Morgan-Kaufmann, 1986.

[Ungar & Smith 87] David Ungar and Randall B. Smith. SELF: The Power of Simplicity. In *OOPSLA '87 Conference Proceedings*, pp. 227-241, Orlando, FL, October, 1987. Published as *SIGPLAN Notices 22(12)*, December, 1987. Also published in *Lisp and Symbolic Computation 4(3)*, Kluwer Academic Publishers, June, 1991.

[Ungar *et al.* 91] David Ungar, Craig Chambers, Bay-Wei Chang, and Urs Hölzle. Organizing Programs without Classes. In *Lisp and Symbolic Computation 4(3)*, Kluwer Academic Publishers, June, 1991.

[Wadler 87] Phillip Wadler. Views: A Way for Pattern Matching to Cohabit with Data Abstraction. In *Proceedings of the Fourteenth ACM Conference on Principles of Programming Languages*. Munich, Germany, January, 1987.

[Yelland 92] Phillip M. Yelland. Experimental Classification Facilities for Smalltalk. In *OOPSLA '92 Conference Proceedings*, pp. 235-246, Vancouver, Canada, October, 1992. Published as *SIGPLAN Notices 27(10)*, October, 1992.