CS4215—Programming Language Implementation

Martin Henz and Chin Wei Ngan

Sunday 8th January, 2017

Chapter 11

oPL: A Simple Object-oriented Language

> "My guess is that object-oriented programming will be in the 1980s what structured programming was in the 1970s. Everyone will be in favor of it. Every manufacturer will promote his products as supporting it. Every manager will pay lip service to it. Every programmer will practice it (differently). And no one will know just what it is." *Tim Rentsch in [?]*

In this chapter we explore essential aspects of object-oriented programming languages in a top-down approach. Section 11.1 gives a view of object-oriented programming from the perspective of knowledge representation and argues that object-oriented programming supports a number of important abstraction principles. Section 11.2 covers aspects of software development. Here the notions of late binding, inheritance and encapsulation play a central role.

11.1 Knowledge Representation View

Software is used to solve problems in given domains. To this aim, software expresses properties of entities, their relationship and

interaction in a language accessible to automatic treatment such as compilation to processor instructions. The complexity of a given domain of application must be matched by the expressivity of the language in use. To understand a complex problem it is necessary to view it from different angles provided by abstraction principles. An often quoted hallmark of object-oriented programming is its support for the knowledge representation principles of aggregation, classification, and specialization [?].

Aggregation This principle allows to form new concepts as collections of other concepts. For example, vehicles such as semitrucks are entities composed

of parts such as cabin and trailer, each of these again being composed of wheels, axles, etc. In programming, aggregation is achieved by compound data structures that are called objects in the framework of object-oriented programming. The components are called attributes and can be referred to by attribute identifiers. During the lifetime of an object, attributes may change but usually the object structure, which means the names through which attributes are accessible, is fixed.

In imPL, we may use either records or tuples for aggregation. In the case of record, a car object with attribute `MaxSpeed` is represented as follows.

```
let myVehicle = [MaxSpeed: 85] in ... end
```

Classification This principle aims at grouping things together into classes such that common properties of the members can be identified. For example, it is useful to classify all individual participants of a road traffic scenario as vehicles that have properties like size, speed and direction of movement. Collectively, the instances of a class form the extension of that class. Object-oriented languages provide support for classification by allowing to define classes that describe the properties of their instances.

Functions can serve to provide classification. We can define a function `newVehicle` that constructs a new vehicle object `myVehicle` as follows.

```
let newVehicle = fun ms -> [MaxSpeed: ms]
in
   ...
   let myVehicle = (newVehicle 85)
   in
      ...
   end
end
```

Specialization This principle allows to describe a concept as a more specific version of another concept. A concept `Cs` can be regarded as a specialization of another concept `C` if the extension of `Cs` is a subset of the extension of C. This relationship is often called is-a in the context of object-oriented programming [**?**]. For example, concepts such as “car” and “truck” can be seen as specializations of the concept “vehicle”.

The following function `newCar` constructs car objects by specializing vehicle objects.

```
let newCar = fun mp -> let c = (newVehicle 95)
                       in c.MaxPassengers := mp; c
                       end
            end
in ...
   let myCar = (newCar 5)
   in ...
```

```
   end
end
```

To make the presentation of object-oriented concepts as simple as possible, we assume in this chapter that record property assignment adds the given property to the record if it does not have the property yet. In this way, the expression `c.MaxPassengers := pm` adds a new property `MaxPassengers` to the record `[MaxSpeed: 95]`, resulting in the record

```
[MaxSpeed: 95, MaxPassengers: 5]
```

In object-oriented programming, specialization can be achieved by defining classes as specialized versions of other classes using inheritance. A class `Cs` that inherits from another class `C` is called its subclass and `C` is called superclass of `Cs`. However, inheritance as provided by most object-oriented programming languages is more general than specialization.

In particular, properties of a superclass can be overridden by a subclass. We shall see in Section 11.2.4 that the identification of inheritance with specialization is the source of much confusion in object-oriented programming.

When it comes to the design of a particular formalism such as an object-oriented programming language, interactions between these abstractions emerge. For example, the principle of aggregation suggests that classes describe the attributes of their instances and that these attributes are inherited.

11.2 Software Development View

Over the years, a host of object-oriented analysis and design methods have been proposed (for overviews of this field see [**?**, **?**]). Usually issues like classification, aggregation and specialization play a central role in object-oriented analysis. The objects involved in a computation are identified and their properties are described. These issues can be characterized as static. Dynamic aspects come into play when the functionality of these objects is designed.

11.2.1 Focusing on Objects

In conventional programming languages, functions are a central means to structure functionality. At runtime, control can be passed from one function to another by function application. The resulting control flow in conventional languages is depicted in Figure 11.1. Many languages allow to structure functions according to their functionality, leading to the concept of modules.

A central idea behind object-oriented programming is that such structuring can be effectively guided by the data involved in the computation. At any point in time there is one dedicated data object *on which* the respective function is carried out. This current object is referred to as “this”. In object-oriented programming, functions are called methods. The invocation of a method can either change “this” to another object or leave it the same. In the former case,

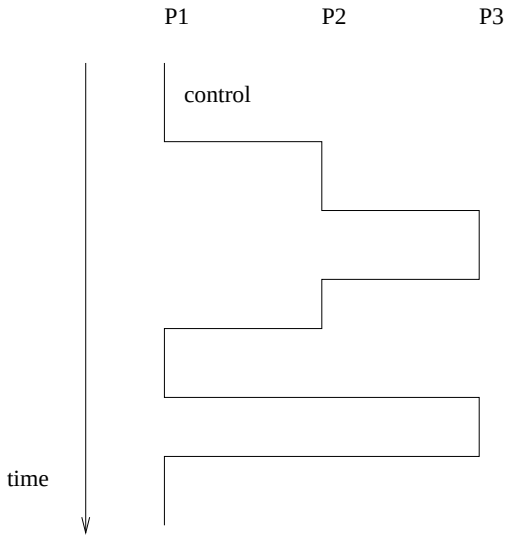

Figure 11.1: Control Flow in Functional Languages (time centered)

we say that the object is applied to a message (object application); often this is called message sending. The latter case can be achieved either by a special case of object application, called "this" application, or by method application; we shall discuss these two possibilities in Section 11.2.3. In all cases, the operation leads to execution of a corresponding method. The resulting refinement of the functional control flow is depicted in Figure 11.2. It is convenient to group the methods that operate on a certain kind of object into classes. Classes are modules containing methods that all operate on the same kind of objects, which are called its instances.

Each instance of a class carries its own identity distinguishing it from all other objects. We call this approach to equality *token equality* as opposed to *structural equality* which defines two objects to be equal if they have the same structure and all their components are equal.

It is useful to group the values that can be referred to by identifiers into types. With respect to a given type structure, an identifier occurrence is polymorphic, if it can refer to values of different type at different points in time. Cardelli and Wegner [**?**] distinguish between two ways an operation can be applied to a polymorphic identifier occurrence. Either the operation can be performed uniformly on all values of the different types (e.g. we can compute the length of a list regardless of the type of its elements), or different operations will be performed for different types (an addition $x + y$ works differently if x and y are integers or floats). Operations of the former kind are called universally polymorphic and the latter ad-hoc polymorphic. Both kinds of polymorphism play a central role in object-oriented programming.

Object-oriented languages handle both kinds of polymorphism by using late

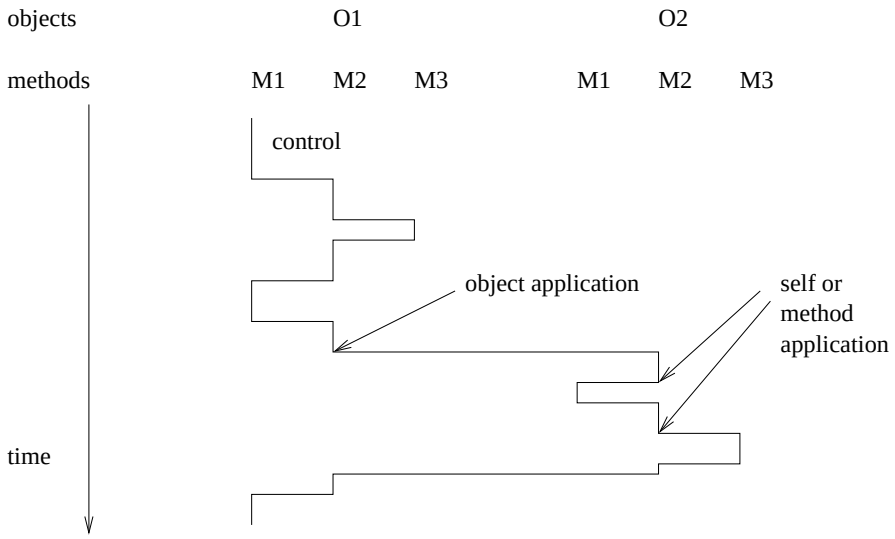

Figure 11.2: Control Flow in Object-Oriented Languages (time centered)

binding for object application. Late binding introduces an indirection between the object application and execution of the corresponding method. An object is applied to a message which consists of a method identifier and further arguments. The method identifier and the class of the object being applied determine the method to be executed. Thus the class provides a mapping from method identifiers to methods. The other arguments are simply passed as arguments to the method. Figure 11.3 depicts the execution of an object application (using Java notation).

The object-oriented extension of Lisp, CLOS [**?**], generalizes this scheme and allows all arguments to be considered for determining the method, which is therefore called multimethod. Late binding supports universal and ad-hoc polymorphism, since application of objects of different classes can lead to execution of the same or different methods as we shall see.

Polymorphism can be the source of programming errors, because the programmer may not be fully aware of the argument types of identifiers. Statically typed programming languages limit the polymorphism before the program gets executed by refusing programs that violate certain typing rules at compile time. Statically typed object-oriented languages usually introduce a type for every class. Polymorphism is restricted along the inheritance relation. Often the programmer can rely on the following invariant. If an identifier may hold instances of a class `C`, it may also hold instances of a class that inherits from `C`. This invariant is important in practice and is a central issue in defining type systems for object-oriented languages [**?**].

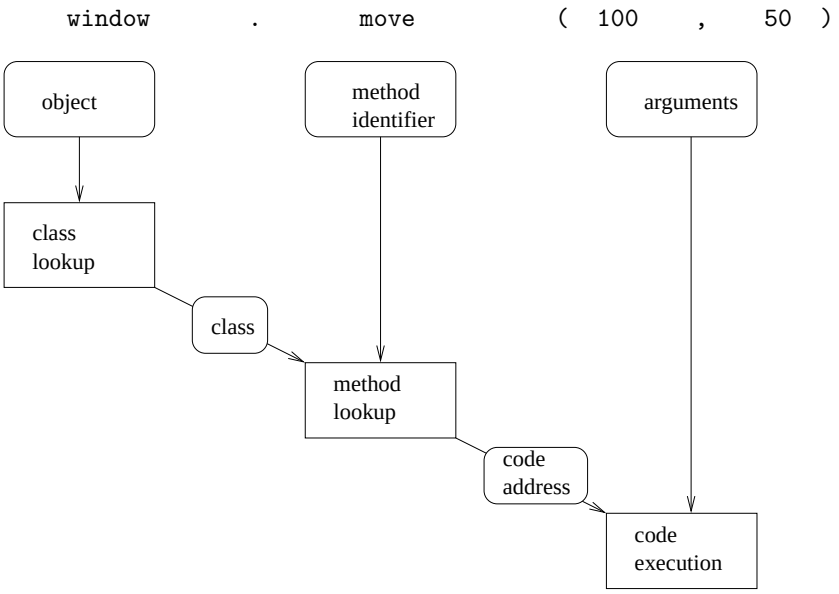

Figure 11.3: Execution of Object Application

11.2.2 Code Reuse

Often during development and maintenance of software, new functionality needs to be provided in addition to supporting old functionality. For example, a window system may provide for simple windows. In the next version of the software, labeled windows need to be supported in addition to simple windows. Without inheritance, the programmer can either "copy-and-modify" the code, or introduce conditional statements where the two kinds of windows must be distinguished. Both schemes lead to a proliferation of code, but not of programming productivity. Inheritance allows to reuse code more elegantly (but at the expense of certain intricacies as we shall see).

Conservative Extension Let us first consider the possibility of code reuse by conservatively adding functionality. In order to provide for labeled windows, we define the class `LabeledWindow` by inheriting from Window and adding methods such as `setLabel` and `drawLabel`. Instances of `LabeledWindow` provide all functionality that instances of `Window` provide and in addition more specialized behavior related to their label. Thus, the class `LabeledWindow` is a specialization of `Window`. Late binding provides the mechanism with which instances of the subclass can access the functionality of the superclass.

Non-conservative Extension Most object-oriented languages provide for overriding inherited methods in sub-classes by declaring that a method identifier refers to a new method in the subclass instead of an inherited method

with the same identifier. An invocation of an instance of the subclass using this identifier will lead to execution of the new method.

As an example, let us assume that the class Window supports a method `redraw` that displays the content that the window currently holds. In our class `LabeledWindow`, we would like to redefine the method `redraw` such that it also displays the current label of the window.

Overriding is a powerful and potentially dangerous tool in the hands of the programmer. It is powerful since it allows to reuse code and radically change it along the way. It is dangerous, because the code being reused may not be prepared for the change. Overriding is the issue where an object-oriented language departs from the idea that inheritance models specialization, because in the presence of overriding, a superclass does not necessarily characterize properties of instances of subclasses. Overriding is a heatedly debated feature of object-oriented programming. Taivalsaari [**?**] gives an excellent introduction to the literature in this field.

11.2.3 Late and Early Binding

> "Broadly speaking, the history of software development is the history of ever-later binding time." *Encyclopedia of Computer Science [**?**]*

Late Binding We saw that object-oriented programming allows a subclass to override a method inherited from a superclass. An application of an instance of the overriding class will result in a call to the new method. We saw in Section 11.2.1 that—apart from defining interfaces to objects—methods are used for structuring functionality similar to functions in functional languages. A method may pass control to another method without changing "this". One problem emerges here. What happens to the calls to the overridden method issued by methods in the superclass? For example, a method `deiconify` defined in Window may call the method redraw that we override in `LabeledWindow`. In a framework in which methods call each other directly, the old method will remain in use by methods of the superclass, thus contradicting the user's intention to completely replace it by the new method.

A particular usage of late binding can solve this problem. If we arrange that "this" is applied to the message `redraw` in the `deiconify` method of `Window` instead of calling the method directly, late binding will lead to execution of the new method `redraw`. If we insist on using late binding for every call of the method in the superclass, we can completely override it in a subclass. With "this" application, a programmer can open his code for change. On the other hand, "this" application in combination with overriding can make programs considerably more complex because it is not fixed by the programmer which code is being executed as result of the application. As Coleman and others [**?**] note "the increased re-usability of class hierarchies must be balanced against

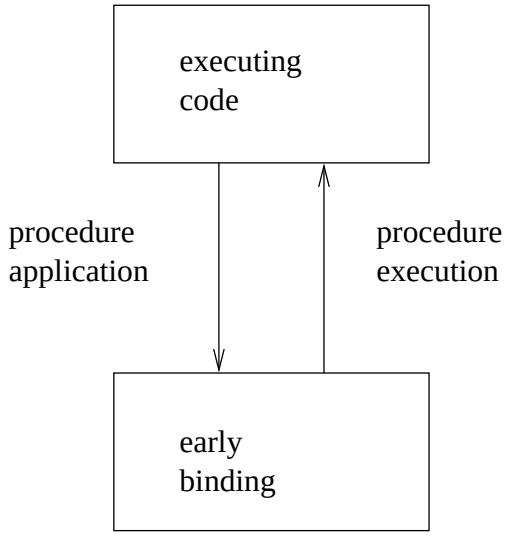

Figure 11.4: Control Flow in Functional Languages (state centered)

the higher complexity of such hierarchies" and later "on the one hand, [inheritance] enables developers to make extensive use of existing components when coping with new requirements; conversely, clients can be exposed to a source of instability that discourages them from depending on a hierarchy of classes".

Early Binding Late binding enforces the use of the method given by the class of the object being applied. Often, this is too restrictive. Consider the frequent case that an overriding method needs to call the overridden method. The use of late binding here would instead call the overriding method! Instead a mechanism is needed to call the overridden method directly. The classical idiom for this situation is the "super" call, which calls the method of the direct predecessor of the class that defines the method in which the call appears. A super call in a given method always calls the same method and thus implements early binding. The term "early binding" refers to the fact that the class, whose method matches the method identifier is known at the time of definition of the method in which the call occurs. A generalization of the super call is a construct that directly calls the method of a given class; we call this *method application*.

Early binding can be used to ensure the execution of a particular method. The programmer can limit the flexibility of designers of derived classes, and thus rely on stronger invariants. In practice, early binding is often used for efficiency reasons. Some object-oriented languages such as SIMULA [**?**] and C++ [**?**] treat early binding as the default and require special user annotations such as "virtual" for methods that may be overridden by descendant classes.

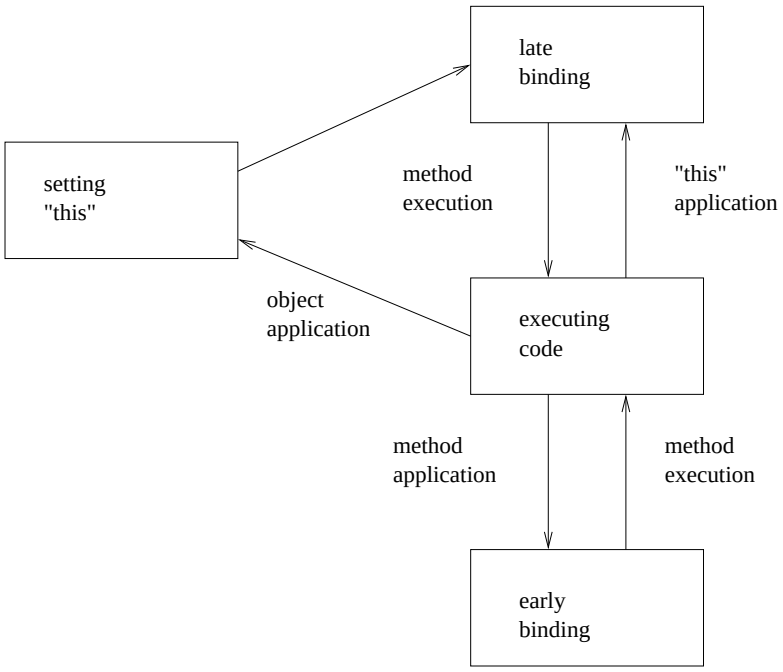

Figure 11.5: Control Flow in Object-oriented Languages (state centered)

Control Flow We saw that object application changes the current "this", "this" application does not change "this" and both use late binding, whereas method application uses early binding. We contrast this somewhat sophisticated control flow to the control flow in functional languages, which is depicted in Figure 11.4. In these languages, control flows from one function to the next through function application, with the possibility to pass parameters along.

The control flow in object-oriented languages is depicted in Figure 11.5. Method application corresponds to function application. General object application sets "this" and uses late binding whereas "this" application does not change "this", but also uses late binding.

11.2.4 Encapsulation

Software consists of different parts that interact with each other. Encapsulation allows us to confine this interaction to a specified interface. No interaction between the parts is possible unless this interface is used. Encapsulation is crucial to software development for several reasons:

Independent development. After the interfaces have been defined, different programmers can design and implement the individual parts.

Structure. Encapsulation forces a structure on the software that is often ben-

eficial for implementation and maintenance.

Change. The definition of interfaces can be done according to the expected rate of change. If the interfaces are stable over time, but the individual parts change frequently, then the encapsulation supports maintainability.

Encapsulation allows us to view aspects of one part of a software as internal and of no significance to other parts. As Ingalls remarks [**?**] "No part of a complex system should depend on the internal details of any other part."

Encapsulation for Attributes and Methods Methods often enjoy privileged access to the current object. We say the method is inside the current object and outside all other objects. For example, Smalltalk [**?**] generally allows access to attributes of only the current object.

Other languages allow to restrict the visibility of attributes statically. C++ [**?**] and Java [**?**] allow to declare attributes as private in which case they can only be accessed within the class in which the attribute was declared, or protected in which case they can be accessed additionally in all classes that inherit from this class (in Java additionally within the package of this class).

Object-oriented languages provide access to the current object such that it can be passed around in messages and stored in the state. For the current object the keyword `this` was introduced by SIMULA and adopted by Beta [**?**], C++ [**?**], and Java [**?**]. Smalltalk uses the keyword `self`. Other languages force (CLOS [**?**]) or allow (Objective Caml [**?**]) user variables to play the role of "self".

The languages C++ and Java allow to statically restrict the visibility of methods similar to attributes. Object and method application can be limited by declaring methods `private` or `protected`. Private methods can be accessed only within the defining class and protected methods additionally in all classes that inherit from it.

11.3 A Simple Object System in imPL

We have seen in Section 11.1 that the aggregation aspect of objects can be represented in imPL by records. In this section, we refine this view by adding classes and methods. The idea is that a method is a function with "this" as extra argument, and a class is a record of methods.

For example, a stack class can be represented in imPL as follows.

```
let stack = [Push: fun this x ->
                      this.Content := x :: this.Content
                   end,
             Pop:  fun this ->
                      let top = this.Content.First
                      in
                         this.Content := this.Content.Second;
```

```
                        top
                     end
                end,
          Makeempty:
                fun this ->
                   this.Content := [];
                   this
                end
          ]
in ...
end
```

A new instance of the stack can now be created by the following expression.

```
let mystack = (stack.Makeempty []) in ... end
```

Note that record property assignment adds a new property to the empty record. We can push and pop numbers as in the following expression.

```
(stack.Push mystack 1);
(stack.Push mystack 2);
(stack.Pop mystack) + (stack.Pop mystack)
```

This expression first pushes the numbers 1 and 2 onto the stack, then pops and adds them.

11.4 First-class Properties

In order to implement late binding and inheritance, we need to extend the semantics of imPL by adding first-class properties. The idea is to add property names as values, and allow property access and assignment to operate on values instead of statically given properties.

We denote properties as usual with q, and add the following rules to the syntax of oPL.

$$\frac{}{q} \qquad \frac{E_1 \qquad E_2}{E_1 \,.\, E_2} \qquad \frac{E_1 \qquad E_2}{E_1 \texttt{ hasproperty } E_2}$$

Recall that in imPL, record property assignment was only allowed to change properties, if they are already present in the given record. For example,

```
let r = [A: 1, B: 2]
in  r.C := 3
    ...
end
```

would fail in imPL, according to the denotational semantics described in the previous chapter. However, for oPL, it will be convenient to modify the semantics in such a way that record property assignment adds the given property if it is not present. In oPL, therefore, the record property assignment in the program above will execute successfully and the program will be indistinguishable from

```
let r = [A: 1, B: 2, C: 3]
in  ...
end
```

We call properties in oPL "first-class" because they can appear anywhere, where expressions are allowed, for example as as arguments and return values of functions. The following function makes use of first-class properties and the modified record property assignment.

```
let addProperty = fun r p v -> r.p := v end in ... end
```

The function application `(addProperty [A: 1, B: 2] C 3)` adds the property `C` to the given record.

11.5 Late Binding in imPL

The object system in Section 11.3 implements early binding. In order to perform an operation on an object, the method to be applied needs to be explicitly retrieved from the class. How can we modify the object system to achieve late binding?

The idea is to have a property in each object that contains its class. For that we add a function `new` which is from now on used all for object creations.

```
let new = fun theClass -> [Class: theClass] end in ... end
```

Instead of creating a stack by calling `Makeempty`, we insist on using `new` for all object creations as in the following program.

```
let mystack = (new stack) in ... end
```

After the object is created, the initialization method `Makeempty` can add the object fields, in this case the field `Content`.

```
(stack.Makeempty mystack)
```

Since every object knows its class, we can now implement late binding through the following function `lookup`.

```
let lookup = fun object methodname ->
                object.Class.methodname
             end
in ...
end
```

Note that here we make use of first-class properties. Instead of early binding, we can now use late binding for operations on objects as follows.

```
( (lookup mystack Push) mystack 1 )
```

"This" application is done in this object system by passing `this` as first argument to a method. For example, the following method `Pushtwice` performs two "this" applications.

```
let stack = [Push: fun this x ->
                      this.Content := x :: this.Content
                   end,
             Pop:  fun this ->
                      let top = this.Content.First
                      in
                         this.Content := this.Content.Second;
                         top
                      end
                   end,
             Makeempty:
                   fun this ->
                      this.Content := [];
                      this
                   end,
             Pushtwice:
                   fun this x ->
                      ( (lookup this Push) this x );
                      ( (lookup this Push) this x )
                   end
            ]
in ...
end
```

11.6 Inheritance

The final addition to our object system is the concept of inheritance. We achieve inheritance by adding an extra property `Parent` to classes that indicate their parent class.

```
...
let stackWithTop = [Parent: stack,
                    Top: fun this -> this.Content.First end]
in ...
end
```

The class `stackWithTop` extends the `stack` class by adding the method `Top` to the methods defined by `stack`.

Correspondingly, we need to modify the `lookup` function to access inherited methods as follows.

```
let lookupInClass = recfun lookupInClass theClass methodname ->
                        if theClass hasproperty methodname
                        then theClass.methodname
                        else (lookupInClass theClass.Parent methodname)
                        end
                    end
in
   let lookup = fun object methodname ->
                   (lookupInClass object.Class methodname)
                end
   in ...
   end
end
```

Note that this design allows for non-conservative extension. The new class can override inherited methods and thereby non-conservatively change the semantics of its instances compared to parent instances.

11.7 Object-oriented Syntax for oPL

Although object-oriented programming is in principle possible using the idioms described in the previous sections, it is in practice useful to support these idioms with an intuitive syntax. To this aim, we introduce the following abbreviation for methods and classes. The syntax

$$\texttt{method}\ q\texttt{(}x_1 \cdots x_n\texttt{)->}E\ \texttt{end}$$

is an abbreviation for the association

$$q\texttt{:fun this}\ x_1 \cdots x_n\texttt{->}E\ \texttt{end}$$

We denote such associations by the letter M. Using these associations, we introduce the following abbreviations for classes.

$$\texttt{class}\ M_1\ \cdots M_n\ \texttt{end}$$

stands for

$$\texttt{[}\ M_1,\ \cdots, M_n\texttt{]}$$

and

$$\texttt{class extends}\ x\ M_1\ \cdots M_n\ \texttt{end}$$

stands for

$$\texttt{[Parent:}x,\ M_1,\ \cdots, M_n\texttt{]}$$

Finally, we introduce the following abbreviation for object applications.

$$E\,.\,q(E_1 \cdots E_n)$$

stands for

`let obj =` E `in ((lookup obj` q`) obj` $E_1 \cdots E_n$`) end`

With this syntax in place, we can now write our stack class as follows.

```
let stack =
    class method Push(x) -> this.Content := x :: this.Content
          end
          method Pop()   -> let top = this.Content.First in
                                this.Content := this.Content.Second;
                                top
                            end
          end
          method Makeempty()
                         -> this.Content := []; this
          end
          method Pushtwice(x)
                         -> this.Push(x);
                            this.Push(x)
          end
    end
in
   let stackWithTop =
       class extends stack
             method Top()   -> this.Content.First
             end
       end
   in
      let myStackWithTop = (new stackWithTop)
      in  myStackWithTop.Makeempty();
          myStackWithTop.Push(1);
          myStackWithTop.Push(2);
          myStackWithTop.Push(myStackWithTop.Pop() + myStackWithTop.Pop());
          myStackWithTop.Top()
      end
   end
end
```

11.8 Implementation of oPL

Since oPL is just a syntactic extension of imPL, the most obvious implementation of oPL translates oPL classes to their corresponding imPL code as described

in the previous section. The resulting imPL program is then implmented using an interpreter or compiler as described in previous chapters.

However, the efficient implementation of records described in the previous chapter is not possible for oPL, since properties are first-class values, and new properties can be added to records at runtime. Thus a virtual machine-based implementation of oPL needs to revert to representing records using hashtables, mapping property names to the stored values. Existing languages such as Java avoid hashing for object fields through a type system that takes account of the class hierarchy.

In addition to representing objects themselves, the efficient implementation of object-oriented languages faces another challenge, namely the efficient implementation of late binding. It lies in the nature of late binding that the target function can only be determined at runtime. However, the process of determining the target function can be sped up through specific optimizations.

Consider the function `lookup` given in Section 11.6. The function `lookup` needs to follow the ancestor line of the class of the given object until a class is found that contains a method under the given property. Since late binding is encouraged in object-oriented languages, this process of method lookup can become a bottleneck in the implementation of object-oriented languages.

Some virtual machine-based implementations of object-oriented languages provide specific support for an efficient implementation of `lookup`. As an example, we shall study a technique called *inline caching* [**?**]. In this optimization, the compiler translates `(lookup obj` q`)` to a machine instruction sequence

```
LD <index of obj>
LDPS q
LOOKUP <class heap address> <cache>
```

Here, the `LOOKUP` instruction has two extra parameters that each can accommodate a heap address. Initially, the `LOOKUP` instruction proceeds as given in Section 11.6, looking up what class in the ancestor line of the class of the argument object has a method under property `q`. It then stores the heap address of class of the argument `obj` in the provided slot `<class heap address>`, and the heap address of the lookup result in the slot `<cache>`. Subsequent invocations of `lookup` first compare the heap address of the class of the current argument `obj` with the stored address in `<class heap address>`. If these addresses are the same, there is no need to conduct the lookup; we can immediately return the address stored in the `<cache>` slot. Only if the addresses are not the same, the lookup proceeds up the ancestor line of the class of the current argument object.

The rationale behind this optimization is that the actual argument objects may change frequently between invocations of `lookup`, whereas the classes of those argument objects change much less frequently. This rationale has been confirmed by statistics on real-world programs [**?**]. Since the technique uses the machine code as the place to cache the lookup result, it is called *inline caching*.

In an implementation of this optimization on a machine with automatic memory management, care must be taken that garbage collections flush the

caches at appropriate times such that `LOOKUP` always returns valid heap locations.