Object-oriented Programming with Gradual Abstraction

Kurt Nørmark
Department of Computer Science,
Aalborg University, Denmark
normark@cs.aau.dk

Lone Leth Thomsen
Department of Computer Science,
Aalborg University, Denmark
lone@cs.aau.dk

Bent Thomsen
Department of Computer Science,
Aalborg University, Denmark
bt@cs.aau.dk

Abstract

We describe an experimental object-oriented programming language, ASL2, that supports program development by means of a series of abstraction steps. The language allows immediate object construction, and it is possible to use the constructed objects for concrete problem solving tasks. Classes and class hierarchies can be derived from the objects - via gradual abstraction steps. We introduce two levels of object classification, called weak and strong object classification. Strong object classification relies on conventional classes, whereas weak object classification is looser, and less restrictive. As a central mechanism, weakly classified objects are allowed to borrow methods from each other. ASL2 supports class generalization, as a counterpart to class specialization and inheritance in mainstream object-oriented programming languages. The final abstraction step discussed in this paper is a syntactical abstraction step that derives a source file with a syntactical class form.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Classes and objects.

General Terms Languages, Design, Experimentation.

Keywords ASL2, objects before classes, weak and strong classification of objects, abstraction steps, Scheme.

1. Introduction

This paper describes a continuation of our work on *computational abstraction steps* [35] and the experimental ASL programming language. The main idea in our work is to allow the programmer to start an object-oriented programming process by constructing concrete objects, without having to deal with classes and inheritance. The objects can be used to solve a particular problem. After having solved the problem, which motivated the programming process in the first place, it is possible to derive concrete and more abstract classes on top of the objects.

The introduction of classes at a late point in time is seen as a *consolidation of the program* that supports additional development on a more general ground. The development approach proposed in this paper turns the class-based object-oriented programming process upside down. Instead of starting the programming process with general classes, which are specialized to classes that can be used to create objects, which eventually can be used to solve the problem that motivated all of this, ASL2 reverses the process.

Object-oriented programming without classes is not a new idea. Already in the first OOPSLA proceedings in 1986, Henry Lieberman's seminal paper *Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems* [22] outlined the idea of object-oriented programming without classes. In Lieberman's work, a prototype is an object that represents default behavior. An object can access default behavior by delegating messages to the prototype. The use of prototypical objects is seen as an alternative to representing shared objects in classes. In the mid-eighties a number of existing programming languages (Lisp-based languages, Smalltalk derivatives, and SELF) relied on the ideas discussed by Lieberman. More recently ECMAScript [2], JavaScript[4] and ActionScript [1] have adopted the prototype based object-oriented approach. In Section 7.4 we will review this work, together with elements from contemporary programming languages that support direct creation of objects.

The starting point of our current research is rooted in two different, but slightly overlapping observations:

- **A pragmatic observation**: It is interesting to initiate many practical problem solving tasks with the construction of concrete objects that gradually evolve to more abstract objects representing concepts and shared properties.
- **A pedagogical observation**: It is difficult and confusing to grasp the idea of object-oriented programming; If classes are needed before objects can be created; If the more general classes must be defined before specialized classes; And if a source program containing the definition of a class hierarchy is needed before any concrete object can be utilized.

In the following sections we introduce the ASL2 approach as a truly object-first approach to object-oriented programming. In each section we present realistic programming scenarios that illustrate sample problem solving based on a number of simple objects.

The programming language used in the rest of this paper is called ASL2, to distinguish it from the version of ASL used in and earlier paper [35]. ASL2 uses Scheme syntax, and it relies on a Scheme-based implementation that allows ASL2 to plug into the existing pool of native Scheme functions for various mundane problem solving steps.

2. Non-classified objects

As the start of an object-oriented programming endeavor it is natural to make some concrete objects, which take part in a real problem solving process. We do not aim at creating a number of toy objects which are discarded at an early point in time. Rather, the objects

DLS'12, October 22, 2012, Tucson, Arizona, USA.

that are created at the very beginning encapsulate state and behavior, which gradually will be brought into the final abstractions later in the development process. This way of working justifies our use of the word ”abstraction”, which signifies a change from concrete and tangible phenomena to concepts at a higher level.

2.1 Concepts

A *non-classified object* is an encapsulation of data (fields) and functions (methods). An object is created in terms of the data fields, which carry the state of the objects. There is no attempt to relate the object to other existing objects, hence the term ”non-classified” applies. The object can be supplied with methods that utilize the object in a context, which initially may be narrowly oriented towards solving a particular problem. Methods added to non-classified objects may be seen as the first, modest abstraction step. It is also possible to add additional state to an object after the initial creation step.

Every object relies on common behavior from a particular pre-existing object called `Object`. This is true for non-classified objects as well as the two kinds of classified objects that will be described later. If an object receives a message, which is not understood, it propagates the message to `Object`. Typical methods in `Object` include ASL2 related meta functionality, such as `AddMember` (which adds a member to its receiver), `DeleteMember` (which deletes a member from the receiver), and `Clone` (which copies the receiver). In ASL2 it is possible, and often attractive, to add members to `Object` with the purpose of adapting ASL2 to particular needs. We will see examples of that in Section 2.2.

From the outside, it is only possible to interact with an object by means of message passing. An object with a field `f` supports, in the starting point, two methods `get-f` and `set-f`, which access `f`. These are *automatically generated getter and setter methods*.

From inside an object there is lexical access to the fields of the object. Thus, in a function which is added as a method in an object, some of the free names of the function will be bound to the fields of the object. This is a kind of dynamic binding of free names in the function. It is, however, not apparent which of the free names will be bound to field names. In order to remedy this problem, a function can be nested in a form called `with-fields`, which enumerates the names in the function body that must be connected to object fields.

Information hiding is of primary importance in the object-oriented programming paradigm. With use of information hiding it is possible to hide data representation details from other objects, thereby facilitating future local changes of the object representation. In ASL2 we handle information hiding by disabling the automatically generated getter and/or setter methods. The examples shown in Section 2.2 include a concrete example.

With the lack of traditional, declarative source program descriptions the overall composition of the ASL2 program is done by mutation of the existing objects. Some of these mutations are done by passing ASL2 related messages to the objects. In the current implementation of ASL2, top-level message passing takes place in an interactive read-eval-print loop (a command interpreter).

2.2 Examples

The examples of non-classified objects are taken from a development scenario that works with points in a two-dimensional plane. We first create a point object located at *(5, 6)*:

```
(define p1 (make-object 'x 5 'y 6))
```

The expression returned by `make-object` returns a reference to a new object. The parameters passed to `make-object` are name-value pairs (represented as a Lisp property list), where names must be Scheme symbols, and values can be of arbitrary types. Objects are not named, but in the following discussion we often use the name of a variable, which refers to an object, as an informal name of the object.

Following the construction of an object, such as `p1`, it is possible to add more data properties and function properties by sending messages to the object. Here we add a function to `p1`, which serves as a method in the object.

```
(send p1 'AddMember 'DistanceTo
  (with-fields '(x y)
    (function (otherPoint)
      (sqrt
        (+ (square (- x (send otherPoint 'get-x)))
           (square (- y (send otherPoint 'get-y))))))))
```

This adds a new method member `DistanceTo` to `p1`. Notice the use of `with-fields` which states that the free names x and y in the function must be connected to fields in an object. The `AddMember` method is located in `Object`, as described in Section 2.1. When used in a method, the variable `this` refers to the receiver of the `DistanceTo` method.

With the introduction of another object, the point `p2` defined by

```
(define p2 (make-object 'x -3 'y 2))
```

it is now possible to find the distance between `p1` and `p2`:

```
(send p1 'DistanceTo p2)
8.94427190999916
```

It is, however, not possible to send the `DistanceTo` message to `p2`:

```
(send p2 'DistanceTo p1)
No appropriate method with selector DistanceTo
```

The problem is that the `DistanceTo` method is only added to `p1`. Both `p1` and `p2` are non-classified objects, and therefore the method `DistanceTo` in `p1` is not available in `p2`. We deal with this annoying issue when we introduce weakly classified objects in Section 3. Until then, we may explicitly pass a copy of the `DistanceTo` method from `p1` to `p2`:

```
(send p2 'Borrow 'DistanceTo p1)
(send p2 'DistanceTo p1)
8.94427190999916
```

The `Borrow` method copies a member from one object to another. As such, the `Borrow` method is an example of a domain-independent method, which is potentially useful in many different contexts. Therefore we place `Borrow` in `Object`:

```
(send Object 'AddMember 'Borrow
  (function (member-name from-obj)
     (send this 'AddMember member-name
           (send from-obj 'GetField member-name))))
```

It simply gets the member from `from-obj` and adds it to the receiver of the `Borrow` message.

We add additional methods to `p1`, which allows “polar access” to the point:

```
(send p1 'AddMember 'get-angle
  (with-fields '(x y)
    (function ()  (atan y x))))

(send p1 'AddMember 'get-radius
  (with-fields '(x y)
    (function ()
      (sqrt (+ (* x x) (* y y))))))
```

We may now decide to disable ”rectangular access” to `p1`, hereby creating the illusion that `p1` is represented in terms of polar coordinates. With this decision the rectangular coordinates should be private in `p1`:

```
(send p1 'PrivateField 'x)
(send p1 'PrivateField 'y)
```

Like the `Borrow` method, `PrivateField` is a method that easily can be defined in `Object`:

```
(send Object 'AddMember 'PrivateField
 (function (field-name)
   (send this 'AddMember (getter-name field-name)
         (function () (error "ERROR: ...")))
   (send this 'AddMember (setter-name field-name)
         (function (value) (error "ERROR ..."))))))
```

As it appears, we have hidden the `x` and `y` fields of `p1`. This is done by invalidation of the `get-x`, `set-x`, `get-y`, and `set-y` methods. We can now interact with `p1` in the following ways:

```
(send p1 'get-x)
ERROR: ...
(send p1 'get-y)
ERROR: ...
(send p1 'get-angle)
0.8760580505981934
(send p1 'get-radius)
7.810249675906654
```

The point `p2` is not affected at all by these interactions with `p1`.

The example from this section is continued in Section 5.2 in the scope of our discussion of class generalization.

3. Weakly classified objects

When we create an object we are typically aware of the role played by this particular object. The objects introduced in Section 2.2 were all envisioned as points, and therefore supposed to have shared access to "point-related properties". It is attractive to represent the *object role* explicitly when we make a new object, because objects of the same role have a lot in common.

3.1 Concepts

A *weakly classified object* is an object that carries an *object role*. In the context of ASL2, an object role is just a name, represented by a text string.[1] When a new object is created it is possible to associate the object with a role. The object role may be seen as a tiny piece of documentation, which reflects the thoughts and intentions of the programmer when the object is created. The role name can also be understood as a weak form of object classification, in the sense that all objects with the same role attached share a number of properties.

The most important benefit of working with weakly classified objects is the possibility of method sharing among objects of the same role. If a weakly classified object receives a message, which it cannot handle itself, it consults the objects of the same role to see if one of these related objects knows how to handle the message. If a "neighbor object" can handle the message, the method of the neighbor is used.

The method lookup process, as implemented by the ASL2 `send` primitive, goes through the following steps when an object `o` receives a message `m`:

1. If `o` contains a function named `m`, this function becomes the result of the method lookup.
2. If `o` is strongly classified (an instance of a class), the method is looked up in the class of `o` (and in its superclasses if necessary). This will be described in Section 4.
3. If `m` is a getter or setter name (of the form `get-f` or `set-f`) for an existing field `f`, getting or setting will be carried out.
4. If `m` is the name of a function in one of the objects, with the same role as `o`, this function is returned as the result of the method lookup.
5. As the last resort, a method lookup error occurs.

[1] In future versions of ASL the object role may be represented as an object itself, in order to organize role-related management via message passing to the role object.

Method borrowing, as used in step 4, is similar to applying the `Borrow` method in Section 2.2, but not identical to it. The `Borrow` method from Section 2.2 copies a method from one object to another. Step 4 in the method lookup process simply uses a method from an appropriate "neighbor", which belongs to the same role as the original message receiver. When such a method `m` is called, lexically accessed field names are bound to the receiving object `o`.

It may be the case that two different objects, of the same role, have different[2] methods of the same name. In that case, the borrowing is ambiguous, and it will not take place.

Method borrowing, in the sense discussed in this section, is an alternative to organizing shared methods in trait objects [38], from which a number of objects inherit. The creation and organization of shared trait objects would, to some degree, disrupt the work of the programmer. Recall from Section 2 that most objects will be created as part of a development process, which in the early phases is strictly narrowed to solving a particular problem. Using method borrowing, it is not necessary to introduce any additional "artificial" object for organizational purposes. It is enough to state the role played by each object being constructed, and to play the game that objects of the same role can share behavior by means of method borrowing.

The mechanism of method borrowing requires that we keep track of all objects that belong to a given role `r`. In the current ASL2 implementation, we maintain an organization that maps an object role `r` to the set of all objects of that particular role.

In a straightforward implementation of the method lookup process, borrowing of methods is not efficient. The reason is that all weakly classified objects with a given role must be searched linearly. In addition, all weakly classified objects must be kept away from the garbage collector, in order to prevent deletion of methods that are shared with other objects. Thus, in a naive implementation the price for programmer convenience may lead to a time consuming method lookup process and less effective object management. In ASL2, this is not a serious concern, because the use of weakly classified objects is an intermediate phase in the overall development process. As we will see in Section 4, weakly classified objects may eventually be converted to strongly classified objects (instances of traditional classes). Therefore, it is not critical if the method lookup is inefficient at an early stage of development. It is more important that the programmer is able to solve concrete problems in a natural way, based on objects that conveniently share some properties.

3.2 Examples

In this section we will give examples of weakly classified objects with the roles of `point`, `line-segment`, and `triangle`. The creation and use of the objects illustrate a geometric construction process, based purely on weakly classified objects. The end goal - the problem that causes us to create the objects - is to facilitate construction of triangles on which it is possible to calculate areas, circumferences, sides, angles, etc.

As in Section 2.2 we start with the construction of a few point objects.

```
(define p (make-object "point" 'x 5 'y 6))
(define q (make-object "point" 'x 8 'y -3.5))
(define r (make-object "point" 'x 0 'y -7.3))
```

If the first parameter passed to `make-object` is a text string (as opposed to a symbol which represents a field name), the text string represents the role of the new object. As in Section 2.2 we also add the `DistanceTo` method to `p` (not shown here).

[2] Method or function equality is not possible to implement in the general case. We use a simple syntactical equality criterion to decide if borrowing from neighbors is unambiguous.

```
(send t1 'AddMember 'Angle
  (function (cs)
    (let ((opposing-line-seg   (send this 'LineOpposing cs))
          (adjacent-line-seg-1 (send this 'LineOpposing (other-corner-1 cs)))
          (adjacent-line-seg-2 (send this 'LineOpposing (other-corner-2 cs))))
      (let ((a (send adjacent-line-seg-1 'Length))
            (b (send adjacent-line-seg-2 'Length))
            (c (send opposing-line-seg 'Length)))
        (radian-to-degree (acos (/ (+ (square a) (square b) (- (square c))) (* 2 a b))))))))
```

Figure 1. The `Angle` method, as added to the triangle `t1`.

Next, we introduce examples of line segments, each represented in a natural way by two end points:

```
(define line-p-q
  (make-object "line-segment" 'p1 p 'p2 q))
(define line-q-r
  (make-object "line-segment" 'p1 q 'p2 r))
```

We equip one of the line segments with a `Length` method

```
(send line-p-q 'AddMember 'Length
  (with-fields '(p1 p2)
    (function () (send p1 'DistanceTo p2))))
```

and we can ask for the length of both line segments

```
(send line-p-q 'Length)
(send line-q-r 'Length)
```

The first expression works because the `Length` method is explicitly defined in `line-p-q`. The second works because `line-q-r` borrows the `Length` method from the other line segment object. In addition, the `DistanceTo` method, which is called by the `Length` method, can be used on any object with role `point` by borrowing it from the point `p`.

With some minimal point and line segment functionality in place, we are ready to construct our first triangle:

```
(define t1 (make-object "triangle" 'c1 p 'c2 q 'c3 r))
```

As it appears, t1 is an object of role triangle with three corner fields (named `c1`, `c2`, and `c3`). The corner values of `t1` are the three existing points `p`, `q`, and `r`. Let us first define a `Circumference` method in `t1`:

```
(send t1 'AddMember 'Circumference
  (with-fields '(c1 c2 c3)
    (function ()
      (+ (send c1 'DistanceTo c2)
         (send c2 'DistanceTo c3)
         (send c3 'DistanceTo c1)))))
```

The next method creates the line segment opposing a given corner of the triangle. In this method a corner is represented by a symbol (one of `cs1`, `cs2`, and `cs3`).

```
(send t1 'AddMember 'LineOpposing
 (function (cs)
  (make-object "line-segment"
   'p1 (send this (getter-of (other-corner-1 cs)))
   'p2 (send this (getter-of (other-corner-2 cs))))))
```

A corner symbol is passed as a parameter to the `LineOpposing` method. `other-corner-1` and `other-corner-2` are Scheme functions that return names of the other corners relative to a given corner name. `getter-of` generates the name of an ASL2 getter method. As an example, the value of `(getter-of 'c1)` is the symbol `get-c1`.

Next we add an `Angle` method to `t1`, shown in Figure 1. Based on a given corner symbol `cs` the method encapsulates a possible, underlying geometric construction process. It constructs the three line segments of the triangle, calculates their lengths, and uses the law of cosines to calculate the angle of the corner. The method returns the angle in degrees.

The method relies on a few auxiliary functions, programmed in Scheme: `radian-to-degree` converts radians to degrees; `square` is the usual square function; `acos` the R5RS arccosine function; In the outer `let` the `Angle` method constructs the three line segments of the triangle. In the inner `let` the lengths of the line segments are calculated. The body of the inner `let` applies the law of cosines on the sides of the triangle.

In a similar way we may add a method to the triangle `t1` which calculates the area of the triangle (not shown here).

We observe that we now are able to work with triangles, and we can carry out most triangle calculations (finding angles, sides, area, circumference). The calculations are done in terms of intuitive, easy to follow geometric construction steps on points, line segments, and triangles. The triangle calculations are organized as methods located in the weakly classified objects. Using this simple machinery we can do useful problem solving in our domain. When we have solved our immediate problems we may decide to step up the ladder to the next abstraction level. Hereby we wish to consolidate the existing software, which at this point in the development process is scattered throughout a number of arbitrary concrete geometric objects.

The geometrical construction example is continued in Section 4.2 in the scope of our discussion of strongly classified objects.

4. Strongly classified objects

A weak classification of objects is useful in the early phases of a development process, where a few objects are used to solve a concrete problem. As explained in Section 3.1 the main asset of weak object classification is the sharing of methods among objects that belong to the same (weak) class. Objects in a given weak class may, however, easily drift apart. This can happen if new instance variables are added to selected objects in the same class. It can also happen if a method `M` ends up being defined differently (perhaps with different signatures) in two different objects, say in `o1` and `o2`. This causes `o1` and `o2` to behave differently when the message `M` is passed to them. In addition, it prevents other objects in the class from borrowing the method `M`, because the borrowing (as explained in Section 3.1) is ambiguous.

If the data type `DT` of a weakly classified object, such as `Point`, `Line-segment` and `Triangle` from Section 3.2, should survive in a longer lasting software development process it is valuable to ensure that objects in `DT` conform to each other in a stronger sense. This is the motivation for introducing strongly classified objects in ASL2.

4.1 Concepts

A *strongly classified object* is an instance of a class. A *class* prescribes and constrains all the instances to observe similar behavior. Instances of a strong class cannot drift apart. Instances of a class can be created by the `new` operation:

```
(new class 'field1 val1 ... 'fieldn valn)
```

In ASL2, a class is represented by a *class object*, which in most respects is an ordinary ASL2 object. All class objects share behavior from a particular object class `Class`, which in turn relies on the shared behavior of `Object` (as described in Section 2.1).

In ASL2 it is possible to create a class in the conventional way, by specializing an existing superclass:

```
(define class
  (make-class superclass 'field1 value1 ...))
```

This is a "concretization step" because it builds a specialized class on a more general class. As such, this mode of class creation is not of particular interest to the topic of this paper.

The derivation of a class from a number of objects is an important link in the gradual abstraction chain. A class object can be derived from one or more "ordinary objects", either non-classified objects or weakly classified objects. The derivation operation is called `kappa`[3], and it is used in the following way on a number of objects `o1`, `o2`, ... `on`:

```
(define class (kappa o1 o2 ... on))
```

The first object `o1` determines the *instance variables* of the class, and the *default values* of these instance variables. The subsequent objects `o2`, ... `on` are all required to possess at least the instance variables of `o1`. The methods of the class is the union of the methods in `o1`, `o2`, ..., `on`, as identified by method names.

The `kappa` operation transforms a set of objects, such as `o1`, `o2`, ..., `on`, to an instance of a new class (a class object). The methods are removed from the individual objects to the class object. The data fields of the individual objects `o1`, `o2`, ..., `on` are not affected by the `kappa` operation.

If only a single role *r* (as used for representation of weakly classified objects) is represented among the objects `o1`, `o2`, ... `on` in activation of `kappa` shown above, the role *r* is mapped to the new class. In addition, the involved objects are detached from their role, and therefore they are no longer weakly classified objects.

The *role to class mapping* affects the behavior of the `make-object` primitive in the following way:

> If the expression `(make-object r 'f1 v1 ... 'fm vm)` is used to create a weakly classified object of role `r`, and if `r` is mapped to a class `c`, then an appropriate instance of class `c` is created by `make-object`. Thus, `(make-object r 'f1 v1 ... 'fm vm)` will be equivalent to `(new c 'f1 v1 ... 'fm vm)`. With this interpretation, `make-object` is similar to a *factory method* [16] for production of `c` instances.

At any time it is possible to manually transform an object `o` to a strongly classified object (and an instance of a class `c`) provided that `o` conforms to `c`. The operation `as-instance-of-class` is used for that purpose. It is also possible to detach an instance from its class.

Once a class `c` has been defined, by derivation as explained above, instances of the class are forced to behave as prescribed by the class. This constrains an instance `o` of class `c` in the following ways:

1. It is not possible to add new members (data as well as methods) to `o`. As an exception, it is possible to add a method `m` to `o`, if `m` (identified by name and parameters) exists in `c`.
2. It is possible to delete a field `f` from `o`. The field `f` will exist in `c`, and the class field will act as the future default value of `f` in `o`.
3. It is possible to add and delete methods in `c`. Added methods in `c` immediately apply to objects such as `o`. Methods deleted from `c` can no longer be used on `o`.
4. It is possible to add a new field `f` with (default) value `v` to `c`. The `c` instance `o` is updated "lazily". If `f` eventually is accessed from `o`, the value `v` from `c` is returned. If `f` eventually is set in `o`, the field `f` is added to `o`, and assigned to the appropriate value. The "lazy updating of objects" is convenient, because there is not necessarily access to all instances of a class in ASL2.

The possibility of specializing an existing method on an particular object (as described in item 1) can be seen as a shortcut to making a singleton subclass of the class `c`. This ASL2 feature is similar to methods in CLOS specialized on singles objects [31].

The possibility of deleting a field `f` from an object `o` (as described in item 2) may seem to contradict the purpose of introducing classes altogether. But "deleting `f` from `o`" is really the same as stating "use the default value of `f` from the class of `o`". As such, deleting a field from an instance of a class `c` is a way of sharing a field between several instances of `c` (a reminiscence of *class variables* or *static variables* in `c`).

4.2 Examples

Our examples of strongly classified objects are build on top of the example from Section 3.2 where we created a number of objects weakly classified as points, line segments, or triangles. We ended up with a number of triangle methods for computation of sides, angles, area, and circumference.

Recall that the main agenda of our work with the weakly classified objects was to carry out geometric construction and calculation processes on concrete objects, with the purpose of solving particular problems. In the context of this section we are interested in consolidating the geometric concepts in classes that define and constrain future work with points, line segments and triangles. Thus, we are transitioning from a world with numerous weakly classified objects that individually carry their own methods to a situation with relatively few classes which contain almost all the longer lasting program details.

Based on the points `p`, `q`, and `r` from Section 3.2 we derive class `Point` from these three objects:

```
(define Point (kappa p q r))
```

This defines a class object `Point` with default `x` and `y` coordinates taken from `p`, (`x` = 5, `y` = 6), and with the `DistanceTo` method. All three existing points are transformed to be instances of class `Point`. In addition, the object role "point" of `p`, `q` and `r` is mapped to class `Point`. As a consequence, future applications of the factory method `make-object` will instantiate class `Point`. Therefore, the object `s1` created by

```
(define s1 (make-object "point" 'x 0 'y -3))
```

and the object `s2` created by

```
(define s2 (new Point 'x 0 'y -3))
```

are structurally equal to each other, and both instances of class `Point`.

[3] The name "kappa" is a reminiscence of the use of "lambda". Kappa is used in expressions that generate classes, in the same way as lambda is used in expressions that create functions.

It is crucial that `make-object` acts as factory method that can instantiate the new class. Without the underlying role-class mapping facility, the `LineOpposing` method described in Section 3.2 would create a line segment that cannot respond to the line segment methods, such as `Length`, as used in the `Angle` method. As a consequence, we would be forced to rewrite methods such as `LineOpposing`, to make use of the class instantiation primitive, called `new`, instead of the `make-object` facility.

We also derive the classes `Line-segment` and `Triangle` from existing and weakly classified objects:

```
(define Line-segment (kappa line-p-q line-q-r))
(define Triangle (kappa t1))
```

We are now in a situation where all the existing objects are constrained by the three new classes. All methods are shared, because methods are looked up in the classes.

It is not possible to add new fields or new methods to individual objects. If a field is deleted from an existing object `o`, it is interpreted as "use the default value of the field from the class of `o`". These observations are illustrated by the following interactions on point objects:

```
> (send p 'AddMember 'z 0)
It is not possible to add a data member to an
instance of a class.

> (send p 'AddMember 'DistanceToOrigo (function () ...))
It is only possible to add a method to an instance
of a class if the method is compatible with a
method in the class.

> (send q 'get-x)
8

> (send q 'DeleteMember 'x)

> (send q 'get-x)
5

> (send Point 'get-x)
5
```

If a field is added to a class, it becomes a field of all existing objects:

```
> (send Point 'AddMember 'z 0)

> (send p 'get-z)
0

> (send p 'set-z -7.5)

> (send p 'get-z)
-7.5

> (inspect p)
An instance of a class.  Instance members:
  x: 5
  y: 6
  z: -7.5
Class members:
  x: 5
  get-x: Automatic getter
  set-x: Automatic setter
  y: 6
  get-y: Automatic getter
  set-y: Automatic setter
  DistanceTo: Function(otherPoint)...
  z: 0
  get-z: Automatic getter
  set-z: Automatic setter
```

As a matter of future development of points, line segments and triangles in our program, it will take place by refining classes by means of generalization. When we work with strongly classified objects, programming in individual objects is either prevented, or forced to be aligned with the prescribed fields and methods of the classes.

5. Class Generalization

Given a situation with a number of classes, as generated from objects by the `kappa` operation, the next natural abstraction step is class generalization. Class generalization is the opposite of class specialization, which is provided by use of inheritance in most mainstream object-oriented programming languages.

5.1 Concepts

Based on a number of classes, say `c1`, `c2`, ... `cn`, it may be attractive to generalize these to a common superclass `sc`. Generalization of classes involves a refactoring of the classes relative to the new superclass, where some fields and methods are lifted to a new and higher level of abstraction.

ASL2 allows generalization of a number of classes `c1`, `c2`, ... `cn` with respect to an enumerated set of members (fields or methods) `m1`, `m2`, ... `mk`:

```
(define sc
  (generalize (list c1 ... cn) (list m1 ... mk)))
```

For notational convenience, the list of classes `c1`, `c2`, ... `cn` is called `C`, and the list of members `m1`, `m2`, ... `mk` is called `M`. The abstraction step behind the generalization of the classes in `C` to `sc` involves the following actions:

1. Create a new class that represents the superclass `sc`.
2. For each member `m` in `M` copy a representative of `m` from one of the classes in `C` to the new superclass `sc`.
3. For each class `c` in `C` delete those of the members in `c` that belong to `M`.
4. For each class `c` in `C` arrange that `sc` becomes its superclass.

As a potential complication, it may be the case that a method `m` in `M` is not present in all the classes in `C`. If, for instance, `m` is not present in the class `c`, the class `c` will be extended with a new method after the generalization step.

As another complication, it may be the case that a named method `m` is located in two or more of the classes in `C` in diverging versions. In that case it is difficult to select "the right one" to transfer to the new superclass. Instead of blocking the generalization step by an error message, the ASL2 processor assumes that an arbitrary choice of `m` will succeed. Therefore the `m` method, which is elevated from a class in `C` to `sc`, is selected randomly.

The ASL2 implementation decisions, mentioned above, have a couple of implications. First, if the classes in `C` do not contain "enough shared properties" the generalization may end up being too broad (enriching some of the subclasses with a number of new members). Second, if the classes in `C` contain different versions of "the same method", it is hard to predict the exact definition of the common superclass. We have decided to implement this rather liberal version of `generalize`, as opposed to a stricter version that rejects generalization in a number of situations. The current implementation works best - and most natural - in cases where the classes in `C` contain a homogeneous subset of members that can be factored into the new superclass.

5.2 Examples

In this section we will continue the example from Section 2.2 which involved rectangular and polar points. Let us assume that we have derived two classes, based on either non-classified or weakly classified objects:

```
(define RectangularPoint (kappa ...))
(define PolarPoint (kappa ...))
```

We will assume that both classes have polar getters, polar setters, rectangular getters, and rectangular setters. In addition, both classes are assumed to have a `DistanceTo` and a `Move` method. `PolarPoint` is also assumed to have a `Rotate` method.

We now generalize these two classes to a common superclass, `Point`:

```
(define Point
  (generalize (list RectangularPoint PolarPoint)
              (list 'Move 'Rotate 'DistanceTo)))
```

This causes a `Move` method and a `DistanceTo` method to be moved to `Point` from one of the subclasses. Because these methods are implemented in terms of the common getters and setters, the choice between the two is not critical. Also the `Rotate` method (implemented in terms of the polar getters and setters) is moved from `PolarPoint` to `Point`. This effectively extends the class `RectangularPoint` with the `Rotate` method.

We continue the example from above in Section 6.2 in the scope of generating syntactic class forms.

6. Source Class Generation

After a series of abstraction steps it is possible to end up with a hierarchy of classes. Each of the classes is represented by an ASL2 object (a so-called class object). Directly or indirectly, the members of the classes have been added individually, by passing messages, such as `AddMember`, to the classes (or to the objects that existed in earlier parts of the abstraction chain).

In this section we will finalize the abstraction process by generating syntactic forms for the classes in the class hierarchy. Thus, the last link in the abstraction chain is a *syntactic abstraction* step.

6.1 Concepts

In conventional development of object-oriented programs classes are written in source files, and all use of the classes is based on the descriptions in these files. Such "use" does - in most cases - involve concretization steps. This is exactly the opposite of the ASL2 approach explained in this paper, where each development step has been an abstraction step. The last abstraction step brings the ASL2 objects, that represent the classes, to the level of source files. These source files contain complete and self-contained descriptions of the classes and their members. Future development steps may therefore by carried out by editing the syntactic forms in the source files.

The operation that performs the syntactic abstraction is called `class-source`:

```
(class-source class-object class-name
              superclass-name file-path)
```

The syntactic form of the class in `class-object` is written to a text file, the path to which is given by the last parameter. It is necessary to introduce names of classes when we carry out the syntactic abstraction step. The second parameter, `class-name`, names the class referred by the first parameter, `class-object`. The third parameter names the superclass, which is referred by `class-object` if it has been generalized as described in Section 5. If class-object does not have a superclass, a distinguished "no-parameter-pass value" is passed instead of the superclass name.

Until this level, classes have been represented as objects, and as such accessed by references. As explained in Section 2.2, variables that hold these references may informally serve as object names or class names. But in reality, ASL2 objects, including class objects, are anonymous. In the abstraction step discussed in this section, names are finally attached to the classes.

An additional parameter may be passed to the `class-source` operation, namely the role name of objects, as used by the factory method `make-object` for creation of objects. This parameter is optional. Recall from Section 3 that role names were introduced at the abstraction level where we worked with weakly classified objects. The role name is mapped to the object that represents the class. With this mapping we are able to keep the calls to `make-object`, which uses the role name, hereby leaving method bodies intact, exactly as written by the programmer in earlier phases of the development process. Alternatively, we could have made a syntactic transformation of the method bodies, substituting calls of `make-object` with an activation of the `new` operation.

6.2 Examples

We will exemplify the syntactic abstraction steps, and the use of the `class-source` operation, by continuing the development of points (rectangular/polar) from Section 2.2 and Section 5.2. In Section 5.2 we ended up with objects that represent three classes in a class hierarchy. The classes are informally named `Point`, `RectangularPoint`, and `PolarPoint`. These three names are variables that refer to ASL2 class objects.

The following calls to `class-source` generate the syntactic forms of the three class objects:

```
(class-source Point "point" #f "point.asl2")

(class-source RectangularPoint "rectangular-point"
              "point" "rectangular-point.asl2"
              "rec-point")

(class-source PolarPoint "polar-point"
              "point" "polar-point.asl2" "pol-point")
```

The Scheme notion for boolean *false*, `#f`, is used as the no-parameter-pass value. `"rec-point"` and `"pol-point"` happen to be the role names of rectangular objects and polar objects respectively, when these were introduced as weakly classified objects earlier in the development process. (These details have not been shown in this paper).

The syntactic forms, as generated by `class-source`, are shown in Appendix A. We have manually improved the pretty printing (line breaks and indentations) of the classes and methods.

It is now possible to load the three classes from Appendix A. The three classes contain the important details - of lasting value - as developed in Section 2.2 and Section 5.2. The future use and development of points in ASL2 programs is supposed to be based on these classes.

7. Related work

We start the discussion of related work by comparing the work presented in this paper to the so-called *objects first* approach to object-oriented programming. After that, *gradual abstraction* is contrasted with *gradual typing*, as related to the body of work that introduces types at a late time in the development process. We also relate the ASL2 work to other languages and systems that do not use traditional syntactical means for definition of all program aspects. Finally, we review a number of programming languages that contain facilities that are related to ASL2.

7.1 Objects-first

As mentioned in the introduction, our current work was launched on pragmatic and pedagogical observations. Our starting point is the pedagogical observation that today many educators use an objects-first approach when teaching object-oriented programming [10, 15]. The objects first approach starts with object-oriented analysis and design which emphasizes partitioning system behavior into small, cohesive parts, and composing the final solution by making the parts cooperate. Many proponents of the objects first approach encourage students to create objects and to interact with them in an experimental and iterative development approach until the students reach an understanding of the implementation that is suitable for a class definition. The BlueJ development environment makes an attempt at supporting such a development process [9].

Mainstream object-oriented languages such as Java only support interacting with objects that have been created by instantiating classes. In reality, the objects first approach should be called the *classes first* approach, as the programmer has to identify and define the needed set of classes before instantiating such classes to objects. Our experience [24, 34] from teaching introductory programming courses shows that many novice programmers find the process of going from classes at a high abstraction level to concrete objects via inheritance and object creation extremely counter intuitive. We presume this is because going in the opposite direction, from objects to classes, is either not supported or only supported in limited and cumbersome ways by mainstream object-oriented programming languages [18].

Java allows creation of objects from anonymous classes and C# allows creation of objects of anonymous reference types. Both in Java and C# it is possible, using reflection and run-time code generation, to create the illusion of constructing a type hierarchy at run-time as demonstrated in one of our earlier papers [25]. These facilities in Java and C# do not, however, support introductory students in need of a more intuitive approach to developing object-oriented programs.

7.2 Gradual typing

A considerable amount of work has been devoted to studies of gradual introduction of static type information in source programs. Gradual typing has been studied in the context of scripting [36], functional programming [20, 28] and more recently also in the context of object-oriented programming [13, 29]. In recent years (2009, 2011, and 2012) an international workshop series called STOP (Scripts TO Programs) has been setup to encourage work in this area [39].

In general it is believed that the development speed is faster with the use of dynamically typed programming languages compared with statically typed languages. In addition, it is generally believed that the number of errors discovered late (or never) in a dynamically typed program is larger than for statically typed programs. It turns out, however, that it is difficult to confirm these beliefs in empirical research [19, 32].

With the use of gradual typing a programmer can start the programming process without having to decorate a program with type information. Furthermore, throughout the programming process gradual typing allows a programmer to mix static and dynamic type checking in a program. Gradual typing provides ways to control which parts of a program are statically checked and which parts are dynamically typed.

Although gradual typing is still a research topic, mainstream C#4.0 [11, 23] supports the notion of type `dynamic`. The type `dynamic` allows static and dynamic typing to be mixed, or rather allows dynamically typed components to live and interact with statically typed components albeit in a very limited and rudimentary form.

Gradual typing deals with the idea of gradual introduction of type information in the source program text. Gradual abstraction (as of this paper) deals with the idea of gradual introduction of more and more abstract object notations. Gradual typing represents an evolutionary approach, both seen relative to the software concepts involved, and seen as a trend on a larger scale. In contrast we consider gradual abstraction more like a revolutionary approach, which in several respects reverses the order in which we encounter some of the important programming artifacts. As such,we do not expect the ideas presented in this paper to enter the landscape of mainstream, industrial programming languages, in the near future, but as illustrated in [35] there are already application areas such as generating class hierarchies from data in XML or comma separated files.

It may be the case, however, that gradual abstraction can be used in niches, such as in pedagogical programming, in special-purpose programmable systems, and within selected application domains. More important, we find it worthwhile to demonstrate - both academically and practically - that it is indeed possible and operational to deal with an *upside down* object-oriented development process.

7.3 Object-based program development

Many programming steps in ASL2 can only be performed by sending program-development messages to an existing object (or by calling operations on a number of such objects). As an example, which has appeared several times in this paper, the only way to add a method to an object `o` is to send the `AddMember` message to `o`. Program-development messages (such as `AddMember` and `DeleteMember`) should be contrasted with application domain messages (such as `DistanceTo` and `Area` for geometric objects). There is no ASL2 syntax for adding a method to an object. In addition, there exists no ASL2 syntax for adding a method to a class prior to the derivation of the class source form in the last abstraction step (described in Section 6). With respect to this property ASL2 is similar to Smalltalk-80 [17].

In a paper about meta programming and reflection [14] Bracha and Ungar write that “Smalltalk-80 differs from most languages in that a program is not defined declaratively”. Instead, Smalltalk relies on the programmers interaction with objects that represent classes and other program related entities. Most Smalltalk and ASL2 objects respond to messages that are related to the development of the program as well as messages that are related to the application domain. Following the phrasing used by Bracha and Ungar, the kind of programming discussed here could be called non-declarative programming. But in order not to mess up with important classic computer science distinctions we prefer to use the term *object-based program development*. The point is, that some aspects of ASL2 are not described in a traditional textual program source. Rather, these aspects are captured as object state mutations, and triggered by the reception of program-development related messages.

In ASL2 the programmer can freely mix between program development code (i.e. code that builds the program) and application code. This is in contrast to the ideas in multi-staged programming, e.g. in MetaOCaml [5, 33], where application code and program development code is separated in stages and where the programming language has syntactic support for runtime program generation.

It would not be easy to base ASL2 program development on conventional source program text. The source text of an object-oriented program consists of class declarations, and ASL2 classes first appear when strongly classified objects are introduced. Recall from Section 6 that the generation of syntactical class forms is the end result of an ASL2 program development process - after a number of abstraction steps has been carried out. Thus, ASL2 relies to a large extent on object-based program development.

In this paper we have illustrated a relatively primitive elaboration of the object-based program development process using ASL2. The process relies on message passing (and function calls) in an interactive command interpreter, also known as a read-eval-print-loop (REPL). As an alternative, the object-based program development process may be managed by a specialized set of browsing tools, in the style of the browsers known from Smalltalk-80. Such programming environments have more recently been put forward in the Impromptu programming environment built around the Scheme language for programming on the Mac OS X [3] and to some extent in Visual Studio from Microsoft for programming in the F# programming language [6].

7.4 Related programming languages

As mentioned in the introduction of this paper, object-oriented programming without classes rests on some rather old ideas, most prominently outlined by Lieberman in the mid eighties [22]. At that point in time some Lisp-based languages [30] had been built based on these ideas. Also some Smalltalk related work was oriented in direction of "examplars" and "prototypes" [21]. Much territory was pioneered in SELF [37, 38], which was initiated in 1986. SELF organizes objects in multiple inheritance hierarchies, with emphasis on shared behavior in so-called trait objects.

More recently ECMAScript [2], JavaScript [4] and ActionScript [1] have adopted the prototype based object-oriented approach. In these languages objects are created by constructors. Each object implicitly references the constructor's associated prototype, and properties added to an object's prototype are shared by all objects sharing the prototype. Constructors are function objects that create and initialize objects. Constructors have their own prototypes, which the object prototype references and in this way implements inheritance hierarchies. Although we have not carried out the experiment, we are convinced that ASL2 could be encoded in JavaScript by relying on the above mentioned object model, the use of reflection and the eval construct.

Newer languages such as Scala [27] and Fortress [8] support object creation without instantiation from a class. But in these languages there are no constructs allowing an object to be classified as belonging to a class after it has been created. Once created a "class-less object" belongs to its own anonymous class subclassed by the top element in the class hierarchy. Furthermore, both Scala and Fortress are large general purpose languages vying for entering mainstream. As such they embrace many new complex concepts that hamper their suitability as candidates for languages used in teaching introductory programming.

The Grace language [12] under development by a large group of language educators headed by Andrew P. Black, Kim B. Bruce and James Noble, tries to address the issue of the lack of a good language for teaching introductory programming. The Grace language also takes the approach of allowing objects to be created directly. Objects can be abstracted over by methods, and it is easy to create factory methods producing objects of the same kind, instantiated with different initial values by parameters to the factory method. Classes in Grace are just syntactic sugar for such factory methods. Grace is dynamically typed, but constructs can optionally be annotated with type constraints which should be seen as assertions giving rise to run time type checks. Although conceptually Grace's class declarations can be understood in terms of a flattening translation to object constructor expressions that build the factory object, Grace does not have constructs that allow a programmer to write a program that realizes this construction.

8. Conclusions

ASL2 is an experimental programming language, which has been developed for a truly "objects-first approach" in the area of object-oriented programming. ASL2 is implemented in R5RS Scheme, in part by adaption of a meta circular interpreter [7]. ASL2 is tightly connected to the underlying implementation language, and therefore Scheme functions can be used from ASL2 for plain and common computations.

The most important results of our work with ASL2 are the following:

- The possibility of starting an object-oriented development process, by creating concrete objects that are equipped with methods that allow the programmer to solve real problems as *early* as possible. This can be seen as problem-solving with examples/prototypes of objects, which belong to classes that have not yet been established and fully shaped.
- The concept of a weak object classification that allows objects to borrow methods from each other. This provides for convenient method sharing among objects, without having to establish a particular "trait object" that holds the common functionality.
- Program development by gradual abstraction, in which more abstract objects are derived from less abstract objects. Less abstract objects are flexible for immediate problem solving purposes, but they are not suitable for shaping the program in the longer run. More abstract objects, such as class objects and their generalizations, constrain both existing and future objects in a direction that allows the programmer to reason about the object system.
- A final result, which is a number of syntactic class forms in source files. Instead of dealing with these descriptions at a high level of abstraction early in the development process, the class sources can be seen as the final product of an ASL2 development effort. A considerable amount of experience with instances of these classes has been fused into the development process, already at the moment where these class source files materialize.

ASL2 is an experimental language, far from having found its final design, and there is plenty of future work to be undertaken. For example some design choices, such as `generalize` nondeterministically choosing one implementation of a method from a set of classes with different implementations for a given method could be changed to a version where `generalize` is restricted to a binary operator giving preference to methods from the first argument. The current version could then be implemented by using `reduce generalize` on a list of randomly permuted classes.

One could also imagine a design where a list of methods is added in a second and separate stage, thus establishing a closer symmetry between `generalize` and `kappa`. Another design choice that could be investigated is the design of method lookup when a weakly classified object wants to borrow a method. Currently this requires a linear scan of all objects in the weakly classified set and the members of this set are only identified by a string. An alternative design could introduce a common object for weakly classified objects which could contain a list of all objects belonging to its set, and it could even contain a lookup table for methods to optimize the implementation further. This may also make it easier for a garbage collector to identify weakly classified objects suitable for garbage collection. When no more references exist to any of the objects in the set and to the common object, it is safe to garbage collect weakly classified objects.

A somewhat orthogonal direction involves the introduction of types in ASL2 where we expect the growing body of knowledge about gradual typing to be very relevant.

A more long term process is to validate the design experimentally to investigate the productivity gain from using our approach.

However, such studies are notoriously time consuming and difficult to undertake as reported by Hanenberg [19].

The ASL2 homepage [26] contains additional information, including the Scheme implementation of ASL2 and language reference API documentation. In addition, the homepage gives access to a number of videos that illustrate the development scenarios from this paper in more details.

Acknowledgments

We thank the anonymous reviewers for their constructive comments, especially the two reviewers who pointed out a mistake we made in the implementation of the `AddMember` function. We would also like to thank James Noble for comments helping us better understand subtle points about Grace.

References

[1] Actionscript home page, 2012. http://www.actionscript.org/.

[2] Ecmascript language specification, 2012. http://www.ecma-international.org/publications/standards/Ecma-262.htm.

[3] What is Impromptu?, 2012. http://impromptu.moso.com.au/.

[4] Javascript home page, 2012. https://developer.mozilla.org/en/About_JavaScript.

[5] Metaocaml home page, 2012. http://www.metaocaml.org/.

[6] Visual studio developer center, visual F#, 2012. http://msdn.microsoft.com/en-us/vstudio/hh388569.aspx.

[7] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, 1985.

[8] E. Allen, D. Chase, C. Flood, V. Luchangco, J.-W. Maessen, S. Ryu, and G. L. Steele. Project Fortress. *Linux Magazine*, September 2007.

[9] D. J. Barnes and M. Kölling. *Objects First with Java: A Practical Introduction Using BlueJ*. Prentice Hall, October 2002.

[10] J. Bennedsen and C. Schulte. What does 'objects-first' mean? an international study of teachers' perceptions of objects-first. In R. Lister and Simon, editors, *Seventh Baltic Sea Conference on Computing Education Research (Koli Calling 2007)*, volume 88 of *CRPIT*, pages 21–29, Koli National Park, Finland, 2007. ACS.

[11] G. Bierman, E. Meijer, and M. Torgersen. Adding dynamic types to c. In *Proceedings of the 24th European conference on Object-oriented programming*, ECOOP'10, pages 76–100, Berlin, Heidelberg, 2010. Springer-Verlag.

[12] A. P. Black, K. B. Bruce, and J. Noble. Panel: designing the next educational programming language. In *SPLASH/OOPSLA Companion*, pages 201–204, 2010.

[13] G. Bracha. Pluggable type systems. In *OOPSLA04 Workshop on Revival of Dynamic Languages*, 2004.

[14] G. Bracha and D. Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of OOPSLA 2004*, pages 331–344. ACM Press, October 2004.

[15] S. Cooper, W. Dann, and R. Pausch. Teaching objects-first in introductory computer science. *SIGCSE Bulletin*, 35(1):191–195, Jan. 2003.

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison Wesley, Reading, 1996. ISBN 0-201-63361-2.

[17] A. Goldberg and D. Robson. *Smalltalk-80 The Language and its Implementation*. Addison-Wesley Publishing Company, 1983.

[18] I. Hadar and U. Leron. How intuitive is object-oriented design? *Commun. ACM*, 51(5):41–46, 2008.

[19] S. Hanenberg. An experiment about static and dynamic type systems: doubts about the positive impact of static type systems on development time. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pages 22–35, New York, NY, USA, 2010. ACM.

[20] K. Knowles, A. Tomb, J. Gronski, S. N. Freund, and C. Flanagan. Sage: Unified hybrid checking for first-class types, general refinement types, and dynamic (extended report.), 2007. http://sage.soe.ucsc.edu/sage-tr.pdf.

[21] W. R. LaLonde, D. A. Thomas, and J. R. Pugh. An exemplar based Smalltalk. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPLSA '86, pages 322–330, New York, NY, USA, 1986. ACM.

[22] H. Lieberman. Using prototypical objects to implement shared behavior in object oriented systems. In *The proceedings of OOPSLA'86*, pages 214–223, 1986.

[23] Microsoft Corporation. The C# language specification version 4.0, 2010. http://www.microsoft.com/downloads/.

[24] K. Nørmark, L. Leth-Thomsen, and K. Torp. *Reflections on the teaching of programming*, chapter Mini Project Programming Exams, pages 229–243. Springer Verlag, LNCS 4821, 2008.

[25] K. Nørmark, B. Thomsen, and L. Leth-Thomsen. Mapping and visiting in functional and object-oriented programming. *Journal of Object Technology*, 7(7), September-October 2008.

[26] K. Nørmark, B. Thomsen, and L. L. Thomsen. The ASL2 home page, 2012. `http://people.cs.aau.dk/~normark/asl2/`.

[27] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala, Second Edition*. Artima Incorporation, USA, 2010.

[28] J. G. Siek and W. Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, September 2006.

[29] J. G. Siek and W. Taha. Gradual typing for objects. In *ECOOP'07: 21st European Conference on Object-Oriented Programming*, 2007.

[30] S. Slade. *The T programming language - A dialect of Lisp*. Prentice-Hall, 1987.

[31] G. L. Steele. *Common Lisp, the language, 2nd Edition*. Digital Press, 1990.

[32] A. Stuchlik and S. Hanenberg. Static vs. dynamic type systems: an empirical study about the relationship between type casts and development time. In *Proceedings of the 7th symposium on Dynamic languages*, DLS '11, pages 97–106, New York, NY, USA, 2011. ACM.

[33] W. Taha. A gentle introduction to multi-stage programming. In C. Lengauer, D. Batory, C. Consel, and M. Odersky, editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 30–50. Springer Berlin / Heidelberg, 2004.

[34] B. Thomsen. *Reflections on the teaching of programming*, chapter Using On-Line Tutorials in Introductory IT Courses, pages 68–74. Springer Verlag, LNCS 4821, 2008.

[35] L. L. Thomsen, B. Thomsen, and K. Nørmark. Computational abstraction steps. *Journal of Object Technology*, 9(6):1–23, November 2010.

[36] S. Tobin-Hochstadt and M. Felleisen. Interlanguage migration: from scripts to programs. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 964–974, New York, NY, USA, 2006. ACM.

[37] D. Ungar and R. B. Smith. SELF: The power of simplicity. *Lisp and Symbolic Computation: An International Journal*, 4(3):187 – 205, 1991.

[38] D. Ungar, C. Chambers, B. wei Chang, and U. Hölzle. Organizing programs without classes. In *Lisp and Symbolic Computation*, pages 223–242. Kluwer Academic Publishers, 1991.

[39] T. Wrigstad. StOP - internal workshop series on scripts to programs. http://wrigstad.com/stop/. http://wrigstad.com/stop/.

A. Source Programs

In this appendix we show the source programs generated by the `class-source` operation, as applied to the class objects `Point`, `RectangularPoint`, and `PolarPoint` from Section 5.2.

```
(class point ()
   (Rotate (function (da)
     (send this 'set-a (+ (send this 'get-a) da))))

   (DistanceTo (function (otherPoint)
     (sqrt
      (+
       (square (- (send this 'get-x) (send otherPoint 'get-x)))
       (square (- (send this 'get-y) (send otherPoint 'get-y)))))))

   (Move (function (dx dy)
      (send this 'set-x (+ (send this 'get-x) dx))
       (send this 'set-y (+ (send this 'get-y) dy))))
)


(class polar-point (point)
   (r 6.999999999999999)
   (a 0.39269952499999994)

   (get-x (function () (* r (cos a))))

   (get-y (function () (* r (sin a))))

   (set-x (function (new-x)
     (let ((existing-y (send this 'get-y)))
       (set! r (sqrt (+ (* new-x new-x) (* existing-y existing-y))))
       (set! a (atan existing-y new-x)))))

   (set-y (function (new-y)
     (let ((existing-x (send this 'get-x)))
       (set! r (sqrt (+ (* existing-x existing-x) (* new-y new-y))))
       (set! a (atan new-y existing-x)))))
)

(map-role-to-class! "pol-point" polar-point)


(class rectangular-point (point)
   (x 0)
   (y 0)

   (get-a (function () (atan y x)))

   (get-r (function () (sqrt (+ (* x x) (* y y)))))

   (set-a (function (new-a)
     (let ((existing-r (sqrt (+ (* x x) (* y y)))))
       (set! x (* existing-r (cos new-a)))
       (set! y (* existing-r (sin new-a))))))

   (set-r (function (new-r)
     (let ((existing-a (atan y x)))
       (set! x (* new-r (cos existing-a)))
       (set! y (* new-r (sin existing-a))))))
)

(map-role-to-class! "rek-point" rectangular-point)
```