Tampereen teknillinen yliopisto. Ohjelmistotekniikan laitos. Raportti 6
Tampere University of Technology. Department of Software Systems. Report 6

Antero Taivalsaari

Simplifying JavaScript with Concatenation-Based Prototype Inheritance

TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

Tampereen teknillinen yliopisto. Ohjelmistotekniikan laitos. Raportti 6
Tampere University of Technology. Department of Software Systems. Report 6

Antero Taivalsaari

Simplifying JavaScript with Concatenation-Based Prototype Inheritance

Tampereen teknillinen yliopisto. Ohjelmistotekniikan laitos
Tampere 2009

ISBN 978-952-15-2275-8 (PDF)
ISSN 1797-836X

Simplifying JavaScript with Concatenation-Based Prototype Inheritance

Antero Taivalsaari
Tampere University of Technology
Korkeakoulunkatu 1
FI-33720 TAMPERE, FINLAND
antero.taivalsaari@tut.fi

ABSTRACT
As a result of the ongoing paradigm shift towards web-based software, the JavaScript language has become one of the most widely used programming languages in the world. In this paper we propose some extensions to JavaScript in order to support concatenation-based prototype inheritance – an alternative model of object-oriented programming that was first introduced in the early 1990s.

1. INTRODUCTION
Back in the late 1980s and early 1990s, one of the most exciting research topics in the area of object-oriented programming was *prototype-based programming* [8]. Prototype-based languages such as NewtonScript [9, 10] and Self [2, 14] were discussed widely in conferences on object-oriented programming. Many of the innovations that are used in advanced software systems today – such as generational garbage collection and adaptive just-in-time compilation – were first introduced in the context of prototype-based programming languages about twenty years ago.

Towards the late 1990s, research into prototype-based programming withered and waned, as the focus in the software industry shifted to the Java programming language that had been introduced in the mid-1990s. For the next ten years, Java was the predominant programming language in the world, capturing the minds of many of the sharpest minds in the software industry and in the academic world.

As a result of the ongoing paradigm shift towards web-based software, the programming language that is receiving most attention today is *JavaScript* – a language that was once viewed as the inferior little sister of Java. The current popularity of JavaScript is partly accidental, arising from the fact that JavaScript is the only programming language that is supported by all the commercial web browsers. However, many qualities of the JavaScript language – such as its built-in reflective capabilities and the fact that JavaScript applications require no compilation or binaries – make JavaScript inherently better suited to web applications than more static, compiled programming languages such as C, C++ and Java. Therefore, it is not surprising that JavaScript has started displacing its sister language from its once dominant role especially in client-side web application application development.

From research perspective, one of the most interesting aspects of the JavaScript language is its object model. Unlike most object-oriented programming languages such as Smalltalk, C++ and Java that are based on a classical (class-based) object model, JavaScript uses a *prototype-based* object model. In a prototype-based language, there are no classes; instead, new types of objects are created by constructing prototypical objects that are then copied and extended to create further object types.

In a nutshell, the JavaScript programming language combines a prototype-based object model with a syntax that is instantly recognizable to C, C++ and Java programmers. The fact that JavaScript uses a very different object model went largely unnoticed as long as the language was used only for simple scripting tasks. However, now that JavaScript is commonly used for writing serious web applications that consist of tens or even hundreds of thousands of line of code, the object model of JavaScript has started to spark more discussion.

In this paper we propose some extensions to the JavaScript programming language in order to support prototype-based programming in an alternative way. This alternative model – concatenation-based (or replication-based) prototype inheritance – was proposed by the author originally in the early 1990s. The key argument in the paper is that with some relatively simple syntactic and semantic extensions, the expressive power of the JavaScript programming language can be increased without sacrificing compatibility with existing applications. It is also argued that the proposed alternative inheritance model can be more intuitive especially to novice programmers.

The structure of the paper is as follows. Section 2 starts with a quick introduction to class-based versus prototype-based object-oriented programming, followed by an overview of different prototype-based object models in Section 3. Section 4 provides an introduction to the prototype-based object model of JavaScript. Section 5 introduces our proposed extensions, followed by some discussion and comments in Section 6. Section 7 discussed related work. Finally, Section 8 summarizes the paper.

2. CLASSES VERSUS PROTOTYPES
Object-oriented programming languages are usually based on *classes*. Classes are descriptions of objects that are capable of serving as “cookie-cutters” from which *instances*, the actual objects described by classes, can be created. This creation process is known as *instantiation*. In broad terms, a class represents a generic concept, or a “recipe”, while an instance represents an individual, or a particular occurrence of a concept. A class describes the similarities among a group of objects, defining the structure and behavior of its instances, whereas instances hold the local data representing the state of each object.

To many people, the notion of a class is an essential, indistinguishable part of object-oriented programming. However, there are object-oriented systems in which there are no classes at all. These systems are based on an alternative model known as *prototype-based object-oriented programming*, or simply *prototype-based programming*. Prototype-based programming was a lively research topic especially in the late 1980s, and numerous papers on the topic were published in the ECOOP and OOPSLA conferences in the late 1980s and early 1990s.

In prototype-based object-oriented systems there are no classes. Rather, new objects are formed in a more direct fashion by constructing concrete, full-fledged objects that are referred to as *prototypes* or *exemplars*. A prototype can be thought of as a standard "example instance" that represents the default behavior of some concept. Because of the absence of classes, there is usually no notion of instantiation. Rather, new objects are created by *cloning*, that is, by *copying* the prototype or other existing objects. In some prototype-based programming languages the initial prototype instance of each object type has a special role, and new objects can be created only by using the prototype. In other prototype-based languages all objects are treated equally, that is, every object in the system can be used as a prototype for new objects.

The best known prototype-based programming language – before the introduction of JavaScript – was *Self*, developed originally at Stanford University and later at Sun Microsystems Laboratories. The most successful prototype-based language from the commercial viewpoint was *NewtonScript* – the language that was used for building Apple's Newton PDA platform. Many other prototype-based languages such as Omega [1] and Kevo [11] were introduced during the golden era of prototype-based programming in the late 1980s and early 1990s.

3. TWO FUNDAMENTAL MODELS OF PROTOTYPE-BASED PROGRAMMING

As with class-based systems, there are significant differences between different prototype-based programming languages and systems. The different models of prototype-based programming were summarized and evaluated by Dony, Malenfant and Cointe in 1992 [3]. The author of this paper also wrote an extensive summary as part of his Ph.D. Thesis back in 1993 [12].

For the purposes of this paper, we divide prototype-based programming systems into two categories: (1) those that are based on *delegation* and (2) those that are based on *concatenation* (or *replication*). In delegation, objects inherit properties from their parents by physically *sharing* the properties of their parents. In replication, objects inherit properties by *replicating* (copying) the properties of their parents. Both approaches are introduced in more detail below.

3.1 Class-Based Example

To illustrate the differences between alternative models of object-oriented programming, let us suppose that we would like to define a turtle graphics system familiar from the Logo programming language [7]. We assume that there are two kinds of turtle objects: simple `Turtle` objects that are drawn in black color only, and `ColorTurtle` objects that can be drawn in any given color. In a class-based language, the necessary classes would be defined as shown below. Note that we use a Java-like syntax here, except that we have omitted static types in order to make the code more consistent with the JavaScript code shown later in this paper. To shorten the examples, we have also omitted the actual method code from the examples.

```
class Turtle {
  var x; // Current x-coordinate of the turtle
  var y; // Current y-coordinate of the turtle
  var heading; // Current heading of the turtle

  function forward(dist) { ... };
  function rotate(angle) { ... };
  function Turtle(x, y, heading) { ... };
};

class ColorTurtle extends Turtle {
  var color; // Current drawing color

  function setColor(newColor) { ... };
  function forward(dist) { ... }; // Override!
  function ColorTurtle(x, y, h, color) { ... };
};
```

Class `Turtle` defines three instance variables `x`, `y` and `heading`, and two methods (functions) `forward` and `rotate`. Variables `x` and `y` hold the current location (x and y coordinate) of the turtle on the screen, while instance variable `heading` holds the angle in which the turtle will move when the turtle is instructed to move forward. Method `forward` is used for moving the turtle forward, and method `rotate` changes the heading of the turtle. We also define a constructor method `Turtle` that can be used for creating instances of the class. The constructor takes the starting position and heading of the new turtle as parameters.

Class `ColorTurtle` refines the behavior inherited from class `Turtle` by defining an additional instance variable called `color`, as well as two additional methods `setColor` and `forward`. Variable `color` holds the current drawing color of the turtle, while method `setColor` can be used for changing the current drawing color. Method `forward` overrides the behavior inherited from class `Turtle`, so that the drawing code takes into account the current drawing color.

The sample code in Listing 2 shows the turtle graphics system in action. In that code, we create instances `t1` and `t2` of classes `Turtle` and `ColorTurtle`, respectively. We then use these instances to perform some drawing on the screen.

```
var t1 = new Turtle(100, 100, 0);
var t2 = new ColorTurtle(200, 200, 0,
                         Color.blue);

t1.forward(10).rotate(10).forward(10);
t2.setColor(Color.green).rotate(20).forward(20);
```

Note that for simplicity, many essential features of a turtle graphics system have been omitted from this example. For instance, a more realistic implementation would have methods such as `penUp` and `penDown` to raise and lower the pen, so that the user could move the pen without drawing.

3.2 Delegation

The interest in prototype-based programming was sparked by Henry Lieberman's paper on delegation presented in the OOPSLA Conference in Portland, Oregon, in September 1986. That paper was followed by numerous other research papers published primarily in the ECOOP and OOPSLA conferences in the late 1980s and early 1990s.

As in real life, delegation implies *shared responsibility for completing a task*. By utilizing delegation, objects can share each others' properties (behavior and state, i.e., methods and variables) through the forwarding of messages. If an object cannot handle a request to invoke a certain method or access a certain variable, it can delegate the request to its parent(s). Any object can serve as a parent for another object. Typically, each object has a `parent` pointer that indicates to which object messages are forwarded if the object itself cannot handle the request.

Figure 1 shows delegation in action. In this example, we have implemented the turtle graphics system using two concrete objects. The upper object in the figure is the prototypical instance of the `Turtle` class shown earlier. The lower object is the equivalent of an instance of the `ColorTurtle` class. This object has a `parent` pointer that indicates that messages are automatically delegated to the other object if the object receives a message (such as `rotate`) that it cannot handle itself.

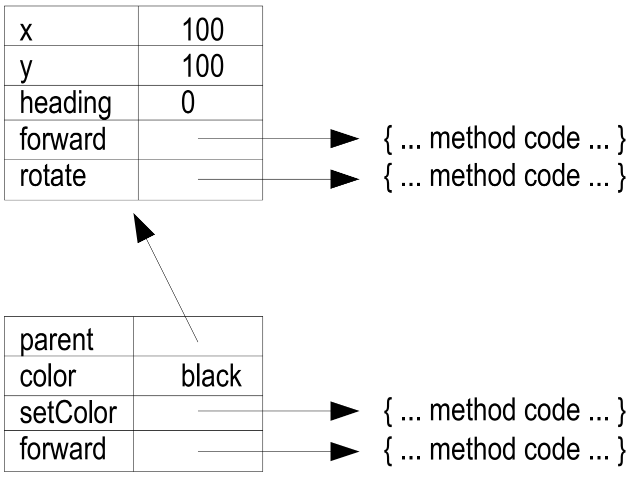

Figure 1: Delegation illustrated

Note that this example suffers from a few major drawbacks. Since delegation implies that the properties inherited from a parent are shared, any changes to the parent object's properties (including its state variables) are immediately reflected to its children. In this example, when the location or heading of the "colorless" turtle is changed, the location and/or heading of the color turtle would be changed as well. This is not desirable, since in a turtle graphics system all the turtle instances are intended to be independently maneuverable.

Another drawback in this example is related to object instantiation. In prototype-based systems new objects are typically created by *cloning* (copying) existing objects. However, without some additional restructuring from the programmer's behalf, the cloning of the objects above does not really yield the desired results. For instance, if the programmer were to clone the color turtle object shown in Figure 1, the parent object would have to be cloned as well, or otherwise the state variables of the parent would be shared with yet another object. Alternatively, to avoid sharing of state, the programmer would have to override (redefine) all the variables (`x`, `y`, `heading`) in the child object. The resulting system would soon be a mess, with state variables and methods sprinkled all over the system without any apparent reason or structure.

A more practical example of delegation. Figure 2 shows a more practical example of delegation. In this example, we assume that the programmer has explicitly structured the system so that behavior and state of objects are represented in separately. Objects on the left side of the figure represent "class-like" structures that define the behavior of the turtle objects. Objects on the right hand side represent the actual turtle and color turtle object instances that hold the local state of each object. When the turtle instance receives a message to manipulate the state of the turtle – such as `forward` or `rotate` – the object utilizes its `parent` pointer to delegate the request to the class-like structure holding the corresponding methods.

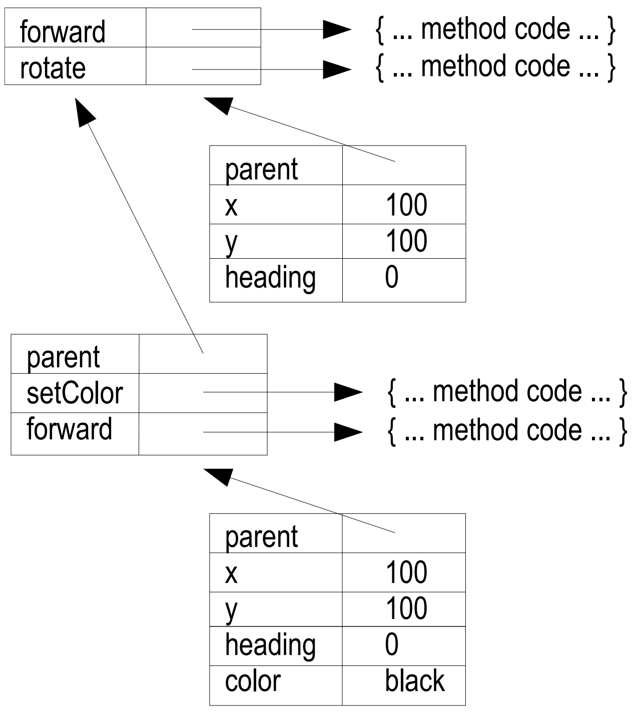

Figure 2: A more practical example of delegation (with separate traits and instance objects)

This kind of object structure makes it easy to instantiate new turtles or color turtles. New instances can be created simply by copying existing instances. State of each object is independent of other objects, i.e., each turtle instance can be maneuvered independently, and the methods of the objects are not unnecessarily copied while creating new instances. The resulting object structure is surprisingly similar to a class-based system, with a relatively clean separation between class-like and instance-like structures. Of course, such structuring does not occur automatically. Rather, the programmer must explicitly create the separate structures so that the behavior and local state of objects is represented separately.

Before the JavaScript language was introduced, the best known prototype-based programming language supporting delegation was *Self* [2, 14]. The program structuring technique illustrated in Figure 2 was originally pioneered in the Self system, in which the class-like structures shown in Figure 2 were referred to as *traits* objects. Self also supported *dynamic inheritance* (also known as dynamic delegation) by allowing the parent-child relationships of objects to be changed dynamically while the system is running. Such "reparenting" operations must be performed with utmost care, since not all the parent objects have the necessary behavior available. Some versions of the Self language also supported *multiple inheritance* by allowing each object to have multiple parents. Since this feature added considerable complexity to the semantics of the Self language, support for multiple inheritance was abandoned in later versions of the language.

3.3 Concatenation (Replication)

There is an alternative, less widely known model of prototype inheritance that is known as *concatenation* [11, 13]. In this alternative model, new object types are created by copying (replicating) existing objects (usually by shallow copying them so that only the namespace of the object is copied) and then extending (concatenating) the resulting new object with additional properties.

Consider the turtle graphics example shown in Figure 3. The upper object shown in Figure 3 represents the prototypical colorless turtle object. The lower object represents the prototypical color turtle object that has been created by first shallow copying the turtle object and then adding the `color` variable and the `setColor` and overriding `forward` methods that are specific to color turtle objects.

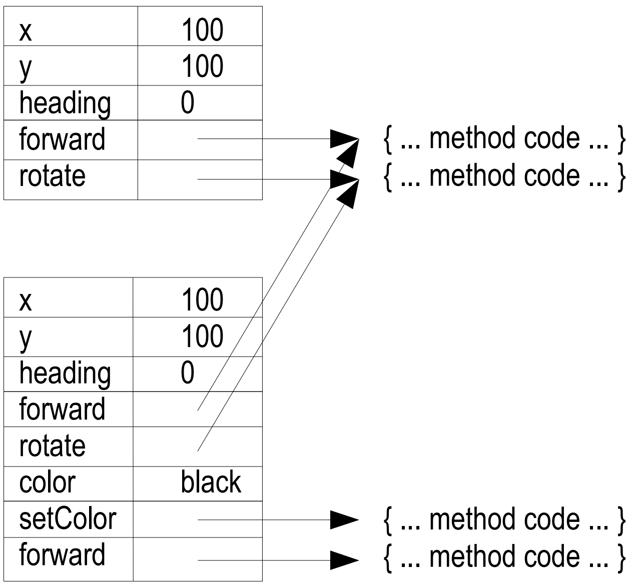

Figure 3: Concatenation illustrated

Note that concatenation is a two-phase process. First the object to be inherited is replicated, and then the new object is concatenated with additional methods and variables. Once the process is complete, there is no direct sharing relationship between the parent and the child, apart from method code and other possible structures that are referred to by pointers from the original object (see as the shared method code in Figure 3).

As in delegation-based systems, new object instances are created by cloning (copying) existing objects. More conventional object instantiation can be supported as well by defining constructor methods that reinitialize the copied state variables to desired initial values.

Concatenation-based prototype inheritance is sometimes referred to as *replication*, since this form of inheritance resembles biological inheritance in which the DNA is replicated during the creation of a new cell. Once replication has been completed, there is no sharing relationship between the original and the new cell.

In general, the essential difference between delegation and concatenation boils down to *sharing* versus *copying* of data structures. In delegation, the properties of the parent object are physically shared between the parent and its children during the lifetime of those objects. This is referred to as "*life-time sharing*" by Dony, Malenfant and Cointe [3]. In concatenation, the properties (behavior and state) of the parent object are copied to the child at the creation time. Once copying has been completed, there is no direct sharing relationship between the original object (parent) and the copy (child). According to Dony, Malenfant and Cointe, this is referred to as "*creation-time sharing*".

The archetypical prototype-based programming language supporting concatenation-based inheritance was *Kevo*, designed and implemented by the author of this paper in the early 1990s [11]. Kevo implemented a pure concatenation-based object model in which new objects were created by copying and the namespaces of all the objects were always fully self-contained. *Module operations* were provided to manipulate objects on the fly, e.g., to add or remove methods or variables flexibly. Furthermore, Kevo had an internal *clone family* mechanism that made it possible to track the "genealogy" of changes among groups of objects, so that changes to individual objects could be propagated to other objects when necessary. The clone family mechanism was also utilized internally by the system to optimize the implementation of object structures inside the Kevo virtual machine. The semantics of the clone family mechanism turned out to be rather complex in practice.

4. THE OBJECT MODEL OF JAVASCRIPT

JavaScript is a prototype-based object-oriented programming language that is based on delegation[1]. By default, JavaScript does not support classes. Instead, new object types are defined by defining *constructor functions* that initialize the state of the new object to desired values. The code sample below illustrates how to define our `Turtle` "pseudoclass" in JavaScript.

```
function Turtle(x, y, heading) {
    this.x = x;
    this.y = y;
    this.heading = heading;
}
```

New `Turtle` instances can be created using keyword `new`.

```
var t1 = new Turtle(100, 100, 0);
var t2 = new Turtle(200, 200, 45);
```

Methods and instance variables can be added to the pseudoclass by utilizing a property called `prototype` that defines a delegation relationship between object instances and their prototypes (the actual method code is not shown here).

```
Turtle.prototype.forward = function(dist) { };
Turtle.prototype.rotate  =  function(angle)  {  };
```

By adding methods to the prototype object, the programmer can ensure that the methods are shared between the instances and

[1] This statement is an oversimplification. JavaScript is a *multi-paradigm* language that can support various programming styles and paradigms. However, the object model of JavaScript is based on the prototype-based approach, and it utilizes delegation for implementing prototype inheritance.

not unnecessarily copied to all the instances. In other words, the methods behave as instance methods. These functions can be invoked in the expected fashion, e.g.,

```
t1.forward(100);
t2.rotate(10);
```

Defining subclasses. Subclass definition in standard JavaScript is a bit tricky. Basically, the programmer must first define a constructor function that invokes constructor function of the "superprototype" (in order to initialize the properties inherited from the superclass) and that also initializes the properties (instance variables) that are specific to the instances of the subclass. This process is known as *constructor chaining*.

Below we define the constructor for our `ColorTurtle` pseudoclass. When invoked, this constructor will first call the constructor of the earlier defined `Turtle` pseudoclass and then initialize the `color` property that is specific to `ColorTurtle` instances.

```
function ColorTurtle(x, y, heading, color) {
    Turtle.call(this, x, y, heading);
    this.color = color;
}
```

After defining the constructor, the programmer must manually set the `prototype` property and the associated `constructor` property (which defines the function that will be invoked when the programmer uses the keyword `new`) so that the object behaves as a subclass of `Turtle`. This can be accomplished by associating an instance of `Turtle` as the prototype of `ColorTurtle` (in order to establish a delegation relationship between the two prototypes), and then by associating the `constructor` property of the prototype with the `ColorTurtle` constructor that we defined above.

```
ColorTurtle.prototype = new Turtle();
ColorTurtle.prototype.constructor = ColorTurtle;
```

Once this process is complete, we can now add new methods to the subclass, create instances of the new class, as well as invoke methods of the instances in the expected fashion.

```
ColorTurtle.prototype.setColor = function(col) {
    this.color = col;
};

var t3 = new ColorTurtle(200, 200, 0,
                         Color.blue);
t3.forward(10);
t3.setColor(Color.green);
```

Comments and criticism. At the first glance, the programming style shown above may seem rather strange, especially to those programmers who are unfamiliar with the prototype-based programming approach. However, the programming style is instantly recognizable to people who have experience with prototype-based languages such as Self. In fact, the resulting object structures are very similar to those shown in Figure 2 earlier. The class-like prototype objects in JavaScript correspond to *traits* objects in Self (see object structures on the left in Figure 2). Instead of a `parent` pointer, JavaScript uses the `prototype` property to establish a delegation relationship between instances and prototypes and between the prototypes. Furthermore, JavaScript uses the `constructor` property to introduce a more class-like syntax for object instantiation. This feature does not exist in Self, in which new objects are created in a more pure prototype-based fashion by cloning (copying) objects instead.

The general problem with the object model of JavaScript is that it requires the programmer to be intimately aware of the details of the prototype-based programming approach. Furthermore, this programming style exposes low-level details of the JavaScript language, such as the `prototype` and `constructor` properties, directly to the programmer. There are a number of JavaScript libraries such as Prototype (http://www.prototypejs.org/) that introduce additional syntactic sugar to hide the implementation details, but the proposed syntactic enhancements vary from one library to another, reducing application portability and increasing confusion among programmers. Until a more conventional class definition and inheritance mechanism is added to the JavaScript (ECMAScript) language specification [4], the development of larger JavaScript applications will remain challenging at least for those programmers who have not been trained to think in a prototype-based fashion.

5. CONCATENATION-BASED INHERITANCE FOR JAVASCRIPT

In this section we introduce our proposal for concatenation-based prototype inheritance for JavaScript. The proposal builds upon the *object literal notation* of the JavaScript language, as well as on some modifications to the semantics of the concatenation ("+") operation for JavaScript objects.

Using object literals for object definition. One of the most convenient features of the JavaScript programming language – especially compared to its sister languages C, C++ and Java – is the object literal notation that allows new object structures to be defined easily on the fly. An *object literal* in JavaScript is a comma-separated list of property name/value pairs, enclosed within curly brackets [5, p.106].

Using the object literal definition, a turtle object similar to those in our earlier examples can be defined in JavaScript as follows:

```
var turtle = {
    x: 100,
    y: 100,
    heading: 0,
    forward: function(dist) { ... },
    rotate: function(angle) { ... }
}
```

The object literal definition above (between the curly brackets) creates a JavaScript object instance with five properties (`x`, `y`, `heading`, `forward` and `rotate`). The pointer to the object is assigned to a variable called `turtle`. For all practical purposes, this object behaves like an instance of class `Turtle`, although no class is defined at all. For instance, the programmer can call the `forward` and `rotate` methods as follows:

```
turtle.forward(10);
turtle.rotate(10);
```

Supporting concatenation-based inheritance. As summarized in Section 3.3, the key idea in concatenation-based inheritance is to create new object types by copying (replicating) existing objects (usually by shallow copying) and then extending (concatenating) the resulting new object with additional properties ("differences"). Our proposal is to use the JavaScript object literal notation for object definition, but modify the semantics of the concatenation operator ("+") in JavaScript to perform concatenation-based prototype inheritance automatically when the "+" operator is applied to objects.

Using our proposal, a color turtle object similar to those in our earlier examples could be defined as follows:

```
var colorTurtle = turtle + {
    color: Color.black,
    setColor: function(newColor) { ... },
    forward: function(dist) { ... }
}
```

The gist of our proposal is to use the "+" operator to perform object concatenation in the same way as described earlier in Section 3.3. The more general syntax is as follows:

```
  { /*original object*/ } + { /*differences*/ }
```

At the implementation level, this operation will make a shallow copy of the object on the left, and then augment the resulting new object with a shallow copy of the properties of the object on the right (the differences). The original objects are not modified in any way. The resulting new object will have its own namespace, i.e., there is no sharing of properties (name/value pairs) between the objects. However, if some of the properties of the original object contain pointers to additional structures, those structures will be shared between the original object and the new one (see the earlier Figure 3 for an illustration of this).

Instantiating new objects by copying. As in other prototype-based systems, object instantiation in our proposal is performed primarily by copying. By default, JavaScript does not provide an object copy or clone operation. However, many JavaScript libraries such as Prototype include such an operation. We have adopted the `clone` operation from the Prototype library:

```
var t1 = turtle.clone();
```

More conventional object instantiation can be supported by adding an explicit instantiation method to each object. Below we illustrate how such an instantiation method `anew`[2] could be defined:

```
turtle = turtle + {
    anew: function(x, y, heading) {
        var other = this.clone();
        other.x = x;
        other.y = y;
        other.heading = heading;
        return other;
    }
}
```

After adding the `anew` method, new turtle instances can be created (with desired initial instance variable values) as follows:

```
var t2 = turtle.anew(200, 200, 45);
```

This operation will first clone the `turtle` object, and then set the instance variables of the copy to the requested new values. The resulting behavior is effectively the same as in conventional object instantiation.

Accessing inherited properties. A key feature of any object-oriented programming language is the ability to invoke overridden methods that have been inherited from superclasses or parent objects. Most object-oriented programming languages provide a keyword called `super` or something equivalent to enable this.

In our current implementation of the proposed new approach, we utilize a synthesized "`$super`" property (similar to the one in the Prototype library) that has been combined with the implementation of the object concatenation operation. If support for concatenation-based prototype inheritance were built directly into the JavaScript language itself, a more conventional implementation of `super` (which is already one of the reserved future keywords in the ECMAScript Specification [4]) should be provided.

6. DISCUSSION

Our proposal has a number of benefits and drawbacks. The main advantages and disadvantages are discussed below.

Advantages of the proposed approach. Concatenation-based prototype inheritance fits generally very well with the object literal notation of JavaScript. In many ways, the object literal notation is a far more natural notation for defining prototype-based objects than the standard JavaScript approach that is based on defining rather awkward constructor functions separately from the rest of the properties of objects. Object concatenation complements the object literal notation in a syntactically clean fashion, and requires only minor changes to the semantics of the JavaScript language.

From the programmer's viewpoint there are many benefits. There is no need to define separate prototype objects or traits. Rather, all the objects are concrete and self-contained, and include not only the state of the objects but also their behavior. Namespaces of objects are self-contained, meaning that there is no need to traverse a complex inheritance or delegation hierarchy in order to understand the behavior of an object. Furthermore, there is no need to manually set the values of objects' `prototype` or `constructor` properties. Actually, in a pure concatenation-based system, there would be no need to have these properties *at all*. The `prototype` or `constructor` properties are implementation-level details that have a tendency to cause confusion among novice JavaScript programmers; the less these features are exposed to application developers, the better.

As discussed in [12], concatenation-based prototype inheritance has some benefits also when implementing visual development tools that are intended for interactive application development. The self-contained nature of the objects means that each object can be visualized as an independent structure with its entire behavior and state. There is no need to provide separate class browsers and object inspectors, or to open a large number of separate browser or inspector windows in order to analyze the behavior of an object.

Disadvantages – performance and memory consumption. At the first blush, the concatenation-based approach suffers from some major drawbacks. For instance, since both object instantiation and inheritance in this approach is based on object copying, the efficiency of the object copying operation is critical to the overall performance of the system. If object copying is implemented naïvely by copying the entire namespace of each object, the system will be very inefficient both in terms of performance and memory consumption. Since JavaScript objects are represented as associative (and potentially sparse) arrays, copying of JavaScript objects can be extremely inefficient. Furthermore, in a concatenation-based

[2] Ideally, this method should be called `new`. However, `new` is a reserved word in JavaScript.

system method pointers are replicated far more extensively than in other types of object-oriented systems, and this can also introduce a significant memory consumption overhead.

Fortunately, there are well known optimization techniques for solving the above mentioned problems. The designer of the JavaScript virtual machine (VM) may choose to share many of the data structures internally in the VM, without exposing such optimizations to the application developer. For instance, when an application developer creates a hundred instances (copies) of an object, the VM implementation can internally share the common parts between those objects (that is, nearly everything except object-specific state), by generating an internal map structure inside the VM to contain the shared parts. Such implementation optimization techniques were utilized extensively, e.g., in the Self virtual machine [2]. In the same fashion, the VM designer may choose to share most of the structures between parent and child objects when the programmer uses concatenation to inherit and extend the behavior of objects with new functionality. As long as the properties inherited from parents are not changed, those properties can be shared between the parents and their children. The Kevo language had a special *clone family* mechanism specifically to keep track of such changes and to allow implementation-level sharing of data and behavior based on copy-on-write semantics [11, 12].

The optimization techniques needed for creating an efficient implementation of a concatenation-based system place a significant burden on the JavaScript VM designer. However, since JavaScript is – first and foremost – an interactive scripting language intended to make end-user programming as simple and convenient as possible, it seems far more appropriate to place the optimization burden on the VM designer than on the tens of thousands of relatively inexperienced JavaScript application developers all over the world. Using the current delegation-based approach, the application developers have to spend a considerable amount of time structuring their applications in a specific, rather unintuitive way.

Additional disadvantages – group manipulation and type checking. Concatenation-based prototype inheritance has additional drawbacks that are related to the inability to modify large groups of objects at once. Since delegation-based systems implement parent-child relationships by physically *sharing* the properties inherited from parents, all the changes that are subsequently made to the parent objects will have an immediate impact on all the children as well. While in some rare cases this behavior can be undesirable, in most cases the ability to change the behavior of a large group of objects simply by modifying a common parent can be very convenient.

In a concatenation-based system, in contrast, the addition or removal of a method or a variable will have an impact only on an individual object. If the programmer wants to add certain new properties (e.g., a few new methods) to an entire "class" of objects, the programmer will have to use a `for` loop or some other iterative construct to apply the same changes/differences to all the objects individually. This can be clumsy and inefficient. On the positive side, the benefit of this approach is that the same changes can be applied to *any* group of objects – not just those objects who happen to share a common parent.

If the virtual machine includes a clone family mechanism or some other "computed" inheritance hierarchy, such hierarchies can be used for change propagation as well.

An addition problem arising from the lack of clear inheritance or delegation hierarchy in the system is the difficulty in implementing type checking. Although most scripting languages (including JavaScript) do not support static type checking, there is still a need to be able to precisely identify and compare the dynamic types of objects at runtime. In the absence of delegation and the `prototype` property, the implementation of JavaScript operators such as `typeof` or `instanceof` can be considerably more challenging.

There are a number of solutions to these problems. For instance, if the virtual machine includes an aforementioned clone family mechanism to track the "genealogy" of changes to objects, the internally maintained "computed" inheritance hierarchies can be used for dynamic type checking. The virtual machine can also calculate "checksums" of objects whenever the properties (namespaces) of objects change, and then using those values for type comparison. At the application level, duck typing (see, e.g., [5, Section 9.7.3]) can be used as well.

7. RELATED WORK

The work presented in this paper is based on the author's earlier work on concatenation-based prototype inheritance and the Kevo language in the early 1990s [11, 12, 13]. That work precedes the introduction of the JavaScript language by many years.

Most likely, people involved in the design and standardization of the JavaScript language were not aware of any alternative models of prototype-based inheritance. After all, research into prototype-based programming faded into obscurity quickly in the mid-1990s. Nevertheless, it is interesting to note that many JavaScript libraries such as Prototype utilize forms of concatenation-based inheritance. Sun Labs Lively Kernel – an interactive web programming environment built entirely in JavaScript – also utilizes a variant of concatenation in implementing its subclassing mechanism [6]. Furthermore, this form of inheritance has been discussed in Flanagan's JavaScript book [5], Section 9.6 "Extending Without Inheriting".

In general, JavaScript is such a flexible language that it is not surprising that many people have independently come up with solutions for these alternative forms of inheritance.

As far as we know, we have made the first proposal to alter the semantics of the concatenation operator ("+") in JavaScript so that concatenation-based inheritance is performed automatically when the "+" operator is applied to objects. It remains to be seen whether this proposal actually catches up or stirs any discussion in the JavaScript community. Over the past several years, JavaScript standardization work has progressed very slowly, with considerable disagreements among the companies participating in the standardization process. In any case, it seems much safer to predict that the future versions of JavaScript will move closer towards conventional class-based inheritance, rather than adopt any alternative form of prototype inheritance.

8. CONCLUSION

In this paper we have proposed some extensions to the JavaScript programming language in order to support concatenation-based prototype inheritance – an alternative model of object-oriented programming that was first introduced in the early 1990s. We argued that with some relatively simple syntactic and semantic extensions, the expressive power of the JavaScript programming language can be increased and the entire language made more amenable and intuitive especially to novice programmers.

9. REFERENCES

[1] Blaschek, G. *Object-Oriented Programming with Prototypes.* Springer-Verlag 1994.

[2] Chambers, C., Ungar, D. and Lee, E. An Efficient Implementation of Self, a Dynamically-Typed Object-Oriented Language Based on Prototypes. In *Proceedings of OOPSLA'89 Conference* (New Orleans, Louisiana, October 1-6, 1989), ACM SIGPLAN Notices 24, 10 (October 1989), 49-70.

[3] Dony, C., Malenfant, J. and Cointe, P. Prototype-Based Languages: From a New Taxonomy to Constructive Proposals and Their Validation. In *Proceedings of OOPSLA'92 Conference* (Vancouver, Canada, October 18-22, 1992), ACM SIGPLAN Notices 27, 10 (October 1992), 201-217.

[4] *ECMAScript Language Specification*, 3rd Edition. Standard ECMA-262, December 1999. URL: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf

[5] Flanagan, D. *JavaScript: The Definitive Guide,* 5th edition. O'Reilly Media, 2006.

[6] Ingalls, D., Palacz, K., Uhler, S., Taivalsaari, A. and Mikkonen, T. The Lively Kernel – A Self-Supporting System on a Web Page. In *Proceedings of the 2008 Workshop on Self-Sustaining Systems* (Potsdam, Germany, May 15-16, 2008), Lecture Notes in Computer Science LNCS5146, Springer-Verlag, 2008, 31-50.

[7] Lukas, G. and Lukas, J. *The LOGO Language: Learning Mathematics Through Programming*. Entelek Press, 1977.

[8] Noble, J., Taivalsaari, A. and Moore, I. (eds), *Prototype-Based Programming: Concepts, Languages and Applications*. Springer, 1999.

[9] Smith, W.R., The Newton Application Architecture. In *Proceedings of the 39th IEEE Computer Society International Conference* (San Francisco), 1994, 156-161.

[10] Smith, W.R., NewtonScript: Prototypes on the Palm. In Noble, J., Taivalsaari, A., Moore, I. (eds), *Prototype-Based Programming: Concepts, Languages and Applications*. Springer, 1999, 109-139.

[11] Taivalsaari, A. *Kevo – a Prototype-Based Object-Oriented Language Based on Concatenation and Module Operations*. Technical report DCS-197-1R, University of Victoria, B.C., Canada, June 1992.

[12] Taivalsaari, A. *A Critical View of Inheritance and Reusability in Object-Oriented Programming*, Ph.D. thesis, Jyväskylä Studies in Computer Science, Economics and Statistics 23, University of Jyväskylä, Finland, December 1993, ISBN 951-34-0161-8, 276 pages.

[13] Taivalsaari, A. On the Notion of Inheritance. *ACM Computing Surveys* 28, 3 (September 1996), 438-479.

[14] Ungar, D. and Smith, R.B. Self: The Power of Simplicity. In *Proceedings of OOPSLA'87 Conference* (Orlando, Florida, October 4-8, 1987), ACM SIGPLAN Notices 22, 12 (December 1987), 227-241.

ABOUT THE AUTHOR

Dr. Antero Taivalsaari is a Docent at Tampere University of Technology and [until October 2009] a Principal Investigator at Sun Microsystems Laboratories. Antero is best known for his seminal role in the design of the Java™ Platform, Micro Edition (Java ME) – one of the most widespread commercial software platforms in the world, with about three billion devices deployed so far. Antero has received Sun's Chairman's Award for Innovation twice (in 2000 and 2003), as well as the Mobile Entertainment Forum's Outstanding Contribution Award (http://m-e-f.blogspot.com/2009/06/inventors-of-java-on-mobile-receive.html) in 2009 for his work on Java ME technology.

In August 2006-October 2009, Antero co-led the Lively Kernel research project (http://research.sun.com/projects/lively) with Dan Ingalls at Sun Labs, focusing on the creation of a highly interactive, visual JavaScript programming environment for the Web. Results of this work have been reported in various conference papers and technical reports (see http://lively.cs.tut.fi/publications.html).

Starting from November 2009, Antero is a Distinguished Engineer at Nokia. For more information, refer to http://www.taivalsaari.com/.