Worlds: Controlling the Scope of Side Effects

Alessandro Warth and Alan Kay

**NOTE This document is superseded by VPRI Technical Report TR-2011-001

VPRI Research Note RN-2008-001

This material is based upon work supported in part by the National Science Foundation under Grant No. 0639876. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Viewpoints Research Institute, 1209 Grand Central Avenue, Glendale, CA 91201 t: (818) 332-3001 f: (818) 244-9761

Worlds: Controlling the Scope of Side Effects

Alessandro Warth and Alan Kay
Viewpoints Research Institute
{alex, alan}@vpri.org

September 6, 2008

Abstract

The state of an imperative program—e.g., the values stored in global and local variables, objects' instance variables, and arrays—changes as its statements are executed. These changes, or side effects, are visible globally: when one part of the program modifies an object, every other part that holds a reference to the same object (either directly or indirectly) is also affected. This paper introduces *worlds*, a language construct that reifies the notion of program state, and enables programmers to control the scope of side effects. We investigate this idea as an extension of JavaScript, and provide examples that illustrate some of the interesting idioms that it makes possible.

1 Introduction

Suppose that, while browsing the web, you get to a page that has multiple links and it is not clear which one (if any) will lead to the information you're looking for. Maybe the desired information is just one or two clicks away, in which case it makes sense to click on a link, and if you don't find you're looking for, click the back button and try the next link. If the information is more than a few clicks away, it might be better to open the link in a new *tab* in which you can explore it to arbitrary depths. That way, if you eventually decide that wasn't the way to go, you can close the tab, and easily try a different path. Another option is to open each link in its own tab, and explore all of them "concurrently".

Something like the tabs of a web browser would be even more useful in a programming language, where undoing actions is a lot trickier than clicking a back button. As an example, consider the task of programming a robot to open a locked safe, as shown in Figure 1. There are two keys, A and B (each in its own room), but only one of them unlocks the safe. Using a conventional programming language, we might tell the robot to grab key A from room A, then go to the safe and try to unlock it. At this point, if we find that key A does not open the safe, we probably want to have the robot clean up after himself before trying the next alternative (nobody likes a messy robot). So we must tell the robot to take key A back to room A, and then return to its initial position.

In a programming language that supports "tabs", these clean-up actions would not be required: we could simply open a new tab, and inside it try to open the safe with key A. If A turns out to be the wrong key, we can simply close this new tab to return to the initial conditions.

This paper explores the idea of "tabs for programming languages", which we call *worlds*.

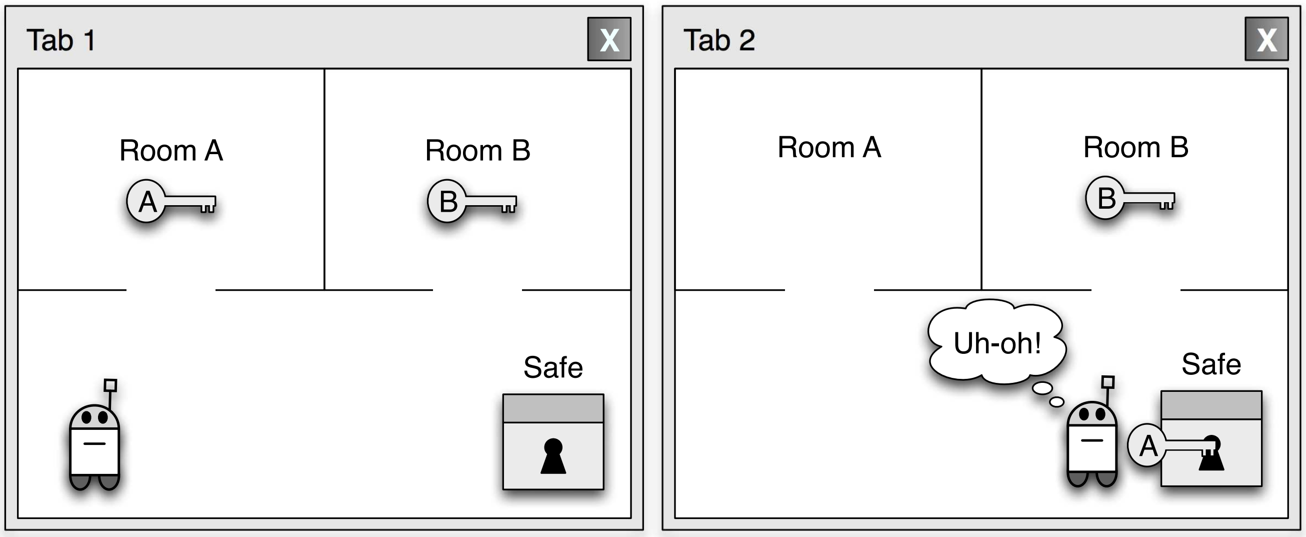

Figure 1: “Tabs” 1 and 2 show the state of the world initially, and when the robot discovers that key A does not unlock the safe, respectively.

2 Approach

The state of a program is scattered around the computer’s memory in several kinds of data structures: arrays, objects, activation records, etc. We normally think of these data structures as little “bundles of state”, and they are often implemented as such (see Figure 2-A). Alternatively, we can think of program state *itself* as a data structure—a kind of lookup table or associative array—that is used to represent all other data structures in the system. In this model, each object or data structure is uniquely identified by a *tag*, and the program state maps (tag, property name) pairs to their values (see Figure 2-B).

Reifying the notion of program state raises some interesting questions:

- Is it useful for the program state to be a first-class value/object?
- Does it make sense for multiple “program states” to co-exist in the same program?
- If so, should a program state be able to inherit from (or delegate to) another program state?

We answer all of these questions with a resounding “yes”.

2.1 Worlds

The *world* is a new language construct that reifies the notion of program state. All computation takes place inside a world, which captures all of the side effects—changes to global, local, and instance variables, arrays, etc.—that happen inside it.

A new world can be “sprouted” from an existing world at will. The state of a *child world* is derived from the state of its parent, but the side effects that happen inside the child do not affect the parent. (This is analogous to the semantics of delegation in prototype-based languages with copy-on-write slots.) At any time, the side effects captured in the child world can be propagated to its parent via a *commit* operation.

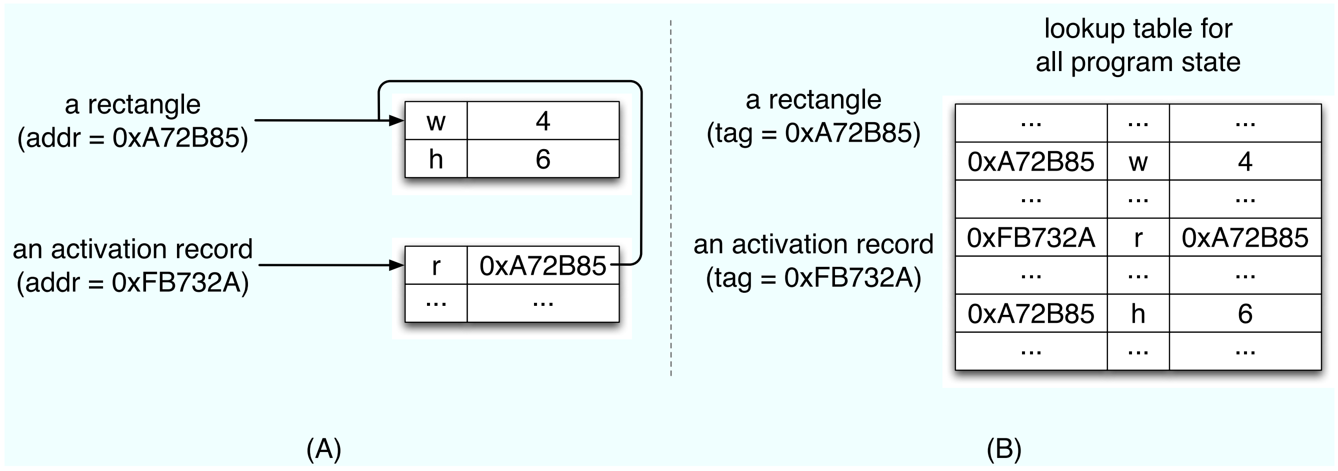

Figure 2: Two ways to represent program state. In (A), an object is uniquely identified by the address of the block of memory in which its state is stored. In (B), objects are just tags and their state is stored externally in a single lookup table.

2.2 Worlds/JS

A programming language that supports worlds must provide some way for programmers to:

- refer to the current world,
- sprout a new world from an existing world,
- commit a world's changes to its parent world, and
- execute code in a particular world.

We now describe the particular way in which these operations are supported in Worlds/JS, an extension of JavaScript [6] we have prototyped in order to experiment with the ideas discussed in this paper.[1]

Worlds/JS extends JavaScript with the following additional syntax:

- `thisWorld` is an expression that evaluates to the current world, and
- `in` *expr block* is a statement that executes *block* inside the world obtained by evaluating *expr*.

Worlds are first-class values: they can be stored in variables, passed as arguments to functions, etc. They can even be garbage-collected just like any other object. All worlds delegate to the world prototype, whose `sprout` and `commit` methods can be used to create a new world that is a child of the receiver, and propagate the side effects captured in the receiver to its parent, respectively.

In the following example, we modify the height of the same instance of `Rectangle` in two different ways, each in its own world, and then commit one of them to the original world. This serves the dual purpose of illustrating the syntax of Worlds/JS as well as the semantics of `sprout` and `commit`.

```
A = thisWorld;
r = new Rectangle(4, 6);
```

[1]Our prototype implementation of Worlds/JS is available at `http://jarrett.cs.ucla.edu/ometa-js/#Worlds_Paper`. No installation is necessary; you can experiment with the language directly in your web browser.

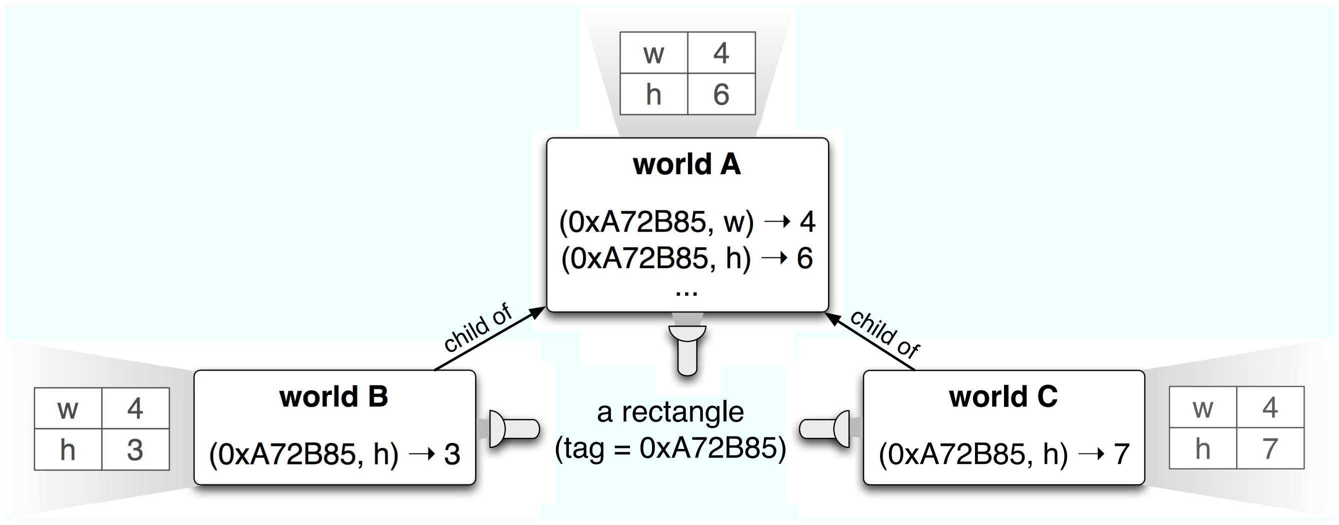

Figure 3: Projections/views of the same object in three different worlds.

```
B = A.sprout();
in B { r.h = 3; }

C = A.sprout();
in C { r.h = 7; }

C.commit();
```

Figures 3 and 4 show the state of all worlds involved before and after the commit operation, respectively.

3 Examples

The following examples illustrate some of the applications of worlds. Other obvious applications (not discussed here) include heuristic search and sand-boxing.

3.1 Better Support for Exceptions

In languages that support exception-handling mechanisms (e.g., the `try/catch` statement), a piece of code is said to be *exception-safe* if it guarantees not to leave the program in an inconsistent state when an exception is thrown. Writing exception-safe code is a tall order, as we illustrate with the following example:

```
try {
  for (var idx = 0; idx < xs.length; idx++)
    xs[idx].update();
} catch (e) {
  // ...
}
```

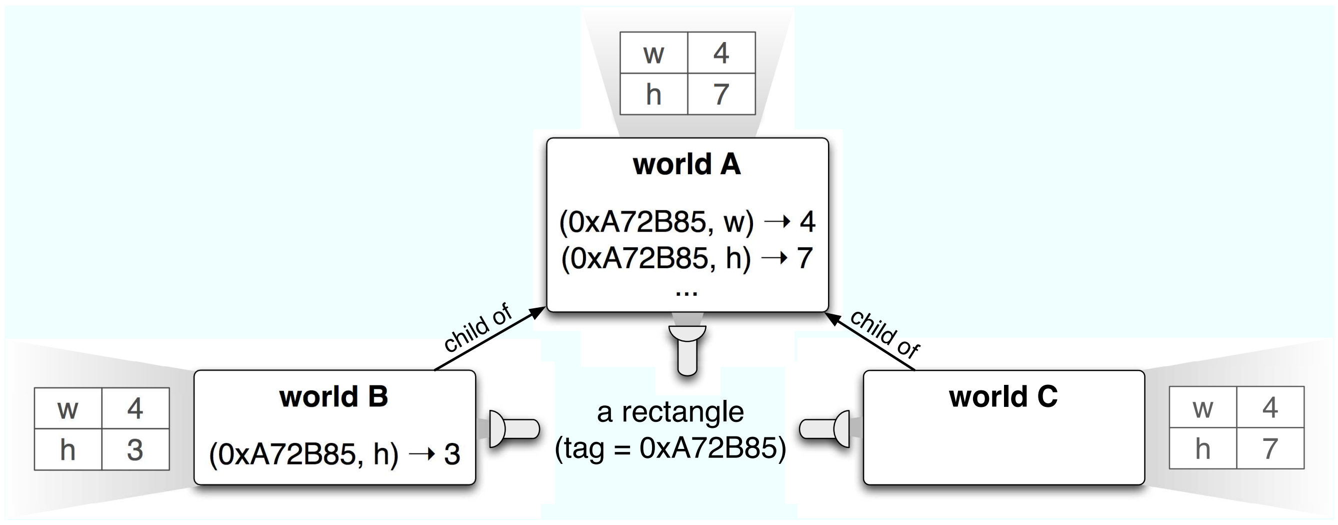

Figure 4: The state of the "universe" shown in Figure 3 after a *commit* on world C.

Our intent is to update every element of `xs`, an array. The problem is that if one of the calls to `update` throws an exception, some (but not all) of `xs`' elements will have been updated. So in the `catch` block, the program should restore `xs` to its previous consistent state, in which none of its elements was updated.

One way to do this might be to make a copy of every element of the array before entering the loop, and in the `catch` block, restore the successfully-updated elements to their previous state. In general, however, this is not sufficient since `update` may also have modified global variables and other objects on the heap. Writing truly exception-safe code is difficult and error-prone.

Versioning exceptions [11] offer a solution to this problem by giving `try/catch` statements a transation-like semantics: if an exception is thrown, all of the side effects resulting from the incomplete execution of the `try` block are automatically rolled back before the `catch` block is executed . In a programming language that supports worlds and a traditional (non-versioning) `try/catch` statement, the semantics of versioning exceptions can be implemented as a design pattern. We illustrate this pattern with a rewrite of the previous example:

```
w = thisWorld.sprout();
try {
  in w {
    for (var idx = 0; idx < xs.length; idx++)
      xs[idx].update();
  }
} catch (e) {
  w = null;
  // ...
} finally {
  if (w != null)
    w.commit();
}
```

First, we create a new world `w` in which to capture the side effects of the `try` block. If an exception is thrown, we simply discard `w`. Otherwise—if the `try` block completes successfully—we propagate (commit) the side effects to the "real world". This is done inside a finally block to ensure that the side effects will be propagated even if the try block returns or invokes a continuation.

3.2 Undo for Applications

We can think of the "automatic clean-up" supported by versioning exceptions as a kind of one-level *undo*. In the last example, we implemented this by capturing the side effects of the try block—the operation we may need to undo—in a new world.

The following framework uses the same idea to enable programmers to construct applications that automatically support multi-level undo. Clients can issue commands to the application via its perform method, which takes as an argument the name of an action and calls the corresponding function in a new world. This new world is sprouted from the world that holds the previous version of the application's state (i.e., the results of the previous action). The undo method discards the world that holds the effects of the last action, returning the application to its previous state. Finally, the flattenHistory method coalesces the state of the application into a single world, which has the effect of preventing the client from undoing past the current state of the application.

```
app = function() { };
app.prototype = {
  worlds: [thisWorld],
  perform: function(action) {
    var w = this.worlds.last().sprout();
    this.worlds.push(w);
    in w { return this[action]() }
  },
  undo: function() {
    if (this.worlds.length > 0)
      this.worlds.pop();
  },
  flattenHistory: function() {
    while (this.worlds.length >= 2) {
      var w = this.worlds.pop();
      w.commit();
    }
  }
};
```

Applications built using our framework do not have to do anything special—e.g., use the *command* design pattern [7]—in order to support undo:

```
counter = function() {
  var count    = 0;
  var me       = new app();
  me.inc       = function() { count++;      };
  me.dec       = function() { count--;      };
  me.getCount = function() { return count; };
  return me;
}();
```

The following code fragment illustrates the way a client interacts with the application.

```
counter.perform("inc");
counter.perform("inc");
counter.perform("dec");
counter.undo();              // undo the decrement operation
counter.perform("getCount"); // returns 2
```

Note that the application's public interface (the `perform` and `undo` methods) essentially models the way in which web browsers interact with online applications, so this technique could be used in a web application framework like Seaside [5].

3.3 Extension Methods in JavaScript

In JavaScript, functions and methods are "declared" by assigning into properties. For example,

```
Number.prototype.fact = function() {
  if (this == 0)
    return 1;
  else
    return this * (this - 1).fact();
};
```

adds the `factorial` method to the `Number` prototype. Similarly,

```
inc = function(x) { return x + 1 };
```

declares a function called `inc`. (The left hand side of the assignment above is actually shorthand for `window.inc`, where `window` is bound to JavaScript's *global* object.)

JavaScript does not support modules, which makes it difficult, sometimes even impossible for programmers to control the scope of declarations. But JavaScript's declarations are really side effects, and worlds enable programmers to control the scope of side effects. We believe that worlds could serve as the basis of a powerful module system for JavaScript, and have already begun experimenting with this idea.

Take extension methods, for example. In dynamic languages such as JavaScript, Smalltalk, and Ruby, it is common for programmers to extend existing objects / classes (e.g., the `Number` prototype) with new methods that support the needs of their own application. This practice is informally known as *monkey-patching* [3].

Monkey-patching is a bad idea: in addition to polluting the interfaces of the objects involved, it makes programs vulnerable to name clashes that are impossible to avoid. Certain module systems, including those of MultiJava [4] and eJava [13], eliminate these problems by allowing programmers to declare *lexically-scoped* extension methods. These must be explicitly imported by the parts of an application that wish to use them, and are invisible to the rest of the application.

The following example shows that worlds can be used to support this kind of modularity:

```
ourWorld = thisWorld.sprout();
in ourWorld {
  Number.prototype.fact = function() { ... };
}
```

The `factorial` method defined above can only be used inside `ourWorld`,

```
in ourWorld {
  print((5).fact());
}
```

and therefore does not interfere with other parts of the program.

This idiom can also be used to support *local rebinding*, a feature found in some module systems [1, 2] that enables programmers to locally replace the definitions of existing methods. As an example, we can change the behavior of `Number`'s `toString` method only when used inside `ourWorld`:

```
in ourWorld {
  numberToEnglish = function(n) {
    ...
  };
  Number.prototype.toString = function() {
    return numberToEnglish(this);
  };
}
```

and now the output generated by

```
arr = [1, 2, 3];
print(arr.toString());
in ourWorld {
  print(arr.toString());
}
```

is

```
[1, 2, 3]
[one, two, three]
```

4 Future Work

We believe that worlds can greatly simplify the management of state in backtracking programming languages such as Prolog and OMeta [12]. To validate this claim, we intend to implement a variant of OMeta/JS (using Worlds/JS) in which the choice operator implicitly sprouts a new world for each of its operands, and discards the side effects of failed alternatives. This should enable rather elegant implementations of inherently stateful grammars, e.g., the Python lexer, which has to maintain an indentation stack in order to implement the offside rule.

There are many problems in computer science for which there are several known algorithms, each with its own set of performance tradeoffs. In general, it is difficult to tell when one algorithm (or optimization) should be used over another. As part of the STEPS project [9, 10], we intend to investigate the feasibility of an efficient, hardware-based implementation of worlds. This implementation could make it practical for a program to choose among optimizations simply by sprouting multiple "sibling worlds"—one for each algorithm—and running all of them in parallel. The first one to complete its task would be allowed to propagate its results, and the others would be discarded.

The (abstract) lookup table shown in Figure 2-B is indexed by two keys: object tag and property name. While this is sufficient to support worlds in a "meat-and-potatoes" programming language like JavaScript, certain advanced features may require more keys. In order to *directly* support module-specific state, for example, we might add a third key to our lookup table that associates each piece of state with a particular module. Similarly, in order to support context-oriented programming (COP) [8], we may want to add a third key to our lookup table that identifies a context. If we want to support both modules and COP, we will need four keys. Therefore it may be interesting to look into supporting an arbitrary number of keys, although this will have to be done carefully in order to avoid over-complicating the programming model.

Finally, worlds control the way we view situations and the properties of objects. We would also like to include in our notion of viewing a mechanism that automatically treats objects at all levels of scale as "inter-relationships between their properties", so that one object's view of another will always see a valid relationship. This will need to include the exceptional cases of views that a debugger might take of "relationships not yet coherent".

References

[1] A. Bergel, S. Ducasse, and O. Nierstrasz. Classbox/J: Controlling the scope of change in Java. In *OOPSLA'05: Proceedings of 20th International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 177–189, New York, NY, USA, 2005. ACM Press.

[2] R. Bergel, S. Ducasse, and R. Wuyts. Classboxes: A minimal module model supporting local rebinding. In *In Proceedings of JMLC 2003 (Joint Modular Languages Conference), volume 2789 of LNCS*, pages 122–131. Springer-Verlag, 2003.

[3] G. Bracha. Monkey patching (blog post). `http://gbracha.blogspot.com/2008/03/monkey-patching.html`.

[4] C. Clifton, T. Millstein, G. T. Leavens, and C. Chambers. MultiJava: Design rationale, compiler implementation, and applications. *ACM Trans. Prog. Lang. Syst.*, 28(3), May 2006.

[5] S. Ducasse, A. Lienhard, and L. Renggli. Seaside: A flexible environment for building dynamic web applications. *IEEE Softw.*, 24(5):56–63, 2007.

[6] ECMA. ECMAScript language specification.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addison-Wesley Professional, January 1995.

[8] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented programming. *Journal of Object Technology (JOT)*, 7(3):125–151, March-April 2008.

[9] A. Kay, D. Ingalls, Y. Ohshima, I. Piumarta, and A. Raab. Proposal to NSF, granted on August 31st, 2006. `http://www.vpri.org/pdf/NSF_prop_RN-2006-002.pdf`.

[10] A. Kay, I. Piumarta, K. Rose, D. Ingalls, D. Amelang, T. Kaehler, Y. Ohshima, C. Thacker, S. Wallace, A. Warth, and T. Yamamiya. Steps toward the reinvention of programming (first year progress report). `http://www.vpri.org/pdf/steps_TR-2007-008.pdf`.

[11] V. K. Nandivada and S. Jagannathan. Dynamic state restoration using versioning exceptions. *Higher Order Symbol. Comput.*, 19(1):101–124, 2006.

[12] A. Warth and I. Piumarta. OMeta: an object-oriented language for pattern matching. In *DLS '07: Proceedings of the 2007 Dynamic Languages Symposium*, pages 11–19, New York, NY, USA, 2007. ACM.

[13] A. Warth, M. Stanojević, and T. Millstein. Statically scoped object adaptation with expanders. In *OOPSLA ’06: Proceedings of the 21st ACM SIGPLAN International Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 37–56, New York, NY, USA, 2006. ACM Press.