Worlds: Controlling the Scope of Side Effects

Alessandro Warth, Yoshiki Ohshima, Ted Kaehler, and Alan Kay

Viewpoints Research Institute
1209 Grand Central Ave.
Glendale, CA 91201
{alex,yoshiki,ted,alan}@vpri.org

Abstract. The state of an imperative program—e.g., the values stored in global and local variables, arrays, and objects' instance variables—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 by extending both JavaScript and Squeak Smalltalk with support for worlds, provide examples of some of the interesting idioms this construct makes possible, and formalize the semantics of property/field lookup in the presence of worlds. We also describe an efficient implementation strategy (used in our Squeak-based prototype), and illustrate the practical benefits of worlds with two case studies.

1 Introduction

Solutions to many problems in computing start with incomplete information and must gather more while the solution is in progress. An important class of problems have to perform *speculations* and *experiments*, often in parallel, to discover how to proceed. These include classical non-deterministic problems such as certain kinds of parsing, search and reasoning, dealing with potential and actual error conditions, doing, undoing, and redoing in user interfaces, supporting multiple forked versions of files and other structures that may need to be both ramified and retracted, etc. The "need to undo" operates at all levels of scale in computing and goes beyond simple backtracking to being able to support multiple speculative world-lines.

Most of the ploys historically used to deal with "undoing" have been ad hoc and incomplete. For example, features such as `try/catch` enable some speculation, but only unwind the stack on failure; side effects are not undone automatically. Programmers have little choice but to rely on error-prone idioms such as the command design pattern [13]. This is analogous to the manual storage management mechanisms found in low-level languages (e.g., `malloc` and `free` in C). In contrast, garbage collection trades a little efficiency for enormous safety and convenience, and the *worlds* mechanism we present in this paper provides a similar service for all levels of "doing-and-undoing."

Web surfing is a useful analogy for thinking about worlds: during a simple exploration of the web, you might just use the back button, but more complex explorations

M. Mezini (Ed.): ECOOP 2011, LNCS 6813, pp. 179–203, 2011.

(speculations) are more easily done with multiple tabs. All the changes you made during your explorations remain local to the tab that you used, and can be made "global" or not by your choice.

This is somewhat similar to *transactions*, which are another example of a general mechanism that can handle some of the "computing before committing" problems at hand here. But whereas the purpose of transactions is to provide a simple model for parallel programming, the goal of worlds is to provide a clean and flexible mechanism for *controlling the scope of side effects*. Unlike transactions, worlds are first-class values and are not tied to any particular control structure—a world can be stored in a variable to be revisited at a later time. This novel combination of design properties makes worlds more general (albeit more primitive) than transactions. For example, neither the module system shown in Section 3.3 nor the tree-undo feature presented in Section 6 could be implemented using transactions. Furthermore, we show in Section 8 that it is straightforward to implement transactions in a language that supports worlds.

The rest of paper is structured as follows. Section 2 introduces the notion of worlds and its instantiation in Worlds/JS. Section 3 illustrates some of the interesting idioms made possible by this construct. Section 4 details the semantics of property lookup in the presence of worlds. Section 5 describes the efficient implementation strategy used in our Squeak-based prototype. Sections 6 and 7 present two case studies, the first of which is used to benchmark the performance of our implementation. Section 8 compares worlds with related work, and Section 9 concludes.

2 Approach

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 and local variables, arrays, objects' instance variables, etc.—that happen inside it.

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.

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.

2.1 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.

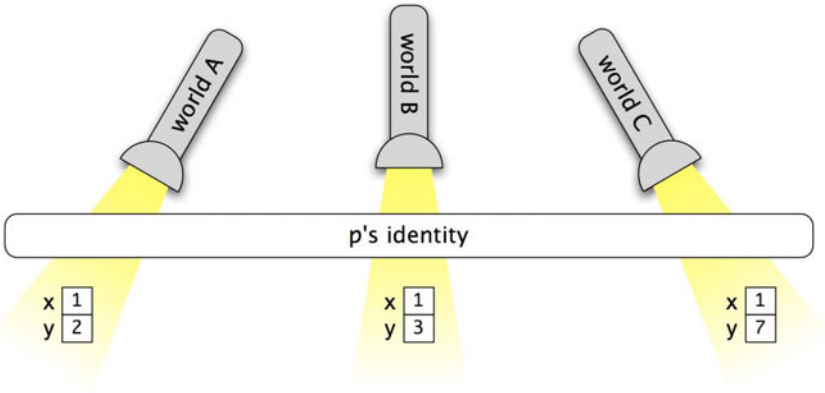

Fig. 1. Projections/views of the same object in three different worlds

We now describe the way in which these operations are supported in Worlds/JS, an extension of JavaScript [9] we have prototyped in order to explore with the ideas discussed in this paper. (Worlds/JS is available at `http://www.tinlizzie.org/ometa-js/#Worlds_Paper`. No installation is necessary; you can experiment with the language directly in your web browser.)

Worlds/JS extends JavaScript with the following new syntax:

- `thisWorld` — is an expression whose value is the world in which it is evaluated (i.e., the "current world"), and
- `in` <expr> <block> — is a statement that executes <block> inside the world obtained from evaluating <expr>.

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 `y` property of the same instance of `Point` 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 and the semantics of the *sprout* and *commit* operations.

```
A = thisWorld;
p = new Point(1, 2);

B = A.sprout();
in B { p.y = 3; }

C = A.sprout();
in C { p.y = 7; }

C.commit();
```

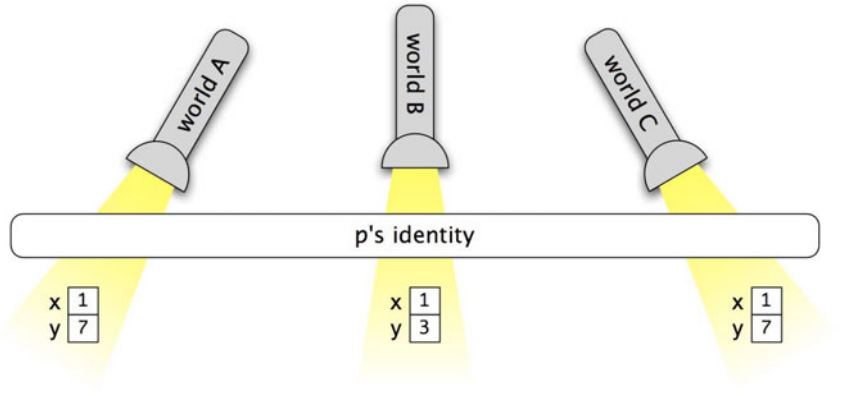

Fig. 2. The state of the "universe" shown in Figure 1 after a *commit* from world `C`

Figures 1 and 2 show the state of the point in each world, before and after the commit operation, respectively. Note that `p`'s identity is "universal," and each world associates it with `p`'s state in that world.

2.2 Safety Properties

Programming with worlds should not be error-prone or dangerous. In particular, if w_{child} is a world that was sprouted from w_{parent}:

- Changes in w_{parent}—whether *explicit* (caused by assignments in w_{parent} itself) or *implicit* (caused by a *commit* from one of w_{child}'s siblings)—should never make variables appear to change spontaneously in w_{child}. We call this the "*no surprises*" property.
- Similarly, a *commit* from w_{child} should never leave w_{parent} in an inconsistent state, e.g., because the changes being committed are incompatible with changes made in w_{parent} after w_{child} was sprouted. We call this property "*consistency*."

In this section, we explain how the semantics of worlds ensures these properties.

Preventing "Surprises". Once a variable (or slot, memory location, etc.) has been read or modified in a world w, subsequent changes to that variable in w's parent world are not visible in w. This ensures that variables do not appear to change spontaneously in child worlds.

For example, Figure 2 shows that the effects of the *commit* from world `C` (`p.y` ← `7`) are not visible in world `B` (because it has also modified `p.y`). However, if yet another world that is sprouted from `A` changes the value of `p.x` and then *commits*, as shown below,

```
D = A.sprout();
in D {
  p.x = 5;
}
D.commit();
```

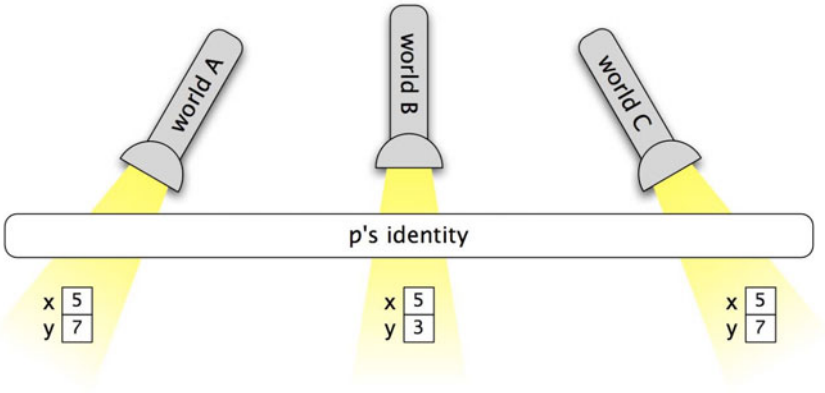

Fig. 3. The state of the "universe" shown in Figure 2 after a *commit* from world `D` (not pictured)

the new value of `p.x` becomes visible not only in `A`, because it is `D`'s parent, but also in `B` and `C`, since neither of them has read or modified `p.x`. (See Figure 3.) This is safe because up to this point:

- neither `B` nor `C` has read the old value of `p.x`, so they will not be able to tell that it has changed, and
- whatever writes have already been done in `B` and `C` (assuming the program is correct) are guaranteed not to depend on the value of `p.x`—otherwise `p.x` would have been read in those worlds.

Preserving Consistency. Note that the "no surprises" property does not prevent the state of a parent world from changing as a result of *commits* from its children—after all, the sole purpose of the commit operation is to change the state of the parent world. But not all such changes are safe: certain kinds of changes could leave the parent world in an inconsistent state. This is why the commit operation of worlds, like its counterpart in transaction-processing systems, is guarded by a *serializability check*.

A *commit* from w_{child} to w_{parent} is only allowed to happen if, at *commit*-time, all of the variables (or slots, memory locations, etc.) that were read in w_{child} have the same values in w_{parent} as they did when they were first read by w_{child}. If this is not the case, some of the assignments that were made in w_{child} may have been based on values that are now out of date. A *commit* that fails the serializability check is aborted, leaving both child and parent worlds unchanged, and throws a `CommitFailed` exception.

Section 5 describes the implementation of the commit operation, including the serializability check described here.

3 Worlds by Example

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

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) {
  // ...
}
```

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 [23] offer a solution to this problem by giving `try/catch` statements a transaction-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:

```
try {
  in thisWorld.sprout() {
    for (var idx = 0; idx < xs.length; idx++)
      xs[idx].update();
    thisWorld.commit();
  }
} catch (e) {
  // no clean-up required!
}
```

Note that the statements of the original `try` block are now evaluated in a new world that will capture their side effects. Note also that inside the `in` statement, the pseudo-variable `thisWorld` refers to this new world, and not its parent world. Therefore, if the loop terminates normally (i.e., without throwing an exception), the statement `thisWorld.commit();` will propagate the side effects to the parent world. On the other hand, if an exception is thrown, control will pass to the `catch` block before the commit

operation is executed, and thus the side effects will be discarded. (In fact, the new world will eventually be garbage-collected, since it is not referenced by any variables.)

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 same idea can be used as the basis of a framework that makes it easy for programmers to implement applications that support multi-level undo.

Applications built using this framework are objects that support two operations: `perform` and `undo`. Clients use the `perform` operation to issue commands to the application, and the `undo` operation to restore the application to its previous state (i.e., the state it was in before the last command was performed). The example below illustrates how a client might interact with a counter application that supports the commands `inc`, `dec`, and `getCount`, for incrementing, decrementing, and retrieving the counter's value, respectively. (The counter's value is initially zero.)

```
counter.perform('inc');
counter.perform('inc');
counter.perform('dec');
counter.undo();                       // undo 'dec'
print(counter.perform('getCount')); // outputs '2'
```

The interesting thing about our framework is that it allows programmers to implement applications that support multi-level undo *for free*, i.e., without having to use error-prone idioms such as the *command* design pattern [13]. The implementation of the counter application—or rather, a factory of counters—is shown below:

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

Note that the counter application is an instance of the `Application` class. `Application` *is* our framework; in other words, it is where all of the undo functionality is implemented. Its source code is shown in Figure 4.

The state of the application is always accessed in a world that "belongs" to the application. When the application is instantiated, it has only one world. Each time a client issues a command to the application via its `perform` operation, the method that corresponds to that command (the one with the same name as the command) is invoked 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 one in which the last command was executed).

```
Application = function() { };
Application.prototype = {
  worlds: [thisWorld],
  perform: function(command) {
    var w = this.worlds.last().sprout();
    this.worlds.push(w);
    in w { return this[command](); }
  },
  undo: function() {
    if (this.worlds.length > 0)
      this.worlds.pop();
  },
  flattenHistory: function() {
    while (this.worlds.length > 1) {
      var w = this.worlds.pop();
      w.commit();
    }
  }
};
```

Fig. 4. A framework for building applications that support multi-level undo

The `undo` operation simply discards the world in which the last command was executed, effectively returning the application to its previous state. Lastly, the (optional) `flattenHistory` operation coalesces the state of the application into a single world, which prevents clients from `undo`ing past the current state of the application.

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 [8].

Our Worlds/Squeak image includes a text editor implementation that supports multi-level undo using the idiom described in this section. It is available for download at `http://www.tinlizzie.org/~awarth/worlds`.

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 `fact`orial 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 in JavaScript) with new methods that support the needs of their particular application. This practice is informally known as *monkey-patching* [4]. Monkey-patching is generally frowned upon because, in addition to polluting the interfaces of the objects involved, it makes programs vulnerable to name clashes that are impossible to anticipate. Certain module systems, including those of MultiJava [6] and eJava [33], 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 form of modularity:

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

The `fact`orial method defined above can only be used inside `ourModule`, e.g.,

```
in ourModule {
  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 [3,2,7] 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 `ourModule`:

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

and now the output generated by

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

is

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

A more detailed discussion of how worlds can be used to implement a module system for JavaScript, including a variation of the idiom described above that supports side-effectful extension methods, can be found in the first author's Ph.D. dissertation [31].

4 Property Lookup Semantics

This section formally describes the semantics of property lookup in Worlds/JS, which is a natural generalization of property lookup in JavaScript. We do not formalize the semantics of field lookup in Worlds/Squeak, since it is just a special case of the former in which all prototype chains have length 1 (i.e., there is no delegation).

4.1 Property Lookup in JavaScript

JavaScript's object model is based on single delegation, which means that every object inherits (and may also override) the properties of its "parent" object. The only exception to this rule is `Object.prototype` (the ancestor of all objects), which is the root of JavaScript's delegation hierarchy and therefore does not delegate to any other object.

The semantics of property lookup in JavaScript can be formalized using the following two primitive operations:

(i) $\mathit{getOwnProperty}(x, p)$, which looks up property p in object x *without looking up the delegation chain*. More specifically, the value of $\mathit{getOwnProperty}(x, p)$ is
  - v, if x has a property p *that is not inherited from another object*, and whose value is v, and
  - the special value **none**, otherwise;

(ii) $\mathit{parent}(x)$, which evaluates to
  - y, the object to which x delegates, or
  - the special value **none**, if x does not delegate to any other object.

and the following set of inference rules:

$$\frac{\mathit{getOwnProperty}(x, p) = v \qquad v \neq \textbf{none}}{\mathit{lookup}(x, p) = v} \qquad \text{(JS-LOOKUP-OWN)}$$

$$\frac{\mathit{getOwnProperty}(x, p) = \textbf{none} \qquad \mathit{parent}(x) = \textbf{none}}{\mathit{lookup}(x, p) = \textbf{none}} \qquad \text{(JS-LOOKUP-ROOT)}$$

$$\frac{\mathit{getOwnProperty}(x, p) = \textbf{none} \qquad \mathit{parent}(x) = y \qquad y \neq \textbf{none} \qquad \mathit{lookup}(y, p) = v}{\mathit{lookup}(x, p) = v} \qquad \text{(JS-LOOKUP-CHILD)}$$

4.2 Property Lookup in Worlds/JS

In Worlds/JS, property lookup is always done in the context of a world. And since it may be that an object x has a property p in some world w but not in another, the primitive operation $getOwnProperty(x,p)$ must be replaced by a new primitive operation, $getOwnPropertyInWorld(x,p,w)$.

Another primitive operation we will need in order to formalize the semantics of property lookup in Worlds/JS is $parentWorld(w)$, which yields w's parent, or the special value **none**, if w is the top-level world.

Using these two new primitive operations, we can define a new operation, $getOwnProperty(x,p,w)$, which yields the value of x's p property in world w, or (if $x.p$ is not defined in w) in w's closest ancestor:

$$\frac{getOwnPropertyInWorld(x,p,w) = v \quad v \neq \textbf{none}}{getOwnProperty(x,p,w) = v} \text{ (WJS-GETOWNPROPERTY-OWN)}$$

$$\frac{getOwnPropertyInWorld(x,p,w) = \textbf{none} \quad parentWorld(w) = \textbf{none}}{getOwnProperty(x,p,w) = \textbf{none}} \text{ (WJS-GETOWNPROPERTY-ROOT)}$$

$$\frac{getOwnPropertyInWorld(x,p,w_1) = \textbf{none} \quad parentWorld(w_1) = w_2 \quad w_2 \neq \textbf{none} \quad getOwnProperty(x,p,w_2) = v}{getOwnProperty(x,p,w_1) = v} \text{ (WJS-GETOWNPROPERTY-CHILD)}$$

And finally, using the worlds-friendly variant of $getOwnProperty$ defined above, the inference rules that formalize the semantics of lookup in Worlds/JS can be written as follows:

$$\frac{getOwnProperty(x,p,w) = v \quad v \neq \textbf{none}}{lookup(x,p,w) = v} \text{ (WJS-LOOKUP-OWN)}$$

$$\frac{getOwnProperty(x,p,w) = \textbf{none} \quad parent(x) = \textbf{none}}{lookup(x,p,w) = \textbf{none}} \text{ (WJS-LOOKUP-ROOT)}$$

$$\frac{getOwnProperty(x,p,w) = \textbf{none} \quad parent(x) = y \quad y \neq \textbf{none} \quad lookup(y,p,w) = v}{lookup(x,p,w) = v} \text{ (WJS-LOOKUP-CHILD)}$$

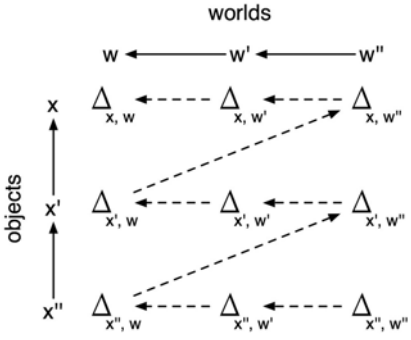

Fig. 5. The property lookup order used when evaluating $x''.p$ in world w'' (the notation $\Delta_{x,w}$ represents the properties of x that were modified in w)

Note that these rules closely mirror those that describe the semantics of lookup in JavaScript—the only difference is that *getOwnProperty* and *lookup* now each take a world as an additional argument.

Figure 5 illustrates the property lookup order that results from the algorithm described above. The solid vertical lines in the diagram indicate *delegates-to* relationships (e.g., object x' delegates to x), whereas the solid horizontal lines indicate *is-child-of* relationships (e.g., world w' is a child of w). Note that the chain of worlds gets precedence over the object delegation chain; in other words, any relevant "version" of an object may override the properties of the object to which it delegates. This lookup order preserves JavaScript's copy-on-write delegation semantics, i.e., if a delegates to b, and then we assign into a's p property, subsequent changes to b's p property will not affect a. So no matter what world a statement is executed in—whether it is the top-level world, or a world that sprouted from another world—it will behave in exactly the same way as it would in "vanilla" JavaScript.

5 Implementation

Our Worlds/JS prototype works by translating Worlds/JS programs into JavaScript code that can be executed directly in a standard web browser, and it is useful for doing "quick and dirty" experiments. However, JavaScript's lack of support for weak references required a costly work-around that makes Worlds/JS unsuitable for larger experiments.

In this section, we describe the implementation strategy used in our more performant prototype of worlds, which is based on Squeak Smalltalk [16]. (The Worlds/Squeak image, which also includes the case study discussed in Section 6, is available for download at `http://www.tinlizzie.org/~awarth/worlds`.)

5.1 Data Structures

The core of our implementation consists of two classes: `WObject`, which is the superclass of all "world-friendly" objects (i.e., objects that exhibit the correct behavior when

viewed/modified inside worlds), and `WWorld`, which represents worlds. The methods of `WObject` and its subclasses are automatically instrumented in order to indirect all instance variable accesses (reads and writes) through the world in which they are being evaluated.

A `WWorld` w contains a hash table that, similar to a *transaction log*, associates `WObjects` with

- a *reads* object, which holds the "old" values of each slot of the `WObject` when it was first read in w, or the special `Don'tKnow` value for each slot that was never read in w, and
- a *writes* object, which holds the most recent values of each slot of the `WObject`, or the special `Don'tKnow` value for slots that were never written to in w.

The keys of this hash table are referenced weakly to ensure that the *reads* and *writes* objects associated with a `WObject` that is no longer referenced by the program will be garbage collected. Also, *reads* and *writes* objects are instantiated lazily, so (for example) an object that has been read but not written to in a world will have a *reads* object, but not a *writes* object, in that world.

5.2 The Slot Update Operation: $(x_i \leftarrow v)_w$

To store the value v in x's i^{th} slot in world w,

1. If w does not already have a *writes* object for x, create one.
2. Write v into the i^{th} slot of the *writes* object.

5.3 The Slot Lookup Operation: $(x_i)_w$

To retrieve the value stored in x's i^{th} slot in world w,

1. Let $w_{current} = w$ and $ans = undefined$.
2. If $w_{current}$ has a *writes* object for x and the value stored in the i^{th} slot of the *writes* object is not `Don'tKnow`, set *ans* to that value and go to step 5.
3. If $w_{current}$ has a *reads* object for x and the value stored in the i^{th} slot of the *reads* object is not `Don'tKnow`, set *ans* to that value and go to step 5. (This step ensures the "no surprises" property, i.e., that a slot value does not appear to change spontaneously in w when it is updated in one of w's ancestors.)
4. Otherwise, set $w_{current}$ to $w_{current}$'s parent, and go to step 2.
5. If w does not already have a *reads* object for x, create one.
6. If the value stored in the i^{th} slot of the *reads* object is `Don'tKnow`, write *ans* into that slot.
7. Return *ans*.

Note that the slots of a new `WObject` are always initialized with `nils` in the top-level world. This mirrors the semantics of object instantiation in Smalltalk and ensures that lookup always terminates.

(We initially implemented the slot lookup operation in Smalltalk, but later re-implemented it as a primitive, which resulted in a significant performance improvement. See Section 6.3 for details.)

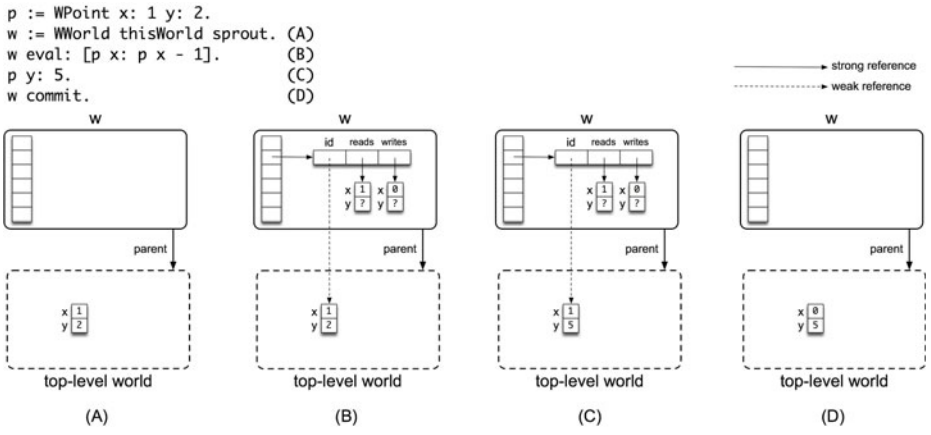

Fig. 6. A successful commit

5.4 Reads and Writes in the Top-Level World

The top-level world has no need for *reads* objects, since it has no parent world to commit to. This gave us an idea: if we made every `WObject` be its own *writes* object in the top-level world, there would be no need for a hash table. This optimization minimizes the overhead of using `WObjects` in the top-level world—after all, the slot update and lookup operations can manipulate the `WObject` directly, without the additional cost of a hash table lookup.

5.5 The Commit Operation

To commit the side effects captured in world w_{child} to its parent, w_{parent},

1. **Serializability check:** for all $x_i = v$ in each of w_{child}'s *reads* objects, make sure that either $v =$ `Don'tKnow` or the current value of x_i in w_{parent} is equal to v. (Otherwise, throw a `CommitFailed` exception.)
2. Propagate all of the information in w_{child}'s *writes* objects to w_{parent}'s *writes* objects, overriding the values of any slots that have already been assigned into in w_{parent}.
3. Propagate all of the information in w_{child}'s *reads* objects to w_{parent}'s *reads* objects, except for the slots that have already been read from in w_{parent}. (This step ensures that the serializability check associated with a later *commit* from w_{parent} will protect the consistency of its own parent.)
4. Clear w_{child}'s hash table.

Note that in step 2, new *writes* objects must be created for any objects that were written to in w_{child}, but not in w_{parent}. Similarly, in step 3, new *reads* objects must be created for any objects that were read from in w_{child}, but not in w_{parent} (unless w_{parent} is the top-level world, which, as discussed in Section 5.4, has no need for *reads* objects).

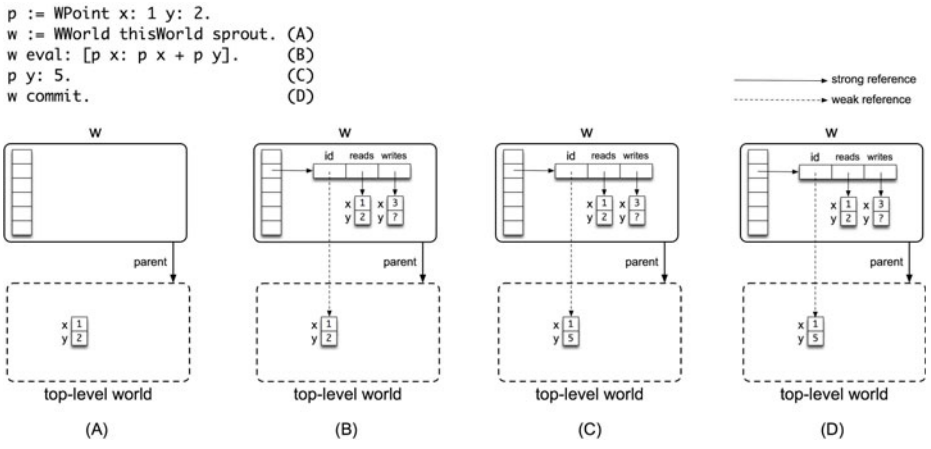

Fig. 7. A failed commit

5.6 Pulling It all Together

The Worlds/Squeak code and diagrams in Figures 6 and 7 show these data structures and operations in action.

In Figure 6 (A), we see a point `p`[1] and a world `w` that was sprouted from the top-level world. In (B), we see what happens when we decrement `p`'s `x` field in `w`. (`WWorld>>eval:` is equivalent to the `in`-statement in Worlds/JS.) Note that this step results in the creation of a hash table entry for `p`. (Note also that the key used to index the hash table is `p` itself—the identity of an object never changes, no matter what world it's in.) The old (`1`) and new (`0`) values of `x` are stored in `p`'s *reads* and *writes* objects, respectively; the question marks denote the `Don'tKnow` value. In (C), `p`'s `y` field is updated in the top-level world with the value `5`. Lastly, (D) shows the effects of a commit on `w`. The old values stored in the *reads* objects in the child world (in this case there is only one, `p.x = 1`) are compared against the corresponding (current) values in the parent world. Since these are equal, the changes that were stored in the child world (`p.x` $\leftarrow$ `0`) are propagated to the parent world. Note that a successful *commit* does not cause the child world to be discarded; it simply clears the information stored in that world so that it can be used again.

Figure 7 illustrates a slightly different senario. In (B), both `x` and `y` are read in order to compute the new value of `x` and as a result, both values are recorded in the *reads* object. In (C), `y` is updated in the top-level world. In (D), `w` tries to commit but fails the serializability check because the value of `y` that was recorded in the *reads* object is different from the current value of `y` in the top-level world. This results in a `CommitFailed` exception, and the commit operation is aborted. Note that (C) and (D) are identical—a failed *commit* leaves both the parent and child worlds unchanged. This enables the programmer to examine (and also extract) potentially useful values in the child world

[1] `p` is an instance of `WPoint`, which is a subclass of `WObject`.

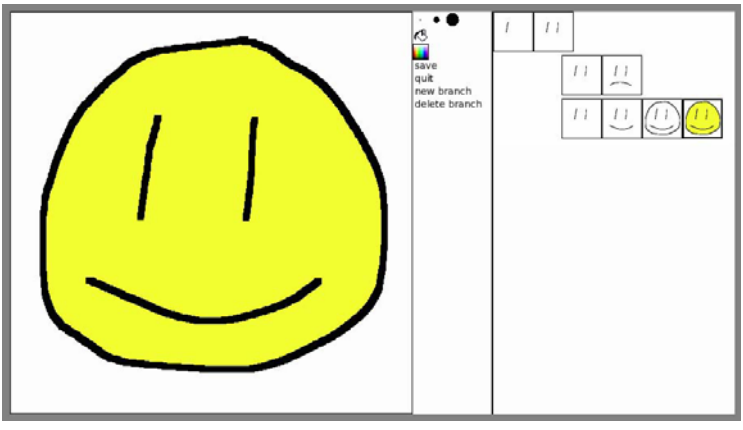

Fig. 8. Our bitmap editor, implemented in Worlds/Squeak, supports tree-undo

using its `#eval:` method. The programmer may also start anew by sprouting a new child world.

6 Case Study #1: A Bitmap Editor

To gauge the utility of worlds in practical applications, we have implemented a bitmap editor that supports a sophisticated tree-undo mechanism. This is an interesting challenge for worlds because it involves large bitmap objects ($\sim$1M slots per bitmap) that must not only be manipulated interactively and efficiently, but also passed to Squeak's graphics primitives, which do not support worlds.

Applications that support (traditional) linear undo allow users to undo/redo a series of actions. *Tree-undo* is a more general model for undo that enables users to create new "branches" in the edit history and move from freely from branch to branch, which in turn makes it convenient for different choices/possibilities to be explored concurrently.

Figure 8 shows the Graphical User Interface (GUI) of our bitmap editor. To the right of the large painting area there are several tool-selecting buttons: three different brushes, a bucket fill tool, and a color picker. Below these buttons, in addition to the self-explanatory "save" and "quit," there are buttons for managing the undo tree: "new branch" creates a new branch that stems from the current undo record, and "delete branch" discards the current branch. Last but not least, to the right of these buttons is a view of the undo tree in which each thumbnail represents an undo record (the one with the thick border is the current undo record), and each row of thumbnails represents a sequence of states in the same branch.

While our bitmap editor (which comprises approximately 300 lines of Lesserphic2/Squeak code) is not particularly feature-rich, it is interesting enough to serve as a benchmark for measuring the performance of Worlds/Squeak.

6.1 The Implementation of Tree Undo

As illustrated in Section 3.2, the semantics of worlds provides a good basis for implementing the undo mechanism: all we have to do is create a chain of worlds in which to capture changes to the state of the application. The undo operation, then, involves merely switching the world from which we view the state of the application. This idiom gives the application designer complete control over what parts of the application's state should be undoable (i.e., what parts are always modified in and viewed from a world in the undo chain), and the granularity of the undo operation (i.e., how often new worlds, or "undo records," are created).

In our bitmap editor, the only piece of application state we chose to make undoable was the bitmap itself. So whenever the user performs an action that modifies the bitmap, e.g., a pen stroke, we sprout a new world from the current world, and make the appropriate changes to the bitmap in that new world. After drawing three strokes for example, we are left with a chain of four worlds (the first represents the state of the bitmap before the first stroke).

Extending this idiom to support tree-undo was straightfoward. To make a new branch, we simply sprout a new world from the current world, and to delete a branch, we remove all references to the root of that branch (as a result, its undo records and their associated worlds will eventually be garbage-collected). Our program must maintain undo records, which are data structures that represent the branches of the undo tree, because a world does not retain a list of its children (after all, this would prevent worlds from ever being garbage-collected, since every world is a descendant of the top-level world).

6.2 Bitmap Representation

We wanted our editor to feel "smooth." A user should, for example, be able to edit a 500×500-pixel bitmap while the screen is updated 30 times per second. This meant that our application had to sustain 7.5M slot lookup operations per second just to display the bitmap. Our first (naive) implementation of Worlds/Squeak, in which `WObject`'s lookup operation was implemented entirely in Smalltalk, did not meet these requirements: while drawing a line on a 2.4 GHz Core Duo computer, we observed a refresh rate of about 1 fps—far from usable in any reasonable standard.

To increase the performance of our application, we created a variable-length, non-pointer subclass of `WObject` called `WBitmap`. We made `WBitmaps` structurally identical to Squeak's `Bitmaps` so that the BitBlt primitives could be used to draw on `WBitmaps`.[2] We implemented "write-only" features like line drawing, for example, by passing the `WBitmap`'s "delta" to a BitBlt primitive that efficiently mutates it as desired. (By modifying the delta, we ensure that the changes are only visible in the appropriate world.)

We also realized that `WObject`'s slot lookup method could be implemented as a Squeak primitive; in fact, we took this idea one step further and implemented a primitive that reads *all of the slots* of a `WObject`, and returns a "flattened snapshot" of that object, as shown in Figure 9. Note that, because all of its state is stored in a single Squeak object (i.e., in a contiguous block of memory), a flattened `WObject` can be used

[2] Some primitives turned out to have strict type checks that prevented us from using them.

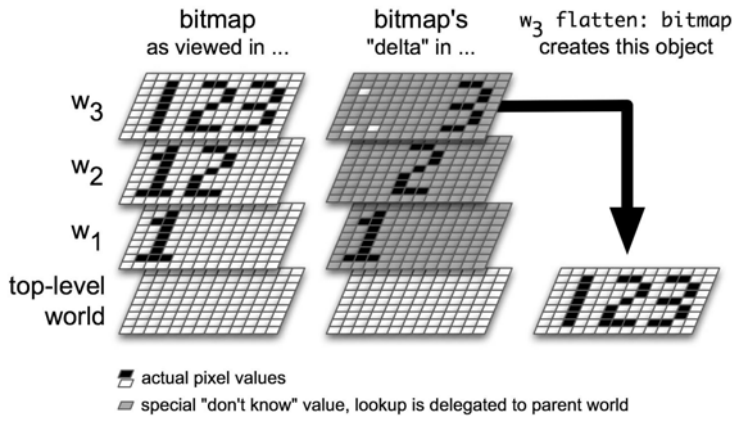

Fig. 9. The flatten operation

to interface with parts of the system that are not "worlds-aware," including Squeak's graphics system. To flood-fill an area of a `WBitmap`, for instance, we pass the bitmap's delta *and* a flattened snapshot to a primitive; the flattened object is used for reading values out of the bitmap, and the delta is used as a target for the writes.

The downside of representing `WBitmaps` as arrays of "immediate" (i.e., non-pointer) values is that it makes it impossible for their slots to reference the special `Don'tKnow` value that causes lookup to be delegated to the parent world—after all, every 4-byte value is a valid (ARGB) pixel value. We got around this problem by using the value "0x00000001" (the darkest, completely transparent blue) as `WBitmap`'s equivalent of `WObject`'s `Don'tKnow` value. A more general work-around—which would be necessary for applications that require access to all possible 4-byte values—would be to use an additional bit array (one bit per slot in the object) whose values indicate whether or not a slot in a "delta" or "orig" object contains a valid value.

6.3 Benchmarking the Bitmap Editor

After we introduced the flatten primitive, our bitmap editor became **really responsive**: while editing a 500×500-pixel bitmap on the same 2.4 GHz Core Duo computer, the frame rate only drops below 30 once the length of the delegation chain—i.e., the distance between the current undo record and the root of the undo tree—reaches about 50. (At this point, a casual user may start to notice a slowdown.)

In order to get a better idea of the performance characteristics of our bitmap editor (and of our Squeak-based implementation of worlds), we conducted the following experiment. We started with a 20,000-pixel bitmap, created a chain of worlds, and measured the time it takes to read all of the slots (pixel values) of the bitmap from the world at the end of the chain. Looking up a slot's value requires a traversal of the world chain until a valid (non-`Don'tKnow`) value is found, so the "load factor," i.e., the number of value-holding slots at each level, dictates the typical length a lookup operation, which in turn determines the performance of the application.

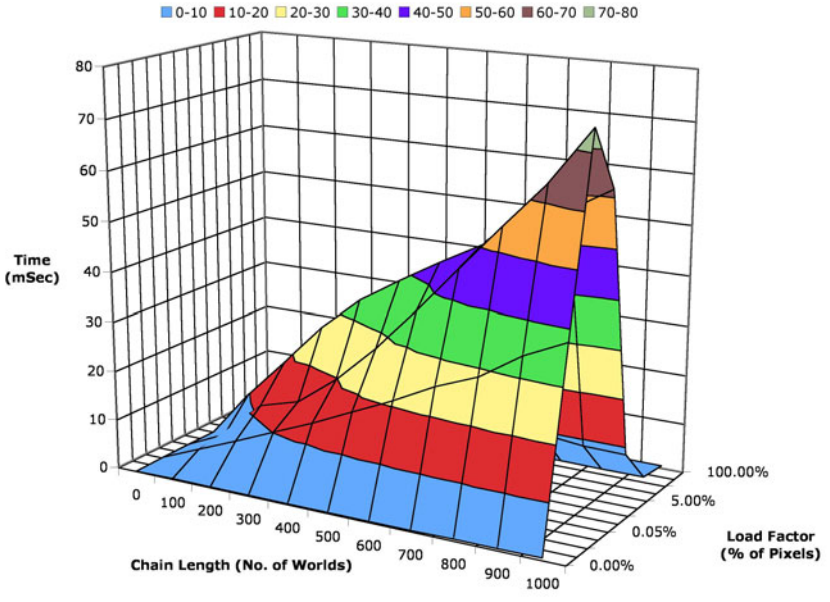

Fig. 10. time to flatten a 20,000-slot bitmap

Figure 10 shows the results of this experiment. When the load factor is close to zero, the lookup primitive must nearly always scan all of the worlds in the chain, and as a result, time grows linearly with the number of worlds in the chain. When the load factor is high, e.g., when 50% of the slots are filled randomly at each level, the lookup primitive only has to inspect a few worlds—two, on average—so the length of the chain has no measurable effect on the performance of our application.

The "lookup" and "flatten" primitives provide significant performance improvements. Without them, reading every slot of a 500×500-pixel bitmap through a chain of 1000 worlds takes 27.17 seconds and 137 milliseconds with a load factor of 0.005% and 50%, respectively. Using the "lookup" primitive to access each slot reduces these times to 0.36 seconds and 10 milliseconds, respectively. Using the "flatten" primitive further reduces these times to 0.038 seconds and less than 1 millisecond, respectively. On average, the use of these primitives improved the performance of our application by two orders of magnitude.

7 Case Study #2: OMeta + Worlds

Consider the semantics of the ordered choice operator (|) in OMeta [32], or if you prefer, in Parsing Expression Grammars and Packrat Parsers [11,12]. If a match fails while the first operand is being evaluated, the parser, or more generally, the *matcher* has to backtrack to the appropriate position on the input stream before trying the second operand. We can think of this backtracking as a limited kind of undo that is only

```
OMeta._or = function() {
  var origInput = this.input;
  for (var idx = 0;
       idx < arguments.length;
       idx++) {


    try {
      this.input = origInput;
      return arguments[idx]();
    }
    catch (f) {

      if (f != fail)
        throw f;
    }




  throw fail;
};
```

(A)

```
OMeta._or = function() {

  for (var idx = 0;
       idx < arguments.length;
       idx++) {
    var ok = true;
    in thisWorld.sprout() {
      try {

        return arguments[idx]();
      }
      catch (f) {
        ok = false;
        if (f != fail)
          throw f;
      }
      finally {
        if (ok)
          thisWorld.commit();
      }
    }
  }
  throw fail;
};
```

(B)

Fig. 11. Two different implementations of OMeta's ordered choice operator

concerned with changes to the matcher's position on the input stream. Other kinds of side effects that can be performed by semantic actions—e.g., destructive updates such as assigning into one of the fields of the matcher object or a global variable—are not undone automatically, which means that the programmer must be specially careful when writing rules with side effects.

To show that worlds can greatly simplify the management of state in backtracking programming languages like OMeta and Prolog, we have implemented a variant of OMeta/JS [30] in which the choice operator automatically discards the side effects of failed alternatives, and similarly, the repetition operator (*) automatically discards the side effects of the last (unsuccessful) iteration. This was modification straightforward: since Worlds/JS is a superset of JavaScript—the language in which OMeta/JS was implemented—all we had to do was redefine the methods that implement the semantics of these two operators.

Figures 11 (A) and (B) show the original and modified implementations of the ordered choice operator, respectively. Note that the modified implementation sprouts a new world in which to evaluate each alternative, so that the side effects of failed alternatives can be easily discarded. These side effects include the changes to the matcher's input stream position, and therefore the code that implemented backtracking in the

original version (`this.input = origInput`) is no longer required. Finally, the side effects of the first successful alternative are committed to the parent world in the `finally` block.

Similarly, the alternative implementation of the repetition operator (omitted for brevity) sprouts a new world in which to try each iteration, so that the effects of the last (unsuccessful) iteration can be discarded.

And thus, with very little additional complexity, worlds can be used to make the use of side effects safer and easier to reason about in the presence of backtracking.

8 Related Work

In languages that support Software Transactional Memory (STM) [27,14], every transaction that is being executed at a given time has access to its own view of the program store that can be modified in isolation, without affecting other transactions. Therefore, like worlds, STM enables multiple versions of the store to co-exist. But whereas the purpose of transactions is to provide a simple model for parallel programming, the goal of worlds is to provide a clean and flexible mechanism for *controlling the scope of side effects*. Unlike transactions, worlds are first-class values and are not tied to any particular control structure—a world can be stored in a variable to be revisited at a later time. This novel combination of design properties makes worlds more general (albeit more primitive) than transactions. For example, neither the module system shown in Section 3.3 nor the tree-undo feature presented in Section 6 could be implemented using transactions. Furthermore, in a language that supports threads, it would be straightforward to implement the semantics of STM using worlds:

```
executeInNewThread {
  in thisWorld.sprout() {
    ... // statements to run in the transaction
    thisWorld.commit();
  }
}
```

(The code above assumes that commits are executed atomically.)

Burckhardt et al.'s recent work on concurrent programming with revisions and isolation types [5] provides a model in which programmers can declare what data they wish to share between tasks (threads), and execute tasks concurrently by forking and joining revisions. While a revision's `rjoin` operation provides similar functionality to a world's `commit` operation, there are some important differences. For example, unlike `rjoin`, the `commit` operation detects read-write conflicts (these result in a `CommitFailed` exception). Also, "revisions do not guarantee serializability ... but provide a different sort of isolation guarantee" and the authors "posit that programmers, if given the right abstraction, are capable of reasoning about concurrent executions directly." [5]. While this may indeed be the case, we believe that serializability (which is supported by worlds) makes for a much more understandable programming model.

The idea of treating the program store as a first-class value and enabling programmers to take snapshots of the store which could be restored at a later time first appeared

in Johnson and Duggan's GL programming language [18].This model was later extended by Morrisett to allow the store to be partitioned into a number of disjoint "substores" (each with its own set of variables) that could be saved and restored separately [22].

The main difference between previous formulations of first-class stores and worlds lies in the programming model: whereas first-class stores have until now been presented as a mechanism for manipulating *a single store* (or multiple partitions of the store, as is the case with Morrisett's work) through a save-and-restore interface, worlds enable multiple versions of the store—several "parallel universes"—to co-exist in the same program. This makes worlds a better fit for "programming via experiments", i.e., a programming style that involves experimenting with multiple alternatives, sometimes making mistakes that require retraction in order to make fresh starts. (This is difficult to do in mainstream imperative programming languages, due to the unconstrained nature of side effects.) It also makes it possible for multiple "experiments" to be carried out in parallel, which is something we intend to investigate in future work.

Tanter has shown that values that vary depending on the context in which they are accessed or modified—(implicitly) *contextual values*—can be used to implement a *scoped assignment* construct that enables programmers to control the scope of side effects [29]. Although Tanter's construct does not support the equivalent of the commit operation on worlds, it is more general than worlds in the sense that it allows any value to be used as a context. However, it is not clear whether this additional generality justifies the complexity that it brings to the programming model. For example, while it is straightforward to modify a group of variables in the context of the current thread (e.g., thread id 382), or the current user (e.g., awarth), it is difficult to reason about the state of the program when both contexts are active, since they need not be mutually exclusive. (This is similar to the semantic ambiguities that are caused by multiple inheritance in object-oriented languages.)

Smith and Ungar's *Us* [28], a predecessor of today's Context-Oriented Programming (COP) languages [15], explored the idea that the state and behavior of an object should be a function of the perspective from which it is being accessed. These perspectives, known as *layers*, were very similar to worlds (they were first-class objects that provided a context in which to evaluate expressions) but did not support the equivalent of the *commit* operation.

Worlds enable programmers to enjoy the benefits of Functional Data Structures (FDSs) [24] without having to plan for them ahead of time. For instance, the example in Section 3.1 (Better Support for Exceptions) would require whatever data is modified in the `try` block to be represented as a FDS, which is inconvenient and potentially very time-consuming for the programmer. To make matters worse, the `try` block might call a function—in this case, the programmer would have to (non-modularly) inspect the code that implements that function in order to find out what data it may modify so that she can change it into a FDS. With worlds, none of this is necessary.

Lastly, a number of mechanisms for synchronizing distributed and decentralized systems (e.g., TeaTime [25,26] and Virtual Time / Time Warp [17]) and optimistic methods for concurrency control [21] rely on the availability of a rollback (or undo) operation. As shown in Section 3.2, a programming language that supports worlds greatly sim-

plifies the implementation of rollbacks, and therefore could be a suitable platform for building these mechanisms.

9 Conclusions and Future Work

We have introduced worlds, a new language construct that enables programmers to control the scope of side effects. We have instantiated our notion of worlds in Worlds/JS and Worlds/Squeak (extensions of JavaScript and Squeak Smalltalk, respectively), formalized the semantics of property/field lookup in these languages, and shown that this construct is useful in a wide range of applications. We have also described the efficient implementation strategy that was used in our Squeak-based prototype.

We believe that worlds have the potential to provide a tractable programming model for multi-core architectures. As part of the STEPS project [19,20], we intend to investigate the feasibility of an even more efficient (possibly hardware-based) implementation of worlds that will enable the kinds of experiments that might validate this claim. For example, 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. Our hardware-based implementation should 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.

Also as part of the STEPS project, we intend to build a multimedia authoring system that supports "infinite" undo. Our bitmap editor and text editor (referenced in Section 3.2) are two distinct examples of undo, but we hope use worlds to implement a data model for all media types.

The top-level world's commit operation, which is currently a no-op, might be an interesting place to explore the potential synergy between worlds and persistence. For example, a different implementation of `TopLevelWorld>>commit` that writes the current state of every object in the system to disk (to be retrieved at a later time) could be the basis of a useful checkpointing mechanism.

One current limitation of worlds is that they only capture the *in-memory* side effects that happen inside them. Programmers must therefore be careful when executing code that includes other kinds of side effects, e.g., sending packets on the network and obtaining input from the user. It would be interesting to investigate whether some of the techniques used in reversible debuggers such as EXDAMS [1] and IGOR [10] could be used to ensure that, for example, when two sibling worlds read a character from the console, they get the same result.

Acknowledgments. The authors would like to thank Gilad Bracha, Marcus Denker, Robert Hirschfeld, Mark Miller, Todd Millstein, Eliot Miranda, James Noble, Jens Palsberg, David Reed, Hesam Samimi, and the anonymous reviewers for their useful feedback, and Bert Freudenberg for writing the Slang code that enabled our Squeak primitives to query `IdentityDictionary`s.

References

1. Balzer, R.M.: EXDAMS—EXtendable debugging and monitoring system. AFIPS Spring Joint Computer Conference 34, 567–580 (1969)
2. Bergel, A., Ducasse, S., Nierstrasz, O.: Classbox/J: Controlling the scope of change in Java. In: OOPSLA 2005: Proceedings of 20th International Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 177–189. ACM Press, New York (2005)
3. Bergel, A., Ducasse, S., Wuyts, R.: Classboxes: A minimal module model supporting local rebinding. In: Böszörményi, L., Schojer, P. (eds.) JMLC 2003. LNCS, vol. 2789, pp. 122–131. Springer, Heidelberg (2003)
4. Bracha, G.: Monkey Patching (blog post) (2008), `http://gbracha.blogspot.com/2008/03/monkey-patching.html`
5. Burckhardt, S., Baldassion, A., Leijen, D.: Concurrent programming with revisions and isolation types. In: OOPSLA 2010 (October 2010)
6. Clifton, C., Millstein, T., Leavens, G.T., Chambers, C.: MultiJava: Design rationale, compiler implementation, and applications. ACM Transactions on Programming Languages and Systems 28(3) (May 2006)
7. Denker, M., Gîrba, T., Lienhard, A., Nierstrasz, O., Renggli, L., Zumkehr, P.: Encapsulating and exploiting change with changeboxes. In: ICDL 2007: Proceedings of the 2007 International Conference on Dynamic Languages, pp. 25–49. ACM Press, New York (2007)
8. Ducasse, S., Lienhard, A., Renggli, L.: Seaside: A flexible environment for building dynamic web applications. IEEE Software 24(5), 56–63 (2007)
9. ECMA International. ECMA-262: ECMAScript Language Specification. European Association for Standardizing Information and Communication Systems, Geneva, Switzerland, third edition (December 1999)
10. Feldman, S.I., Brown, C.B.: IGOR: a system for program debugging via reversible execution. ACM SIGPLAN Notices 24(1), 112–123 (1989)
11. Ford, B.: Packrat parsing: simple, powerful, lazy, linear time, functional pearl. In: ICFP 2002: Proceedings of the seventh ACM SIGPLAN International Conference on Functional Programming, pp. 36–47. ACM Press, New York (2002)
12. Ford, B.: Parsing expression grammars: a recognition-based syntactic foundation. In: POPL 2004: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 111–122. ACM Press, New York (2004)
13. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison-Wesley Professional, Reading (1995)
14. Hårris, T., Marlow, S., Peyton-Jones, S., Herlihy, M.: Composable memory transactions. In: PPoPP 2005: Proceedings of the tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 48–60. ACM Press, New York (2005)
15. Hirschfeld, R., Costanza, P., Nierstrasz, O.: Context-Oriented Programming. Journal of Object Technology (JOT) 7(3), 125–151 (2008)
16. Ingalls, D., Kaehler, T., Maloney, J., Wallace, S., Kay, A.: Back to the future: the story of Squeak, a practical Smalltalk written in itself. SIGPLAN Notices 32(10), 318–326 (1997)
17. Jefferson, D.R.: Virtual time. ACM Transactions on Programming Languages and Systems 7(3), 404–425 (1985)
18. Johnson, G.F., Duggan, D.: Stores and partial continuations as first-class objects in a language and its environment. In: POPL19'88: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 158–168. ACM Press, New York (1988)

19. Kay, A., Ingalls, D., Ohshima, Y., Piumarta, I., Raab, A.: Proposal to NSF (2006), http://www.vpri.org/pdf/NSF_prop_RN-2006-002.pdf (granted on August 31, 2006)
20. Kay, A., Piumarta, I., Rose, K., Ingalls, D., Amelang, D., Kaehler, T., Ohshima, Y., Thacker, C., Wallace, S., Warth, A., Yamamiya, T.: Steps toward the reinvention of programming (first year progress report) (2007), http://www.vpri.org/pdf/steps_TR-2007-008.pdf
21. Kung, H.T., Robinson, J.T.: On optimistic methods for concurrency control. ACM Transactions on Database Systems 6(2), 213–226 (1981)
22. Morrisett, J.G.: Generalizing first-class stores. In: SIPL 1993: Proceedings of the ACM SIGPLAN Workshop on State in Programming Languages, pp. 73–87 (1993)
23. Nandivada, V.K., Jagannathan, S.: Dynamic state restoration using versioning exceptions. Higher-Order and Symbolic Computation 19(1), 101–124 (2006)
24. Okasaki, C.: Purely functional data structures. Cambridge University Press, Cambridge (1999)
25. Reed, D.P.: Naming and synchronization in a decentralized computer system (PhD dissertation). Technical Report TR-205, Massachusetts Institute of Technology, Cambridge, MA, USA (1978)
26. Reed, D.P.: Designing Croquet's TeaTime: a real-time, temporal environment for active object cooperation. In: OOPSLA 2005: Companion to the 20th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 7–7. ACM Press, New York (2005)
27. Shavit, N., Touitou, D.: Software transactional memory. In: PODC 1995: Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing, pp. 204–213 (1995)
28. Smith, R.B., Ungar, D.: A simple and unifying approach to subjective objects. Theor. Pract. Object Syst. 2(3), 161–178 (1996)
29. Tanter, E.: Contextual values. In: DLS 2008: Proceedings of the 2008 Dynamic Languages Symposium, pp. 1–10. ACM Press, New York (2008)
30. Warth, A.: OMeta/JS website, http://www.tinlizzie.org/ometa-js/
31. Warth, A.: Experimenting with Programming Languages. PhD dissertation, University of California, Los Angeles (2009)
32. Warth, A., Piumarta, I.: OMeta: an Object-Oriented Language for Pattern-Matching. In: OOPSLA 2007: Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, ACM Press, New York (2007)
33. Warth, A., Stanojević, M., Millstein, T.: Statically scoped object adaptation with Expanders. In: OOPSLA 2006: Proceedings of the 21st ACM SIGPLAN International Conference on Object-Oriented Programming Systems, Languages, and Applications, pp. 37–56. ACM Press, New York (2006)