A Whole New World

Final Project Write-Up

Kenneth Hwang

John Meeker

March 22nd, 2012

Professor Hardekopf

CS 162

**I: Introduction**

When working on a computer program, the problem often arises of speculating and experimenting with new lines or blocks of code. The usual process entails writing several new additions to your program, testing the outcome of these changes, and often undoing or redoing some code until your additions run without error. While there have been attempts to solve this problem throughout the history of coding (an example being try/catch blocks), none of them come close to offering developers a painless, intuitive way to test new blocks of code. A solution to this problem should enable programmers to control the scope and side effects of new blocks of code, while also remaining unobtrusive to the regular flow of development. We call this solution *Worlds*.

The notion of worlds is to make a new language construct that redefines the role of a program state. All computation takes place within a world (changes to global and local variables, arrays, objects, etc.), and the world also keeps track of all these changes. Worlds are first class values that can be handled as variables, passed as arguments to a function, and can even be garbage-collected like any other object. A new world can be "sprouted" from an existing world at any given time, and the state of a child world is derived from the state of its parent. However, changes and side effects that occur inside the child do not affect the parent. At the same time, the side effects captured in the child world can be propagated to its parent with a commit operation. To understand how the *Worlds* language construct works, it is important to first take a look at the semantics.

**II: Semantics**

The semantics of the *Worlds* extension has the same semantics as notJs, including object literals, field access, field update, method, and method call.

object.tiff

Figure 2.1 Object Semantics

fieldAccess.tiff

Figure 2.2 Field Access Semantics

fieldUpdate.tiff

Figure 2.3 Field Update Semantics

method.tiff

Figure 2.4 Method Semantics

methodCall.tiff Figure 2.5 Method Call Semantics

Additional properties of *Worlds*:

* A world is simply a new environment with variables mapped to different addresses than that of the parent world.
* Before code is run inside a world, the configuration for the world is loaded.
* Committing a world replaces the parent’s environment with the child’s environment.
* Changes in the parent world, whether explicit or implicit, should never make variables appear to change spontaneously in any child world.
* A commit from a child world should never leave the parent world in an inconsistent state. Known as “consistency.”
* Once a variable has been read or modified in a world, subsequent changes to that variable’s parent world are not visible in w. This ensures that variables do not spontaneously change in child worlds.
* During the commit operation, a commit from the child to parent world is only allowed to occur if all the variables that were read in the child world have the same values in the parent world as they did when they were first read by the child world. Known as a “serializability check.”

**III. Syntax**

Our language extension has the following syntax:

b = thisWorld.sprout();

Above is the syntax for sprouting a world from an existing world. “thisWorld” refers to the initial global world.

Assuming “a” and “b” are both worlds, the following is also possible:

b = a.sprout();

The above makes a copy of the current state of world “a”, and allows access to the copy through the variable “b”.

b.commit();

The above syntax is for committing a world. It will commit world “b” back into the parent of world “b”.

inside b {

a:=20

}

The above is an example of code execution inside of a world. The “inside” keyword dictates that we are about to execute a block of code inside a specific world. The variable “b” refers to a world, and the variable “a” is being assigned a value of 20.

**IV. Implementation**

From our original plan in Milestone 2, we thought it would make the most sense to base our *Worlds* project off of the notC syntax. After a tedious amount of time spent trying to get the Sprout and Commit commands to work, we realized that we were taking a slightly incorrect approach to solving the problem. The TA informed us that we would have a much more rewarding experience if we started with notJS as our base syntax. So after weighing out the pros and cons, we decided to abandon our work on the interpreter and syntax which was based off of notC, and start fresh with notJS. This change allowed us to dedicate much more time to understanding the significance of *Worlds*, rather than waste it by modifying a syntax that honestly made more work for us that was unrelated to our end goal.

The plan we had in mind revolved around two key goals: the functionality to sprout new worlds from existing worlds, and being able to commit changes from a world into its parent world. Commit would run the serializability and consistency checks detailed above, and fail if either of them did not pass. These safety precautions solidify the usefulness of the *Worlds* construct by ensuring that programming with them will not be error-prone or dangerous to the normal flow of execution.

The current state of our *Worlds* implementation is working sprout and commit functions. We had originally envisioned the ability to compare worlds to one another (similar to a ‘diff’ function), but a shortage of free time during finals’ week did not permit us to reach this lofty goal. Our ambitions proved to be too great to do proper justice to the *Worlds* language construct. However, we trust that the future youth of CS162 will prove to be much more dedicated and committed to the goal of advancing *Worlds* to take over the world.

**V. Conclusion**

The benefits of the *Worlds* construct are far-reaching and numerous in a wide range of applications. First off, programmers can safely test new blocks of code without the fear and disadvantage of having to spend unnecessary time “doing and undoing” every change to their program. Additionally, *Worlds* provides a way for developers to experiment with and optimize new algorithms in isolated environments that will not interfere with each other during execution. This invaluable feature could foster the quicker development of usually complex problems, such as multi-core programming. While an idea as bold as *Worlds*, might not seem inherently obvious, the benefits of making isolated programming environments are extensive and wide-ranging.