Object Spaces for Safe Image Surgery

Gwenaël Casaccio[1] Damien Pollet[1] Marcus Denker[2] Stéphane Ducasse[1]

[1] RMoD, INRIA Lille Nord Europe
USTL LIFL – CNRS UMR 8022
Lille, France

[2]PLEIAD Laboratory,
DCC University of Chile
Santiago, Chile

ABSTRACT

Long-lived systems rely on reflective self-modification to evolve. Unfortunately, since such a system is at both ends of a causal loop, this means modifications that impact the reflective layer itself can be overly difficult to apply.

This paper introduces ObjectSpaces, a reification of the familiar Smalltalk image as a first-class entity. By confining the system inside an ObjectSpace, we isolate the evolution tools from it, while still giving them reflective access to the confined system. We describe the ObjectSpaces idea, the interface to communicate, inspect, and debug objects contained inside and ObjectSpace, based on a prototype implementation in GNU Smalltalk.

Categories and Subject Descriptors

D.3.2 [**Programming languages**]: Smalltalk; D.2.3 [**Software Engineering**]: Coding Tools and Techniques

General Terms

Languages

Keywords

Reflective Systems, Meta Programming, Dynamic Languages, Smalltalk

1. INTRODUCTION

Smalltalk images exemplify everlasting, evolving software systems. An image is a persistent memory snapshot of a Smalltalk system, complete with all objects that take part in the system's execution. In fact, today's Squeak images inherit objects that date back to the original Smalltalk-80 system.

Images are interesting because they support an unusual approach of program evolution. Since the whole system resides in the image, including tools like the compiler and debugger, a Smalltalk program is a self-modifying persistent environment, more closely resembling the operating system installed on a physical workstation than a single executable file or process. Also, the image is persistent, so programs do not need to boot and initialize themselves from scratch each time they are invoked; instead, programs can just live here and answer requests, keeping and migrating their state across evolutions of the code and data structures.

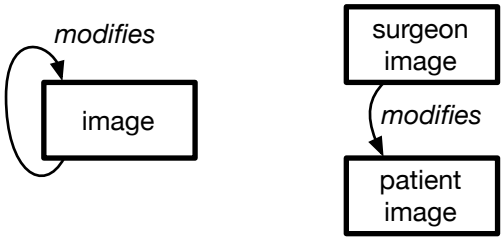

Figure 1: From self-modifying images to separate surgeon/patient images. The development tools in the surgeon image can change core classes and methods in the patient image without being impacted themselves.

However, the self-modifying image is both a blessing and a curse. While it enables to save the image in the middle of a debugging session, and re-open it on another physical machine to find the debugger and halted program in the same bit-identical state, it also makes it very easy to inadvertently break code on which critical systems like the compiler or debugger depend. Moreover, it is not à priori obvious which code is safe to change; an example of such sensitive code is the pervasively used iterator method Array>>#do: [5].

To make a real-life analogy, while it is easy to self-treat small wounds, it's more difficult for a neuro-surgeon to operate on himself. What we want is to clearly separate code that acts as the surgeon from code that acts as the patient in separate images[1], as shown in Figure 1.

In this paper, we introduce ObjectSpaces, a reification of the image as an abstraction for delimiting and isolating a group of objects —the patient— while giving full visibility of its contents to another system —the surgeon. Additionally to any domain objects, an ObjectSpace also contains a copy of the necessary system methods, classes, and globals, so it is in effect a self-contained Smalltalk system. To the surgeon, an ObjectSpace appears as a meta-object with facilities for inspecting and changing objects of the patient system. However, that meta-object ensures that no reference from the surgeon enters the patient, thus preventing the patient from causing any side-effect on the surgeon. Development tools running in the surgeon can thus control the patient, modify its objects, change sensitive methods, or bootstrap core class structures without risk of

[1]. . . and, to perform *self*-surgery, resort to clones and memory backups like in John Varley's novel *The Ophiuchi Hotline* [13].

Preprint, published as:
IWST'09 August 31, 2009, Brest, France.
2009 ACM 978-1-60558-899-5 .

self-interference.

The remainder of the paper is structured as follows: Section 2 presents the problem. We describe why Smalltalk images are very sensitive to changes. Section 3 presents our approach. In Section 4, we illustrate how ObjectSpaces work from the user standpoint, then we describe the prototype implementation in Section 5. Finally, we present related work in Section 6, before discussing some open questions and concluding in Section 7.

2. CONTROLLING THE IMPACT OF SYSTEM CHANGES

Some methods or classes in a Smalltalk image are very sensitive to changes, either because they are used pervasively throughout the system, or take part in important subsystems like the user interface, the compiler, the debugger, the metaclass hierarchy, or classes that are well-known to the virtual machine. This is because Smalltalk has a single scope where not only everything is visible, but also reflectively accessible. Any breakage in this kind of code usually leads to spectacular failure. A simple example would be adding a breakpoint in the iterator method Array>>do:. In Pharo Smalltalk, adding this breakpoint impacts about 90000 Array instances and the image freezes [5]. It is thus very difficult to debug or change this code in a realistic setting, without risking to impact the whole image.

But the fact is, some evolutions do require changes to this sensitive code. If temporary breakage is necessary, then the maintainers must find a way to apply the changes with reduced tools and extra care. Alternatively, some images are destined to run under restricted conditions that make it impractical to include a complete set of development tools, or simply to access them: for instance, images running on remote servers do not have an active graphical interface[2]. This makes development, testing, and maintenance impractical or even impossible.

Current solutions or workarounds are to work on a renamed copy of the classes, or to remotely control a separate image via remote objects. However, in the first case we only delay the problem, because the complete impact of changes cannot be assessed until the copied and modified classes are merged back into the system. In the latter case, if a change causes the remote image to crash, then it will be impossible to assess the problem.

ObjectSpaces to the rescue.

We need a way to control the impact of changes, by making sure the surgeon and patient are different persons, i.e., by clearly separating and isolating the development environment from the domain code, while still giving the surgeon complete reflective access to the patient system.

3. OBJECTSPACES: IMAGES IN THE IMAGE

An ObjectSpace is a reification of a Smalltalk image. An ObjectSpace encloses a self-contained, isolated subsystem of a larger Smalltalk environment: objects can run and communicate inside the ObjectSpace as if they were running in a normal environment, but they cannot reference or interact with objects outside the space. However, the ObjectSpace and its contents can be interacted with from the enclosing environment: the owner of an ObjectSpace can refer to objects inside it, as well as inject new objects or messages into it.

[2]Remote display solutions like VNC do exist, but suffer from usability and portability problems.

ObjectSpaces can be loaded from and written to the image format on disk. We can also create an ObjectSpace ex-nihilo, then recreate a working Smalltalk environment inside it by copying classes, instances, or a full namespace from the surgeon Smalltalk environment. Processes run inside an ObjectSpace just like in a full-fledged image[3].

Once an ObjectSpace is created, tools in the surgeon environment can control it and interact with it to:

- inspect existing objects, via mirrors [1],
- inject new objects into the patient ObjectSpace,
- inject messages to be received by patient objects, and possibly debug their execution,
- handle exceptions raised but unhandled by patient code,
- save the ObjectSpace to an image file, to re-load it later either as an ObjectSpace or as a standard image,

Implementing Self-Surgery.

Changing core classes of the system is akin to a bootstrap process, meaning that the changes could require the image to go through a non-working state, and thus that the usual Smalltalk tools cannot apply directly —the VM can only load working images. The only way to do that is to generate an image from scratch, with the changes applied, but this implies that the control is external to the bootstrapped image. In practice, not all implementations of Smalltalk are able to bootstrap an image from bare sources. Moreover, generating an image from scratch means dealing with the details of the memory representation of objects, and ensuring that the generated image is consistent.

ObjectSpaces are a means to access the contents of an image without running it, at the relatively high abstraction level of reflective access to objects, but also by relying on code inside the ObjectSpace when possible. To implement self-surgery using ObjectSpaces, the original image clones itself to an ObjectSpace —or saves a copy of itself to disk and loads that image file into an ObjectSpace— as shown in Figure 2. It can then assume the role of the surgeon and apply any necessary changes to its inactive patient copy. During the operation, application domain services can be paused, so that their state is migrated according to the changes in the patient. When the changes are applied, the patient can be awaken by loading it into a new VM, taking the place of the surgeon —i.e., taking ownership of any open files, sockets, etc. needed by application domain code. To application code, the operation appears as transparent as saving then re-loading the image, and it resumes work in the same logical state as before the operation. Of course, in a production setting, the surgeon should assess that migrating services to the patient is actually safe, e.g., by running tests. To the surgeon, the only state created after the application services pause should only pertain to applying the changes and migrating data; the surgeon image can terminate itself safely, since it's now redundant with the patient.

4. OBJECTSPACES AT WORK

In this section, we illustrate how to sandbox arbitrary code execution using our ObjectSpace prototype. Consider a simple web site that allows its users to discover the Smalltalk language by browsing the system and evaluating arbitrary expressions. Even ignoring security considerations like access to the server filesystem, user-provided

[3]Ideally, the surgeon controls the patient's process scheduler, but this is not implemented yet in our prototype.

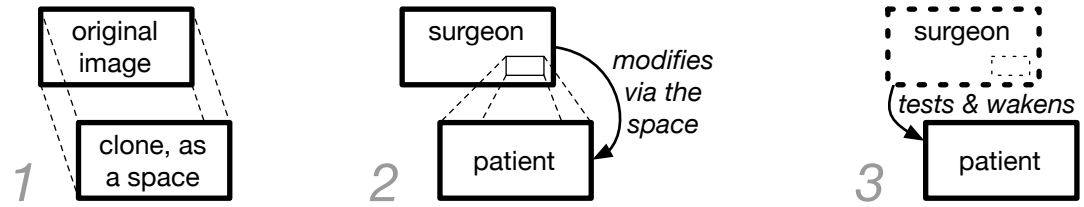

Figure 2: Self-surgery via a clone: 1) the image clones itself into an ObjectSpace; 2) original and clone assume the roles of surgeon and patient; the surgeon modifies core objects in the patient, through the ObjectSpace meta-object; 3) once the patient reaches the desired state, the surgeon activates it, then terminates; except for the changes, the patient awakes in the same state as before the cloning.

expressions should be expected to break the environment they are evaluated in, so users should be sandboxed from each other and from the web framework. To do that, we create an ObjectSpace for each user to evaluate his expressions. The ObjectSpace acts as the user's own disposable environment, but in contrast to spawning a complete image on a separate VM, the web framework can easily inspect the expression effects within its environment.

The first step is to create a new ObjectSpace for each of our users, Alice and Bob:

```
spaces at: #alice put: (ObjectSpace with: Smalltalk).
spaces at: #bob put: (ObjectSpace with: Smalltalk).
```

Here, ObjectSpace with: Smalltalk recreates a self-contained environment inside a new ObjectSpace, based on the Smalltalk namespace.

When the user enters an expression on the web page, the framework injects it inside the ObjectSpace for evaluation:

```
(space at: #bob) eval: 'Array compile: "do: [ self halt ]" '.
```

The eval: message compiles the string within the ObjectSpace's environment and executes the resulting code. Since the ObjectSpace environment runs in complete isolation, this modifies Bob's own copy of the class Array, and only the instances in Bob's ObjectSpace will be affected by this change. Alice is not impacted because the Array class inside her ObjectSpace is an independant copy that is not even known inside Bob's ObjectSpace; in fact, at this point, only Bob's code is broken.

Since Bob's ObjectSpace is sleeping —the code was changed, but not run yet— we can tell it to evaluate a block that exercises the new behavior of Array>>do::

```
| exc |
(space at: #bob)
  do: [#(1 2 3) do: [:i | Transcript show: i]]
  onException: [:e | exc := e ]
```

The message do:onException: *injects* the given block into the ObjectSpace then evaluates it, returning a mirror to its value or catching any exception it raises. Injecting a block is just another way of interacting with code in an ObjectSpace. However, whereas a string is a inert data and is simply copied, a block contains references to objects which we need to pull into the ObjectSpace as well. To inject a block, we thus copy the block itself and its literals to the ObjectSpace, and also translate any global reference the block contains to refer to the patient environment —so here, Transcript will refer to the one inside the ObjectSpace.

Of course, when #(1 2 3) receives #do:, the breakpoint we set earlier raises an exception, which we store in the variable exc. A specialized debugger then takes control of the ObjectSpace execution like so:

```
(space at: #bob) debug: exc; restart; step; step; ...
```

Finally we can reinstate a working version of the method:

```
(space at: #bob)
  compileMethod: 'do: aBlock
    1 to: self size do: aBlock' for: #Array
```

5. IMPLEMENTATION

Our prototype implementation was realized in GNU Smalltalk. While we expect a complete implementation to require VM modifications for mirrors, controlling primitives, and performance, the approach is generic to Smalltalk and does not depend on features of a particular dialect.

5.1 Creating an ObjectSpace

To create an ObjectSpace, we clone classes and namespaces (i.e., the Smalltalk system dictionary) to initialize a new environment.The cloning operation of a class also creates a new instance of the metaclass and makes a deep copy of the class structure: instance variables, method dictionary, compiled methods, method literals, and shared pool. We reset values in the shared pool to nil, since we want all classes to be in a clean state. This process is similar to the one of saving a standard image, so eventually it should support ad-hoc behavior on class shutdown and wake-up.

Since just copying a class will introduce pointers from the patient copy to its origin surgeon environment, we have to rebind the variables to point to their counterparts inside the ObjectSpace. We also rebind the class pointers of copied objects —like method literals— to point to their counterpart class inside the ObjectSpace. Copying global variables is not required unless they are referenced in a copied method; such references are rebound to their ObjectSpace counterparts. Once this is done the process of bootstrapping an ObjectSpace is finished: no references to surgeon objects remain in the patient.

5.2 Injecting Messages Into an Objectspace

Once the bootstrap is achieved, we obtain an ObjectSpace, but at this time it is only a set of inactive classes. To activate the patient ObjectSpace and communicate with its objects, the surgeon can inject a message into it. Note that we use the term *inject* instead of *send*, because from the patient perspective, injected messages appear from nowhere: they have no sender context, like *Do-it* requests.

```
anObjectSpace send: #start to: #StartupClass
```

The ObjectSpace looks up #StartupClass in its environment, then simply sends it the given message. To inject messages with

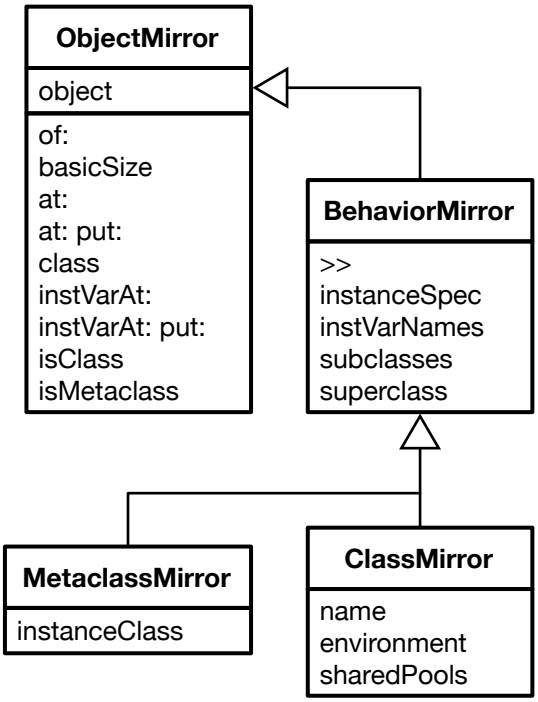

Figure 3: Mirror class hierarchy.

arguments, the arguments themselves have to be copied to the ObjectSpace first, additionally to the message selector. The surgeon obtains a mirror [1] to the result of the message —or to any exception it throws.

5.3 The Case of Reflection

While ObjectSpaces should provide reflective access to their contents, the classical reflective methods cannot be used by the surgeon to reflect on patient objects, because they would defeat the exact purpose of the ObjectSpace:

- First, even reflective methods must maintain the invariant that the ObjectSpaces do not share references; this is not the case with the existing reflective methods, and is not realistic as it would require special-casing when the method is used across an ObjectSpace border.
- Second, and most importantly, the surgeon code cannot assume that patient system has the same meta-structure, and neither can it rely on patient code for reflective access —in the general case at least. Indeed, the patient's reflective methods could themselves be the subject under surgery.

Therefore, to support reflection, we implemented a mirror-based architecture [1].

5.4 Mirrors

Using mirrors, we can refer to, inspect, and modify patient objects, independently of their reflection implementation. In practice, the ObjectSpace object itself behaves as a mirror to the isolated patient environment, and instances of class ObjectMirror provide basic reflective access to instances inside an ObjectSpace, using VM primitives.

The mirror classes provide low-level reflective methods corresponding to those in the original Smalltalk classes; for instance, ObjectMirror provides access to instance variables by index, to the class pointer, and to the size of patient instances. Compared to normal reflective access, mirrors do not rely on code in the patient environment; they act as a proxy to the patient object they represent. For instance, the method Object>>#at: is normally implemented by a primitive call <primtive: VMpr_Object_basicAt>, but it could be redefined inside an ObjectSpace. The mirror ensures that the implementation of #at: available to the surgeon is independent of the patient state, and enforces that references do not cross the ObjectSpace boundary without going through their own mirror.

6. RELATED WORK

The most related family of work is virtualization approaches like Xen [2]. Virtualization makes it possible to run several operating systems at once on a single physical machine. As these approaches target full operating systems, they rely on support from the hardware platform, and in some cases from the guest OS; they also concentrate on performance and production features, and consider the guest system mostly as a black box. In contrast, ObjectSpaces provide full control and reflective access to their contents.

Changeboxes [4] encapsulate and scope changes, allowing several versions of a system to coexist in a single runtime environment, effectively adapting version control from static source code to running systems. Changeboxes scope code changes, while ObjectSpaces scope generic object references; also, Changeboxes do not directly address the problem of applying changes to code that is critical to the runtime system itself.

Scoping side-effects has been the focus of two recents works. Worlds [14] provide a way to control and scope side-effects in Javascript. Similar to ObjectSpaces, side-effects are limited to a first-class environment. Tanter proposed a more flexible scheme: contextual values [12] are scoped by a very general context function.

One problem meta-circular architectures is that meta-objects rely on the same code they reflect upon; therefore there is a risk of infinite meta-recursion when the meta-level instruments code that it relies upon. In [5], Denker et al solve this problem by tracking the degree of metaness of the execution context. Meta-objects can only reflect on objects of a lower metaness, thus simulating the semantics of an infinite tower of distinct meta-interpreters. The existing work on Meta-context is only concerned with scoping behavioral changes. More work is needed to extend this work to structure. We plan to explore how ObjectSpaces can be used to provide a way to control structural reflective change.

One possibility for implementing ObjectSpaces is to differentiate messages depending on whether the sender and the receiver are in the same space or not. Several works use a similarly extended message lookup. Us [11] is a system based on Self that supports subject-oriented programming [6]. Message lookup depends not only on the receiver of a message, but also on a second object, called the *perspective*. The perspective allows for layer activation. ContextL [3, 7] is a language to support Context-Oriented Programming (COP). The language provides a notion of *layers*, which package context-dependent behavioural variations. In practice, the variations consist of method definitions, mixins and *before* and *after* specifications. Layers are dynamically enabled or disabled based on the current execution context. ObjectSpaces provide form a context. The relationship between context-oriented programming, subjectivity and ObjectSpaces is an interesting topic of future research.

Gemstone [10] provides the concept of class versions. Classes are automatically versioned, but existing instances keep the class (shape and behavior) of the original definition. Instances can be migrated at any time. Gemstone provides (database) transaction semantics, thus state can be rolled back should the migration fail. Gemstone's

class versions extend the usual Smalltalk class evolution mechanism for robustness, large datasets, and domain-specific migration policies. In contrast, ObjectSpaces target general reflective access and bootstrap-like evolutions of code that is critical to the environment.

In Java, new class definitions can be loaded using a class loader [9]. Class loaders define namespaces, a class type is defined by the name of the class and its class loader. Thus the type system will prohibit references between namespaces defined by two different loaders. Class loaders can be used to load new versions of code and allow for these versions to coexist at runtime, but they do not provide a first-class model of change. Java also provides JPDA, a remote debugging architecture that specifies a native interface on the debuggee VM, and a matching API for the debugger front-end, running in a separate VM. However, JDPA only supports introspection features like inspection and monitoring, and very limited intercession [8].

7. DISCUSSION AND CONCLUSION

We have presented ObjectSpaces, a reification of the Smalltalk image as a first-class entity. The Smalltalk image is a powerful tool, as it provides persistance without any overhead and the possibility to transfer or restore running systems. However, since an image embeds both the domain code and the developer tools, it can become an obstacle to safe system evolution. An ObjectSpace isolates the tools from the effects of the changes they perform, while still providing them full control and reflective access over the domain system. It thus enables safe low-level changes in existing systems, or means to bootstrap new systems in a practical way. While the idea is still in its early stages, we think ObjectSpaces solve the main problems with image-based development, while embracing the perspective of evolving living systems.

However, ObjectSpaces are still an early idea that leaves many possibilities open. Our current implementation has some limitations. We cannot change the class of instances of Integer, Symbol, BlockContext, MethodContext, or Process because those objects have a special format that encodes the class pointer, or are otherwise known by the VM as they take part in the reflective causal connection with the language.

In our current implementation, we have no control over primitive methods. This means that a program can call a primitive like nextObject which returns all the objects in the image, and thus escape the ObjectSpace boundary. As a solution, we plan to intercept primitive calls from the patient, so that the surgeon can prevent or replace them by other primitive calls or even Smalltalk methods. It would be then possible to create ObjectSpaces where unwanted primitives like file access or inspecting the whole object memory (nextObject or become:) throw an exception to the surgeon.

Finally, there is the question of allowing inter-objectspace messages. If we allow them, we have to make sure that references do not flow between communicating ObjectSpaces. This requires to intercept inter-objectspace messages and to inject all arguments into the space of the receiver, or to wrap them in mirrors. A simpler alternative is to simply forbid inter-space messages and rely on actor-like remote messaging —possibly optimized to take advantage that both ObjectSpaces run on top of the same VM.

Acknowledgments. The authors gratefully acknowledge David Chisnall for his input and discussion on the ideas presented here. Marcus Denker acknowledges the financial support of the Swiss National Science Foundation for the project "Biologically inspired Languages for Eternal Systems" (SNF Project No. PBBEP2-125605, Apr. 2009 - Mar. 2010).

8. REFERENCES

[1] Gilad Bracha and David Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04), ACM SIGPLAN Notices*, pages 331–344, New York, NY, USA, 2004. ACM Press.

[2] David Chisnall. *The Definitive Guide to the Xen Hypervisor*. Open Source Software Development Series. Prentice Hall, 2007.

[3] Pascal Costanza and Robert Hirschfeld. Language constructs for context-oriented programming: An overview of ContextL. In *Proceedings of the Dynamic Languages Symposium (DLS) '05, co-organized with OOPSLA'05*, pages 1–10, New York, NY, USA, October 2005. ACM.

[4] Marcus Denker, Tudor Gîrba, Adrian Lienhard, Oscar Nierstrasz, Lukas Renggli, and Pascal Zumkehr. Encapsulating and exploiting change with Changeboxes. In *Proceedings of the 2007 International Conference on Dynamic Languages (ICDL 2007)*, pages 25–49. ACM Digital Library, 2007.

[5] Marcus Denker, Mathieu Suen, and Stéphane Ducasse. The meta in meta-object architectures. In *Proceedings of TOOLS EUROPE 2008*, volume 11 of *LNBIP*, pages 218–237. Springer-Verlag, 2008.

[6] William Harrison and Harold Ossher. Subject-oriented programming (a critique of pure objects). In *Proceedings OOPSLA '93, ACM SIGPLAN Notices*, volume 28, pages 411–428, October 1993.

[7] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3), March 2008.

[8] Java platform debugger architecture. `http://java.sun.com/javase/technologies/core/toolsapis/jpda/`.

[9] Sheng Liang and Gilad Bracha. Dynamic class loading in the Java virtual machine. In *Proceedings of OOPSLA '98, ACM SIGPLAN Notices*, pages 36–44, 1998.

[10] Allen Otis, Paul Butterworth, and Jacob Stein. The GemStone object database management systems. *Communications of the ACM*, 34(10):64–77, October 1991.

[11] Randall B. Smith and Dave Ungar. A simple and unifying approach to subjective objects. *TAPOS special issue on Subjectivity in Object-Oriented Systems*, 2(3):161–178, 1996.

[12] Éric Tanter. Contextual values. In *Proceedings of the 4th ACM Dynamic Languages Symposium (DLS 2008)*, Paphos, Cyprus, jul 2008. ACM Press. To appear.

[13] John Varley. *The Ophiuchi Hotline*. Dial Press, 1977.

[14] Alessandro Warth and Alan Kay. Worlds: Controlling the scope of side effects. Technical Report RN-2008-001, Viewpoints Research, 2008.