Read-Only Execution for Dynamic Languages

Jean-Baptiste Arnaud[1], Marcus Denker[1], Stéphane Ducasse[1], Damien Pollet[1], Alexandre Bergel[2], and Mathieu Suen

[1] INRIA Lille Nord Europe - CNRS UMR 8022 - University of Lille (USTL)
[2] PLEIAD Lab, Department of Computer Science (DCC), University of Chile, Santiago, Chile

Abstract. Supporting read-only and side effect free execution has been the focus of a large body of work in the area of statically typed programming languages. So far, dynamically typed languages have been poorly addressed despite their increasing presence in the web and multi-language applications. Read-onlyness in dynamically typed languages is difficult to achieve because of the absence of a type checking phase and the support of an open-world assumption in which code can be constantly added and modified.

To address this issue, we propose Dynamic Read-Only references (DRO) that provide a view on an object where this object and its object graph are protected from modification. The read-only view dynamically propagates to aggregated objects, without changing the object graph itself; it acts as a read-only view of complex data structures, without making them read-only globally. Objects can be referenced at the same time by a read-only reference and a standard read-write reference by different clients.

We implement dynamic read-only references by using smart object proxies that lazily propagate the read-only view, following the object graph and driven by control flow. We have implemented dynamic read-only references in Pharo,[3] an open-source Smalltalk, and applied them to realize side-effect free assertions and read-only data-structures.

1 Introduction

During the execution of a program, an object is aliased each time it is passed as a message argument or referenced by a variable. While imperative (and in particular object-oriented) programs rely on aliasing and side effects to produce computation, unwanted side effects occurring through different aliases to a shared object are a source of many bugs.

For instance, contracts, as provided by Eiffel [1], use pre- and post-conditions placed before and after a method body. Contracts should not influence the program behavior; in fact they can be ignored once the program is stable, for performance reasons. However, nothing besides convention actually prevents pre- or postconditions from having side effects, which may lead to insidious bugs. Ensuring functional purity of contracts would be too strong: a pre-condition could perfectly rely on internal side-effects, as long as it does not change the state of the rest of the program. It is also difficult to ensure side-effect free assertions in the presence of both late binding and imperative code [2].

[3] `http://www.pharo-project.org`

Note that in statically typed languages, visibility qualifiers (such as private and protected for methods) have little effect to prevent malicious side effects and other non-local modification. Indeed, an alias may leak critical objects to untrusted parties, as illustrated in the following code snippet of a browser-side script [3]:

```
class DocumentProxy {
  // public API goes here
  private var docServerSocket = new ServerSocket(...);
  public hole() { return docServerSocket }}
```

In this example, the *public* method hole() returns the object referenced by the *private* variable docServerSocket. While we cannot affirm whether this code is poorly designed (it essentially depends on how hole() is used), the encapsulation DocumentProxy defined for itself is definitely broken since docServerSocket can be modified at will once obtained from hole(). A number of mechanisms have flourished to address this situation. They may be classified as either (i) enforcing that a reference cannot escape based on *aliasing control*, or (ii) allowing only non-modifiable methods to be invoked using a *limited interface*.

Alias Control. There have been a number of proposals to detect aliasing problems and control them in statically typed languages [4–7]. Ownership types [6, 8] allow objects to participate in more than one container while Balloon types [5], Islands [4] don't. In the latter case they forbid objects inside the aggregate to refer to objects outside it, making it impossible to share an object between two containers. Ownership types [6] provide full alias encapsulation. They statically restrict programs so that no aliases may *escape* an aggregate object's encapsulation boundary.

Dynamic object ownership [3, 9] is one of the rare answers to this problem in the context of dynamically typed languages. Dynamic ownership protects aliases and enforces encapsulation by maintaining a dynamic notion of object ownership and restricting messages based on their ownership. It classifies aggregated objects into purely internal ones (the representation) and the ones which may be accessible in read-only mode from the outside (the arguments).

Limited Interface. In this family of solutions, references can leak and objects can be freely accessed, but their interfaces are restricted. Several approaches have been proposed, such as capabilities which control the interface of an object [10–13]. In these approaches, an object offers a limited interface when it wants to limit access. Birka *et al* add a readonly type qualifier, which makes all state transitively reachable from a read-only reference immutable [14]. Encapsulation policies also propose different per-reference encapsulation interfaces [15, 16].

Our solution: Dynamic read-only references. Objects do not exist in isolation but are connected to other objects, thus forming a graph. In this case, there is a need to control when side effects can occur and how the immutability propagates to aggregated objects. In particular, the same object may need to offer full access from a reference, but be read-only from another reference. This situation occurs, for example, when one implements side-effect free assertions or Design by Contract precondition checks. The objects involved in the assertions should not be modifiable from the assertion itself, but they can be referenced and changed from other parts of the program. Another example

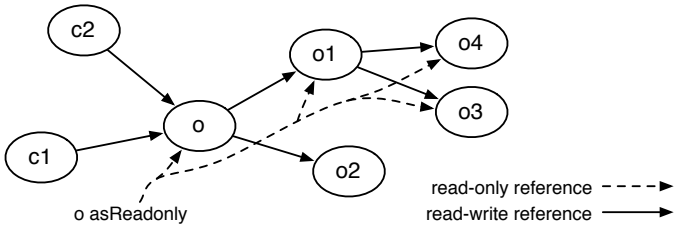

Fig. 1. A control flow lazy propagation of readonly.

is when an aggregate object should be able to mutate its elements and at the same time give read-only references of such elements to its clients [3, 9].

Instead of relying on static types (which we simply cannot in a dynamic language) or restricted interfaces, we opt for an approach based on dynamic immutability propagation over an overlay of references starting from an object. Our solution to support dynamic read-only references (DRO) in dynamically typed languages is the following: A read-only reference is a transparent proxy where methods attempting side effects raise an exception instead. By transparent, we mean that, from a programmer perspective, it is not possible to tell the object apart from its proxy.

Finally, our approach is dynamic in the sense that object immutability is lazily propagated *following the execution flow and without modifying* the object graph itself. Figure 1 shows that two objects `c1` and `c2` refer to `o` which in turn refers to `o1` and `o2` and so on. A readonly reference to `o` is a dynamic layer following the control flow of the execution (here object `o2` was not involved in the execution while `o3` and `o4` were). From the perspective of the readonly reference on `o` it is not possible to modify the state of the reached objects (`o`, `o1`, `o3`, `o4`). However via the objects `c1` and `c2` the object `o` can be modified.

Contributions. In this paper:

- We identify the problems with immutability and the need for dynamic read-only views (Section 2);
- We propose a flexible model of dynamic read-only execution adapted to dynamically typed languages, and illustrate it (Sections 3 and 4);
- Section 5 formally describe key properties of Dynamic Read-Only references using the operational semantics of SmalltalkR, an extension of SmalltalkLite [17];
- We describe the current implementation and its performance, and finally we discuss key design points and their consequences (Sections 6–8).

2 The Challenges of Dynamic Read-Only Execution

Read-only execution is appealing for a number of reasons, ranging from easier synchronization to lowering the number of potential bugs due to side effects, aliasing and encapsulation violation [2, 3, 5]. Mechanisms prohibiting variable mutation are now

common in programming languages (*e.g.,* Java has the final keyword, C++ and C# have const). This prevents the reference and not the referee to be modified.

In dynamically typed languages, few attempts have been made to provide immutable objects. VisualWorks Smalltalk[4] and Ruby support immutable objects using a per object flag that tells whether the fields of the object may be modified. Since immutability is a property embedded in the object itself, once created, immutable objects cannot be modified via any alias. In this section, we discuss the challenges of realizing read-only execution in dynamically typed languages.

Different Views for Multiple Usages. Read-only execution is often needed in cases where objects taking part in the read-only execution are at the same time (*e.g.,* from another thread of execution) referenced normally with the ability to do side-effects. A clear illustration of this need is assertion execution: within the scope of an assert method invocation, objects should not be modified, else this could lead to subtle bugs depending on the evaluation of assertions.

What we want to stress here is not only the need for immutability but also the fact that such a property can be dynamic and that the same object can be simultaneously referenced read-only from one object, and write-enabled from another object.

The Case of Object Creation. One interesting question appears when thinking about objects that are created locally and used in a method but never stored in an object. As far as the rest of the system is concerned, these objects are disconnected: changing them does not change the overall state of the system. It is natural to think that newly created objects are not read-only even when referenced from a read-only execution context. Storing a newly created object within a read-only view should raise an error.

Flexibility to Support Experiments. Having read-only object references is one solution of a larger problem space. Several possible reference semantics and variants have been proposed in the literature such as Lent, Shared, Unique or Read-only. In addition, read-only is just one element in the larger spectrum of capability-based security model [13]. Therefore the current model should be flexible to be able to explore multiple different semantics. Just providing one fixed model for how the read-only property works is not enough (see Section 7). The model and its implementation should be flexible enough to be able to grow beyond pure read-only behavior.

Challenges for Read-Only Execution in Dynamically Typed languages. The key challenge is then:

> How do we provide flexible read-only execution on a per reference-basis in the context of complex object graph with shared state without static type analysis support?

Some work such as encapsulation policies [15, 16] offer the possibility to restrict an object's interface on a per reference basis, but they do not address the problem of propagating the read-only property to aggregate references. Encapsulation policies are static from this point of view. ConstrainedJava offers dynamic ownership checking [3] but the focus is different in the sense that ConstrainedJava makes sure that object references do not leak while we are concerned about offering a read-only view.

[4] `http://www.cincomsmalltalk.com`

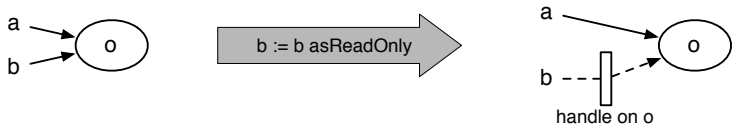

Fig. 2. A reference to an object can be turned into a read-only reference.

3 Dynamic Read-Only References

Our approach to introduce immutability in a dynamically typed language is based on dynamic read-only references. Dynamic read-only references can offer different behavior for immutability to *different* clients. Such references are based on the introduction of a special object, called handle, by which clients interact with the target object. The handle therefore forms a reification [18] of the concept of an object reference: a reference is now modeled by an object. Similar reifications have been realized in other contexts, one example is *Object Flow Analysis*, which models aliases as objects [19]. A handle is a *per reference* transparent proxy, *i.e.,* identity is the same as the object they represent.

3.1 Handle: a Transparent per Reference Proxy

Conceptually, a handle is an object that represents its target; it has the same identity as its target, but redefines its behavior to be read-only.

At the implementation level, handles are special objects. When a message is sent to a handle, the message is actually applied to the target object, but with the handle's behavior —*i.e.,* the receiver is the target object, but the method is *resolved in the handle*. In essence, a handle never executes messages on itself, rather it replaces the behavior of messages that are received by its target.

Handler creation example. Figure 2 illustrates the creation of a read-only reference. First, we get an object and we assign it into the variables a and b. Second, via the variable b, the object is asked to become read-only. This has as effect to create a handle.

Identity of Handle and Object. Contrary to traditional proxies [20] and similarly to Encapsulators [21], a handle and its target have the same identity — the handle is transparent. Not having transparency could lead to subtle bugs because most of the code is not (and does not have to be) aware of the existence of handles [22]: for example, without transparency, adding a target and its handle to a set would break the illusion that the handle is the same object as the target since both would be added to the set.

3.2 Enabling Read-only Behavior

As explained above, a handle can have a different behavior than its target for the same set of messages. Specifically, we redefine all the methods to offer the expected read-only behavior. To install a handle on a target, we rewrite the target's methods and install them into a *Shadow Class* that the handle references:

1. Store accesses in globals and in instance variables signal an exception;

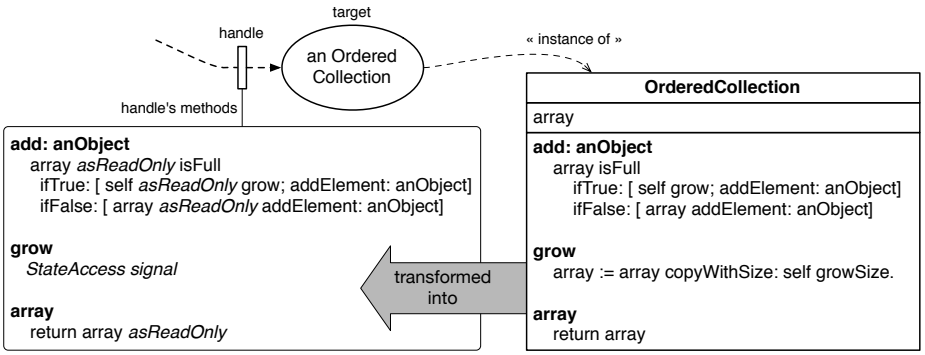

Fig. 3. The handle holds the read-only version of the target's code, with instance variable accesses and affectations rewritten.

2. Read accesses to globals, instance variables and to the self pseudo-variable are dynamically wrapped in read-only references.

Figure 3 illustrates this behavioral transformation. The class OrderedCollection defines an instance variable array, and three methods: the method grow which modifies the state of the collection by modifying the contents of the instance variable array, the method add: which conditionally invokes other methods and the method array which returns a reference to this object. The Shadow class of the handle for this object contains three methods. The method grow which raises an exception. The method array which returns a read-only reference to the internal array. The method add: is transformed according to the transformation we described above: self/this, instance variables and globals are asked to be read-only and modification raise an error.

Objects referenced by temporary variables are not transformed into read-only references because they do not change the state of the object. Objects referenced by arguments may be read-only references, but only because of an earlier transformation.

Note that this transformation is recursive and dynamic. It happens at run-time and is driven by the control-flow. This recursive propagation is explained in the next section.

3.3 Step by Step Propagation

The read-only property dynamically propagates to an object's state when it is accessed. Figure 4 illustrates how immutability is propagated to the state of a list of students. When the graduation: message is sent to a read-only Promotion instance, first the message do: is sent to the read-only students instance variable. This invokes the read-only version of the Array»do: method, and in turn the read-only version of Student»hasGoodGrades. Since hasGoodGrades does not modify its receiver (here the Student instance), the execution continues this way until it reaches a student with grades good enough to pass. At that point, we are still in the read-only version of the Promotion»graduation method, so we evaluate the read-only version of graduate, which throws an exception, since in its original version it modifies the student's diploma instance variable.

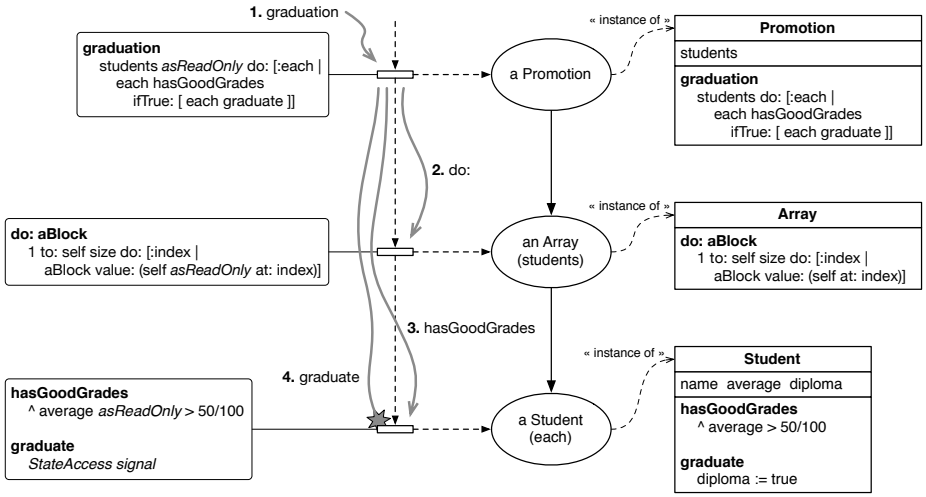

Fig. 4. When we send the graduation message via a read-only reference to the Promotion instance, evaluation proceeds via read-only handles, until the graduate method attempts a side-effect.

This example shows that immutability is propagated to the references of instance variables and global variables recursively and on demand, based on the execution flow of the application. Note that execution fails only when it reaches an assignment to an instance variable or a global variable.

Global Variables and Classes. Access to global variables is controlled the same way as access to instance variables: write access is forbidden (it raises an exception), any read is wrapped in a creation of a read-only handle. As classes are objects in Smalltalk, accesses to classes leads to the creation of a read-only handle for that class. Any change of the structure of the class, *e.g.,* adding or removing methods or changing the inheritance hierarchy is therefore forbidden.

Newly Created Objects. Even though referencing a class will result in a read-only handle, using the class to create a new Object will not result on this object being wrapped in a read-only handle. A new object has only one reference, therefore changing this object will not lead to a side effect: all other objects of the program remain unchanged. We can therefore just have new objects be created normally, allowing modification. Any try to store the newly created object will result in an error, making sure that the object never has any influence on the rest of the system.

Temporary Variables. Temporary variables only live for the extend of the execution of one method. They are not stored in an object if not done explicitly. Therefore, we can safely keep temporary variables with read-write semantics: reading does not wrap the reference in a handle, writing is allowed.

Arguments. We do not wrap arguments or return values. If a value is used as an argument, it has to come from somewhere: it is either read from an instance variable, and therefore already read-only. Or it was created locally in a method and is not connected to any

other object in the system. We can therefore hand over arguments without any special wrapping: either the reference passed is already read-only or it is read-write, it is handed over unchanged.

Block Closures. For closures, there are two cases to distinguish: executing a block read-only and passing a block to a method of a read-only data-structure as an argument. For read-only execution, the objects representing closures need to support a read-only execution mode. For this, all instance and global variable reads are wrapped in a handler. In addition, we need to wrap all variable reads defined outside of the closure and the arguments.

4 Examples

We now present how DRO is applied to solve two critical problems: side effect-free assertions and read-only collections.

4.1 Example Revisited: Side Effect Free Assertions

An assertion is a boolean expression that should hold at some point in a program. Assertions are usually used to define pre- and post-conditions: in a service contract, if the precondition is fulfilled, then the postcondition is guaranteed. Assertions are supported by a number of languages, including Eiffel, Java, Smalltalk, and C#. It is reasonable to expect a program to behave to the same bit when assertions are removed to gain performance. However, none of the languages supporting assertion can enforce behavior preservation when removing assertions, because the assertion code could perform any method call or side effect.

Writing an assertion for *e.g.,* a ServerSocket object requires that the object is not changed during the execution of the assertion (which can be repeated multiple times), while it can be modified outside the contract. To illustrate our point, consider the following method definition extracted from the trait implementation of Pharo:

```
Behavior>>flattenDown: aTrait
  | selectors |
  self assert: [self hasTraitComposition and:
     [self traitComposition allTraits includes: aTrait]].
  selectors := (self traitComposition transformationOfTrait: aTrait) selectors.
  self basicLocalSelectors: self basicLocalSelectors , selectors.
  self removeFromComposition: aTrait.
```

The assertion is highlighted in **bold**. It contains five message sends (hasTraitComposition, and:, traitComposition, allTraits, and includes:). If one of these message leads to a side effect, then the assertion will be not easily removable and may lead to subtle bugs.

Using dynamic read-only object references, we define a safe alternative for the assertion mechanism as follows:

```
Object>>safeAssert: aBlock
  self assert: aBlock asReadOnly
```

Like assert:, the method safeAssert: is defined in the class Object, the common superclass of all classes in Pharo. All classes will therefore be able to call this method. The parameter aBlock is a closure that will be evaluated by the original assert: with aBlock value. However, before delegating to the original implementation, we make the block read-only by rewriting all accesses the block does to variables defined outside itself. This includes all instance variable accesses, accesses to temporary variables from an outer block or method, as well as the special variables self and thisContext. Temporary variables are not wrapped in read-only handlers. The new assertion in our example thus becomes:

```
self safeAssert: [self hasTraitComposition and:
    [self traitComposition allTraits includes: aTrait]].
```

The preservation of object identity is important in our case: an object has the same identity as its read-only counterpart. When defining assertions, the self reference within the block provided to safeAssert: is the same identity than outside the block.

With DRO, the contract code can be executed guaranteeing that no side-effect occurs. Therefore, no bugs concerning unwanted state changes can happen.

4.2 Read-Only Data Structures

In general, read-only references provide the programmer with the possibility to hand out safe references to internal data-structures. As soon as the reference that is handed to a client as a read-only handle, it is guaranteed that no modification may happen through this reference. For example, we can use read-only structures to make sure that the programmer gets notified if (s)he attempts to modify a collection while iterating on it. This is indeed a subtle way to shot oneself in the foot; bugs introduced this way are difficult to track because they may rely on the order and identity of the elements. To be safe, the idiomatic way is to iterate on a copy of the collection and modify the original.

Java provides support for requesting an *unmodifiable collection* for any collection. This is a wrapper object that protects the collection from modification, while allowing the programmer to access or enumerate the content like a standard collection object. In Smalltalk, it is common to hand out a copy of a collection if the collection itself is used internally.

With dynamic read-only references, one solution is to offer a safeDo: method that propagates read-only status to the collection. Attempts to change the collection will then lead to an error.

```
Collection>>fullSafeDo: aBlock
  ^ self asReadOnly do: aBlock
```

Note here that the read-only status gets propagated to the block arguments, as we can easily see in the implementation of do:. The parameter passed to the block is read via the instance variable array, which is automatically wrapped in a read-only handler. This read-only reference provides read-only versions of all methods of class Array, including at:, resulting in a read-only reference passed to the block.

```
OrderedCollection>>do: aBlock
  "Override the superclass for performance reasons."
  | index |
  index := firstIndex.
  [ index <= lastIndex]
    whileTrue:
      [ aBlock value: (array at: index).
       index := index + 1 ]
```

All other variables referenced in the block, *i.e.,* globals, temporary variables of the enclosing methods or instance variables of the class are not read-only. The execution of the block itself is not done in a read-only context, just accesses to the collection (and all objects contained) are read-only.

Our solution with DRO provides a way to protect accesses to the entire object graph, starting at the read-only reference to the collection. In Section 7, we discuss that a better way of controlling the propagation is needed.

5 SMALLTALK/R: DRO Operational Semantics

We now formalize the model described in Section 3 by providing an operational semantics for Dynamic Read-Only execution using the formalism of ClassicJava [23]. The goal of this formalization is to provide the necessary technical description when implementing DRO in one's favorite language. It should be noted that this formalism models the read-only behavior. To keep it simple, it does not follow the implementation. This means that we do not model references. The time when the read-only behavior is propagated therefore is different: it happens at the time of message send, not handle creation. We decided to provide this simplified formalism as a first step, we plan to extend it as future work when we go beyond pure read-only behavior.

For this purpose we extend SMALLTALKLITE whose description is given in appendix. SMALLTALKLITE has been presented in our previous work [17], it should therefore not be considered as a contribution. SMALLTALKLITE is a Smalltalk-like dynamic language featuring single inheritance, message-passing, field access and update, and **self** and **super** sends. It is similar to CLASSICJAVA, but removes interfaces and static types, and fields are private, so only local or inherited fields may be accessed. SMALLTALKLITE is generic enough to be considered as a formal foundation targeting languages other than Smalltalk (e.g., Ruby).

We provide the necessary extension of SMALLTALKLITE to capture the semantics of Dynamic Read-Only execution. We call SMALLTALK/R the resulting extended language (Figure 5).

First, we augment the set of expressions and evaluating contexts with a new keyword, readonly. This keyword takes an expression as parameter and, at execution time, evaluates the expression and makes the result as read-only. Read-onlyness is expressed with $\lfloor ... \rfloor$. A value enclosed into these half square brackets cannot be mutated. A read-only value is written $\lfloor v \rfloor$ and can only result from evaluating a readonly expression. The readonly keyword belongs to the surface syntax, whereas $\lfloor ... \rfloor$ designates a read-only reference. The expression readonly e evaluates to a $\lfloor o \rfloor$ reference.

$$
\begin{array}{rl}
e = ... & \mid \textbf{readonly}\ e \\
E = ... & \mid \textbf{readonly}\ E \mid \lfloor E \rfloor \\
v, o = ... & \mid \lfloor oid \rfloor
\end{array}
$$

$$P \vdash \langle E[\textbf{readonly}\ o], \mathcal{S}\rangle \hookrightarrow \langle E[\lfloor o \rfloor], \mathcal{S}\rangle \quad [\textit{read-only}]$$

$$P \vdash \langle E[\lfloor o \rfloor .f], \mathcal{S}\rangle \hookrightarrow \langle E[\lfloor v \rfloor], \mathcal{S}\rangle \quad [\textit{get}]$$
where $\mathcal{S}(o) = \langle c, \mathcal{F}\rangle$ and $\mathcal{F}(f) = v$

$$P \vdash \langle E[\lfloor o \rfloor .f{=}v], \mathcal{S}\rangle \hookrightarrow \langle E[\textbf{error}], \mathcal{S}]\rangle \quad [\textit{set}]$$

$$P \vdash \langle E[\lfloor o \rfloor .m(v^*)], \mathcal{S}\rangle \hookrightarrow \langle E[\lfloor o \rfloor\ [\![e[v^*/x^*]]\!]_{c'}], \mathcal{S}\rangle \quad [\textit{send}]$$
where $\mathcal{S}(o) = \langle c, \mathcal{F}\rangle$ and $\langle c, m, x^*, e\rangle \in^*_P c'$

$$P \vdash \langle E[\textbf{super}\langle \lfloor o \rfloor , c\rangle .m(v^*)], \mathcal{S}\rangle \hookrightarrow \langle E[\lfloor o \rfloor\ [\![e[v^*/x^*]]\!]_{c''}], \mathcal{S}\rangle \quad [\textit{super}]$$
where $c \prec_P c'$ and $\langle c', m, x^*, e\rangle \in^*_P c''$ and $c' \leq_P c''$

Fig. 5. Extensions made on SMALLTALKLITE resulting in SMALLTALK/R

The immutability property is propagated through instance variable and global access such as class references (Section 3.2). In SMALLTALKLITE, the only global accesses permitted are class references in new expressions.

A new rule for the readonly keyword has to be added. The expression readonly o is simply reduced into $\lfloor o \rfloor$.

Subsequently, reduction rules must be adjusted to take the immutability property into account: field assignment leads to an error; calling a method propagates the read-only object reference into the method body; the read-only object reference is also propagated with super calls.

Message lookup is achieved through the notation $\langle c, m, x^*, e\rangle \in^*_P c'$: we look for a method called m, with the arguments x^*, and a method body e. The lookup begins in the class c and c' is the first class among superclasses of c that defines this method. $\langle m, x^*, e\rangle$ is a method definition, $\langle c, m, x^*, e\rangle \in c'$ is a different notation that says $\langle m, x^*, e\rangle$ is looked up starting in c.

No special treatment is needed for super calls. If the receiver is readonly (as in **super**$\langle \lfloor o \rfloor , c\rangle$), then the method lookup starts in the super class of c, that we call c'. Following the notation of CLASSICJAVA, direct subclass is designed using $c \prec_P c'$. The method to lookup is $\langle m, x^*, e\rangle$ and it is searched starting in c'. It is found in c'', using the operator $\in^*_P$. Naturally, we have the relation $c' \leq_P c''$. The formulation illustrates the simplicity of readonly feature of the DRO model.

6 Implementation

We implemented DRO in Pharo by (1) extending the virtual machine and (2) using a byte-code rewriting engine [24].

VM changes. The Squeak/Pharo Virtual machine was modified to support transparent proxies. Our implementation is more general than required for strict read-only execution.

It offers a per reference handle as presented in Section 3, while for DRO we only need a per class method dictionary containing the rewritten methods. We did that because we see dynamic read-only references as just one case of a more general scheme for security alternatives in the realm of object capability model [13].

Virtual machine modifications were required because in Smalltalk, reflection would allow one to easily change handles. Thus the user of a handle could corrupt the handle integrity and send forbidden messages. By implementing handles at the VM level we ensure that handles are tamperproof. In addition, handles have a different method lookup than normal objects: when a handle receives a message, the VM looks the method up in the handle's shadow class, but applies it on its target object. Also, the current implementation, never makes read-only integers, booleans, or nil objects, since those are naturally immutable. We changed the virtual machine to treat a handle and the object pointed to as identical. Technically this is realized by a special check in the byte-code implementing identity.

Runtime infrastructure. In addition to the VM changes, Figure 6 presents the principles of our implementation at the runtime level. When an object should offer a read-only behavior, a handle is created. The creation of the handle leads to the definition of a shadow class and its inheritance chain - in the style of Encapsulators [21, 25] A shadow class is an anonymous class which in the context of DRO will be used by the modified handle method lookup. In such shadow classes, methods are rewritten to implement the semantics described before.

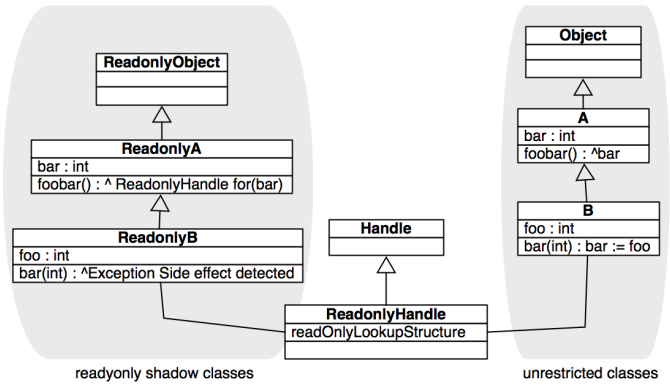

Fig. 6. Shadow classes as an implementation of DRO.

We use the byte-code transformation framework ByteSurgeon [24] to dynamically rewrite methods and install them in the shadow classes.

Shortcomings of the Current Impementation In the current implementation we do not support primitive virtual machine calls and weak references (*i.e.,* referenced ignored by the garbage collector). Both limitations will be resolved in a future version.

Benchmarks. We present preliminary benchmarks to asses the cost of implementing DRO on a real system. The benchmarks were done on a MacBook Pro Core2Duo 2.4Ghz using the MessageTally profiler.

MD ►*What kind of benchmark programs were used to obtain the data in section 6? Seems to me like some microbenchmark, but in any case it should be clearly qualified!*◄

We compare performance of a DRO enabled vm to a standard virtual machine, for both the case of using standard references and using read-only references. As we can see in the table, message sends are slowed down between 4 and 11 percent. In the case of read-only references, the overhead is maximal 20%.

MD ► *The rows and columns of the table on page 13 are not explained. What is "propagate"? What is "DRO VM (to handle)"?*◄

Very time-consuming is the creation of a handle in the case when the shadow class needs to be created. In this case, all methods of the class of the object and all superclasses are copied and modified. This naturally takes time but needs to be done only once for each class.

MD ►*The second part of the table looks strange as it contains mostly NA, so it seems as if the table is also NA to this situation.*◄

Message Passing (one send)	Classic VM	DRO VM	Slowdown	DRO VM (to handle)	Slowdown
normal	1.09 μs	1.21 μs	11.36 %	1.30 μs	19.15 %
super	1.63 μs	1.69 μs	4.06 %	1.79 μs	10 %
literal	1.11 μs	1.20 μs	8.22 %	1.29 μs	16.30 %
propagate	NA	NA	NA	45.73 μs	NA
Creating					
instance	5.96 μs	7.80 μs	30.96 %	NA	NA
handle (first)	NA	NA	NA	3,06 s	NA
handle (only access)	NA	NA	NA	46.18 μs	NA

7 Discussion

Scoping Propagation. In our approach, we cannot control the scope of immutability propagation: for the whole subgraph of objects reached during execution we construct the read-only overlay. Therefore a collection accessed as read-only cannot have elements with a different access behavior: in the following example, accessing the elements of the collection will propagate the immutability to the elements:

```
(aCol asReadOnly at: 1) changeStateOfElement
```

Our approach supports argument exposure [7] in the sense that when the collection is accessed as read-only, its elements are read-only too. A related problem is the question of how to allow harmless side-effects. For example, often programs need to cache calculated values. With a strict read-only execution model, this is not possible.

The problem is that it is not clear how a scoping mechanism for limiting read-only propagation should look like. Care needs to be taken to not break the read-only model

completely. This is part of our future work: we imagine both a static scoping mechanism based on program annotations and one based on dynamically scoped variables.

Implementation Alternatives. The implementation approach chosen uses rewriting byte-codes to change behavior of methods combined with hidden (per reference) object proxies that are realized by changing the virtual machine. Alternatively, we could have generalized the idea of a per-object immutability bit and encoded immutability in the object pointer. Dynamic languages already encode small integers directly in the pointer using a tag bit. Extending tagging to multiple bits has been done and is especially feasible in systems with 64bit pointers.

We decided against this implementation strategy as it would constrain the model to just be about immutability, with no way of controlling the propagation. As we discussed before, scoping the propagation of read-only behavior is essential. Besides propagation, with having handles be special objects (hidden by the virtual machine), but in the end fairly normal nevertheless, we can start to experiment with other mechanism besides pure read-only behavior. All these experiments would not be possible in a system that encodes behavioral modification in a single bit.

An interesting question is how much of the implementation of the transparent handles can be realized without virtual machine changes. The reason for the VM change is the need to hide the proxy completely: the proxy is indistinguishable from the object and there is no way (not even using reflection) to reference the object directly. Purely reflective approaches (*i.e.,* rewriting bytecode for identity) are always visible to introspection and therefore not easy to realize in a completely transparent way.

Threads. Threads are interesting to look at in the context of our read-only model. We do not construct globally immutable object graphs, the object graph is only read-only when accessed through a read-only reference. In general, it is of course not guaranteed that objects are only referenced by read-only references. Therefore, DRO can not guarantee any immutability properties for shared resources. The non-atomic step-by-step propagation of read-only itself is thread-safe. Read-only execution does not change the state of any existing object, reads happen the same as in normal execution (non-atomic) and of course are impacted by other threads changing the same data-structure just as any normal execution.

In the past, work on context-orientation often encoded context as a per-thread property. [26, 27]. But it is not clear that the concepts of *context* and *thread* should be that closely related. In our model, read-only is orthogonal to threads. Propagation happens along the flow of references lazily driven by the control flow. This property will be especially interesting as soon as the ideas are used for other properties than read-only behavior.

Towards more Secure Systems. Dynamic Read-Only references offer a good infrastructure but not the complete solution to build more secure systems. First, the kind of access rights they offer is limited, it is either full access or read-only, while we expect a large range of different rights [7]. Second, a dedicated security model is missing. Still this is interesting to discuss the limits of DRO in this specific scenario.

Let's illustrate this point. In Smalltalk, the reflective structure, *i.e.,* the hierarchy of all classes and methods, is represented by normal objects and can therefore be changed at runtime. Although this is the basis for reflection, it may lead to problematic situations. If

we consider method addition to a class, we can either use the official API by sending the message #addSelector:withMethod: to the class, or directly modify the internal structure of the class, as any object. A class keeps its methods in a dictionary, which can be easily accessed through the accessor methodDict. Therefore, it is tempting to directly modify the method dictionary: aClass methodDict at: aSelector put: aMethod.

But manipulating the method dictionary directly will lead to errors [3], for example internal data-structures are not updated and the cache of the VM are not reset.To solve this problem, we can offer a read-only interface to the class internal representation: the user will not be able to modify the internal class structure. However, we then have to grant access accordingly to the public API. If we offer a read-only version of the object class, then even using the public API method #addSelector:withMethod: is useless, since the complete object graph is read-only. Otherwise, we can provide a dedicated read-only interface, but without an additional model to grant access, a malicious developer can still obtain references to the unprotected class. Generalizing DRO to solve this problem is clearly a future research direction.

8 Related Work

The work presented in this paper takes place in a large spectrum of other works ranging from ownership control to capabilities, via controlling interfaces and context-oriented programming. We present here the most significant work with a stress on dynamically typed solutions, but the list is not exhaustive.

Immutability. VisualWorks Smalltalk supports immutable objects: a per-object flag that tells whether the fields of the object may be modified. Once created, immutable objects cannot be modified via any reference, and the flag does not propagate to the object subgraph.

In School [28], the old qualifier prevents the modification of the fields of old objects. On old objects, the type checker only allows invoking side-effect free methods, and treats the return values from these invocations as old. This ensures that the only non-old (and thus mutable) objects that a method can use are the ones it creates itself.

Hakonen *et al* [10] propose the concept of *deeply immutable references*; they only discuss possible implementation strategies without presenting a working implementation. In Javari, Birka *et al* [14] extend Java with a static type system of transitively read-only references. These works are the most similar to our dynamic read-only references; the main difference is that they are proposed for statically typed languages. In particular, Javari methods have to be declared read-only *à priori*; unmodified Java code is conservatively considered to have side-effects. In contrast, our approach does not require any modification besides the initial creation of a read-only reference. Javari's read-only references can still authorize side effects on some fields that are not part of the object's abstract state; for instance, this allows caching or immutable collections of mutable elements. Currently, our implementation propagates immutability to all fields. Finally, since Javari programs runs on the unmodified JVM, it is possible that object wrappers break identity, while our handles are transparent.

Joe-E is a subset of Java based on an object-capability model supporting purely functional methods and type checking [2, 13]. In Joe-E, a purely functional method does

not have side-effects and its behavior only depends on its arguments. In our example of contracts, ensuring functional purity would be too strong: a pre-condition could perfectly rely on internal side-effects, as long as it does not change the state of the rest of the program. Functional purity is also difficult to ensure in the presence of both late binding and imperative code, without resorting to an entirely different programming style.

Alias Control and Dynamic Object Ownership. Dynamic object ownership [3, 9] is one of the rare propositions to control alias in the context of dynamically typed languages. Dynamic object ownership implements Flexible Alias Ownership [8]: every object which is part of the representation of an aggregate object is owned by that object and should not be visible outside the aggregate. The ownership of every object is stored into a dedicated field and it is used to verify the validity of every message send. Dynamic ownership enforces *representation encapsulation*, which states that an encapsulated object (representation object in Noble's jargon) can only be accessed via its encapsulating object, and *external independence*, which states that an object should not depend on the mutable state of an object that is external to it. Using ownership, a visibility rule is defined: an object is visible to another one if they are both encapsulated inside the same object (if they belong to the same ownership tree). Messages are then only sent if the receiving object is visible to the sending object.

The problem that DRO solves is different but related to the one of dynamic ownership. The goal of dynamic read-only references is not to enforce encapsulation per se, but to offer different interfaces of the same object, dynamically and to different clients. We do not distinguish object ownership or containment, nor do we enforce that components should be accessed through their owner.

Limiting interface approaches. Encapsulation policies [15, 16] belong to the family of work that restrict interfaces. Like DRO, encapsulation policies have per-reference semantics. An object can expose different interfaces based on its different references. However, the approaches have two differences. First, there is no propagation of propriety in encapsulation policies. Second, DRO do not change the original object interface, and the cancellation occurs inside method body depending on the execution path, while with encapsulation policies forbid whole methods *à priori*.

In Erights [13], capabilities are used to enforce security by not giving the possibility of a client to access the object interface. While the focus of DRO is different from capabilities (restricting interface), we plan to generalize it in the future to support capability-based security propagation [7].

Context-Oriented Programming. To a certain extent, the work presented in this paper is related to context-oriented programming in the sense that the functionality of an object is modified depending on a context [26]: we offer an additional read-only context for object execution. But other than prior work on context oriented programming, our read-only context is not purely defined by the thread of execution.

ContextL [29] 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. With DRO, both the definition and propagation of changed behavior is dynamic and lazily following the flow of data in the application. We believe that this

is an interesting property for context-oriented programming. Scoping side-effects has been the focus of two recents works. Worlds [30] provide a way to control and scope side-effects in Javascript. Tanter proposed a more flexible scheme: contextual values [31] are scoped by a very general context function.

MD ►*Missing here: the wrapper based AOP stuff of Michael Haupt*◄

9 Conclusion

Being able to assert that during a given scope an object cannot be modified is a valuable property, for example for assertions and design-by-contract. While most existing prior work focused on type systems to support the confinement and controlled accesses to aliases, there is a need to offer a solution for latently typed languages where a static approach is not possible. In this paper, we present Dynamic Read-Only references. Our solution to the propagation of immutability is based on the dynamic propagation of the property following the object graph, lazily driven by the control flow of the program.

Our future work is to understand what abstractions should be proposed to the programmer to control the scope of the propagation. In addition, following [7], we would like to see if the same mechanism can be applied to various alias semantics: immutable, borrowed, unique, shared. Finally one question we want to investigate is how the concepts presented here can be applied to offer a basis for security abstractions in dynamically typed languages.

Acknowledgments We warmly thank James Noble for his excellent feedback on early versions of the paper and discussions about *ConstraintedJava*. We gratefully acknowledge the financial support of the DGA of the French government for the grant of M. Suen. Marcus Denker gratefully 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). This work has been partially sponsored by the STICAmSud CoRea Project.

References

1. Meyer, B.: Applying design by contract. IEEE Computer (Special Issue on Inheritance & Classification) **25**(10) (October 1992) 40–52
2. Finifter, M., Mettler, A., Sastry, N., Wagner, D.: Verifiable functional purity in java. In: CCS'08. (2008) 27–31
3. Gordon, D., Noble, J.: Dynamic ownership in a dynamic language. In: DLS '07: Proceedings of the 2007 symposium on Dynamic languages, New York, NY, USA, ACM (2007) 41–52
4. Hogg, J.: Islands: Aliasing protection in object-oriented languages. In: Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'91), ACM SIGPLAN Notices. Volume 26. (November 1991) 271–285
5. Almeida, P.S.: Balloon types: Controlling sharing of state in data types. In: Proceedings of ECOOP '97. LNCS, Springer Verlag (June 1997) 32–59
6. Clarke, D.G., Potter, J.M., Noble, J.: Ownership types for flexible alias protection. In: Proceedings OOPSLA '98, ACM Press (1998) 48–64
7. Boyland, J., Noble, J., Retert, W.: Capabilities for aliasing: A generalisation of uniqueness and read-only. In Knudsen, J.L., ed.: Proceesings ECOOP 2001. Number 2072 in Lecture Notes in Computer Science, Springer (June 2001) 2–27

8. Noble, J., Potter, J., Vitek, J.: Flexible alias protection. In Jul, E., ed.: Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP'98). Volume 1445 of LNCS., Brussels, Belgium, Springer-Verlag (July 1998) 158–185
9. Noble, J., Clarke, D., Potter, J.: Object ownership for dynamic alias protection. In: Proceedings TOOLS '99. (November 1999)
10. Hakonen, H., Leppänen, V., Raita, T., Salakoski, T., Teuhola, J.: Improving object integrity and preventing side effects via deeply immutable references. In: Fenno-Ugric Symposium on Software Technology. (1999) 139–150
11. Miller, M.S., Shapiro, J.S.: Paradigm regained: Abstraction mechanisms for access control. In: Proceedings of the Eigth Asian Computing Science Conference. (2003) 224–242
12. Fong, P.W.L., Zhang, C.: Capabilities as alias control: Secure cooperation in dynamically extensible systems. Technical report, Department of Computer Science, University of Regina (2004)
13. Miller, M.S.: Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA (May 2006)
14. Birka, A., Ernst, M.D.: A practical type system and language for reference immutability. In: OOPSLA'2004. (2004) 35–49
15. Schärli, N., Ducasse, S., Nierstrasz, O., Wuyts, R.: Composable encapsulation policies. In: Proceedings of European Conference on Object-Oriented Programming (ECOOP'04). Volume 3086 of LNCS., Springer Verlag (June 2004) 26–50
16. Schärli, N., Black, A.P., Ducasse, S.: Object-oriented encapsulation for dynamically typed languages. In: Proceedings of 18th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'04). (October 2004) 130–149
17. Bergel, A., Ducasse, S., Nierstrasz, O., Wuyts, R.: Stateful traits and their formalization. Journal of Computer Languages, Systems and Structures **34**(2-3) (2008) 83–108
18. Friedman, D.P., Wand, M.: Reification: Reflection without metaphysics. In: LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming, New York, NY, USA, ACM (1984) 348–355
19. Lienhard, A.: Dynamic Object Flow Analysis. Phd thesis, University of Bern (December 2008)
20. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, Reading, Mass. (1995)
21. Pascoe, G.A.: Encapsulators: A new software paradigm in Smalltalk-80. In: Proceedings OOPSLA '86, ACM SIGPLAN Notices. Volume 21. (November 1986) 341–346
22. Pratikakis, P., Spacco, J., Hicks, M.: Transparent proxies for java futures. In: OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications, New York, NY, USA, ACM Press (2004) 206–223
23. Flatt, M., Krishnamurthi, S., Felleisen, M.: A programmer's reduction semantics for classes and mixins. Technical Report TR 97-293, Rice University (1999)
24. Denker, M., Ducasse, S., Tanter, É.: Runtime bytecode transformation for Smalltalk. Journal of Computer Languages, Systems and Structures **32**(2-3) (July 2006) 125–139
25. Ducasse, S.: Evaluating message passing control techniques in Smalltalk. Journal of Object-Oriented Programming (JOOP) **12**(6) (June 1999) 39–44
26. Hirschfeld, R., Costanza, P., Nierstrasz, O.: Context-oriented programming. Journal of Object Technology **7**(3) (March 2008)
27. Denker, M., Suen, M., Ducasse, S.: The meta in meta-object architectures. In: Proceedings of TOOLS EUROPE 2008. Volume 11 of LNBIP., Springer-Verlag (2008) 218–237
28. Ierusalimschy, R., de la Rocque Rodriguez, N.: Side-effect free functions in object-oriented languages. Computer Languages **3/4**(21) (1995) 129–146

29. Costanza, P., Hirschfeld, R.: Language constructs for context-oriented programming: An overview of ContextL. In: Proceedings of the Dynamic Languages Symposium (DLS) '05, co-organized with OOPSLA'05, New York, NY, USA, ACM (October 2005) 1–10
30. Warth, A., Kay, A.: Worlds: Controlling the scope of side effects. Technical Report RN-2008-001, Viewpoints Research (2008)
31. Tanter, É.: Contextual values. In: Proceedings of the 4th ACM Dynamic Languages Symposium (DLS 2008), Paphos, Cyprus, ACM Press (jul 2008) To appear.
32. Flatt, M., Krishnamurthi, S., Felleisen, M.: Classes and mixins. In: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press (1998) 171–183
33. Felleisen, M., Hieb, R.: The revised report on the syntactic theories of sequential control and state. Theor. Comput. Sci. **103**(2) (1992) 235–271

$$
\begin{array}{rl}
P = & \mathit{defn}^* e \\
\mathit{defn} = & \textbf{class}\ c\ \textbf{extends}\ c\ \{\ f^* \mathit{meth}^*\ \} \\
e = & \textbf{new}\ c \mid x \mid \textbf{self} \mid \mathsf{nil} \\
& \mid f \mid f{=}e \mid e.m(e^*) \\
& \mid \textbf{super}.m(e^*) \mid \textbf{let}\ x{=}e\ \textbf{in}\ e
\end{array}
\qquad
\begin{array}{rl}
\mathit{meth} = & m(x^*)\ \{\ e\ \} \\
c = & \text{a class name} \mid \mathsf{Object} \\
f = & \text{a field name} \\
m = & \text{a method name} \\
x = & \text{a variable name}
\end{array}
$$

Fig. 7. SMALLTALKLITE syntax

$$
\begin{array}{rl}
\epsilon = & v \mid \textbf{new}\ c \mid x \mid \textbf{self} \mid \epsilon.f \mid \epsilon.f{=}\epsilon \mid \epsilon.m(\epsilon^*) \mid \textbf{super}\langle o, c\rangle.m(\epsilon^*) \mid \textbf{let}\ x{=}\epsilon\ \textbf{in}\ \epsilon \\
E = & [\,] \mid o.f{=}E \mid E.m(\epsilon^*) \mid o.m(v^*\ E\ \epsilon^*) \\
& \mid \textbf{super}\langle o, c\rangle.m(v^*\ E\ \epsilon^*) \mid \textbf{let}\ x{=}E\ \textbf{in}\ \epsilon \\
v, o = & \mathsf{nil} \mid \mathit{oid}
\end{array}
$$

Fig. 8. SMALLTALKLITE annotated syntax

Appendix: SMALLTALKLITE

The syntax of SMALLTALKLITE is shown in Figure 7. SMALLTALKLITE is similar to CLASSICJAVA, but eliding the features related to static typing.

To simplify the reduction semantics of SMALLTALKLITE, we adopt an approach similar to that used by Flatt *et al* [32], we annotate field accesses and **super** sends with additional static information that is needed at "run-time". This annotated syntax is shown in Figure 8. The figure also specifies the evaluation contexts for the annotated syntax in Felleisen and Hieb's notation [33].

$P \vdash \langle \epsilon, \mathcal{S}\rangle \hookrightarrow \langle \epsilon', \mathcal{S}'\rangle$ means that we reduce an annotated expression ϵ in the context of a (static) program P and a (dynamic) store of objects $\mathcal{S}$ to a new expression ϵ' and (possibly) updated store $\mathcal{S}'$. An annotated expression ϵ is essentially an expression e in which field names are decorated with their object contexts, *i.e.,* f is translated to $o.f$, and **super** calls are decorated with their object and class contexts. Annotated expressions and their subexpressions reduce to a value, which is either an object identifier or nil. Subexpressions may be evaluated within an expression context E.

The store consists of a set of mappings from object identifiers $\mathit{oid} \in \mathrm{dom}(\mathcal{S})$ to tuples $\langle c, \{f \mapsto v\}\rangle$ representing the class c of an object and the set of its field values. The initial value of the store is $\mathcal{S} = \{\}$.

Translation from the main expression to an initial annotated expression is specified out by the $o[\![e]\!]_c$ function (see Figure 9). This binds fields to their enclosing object context and binds **self** to the *oid* of the receiver. The initial object context for a program is nil. (*i.e.,* there are no global fields accessible to the main expression). So if e is the main expression associated to a program P, then $\mathsf{nil}[\![e]\!]_{\mathsf{Object}}$ is the initial annotated element. The reductions are summarized in Figure 10.

new c [*new*] reduces to a fresh *oid*, bound in the store to an object whose class is c and whose fields are all nil. A (local) field access [*get*] reduces to the value of the field. Note that it is syntactically impossible to access a field of another object. The annotated expression notation $o.f$ is only generated in the context of the object o. Field update

$$
\begin{array}{rlrl}
o[\![\textbf{new } c]\!]_c &= \textbf{new } c & o[\![f]\!]_c &= o.f \\
o[\![x]\!]_c &= x & o[\![f{=}e]\!]_c &= o.f{=}o[\![e]\!]_c \\
o[\![\textbf{self}]\!]_c &= o & o[\![e.m(e_i^*)]\!]_c &= o[\![e]\!]_c.m(o[\![e_i]\!]_c^*) \\
o[\![\mathsf{nil}]\!]_c &= \mathsf{nil} & o[\![\textbf{super}.m(e_i^*)]\!]_c &= \textbf{super}\langle o, c\rangle.m(o[\![e_i]\!]_c^*) \\
 & & o[\![\textbf{let } x{=}e \textbf{ in } e']\!]_c &= \textbf{let } x{=}o[\![e]\!]_c \textbf{ in } o[\![e']\!]_c
\end{array}
$$

Fig. 9. Annotating expressions

$$
\begin{array}{ll}
P \vdash \langle E[\textbf{new } c], \mathcal{S}\rangle \hookrightarrow \langle E[\mathit{oid}], \mathcal{S}[\mathit{oid} \mapsto \langle c, \{f \mapsto \mathsf{nil} \mid \forall f, f \in_P^* c\}\rangle]\rangle & [\mathit{new}] \\
\quad \text{where } \mathit{oid} \notin \mathrm{dom}(\mathcal{S}) & \\
P \vdash \langle E[o.f], \mathcal{S}\rangle \hookrightarrow \langle E[v], \mathcal{S}\rangle & [\mathit{get}] \\
\quad \text{where } \mathcal{S}(o) = \langle c, \mathcal{F}\rangle \text{ and } \mathcal{F}(f) = v & \\
P \vdash \langle E[o.f{=}v], \mathcal{S}\rangle \hookrightarrow \langle E[v], \mathcal{S}[o \mapsto \langle c, \mathcal{F}[f \mapsto v]\rangle]\rangle & [\mathit{set}] \\
\quad \text{where } \mathcal{S}(o) = \langle c, \mathcal{F}\rangle & \\
P \vdash \langle E[o.m(v^*)], \mathcal{S}\rangle \hookrightarrow \langle E[o[\![e[v^*/x^*]]\!]_{c'}], \mathcal{S}\rangle & [\mathit{send}] \\
\quad \text{where } \mathcal{S}[o] = \langle c, \mathcal{F}\rangle \text{ and } \langle c, m, x^*, e\rangle \in_P^* c' & \\
P \vdash \langle E[\textbf{super}\langle o, c\rangle.m(v^*)], \mathcal{S}\rangle \hookrightarrow \langle E[o[\![e[v^*/x^*]]\!]_{c''}], \mathcal{S}\rangle & [\mathit{super}] \\
\quad \text{where } c \prec_P c' \text{ and } \langle c', m, x^*, e\rangle \in_P^* c'' \text{ and } c' \leq_P c'' & \\
P \vdash \langle E[\textbf{let } x{=}v \textbf{ in } \epsilon], \mathcal{S}\rangle \hookrightarrow \langle E[\epsilon[v/x]], \mathcal{S}\rangle & [\mathit{let}]
\end{array}
$$

Fig. 10. Reductions for SMALLTALKLITE

[*set*] simply updates the corresponding binding of the field in the store. When we send a message [*send*], we must look up the corresponding method body e, starting from the class c of the receiver o. The method body is then evaluated in the context of the receiver o, binding **self** to the receiver's *oid*. Formal parameters to the method are substituted by the actual arguments. We also pass in the actual class in which the method is found, so that **super** sends have the right context to start their method lookup.

super sends [*super*] are similar to regular message sends, except that the method lookup must start in the superclass of class of the method in which the **super** send was declared. When we reduce the **super** send, we must take care to pass on the class c'' of the method in which the **super** method was found, since that method may make further **super** sends. **let in** expressions [*let*] simply represent local variable bindings.

Errors occur if an expression gets "stuck" and does not reduce to an *oid* or to nil. This may occur if a non-existent variable, field or method is referenced (for example, when sending any message to nil). For the purpose of this paper we are not concerned with errors, so we do not introduce any special rules to generate an error value in these cases.