HAL
open science

Propagation of Behavioral Variations with Delegation Proxies

Camille Teruel, Erwann Wernli, Stéphane Ducasse, Oscar Nierstrasz

To cite this version:

Camille Teruel, Erwann Wernli, Stéphane Ducasse, Oscar Nierstrasz. Propagation of Behavioral Variations with Delegation Proxies. LNCS Transactions on Aspect-Oriented Software Development, 2015, 8989, pp.63-95. 10.1007/978-3-662-46734-3_2 . hal-01135706

HAL Id: hal-01135706

https://hal.science/hal-01135706v1

Submitted on 23 Aug 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Copyright

Propagation of Behavioral Variations with Delegation Proxies (Preprint)*

Camille Teruel[1], Erwann Wernli[2], Stéphane Ducasse[1], and Oscar Nierstrasz[2]

[1] RMOD, INRIA Lille Nord Europe, France
[2] Software Composition Group, University of Bern, Switzerland

Abstract. Scoping behavioral variations to dynamic extents is useful to support non-functional concerns that otherwise result in cross-cutting code. Unfortunately, such forms of scoping are difficult to obtain with traditional reflection or aspects. We propose *delegation proxies*, a dynamic proxy model that supports behavioral intercession through the interception of various interpretation operations. Delegation proxies permit different behavioral variations to be easily composed together. We show how delegation proxies enable behavioral variations that can *propagate* to dynamic extents. We demonstrate our approach with examples of behavioral variations scoped to dynamic extents that help simplify code related to safety, reliability, and monitoring.

Keywords: Reflection, proxy, delegation, propagation, dynamic extent

1 Introduction

Non-functional concerns like monitoring or reliability typically result in code duplication in the code base. The use of aspects is the de-facto solution to factor out such boilerplate code into a single place. Aspects enable the scoping of behavioral variations in space (with a rich variety of static pointcuts), in time (with dynamic aspects), and in the control flow (with the corresponding pointcuts). Scoping a behavioral variation to the *dynamic extent* [TFD+09] of an expression is however challenging, since scoping between threads is not easily realized with aspects. Traditional reflection and meta-object protocols suffer from similar limitations.

This is unfortunate since scoping behavioral variations to dynamic extents increases the expressiveness of the language in useful ways [Tan08,TFD+09]. With such a form of scoping, it is possible to execute code in a read-only manner [ADD+10] (thus improving safety), or to track all state mutations to ease

* In Transactions on Aspect-Oriented Software Development XII, Lecture Notes in Computer Science 8989 p. 63-95, Springer Berlin Heidelberg, 2015.
DOI: 10.1007/978-3-662-46734-3_2

recovery in case of errors (thus improving reliability), or to trace and profile code at a fine-grained level (thus improving monitoring).

We show in this paper that with minor changes to the way dynamic proxies operate, it becomes possible to implement behavioral variations that are scoped to dynamic extents. A dynamic proxy [VCM10,MPBD+11,Eug06][3] is a special object that mediates interactions between a client object and another target object. When the client sends a message to the proxy, the message is intercepted and reified to allow specific processing. To scope variations to dynamic extents using proxies, we must first slightly adapt the dynamic proxy mechanism. We refer to our approach as *delegation proxies*.

Delegation proxies have the following characteristics:

- They operate by delegation [Lie86], *i.e.*, by rebinding self-references as in prototype-based languages,
- they intercept state accesses, both for regular fields and variables captured in closures, and
- they intercept object creation.

With delegation proxies, a proxy can encode a behavioral variation that will be consistently *propagated* to all objects accessed during the evaluation of a message send (*i.e.*, its dynamic extent).

Delegation proxies have several positive properties. First, delegation proxies can avoid *infinite regressions*. In aspect-oriented programming, infinite regressions can arise when an advice triggers the associated pointcut.[4] In reflective architectures, infinite regression arises when reflecting on code that is used to implement the reflective behavior itself. The conventional solution to this problem is to explicitly model the different levels of execution [CKL96,DSD08,Tan10]. With delegation proxies, a proxy and its target are distinct objects and the propagation is enabled only for the proxy. No variation is active when executing the code that implements the behavioral variation as long as all messages are sent to base objects. Second, behavioral variations expressed with delegation proxies *compose*, similarly to aspects. For instance, tracing and profiling behavioral variations can be implemented by separate proxies that can be combined to apply both behavioral variations. Third, like other dynamic proxy implementations, delegation proxies naturally support *partial reflection* [TNCC03] at an object-level granularity. Behavioral intercession is enabled only for proxies. All other objects in the system — including the target — remain unaffected and pay no performance overhead.

In this paper, we explore and demonstrate the flexibility of delegation proxies in Smalltalk with the following contributions:

- A model of proxies based on delegation that intercepts object instantiations and state accesses (including variables in closures) (Section 2);

[3] A similar mechanism is called encapsulator [Pas86]

[4] A typical workaround for this problem is to constrain pointcut definitions: *e.g.*, a pointcut p within an aspect A is rewritten: p `&& !cflow(adviceexecution()) && within(`A`)`

– A technique to use delegation proxies to scope variations to dynamic extents (Section 2);
– Several examples of useful applications of variations scoped to dynamic extents (Section 4);
– A formalization of delegation proxies and the propagation technique (Section 6);
– An implementation of delegation proxies in Smalltalk based on code generation (Section 7).

2 Delegation Proxies

Delegation proxies are dynamic proxies that operate by delegation in contrast to classical dynamic proxies that operate by forwarding. After presenting dynamic proxies, we show why delegation is a better choice than forwarding for proxy-based behavioral intercession.

To the best of our knowledge only EcmaScript 6 proxies use delegation. However, this is incidental since EcmaScript, as a prototype-based object language, uses delegation to implement object inheritance [Lie86].

We will see that delegation enables the interception of interpretation operations that occur during method execution — like object state accesses and object creation — and allows behavioral variations to be composed naturally. This is an important matter to consider in the context of class-based object-oriented languages.

2.1 Dynamic proxies

A dynamic proxy is an object that acts as a surrogate for another object, called its *target*. A proxy mediates interactions between its target and its clients. The behavior of a proxy is typically defined by a separate object called its *handler*, whose methods are called *traps* [VCM10]. When an interpretation operation — like the reception of a message, an access to an object instance variable, *etc.*— is applied to a proxy, the proxy reifies the operation and instead invokes the trap that corresponds to that operation in its handler. The handler can take some actions before, after or even instead of performing the original operation on the target. Figure 1 shows the relationships between a proxy, its handler and its target with an example of message interception.

The dynamic proxy mechanism implements a kind of behavioral intercession that alters the interpretation of a program at the granularity of objects. We refer to the different alterations of the interpretation process as *behavioral variations*. Tracing, profiling, read-only, *etc.*, are examples of behavioral variations.

The distinction between proxies and handlers is the application of a principle called *stratification* [BU04,VCM10,MPBD+11]. Stratification stipulates that the meta-level must be separated from base-level. In the context of dynamic proxies, this principle avoids name conflicts between application methods and handler

traps, *i.e.*, between the base-level and the meta-level. This means that a proxy can expose exactly the same interface as its target.

By using proxies as the targets of other proxies, we obtain chains of proxies as depicted in Figure 2. In this case, when an interpretation operation is intercepted, the corresponding trap should be triggered in each of the handlers of the proxies in the order specified by the chain. This offers a natural and convenient way to compose multiple behavioral variations.

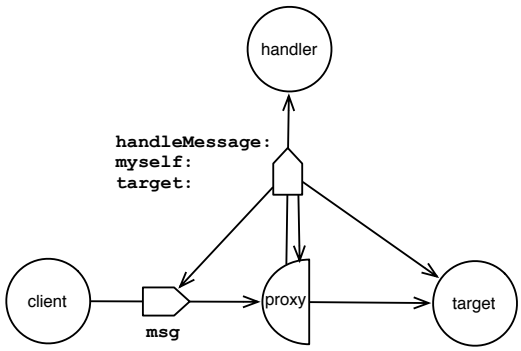

Fig. 1. Example of message interception. First, the client sends the message `msg` to a proxy. Then, the proxy intercepts the message and invokes the handler trap associated with message reception (`handleMessage:proxy:target:`) with three arguments: the reified message, the proxy itself and the target.

2.2 Forwarding *vs* Delegation

We explore the differences between forwarding and delegation semantics in the context of proxy-based intercession. The operational difference reduces to how self-references are bound in methods that match intercepted messages. Traditional dynamic proxy implementations found in class-based object-oriented languages operate by forwarding. Once a message has been intercepted by a proxy, the proxy may decide to *forward* the message to its target: the method corresponding to the message is executed with self-references bound to the target. This implies that the proxy loses control of the execution. With delegation, the proxy may decide to *delegate* the message: the method corresponding to the intercepted message is executed with self-references bound to the proxy itself. This implies that the proxy keeps the control of the execution.

With delegation, the proxy can intercept interpretation operations that occur during a method execution: object state reads, object state writes, object creation, literal resolution, *etc.* We refer to these interpretation operations as *sub-method operations*. Moreover, with delegation the identity of the proxy that

originally intercepted the operation is maintained; this permits behavioral variations to be composed in the case of chains of proxies.

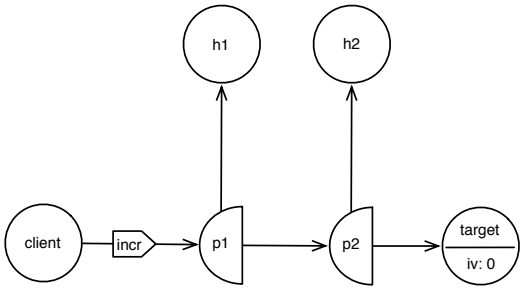

Fig. 2. A chain of two proxies. When an interpretation operation is intercepted, the corresponding trap is triggered in the handlers *h1* and *h2* in order.

In Figure 2, we have two proxies *p1* and *p2*, their respective handlers *h1* and *h2*, and an object *target*. The target of *p1* is *p2* and the target of *p2* is *target*. This forms a chain of two proxies. A client object sends a message *incr* that is supposed to change the state of the target object by modifying its instance variable *iv*.

With forwarding execution begins with the message by *p1* being intercepted and triggering the *message-reception* trap of its handler *h1*. The handler may at some point forward the intercepted message to the target of *p1* — namely the proxy *p2* — losing control of the execution at the same time. The same scenario applies to *p2*: it intercepts the message, invokes its handler's message-reception trap, its handler forwards the message to *target* which then executes the method associated with the message normally and increments the value of its instance variable *iv*.

We can see that the behavioral variations of *p1* and *p2* are necessarily limited to interception of message reception since *p1* — the proxy that originally intercepted the message — loses the control of the execution. It thus cannot intercept sub-method operations. Even if the execution of the method of *target* associated with the message performs a self-send, this latter message send will not be intercepted.

With delegation the execution begins the same: *p1* intercepts the message and invokes the message-reception trap of its handler. But instead of forwarding the message to *p2*, *h1* instead delegates the message: it specify that the receiver should be rebound to *p1*. Then *p2* intercepts that message, passes it to *h2*, which applies its behavioral variation and delegates the message to *target*. At this point *p1* is still the receiver and has still control over the execution: it can intercept sub-method operations, in particular the modification of *iv*. In reaction to this interception, *p1* invokes the state-write trap of its handler *h1*. The state-

write trap of *h2* is then invoked. This example shows that delegation permits behavioral variations to be composed in the context of sub-method operation interception.

To sum up, using delegation instead of forwarding for proxy-based intercession permits proxies to intercept sub-method operations and to compose behavioral variations by forming chains of proxies.

3 Propagation

In the previous section we saw how delegation proxies work. In this section we present the concept of *propagation* of behavioral variation and its reflective implementation in terms of delegation proxies. This technique permits a proxy to be created for a target object that will scope a behavioral variation to the dynamic extents of the messages it receives. These dynamic extents are the parts of the execution delimited by the processing of a message received by the proxy, from the reception of the message until the corresponding method returns. All objects accessed during this dynamic extent are consistently represented by proxies that are created on-demand. We refer to the first proxy that initiates the propagations as the *root proxy*. Other proxies created during the propagation are called *non-root proxies*. The root proxy can be seen as the entry point to a lazily-created parallel object graph as depicted in Figure 3.

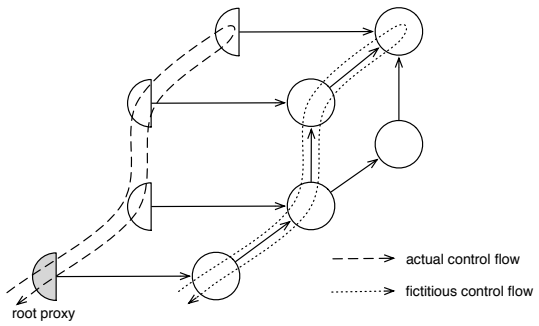

Fig. 3. A depiction of propagation (handlers are omitted for clarity). A root proxy (in grey) wraps a target object that is connected to some object graph. The dashed line depicts the actual control flow of the execution. Proxies are created on need and are eligible for garbage collection once the control flow leaves them. The actual control flow parallels a fictitious control flow (dotted line) *i.e.*, the control flow that would have resulted if execution had not been intercepted by the root proxy.

3.1 Tracing example

We will illustrate propagation with a tracing example. Let us consider the Smalltalk method `Integer»fib`[5] which computes the Fibonacci value of an integer using recursion:

```
Integer»fib
    self < 2 ifTrue: [ ↑ self ].
    ↑ (self - 1) fib + (self - 2) fib
```

Listing 1. Fibonacci computation

The computation of the Fibonacci value of `2` corresponds to the following sequence of message sends (first the receiver of the message, then the message with its arguments):

```
2 fib
2 < 2
false ifTrue: [ ↑ self ]
2 - 1
1 fib
1 < 2
true ifTrue: [ ↑ self ]
[ ↑ self ] value
2 - 2
0 fib
0 < 2
true ifTrue: [ ↑ self ]
[ ↑ self ] value
1 + 0
```

Listing 2. Trace of `2 fib`

To automatically trace message sends, we can use a proxy to intercept message sends and print them. A tracing proxy can be obtained by instantiating a proxy with a tracing handler. For convenience, we define the method `Object»tracing`, which returns a tracing proxy for any object. For instance, the expression `2 tracing` returns a tracing proxy for the number `2`.

```
Object»tracing
  ↑ Proxy handler: TracingHandler new target: self.
```

Listing 3. Creation of a tracing proxy

To trace messages, the tracing handler must define a *message reception* trap that prints the name of the reified message. Listing 4 shows the code of such a *message* trap:

[5] The notation `Integer»fib` refers to the method `fib` of the class `Integer`. In Smalltalk, closures are expressed with square brackets (`[ ... ]`) and booleans are objects. The method `ifTrue:` takes a closure as argument: if the receiver is the object `true`, the closure is evaluated by sending it the message `value`. The up-arrow (`↑ ...`) denotes a return expression.

```
TracingHandler»handleMessage: m myself: p target: t
    Transcript
          print: t asString;
          space;
          print: m asString;
          cr.
    ↑ t perform: m myself: p.
```

Listing 4. A simple tracing handler

The reflective invocation with `perform:` takes one additional parameter `myself`, which specifies how `self` is rebound in the reflective invocation. This permits us to encode delegation. The handler can thus either rebind `self` to the proxy (delegation) or rebind `self` to the target (forwarding). Delegation proxies thus trivially subsume traditional forwarding proxies.

Delegation ensures that messages received by the proxy are traced, including self-sends in the method executed with delegation. However, it would fail to trace messages sent to other objects. The evaluation of `2 tracing fib` would print `2 fib, 2 < 2, 2 - 1, 2 - 2`, but all the messages sent to `1, 0, true, false` and `[ ↑ self ]` would not be traced.

To consistently apply a behavioral variation during the execution of a message send, all objects accessed during the execution must be represented with proxies.

3.2 Wrapping rules

To scope a behavioral variation to a dynamic extent, we can implement a handler that replaces all object references accessed by proxies. This way, the behavioral variation will *propagate* during the execution.

In a given method activation, a reference to an object can be obtained from:

- an argument,
- a field read,
- the return value of message sends,
- the instantiation of new objects
- the resolution of literals[6]

The following rules suffice to make sure that all objects are represented by proxies:

- *Wrap the initial arguments.* When the root proxy receives a message, the arguments are wrapped with proxies. We don't need to wrap the arguments of messages sent to non-root proxies because the other rules ensure that the arguments are already wrapped in the context of the caller.
- *Wrap field reads.* References to fields are represented by proxies.
- *Wrap object instantiation.* The return value of primitive message sends that "create" new objects must be wrapped. Such primitive messages include explicit instantiations with `new` and arithmetic computations that are typically implemented natively.

[6] We consider closures to be special kinds of literals.

– *Wrap literals.* Similarly, literals occurring in the method must be wrapped.

We don't need to wrap the return value of other message sends. Indeed, if the receiver and the arguments of a message send are already wrapped, and if the results of state reads and object instantiations are also wrapped in the execution of the method triggered by this message send, this message send will necessarily return a proxy.

Additionally, we need two rules to control how objects must be unwrapped:

– *Unwrap field writes.* When a field is written, we unwrap the value of the assignment before performing it. This way, the proxies created during the propagation are only referred to from within the call stack and don't corrupt the object graph connected to the target.
– *Unwrap the initial return value.* The root proxy unwraps the objects returned to the clients. This rule may be omitted to implement other forms of propagation as discussed in subsection 5.2.

Applying this technique to the code in Listing 1, the subtractions `self-2` and `self-1` return proxies as well. Figure 4 depicts the situation. This way, tracing is consistently applied during the computation of Fibonacci numbers.

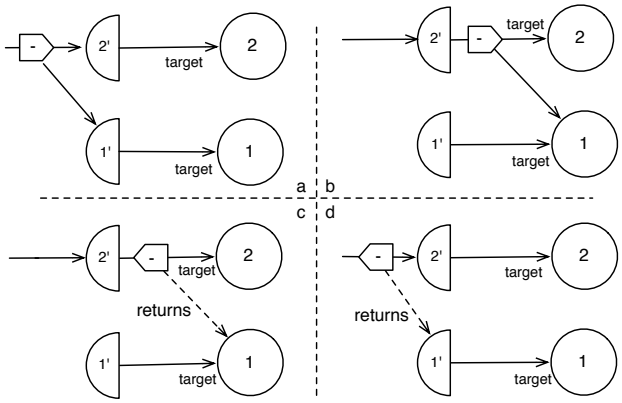

Fig. 4. Illustration of propagation during the subtraction `2 - 1`. A proxy to `2` receives the subtraction message "`-`" with a proxy to `1` as argument (a). The message is forwarded to `2` to perform the actual subtraction (b) that returns `1` (c). Finally the result is wrapped (d).

3.3 Propagation handler

To implement a handler that applies the wrapping rules presented previously, we need to have specific traps. In Smalltalk, an object is instantiated by sending

the message `new` to a class, which is an object as well. The interception of object instantiations does thus not require a specific trap and is realized indirectly. The following set of traps is thus sufficient to intercept all method invocations, state accesses, and object instantiations:

- `handleMessage:myself:target:`
  The trap for message sends takes as parameters the reified message, the original proxy[7] and the target.
- `handleReadField:myself:target:`
  The trap for field reads takes as parameters the field name, the original proxy and the target.
- `handleWriteField:value:myself:target:`
  The trap for field writes takes as parameters the field name, the value to write, the original proxy and the target.
- `handleLiteral:myself:target:`
  The trap for the resolution of literals (symbols, string, numbers, class names, and closures) takes as parameters the resolved literal, the original proxy and the target.

This set of traps is sufficient to implement propagation in Smalltalk. However, the set of necessary traps depends on the host language. In Java, additional traps would be needed to intercept constructor invocations, accesses to static fields and invocation of static methods.

Similarly to `perform:`, reflective methods used to read fields, to write fields and to resolve literals need to be extended with an additional parameter `myself`. They become `instVarNamed:myself:`, `instVarNamed:put:myself` and `literal:myself:`. In the case of a chain of proxies, the parameter `myself` is passed to the traps along the chain to preserve the identity of the proxy that originally intercepted the operation.

Listing 5 shows the traps for state writes and state reads traps. Other traps are similar. This handler just applies the previous wrapping rules. It is instantiated with another handler on which it will hand-over trap invocations. This allows propagation to be used with handlers that are not prepared for this technique.

It is impossible to deconstruct a proxy to obtain its handler or its target without using reflective capabilities. For simplicity, we assume the existence of a class `Reflect` that exposes the following methods globally:

- `Reflect class>>isProxy: aProxy`
  Returns whether the argument is a proxy or not.
- `Reflect class>>handlerOf: aProxy`
  If the argument is a proxy, returns its handler. Fails otherwise.
- `Reflect class>>targetOf: aProxy`
  If the argument is a proxy, returns its target. Fails otherwise.

[7] In case of chain proxies, the original proxy is not necessarily the one that intercepted the operation but the root of the chain.

For increased security, these methods could be stratified with *mirrors* (*i.e.*, dedicated objects that provide access to reflective features for a given object [BU04]), in which case handlers would need to have access to a mirror when they are instantiated.

```
PropagationHandler»handleReadField: f myself: p target: t
  ↑ self wrap: (h handleReadField: f myself: p target: t).

PropagationHandler»handleWriteField: f value: v proxy: p target: t
  h handleWriteField: f value: (self unwrap: v) proxy: p target: t.
  ↑ v
```

Listing 5. State reads and state writes traps of a propagating handler.

3.4 Closures

Closures deserve special treatment. A closure should be evaluated with the variations that are active in the current dynamic extent, and not the variations that were active when it was created. Consider, for instance, if the closure `[ self printString ]` is created when tracing is enabled, its evaluation during a regular execution should not trace the message `printString`. Conversely, if the closure `[ self printString ]` is created during a regular execution, its evaluation when tracing is enabled should trace the message `printString`. For this to work correctly, closures are always created in an *unproxied* form, and are transformed on demand when wrapped.

Variables captured in a closure are stored in indexed fields. Let us see first how creation works and illustrate it with the closure `[ self printString ]` and tracing:

1. The closure is created by the runtime and captures variables as-is. *Tracing example:* the closure captures `self`, which refers to a proxy.
2. The closure creation is intercepted by the *literal* trap of the creator. *Tracing example:* the closure is treated like other literals and thus proxied.
3. If the closure was proxied, the runtime invokes the *write* trap of the closure's proxy for all captured variables. *Tracing example:* the runtime invokes the *write* trap of the closure's proxy passing `0` as field index and the `self` proxy as value. The trap unproxies the value and reflectively invokes `instVarNamed:put:myself:` for field `0`. This overwrites the previous value in the closure with a reference to the base object.

Evaluation of closures follows the inverse scheme:

1. If the closure is evaluated via a proxy, the runtime invokes the *read* trap each time a captured variable is accessed. *Tracing example:* the runtime invokes the *read* trap of the closure's proxy passing `0` as field index. The trap reflectively invokes `instVarNamed:` for field `0` and wraps the result with a proxy. The message `printString` is sent to the proxy.

Note that this scheme is quite natural if we consider that closures could be encoded with regular objects, similarly to anonymous classes in Java. In that case, captured variables are effectively stored in synthetic fields initialized in the constructor. The instantiation of the anonymous class would trigger *write* traps, and evaluation would trigger *read* traps.

Adding method `valueWithHandler:` in `BlockClosure`, tracing `2 fib` can also be achieved with `[ 2 fib ] valueWithHandler: TracingHandler new` instead of `2 tracing fib`. Closures provide a convenient way to activate a behavioral variation in the dynamic extent of expression.

```
BlockClosure»valueWithHandler: aHandler
    ↑ (Proxy handler: aHandler target: self) value.
```

Listing 6. Convenience method to wrap and evaluate a closure

4 Examples

Delegation proxies subsume dynamic proxies and can be used to implement all classical examples of dynamic proxies such as lazy values, membranes, *etc.* We omit examples that can be found in the literature [VCM13,Eug06,MPBD+11].

We focus in this section on new examples enabled by delegation proxies. They all rely on the propagation technique presented earlier. We assume that the handlers are wrapped with a `PropagationHandler` that implements the propagation technique for reuse (see Listing 5). There are advantages to using delegation proxies to scope these behavioral variations to a dynamic extent. Behavioral variations can be composed: different parties can add their own variations without being aware of others already active for the same target. A behavioral variation is enabled only for the proxy: a variation is enabled for clients who possess a reference to the proxy while other clients may have a reference to the target or to another proxy implementing another behavioral variation. It is up to the creator of the proxy to decide whether to pass the proxy or the target.

4.1 Read-only Execution

Read-only execution [ADD+10] prevents mutation of state during evaluation. Read-only execution can dynamically guarantee that the evaluation of a given piece of code is either free of side effects or raises an error.

Classical proxies could restrict the interface of a given object to the subset of read-only methods. However, they would fail to enable read-only execution of arbitrary methods, or to guarantee that methods are deeply read-only. Read-only execution can be implemented trivially by wrapping a handler that fails upon state writes with a propagation handler.

```
ReadOnlyHandler»handleWriteField: f value: v myself: p target: t
  ReadOnlyError signal: 'Illegal write'.
```

Listing 7. Read-only handler

Thanks to proxy-based intercession, the target object is still available to trusted clients that can modify it. Only clients holding a reference to the proxy are affected by the read-only policy.

4.2 Object Versioning

To tolerate errors, developers implement recovery blocks that undo mutations and leave the objects in a consistent state [PLW09]. Typically, this entails cloning objects to obtain snapshots. Our propagation technique enables the implementation of object versioning concisely. Before any field is mutated, the handler shown below records the old value into a log using a reflective field read. The log can be used in recovery block, for instance to implement rollback. Similarly to the other examples that follow, we assume that the handler is wrapped with a propagation handler.

```
RecordingHandler»handleWriteField: f value: v myself: p target: t
  | oldValue |
  oldValue := t instVarNamed: f myself: p.
  log add: { t. f. oldValue }.
  t instVarNamed: f put: v myself: p.
  ↑ v
```

Listing 8. Recording handler

A convenience method can be added to enable recording with `[...] recordInLog: aLog`.

```
BlockClosure»recordInLog: aLog
  ↑ self valueWithHandler: (RecordingHandler log: aLog)
```

Listing 9. Enabling recording

The log can then be used to reflectively undo changes if needed.

```
aLog reverseDo: [ :change |
  change first instVarNamed: change second put: change third
]
```

Listing 10. Undoing changes

4.3 Dynamic Scoping

In most modern programming languages, variables are lexically scoped and can't be dynamically scoped. Dynamic scoping is sometimes desirable, for instance in web frameworks to access easily the ongoing request. Developers must in this case use alternatives like thread locals. It is for instance the strategy taken by Java Server Faces in the static method `getCurrentInstance()` of class `FacesContext`[8]).

[8] `http://www.webcitation.org/6FOF4DFab`

Dynamic scoping can be realized in Smalltalk using stack manipulation [Deu81] or by accessing the active process. Delegation proxies offer an additional approach to implement dynamic bindings by simply sharing a common (key,value) pair between handlers. If multiple dynamic bindings are defined, objects will be wrapped multiple times, once per binding. When a binding value must be retrieved, a utility method locates the handler corresponding to the request key, and returns the corresponding value:

```
ScopeUtils»valueOf: aKey for: aProxy
   | h p |
   p := aProxy.
   [ Reflect isProxy: p ] whileTrue: [
      h := Reflect handlerOf: p.
      ( h bindingKey == aKey ) ifTrue: [
         ↑ h bindingValue.
      ].
      p := Reflect targetOf: p.
   ].
   ↑ nil. "Not found"
```

Listing 11. Inspection of a chain of proxies

During the evaluation of a block, a dynamic variable can be bound with `[...] valueWith: #currentRequest value: aRequest` and accessed pervasively with `ScopeUtils valueOf: #currentRequest for: self`.

4.4 Profiling

Previous sections already illustrated delegation proxies using tracing. The same approach could be used to implement other interceptors like profiling or code contracts. The following handler implements profiling. It stores records of the different execution durations in an instance variable `tallies` for later analysis.

```
ProfilingHandler»initialize
    tallies := OrderedCollection new

ProfilingHandler»handleMessage: m myself: p target: t
  | start |
  start := Time now.
  [ ↑ t perform: m myself: p ]
    ensure: [
        | duration |
        duration := Time now - start.
        tallies add: {t. m. duration} ]
```

Listing 12. A simple profiling handler

5 Other forms of propagation

Since propagation is implemented reflectively, it can be customized in many ways. Delegation proxies provide flexible building blocks to implement various

forms of scopes, possibly blurring the line between static and dynamic scoping, similarly to Tanter's *scoping strategies* [TFD+09].

5.1 Filtering on package

Propagation can for instance be adapted to enable a behavioral variation only for instances of classes belonging to specific packages. This can be used to select which parts of an execution are subject to a behavioral variation such as tracing. It is especially useful for excluding kernel classes (string, dictionaries, arrays, *etc.*) and focusing instead on the classes of the analysed application.

To implement this form of scoping, it is possible to implement a filtering handler with a set of packages. This handler will wrap another handler and ask it to apply its behavioral variation only when the class of the target is declared in one of the packages of interest. Listing 13 shows the message trap of that filtering handler. A filtering handler is then wrapped into a propagating handler.

```
FilteringHandler»handleMessage: m myself: p target: t
  ↑ (packages includes: t class package)
    ifTrue: [ innerHandler handleMessage: m myself: p target: t ]
    ifFalse: [ t perform: m myself: t ]
```

Listing 13. Message trap of the filtering handler

5.2 Defensive proxies

Behavioral variations that are concerned with security can be used in two cases. The first case is when we want to apply the behavioral variation to protect the target from its clients. The second case is when we want to apply the behavioral variation to protect the clients from the target.

Protecting the target from the clients. With full propagation a read-only behavioral variation ensures that no state is modified from within the dynamic extent of messages received by the root proxy. We can customize propagation to relax the constraint imposed on clients. We can ensure that no state *of the object graph connected to the target* is modified from within the dynamic extent of messages received by the root proxy. To achieve that we can have an alternative propagation handler that does not wrap initial arguments, *i.e.*, the arguments of messages sent to the root proxy. The rationale is that the client necessarily has a reference to each of the objects it passes as arguments in the message to the root proxy. The client can therefore access these objects with the behavioral variation disabled in any case. In such scenarios, we also typically want initial returns to be wrapped so that objects returned by the target are still protected by the behavioral variation. The alternative propagation handler would not apply this unwrapping rule.

Protecting the clients from the target. There are other security-related behavioral variations that need to protect the clients from the target. This means that propagation should be enabled only for the arguments that are passed to the proxy. In that case, the proxy created to wrap the initial target (initial proxy) does not propagate by itself but it would wrap the arguments of messages it receives with proxies that do propagate. The return values of messages sent to that initial proxy would also typically be other proxies with the same wrapping rules to ensure that clients are also protected from these objects. This scenario is in fact a combination of a membrane [VCM10,TCD13] with our propagation technique. The initial proxy's handler would implement the wrapping rules of membranes. The membrane handler would be parameterized with two handlers: one that defines the inside-out policy and another that defines the outside-in policy [TCD13]. The inside-out handler would be an identity handler and the outside-in handler would be a propagating handler, itself parameterized with the handler that implements the behavioral variations used to protect the clients from the target.

6 Semantics

We formalize delegation proxies by extending SmalltalkLite [BDNW08], a lightweight calculus in the spirit of ClassicJava [FKF98] that captures the core execution semantics of a Smalltalk-like language and omits static types. We assume no prior knowledge of it. Our formalization simplifies three aspects of the semantics presented in the previous sections: it doesn't model first-class classes, literals or closures. Consequently, *literal* traps are not considered. Instead, we introduce a *new* trap that intercepts object instantiations.

The syntax of our extended calculus, SmalltalkProxy, is shown in Figure 5. The only addition to the original syntax is the new expression **proxy** $e\ e$.

$$
\begin{aligned}
P &= \mathit{defn}^* e \\
\mathit{defn} &= \textbf{class}\ c\ \textbf{extends}\ c\ \{\ f^*\, \mathit{meth}^*\ \} \\
\mathit{meth} &= m(x^*)\ \{\ e\ \} \\
e &= \textbf{new}\ c \ \mid\ x \ \mid\ \textbf{self} \ \mid\ \mathsf{nil} \ \mid\ f \ \mid\ f = e \\
&\mid\ e.m(e^*) \ \mid\ \textbf{super}.m(e^*) \ \mid\ \textbf{let}\ x = e\ \textbf{in}\ e \\
&\mid\ \textbf{proxy}\ \ e\ e
\end{aligned}
$$

Fig. 5. Syntax of SmalltalkProxy

During evaluation, the expressions of the program are annotated with the object and class context of the ongoing evaluation, since this information is missing from the static syntax. An annotated expression is called a *redex*. For instance, the super call $\textbf{super}.m(v^*)$ is decorated with its object and class into

$\mathbf{super}\langle c\rangle.m\langle o\rangle(v^*)$ before being interpreted; **self** is translated into the value of the corresponding object; message sends $o.m(v^*)$ are decorated with the current object context to keep track of the sender of the message. The rules for translating expressions into redexes are shown below.

$$
\begin{aligned}
o[\![\mathbf{new}\ c']\!]_c &= \mathbf{new}\langle o\rangle\ c' \quad \textit{(where o is fresh)}\\
o[\![x]\!]_c &= x\\
o[\![\mathbf{self}]\!]_c &= o\\
o[\![\mathsf{nil}]\!]_c &= \mathsf{nil}\\
o[\![f]\!]_c &= f\langle o\rangle\\
o[\![f = e]\!]_c &= f\langle o\rangle = o[\![e]\!]_c\\
o[\![e.m(e_i^*)]\!]_c &= o[\![e]\!]_c.m\langle o\rangle(o[\![e_i]\!]_c^*)\\
o[\![\mathbf{super}.m(e_i^*)]\!]_c &= \mathbf{super}\langle c\rangle.m\langle o\rangle(o[\![e_i]\!]_c^*)\\
o[\![\mathbf{let}\ x = e\ \mathbf{in}\ e']\!]_c &= \mathbf{let}\ x = o[\![e]\!]_c\ \mathbf{in}\ o[\![e']\!]_c\\
o[\![\mathbf{proxy}\ \ e\ e']\!]_c &= \mathbf{proxy}\ \ o[\![e]\!]_c\ o[\![e']\!]_c
\end{aligned}
$$

Fig. 6. Translating expressions to redexes

Redexes and their subredexes reduce to a value, which is either an address a, nil, or a proxy. A proxy has a handler h and a target t. A proxy is itself a value. Both h and t can be proxies as well. Redexes may be evaluated within an expression context E. An expression context corresponds to an redex with a hole that can be filled with another redex. For example, $E[expr]$ denotes an expression that contains the sub-expression $expr$.

$$
\begin{aligned}
\epsilon = {} & o \mid \mathbf{new}\langle o\rangle\ c \mid x \mid \mathbf{self} \mid \mathsf{nil} \mid f\langle o\rangle \mid f\langle o\rangle = \epsilon\\
& \mid \epsilon.m\langle o\rangle(\epsilon^*) \mid \mathbf{super}\langle c\rangle.m\langle o\rangle(\epsilon^*) \mid \mathbf{let}\ x = \epsilon\ \mathbf{in}\ \epsilon\\
& \mid \mathbf{proxy}\ \epsilon\ \epsilon\\
E = {} & [\ ] \mid f\langle o\rangle = E \mid E.m\langle o\rangle(\epsilon^*) \mid o.m\langle o\rangle(o^*\ E\ \epsilon^*)\\
& \mid \mathbf{super}\langle c\rangle.m\langle o\rangle(o^*\ E\ \epsilon^*) \mid \mathbf{let}\ x = E\ \mathbf{in}\ \epsilon\\
& \mid \mathbf{proxy}\ E\ \epsilon \mid \mathbf{proxy}\ o\ E\\
o = {} & a \mid \mathsf{nil} \mid \mathbf{proxy}\ o\ o
\end{aligned}
$$

Fig. 7. Redex syntax

Translation from the main expression to an initial redex is carried out by the function $o[\![e]\!]_c$ (see Figure 6). This binds fields to their enclosing object context and binds **self** to the value o of the receiver. The initial object context for a program is nil. (*i.e.*, there are no global fields accessible to the main expression).

$$
\begin{array}{ll}
P \vdash \langle E[\mathbf{new}\langle r\rangle\ c], \mathcal{H}\rangle & \\
\quad \hookrightarrow \langle E[a], \mathcal{H}[a \mapsto \langle c, \{f \mapsto \mathsf{nil} \mid \forall f, f \in^*_P c\}\rangle]\rangle & [\textit{new}] \\
\quad \text{where } a \notin \mathrm{dom}(\mathcal{H}) & \\
P \vdash \langle E[\mathbf{new}\langle \mathbf{proxy}\ h\ t\rangle\ c], \mathcal{H}\rangle & \\
\quad \hookrightarrow \langle E[h.\mathbf{newTrap}(\mathbf{new}\langle t\rangle c, \mathbf{proxy}\ h\ t)], \mathcal{H}\rangle & [\textit{new-proxy}] \\
P \vdash \langle E[f\langle a\rangle], \mathcal{H}\rangle & \\
\quad \hookrightarrow \langle E[o], \mathcal{H}\rangle & [\textit{get}] \\
\quad \text{where } \mathcal{H}(a) = \langle c, \mathcal{F}\rangle \text{ and } \mathcal{F}(f) = o & \\
P \vdash \langle E[f\langle \mathbf{proxy}\ h\ t\rangle], \mathcal{H}\rangle & \\
\quad \hookrightarrow \langle E[h.\mathbf{readTrap}(t, f, \mathbf{proxy}\ h\ t)], \mathcal{H}\rangle & [\textit{get-proxy}] \\
P \vdash \langle E[f\langle a\rangle = o], \mathcal{H}\rangle & \\
\quad \hookrightarrow \langle E[o], \mathcal{H}[a \mapsto \langle c, \mathcal{F}[f \mapsto o]\rangle]\rangle & [\textit{set}] \\
\quad \text{where } \mathcal{H}(a) = \langle c, \mathcal{F}\rangle & \\
P \vdash \langle E[f\langle \mathbf{proxy}\ h\ t\rangle = o], \mathcal{H}\rangle & \\
\quad \hookrightarrow \langle E[h.\mathbf{writeTrap}(t, f, o, \mathbf{proxy}\ h\ t)], \mathcal{H}\rangle & [\textit{set-proxy}] \\
P \vdash \langle E[a.m\langle s\rangle(o^*)], \mathcal{H}\rangle & \\
\quad \hookrightarrow \langle E[a[\![e[o^*/x^*]]\!]_{c'}], \mathcal{H}\rangle & [\textit{message}] \\
\quad \text{where } \mathcal{H}[a] = \langle c, \mathcal{F}\rangle \text{ and } \langle c, m, x^*, e\rangle \in^*_P c' & \\
P \vdash \langle E[(\mathbf{proxy}\ h\ t).m\langle s\rangle(o^*)], \mathcal{H}\rangle & \\
\quad \hookrightarrow \langle E[h.\mathbf{messageTrap}(t, m, o^*, \mathbf{proxy}\ h\ t)], \mathcal{H}\rangle & [\textit{message-proxy}] \\
P \vdash \langle E[\mathbf{super}\langle c\rangle.m\langle s\rangle(o^*)], \mathcal{H}\rangle & \\
\quad \hookrightarrow \langle E[s[\![e[o^*/x^*]]\!]_{c''}], \mathcal{H}\rangle & [\textit{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[\mathbf{let}\ x = o\ \mathbf{in}\ \epsilon], \mathcal{H}\rangle & \\
\quad \hookrightarrow \langle E[\epsilon[o/x]], \mathcal{H}\rangle & [\textit{let}]
\end{array}
$$

Fig. 8. Reductions for SmalltalkProxy

So if e is the main expression associated to a program P, then $\mathsf{nil}[\![e]\!]_{\mathsf{Object}}$ is the initial redex.

$P \vdash \langle \epsilon, \mathcal{H}\rangle \hookrightarrow \langle \epsilon', \mathcal{H}'\rangle$ means that we reduce an expression (redex) ϵ in the context of a (static) program P and an object heap $\mathcal{H}$ to a new expression ϵ' and (possibly) updated heap $\mathcal{H}'$. The heap consists of a set of mappings from addresses $a \in \mathrm{dom}(\mathcal{H})$ 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 heap is $\mathcal{H} = \{\}$.

The reductions are summarized in Figure 8. Predicate $\in^*_P$ is used for field lookup in a class ($f \in^*_P c$) and method lookup ($\langle c, m, x^*, e\rangle \in^*_P c'$, where c' is the class where the method was found in the hierarchy). Predicates $\leq_P$ and $\prec_P$ are used respectively for subclass and direct subclass relationships.

If the object context $\langle o\rangle$ of an instantiation with $\mathbf{new}\langle o\rangle\ c$ is a regular object (*i.e.*, not a proxy), the expression reduces to a fresh address a, bound in the

heap to an object whose class is c and whose fields are all nil(reduction [*new*]). If the object context of the instantiation is a proxy, the **newTrap** is invoked on the handler instead (reduction [*new-proxy*]). The trap takes the result of the instantiation **new**$\langle t \rangle$ c as parameter; it can take further action or return it as-is.

The object context $\langle o \rangle$ of field reads and field writes can be an object or a proxy. A local field read in the context of an object address reduces to the value of the field (reduction [*get*]). A local field read in the context of a proxy invokes the trap **readTrap** on the handler h (reduction [*get-proxy*]). A local field write in the context of an object simply updates the corresponding binding of the field in the heap (reduction [*set*]). A local field read in the context of a proxy invokes the trap **writeTrap** on the handler h (reduction [*set-proxy*]).

Messages can be sent to an object or to a proxy. When we send a message to an object, the corresponding method body e is looked up, starting from the class c of the receiver a. The method body is then evaluated in the context of the receiver, binding **self** to the address a. 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 (reduction [*message*]). When a message is sent to a proxy, the trap **messageTrap** is invoked on the handler. The object context $\langle s \rangle$ that decorates the message corresponds to the sender of the message. The trap takes as parameters the message and its arguments, and the initial receiver of the message **proxy** h t.

Super-sends are similar to regular message sends, except that the method lookup must start in the superclass of the class of the method in which the **super** send was declared. In the case of a super-send, the object context $\langle s \rangle$ corresponds to the sender of the message as well as the receiver. The object context is used to rebind **self** (reduction [*super*]). When we reduce the super-send, we must take care to pass on the class c'' of the method in which the super reference was found, since that method may make further super-sends. Finally, **let in** expressions simply represent local variable bindings (reduction [*let*]).

Errors occur if an expression does not reduce to an a or to nil. This may occur if a non-existent variable, field or method is referenced (for example, when sending any message to nil, or applying traps on a handler h that isn't suitable). 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.

6.1 Identity Proxy

The language requires the ability to reflectively apply operations for proxies to be useful. For simplicity, we extend the language with three additional non-stratified reflective primitives: **send**, **read**, and **write**. The semantics of these primitives is given in Figure 9.

All three primitives take a final argument *my* (shortcut for "myself") representing the object context that will be rebound. When applied to a proxy, the operations invoke the corresponding trap in a straightforward manner, passing *my* as is. When **read** or **write** are applied to an object address, the argument

$$
\begin{array}{ll}
P \vdash \langle E[a.\mathbf{send}(m, o^*, my)], \mathcal{H}\rangle & \\
\quad \hookrightarrow \langle E[my[\![e[o^*/x^*]]\!]_{c'}], \mathcal{H}\rangle & [\textit{reflect-message}] \\
\quad \text{where } \mathcal{H}[a] = \langle c, \mathcal{F}\rangle \text{ and } \langle c, m, x^*, e\rangle \in^*_P c' & \\
P \vdash \langle E[(\mathbf{proxy}\ h\ t).\mathbf{send}(m, o^*, my)], \mathcal{H}\rangle & \\
\quad \hookrightarrow \langle E[h.\mathbf{messageTrap}(t, m, o^*, my)], \mathcal{H}\rangle & [\textit{reflect-message-proxy}] \\
P \vdash \langle E[a.\mathbf{read}(f, my)], \mathcal{H}\rangle & \\
\quad \hookrightarrow \langle E[o], \mathcal{H}\rangle & [\textit{reflect-get}] \\
\quad \text{where } \mathcal{H}(a) = \langle c, \mathcal{F}\rangle \text{ and } \mathcal{F}(f) = o & \\
P \vdash \langle E[(\mathbf{proxy}\ h\ t).\mathbf{read}(f, my)], \mathcal{H}\rangle & \\
\quad \hookrightarrow \langle E[h.\mathbf{readTrap}(t, f, my)], \mathcal{H}\rangle & [\textit{reflect-get-proxy}] \\
P \vdash \langle E[a.\mathbf{write}(f, o, my)], \mathcal{H}\rangle & \\
\quad \hookrightarrow \langle E[o], \mathcal{H}[a \mapsto \langle c, \mathcal{F}[f \mapsto o]\rangle]\rangle & [\textit{reflect-set}] \\
\quad \text{where } \mathcal{H}(a) = \langle c, \mathcal{F}\rangle & \\
P \vdash \langle E[(\mathbf{proxy}\ h\ t).\mathbf{write}(f, o, my)], \mathcal{H}\rangle & \\
\quad \hookrightarrow \langle E[h.\mathbf{writeTrap}(t, f, o, my)], \mathcal{H}\rangle & [\textit{reflect-set-proxy}] \\
P \vdash \langle E[\mathbf{unproxy}\ (\mathbf{proxy}\ h\ t)], \mathcal{H}\rangle & \\
\quad \hookrightarrow \langle E[t], \mathcal{H}\rangle & [\textit{unproxy}] \\
P \vdash \langle E[\mathbf{unproxy}\ a], \mathcal{H}\rangle & \\
\quad \hookrightarrow \langle E[a], \mathcal{H}\rangle & [\textit{unproxy-object}]
\end{array}
$$

Fig. 9. Reflective facilities added to SmalltalkProxy

my is ignored. When **send** is applied to an object address, *my* defines how **self** will be rebound during the reflective invocation.

With these primitives, we can trivially define the identity handler idHandler that defines the following methods :

```
class idHandler extends Object {
    newTrap(t, my){ t }
    readTrap(t, f, my){ t.read(f, my) }
    writeTrap(t, f, o, my){ t.write(f, o, my) }
    messageTrap(t, m, o*, my){ t.send(m, o*, my) }}
```

6.2 Proof of Compositionality

Here we prove that proxy composition works as expected. That is, when the first proxy in a chain of proxies intercepts a message send, the message traps of the corresponding handlers are evaluated in order (given that these traps delegate the message to the corresponding targets). In addition, the operations applied during the execution of the matching method are also intercepted and consequently, the corresponding traps are also evaluated in order.

Theorem 1. *Let $p_1, p_2, \ldots, p_n$ be a proxy chain (*i.e.*, $p_n =$ **proxy** h_n t and $\forall i \in [1, n[$. $p_i =$ **proxy** h_i p_{i+1}) such that each handler h_i unconditionally delegates the intercepted operations on their respective target. Then, the behavioral variations implemented by each h_i compose,* i.e.*, (1) a message m intercepted by p_1 triggers the message trap of each handler h_i in order for $i \in [1, n]$ and (2) each consequent operations performed in the method that t associates with the message m, will be intercepted and will trigger the corresponding trap of each handler h_i in order for $i \in [1, n]$.*

Proof:

Lemma 1. *For all $i \in [1, n[$, invoking the message trap of a handler h_i with p bound to the my parameter triggers the message trap of each subsequent handler h_j in order for $j \in]i, n]$ with p still bound to the my parameter.*

Proof of Lemma 1 by reverse induction:
Basis: The proposition holds for $n - 1$.

$$\begin{array}{lll} & \langle h_{n-1}.\mathbf{messageTrap}(p_n, m, o^*, p), \mathcal{H}\rangle & \\ \hookrightarrow^* & \langle E[p_n.\mathbf{send}(m, o^*, p)], \mathcal{H}\rangle & \text{by hypothesis} \\ \hookrightarrow & \langle E[h_n.\mathbf{messageTrap}(t, m, o^*, p)], \mathcal{H}\rangle & \text{by } [\textit{message-proxy}] \end{array}$$

Inductive step: If the proposition holds for $i + 1 > 1$ then it holds for i.

$$\begin{array}{lll} & \langle h_i.\mathbf{messageTrap}(p_{i+1}, m, o^*, p), \mathcal{H}\rangle & \\ \hookrightarrow^* & \langle E[p_{i+1}.\mathbf{send}(m, o^*, p)], \mathcal{H}\rangle & \text{by hypothesis} \\ \hookrightarrow & \langle E[h_{i+1}.\mathbf{messageTrap}(target_{i+1}, m, o^*, p)], \mathcal{H}\rangle & \text{by } [\textit{message-proxy}] \\ & \text{where } target_{i+1} = t \text{ when } i = n - 1 \text{ and } p_{i+2} \text{ otherwise} & \end{array}$$

Hence the proposition holds for all $i \in [1, n[$. □

We know from the reduction rule [*message-proxy*] that a message sent to a proxy triggers the message trap of that proxy's handler, that is $\langle E[p_1.m(o^*)], \mathcal{H}\rangle$ reduces to $\langle E[h_1.\mathbf{messageTrap}(p_2, m, o^*, p_1)], \mathcal{H}\rangle$. Together with Lemma 1, we know that the proposition *(1)* of Theorem 1 holds.

Now, we need to prove the proposition *(2)* also holds in these conditions. First we know by hypothesis that $\langle E[h_n.\mathbf{messageTrap}(t, m, o^*, p_1)], \mathcal{H}\rangle$ will eventually reduce to $\langle E[t.\mathbf{send}(m, o^*, p_1)], \mathcal{H}\rangle$. Then, we know from the reduction rule [*reflect-message*] that this reduces to $\langle E[p_1[\![meth[o^*/x^*]]\!]_{c'}, \mathcal{H}\rangle$ where $\mathcal{H}[t] = \langle c, \mathcal{F}\rangle$ and $\langle c, m, x^*, meth\rangle \in^*_P c'$. In other words, the code $meth$ of the method that t associates with the message m is executed with p_1 as receiver.

This method can perform different operations: self or super sends, writes, reads.

In case of self-sends, similarly to the proposition (1) we know from Lemma 1 and from the reduction rule [*message-proxy*] that the proposition (2) holds.

Lets now consider the case of writes.

Lemma 2. *For all $i \in [1, n[$, invoking the write trap of a handler h_i with p_1 bound to the my parameter triggers the write trap of each subsequent handler h_j in order for $j \in]i, n]$ with p_1 still bound to the my parameter.*

The proof of Lemma 2 is similar to that of Lemma 1. From [*set-proxy*] we see that $\langle E[f\langle p_1\rangle = o], \mathcal{H}\rangle$ gives $\langle E[h_1.\mathbf{writeTrap}(p_2, f, o, p_1)], \mathcal{H}\rangle$. Together with Lemma 2 and the fact that p_1 is the receiver during the evaluation of *meth*, we know that proposition *(2)* holds for writes. The situation is similar for reads, super-send and instantiation. We omit these cases for the sake of conciseness as the proof can easily be expended to take them into account. Hence the proposition *(2)* holds and so does the Theorem 1. □

6.3 Propagating Identity Proxy

Following the technique of propagation presented in Section 3, we propose a propagating identity handler, PropHandler. This handler defines the behavior of the root proxy and uses another handler PropHandler* to create the other proxies during the propagation. This technique requires the ability to unwrap a proxy. The expression **unproxy** is added to the language as defined in Figure 9. We also assume the existence of the traditional sequencing operation (*;*). We allow ourselves to use a notation for the multiple arguments of **messageTrap** that deviates from the given syntax and semantics.

The handler PropHandler is defined as follows:

```
class PropHandler extends Object {
    newTrap(t, my){ proxy PropHandler* t }
    readTrap(t, f, my){ t.read(f, my) }
    writeTrap(t, f, o, my){ t.write(f, o, my) }
    messageTrap(t, m, (o1, ..., on), my){
        unproxy(t.send(m,
            (proxy PropHandler* o1, ..., proxy PropHandler* on), my)) }}
```

The handler PropHandler* is defined as follows:

```
class PropHandler* extends PropHandler {
    messageTrap(t, m, (o1, ..., on), my){ t.send(m, (o1, ..., on), my) }}
```

6.4 Proof of Soundness of Propagation

We can formally express the intuitive explanation of Section 3 about soundness of the propagation.

Theorem 2. *Lets t, o_1, ..., o_n be objects. Then, during the evaluation of the expression $Expr := (\textbf{proxy}\ \mathsf{PropHandler}\ t).m(o_1, \ldots, o_n)$ everything reduces to a proxy or to* nil *except $Expr$ itself.*

Proof:

Let $pr = \textbf{proxy}\ \mathsf{PropHandler}\ t$. We begin by reducing the expression $pr.m(o^*)$ in the context of a program P where $o^* = (o_1, \ldots, o_n)$.

$$
\begin{array}{ll}
 & \langle pr.m(o_1,\ldots,o_n),\mathcal{H}\rangle \\
\hookrightarrow & \langle \mathsf{PropHandler}.\mathbf{messageTrap}(t,m,(o_1,\ldots,o_n),p),\mathcal{H}\rangle \quad \text{by } [\textit{message-proxy}] \\
\hookrightarrow^* & \langle \mathbf{unproxy}(t.\mathbf{send}(m,po^*,p)),\mathcal{H}\rangle \\
 & \text{where } po^* = ((\mathbf{proxy}\ \mathsf{PropHandler}^* o_1),\ldots,(\mathbf{proxy}\ \mathsf{PropHandler}^* o_n)) \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{by definition} \\
\hookrightarrow & \langle \mathbf{unproxy}(pr[\![e[po^*/x^*)]\!]_{c'},\mathcal{H}\rangle \\
 & \text{where } \mathcal{H}[t] = \langle c,\mathcal{F}\rangle \text{ and } \langle c',m,x^*,e\rangle \in^*_P c \qquad \text{by } [\textit{reflect-message}]
\end{array}
$$

Now we prove by structural induction on e that during the evaluation of $\langle pr[\![e[po^*/x^*]]\!]_{c'},\mathcal{H}\rangle$ everything reduces to a proxy or to nil. Then, the remaining **unproxy** operation ensures that the value of the initial expression is unwrapped and thus that Theorem 2 always holds.

We call *(H)* the inductive hypothesis: *during the evaluation of e everything reduces to a proxy or to nil.*

Case $e := x$: Here x can refer to an argument or to a local variable (see [*let*]). In the first case we have $\langle pr[\![x[po^*/x^*]]\!]_{c'},\mathcal{H}\rangle = \langle p_{o_i},\mathcal{H}\rangle$ where i is the index of the argument in question. In the second case we know by inductive hypothesis that it reduce to a proxy or nil.

Case $e := \mathbf{self}$: $\langle pr[\![\mathbf{self}[po^*/x^*]]\!]_{c'},\mathcal{H}\rangle = \langle pr,\mathcal{H}\rangle$

Case $e := \mathsf{nil}$: $\langle pr[\![\mathsf{nil}[po^*/x^*]]\!]_{c'},\mathcal{H}\rangle = \langle \mathsf{nil},\mathcal{H}\rangle$

Case $e := \mathbf{new}\ c$:

$$
\begin{array}{lll}
 & \langle pr[\![\mathbf{new}\ c[po^*/x^*]]\!]_{c'},\mathcal{H}\rangle = \langle \mathbf{new}\langle pr\rangle\ c,\mathcal{H}\rangle & \\
\hookrightarrow & \langle \mathsf{PropHandler}.\mathbf{newTrap}(\mathbf{new}\langle t\rangle c,pr),\mathcal{H}\rangle & \text{by } [\textit{new-proxy}] \\
\hookrightarrow & \langle \mathsf{PropHandler}.\mathbf{newTrap}(a,pr),\mathcal{H}'\rangle & \\
 & \text{where } \mathcal{H}' = \mathcal{H}[a \mapsto \langle c,\{f \mapsto \mathsf{nil} \mid \forall f, f \in^*_P c\}\rangle] & \text{by } [\textit{new}] \\
\hookrightarrow^* & \langle \mathbf{proxy}\ \mathsf{PropHandler}^*\ a,\mathcal{H}'] \rangle & \text{by definition}
\end{array}
$$

Case $e := f$:

$$
\begin{array}{lll}
 & \langle pr[\![f[po^*/x^*]]\!]_{c'},\mathcal{H}\rangle = \langle f\langle pr\rangle,\mathcal{H}\rangle & \\
\hookrightarrow & \langle \mathsf{PropHandler}.\mathbf{readTrap}(t,f,pr),\mathcal{H}\rangle & \text{by } [\textit{get-proxy}] \\
\hookrightarrow^* & \langle \mathbf{proxy}\ \mathsf{PropHandler}^*\ (t.\mathbf{read}(f,p)),\mathcal{H}\rangle & \text{by definition} \\
\hookrightarrow & \langle \mathbf{proxy}\ \mathsf{PropHandler}^*\ o,\mathcal{H}\rangle & \text{by } [\textit{reflect-get}]
\end{array}
$$

Case $e := f = e'$:

We know by inductive hypothesis that e' reduces to a value which is a proxy or nil. Let v be that value.

$$
\begin{array}{lll}
 & \langle pr[\![f = e'[po^*/x^*]]\!]_{c'}, \mathcal{H}\rangle = \langle f\langle pr\rangle = e', \mathcal{H}\rangle & \\
\hookrightarrow^* & \langle f\langle pr\rangle = v, \mathcal{H}'\rangle & \text{by } \textit{(H)} \\
\hookrightarrow & \langle \mathsf{PropHandler}.\mathbf{writeTrap}(t, f, v, p), \mathcal{H}'\rangle & \text{by } [\textit{set-proxy}] \\
\hookrightarrow^* & \langle t.\mathbf{write}(f, \mathbf{unproxy}\ v, p); v, \mathcal{H}'\rangle & \text{by definition} \\
\hookrightarrow & \langle t.\mathbf{write}(f, v', p); v, \mathcal{H}'\rangle & \text{by } [\textit{unproxy(-object)}] \\
 & \text{where } v' = t' \text{ if } v = \mathbf{proxy}\ h\ t' \text{ or nil if } v = \mathsf{nil} & \\
\hookrightarrow & \langle v'; p', \mathcal{H}''\rangle & \\
 & \text{where } \mathcal{H}'' = \mathcal{H}'[t \mapsto \langle c, \mathcal{F}[f \mapsto o]\rangle] & \\
 & \text{and } \mathcal{H}'(t) = \langle c, \mathcal{F}\rangle & \text{by } [\textit{reflect-set}] \\
= & \langle v, \mathcal{H}''\rangle &
\end{array}
$$

Case $e := e'.m(e''_1, \ldots, e''_n)$:

We know by *(H)* that e', e''_1, ..., e''_n all reduce to a proxy or nil. We need to prove that *(H)* holds in case e' is a proxy p whose handler is a $\mathsf{PropHandler}^*$. Since only **messageTrap** differs from PropHandler in $\mathsf{PropHandler}^*$, we thus just need to prove that the expression $p.m(p_1, \ldots, p_n)$ reduces to a proxy or nil when each p_i are already proxies or nil.

$$
\begin{array}{lll}
 & \langle p.m(p_1, \ldots, p_n), \mathcal{H}\rangle & \\
\hookrightarrow & \langle \mathsf{PropHandler}^*.\mathbf{messageTrap}(t', m, (p_1, \ldots, p_n), p), \mathcal{H}\rangle & \text{by } [\textit{message-proxy}] \\
\hookrightarrow & \langle t'.\mathbf{send}(m, (p_1, \ldots, p_n), p), \mathcal{H}\rangle & \text{by definition} \\
\hookrightarrow & \langle p[\![meth[(p_1, \ldots, p_n)/x^*]]\!]_{c_m}, \mathcal{H}\rangle & \\
 & \text{where } \mathcal{H}[t'] = \langle c_{t'}, \mathcal{F}\rangle \text{ and } \langle c_m, m, x^*, meth\rangle \in^*_P c_m & \text{by } [\textit{reflect-message}]
\end{array}
$$

That last expression is known to reduce to a proxy or nil by *(H)*.

Case $e := \mathbf{super}.m(e'_1, \ldots, e'_n)$: Similar to the previous case.

Case $e := \mathbf{let}\ x = e'\ \mathbf{in}\ e''$: We know by *(H)* that e' reduces to a proxy or nil. In case x is referred in e'', it will be replaced by e' (see case $e := x$).

Case $e := \mathbf{proxy}\ e'\ e''$: This case already satisfies the proposition. □

7 Implementation

We have implemented a prototype of delegation proxies in Pharo[9], a Smalltalk dialect. This implementation generates code using the compiler infrastructure of Pharo. For each existing method in a base class, a hidden method with an additional parameter `myself` and a transformed body is generated and installed in a *proxy class*. Instead of `self`, `myself` is used in the generated method body (this is similar to Python's explicit `self` argument). Following the same approach as Uniform Proxies for Java [Eug06], proxy classes are auto-generated.

7.1 Example

Let us consider the class `Suitcase`:

[9] `http://pharo.org/`

```
Object»subclass: #Suitcase
   instanceVariableNames: 'content'

Suitcase»printString
  ↑ 'Content: ' concat: content
```

Listing 14. Original code of class `Suitcase`

Applying our transformation, the class `Suitcase` is augmented with synthetic methods to read and write the field `content` and to resolve literals.

```
Suitcase»literal: aLiteral myself: myself
  ↑ aLiteral

Suitcase»readContentMyself: myself
  ↑ content

 Suitcase»writeContent: value myself: myself
  ↑ content := value
```

Listing 15. Synthetic methods to read and write instance variable `content` and literal resolution

In Smalltalk, fields are encapsulated and can be accessed only by their respective object. The sender of a state access is always `myself`, and can thus be omitted from the traps. For each existing method in class `Suitcase`, a hidden method with a transformed body and one additional parameter `myself` is generated.

```
Suitcase»printStringMyself: myself
  ↑ (myself literal: 'Content: ' myself: myself)
     concat: (self readContentMyself: myself)
```

Listing 16. A transformed version of method `printString`

A proxy class for `Suitcase` is then generated. It inherits from a class `Proxy`, which defines the `handler` field common to all proxies. The generated class implements the same methods as the `Suitcase` class, *i.e.*, , `printStringMyself:`, `readContentMyself:`, and `writeContent:myself:`. The methods invoke respectively *message*, *read* and *write* traps on the handler. In addition, a method `printString` forwards to `printStringMyself:` with `self` as argument.

```
SuitcaseProxy»printString
  ↑ self printStringMyself: self

SuitcaseProxy»printStringMyself: myself
  | msg |
  msg := Message selector: #printString arguments: {}.
  ↑ handler message: msg myself: myself target: target
```

Listing 17. Sample generated method in proxy class of `Suitcase`

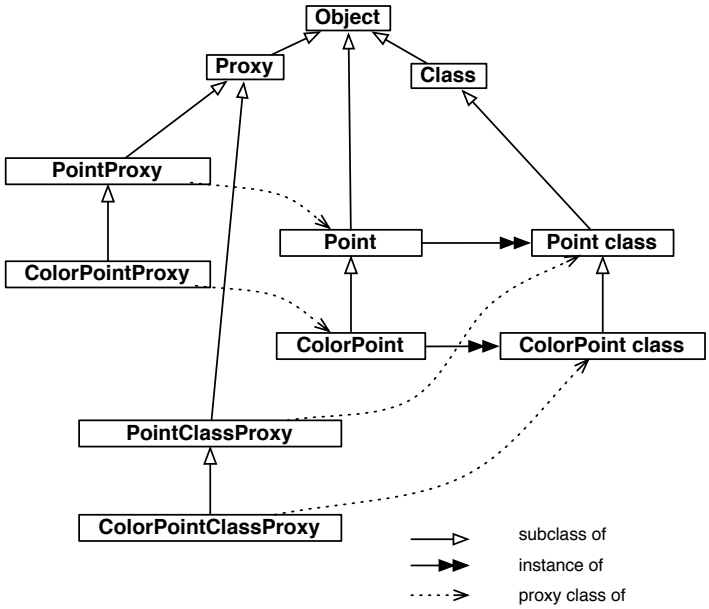

Fig. 10. Inheritance of classes, meta-classes, and auto-generated proxy classes.

Proxy class generation. Smalltalk has first-class classes whose behaviors are defined in meta-classes. The meta-class hierarchy parallels the class hierarchy. Classes can be proxied like any object. Consequently, meta-classes are rewritten and extended with synthetic methods similarly to classes. However, the generated proxy classes do not inherit from `Class`, but `Proxy`, as shown in Figure 10.

Handling closures. Closures are regular objects that are adapted upon creation and evaluation as described in subsection 3.4. If a closure defined in an original uninstrumented method is wrapped with a proxy, the code of the closure is transformed lazily when the closure is evaluated.

Weaving. Sending a message to a proxy entails reification of the message, invocation of the handler's trap, and then reflective invocation of the message on the target. In addition, the handler might take additional actions that entail costs. The handler and the proxy can be woven into specialized classes for fewer levels of indirection. For instance, a `SuitcaseProxy` with a `Tracing` handler can be woven into a `SuitcaseTracingProxy`:

```
SuitcaseTracingProxy»printStringMyself: myself
  | msg |
  msg := Message selector: #printString arguments: {}.
  Transcript
```

```
  print: target asString;
  space;
  print: msg asString;
  cr.
↑ target printStringMyself: myself
```

Listing 18. Sample woven method

We have implemented a simple weaver that works for basic cases. We plan to mature it in the future and leverage techniques for partial evaluation [Fut99] developed for aspect compilers [MKD03].

7.2 Performance

Delegation proxies have no impact on performance when they are not used: the transformation adds new code but does not alter existing code. Only the proxies pay a performance overhead. We need to distinguish between the performance of delegation proxies themselves and the performance of the propagation technique.

Used sparingly, delegation proxies do not entail serious performance issues because the cost of an interception is reasonably low. To measure the cost of a message interception, we use a proxy with an identity handler (*i.e.*, a handler that performs the intercepted operations reflectively) and send a message that dispatches to an empty method. The result is compared with sending the same message to the target object. The average performance degradation is 20.49 over 1 million iterations. With weaving, this number drops to 1.45 because there is less indirections and no reflective call is needed anymore (see the last line of Listing 18 for example).

However, with propagation, all the operations that happen in the dynamic extent of a message send are intercepted. Consequently, the cost of delegation proxies with propagation becomes prohibitive unless weaving is used. With weaving, computing the 20th Fibonacci number 10 thousands times reveals an average performance degradation of 8.72.

We believe these are encouraging results given that delegation proxies enable unanticipated behavioral reflection, which is known to be costly. Also the performance could be improved if delegation proxies were implemented directly by the runtime.

8 Discussion

Myself parameter. To encode the fact that delegation proxies operate by delegation, we add an explicit parameter `myself` to all the available reflective operations. While this exposes the rebinding of self references for the sake of understandability, this extra parameter can be hidden from the users. The reifications of the operations can handle the delegation themselves. Otherwise, delegation can be implemented natively in which case delegation becomes mandatory. It is thus not absolutely required to modify the reflective API of a language to be able to implement delegation proxies to support delegation.

Usability. From the developer's point of view, proxies operating by delegation or by forwarding are not very different. If the `myself` parameter is exposed, the developer has to take care to provide the right receiver. Similarly, if a handler sends additional messages in its traps, it may have to unwrap the receivers if they are proxies. Another consequence for a developer is that delegation proxies permit behavioral variations to be composed together dynamically by forming a chain of proxies.

Static Typing. There is no major obstacle to port our implementation to a statically-typed language. Delegation proxies preserve the interface of their target, like traditional forwarding proxies. For type compatibility, the generated proxy class must inherit from the original class. Reflective operations and traps can fail with run-time type errors. Forwarding and delegation proxies suffer from the same lack of type safety from this perspective.

If closures cannot be adapted at run time with the same flexibility as in Smalltalk, the implementation might require a global rewrite of the sources to adapt the code of the closures at compile-time.

Our implementation of delegation proxies requires that reflective operations have an additional parameter `myself` that specifies how to rebind `self`. Naturally, this parameter must be of a valid type: in practice it will be either the target of the invocation or a proxy of the target. Both implement the same interface.

9 Related Work

Proxy-based intercession. Many languages provide support for dynamic proxies. When a message is sent to a dynamic proxy, the message is intercepted and reified. Dynamic proxies have found many usefully applications that can be categorized as *interceptors* or *virtual objects* [VCM10]. An important question for proxies is whether to support them at the language level or natively at the runtime level. The performance gain brought by native support of proxies is appreciable for serious use in production but native implementations generally lack the flexibility needed for experimentation.

Most dynamic languages support proxies via traps that are invoked when a message cannot be delivered [MPBD$^+$11]. Modern proxy mechanisms stratify the base and meta levels with a handler [MPBD$^+$11,Eug06,VCM10,STHFF12]. These solutions are generally limited to message intercession. However, many interpretation operations — such access to instance variables and literal resolution — are not typically realized via message-sending. It is important to be able to intercept these operations to implement useful behavioral variations. The inability to intercept these operations make the implementation of dynamically-scoped behavioral variations difficult.

A notable exception is that of EcmaScript 6 direct proxies. EcmaScript direct proxies operate by delegation [VCM10] and can intercept additional operations. However, they do not enable the interception of object instantiations. Also, the

variables captured in a closure cannot be intercepted upon reads and writes. This makes the form of propagation presented here is not easily implementable with EcmaScript 6 direct proxies as captured variables cannot easily be unwrapped upon capture and wrapped upon evaluation.

Pointcut-advice model. Proxy-based intercession differs from the traditional poincut-advice model of aspect oriented programming. In the pointcut-advice model, an aspect groups definition of pointcuts with corresponding advices, *i.e.*, a behavioral variation and its deployment. With proxy-based intercession, a handler specifies the actions taken upon interception of certain operations. This looks similar to the pointcut-advice model: the body of a trap is akin to an advice and the trap itself matches certain points of execution, just like a pointcut does. However, a trap does not specify which objects it will affect. A trap defines points of interception relatively to a proxy: only operations applied on a proxy are intercepted and trigger the actions defined by the corresponding traps. This allows developpers to deploy a behavioral variation on specific object references.

Composition Filters. In the *Composition Filters* model [AWB+94], objects filter and transform incoming and outgoing messages. This model could be used to implement dynamic scoping similar to the presented propagation technique. With composition filters, an object can rewrite outgoing messages to change their receiver to an object with that same behavior. Delegation could be implemented with a *dispatch* filter. However, as far as we are aware of, the composition filters model does not offer a mechanism to intercept instance variable accesses. Intercepting instance variables accesses is what permits delegation proxies to implement clean propagation (leaving the target objects unaffected by unwrapping proxies on writes). Adding this facility to the model would allow composition filters to realize the kind of dynamic scoping enabled by the propagation technique presented here.

Infinite regression. AOP and MOP inherently suffer from infinite regression issues unless the meta-levels are explicitly modeled [CKL96,DSD08,Tan10]. In contrast to AOP and MOPs, delegation proxies limit infinite regression issues since the adapted object and the base object are distinct. For instance, the tracing handler in Listing 4 does not lead to an infinite regression since it sends the message `asString` to the target, which is distinct from the proxy (in parameter `myself`). Handlers that send messages only to non-proxy objects do not lead to meta-regressions, but failure to do so can lead to meta-regressions, possibly infinite. Consequently, handlers may have to unwrap proxies before sending messages in traps. Also, delegation proxies naturally enable partial reflection [TNCC03] since objects are selectively wrapped and proxies can be selectively passed around to client code.

Composing Behavior. Inheritance leads to an explosion in the number of classes when multiple variations of a given set of classes must be designed. Static traits

[SDNB03] or mixins enable the definition of units of reuse that can be composed into classes, but they do not solve the issue of class explosion.

One solution to this problem is the use of decorators that refine a specific set of known methods, *e.g.*, the method `paint` of a window. Static and dynamic approaches have been proposed to decoration. Unlike decorators, proxies find their use when the refinement applies to unknown methods, *e.g.*, to trace all invocations. Büchi and Weck proposed a mechanism [BW00] to statically parameterize classes with a decorator (called wrapper in their terminology). Bettini *et al.* [BCG07] proposed a similar construct but composition happens at creation time. Ressia *et al.* proposed *talents* [RGN+12] which enable adaptations of the behavior of individual objects by composing trait-like units of behavior dynamically. Other works enable dynamic replacement of behavior in a trait-like fashion [BCD13].

The code snippet below illustrates how to achieve the decoration of a `Window` with a `Border` and shows the conceptual differences between these approaches. The two first approaches can work with forwarding or delegation (but no implementations with delegation are available). The third approach replaces the behavior or the object so the distinction does not apply.

```
Window w = new Window<Border>(); // Buchi and Weck
Window w = new BorderWrap( new Window() ); // Bettini
Window w = new WindowEmptyPaint(); // Ressia
w.acquire( new BorderedPaint() );
```

Listing 19. Differences between approaches to decoration

Several languages that combine class-based inheritance and object inheritance (*i.e.*, delegation) have been proposed [Kni99,VTB98]. Delegation enables the behavior of an object to be composed dynamically from other objects with partial behaviors. Essentially, delegation achieves trait-like dynamic composition of behavior.

Ostermann proposed delegation layers [Ost02], which extend the notion of delegation from objects to collaborations of nested objects, *e.g.*, a graph with edges and nodes. An outer object wrapped with a delegation layer will affects its nested objects as well. Similary to decorators, the mechanism refines specific sets of methods of the objects in the collaboration.

Dynamic Scoping. The dynamic extent of an expression corresponds to all operations that happen during the evaluation of the expression by a given thread of execution. Control-flow pointcuts are thus not sufficient to scope to dynamic extents, since they lack control over the thread scope. Control-flow pointcuts are popular and supported by mainstream AOP implementations, *e.g.*, AspectJ's `cflow` and `cflowbelow`. Aware of the limitations of control-flow pointcuts, some AOP implementations provide specific constructs to scope to the dynamic extent of a block of code, *e.g.*, CaesarJ's `deploy` [AGMO06]. Implemented naively, control-flow pointcuts are expensive since they entail a traversal of the stack at run time, but they can be implemented efficiently using partial evaluation [MKD03].

In context-oriented programming (COP) [HCN08,vLDN07], variations can be encapsulated into layers that are dynamically activated in the dynamic extent of an expression. Unlike the propagation technique presented in this paper that work better with *homogenous* variations, COP has a better support for *heterogenous* variations [ALS08]. COP can be seen as a form of multi-dimensional dispatch, where the context is an additional dimension.

Other mechanisms to vary the behavior of objects in a contextual manner are roles [Kri96], perspectives [SU96], and subjects [HO93]. Delegation proxies can realize dynamic scoping via reference flow, by wrapping and unwrapping objects accessed during an execution. Delegation proxies may provide a foundation to design contextual variations.

Similarly to our approach, Arnaud *et al.*'s *Handle* model [ADD$^+$10,Arn13] enables the adaptation of references with behavioral variations that propagate. The propagation belongs to the semantics of the handles, whereas in our approach, the propagation is encoded reflectively in a specific handler. Our approach is more flexible since it decouples the notion of propagation from the notion of proxy but the handle approach is more efficient since it is implemented at the runtime level.

10 Conclusions

We can draw the following conclusions about the applicability of delegation proxies:

- *Expressiveness.* Delegation proxies subsume forwarding proxies and enable variations to be propagated to dynamic extents. This suits well non-functional concerns like monitoring (tracing, profiling), safety (read-only references), or reliability (rollback with object versioning). Since the propagation is written reflectively, it can be customized to achieve other forms of scopes.
- *Metaness.* Delegation proxies naturally compose, support partial behavioral reflection, and help developers avoid meta-regression. We can for instance trace and profile an execution by using tracing proxies and profiling proxies that form chains of delegation (composition). Objects are wrapped selectively. Adapting objects during an execution will not affect other objects in the system (partial reflection). Proxies and targets represent the same object at two different levels but have distinct identities (no meta-regression).
- *Encoding.* Delegation proxies can be implemented with code generation. In our Smalltalk implementation, only new code needs to be added; existing code remains unchanged. Delegation proxies have thus no overhead if not used. Delegation proxies do not entail performance issues when used sporadically (same situation as with classical dynamic proxies). The overhead of our propagation technique is of factor 8 when the code of handlers are woven into dedicated proxies classes. For optimal performance, the language could provide native support of delegation proxies.

Acknowledgments

We thank Jorge Ressia, Mircea Lungu, Niko Schwarz and Jan Kurš for support and feedback about ideas in the paper. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Agile Software Assessment" (SNSF project Np. 200020-144126/1, Jan 1, 2013 - Dec. 30, 2015) and of the French General Directorate for Armament (DGA).

References

ADD+10. Jean-Baptiste Arnaud, Marcus Denker, Stéphane Ducasse, Damien Pollet, Alexandre Bergel, and Mathieu Suen. Read-only execution for dynamic languages. In *Proceedings of the 48th International Conference Objects, Models, Components, Patterns (TOOLS'10)*, Malaga, Spain, June 2010.

AGMO06. Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An overview of CaesarJ. *Transactions on Aspect-Oriented Software Development*, 3880:135 – 173, 2006.

ALS08. Sven Apel, Thomas Leich, and Gunter Saake. Aspectual feature modules. *Software Engineering, IEEE Transactions on*, 34(2):162–180, 2008.

Arn13. Jean-Baptiste Arnaud. *Towards First Class References as a Security Infrastructure in Dynamically-Typed Languages*. PhD thesis, Université de Lille, 2013.

AWB+94. Mehmet Aksit, Ken Wakita, Jan Bosch, Lodewijk Bergmans, and Akinori Yonezawa. Abstracting object interactions using composition filters. In Rachid Guerraoui, Oscar Nierstrasz, and Michel Riveill, editors, *Proceedings of the ECOOP '93 Workshop on Object-Based Distributed Programming*, volume 791 of *LNCS*, pages 152–184. Springer-Verlag, 1994.

BCD13. Lorenzo Bettini, Sara Capecchi, and Ferruccio Damiani. On flexible dynamic trait replacement for Java-like languages. *Science of Computer Programming*, 78(7):907–932, 2013.

BCG07. Lorenzo Bettini, Sara Capecchi, and Elena Giachino. Featherweight wrap Java. In *Proceedings of the 2007 ACM symposium on Applied computing*, pages 1094–1100. ACM, 2007.

BDNW08. Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Stateful traits and their formalization. *Journal of Computer Languages, Systems and Structures*, 34(2-3):83–108, 2008.

BU04. 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.

BW00. Martin Büchi and Wolfgang Weck. Generic wrappers. In *ECOOP 2000—Object-Oriented Programming*, pages 201–225. Springer, 2000.

CKL96. Shigeru Chiba, Gregor Kiczales, and John Lamping. Avoiding confusion in metacircularity: The meta-helix. In Kokichi Futatsugi and Satoshi Matsuoka, editors, *Proceedings of ISOTAS '96*, volume 1049 of *Lecture Notes in Computer Science*, pages 157–172. Springer, 1996.

Deu81. L Peter Deutsch. Building control structures in the Smalltalk-80 system. *Special Issues on Smailtalk-80, BYTE*, 6(8), 1981.

DSD08. 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.

Eug06. Patrick Eugster. Uniform proxies for Java. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 139–152, New York, NY, USA, 2006. ACM.

FKF98. Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 171–183. ACM Press, 1998.

Fut99. Yoshihiko Futamura. Partial evaluation of computation process: An approach to a compiler-compiler. *Higher Order Symbol. Comput.*, 12(4):381–391, 1999.

HCN08. Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3), March 2008.

HO93. 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.

Kni99. Günter Kniesel. Type-safe delegation for run-time component adaptation. In R. Guerraoui, editor, *Proceedings ECOOP '99*, volume 1628 of *LNCS*, pages 351–366, Lisbon, Portugal, June 1999. Springer-Verlag.

Kri96. Bent Bruun Kristensen. *Object-oriented modeling with roles.* Springer, 1996.

Lie86. Henry Lieberman. Using prototypical objects to implement shared behavior in object oriented systems. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, volume 21, pages 214–223, November 1986.

MKD03. H. Masuhara, G. Kiczales, and C. Dutchyn. A compilation and optimization model for aspect-oriented programs. In Görel Hedin, editor, *Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 46–60. Springer Berlin / Heidelberg, 2003.

MPBD+11. Mariano Martinez Peck, Noury Bouraqadi, Marcus Denker, Stéphane Ducasse, and Luc Fabresse. Efficient proxies in Smalltalk. In *Proceedings of ESUG International Workshop on Smalltalk Technologies (IWST'11)*, Edinburgh, Scotland, 2011.

Ost02. Klaus Ostermann. Dynamically composable collaborations with delegation layers. In *ECOOP 2002—Object-Oriented Programming*, pages 89–110. Springer, 2002.

Pas86. Geoffrey A. Pascoe. Encapsulators: A new software paradigm in Smalltalk-80. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, volume 21, pages 341–346, November 1986.

PLW09. Frédéric Pluquet, Stefan Langerman, and Roel Wuyts. Executing code in the past: efficient in-memory object graph versioning. In *ACM SIGPLAN Notices*, volume 44, pages 391–408. ACM, 2009.

RGN+12. Jorge Ressia, Tudor Gîrba, Oscar Nierstrasz, Fabrizio Perin, and Lukas Renggli. Talents: an environment for dynamically composing units of reuse. *Software: Practice and Experience*, 2012.

SDNB03. Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits: Composable units of behavior. In *Proceedings of European Conference on Object-Oriented Programming*, volume 2743 of *ECOOP'03*, pages 248–274. Springer Verlag, July 2003.

STHFF12. T Stephen Strickland, Sam Tobin-Hochstadt, Robert Bruce Findler, and Matthew Flatt. Chaperones and impersonators: Run-time support for contracts on higher-order, stateful values. Technical report, NU-CCIS-12-01, 2012.

SU96. 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.

Tan08. Éric Tanter. Expressive scoping of dynamically-deployed aspects. In *Proceedings of the 7th ACM International Conference on Aspect-Oriented Software Development (AOSD 2008)*, pages 168–179, Brussels, Belgium, April 2008. ACM Press.

Tan10. Éric Tanter. Execution levels for aspect-oriented programming. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, pages 37–48. ACM, 2010.

TCD13. Camille Teruel, Damien Cassou, and Stéphane Ducasse. Object Graph Isolation with Proxies. In *DYLA - 7th Workshop on Dynamic Languages and Applications, Collocated with 26th European Conference on Object-Oriented Programming - 2013*, 2013.

TFD+09. Éric Tanter, Johan Fabry, Rémi Douence, Jacques Noyé, and Mario Südholt. Expressive scoping of distributed aspects. In *Proceedings of the 8th ACM International Conference on Aspect-Oriented Software Development (AOSD 2009)*, pages 27–38, Charlottesville, Virginia, USA, March 2009. ACM Press.

TNCC03. Éric Tanter, Jacques Noyé, Denis Caromel, and Pierre Cointe. Partial behavioral reflection: Spatial and temporal selection of reification. In *Proceedings of OOPSLA '03, ACM SIGPLAN Notices*, pages 27–46, nov 2003.

VCM10. Tom Van Cutsem and Mark S. Miller. Proxies: design principles for robust object-oriented intercession APIs. In *Dynamic Language Symposium*, volume 45, pages 59–72. ACM, oct 2010.

VCM13. Tom Van Cutsem and Mark S. Miller. Trustworthy proxies — virtualizing objects with invariants. In *ECOOP'13*, 2013.

vLDN07. Martin von Löwis, Marcus Denker, and Oscar Nierstrasz. Context-oriented programming: Beyond layers. In *Proceedings of the 2007 International Conference on Dynamic Languages (ICDL 2007)*, pages 143–156. ACM Digital Library, 2007.

VTB98. John Viega, Bill Tutt, and Reimer Behrends. Automated delegation is a viable alternative to multiple inheritance in class based languages. *University of Virginia, Charlottesville, VA*, 1998.