Se podrán usar para
propagar refactorings?

Delegation Proxies: The Power of Propagation [*]

Erwann Wernli   Oscar Nierstrasz
Software Composition Group, University of Bern, Switzerland

Camille Teruel   Stéphane Ducasse
RMOD, INRIA Lille Nord Europe, France

Abstract

Scoping behavioral variations to dynamic extents is useful to support non-functional requirements that otherwise result in cross-cutting code. Unfortunately, such variations are difficult to achieve with traditional reflection or aspects. We show that with a modification of dynamic proxies, called *delegation proxies*, it becomes possible to reflectively implement variations that *propagate* to all objects accessed in the dynamic extent of a message send. We demonstrate our approach with examples of variations scoped to dynamic extents that help simplify code related to safety, reliability, and monitoring.

Categories and Subject Descriptors D.3.3 [*Software*]: Programming Languages — Constructs and Features

Keywords Reflection, proxy, 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 such boilerplate code in a single place. Aspects enable scoping 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 variation to the *dynamic extent* [39] 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 variations to dynamic extents increases the expressiveness of the language in useful ways [38, 39]. With such variations, it is for instance possible to execute code in a read-only manner [4] (thus improving safety), or to track all state mutations to ease recovery in case of errors (thus improving reliability), or to trace and profile methods at a fine-grained level (thus improving monitoring).

[*] PREPRINT. Cite as: Erwann Wernli, Oscar Nierstrasz, Camille Teruel, and Stephane Ducasse. *Delegation Proxies: The Power of Propagation*. In Proceedings of the 13th International Conference on Modularity, pp. 1–12, ACM, New York, NY, USA, 2014. doi: 10.1145/2577080.2577081

MODULARITY '14, April 22–26, 2014, Lugano, Switzerland.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2772-5/14/04…$15.00.
http://dx.doi.org/10.1145/2577080.2577081

We show in this paper that with minor changes to the way dynamic proxies operate, it becomes possible to reflectively implement variations that are scoped to dynamic extents. A dynamic proxy [18, 28, 42] 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, reified, and passed to the proxy's *handler* for further processing. To scope variations to dynamic extents using proxies, we must first slightly adapt the proxy mechanism, then implement specific handlers.

Our adaptation of dynamic proxies has the following characteristics: 1) it supports delegation [27] by rebinding self-reference, 2) it intercepts state accesses, both for regular fields and variables captured in closures, 3) it intercepts object creations. We refer to our extension of dynamic proxies as *delegation proxies*.

With delegation proxies, it becomes possible to implement handlers that will wrap all objects the target object accesses. Such a handler wraps the result of state reads and object instantiations, and unwraps the arguments of state writes. A proxy can consequently encode a variation that will be consistently *propagated* to all objects accessed during the evaluation of a message send (*i.e.*, its dynamic extent), without impacting objects in the heap.

Delegation proxies have several positive properties. First, delegation proxies do not lead to *meta-regressions* — infinite recursion that arises in reflective architectures when reflecting on code that is used to implement the reflective behavior itself. In aspect-oriented programming, this arises when an advice triggers the associated pointcut. The solutions to this problem generally add an explicit model of the different levels of execution [15, 40]. With delegation proxies, this problem doesn't appear since the proxy and its target are distinct objects and the propagation is enabled only for the proxy. No variation is active when executing the code of the handler. Second, variations expressed with delegation proxies *compose*, similarly to aspects. For instance, tracing and profiling variations can be implemented with delegation proxies and then composed to apply both variations. Third, delegation proxies naturally support *partial reflection* [41]. Only the objects effectively accessed in the dynamic extent of an execution involving a proxy pay a performance overhead; all other objects in the system remain unaffected, including the target.

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);
- 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 3);

- A formalization of delegation proxies and the propagation technique (Section 4);
- An implementation of delegation proxies in Smalltalk based on code generation (Section 5).

2. Delegation Proxies

We now describe how delegation proxies work and exemplify them with an implementation of tracing. Let us consider the Smalltalk method `Integer>>fib` [1] 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 delegation proxies to intercept message sends and print them. Like a dynamic proxy, a delegation proxy is a special object that acts as a surrogate for another object, called its *target*. The behavior of a proxy is defined by a separate object called its *handler*, whose methods are referred to as *traps* [42]. When an operation (message send, state access, etc.) is applied to a proxy, the proxy reifies the operation and instead invokes the corresponding trap in the handler. When an operation is intercepted, the handler can take some action and can reflectively perform the original operation on the target. Figure 1 shows the relationships between a proxy, a handler and a target.

The distinction between proxies and handlers is called *stratification* [9, 28, 42]. Stratification avoids name conflicts between application methods and traps, *i.e.*, between the base-level and the meta-level.

The target and the handler can be regular objects or proxies as well. By using proxies as the targets of other proxies, we obtain a chain of delegation. Each of the variations implemented by the handlers of the proxies in the chain will be triggered in order. This is a natural way to compose different variations together.

A tracing proxy can be obtained by instantiating a proxy with a tracing handler. For convenience, we add method `Object>> tracing` that returns a tracing proxy for any object. For instance, `2 tracing` returns a tracing proxy for the number `2`.

[1] In Smalltalk, closures are expressed with square brackets (`[ ... ]`) and booleans are objects. The method `ifTrue:` takes a closure as argument: if the receiver is `true`, the closure is evaluated by sending the message `value`. The up-arrow (`↑ ...`) denotes a return statement.

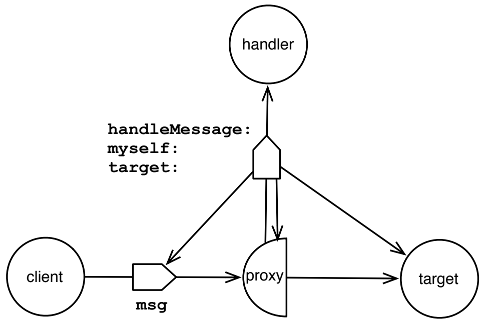

Figure 1. Example of message interception. First, the client sends the message `msg` to a proxy. Then, the proxy intercepts the message and invokes the handler trap associated with message reception (`handleMessage:myself:target:`) with three arguments: the reified message, the proxy itself and the target.

```
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* trap that prints the name of the reified message. Listing 4 shows the code of such a *message* trap:

```
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. 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. In the case of a reflective invocation with delegation, the method is executed with `self` rebound to the proxy, which allows the proxy to further intercept operations happening during this method execution.

If the target of a proxy is another proxy, proxies form chains of delegation. The identity of the proxy that received the original message send is consistently passed to the handlers.

2.1 Propagation

The tracing handler in the previous section defines a *message* trap. Doing so 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 variation during the evaluation of a message send, all objects accessed during the evaluation must be represented with proxies. To achieve this, we can implement a handler that replaces all object references accessed by proxies. This way, the 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 or the resolution of literals.

The following rules suffice to make sure that all objects are represented with proxies. To distinguish between the initial proxy and the proxies created during the propagation, we call the former the *root* proxy. We need three rules to control how objects must be wrapped:

- *Wrap the initial arguments*. When the root proxy receives a message, the arguments must be wrapped with proxies. We don't need to wrap the arguments of other message sends: the following rules ensure that the arguments were already wrapped in the context of the caller.
- *Wrap field reads*. This way, references to fields are represented with a proxy.
- *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 with `+`,`-`,`/`. Similarly, the resolution of literal 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 pollute the heap via the fields of target objects. They can be garbage collected once the propagation is over.
- *Unwrap the initial return value*. The root proxy unwraps the objects returned to the clients.

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

2.2 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[2] and the target.
- `handleReadField:myself:target:`
  The trap for field reads takes as parameters the field name, the original proxy and the target.

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

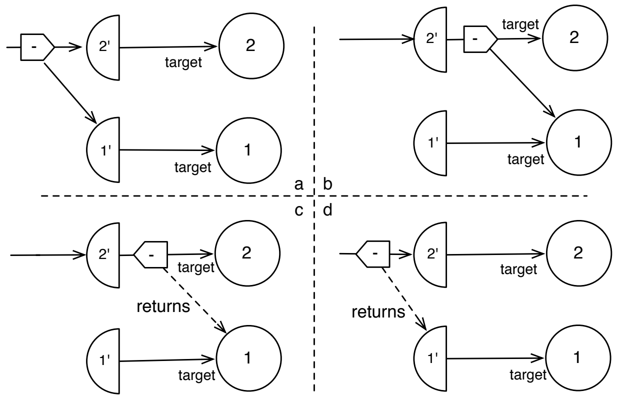

Figure 2. Illustration of propagation during the subtraction 2 - 1. A proxy to 2 receives the subtraction message "-" with a proxy to 1 as argument (a). The message is forwarded to 2 to perform the actual subtraction (b) that returns 1 (c). Finally the result is wrapped (d).

- `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.

Instantiations of objects using literals, *e.g.*, the instantiation of a string, are intercepted with the *literal* trap. In Java, an additional trap would be needed to intercept constructor invocations. Similar considerations hold for access to static fields and invocation of static methods.

Similarly to `perform:`, reflective methods to read fields, to write fields and to resolve literals are extended with an additional parameter `myself`. They become `instVarNamed:myself:`, `instVarNamed:put:myself` and `literal:myself:`. The parameter `myself` is needed, because proxies may form chains of delegation. When invoked on a proxy, the reflective operations will trigger the corresponding trap. The parameter `myself` is passed to the traps along the chain to preserve the identity of the proxy that originally intercepted the operation.

With the support of delegation and the ability to intercept state accesses and object instantiations, it becomes possible to implement a handler that realizes the propagation technique. The code of such a propagating handler is shown in Listing 5. We assume the existence of a class `Reflect` to unwrap proxies.

```
PropHandler>>initialize
  isRoot := true

PropHandler>>handleMessage: m myself: p target: t
  ↑ m selector isPrimitive
    ifTrue: [ self wrap: (t perform: m myself: p) ]
    ifFalse: [
      self isRoot
        ifTrue: [
          m arguments: (self wrapAll: m arguments).
          self unwrap: (t perform: m myself: p) ]
        ifFalse: [ t perform: m myself: p ] ]

PropHandler>>handleReadField: f myself: p target: t
  ↑ self wrap: (t instVarNamed: f myself: p).

PropHandler>>handleWriteField: f value: v proxy: p
```

```
           target: t
  t instVarNamed: f put: (self unwrap: v) myself: p.
  ↑ v

PropHandler>>handleLiteral: l myself: p target: t
  ↑ self wrap: l

PropHandler>>wrap: anObject
  | handler |
  handler := self class new.
  handler isRoot: false.
  ↑ Proxy handler: handler target: anObject

PropHandler>>wrapAll: aCollection
  ↑ aCollection collect: [ :each | self wrap: each ]

PropHandler>>unwrap: aProxy
  ↑ Reflect targetOf: aProxy

PropHandler>>isRoot: aBoolean
    isRoot := aBoolean
```

Listing 5. Tracing handler implementing the propagation technique.

2.3 Closures

A closure should be evaluated with the variations that are active in the current dynamic extent, and not the variations that were active when the closure was created. 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 proxying is only applied on demand.

Variables captured in a closure are stored in indexed fields. Let us describe 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`. When we evaluate the closure via the proxy, the *literal* trap will wrap `2` before it is used. 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

2.4 Transparency

Traps are implemented in a separate handler and not in the proxy itself. If an application defines an application-level method whose name collides with the name of a trap, the explicit invocation of this method will be trapped by the handler. It avoids conflicts between the base-level and the meta-level. The proxy can expose the exact same interface as its target.

It is impossible to deconstruct a proxy to obtain its handler or its target without using reflective capabilities. Since security is not a concern, we assume for simplicity 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.

For increased security, these methods could to be stratified with mirrors [9], in which case handlers would need to have access to a mirror when instantiated.

3. Examples

Since delegation proxies subsume dynamic proxies, they can be used to implement all classic examples of dynamic proxies like lazy values, membranes, remote stubs, etc. We omit such examples that can be found elsewhere in the literature [13, 18, 28].

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 inherit from the class `PropHandler` that implements the propagation technique for reuse (see Listing 5).

3.1 Object Versioning

To tolerate errors, developers implement recovery blocks that undo mutations and leave the objects in a consistent state [33]. Typically, this requires cloning objects to obtain snapshots. Delegation proxies enable the implementation of object versioning elegantly. 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 inherits from a base handler that implements the propagation technique.

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

Listing 7. Recording handler

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

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

Listing 8. Enabling recording

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

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

Listing 9. Undoing changes (`m` stands for mutation)

3.2 Read-only Execution

Read-only execution [4] prevents mutation of state during evaluation. Read-only execution can dynamically guarantee that the evaluation of a given piece of code is either side-effect free 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 functions, or to guarantee that methods are deeply read-only. Read-only execution can be implemented trivially using propagation and a handler that fails upon state writes.

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

Listing 10. Read-only handler

3.3 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`[3]).

Dynamic scoping can be realized in Smalltalk using stack manipulation [16] 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 proxied 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`.

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

3.4 Profiling

Previous sections already illustrated delegation proxies using tracing. The exact 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.
  [ ↑ super handleMessage: m myself: p target: t ]
    ensure: [
        | duration |
        duration := Time now - start.
        tallies add: {t. m. duration} ]
```

Listing 12. A simple profiling handler

4. Semantics

We formalize delegation proxies by extending SMALLTALKLITE [6], a lightweight calculus in the spirit of CLASSICJAVA [19] that omits static types. This paper does not assume any 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, a *literal* trap does not make sense. Instead, we introduce a *new* trap that intercepts object instantiations.

The syntax of our extended calculus, SMALLTALKPROXY, is shown in Figure 3. The only addition to the original syntax is the new expression **proxy** $e\ e$.

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

Figure 3. 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 $\textbf{super}\langle c\rangle.m\langle o\rangle(v^*)$ 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 the translation of expressions into redexes are shown below.

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 $expr$.

Translation from the main expression to an initial redex is carried out by the $o[\![e]\!]_c$ function (see Figure 4). 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

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

Figure 4. Translating expressions to redexes

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

Figure 5. Redex syntax

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 redex.

$P \vdash \langle\epsilon,\mathcal{S}\rangle \hookrightarrow \langle\epsilon',\mathcal{S}'\rangle$ means that we reduce an expression (redex) 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 $\mathcal{S}'$. The store consists of a set of mappings from addresses $a \in \text{dom}(\mathcal{S})$ 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 $\mathcal{S} = \{\}$.

The reductions are summarized in Figure 6. 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 an object (*i.e.*, not a proxy), the expression reduces to a fresh address a, bound in the store 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 $\mathbf{new}\langle t\rangle\ c$ 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 store (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 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 gets ÒstuckÓ and 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.

4.1 Identity Proxy

As was discussed in subsection 2.4, the system requires the ability to reflectively apply operations on base objects and 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 7.

All three primitives take a last 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** is applied to an object address, the argument *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. idHandler is an instance of a handler class that defines the following methods:

$$\begin{array}{lcl}
\mathbf{newTrap}(t, my) & = & t\\
\mathbf{readTrap}(t, f, my) & = & t.\mathbf{read}(f, my)\\
\mathbf{writeTrap}(t, f, o, my) & = & t.\mathbf{write}(f, o, my)\\
\mathbf{messageTrap}(t, m, o^*, my) & = & t.\mathbf{send}(m, o^*, my)
\end{array}$$

We can show that sending a message to an identity proxy will delegate the message to the target, and rebind **self** to the proxy.

Let us consider an object s that sends the message $m(o)$ to a proxy p = **proxy** idHandler t. Object t is an instance of class c which defines method $m(x)$ with body e = **self** $n(x)$.

$$\begin{array}{ll}
p.m\langle s\rangle(o) & \\
\mathsf{idHandler}.\mathbf{messageTrap}(t, m, o, p) & [\textit{send-proxy}]\\
t.\mathbf{send}(m, o, p) & [\textit{send}]\\
p[\![e[o/x]]\!]_c & [\textit{reflect-message}]\\
p[\![\mathbf{self}]\!]_c.n\langle p\rangle(o) & [\textit{translation}]\\
p.n\langle p\rangle(o) & [\textit{translation}]
\end{array}$$

4.2 Propagating Identity Proxy

Following the technique of propagation presented in subsection 2.1, 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 propagation. This technique requires the ability to unwrap a proxy. The expression **unproxy** is added to the language as defined in Figure 7. We also assume the existence of the traditional sequencing (;) operation.

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

Figure 6. Reductions for SMALLTALKPROXY

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

Figure 7. Reflective facilities added to SMALLTALKPROXY

The handler propHandler is defined as follows:

$$
\begin{array}{lcl}
\textbf{newTrap}(t, \mathit{my}) & = & \textbf{proxy}\ \mathsf{propHandler}^*\ t \\
\textbf{readTrap}(t, f, \mathit{my}) & = & \textbf{proxy}\ \mathsf{propHandler}^*\ (t.\textbf{read}(f, \mathit{my})) \\
\textbf{writeTrap}(t, f, o, \mathit{my}) & = & t.\textbf{write}(f, \textbf{unproxy}\ o, \mathit{my}); o \\
\textbf{messageTrap}(t, m, o^*, \mathit{my}) & = & \textbf{unproxy}(t.\textbf{send}(m, \\
 & & \quad (\textbf{proxy}\ \mathsf{propHandler}^* o^1, \ldots, \\
 & & \quad \textbf{proxy}\ \mathsf{propHandler}^* o^n), \mathit{my}))
\end{array}
$$

The handler propHandler* is defined as follows:

$$
\begin{array}{lcl}
\textbf{newTrap}(t, \mathit{my}) & = & \textbf{proxy}\ \mathsf{propHandler}^*\ t \\
\textbf{readTrap}(t, f, \mathit{my}) & = & \textbf{proxy}\ \mathsf{propHandler}^*\ (t.\textbf{read}(f, \mathit{my})) \\
\textbf{writeTrap}(t, f, o, \mathit{my}) & = & t.\textbf{write}(f, \textbf{unproxy}\ o, \mathit{my}); o \\
\textbf{messageTrap}(t, m, o^*, \mathit{my}) & = & t.\textbf{send}(m, o^*, \mathit{my})
\end{array}
$$

We can formally express the intuitive explanation of subsection 2.1 about soundness of the propagation.

Let us assume that all values in the expression $E[e]$ are proxies (using the propHandler*). The reduction rules that can match are [new-proxy], [get-proxy], [set-proxy], [super], [let], and [message-proxy]. According to the definition of the propHandler* traps, rule [new-proxy] will preserve the invariant that all values are proxies. Rule [get-proxy] does so as well. Rule [set-proxy] preserves the assumption since it returns the value written, which we know is a proxy. Rules [super] and [let] do as well since they only bind variables with existing values, which we know are proxies. Similarly, rule [message-proxy] will bind **self** with the expected proxy (see previous section). It also binds the variables with the passed arguments, which are known to be proxies. Since all values remain proxies, the evaluation is consistent.

The initial redex is evaluated with nil as object context: $\mathsf{nil}[\![e]\!]_{\mathsf{Object}}$. If the proxy $p =$ **proxy** propHandler nil is used instead of nil, the assumption is initially true, and will not be broken during evaluation.

5. Implementation

We have implemented a prototype of delegation proxies in Smalltalk that relies on code generation. For each existing method in a base class, a hidden method with an additional parameter `myself` and a transformed body is generated. 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 [18], proxy classes are auto-generated. Let us consider the class `Suitcase`:

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

Suitcase>>printString
  ↑ 'Content: ' concat: content.
```

Listing 13. 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: slf
  ↑ aLiteral

Suitcase>> readContentMyself: slf
  ↑ content

 Suitcase>> writeContent: value myself: slf
  ↑ content := value
```

Listing 14. 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: slf
  ↑ ( slf literal: 'Content: ' myself: slf )
     concat: (self readContentMyself: slf).
```

Listing 15. 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.*, `printString`, `printStringMyself:`, `readContentMyself:`, and `writeContent:myself:`. The methods invoke respectively *message*, *read* and *write* traps on the handler.

```
SuitcaseProxy>> printString
  ↑ self printStringMyself: self

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

Listing 16. Sample generated method in proxy class of `Suitcase`

5.1 Classes

Smalltalk has first-class classes whose behaviors are defined in meta-classes. The class and meta-class hierarchies are parallel. 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 is shown in Figure 8.

5.2 Closures

Closures are regular objects that are adapted upon creation and evaluation according to subsection 2.3. When a closure defined in an original uninstrumented method is proxied, the code of the closure is transformed lazily.

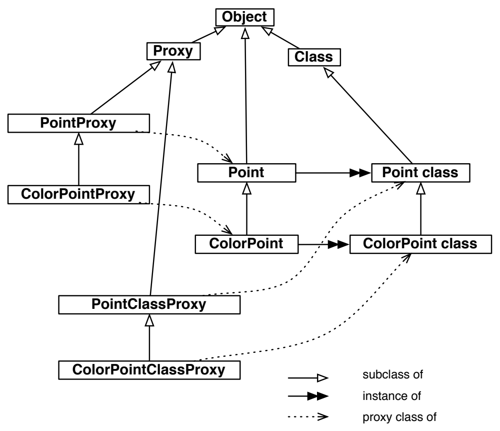

Figure 8. Inheritance of classes, meta-classes, and auto-generated proxy classes.

5.3 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 less levels of indirection. For instance, a `SuitcaseProxy` with a `Tracing` handler can be woven into a `SuitcaseTracingProxy`:

```
SuitcaseTracingProxy>> printStringMyself: slf
  | msg |
  msg := Message selector: #printString arguments: {} .
  Transcript
           print: target asString;
           space;
           print: msg asString;
           cr.
  ↑ target printStringMyself: slf.
```

Listing 17. 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 [20] developed for aspect compilers [29].

5.4 Performance

Delegation proxies have no impact on performance if not used: the transformation adds new code but does not alter the existing one. When used, we need to distinguish between the performance of delegation proxies themselves and the overhead of the propagation technique.

Used sparingly, delegation proxies do not entail performance issues. The situation is similar to traditional forwarding proxies. Used extensively with our propagation technique, the cost of delegation proxies is prohibitive unless weaving is used. With weaving, benchmarks of Fibonacci[4] reveal a performance degradation of below one order of magnitude (8x slower). We believe it is an encouraging result given that delegation proxies enable unanticipated behavioral reflection, which is known to be costly.

With our propagation technique, a given object might be wrapped multiple times, producing multiple equivalent proxies. In the Fibonacci examples, `1`, `2` and `[^ self]` are literals that are intercepted and wrapped thousands of times. This increases the

[4] CogVM 6.0, Mac OS X, 2.3 GHz Intel Core

number of objects created and puts pressure on the garbage collector. Future work could address this issue, possibly with caching.

6. Discussion

Scoping Scoping variations to dynamic extents was the motivation for delegation proxies. However, the propagation technique that we have presented covers only one particular form of scoping that can be realized with delegation proxies. Since the propagation is implemented reflectively, it can be customized in many ways.

If a variation is active in all threads, the propagation can for instance be adapted to proxy only instances of application classes and skip kernel classes (string, dictionaries, arrays, etc.). The application and the kernel form two layers. Any application object referenced by a kernel object has necessarily been provided by the application. If application objects are wrapped, this guarantees that kernel objects hold only references to proxies of application objects. Therefore, if a kernel object sends a message to an application object, the propagation will start again. Kernel objects will not unwrap proxies of application objects upon state writes, and the heap might contain references to proxies (the variation must thus be active in all threads). Omitting kernel objects from the propagation could be desirable to improve performance.

It is also be possible to adapt the propagation so that the root proxies doesn't unwrap the objects they returns. Variations that apply security concerns such as access control usually have this requirement [4].

Closure wrapping rules can also be customized in different ways. It is possible to control which closures should "escape" the dynamic extent as proxy or not. If a method parallelizes work internally using multiple threads, the propagation could for instance be customized to propagate to those threads as well. Or if the system uses callbacks, the callbacks could restore the variation that was active when they were created.

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* [39].

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 must inherit from the original class. Reflective operations 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.

Delegation proxies require that reflective operations have an additional parameter 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.

7. Related Work

Method Dispatch MOPs, AOP and proxies are various approaches that enable the interception and customization of method dispatch. MOPs reify the execution into meta-objects that can be customized [23]. AOP adopts another perspective on the problem and enables the definition of join points where additional logic is woven [24]. MOP and AOP share similarities with method combination of CLOS [14].

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" [42]. An important question for proxies is whether to support them natively at the language level or via lower-level abstractions.

Most dynamic languages support proxies via traps that are invoked when a message cannot be delivered [28]. However, modern proxy mechanisms stratify the base and meta levels with a handler [18, 28, 42], including Java that uses code generation to enable proxies for interfaces. The mechanism was extended to enable proxies of classes as well [18].

AOP and MOP inherently suffer from meta-regression issues, unless the meta-levels are explicitly modeled [11, 15, 40]. In contrast to AOP and MOPs, delegation proxies do not suffer from meta-regression issues since the adapted object and the base object are distinct. For instance, the tracing handler in Listing 4 does not lead to a meta-regression since it sends the message `asString` to the target, which is distinct from the proxy (in parameter `myself`). System code can in this way be adapted. Also, delegation proxies naturally enable partial reflection [41] since objects are selectively proxied.

Recent works on proxies in dynamic languages have studied orthogonal issues related to stratification [12, 42], preservation of abstractions and invariants [13, 37], and traps for values [5]. Only Javascript direct proxies support delegation [13]. However, Javascript proxies do not enable the interception of object instantiations; the variables captured in a closure will not be unproxied upon capture and proxied upon evaluation.

In addition to full-fledged MOPs and AOP, reflective language like Smalltalk provide various ways to intercept message sends [17]. Java and .NET support custom method dispatch via JSR 292 [31] and the Dynamic Language Runtime [30].

Composing Behavior Inheritance leads to an explosion in the number of classes when multiple variations (decorations) of a given set of classes must be designed. Static traits [35] 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 [10] to statically parameterize classes with a decorator (called wrapper in their terminology). Bettini *et al.* [8] proposed a similar construct but composition happens at creation time. Ressia *et al.* proposed *talents* [34] 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 [7].

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 18. Differences between approaches to decoration

Several languages that combine class-based inheritance and object inheritance (*i.e.*, delegation) have been proposed [25, 43]. 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 [32], 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 `flow` and `cfbelow`. Aware of the limitations of control-flow pointcuts, some AOP implementations provide specific constructs to scope to dynamic extents, *e.g.*, CaesarJ's `deploy` [2]. 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 [29].

In context-oriented programming (COP) [22, 44], variations can be encapsulated into layers that are dynamically activated in the dynamic extent of an expression. Unlike delegation proxies that support *homogenous* variations, COP supports best *heterogenous* variations [1]. 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 [26], perspectives [36], and subjects [21]. Delegation proxies can realize dynamic scoping via reference flow, by proxying and unproxying objects accesses during the execution. Delegation proxies can provide a foundation to design contextual variations.

Similarly to our approach, the handle model proposed by Arnaud *et al.* [3, 4] 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. Propagation unfolds from a principled use of delegation. Our approach is more flexible since it decouples the notion of propagation from the notion of proxy.

8. 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 avoid meta-regressions. 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 forwarding proxies). The overhead of our propagation technique is of factor 8 when handlers are woven into dedicated proxies. Excluding system classes, if viable, can improve performance further. For optimal performance, the language should provide native support of delegation proxies.

In the future, we plan to further mature our implementation, notably the weaver, and explore native support at the VM level.

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

[1] S. Apel, T. Leich, and G. Saake. Aspectual feature modules. *IEEE Trans. Softw. Eng.*, 34(2):162–180, Mar. 2008.

[2] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. An overview of CaesarJ. *Transactions on Aspect-Oriented Software Development*, 3880:135 – 173, 2006.

[3] J.-B. Arnaud. *Towards First Class References as a Security Infrastructure in Dynamically-Typed Languages*. PhD thesis, Université des Sciences et Technologies de Lille, 2013.

[4] J.-B. Arnaud, M. Denker, S. Ducasse, D. Pollet, A. Bergel, and M. Suen. Read-only execution for dynamic languages. In *Proceedings of the 48th International Conference on Objects, Models, Components, Patterns (TOOLS EUROPE'10)*. LNCS Springer Verlag, July 2010.

[5] T. H. Austin, T. Disney, and C. Flanagan. Virtual values for language extension. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, volume 46 of *OOPSLA '11*, pages 921–938, New York, NY, USA, Oct. 2011. ACM.

[6] A. Bergel, S. Ducasse, O. Nierstrasz, and R. Wuyts. Stateful traits and their formalization. *Journal of Computer Languages, Systems and Structures*, 34(2-3):83–108, 2008.

[7] L. Bettini, S. Capecchi, and F. Damiani. On flexible dynamic trait replacement for java-like languages. *Science of Computer Programming*, 2011.

[8] L. Bettini, S. Capecchi, and E. Giachino. Featherweight wrap java. In *Proc. of SAC (The 22nd Annual ACM Symposium on Applied Computing), Special Track on Object-Oriented Programming Languages and Systems (OOPS)*, pages 1094–1100. ACM Press, 2007.

[9] G. Bracha and D. 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.

[10] M. Büchi and W. Weck. Generic wrappers. In E. Bertino, editor, *ECOOP 2000 - Object-Oriented Programming, 14th European Conference, Sophia Antipolis and Cannes, France, June 12-16, 2000, Proceedings*, volume 1850 of *Lecture Notes in Computer Science*, pages 201–225. Springer, 2000.

[11] S. Chiba, G. Kiczales, and J. Lamping. Avoiding confusion in metacircularity: The meta-helix. In K. Futatsugi and S. Matsuoka, editors, *Proceedings of ISOTAS '96*, volume 1049 of *Lecture Notes in Computer Science*, pages 157–172. Springer, 1996.

[12] T. V. Cutsem and M. S. Miller. On the design of the ECMAScript reflection api. Technical report, Vrije Universiteit Brussel, 2012.

[13] T. V. Cutsem and M. S. Miller. Trustworthy proxies: Virtualizing objects with invariants. In *ECOOP 2013*, 2013.

[14] L. G. DeMichiel and R. P. Gabriel. The Common Lisp object system: An overview. In J. Bézivin, J.-M. Hullot, P. Cointe, and H. Lieberman,

editors, *Proceedings ECOOP '87*, volume 276 of *LNCS*, pages 151–170, Paris, France, June 1987. Springer-Verlag.

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

[16] P. Deutsch. Building control structures in smalltalk-80. *Byte*, 6(8):322–346, aug 1981.

[17] S. Ducasse. Evaluating message passing control techniques in Smalltalk. *Journal of Object-Oriented Programming (JOOP)*, 12(6):39–44, June 1999.

[18] P. 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.

[19] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 171–183, New York, NY, USA, 1998. ACM Press.

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

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

[22] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3), Mar. 2008.

[23] G. Kiczales, J. des Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[24] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *ECOOP'97: Proceedings of the 11th European Conference on Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242, Jyvaskyla, Finland, June 1997. Springer-Verlag.

[25] G. 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.

[26] B. B. Kristensen. Object-oriented modeling with roles. In J. Murphy and B. Stone, editors, *Proceedings of the 2nd International Conference on Object-Oriented Information Systems*, pages 57–71, London , UK, 1995. Springer-Verlag.

[27] H. Lieberman. Using prototypical objects to implement shared behavior in object oriented systems. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, volume 21, pages 214–223, Nov. 1986.

[28] M. Martinez Peck, N. Bouraqadi, M. Denker, S. Ducasse, and L. Fabresse. Efficient proxies in smalltalk. In *Proceedings of the International Workshop on Smalltalk Technologies*, IWST '11, pages 8:1–8:16, New York, NY, USA, 2011. ACM.

[29] H. Masuhara, G. Kiczales, and C. Dutchyn. A compilation and optimization model for aspect-oriented programs. In *Proceedings of the 12th international conference on Compiler construction*, CC'03, pages 46–60, Berlin, Heidelberg, 2003. Springer-Verlag.

[30] Microsoft. Microsoft .net dynamic language runtime.

[31] Oracle. Jsr 292: Supporting dynamically typed languages on the java platform.

[32] K. Ostermann. Dynamically composable collaborations with delegation layers. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, ECOOP '02, pages 89–110, London, UK, 2002. Springer-Verlag.

[33] F. Pluquet, S. Langerman, and R. Wuyts. Executing code in the past: efficient in-memory object graph versioning. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 391–408, New York, NY, USA, 2009. ACM.

[34] J. Ressia, T. Gîrba, O. Nierstrasz, F. Perin, and L. Renggli. Talents: an environment for dynamically composing units of reuse. *Software: Practice and Experience*, 2012.

[35] N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black. Traits: Composable units of behavior. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP'03)*, volume 2743 of *LNCS*, pages 248–274, Berlin Heidelberg, July 2003. Springer Verlag.

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

[37] T. S. Strickland, S. Tobin-Hochstadt, R. B. Findler, and M. Flatt. Chaperones and impersonators: Run-time support for reasonable interposition. In *OOPSLA '12: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, Oct. 2012. To appear.

[38] É. 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, Apr. 2008. ACM Press.

[39] É. Tanter. Beyond static and dynamic scope. In *Proceedings of the 5th symposium on Dynamic languages*, DLS '09, pages 3–14, New York, NY, USA, 2009. ACM.

[40] É. Tanter. Execution levels for aspect-oriented programming. In *Proceedings of AOSD'10)*, pages 37–48, Rennes and Saint Malo, France, Mar. 2010. ACM Press. Best Paper Award.

[41] É. Tanter, J. Noyé, D. Caromel, and P. Cointe. Partial behavioral reflection: Spatial and temporal selection of reification. In *Proceedings of OOPSLA '03, ACM SIGPLAN Notices*, pages 27–46, nov 2003.

[42] T. Van Cutsem and M. S. Miller. Proxies: design principles for robust object-oriented intercession apis. In *Proceedings of the 6th symposium on Dynamic languages*, DLS '10, pages 59–72, New York, NY, USA, 2010. ACM.

[43] J. Viega, B. Tutt, and R. Behrends. Automated delegation is a viable alternative to multiple inheritance in class based languages. Technical report, University of Virginia, Charlottesville, VA, USA, 1998.

[44] M. von Löwis, M. Denker, and O. 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.