Beyond Static and Dynamic Scope

Éric Tanter *

PLEIAD Laboratory
Computer Science Department (DCC)
University of Chile – Santiago, Chile
http://pleiad.cl/etanter

Abstract

Traditional treatment of scoping in programming languages considers two opposite semantics: static scoping, where the scope of a binding is a block of program text, and dynamic scoping, where a binding is in effect during the whole reduction of an expression to a value. Static scoping and dynamic scoping are however but two points in the design space of scoping mechanisms. As a result, most proposed language mechanisms that rely on some notion of scoping, such as variable bindings of course, but also more exotic ones like aspects and mixin layers, adopt either one or the other semantics. As it turns out, these two semantics are sometimes too extreme, and a mixture of both is needed. To achieve this, language designers and/or programmers have to resort to ad hoc solutions. We present a general scoping model that simply expresses static and dynamic scoping, and that goes further by allowing fine-grained exploration of the design space of scoping. The model, called *scoping strategies*, gives precise control over propagation and activation of language mechanisms. While we have already studied it for aspects, we hereby show that the model is not restricted to aspects, by treating in depth its application to the most standard kind of adaptation: variable bindings. We also briefly discuss its application to mixin layers, and program monitoring. We believe that research in programming language constructs can benefit from a more flexible notion of scoping that goes beyond the static/dynamic dichotomy.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Semantics

General Terms Languages, Design

Keywords Variable bindings, scope, lexical scope, dynamic scope, adaptation.

* Partially funded by FONDECYT projects 11060493 & 1090083.

DLS'09, October 26, 2009, Orlando, Florida, USA.

1. Scope and Adaptations

The notion of *scope* in programming languages is traditionally associated with that of *bindings*. Following Moreau's definition of the terms [18], a binding is an association between a name (or a variable) and a value. The scope of a variable binding is the text where occurrences of this name refer to the binding. According to *lexical* (*a.k.a.* static) scope, a variable in an expression refers to the innermost lexically-enclosing construct declaring that variable (*i.e.* the scope is a block of program text). On the contrary, a *dynamic* binding is an association that exists and can only be used during the dynamic extent of an expression. A variable bound dynamically is also often called a dynamic variable, or a parameter.

Dynamic binding associates data with the current execution context, and therefore allows to pass on data to functions without having to explicitly declare this data in the interface of the function. As nicely put by Kiselyov *et al.*, *"in general, dynamic binding generalises global state and the singleton pattern to multiple application instances that may coexist in the same execution environment"* [16]. A particular feature of dynamic bindings is that they are *not* captured in a lexical closure. This allows for a number of well-known benefits of dynamic binding, like conciseness, modularity and adaptability [16, 18]. Typical examples are redirecting the output of a program, defining exception handlers, handling host-local state in a distributed system, etc.

The notion of scope is however not restricted to variable bindings. It has also been studied for instance in the area of Aspect-Oriented Programming (AOP) [15], first by Tucker and Krishnamurthi when exploring what it means to provide pointcuts and advice in higher-order languages, further refined by Dutchyn and the same authors [7]. Recall that in the pointcut-advice model of aspect-oriented programming [17, 24], as embodied in *e.g.* AspectJ [14], crosscutting behavior is defined by means of pointcuts and advices. A pointcut is a predicate that matches program execution points, called join points, and an advice is the action to be taken at a join point matched by a pointcut. An aspect is a module that encompasses a number of pointcuts and advices. The *scope* of an aspect is the set of join points the aspect *sees*, *i.e.* against which its pointcuts are matched. In AOP, it is therefore possible to define aspects with static scope—such that they only see the join points in the lexical region on which they are deployed—or with dynamic scope—such that they see all join points in the dynamic extent of that region.

There is at least one other area where scoping is discussed, that of class extensions. For instance, in ContextL, class definitions are split into layers that can be dynamically (de-)activated for a certain dynamic extent [5]. There are also proposals for statically-scoped class extensions [3, 25].

adaptation	semantics	deployment expression	scoping	languages
binding	occurrences of identifier substituted by bound value	`let` `dlet`, `fluid-let`, ...	lexical dynamic	Scheme & co Λ_d & co
aspect	execution events matched by pointcuts, may trigger advice	`around` `deploy`, `fluid-around`	lexical dynamic	AspectScheme (AS) CaesarJ, AS
layer	method and field lookup in extended class definition	*declaration* `with-active-layer`	lexical dynamic	ClassboxJ, eJava ContextL

Table 1. Adaptations and scoping.

Table 1 summarizes this brief overview of scoping in different contexts. It shows a number of program *adaptations*, like variable bindings, aspects, layers, and informally describes their semantics. For each, it presents a number of expressions with which these adaptations are *deployed*, their *scoping semantics*, as well as a number of illustrative languages and systems.

The purpose of this non-exhaustive table is to setup a general background for a discussion of scoping, as well as introducing our terminology. Furthermore, it shows that many different language mechanisms are tied to the lexical/dynamic dichotomy. We have previously presented an expressive scoping model for aspects [22], and showed how this model also allows for expressive scoping of distributed aspects [23]. Recent discussions with researchers outside the aspect community convinced us of the necessity to distill the essence of expressive scoping and formulate it in general terms, in order to do it justice.

To this end, this paper progressively introduces the model of *scoping strategies* for the classical case of variable bindings[1]. Section 2 introduces two dimensions of scoping, related to *propagation* of an adaptation. While simple, this model subsumes existing semantics. We extend it in Section 3 by considering dynamic propagation functions, and in Section 4 by introducing the notion of *activation* of an adaptation. In Section 5 we present the formal operational semantics of a language with scoped bindings, Λ_σ, and its prototype implementation. Section 6 briefly discusses instantiations of scoping strategies to aspects, layers, and debugging. Section 7 discusses related work and Section 8 concludes.

2. Scoping Dimensions

The characterization of scoping usually refers to the notion of the *region of program* where a binding is in effect[2]. If this region is statically determinable, that is static (lexical) scoping. If not, then it is dynamic scoping. We believe this characterization to be insufficient to be able to precisely denote fine-grained related "regions" of program execution. As a first step, we argue that seeing scoping as a *propagation* problem is more expressive.

There are two obvious propagation dimensions in a program:

- propagation on the call stack: is a currently-available binding made available in a subsequent stack frame?
- propagation in delayed evaluation: is a currently-available binding made available in a procedural value that is created?

[1] In previous work, the model is called *deployment strategy*, but we now find it clearer to separate the notion of deployment (related to the deployment expression, like `let`, `deploy`, `with-active-layer`, etc.) from the scoping strategy that specifies the scope of the deployed adaptation (binding, aspect, layer, etc.).

[2] From now on, we focus on variable bindings as a particular kind of adaptation. The discussion can be generalized by replacing occurrences of "binding" with "adaptation".

We introduce the notion of a *scoping strategy* as a pair $\sigma = \langle c, d \rangle$, where c denotes call stack propagation, and d denotes delayed evaluation propagation. For now, we just consider c and d to be boolean values. Though trivial, this model is already sufficient to go beyond static and dynamic scoping. To justify our expressiveness claim, we first note that static and dynamic scoping are straightforwardly expressed as scoping strategies, and then show two more interesting semantics that are not possible to obtain otherwise.

2.1 Expressing static and dynamic scoping

Both static and dynamic scoping can be directly expressed as scoping strategies.

Static binding A static binding, *a.k.a.* lexical binding, is available only in the lexical region where it is deployed. This includes all procedural values defined in that region. That is, a static binding always propagates in procedural values. On the other hand, a static binding is not necessarily visible in the dynamic extent of the body of its deployment expression: it does not propagate on the call stack. So, a static binding has scoping strategy $\sigma = \langle false, true \rangle$.

```
(define foo (lambda ()
   (let ((x 2))
     (lambda (a)
       (lambda (b)
         (lambda (c)
           (average (+ a b c) x)))))))
```

Above, the binding $x \rightarrow 2$ is available in all nested lambdas. Similarly, since function parameters are also statically scoped, they are also available in all the nested lambdas. Hence, the innermost lambda in the example has access to all the bindings.

Dynamic binding A dynamic binding is in effect during the extent of the evaluation of the expression on which it is deployed. That is, it always propagates on the call stack. However, a dynamic binding is never captured in lexical closures. Hence, it has scoping strategy $\sigma = \langle true, false \rangle$.

```
(define x 0)
(define foo (lambda (y) (+ x y)))
(define bar (lambda (z) (foo (- z 1))))

> (define f (dlet ((x 2))
              (write (bar 5))
              (lambda () (+ x x))))
6

> (f)
0
```

The binding $x \rightarrow 2$ follows the call stack, such that it is available in the body of `bar`, as well as in the body of `foo`. However, it is *not* available in the body of `f`.

2.2 Beyond static and dynamic scope

In this first model a scoping strategy has two components, each of which can be either true or false. This logically leads to four possible combinations. We have seen that two of them correspond to the well-known static and dynamic scoping semantics. We now introduce the two other semantics, and illustrate their use.

Pervasive binding A pervasive binding is a binding that is in effect during the whole extent of the reduction of the expression on which it is deployed, like a dynamic binding. In addition, the binding is also available for all the procedural values that are created during that extent, as well as the procedural values created by these values, and so on. In other words, a pervasive binding, deployed with `plet`, propagates on both the call stack and delayed evaluation dimensions: it has a scoping strategy $\sigma = \langle true, true \rangle$.

```
(define handler (lambda (e) (write "default")))
(define divide #f)
(define (init-divide)
  (set! divide (lambda (x)
                 (lambda (y)
                   (if (eqv? 0 y) (handler ...)
                       (/ x y))))))

> (plet ((handler (lambda (e) (write "plet"))))
    (init-divide))
> ((divide 10) 0)
"plet"
```

In the above example, the `divide` function is created in the dynamic extent of the `plet` body expression. As a consequence, when applied, the handler that was active when the function was created is used. Note that the pervasive binding is made available down to the innermost lambda of the division, even though the body of the `plet` expression is not under evaluation any more[3]

With only `let` and `dlet`, this semantics could only be achieved by modifying the body of `init-divide` to re-deploy lexically (with `let`) the binding of `handler` made available with a `dlet`. Also, to be correct, all inner lambdas should reinstate the dynamic binding for their body. Obviously, this intrusive approach does not scale to more complex scenarios. With respect to expressiveness [9], it is easy to see that `plet` is *not* macro-expressible with `let` and `dlet`, as the transformation described implies modifying the code of all lambdas that may potentially create other lambdas under the dynamic extent of the `plet` body as well as these lambdas themselves and so forth transitively—a non-local modification indeed.

Flat binding A flat binding is a binding that is available only in the lexical region where it is deployed, like a lexical binding, with the additional restriction that it is *not* captured in lexical closures. In other words, a flat binding, deployed with `flet`[4], propagates neither on the call stack nor on the delayed evaluation dimensions: it has a scoping strategy $\sigma = \langle false, false \rangle$.

```
(define foo (flet ((x 3))
              (let ((y (+ x 1))
                (lambda (z) (+ x y z)))))
> (foo 1)
unbound identifier: 'x
```

[3] The way we simulate exception handling here is good enough to make our point about the scope of bindings. It corresponds to the low-level layer of exception handling provided in PLT Scheme [10], which does not reinstate the context of the handler expression.

[4] This `flet` construct has nothing to do with Common Lisp's `flet`, which is used for introducing local functions.

scoping strategy	construct	description
$\sigma = \langle false, true \rangle$	`let`	lexical binding
$\sigma = \langle true, false \rangle$	`dlet`	dynamic binding
$\sigma = \langle true, true \rangle$	`plet`	pervasive binding
$\sigma = \langle false, false \rangle$	`flet`	flat binding

Table 2. Four scoping strategies.

In the above example, `x` is bound and can be used to evaluate the value bound to `y`, but it is not capturable by the lambda itself. Therefore, applying the lambda results in an unbound identifier error. This can be useful to enforce the discipline that certain bindings are not meant to be captured by procedural values and used later on, *i.e.* when they may not (or should not) even make sense. `flet` cannot be expressed in terms of the other binding constructs we have seen so far, without a global transformation.

Interestingly[5], `flet` bindings are readily found in the Java programming language, for instance. In Java [12], an anonymous inner class only closes over final variables of the lexical environment, but not over non-final variables. This basically means that in Java all non-final local variables (including method arguments) are actually `flet` bindings. Only final variables have the semantics of statically-scoped bindings in that they get captured appropriately. Here, the motivation is rather one of efficiency, *i.e.* to avoid implicit allocation on the heap, necessary for mutable bindings.

2.3 Summary

Table 2 summarizes the four scoping strategies we have expressed thus far, using boolean values for c and d. The two first alternatives are very well known and correspond to static and dynamic scope, while the two last are more exotic, yet potentially useful, as illustrated previously.

A key point here is changing the point of view, from the dichotomy lexical/dynamic to a propagation perspective: bindings (and more generally, adaptations) are available because they *propagate* from their deployment site to another region (either a called procedure, or a nested procedural value). This can be, and in fact is in our model, a dynamic decision, as will be explained and illustrated later on.

3. Expressive Propagation

The scoping model we have presented in the previous section already goes beyond the lexical/dynamic dichotomy traditionally established in programming languages. But it just extends the dichotomy with two more alternatives. We are interested in pushing the model further, to support more flexible scoping strategies, applicable in more advanced scenarios. From now on, we consider several extensions that allow us to illustrate advanced scoping in module systems and context-oriented programming, among others.

Up to now, the components of a scoping strategy $\sigma = \langle c, d \rangle$ are constants. As a result, if a binding propagates along one dimension, it will always do so. Conversely, if we consider c and d as propagation *functions*, we obtain more flexibility by giving the possibility of subjecting propagation to a dynamic condition.

3.1 Propagation functions

At each function application (resp. function creation), call stack (resp. delayed evaluation) propagation is determined for each currently-available binding. This is done by applying the c (resp. d) function. If no information is provided as parameter to a propaga-

[5] We thank Pascal Costanza for bringing this fact to our attention.

tion function, it can base its decision only on externally-accessible information or on its own state[6]. It therefore becomes possible to express that a binding propagates only a fixed number of times, or whether a certain property holds, such as user preferences.

To illustrate this, we introduce a new binding construct, `slet` (for "scoped let"), parameterized by the scoping strategy (a list containing two propagation functions). That is:

```
(slet (list (lambda () #t) (lambda () #f))
      ((x val)...)
   expr...)
```

is equivalent to `(dlet ((x val)...) expr...)`.

In the example below, we deploy an "energetic binding" for `handler`, *i.e.* a binding whose propagation functions c and d are based on a notion of energy, decreased at each propagation step:

```
(define (energy init step)
  (let ((my-energy init))
    (lambda ()
      (if (< my-energy init) #f
          (begin (set! my-energy (- my-energy step))
                 #t)))))

(slet (list (energy 10 1) (energy 5 1))
      ((handler (lambda (e) ...)))
  ...body...)
```

3.2 Reflective propagation functions

The example above is admittedly artificial. It all becomes more interesting and useful if propagation functions are *reflective* procedures that can access reifications of their evaluation context [11]. In particular, in order to be able to make meaningful decisions about whether or not to propagate, it makes sense to provide a propagation function with a reification of the current execution point that causes propagation. For instance, when a function is applied, a reification of the function application event can be passed to the c function. In line with AOP terminology [13], we will call (the reification of) an execution event a *join point*.

This model of non-constant propagation functions permits one to express advanced strategies by characterizing the join points upon which a binding should stop its propagation. The range of applications allowed by this model depends on the expressiveness of the reifications provided to the propagation functions.

For instance, consider the following program written in a Scheme dialect with a simple module system:

```
(module A
  (define (gee a) ... (write ...) ...))

(module B
  (import A)
  (define (foo x)
    (slet fluid-same-module
          ((output-port my-file-out))
      (bar x))
  (define (bar x)
      (write ...)
      (gee x)))
  (foo 1))
```

Module `A` provides a `gee` function that happens to write to the standard output port. In module `B`, `foo` deploys a fluid binding for the output port, available in `bar`, but *not* available in `gee` because the scoping strategy restricts propagation of the fluid binding to stay within the module boundaries. This is done using `fluid-same-module`, simply defined as:

```
(define fluid-same-module
  (list (lambda (jp)
          (eqv? (f-module (jp-fun jp))
                (f-module (jp-target jp))))
        (lambda (jp) #f)))
```

As can be seen, this only relies on the ability of the join point model to expose the applied function (`jp-fun`), the module in which a function is defined (`f-module`), and the target function of a function call (`jp-target`). Typical information that can be provided in join points are function arity, name, type signature, and actual argument values. This obviously depends on the considered language. For example, in [23] we take advantage of this facility to control propagation of aspects in a distributed system, using reifications of host properties.

4. Beyond Propagation

In the previous sections, we have considered scoping as a *propagation* issue: delimiting the boundaries of where an adaptation such as a variable binding is available. Orthogonal to this issue is the question of whether or not an adaptation is *active* at a certain point in time, *within* the boundaries specified by its propagation.

Thus, as a final refinement, we add a third component to a scoping strategy, called activation function, or filter, denoted f. A scoping strategy is now a triple $\sigma = \langle c, d, f \rangle$, where c and d are propagation functions, as explained before, and f is the activation function.

4.1 Activation function

Activation functions make it possible to express bindings that are available only in certain conditions. For instance, in the following the output port is redirected to `/dev/null` if the program is running in an insecure context at the time the port is written to.

```
;; format the data, then write it
(define (write-data data)
    (write (format data)))

;; process data (writing it at some point)
(define (process data)
  (slet dlet-if-insecure ((output-port
                           (make-output "/dev/null"))
     ...(write-data data)...))))

> (process '(1 2 3)) ;; in secure context
(1 2 3)

> (process '(1 2 3)) ;; in insecure context
```

This example illustrates that activation is orthogonal to propagation. While we are interested in refining the binding of the output port for the dynamic extent of the `process` body, we want this rebinding to be effective only in insecure contexts. Also, coming back to the previous example with `fluid-same-module`: stopping propagation on module boundaries implies that the binding is not available if the control flow comes back within the module (*e.g.* due to a callback). To handle reentrancy, it is necessary to use activation (only active within the module) rather than propagation.

Similarly to propagation functions, an activation function can take any reified information as parameter, such as the current join point. This can be used to discriminate whether a binding is active

[6] More precisely, a propagation function that uses mutable state should rather be named a propagation *procedure*.

based on the join point properties. For instance, it could be that if a function is called from an insecure module, the context is said to be insecure. Therefore, an activation function could reflect upon the current control flow, such as with AspectJ's `cflow` construct. This means we could define `dlet-if-insecure` as follows[7]:

```
(define dlet-if-insecure
  (list (lambda (jp) #t)  ;; dynamic binding prop.
        (lambda (jp) #f)  ;
        (lambda (jp)
          ((cflow not-trusted) jp))))
```

where `not-trusted` is a pointcut defined as:

```
(lambda (jp)
  (not (trusted? (jp-module (jp-fun jp))))))
```

4.2 More reflection

As of now, our reflective abilities have been limited to reifying the current join point stack. In our experiments, we have actually gone further by reifying environments as well. We now use this extra reflective power to express a "soft" binding construct.

By default, we give priority to lexical bindings over dynamic bindings. More precisely, bindings that propagate on the call stack can be shadowed by bindings embedded in a closure. This default is a pragmatic choice that mirrors the fact that, as history taught us, lexical scoping is a more reasonable default than dynamic scoping. However, a (sometimes undesirable) consequence of this default is that it is impossible to dynamically rebind an identifier that has been captured in a closure.

To relax this restriction, we propose a new binding construct, called `soft-let`. A soft lexical binding is captured in a closure, but can be shadowed by a dynamic binding, if any. In other words, it reverses our default environment composition strategy.

```
(define x 0)
(define foo (soft-let ((x 10))
              (lambda () x)))

> (dlet ((x 5))
    (foo))
5

> (foo)
10
```

In the first application of `foo`, there is a dynamic binding available for `x`, so it is used. In the second, note that the definition-time soft binding of `x` is used.

`soft-let` is expressed simply as a scoped binding using an appropriate activation function. The activation function reflects on the current state of the environment. This implies appropriately packaging the environment as a base value that can be manipulated. This process is known as reification [11]. In line with Smith [21], but contrary to Friedman and Wand [11], we provide a data structure reification of environments, in order to allow exploration of the environment structures (ribs or frames). In addition, we provide a set of primitive functions that together constitute a simple reflective API on (reifications of) environments. The API allows one to get to the next frame of an environment `env-next`, and provides searching functions, like `env-check` that checks whether a binding is defined in a given environment. In addition, we introduce *frame marks*, which are constants used to discriminate between bindings coming from different sources:

- `l`: binding just introduced by a let form.
- `p`: binding of a formal parameter.
- `d`: binding captured in a closure.
- `c`: binding propagated on the call stack.

We allow access to marks on reified environments, as well as comparing marks, and an extended version of `env-check` with an extra mark parameter, called `env-check-mark`. The activation function used to express `soft-let` is then defined as follows:

```
(lambda (id)
 (lambda (jp env)
  (not (and (env-eq-mark env 'd)
            (env-check-mark id (env-next env) 'c)))))
```

For a given `id`, this activation function deactivates a binding if *a)* it was captured in a closure, *b)* there exists a binding above the current one, which comes from call stack propagation. Note that the activation function not only receives the current join point, it also receives a reification of the environment where the binding is being found.

To be more generic, we actually define `soft` as a strategy transformer, able to transform any scoping strategy σ into a strategy σ' where the rule above is added to the propagation function. This allows us for instance to provide `soft-plet`, the soft version of the pervasive binding construct introduced in Section 2.2.

There is an interesting use case for `soft-plet`: to define by-default exception handlers. Suppose we deploy an exception handler over the dynamic extent of an expression, in a pervasive manner. This means that a lambda created in that extent will capture the handler, and will therefore be able to use it when applied later on, even if this occurs outside of the original extent. Using `soft-plet`, the binding of the definition-time handler is used only if there is no newer handler deployed.

Consider the same code snippet as used in Section 2.2 to illustrate pervasive bindings, replacing `plet` with `soft-plet`. If we now apply `divide` with another handler deployed, it shadows the definition-time binding:

```
> (dlet ((handler (lambda (e) (write "soft-plet"))))
    ((divide 10) 0))
"soft-plet"
```

5. Semantics

We now formalize the semantics of our proposal of expressive scoping strategies in a call-by-value lambda calculus called Λ_σ. Similarly to the Λ_d language of Moreau [18] we use functions instead of special forms to model binding constructs. That is, $\mathsf{let}\ x\ =\ e_0\ \mathsf{in}\ e$ is desugared as $(\lambda x.e)\ e_0$.

In his language, Moreau supports both functions with statically-scoped parameters and functions with dynamically-scoped parameters, the former denoted $\lambda x.e$ and the latter $\lambda \hat{x}.e$. In our proposal, we do not support only two scoping semantics: the scoping semantics is specified by a scoping strategy σ. Therefore, we use the following syntax for *scoped* lambda abstractions: $\lambda x^\sigma.e$. As a consequence, $\mathsf{slet}(\sigma)\ x\ =\ e_0\ \mathsf{in}\ e$ is syntactic sugar for a scoped lambda application $(\lambda x^\sigma.e)\ e_0$. Figure 1 presents the rest of the syntax of our language Λ_σ.

Because we need to precisely account for the management of bindings in the environment, we make the environment explicit in the evaluation steps, similarly to lambda calculi with explicit substitutions [1, 6]. The syntax of a closure value explicitly includes the captured environment ρ, and is denoted as $\lceil \lambda x^\sigma.e, \rho \rceil$. An environment ρ is a finite mapping from variables to values. The empty environment is denoted ϵ. Bindings in the environ-

[7] For definitions of control flow operators in higher-order languages with aspects, we refer the reader to [7, 22].

$$
\begin{array}{rrcl}
Value & v & ::= & \lceil \lambda x^{\sigma}.e, \rho \rceil \mid \sigma \mid \textbf{true} \mid \textbf{false} \\
 & & & \mid \textbf{bv} \mid \textbf{prim} \\
Expr & e & ::= & v \mid x \mid \lambda x^{\sigma}.e \mid e_1\ e_2 \\
 & & & \mid \langle e1, e2, e3 \rangle \\
Env & \rho & ::= & \epsilon \mid [x \mapsto v]_{\sigma} \mid \rho_1 : \rho_2 \\
ScopeStrat & \sigma & ::= & \langle v_1, v_2, v_3 \rangle
\end{array}
$$

Figure 1. Syntax of the Λ_σ language. Metavariable x ranges over variables.

ment are annotated with their scoping strategy. A scoping strategy is a triple of functions, as explained in the previous sections. Since these functions have to be boolean predicates, the language includes **true** and **false** as values. For instance, the value s of $\langle \lambda x.\textbf{false}, \lambda x.\textbf{true}, \lambda x.\textbf{true} \rangle$[8] is the lexical scoping strategy (recall Table 2). Other (optional) values are basic values, **bv**, which model integers, floats, etc., as well as primitive operations **prim**, which model primitive operations on these different types of values.

5.1 Operational semantics

The operational semantics of our language is defined in Figure 2. The small-step operatinal semantics is specified by a reduction relation $\rightarrow_r$ that reduces terms consisting of an expression and an environment. The judgment $\langle e, \rho \rangle \rightarrow_r \langle e', \rho' \rangle$ says that the pair of expression e and environment ρ takes one step of evaluation to a new pair e' and ρ'. We use $\rightarrow_r^*$ to denote multiple reduction steps at once. Evaluation starts in the empty environment $\langle e, \epsilon \rangle$ and ends either with a value v, or an unbound identifier error.

(abs) When a lambda expression is reduced to a procedural value, the associated scoping strategy must be determined. For conciseness, we use a big step in the definition of the rule, to obtain the strategy σ. The closure captures *part of* its surrounding environment ρ. An environment propagation function $prop_d$ is used in order to select the bindings whose delayed evaluation propagation function d matches. The semantics of $prop_d$ and binding matching are given in Fig. 3 and explained below. ζ is the reified information provided to the propagation function, and is explained in Section 5.3.

(app) When a procedural value (obtained through *(appL)*) is applied to a value (obtained through *(appR)*), its body is evaluated in a particular environment ρ'. This environment is first made up of the bindings of the current environment ρ that propagate on the call stack, obtained using the environment propagation function $prop_c$ (Fig. 3). The resulting environment is then extended by the closure environment ρ_0, as well as the binding of the formal parameter of the function. The scoping strategy for the formal parameter is σ, the strategy specified when the function was defined. Note how the definition of ρ' makes explicit the fact that we give priority to the definition-time environment ρ_0 over the bindings that propagate dynamically with $prop_c$.

(var) When a variable has to be looked up, the most recent binding of that variable is found. If that binding is active, as specified by the activation function f of the scoping strategy, the expression is reduced to the value *(varOk)*. If the binding is not active, lookup proceeds in the rest of the environment *(varOff)*; similarly if the identifier does not match *(varFail)*. Looking up an identifier in the empty environment results in an error *(err)*.

(val) A value in an environment is reduced to a standalone value.

[8] Note that we omit the scoping strategy of the formal parameter of these functions, because they do not use it and their body is a value.

$$
\begin{array}{rl}
\textit{(abs)} & \dfrac{\langle e_\sigma, \rho \rangle \rightarrow_r^* \langle v_1, v_2, v_3 \rangle = \sigma \qquad \rho' = prop_d(\rho, \zeta) \qquad \zeta \in Value}{\langle \lambda x^{e_\sigma}.e, \rho \rangle \rightarrow_r \langle \lceil \lambda x^{\sigma}.e, \rho' \rceil, \rho \rangle} \\[2ex]
\textit{(appL)} & \dfrac{\langle e_1, \rho \rangle \rightarrow_r \langle e_1', \rho \rangle}{\langle e_1\ e_2, \rho \rangle \rightarrow_r \langle e_1'\ e_2, \rho \rangle} \\[2ex]
\textit{(appR)} & \dfrac{\langle e, \rho \rangle \rightarrow_r \langle e', \rho \rangle}{\langle v\ e, \rho \rangle \rightarrow_r \langle v\ e', \rho \rangle} \\[2ex]
\textit{(app)} & \dfrac{\rho' = [x \mapsto v]_{\sigma} : \rho_0 : prop_c(\rho, \zeta) \qquad \zeta \in Value}{\langle \lceil \lambda x^{\sigma}.e, \rho_0 \rceil\ v, \rho \rangle \rightarrow_r \langle e, \rho' \rangle} \\[2ex]
\textit{(varOk)} & \dfrac{match(f, \zeta, [x \mapsto v]_{\sigma}) = \textbf{true} \qquad \zeta \in Value}{\langle x, [x \mapsto v]_{\sigma} : \rho \rangle \rightarrow_r \langle v, \rho \rangle} \\[2ex]
\textit{(varOff)} & \dfrac{match(f, \zeta, [x \mapsto v]_{\sigma}) = \textbf{false} \qquad \zeta \in Value}{\langle x, [x \mapsto v]_{\sigma} : \rho \rangle \rightarrow_r \langle x, \rho \rangle} \\[2ex]
\textit{(varFail)} & \dfrac{x \neq y}{\langle x, [y \mapsto v]_{\sigma} : \rho \rangle \rightarrow_r \langle x, \rho \rangle} \\[2ex]
\textit{(err)} & \langle x, \epsilon \rangle \rightarrow_r \mathsf{Error} \\[2ex]
\textit{(val)} & \langle v, \rho \rangle \rightarrow_r v
\end{array}
$$

Figure 2. Semantics of the Λ_σ language. $prop_c$, $prop_d$ and $match$ are defined in Figure 3. ζ is the reified information provided to scoping strategies (Sect. 5.3).

Figure 3 defines the auxiliary functions used in the semantics of Figure 2. Propagation functions filter the given environment using a binding matching function. The matching function (whose definition is curried) is given a scoping strategy accessor, which can be either c, d, or f (depending on which component of the strategy is of interest). The matching function, given a binding, extracts the component of the scoping strategy of the binding. This component is a closure, whose body is then evaluated in the appropriate environment, until obtaining a value, (hopefully) **true** or **false**. The environment used to evaluate the body of the scoping strategy component is the environment captured by the closure, extended with a binding of the formal parameter to ζ, the reified information. We omit this parameter in the following examples, and come back to ζ in Section 5.3.

5.2 Some reductions

Consider the following program:

```
((let ((x 2))
  (lambda (y) x))
 1)
```

Recall that `let` is syntactic sugar for a function definition and application, and that `lambda` is syntactic sugar for a lambda with a statically-scoped parameter.

Binding matching

$$match :: Accessor \to Value \to Env \to Value$$

$$\begin{array}{rcll} match(\gamma, \zeta, \epsilon) & = & \textbf{false} & \\ match(\gamma, \zeta, [x \mapsto v]_\sigma) & = & v & \text{with } \gamma(\sigma) = \lceil \lambda y^{\sigma'}.e, \rho \rceil \\ & & & \langle e, [y \mapsto \zeta]_{\sigma'} : \rho \rangle \to_r^* v \end{array}$$

Scoping strategy accessors

$$\begin{array}{rcl} Accessor = \{c, d, f\} & \subset & ScopeStrat \to Value \\ c(\langle v_1, v_2, v_3 \rangle) & = & v_1 \\ d(\langle v_1, v_2, v_3 \rangle) & = & v_2 \\ f(\langle v_1, v_2, v_3 \rangle) & = & v_3 \end{array}$$

Environment propagation functions

$$\begin{array}{rcl} prop_c(\rho, \zeta) & \equiv & filter(match(c, \zeta), \rho) \\ prop_d(\rho, \zeta) & \equiv & filter(match(d, \zeta), \rho) \end{array}$$

$$\begin{array}{rcl} filter(pred, \epsilon) & = & \epsilon \\ filter(pred, b : \rho) & = & \begin{cases} b : filter(\rho) & \text{if } pred(b) = \textbf{true} \\ filter(\rho) & \text{otherwise} \end{cases} \end{array}$$

Figure 3. Definition of binding matching, scoping strategies accessors, and environment propagation functions.

That is, the expression above is expressed in Λ_σ (with numbers as **bv**) as:

$$((\lambda x^s.(\lambda y^s.x))\ 2)\ 1$$

where s is the static scoping strategy defined previously:

$$s = \langle \lceil \lambda x.\textbf{false}, \epsilon \rceil, \lceil \lambda x.\textbf{true}, \epsilon \rceil, \lceil \lambda x.\textbf{true}, \epsilon \rceil \rangle$$

The program is evaluated in the empty environment, and is subsequently reduced as follows:

$$\begin{array}{rcl} & & \langle ((\lambda x^s.(\lambda y^s.x))\ 2)\ 1, \epsilon \rangle \\ \textit{(appL)} & \to_r & \langle (\lambda x^s.(\lambda y^s.x))\ 2, \epsilon \rangle\ 1, \epsilon \rangle \\ \textit{(abs)} & \to_r & \langle \langle \lceil \lambda x^s.(\lambda y^s.x), \epsilon \rceil, \epsilon \rangle\ 2, \epsilon \rangle\ 1, \epsilon \rangle \\ \textit{(val)} & \to_r & \langle \langle \lceil \lambda x^s.(\lambda y^s.x), \epsilon \rceil\ 2, \epsilon \rangle\ 1, \epsilon \rangle \\ \textit{(app)} & \to_r & \langle \langle (\lambda y^s.x), [x \mapsto 2]_s \rangle\ 1, \epsilon \rangle \end{array}$$

At this point the *(abs)* reduction is applied. This implies determining ρ', *i.e.* the environment to be captured in the closure. To this end, the environment propagation function is applied $prop_d([x \mapsto 2]_s, \zeta)$, and this implies evaluating $match(d, \zeta, [x \mapsto 2]_s)$ to determine whether the binding should be captured. The d component of the binding is $d(s) = \lceil \lambda x.\textbf{true}, \epsilon \rceil$, whose application yields **true**:

$$\langle \textbf{true}, [x \mapsto \zeta]_s \rangle \to_r \textbf{true}$$

so the binding is collected (by $filter$) in the resulting environment ρ'. The *(abs)* rule applies and reduction proceeds:

$$\begin{array}{rcl} \textit{(abs)} & \to_r & \langle \langle \lceil \lambda y^s.x, [x \mapsto 2]_s \rceil, [x \mapsto 2]_s \rangle\ 1, \epsilon \rangle \star \\ \textit{(val)} & \to_r & \langle \lceil \lambda y^s.x, [x \mapsto 2]_s \rceil\ 1, \epsilon \rangle \\ \textit{(app)} & \to_r & \langle x, [y \mapsto 1]_s : [x \mapsto 2]_s \rangle \\ \textit{(varFail)} & \to_r & \langle x, [x \mapsto 2]_s \rangle \\ \textit{(varOk)} & \to_r & \langle 2, [x \mapsto 2]_s \rangle \\ \textit{(val)} & \to_r & 2 \end{array}$$

The *(varOk)* rules apply because the evaluation of $match(f, \zeta, [x \mapsto 2]_s)$ extracts the third component of s, which always yields **true**.

The crucial point that crystalizes the static scoping semantics is the last *(abs)* reduction above (marked with a $\star$), where the binding $[x \mapsto 2]_s$ is captured in the closure. Things go differnt if at this point we use for instance a flat binding, with `flet`. In Λ_σ: $((\lambda x^f.(\lambda y^s.x))\ 2)\ 1$, where

$$f = \langle \lceil \lambda x.\textbf{false}, \epsilon \rceil, \lceil \lambda x.\textbf{false}, \epsilon \rceil, \lceil \lambda x.\textbf{true}, \epsilon \rceil \rangle$$

Reducing the program now results in an unbound identifier error (omitting the 3 first reductions above, which are the same):

$$\begin{array}{rcl} & & \langle ((\lambda x^f.(\lambda y^s.x))\ 2)\ 1, \epsilon \rangle \\ & & \ldots \\ \textit{(app)} & \to_r & \langle \langle (\lambda y^s.x), [x \mapsto 2]_f \rangle\ 1, \epsilon \rangle \\ \textit{(abs)} & \to_r & \langle \langle \lceil \lambda y^s.x, \epsilon \rceil, [x \mapsto 2]_f \rangle\ 1, \epsilon \rangle \star \\ \textit{(val)} & \to_r & \langle \lceil \lambda y^s.x, \epsilon \rceil\ 1, \epsilon \rangle \\ \textit{(app)} & \to_r & \langle x, [y \mapsto 1]_s \rangle \\ \textit{(varFail)} & \to_r & \langle x, \epsilon \rangle \\ \textit{(err)} & \to_r & \mathsf{Error} \end{array}$$

Wrapping the above program with a dynamic binding for x solves the issue:

```
(dlet ((x 3))
  ((flet ((x 2))
     (lambda (y) x))
   1))
```

Expressed in Λ_σ, with

$$d = \langle \lceil \lambda x.\textbf{true}, \epsilon \rceil, \lceil \lambda x.\textbf{false}, \epsilon \rceil, \lceil \lambda x.\textbf{true}, \epsilon \rceil \rangle$$

the program is reduced as follows:

$$\begin{array}{rcl} & & \langle (\lambda x^d.((\lambda x^f.(\lambda y^s.x))\ 2)\ 1)\ 3, \epsilon \rangle \\ \textit{(appL)} & \to_r & \langle \langle (\lambda x^d.((\lambda x^f.(\lambda y^s.x))\ 2)\ 1), \epsilon \rangle\ 3, \epsilon \rangle \\ \textit{(abs)} & \to_r & \langle \langle \lceil \lambda x^d.((\lambda x^f.(\lambda y^s.x))\ 2)\ 1, \epsilon \rceil, \epsilon \rangle\ 3, \epsilon \rangle \\ \textit{(val)} & \to_r & \langle \lceil \lambda x^d.((\lambda x^f.(\lambda y^s.x))\ 2)\ 1, \epsilon \rceil\ 3, \epsilon \rangle \\ \textit{(app)} & \to_r & \langle ((\lambda x^f.(\lambda y^s.x))\ 2)\ 1, [x \mapsto 3]_d \rangle \\ & & \textit{\ldots same as above, except for last environment} \\ \textit{(app)} & \to_r & \langle \langle (\lambda y^s.x), [x \mapsto 2]_f \rangle\ 1, [x \mapsto 3]_d \rangle \\ \textit{(abs)} & \to_r & \langle \langle \lceil \lambda y^s.x, \epsilon \rceil, [x \mapsto 2]_f \rangle\ 1, [x \mapsto 3]_d \rangle \star \\ \textit{(val)} & \to_r & \langle \lceil \lambda y^s.x, \epsilon \rceil\ 1, [x \mapsto 3]_d \rangle \\ \textit{(app)} & \to_r & \langle x, [y \mapsto 1]_s : [x \mapsto 3]_d \rangle\ \star\star \\ \textit{(varFail)} & \to_r & \langle x, [x \mapsto 3]_d \rangle \\ \textit{(varOk)} & \to_r & \langle 3, [x \mapsto 3]_d \rangle \\ \textit{(val)} & \to_r & 3 \end{array}$$

The closure still does not capture a binding for x (see $\star$). However, in the following application reduction ($\star\star$), the dynamic binding of x propagates and is therefore available in the final function body evaluation. This is determined in the premises of the *(app)* rule where the $prop_c$ environment propagation function is applied, resulting in the evaluation of the c component of the dynamic binding strategy, which yields **true**.

5.3 Expressive scoping

In Figures 2 and 3, propagation and activation functions are given an argument ζ that we intentionally left undefined. We have only specified that it is a value $\in Value$. This argument corresponds to the reification of the execution context that is provided to these (base-level) functions [11]. In the end, ζ depends on the reflective power given to scoping strategies. It could simply be void (as in Section 3.1), or a reification of the current execution event and call stack (as in Section 3.2), or additional reified information, such

as a reification of the environment structure itself, as discussed in Section 4.2. This is why we refrain from specifying it further in the general case.

For instance, to provide a reification of the call stack and current execution event, we could extend our semantics to build the join point stack as evaluation progresses. In the small-step operational semantics, we would have to add join points at each step of the evaluation: $\langle e, \rho, jp\rangle \rightarrow_r \langle e', \rho', jp'\rangle$. Then $\zeta = jp$.

Similarly in Section 4.2 the scoping strategy makes use of the environment, so in the semantics we could use $\zeta = \rho$. More precisely, to define a "soft" let construct, we need frame marks. Therefore, we have to extend our syntactic representation of environments to include a mark ω: $[x \mapsto v]^{\omega}_{\sigma}$, where $\omega \in \{\mathsf{p}, \mathsf{c}, \mathsf{d}\}$. As explained in Sect. 4.2, a mark indicates whether a binding is in the environment due to (respectively) parameter binding, call stack propagation, or delayed evaluation propagation. These marks are then set in the corresponding places: for instance, the *(app)* rule should mark with p (parameter) the binding of the formal parameter of the applied function.

5.4 Implementation

We have implemented Λ_σ using a Scheme interpreter. In contrast to the definition of Λ_σ, we have maintained let forms in the language in order to ease debugging. Also, the interpreter supports multiple arguments to both procedures and let forms. Therefore, the environment is rather made up of binding *frames*, instead of chained single bindings. Annotations (scoping strategy, mark) are done at the level of frames themselves. It supports reification of join points as well as environment, making it possible to implement the examples of Section 4.2. The different let forms introduced in this paper, like `dlet`, `plet`, `flet`, `soft-plet`, etc., are all defined as syntactic sugar on top of the generic scoped let (`slet`) form. The interpreter as well as the examples can be obtained from `http://pleiad.cl/research/scope`.

6. Other Instantiations

Since the static/dynamic scoping dichotomy has pervaded many areas of language design, there are as many areas in which the flexible scoping model we propose can be instantiated. We hereby discuss the case of aspects and mixin layers. We also briefly present an experience report on providing scalable omniscient debugging, which implied using advanced scoping semantics.

6.1 Aspects

In a higher-order aspect-oriented procedural language like AspectScheme [7], pointcuts and advices are first-class values. AspectScheme introduces an expression to dynamically deploy an aspect over a body expression. Thus scoping considerations appear: it is necessary to define the precise extent of the jurisdiction of the aspect, *i.e.* what join points it will see. AspectScheme builds upon the familiar reasoning of scope for variable bindings. Therefore, following the static/dynamic dichotomy, it introduces two aspect deployment expressions:

- `fluid-around` deploys a dynamically-scoped aspect. Such an aspect sees all join points occurring in the dynamic extent of its body. This is essentially the same mechanism as a `deploy` expression in CaesarJ [2].
- `around` deploys a statically-scoped aspect. Such an aspect only sees join points occurring lexically in its body, *including* those of unapplied functions, which are exported from the body.

```
(let ((apply-to-orlando (lambda (f) (f "orlando"))))
  (fluid-around (call open-file) trace
    (apply-to-orlando open-file)))
```

Above, the `let` defines and binds a higher-order function that takes another function `f` as parameter and applies it to `"orlando"`; then, we dynamically deploy an aspect consisting of a `trace` advice and a call pointcut. Then we apply the function `apply-to-orlando` to `open-file`. When `open-file` is finally applied (the framebox above), the trace aspect *does see* the join point, because the application occurs in the dynamic extent of the `fluid-around` body expression. Conversely, in:

```
(let ((traced-open (fluid-around (call open-file) trace
                     (lambda (f) (open-file f)))))
  (traced-open "orlando"))
```

The aspect does *not* apply, because the dynamic extent of the aspect body only consists of a function definition. The (framed) application of `open-file` is only in the lexical scope of the `fluid-around` body expression: later applications are out of reach. Statically-scoped aspects serve exactly this purpose:

```
(let ((traced-open (around (call open-file) trace
                     (lambda (f) (open-file f)))))
  (traced-open "orlando"))
```

The aspect applies, because the application of `open-file` occurs lexically in the body of `around`. As a consequence, the aspect is "engrained" in the function that is exported from the `around` expression and bound to `traced-open`. At future applications of the function, the aspect will see the corresponding join point even though the application occurs outside of the dynamic extent of the `around` expression.

The scoping model we propose in this paper was first conceived in the context of aspect-oriented programming, precisely as an extension to the too rigid scoping model of AspectScheme and others [22]. We introduce a single aspect deployment expression parametrized by a scoping strategy, and illustrate its use in a number of applications. In particular, we take advantage of types in an object-oriented language to express advanced scoping semantics. Further, in [23], we show that the scoping model scales to distributed systems: we are able to express distributed scoping strategies without changing the model itself. It is enough to reify more information (such as the host properties of the target of a remote call) in order to be able to address distribution requirements. We refer the reader to the mentioned publications for more details.

6.2 Layers

Since the early days of class-based programming, there have been proposals to allow class extensions. Recently, a number of proposals have be formulated, that allow class extensions to *not* have global scope, but rather be effective only in certain conditions. The scoping semantics follow the static/dynamic dichotomy.

On the one hand, Classboxes [3] and Expanders [25] both introduce *statically-scoped* class extensions. In these proposals, a class extension is only effective in (statically) well-defined regions of code, such as client modules that "import a refinement". On the other hand, ContextL [5] introduces the idea of activating class extensions only in the dynamic extent of the body of `with-active-layer` expressions. For instance, the following defines a layered class (*i.e.* an extensible class) `person`, a layer `employment-layer`, and refines method `display` in that layer:

```
(define-layered-class person ()
 ((name :initarg :name
        :accessor person-name)))

(deflayer employment-layer)
```

```
(defmethod display :in-layer employment-layer
  ((object person))
  ...)
```

Then, a layer can be deployed on an expression:

```
(with-active-layers (employment-layer)
  ...body...)
```

During the evaluation of `body`, `display` messages sent to `person` objects make use of the definition done in the employment layer. Therefore, this deployment follows the discipline of dynamic scoping. But interestingly, the need to "stop" propagation along the call stack axis has been clearly identified: ContextL supports a `with-unactive-layer` construct to deactivate a layer.

Layer deactivation in ContextL has to be done explicitly at all required sites: it is not possible to specify, at layer *deployment time*, a condition upon which a layer must be deactivated. Therefore, a language like ContextL could benefit from supporting scoping strategies. It would be sufficient to support a parametrized layer deployment expression $\mathsf{withlayer}(\sigma)\ l\ \mathsf{in}\ e$ that would then mimic the semantics we have exposed (for variable bindings here, and for aspects in [22]). This would allow several enhancements beyond deployment-time specification of when a layer should be withdrawn. First, the activation function could be used to support context adaptation of the layer environment (for instance to simply deactivate a layer when the battery level of a device becomes low). In addition, delayed evaluation propagation control could be used to partially address the "escaping closure proceed" problem [4], by specifying which layers are captured by (which) closures, and therefore reinstated upon closure application.

6.3 Omniscient Debugging

An omniscient debugger [19] is a debugger that allows a developer to go both backward and forward in time. It permits exploration of the execution history through causal links, answering questions such as *"why is this field bound to null at that point in time?"*. An obvious challenge for omniscient debugging, because it relies on having the whole execution history at hand, is scalability [20]. Among the solutions to address the scalability issue, the TOD omniscient debugger for Java supports *partial traces*[9]. The idea of partial traces is that certain parts of a system are deemed robust and therefore execution of these parts should not (always) be recorded. As we show in [20], this can lead to huge performance benefits, as well as to a reduction of the information overload typically associated with trace-based approaches.

TOD records execution events in a custom format into a specialized database backend. The generation of events is obtained by adapting the debugged program (through bytecode transformation). Monitored code is defined statically, using name- and package-based selection of classes whose execution should be monitored.

However, it turns out that there are special cases for which static scoping is not enough, such as `ArrayList`, `HashMap` and `HashMap.Entry`. These library classes are typically not monitored (for efficiency and bootstrapping reasons), but if some instances are created by monitored code, they should be monitored too. Also, `HashMap.Entry` instances created by monitored hash maps must be monitored. Therefore, some kind of instance-level scoping is required, and it depends on the context in which an instance is created. In the current implementation of TOD, these classes are made to declare a special instance variable that indicates whether an instance has been created by monitored code or not. When a method executes, the instance variable is checked, ensuring that execution of instances created by monitored code is monitored as well.

[9] TOD is available at `http://pleiad.cl/tod`

In addition, for these special classes, monitoring is limited to certain operations, typically just array and field writes. This enhances performance without sacrificing usefulness as one is typically interested in the contents of these structures and not in the details of their operation (such as the fact that obtaining a value in a hash map meant going forward three slots for the corresponding hash key).

As it turns out, the scoping requirements of monitoring fit perfectly in the model of scoping strategies we propose:

- **Call stack propagation.** Monitoring is an adaptation that is deployed over the dynamic extent of the entry point of a program. This means that all events are potentially monitored, and going through code that is not monitored (because of deactivation) does not stop propagation of monitoring (*e.g.* if control flows from non-monitored code to monitored code).
- **Delayed evaluation propagation.** When monitored code creates instances of *e.g.* `ArrayList` and `HashMap`, and when monitored `HashMap` instances create `HashMap.Entry` objects, monitoring propagates to these newly-created objects[10].
- **Activation.** Monitoring is active only for code in static scope. The only exception to this rule are events occurring in objects into which the adaptation has been engrained (according to the delayed evaluation propagation requirement above). This is a kind of "soft deactivation", similar to the soft binding constructs we have discussed. In addition, for these cases, the activation function filters out events that are not write events.

All these requirements are currently implemented in TOD in an ad hoc manner. Activation is partially evaluated in the sense that the corresponding code does not get instrumented at all. This is similar to partial evaluation of pointcuts in aspect languages [17]. Soft deactivation is hand coded by flags that are added to classes and checked at runtime. The flags are actually a case-specific implementation of our environment marks of Section 4.2. Delayed evaluation propagation is also hand coded by controlling instantiation of the concerned classes to properly initialize the flags.

The case of TOD is therefore interesting for two reasons. First, because it is a concrete case of advanced scoping needs in a real world application (TOD can be used to debug Eclipse itself, for instance). Second, because it is an example in which the scoping strategy components are fully known prior to execution, and can therefore be partially evaluated. While for now this partial evaluation is done in an ad hoc manner in TOD, this may give us insights on how to optimize our model in a more general setting.

7. Related Work

Lexical and dynamic scoping have been described, formalized and implemented in many different ways since the early ages of programming languages. We do not even attempt at a correct historical account here. Rather, we briefly discuss approaches that are related to the idea of "controlling" lexical and dynamic scoping in some way.

As mentioned before, Java [12] is an example of a programming language where the programmer is given the means to specify whether a binding should be captured by an anonymous inner class or not: a binding is captured only if it is `final`. The motivation for this decision is primarily (memory) efficiency. One can see a final binding as a truly lexically-scoped binding, while other bindings are actually "flat" in the sense of our flat binding construct of Section 2.2.

[10] As discussed in greater details in [22], object creation can be seen (for the sake of scoping) as the OO equivalent of lambda creation.

Apart from that, to the best of our knowledge, in languages that support lexical scoping, bindings are always captured in procedural values, and dynamic bindings always propagate on the call stack. We can however identify two related pieces of work in slightly different contexts: continuations, and dynamic mixin layers.

With respect to capture, there is a large body of work in the area of delimited control, starting from Felleisen [8] up to more recent developments [16, 10]. In a language where continuations can be captured, the need to only capture *part* of the dynamic environment in a continuation, rather than all or none of it, has appeared. Motivations for this feature are many, among them an important one is encapsulation of control and control-associated state (such as dynamic bindings), for instance for security.Delimiting which part of a continuation has to be captured is achieved by placing *prompts* on the continuation stack, such that a continuation captures the current environment only up to the closest (possibly tagged) prompt. In our proposal, which bindings are being captured is defined at the time the binding is deployed. With continuations, what is "deployed" is the prompt, which actually serves a similar purpose. Tagged prompts add a bit of discriminative power to the language. In our proposal, we do not use tags, but rather propagation functions. A similar (and unusual) design for delimited continuation would consist in placing a "prompt function" that upon continuation capture, decides whether the corresponding subpart of the control context should be captured.

With respect to controlling the propagation of dynamic bindings, it is worth mentioning that some proposals have found it necessary to offer an "undeploy" mechanism in addition to deployment. Typically, a binding is only not in effect in a sub-region of execution if there exists a newer one that shadows it. In a language like ContextL [5], however, there is a mechanism to undeploy a layer for a certain dynamic extent. It is however impossible, at layer *deployment time*, to specify a condition upon which the layer must be either undeployed or deactivated. This has to be done explicitly at all required sites.

Together with Clarke and Costanza, the author has recently looked at a particular issue in higher-order languages with dynamic layers [4]. In such a language, it is possible for a closure to escape its layer environment and applied at a later point, in a different layer environment. In that case, the meaning of `proceed`ing to the "next" layer becomes ill-defined. The issue bears some similarity to the issue of delimited control. In [4] we explain how (a simplified version of) the scoping model presented here can be used to provide fine-grained control over which layers are "sticky" to closures, *i.e.* are captured and later reinstated when the closure is applied. This is therefore another concrete issue for which our model has some contribution to make.

Dezani-Ciancaglini, Giannini and Nierstrasz recently proposed a calculus of evolving objects [6]. The calculus allows for part of a function's environment to be provided at function application time, thereby achieving dynamic binding. An interesting conditional expression is introduced, analogous to a try-catch expression. The idea is that if an expression is evaluated in an environment that does not supply all the necessary bindings for it, then an alternative expression is evaluated instead. The authors then develop a sound type system for this calculus. While no provisions are made to control the propagation of dynamic bindings, we believe this mechanism could be useful if used in conjunction with our scoping model, in order to avoid runtime errors due to unbound identifiers.

8. Conclusions

We have argued that changing the point of view on scoping, from a textual/dynamic dichotomy to a propagation and activation problem, allows us to formulate a general model of scoping that goes well beyond static and dynamic scope as traditionally explored. We have formulated our model in the traditional context of variable bindings, illustrating various usage scenarios, as well as describing its formal semantics in the Λ_σ language. We have also highlighted that this scoping model has applications in various related areas, like context-oriented programming, aspect-oriented programming, and program monitoring.

In this paper, the formalization of Λ_σ has only been used as a descriptive medium, to abstract away from the Scheme interpreter we have built; we are currently looking at how to prove certain properties of Λ_σ. In addition, there are many tracks open for future exploration, ranging from static checking and analysis, integration into existing programming languages, optimized implementations, and applications in other contexts.

Acknowledgments. For invaluable feedback on different aspects of this work, we thank: Jonathan Aldrich, Dave Clarke, Pascal Costanza, Robby Findler, Oscar Nierstrasz and Guillaume Pothier.

References

[1] Martin Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1992.

[2] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An overview of CaesarJ. In *Transactions on Aspect-Oriented Software Development*, volume 3880 of *Lecture Notes in Computer Science*, pages 135–173. Springer-Verlag, February 2006.

[3] Alexandre Bergel, Stéphane Ducasse, and Roel Wuyts. Classboxes: A minimal module model supporting local rebinding. In *Proceedings of the Joint Modular Languages Conference (JMLC'03)*, volume 2789 of *Lecture Notes in Computer Science*, pages 122–131. Springer-Verlag, 2003.

[4] Dave Clarke, Pascal Costanza, and Éric Tanter. How should context-escaping closures proceed? In *Workshop on Context-Oriented Programming*, Genova, Italy, July 2009.

[5] Pascal Costanza and Robert Hirschfeld. Language constructs for context-oriented programming – an overview of ContextL. In *ACM Dynamic Language Symposium (DLS 2005)*, San Diego, CA, USA, October 2005.

[6] Mariangiola Dezani-Ciancaglini, Paola Giannini, and Oscar Nierstrasz. A calculus of evolving objects. *Scientific Annals of Computer Science*, 18:63–98, 2008.

[7] Christopher Dutchyn, David B. Tucker, and Shriram Krishnamurthi. Semantics and scoping of aspects in higher-order languages. *Science of Computer Programming*, 63(3):207–239, December 2006.

[8] Matthias Felleisen. The theory and practice of first-class prompts. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL'88)*, pages 180–190. ACM Press, 1988.

[9] Matthias Felleisen. On the expressive power of programming languages. *Science of Computer Programming*, 17:35–75, 1991.

[10] Matthew Flatt, Gang Yu, Robert Bruce Findler, and Matthias Felleisen. Adding delimited and composable control to a production programming environment. In *Proceedings of the 12th ACM SIGPLAN Conference on Functional Programming (ICFP 2007)*, pages 165–176, Freiburg, Germany, October 2007. ACM Press.

[11] Daniel P. Friedman and Mitchell Wand. Reification: Reflection without metaphysics. In *Proceedings of the Annual ACM Symposium on Lisp and Functional Programming*, pages 348–355, August 1984.

[12] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, 3rd edition*. Addison-Wesley, 2005.

[13] G. Kiczales, J. Irwin, J. Lamping, J. Loingtier, C.V. Lopes, C. Maeda, and A. Mendhekar. Aspect oriented programming. In *Special Issues in Object-Oriented Programming*. Max Muehlhaeuser (general editor) et al., 1996.

[14] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. An overview of AspectJ. In Jorgen L. Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001)*, number 2072 in Lecture Notes in Computer Science, pages 327–353, Budapest, Hungary, June 2001. Springer-Verlag.

[15] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina V. Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP 97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyväskylä, Finland, June 1997. Springer-Verlag.

[16] Oleg Kiselyov, Chung chieh Shan, and Amr Sabry. Delimited dynamic binding. In *Proceedings of the 11th ACM SIGPLAN Conference on Functional Programming (ICFP 2006)*, pages 26–37, Portland, Oregon, USA, September 2006. ACM Press.

[17] Hidehiko Masuhara, Gregor Kiczales, and Christopher Dutchyn. A compilation and optimization model for aspect-oriented programs. In G. Hedin, editor, *Proceedings of Compiler Construction (CC2003)*, volume 2622 of *Lecture Notes in Computer Science*, pages 46–60. Springer-Verlag, 2003.

[18] Luc Moreau. A syntactic theory of dynamic binding. *Higher-Order and Sympolic Computation*, 11(3):233–279, 1998.

[19] Guillaume Pothier and Éric Tanter. Back to the future: Omniscient debugging. *IEEE Software*, 2009. To appear.

[20] Guillaume Pothier, Éric Tanter, and José Piquer. Scalable omniscient debugging. In *Proceedings of the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2007)*, pages 535–552, Montreal, Canada, October 2007. ACM Press. ACM SIGPLAN Notices, 42(10).

[21] Brian C. Smith. Reflection and semantics in Lisp. In *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 23–35, January 1984.

[22] 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.

[23] 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.

[24] Mitchell Wand, Gregor Kiczales, and Christopher Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Transactions on Programming Languages and Systems*, 26(5):890–910, September 2004.

[25] Alessandro Warth, Milan Stanojević, and Todd Millstein. Statically scoped object adaptation with expanders. In *Proceedings of the 21st ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2006)*, pages 37–55, Portland, Oregon, USA, October 2006. ACM Press.