How should context-escaping closures proceed?

Dave Clarke *
DistriNet
Katholieke Universiteit Leuven
B-3001 Heverlee, Belgium
dave.clarke@cs.kuleuven.be

Pascal Costanza †
Programming Technology Lab
Vrije Universitiet Brussel
B-1050 Brussels, Belgium
pascal.costanza@vub.ac.be

Éric Tanter ‡
PLEIAD Laboratory
Computer Science Dept (DCC)
University of Chile, Santiago, Chile
etanter@dcc.uchile.cl

Abstract

Context-oriented programming treats execution context explicitly and provides means for context-dependent adaptation at runtime. This is achieved, for example, using dynamic layer activation and contextual dispatch, where the context consists of a layer environment of a stack of active layers. Layers can adapt existing behaviour using proceed to access earlier activated layers. A problem arises when a call to proceed is made from within a closure that escapes the layer environment in which it was defined. It is not clear how to interpret proceed when the closure is subsequently applied in a different environment, because the layers it implicitly refers to (such as the original layer and/or the remaining layers) may no longer be active. In this paper, we describe this problem in detail and present some approaches for dealing with it, though ultimately we leave the question open.

1. Introduction

An approach to context-oriented programming as embodied by ContextL and others [4] enables programmers to dynamically adapt or replace code at run-time by activating layers which intercept dispatched methods. Code is organised into modules called layers which cross-cut existing classes. Previously activated methods can be invoked using a command proceed, and thus layers can be used to implement a variant of *before*, *after*, and *around* advice. proceed provides delegation within the stack of layers, analogous to super calls in object-oriented languages, 'proceed' in aspect languages, or delegation in general, in that proceed goes through the remaining layers, though new method invocations go to the most recently activated layer.

The structure of the layer environment changes dynamically as layers are activated and deactivated. This can cause problems in higher-order languages, because a closure containing a call to proceed can escape the layer environment in which it originates (and makes sense) and be applied in a new layer environment. It is unclear what proceed means, nor what it should mean, in the new environment.

1.1 The proceed Problem

So far, there seem to be two basic approaches for providing semantics for proceed, and in the following we use ContextJ* and ContextL as two representative context-oriented languages to illustrate the two approaches.

ContextJ* [4], which is based on Java, implements its extension of method dispatch as a search through the current stack of active layers. This suggests that proceed is *the continuation of a search*. Currently, this is not a problem because like Java, ContextJ* does not have first-class closures.[1] Therefore, this search can *always* continue in the current layer environment, at the position in the stack of active layers where it found the currently executing method. However, if ContextJ* had closures, and invocations of proceed could be captured in such closures, these semantics suggest that the search has to continue in an as-yet unspecified stack of layers, and it is unclear which layers to use: the original layers, the currently active layers, or some combination thereof.

ContextL [3] is based on Common Lisp, which provides lexical closures. However, in contrast to ContextJ*, an invocation of proceed[2] is *not* the continuation of a search, due to the semantics of generic functions in the Common Lisp Object System (CLOS, [1]), on which ContextL is based. Generic function invocation, CLOS's equivalent to object-oriented message sending, is performed in three steps:[3]

* Author partially funded by the EU project IST-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Methods (`http://hats-project.eu`).

† Author partially funded by the Research Foundation – Flanders (FWO).

‡ Author partially funded by FONDECYT projects 11060493 and 1090083.

COP'09 July 7, 2009, Genova, Italy

[1] Java's recent incarnation of closure-like constructs are still not closures.

[2] In CLOS, proceed is actually `call-next-method`.

[3] We have simplified the actual semantics somewhat for clarity here.

1. A set of applicable methods is determined, based on the classes the candidate methods are defined for. (In ContextL, the current stack of active layers is also taken into account in this step.)
2. The set of applicable methods is sorted according to specificity. (In ContextL, more recently activated layers render their methods more specific.)
3. The most specific method is invoked, and each method can invoke proceed to call the next most specific method.

The intuition here is that proceed merely navigates through a precomputed, ordered set of applicable methods, just like super calls in other object-oriented languages merely invoke methods in the statically determined direct superclass. A change in the current stack of active layers does not affect this set of applicable methods anymore: Once it is determined, it will remain fixed. This gives a reasonable semantics for proceed, even when captured in closures.

However, the fact that ContextL seems to have reasonable semantics here is actually an accident, due to the reuse of CLOS semantics. In fact, ContextL still fails to reinstate the original stack of active layers when proceed is invoked. So neither the layers that invoked the method in which proceed was captured, nor the layers that provide the definitions of the methods that will be invoked by proceed, may actually be active when that proceed is eventually executed. However, especially the already executed definitions may rely on the presence of their own layers in such a situation.

1.2 How to proceed?

To summarize, this paper tries to pose, and partially answer, the following questions:

- Should the invocation of a proceed find a method either in the stack of layers that was active when it was captured in a closure, in the stack of currently active layers, or in a combination thereof?
- Should the method found by proceed then be executed in a dynamic environment with the original stack of layers, with the current stack of layers, or a combination thereof?
- If the layer environments should be combined in these cases, what should the composition look like?

We also briefly discuss ideas for advanced language constructs to influence the composition of layer environments.

2. A Small COP Language

In order to precisely describe the problem and potential solutions, we adopt the formal calculus Contextλ developed by Clarke and Sergey [2], which derives from the semantics of dynamic binding [5, 6]. Contextλ extends the lambda calculus with layer definitions, layer activation ($\mathsf{with}(l)e$) and deactivation ($\mathsf{without}(l)e$), and contextual dispatch. As layer deactivation is not required for our story, we remove without, resulting in a considerably simpler calculus, which we call Contextλ^-. Its syntax is as follows:

$$\begin{array}{rcl} P & ::= & \Delta; e \\ \Delta & ::= & \overline{l = B} \\ B & ::= & \overline{p = e} \\ v & ::= & \lambda x.e \mid x \\ e & ::= & v \mid p \mid e\ e \mid \mathsf{with}(l)e \\ E[\,] & ::= & [\,] \mid E[[\,]\ e] \mid E[v\ [\,]] \mid E[\mathsf{with}(l)[\,]] \end{array}$$

A program P consists of a collection of layers Δ followed by a single expression e. Each layer is a mapping from the layer name l to a set of bindings of parameters p to expressions e. Values v—the results of successful reductions—consist of lambda abstractions and variables. In closed expressions, the only values are lambda abstractions $\lambda x.e$, though for examples we also use integers.

Expressions e consist of values v, parameters p, function application $e\ e$, and context activation $\mathsf{with}(l)e$. Parameters p are dynamically scoped and dynamically bound to a definition in some layer—called *contextual dispatch* or just dispatch. Variables, however, are lexically scoped. The construct $\mathsf{with}(l)e$ activates layer l for the (dynamic) duration of the reduction of expression e, i.e., until e reduces to a value.

Finally, $E[\ ]$ gives the syntax of evaluation contexts, which are expressions with a single hole. E can be seen as holding the context surrounding an expression, in particular, it contains a *stack* of $\mathsf{with}(l)$ expressions denoting the activated layers. For example, $E[\ ] = \mathsf{with}(l_1)(\mathsf{with}(l_2)([\,]\ e)))$ can be seen as the stack of layers $l_1 : l_2$, where l_2 is the innermost layer. Dispatch operates by searching for a binding of a parameter moving outwards from the innermost layer. This stack of layers is called the *layer environment*.

The reduction rules use the following function to determine the set of parameters active in the layer environment, so which layer to dispatch to for a particular parameter:

Bound parameters.

$$\begin{array}{rcl} \mathsf{BP}([\,]) & = & \emptyset \\ \mathsf{BP}(E[[\,]\ e]) & = & \mathsf{BP}(E) \\ \mathsf{BP}(E[v\ [\,]]) & = & \mathsf{BP}(E) \\ \mathsf{BP}(E[\mathsf{with}(l)[\,]]) & = & \mathsf{BP}(E) \cup \mathrm{dom}(\Delta(l)) \end{array}$$

The reduction rules for Contextλ^- are as follows:

Semantics.

$$\begin{array}{rcll} E[(\lambda x.e)\ v] & \to & E[e\{^v/_x\}] & (\beta) \\ E[\mathsf{with}(l)v] & \to & E[v] & (\textsc{Esc}) \\ E[\mathsf{with}(l)E'[p]] & \to & E[\mathsf{with}(l)E'[e]] & (\textsc{Disp}) \\ & & \text{if } p \notin \mathsf{BP}(E') & \\ & & \text{and } e = \Delta(l)(p) & \end{array}$$

Evaluation is call-by-value. The first rule is the standard β-reduction rule. We use $e\{^{e'}/_x\}$ to denote the substitution of x for e' in e. The second rule states that when an expression

finishes evaluating, the surrounding layer activation has no further effect and is removed. The third rule covers the case of looking up a parameter in some surrounding layer. Here $E'[\,]$ is the (inner) part of the evaluation context which does not contain a binding for parameter p, denoted by $p \notin \mathsf{BP}(E')$. Thus, l is the first layer, from inside to outside, containing a binding for p.

A program $P = \Delta; e$ evaluates by evaluating e, where Δ provides the layers.

3. The Problem

The question we wish to raise and study here is what happens when proceed is added to Contextλ^-. The intention of proceed is to delegate the parameter dispatch to the next surrounding layer. When combined with closures, proceed may be invoked in the body of a closure that has escaped from its definition context and is applied in another layer environment. It is then not clear which layer proceed refers to.[4]

Following is an informal account of how proceed behaves, firstly when no closures are present, and then we indicate what the potential problem is when closures are present.

Let

$$\Delta_0 \quad = \quad \left\{ \begin{array}{l} l_1 = \{p = 10\}, \\ l_2 = \{p = \mathsf{proceed} + 5\} \end{array} \right\}$$

The next example illustrates how proceed works, in this case by marking proceed to indicate which parameter to dispatch to, and which layer to begin the search (we use $\mathsf{with}(l_1, l_2, l_3)[\,]$ to denote $\mathsf{with}(l_1)\mathsf{with}(l_2)\mathsf{with}(l_3)[\,]$):

$$\begin{array}{rl} & \mathsf{with}(l_1, l_2)(p + 3) \\ & \qquad \left\{ \begin{array}{l} \textit{Binding for } p \textit{ found in } l_2. \\ \textit{Mark } \mathsf{proceed} \textit{ to indicate} \\ \textit{where to continue search.} \end{array} \right\} \\ \rightarrow & \mathsf{with}(l_1, l_2)((\mathsf{proceed}_{l_1,p} + 5) + 3) \\ & \qquad \left\{ \textit{ Binding for } p \textit{ found in } l_1 \right\} \\ \rightarrow & \mathsf{with}(l_1, l_2)((10 + 5) + 3) \\ \rightarrow^* & 18 \end{array}$$

This naïve approach falls over when a closure containing a call to proceed escapes the layer environment in which it was created. In this example we use:

$$\Delta_1 = \left\{ \begin{array}{lcl} l_1 & = & \{p = \lambda x.1\} \\ l_2 & = & \{p = \lambda x.\mathsf{proceed}\ x\} \\ l_3 & = & \{p = \lambda x.3\} \end{array} \right\}.$$

[4] Note that although Clarke & Sergey present two languages—Contextλ with layers and closures, but not proceed; and ContextFJ with layers, classes and proceed, but no closures [2]—they do not consider this issue.

$$\begin{array}{rl} & (\lambda f.\mathsf{with}(l_3)(f\ 10))\ (\mathsf{with}(l_1, l_2)p) \\ & \qquad \left\{ \begin{array}{l} \textit{Binding for } p \textit{ found in } l_2. \\ \textit{Mark } \mathsf{proceed} \textit{ to indicate} \\ \textit{where to continue search.} \end{array} \right\} \\ \rightarrow & (\lambda f.\mathsf{with}(l_3)(f\ 10))\ (\mathsf{with}(l_1, l_2)(\lambda x.\mathsf{proceed}_{l_1,p}\ x) \\ \rightarrow^* & (\lambda f.\mathsf{with}(l_3)(f\ 10))\ (\lambda x.\mathsf{proceed}_{l_1,p}\ x) \\ \rightarrow & \mathsf{with}(l_3)((\lambda x.\mathsf{proceed}_{l_1,p}\ x)\ 10) \\ \rightarrow & \mathsf{with}(l_3)(\mathsf{proceed}_{l_1,p}\ 10) \\ & \qquad \left\{ \textit{ Cannot find layer } l_1 \textit{ to begin search.} \right\} \\ \rightarrow & ??? \end{array}$$

This naïve approach of recording where to continue the search fails, when the search continues in a new layer environment. In addition, if the layer l_1 had been present in the environment, this could be purely coincidental.

4. Semantics of proceed

We now present a few approaches to giving semantics to proceed. Without closures, they are all equivalent. For our example with Δ_0, the same program always reduces to 18. They do however differ when involving escaping closures.

4.1 Capture and Reinstate Layer Environment

In this semantics, whenever a parameter is dispatched the layer environment above the layer in which the binding is found is captured; proceed is replaced by a call to p in the captured layer environment. The reduction rule is:

$$E[\mathsf{with}(l)E'[p]] \quad \rightarrow \quad E[\mathsf{with}(l)E'[e\{^{\overline{E[p]}}/_{\mathsf{proceed}}\}]] \quad \text{if } p \notin \mathsf{BP}(E') \text{ and } e = \Delta(l)(p) \qquad (\textsc{Disp2})$$

where $\overline{E[\,]}$, which extracts the surrounding layer environment from an evaluation context, is defined as:

$$\begin{array}{rcl} \overline{[\,]} & = & [\,] \\ \overline{E[[\,]\ e]} & = & \overline{E[\,]} \\ \overline{E[v\ [\,]]} & = & \overline{E[\,]} \\ \overline{E[\mathsf{with}(l)[\,]]} & = & \overline{E}[\mathsf{with}(l)[\,]] \end{array}$$

The following example reduction is in the context of Δ_1:

$$\begin{array}{rl} & (\lambda f.\mathsf{with}(l_3)(f\ 10))\ \mathsf{with}(l_1, l_2)p \\ & \qquad \left\{ \textit{ Binding for } p \textit{ found in } l_2 \right\} \\ \rightarrow & (\lambda f.\mathsf{with}(l_3)(f\ 10)) \\ & \qquad \mathsf{with}(l_1, l_2)((\lambda x.\mathsf{proceed}\ x)\{^{\mathsf{with}(l_1)p}/_{\mathsf{proceed}}\}) \\ = & (\lambda f.\mathsf{with}(l_3)(f\ 10))\ \mathsf{with}(l_1, l_2)(\lambda x.\mathsf{with}(l_1)p\ x) \\ \rightarrow^* & (\lambda f.\mathsf{with}(l_3)(f\ 10))\ (\lambda x.\mathsf{with}(l_1)p\ x) \\ \rightarrow & \mathsf{with}(l_3)((\lambda x.\mathsf{with}(l_1)p\ x)\ 10) \\ \rightarrow & \mathsf{with}(l_3)(\mathsf{with}(l_1)p\ 10) \\ & \qquad \left\{ \textit{ Binding for } p \textit{ found in } l_1 \right\} \\ \rightarrow & \mathsf{with}(l_3)(\mathsf{with}(l_1)((\lambda x.1)\{^{p}/_{\mathsf{proceed}}\})\ 10) \\ = & \mathsf{with}(l_3)(\mathsf{with}(l_1)(\lambda x.1)\ 10) \\ \rightarrow & \mathsf{with}(l_3)((\lambda x.1)\ 10) \\ \rightarrow & \mathsf{with}(l_3)1 \\ \rightarrow & 1 \end{array}$$

Here, whenever proceed dispatches to another parameter, the context in which the evaluation occurs is (part of) the original, not the new context. So the parameter is not dynamically bound in the new context. For example, given layers:

$$\Delta_2 = \left\{ \begin{array}{lcl} l_1 & = & \{p = q, q = \lambda x.4\} \\ l_2 & = & \{p = \lambda x.\mathsf{proceed}\ x\} \\ l_3 & = & \{p = \lambda x.3, q = \lambda x.5\} \end{array} \right\}.$$

the following reduction sequence illustrates that the environment associated with proceed remains longer than expected:

$$\begin{array}{ll} & (\lambda f.\mathsf{with}(l_3)(f\ 10))\ \mathsf{with}(l_1,l_2)p \\ & \qquad \{\ \textit{Binding for } p \textit{ found in } l_2\ \} \\ \rightarrow & (\lambda f.\mathsf{with}(l_3)(f\ 10)) \\ & \qquad \mathsf{with}(l_1,l_2)((\lambda x.\mathsf{proceed}\ x)\{^{\mathsf{with}(l_1)p}/_{\mathsf{proceed}}\}) \\ = & (\lambda f.\mathsf{with}(l_3)(f\ 10))\ \mathsf{with}(l_1,l_2)(\lambda x.\mathsf{with}(l_1)p\ x) \\ \rightarrow^* & (\lambda f.\mathsf{with}(l_3)(f\ 10))\ (\lambda x.\mathsf{with}(l_1)p\ x) \\ \rightarrow & \mathsf{with}(l_3)((\lambda x.\mathsf{with}(l_1)p\ x)\ 10) \\ \rightarrow & \mathsf{with}(l_3)(\mathsf{with}(l_1)p\ 10) \\ & \qquad \{\ \textit{Binding for } p \textit{ found in } l_1\ \} \\ \rightarrow & \mathsf{with}(l_3)(\mathsf{with}(l_1)(q\{^{\mathsf{with}(l_3)p}/_{\mathsf{proceed}}\})\ 10) \\ = & \mathsf{with}(l_3)(\mathsf{with}(l_1)q\ 10) \\ & \qquad \{\ \textit{Binding for } q \textit{ found in } l_1\ \} \\ \rightarrow & \mathsf{with}(l_3)(\mathsf{with}(l_1)((\lambda x.4)\{^{\mathsf{with}(l_3)q}/_{\mathsf{proceed}}\})\ 10) \\ = & \mathsf{with}(l_3)(\mathsf{with}(l_1)(\lambda x.4)\ 10) \\ \rightarrow^* & 4 \end{array}$$

This may be considered wrong, because parameters are by definition dynamically scoped, so one should always use the most recent binding. So q should dispatch to layer l_3. Alternatively, as the layer environment of the original dispatch was $\mathsf{with}(l_1, l_2)$, perhaps q should dispatch to l_1.

4.2 Capture Layer Environment and Use It to Interpret proceed: Variant I

In this semantics, we capture the layer environment to interpret proceed, but do not reinstate the environment. In this semantics, when a parameter p is called in environment E_0, proceed is replaced by a term p_{E_0} recording this information. Term p_{E_0} is evaluated in environment E by finding an appropriate binding in environment $E[E_0[\,]]$, and then continuing evaluation in $E[\,]$. (In comparison, the previous semantics would continue in environment $E[E_0[\,]]$)

To model this, we add the following term to the language, and note that an ordinary parameter is modelled by $p_{[\,]}$:

$$e \quad ::= \quad \cdots \mid p_E$$

The modified rule for parameter dispatch is:

$$\begin{array}{lcll} E[p_{E_0}] & \rightarrow & E[e\{^{p_{E_1}}/_{\mathsf{proceed}}\}] & \\ & & \text{if } E[E_0[\,]] \equiv E_1[\mathsf{with}(l)E_2[\,]], & \\ & & \quad p \notin \mathsf{BP}(E_2) \text{ and } e = \Delta(l)(p) & \\ & & & (\textsc{Disp3}) \end{array}$$

Firstly, the environment $E[\,]$ is extended with $E_0[\,]$. An appropriate layer l is found containing parameter p. In the binding of p in l, proceed is interpreted as p_{E_1}, where E_1 is the remaining layer environment, above layer l.

We now redo the previous examples. Firstly with Δ_1:

$$\begin{array}{ll} & (\lambda f.\mathsf{with}(l_3)(f\ 10))\ \mathsf{with}(l_1,l_2)p \\ & \qquad \{\ \textit{Binding for } p \textit{ found in layer } l_2 \textit{ of } \mathsf{with}(l_1,l_2)[\,].\ \} \\ \rightarrow & (\lambda f.\mathsf{with}(l_3)(f\ 10)) \\ & \qquad \mathsf{with}(l_1,l_2)((\lambda x.\mathsf{proceed}\ x)\{^{p_{\mathsf{with}(l_1)[\,]}}/_{\mathsf{proceed}}\}) \\ = & (\lambda f.\mathsf{with}(l_3)(f\ 10))\ \mathsf{with}(l_1,l_2)(\lambda x.p_{\mathsf{with}(l_1)[\,]}\ x) \\ \rightarrow & (\lambda f.\mathsf{with}(l_3)(f\ 10))\ (\lambda x.p_{\mathsf{with}(l_1)[\,]}\ x) \\ \rightarrow & \mathsf{with}(l_3)((\lambda x.p_{\mathsf{with}(l_1)[\,]}\ x)\ 10) \\ \rightarrow & \mathsf{with}(l_3)(p_{\mathsf{with}(l_1)[\,]}\ 10) \\ & \qquad \{\ \textit{Binding for } p \textit{ found in layer } l_1 \textit{ of } \mathsf{with}(l_3,l_1)[\,].\ \} \\ \rightarrow & \mathsf{with}(l_3)((\lambda x.1)\{^{p_{\mathsf{with}(l_3)}}/_{\mathsf{proceed}}\}\ 10) \\ = & \mathsf{with}(l_3)((\lambda x.1)\ 10) \\ \rightarrow^* & 1 \end{array}$$

The following example, with layers Δ_2, shows that this approach differs from the previous one:

$$\begin{array}{ll} & (\lambda f.\mathsf{with}(l_3)(f\ 10))\ \mathsf{with}(l_1,l_2)p \\ \rightarrow & (\lambda f.\mathsf{with}(l_3)(f\ 10))\ \mathsf{with}(l_1,l_2)(\lambda x.p_{\mathsf{with}(l_1)[\,]}\ x) \\ \rightarrow^* & (\lambda f.\mathsf{with}(l_3)(f\ 10))\ \lambda x.p_{\mathsf{with}(l_1)[\,]}\ x) \\ \rightarrow & \mathsf{with}(l_3)((\lambda x.p_{\mathsf{with}(l_1)[\,]}\ x)\ 10) \\ \rightarrow & \mathsf{with}(l_3)(p_{\mathsf{with}(l_1)[\,]}\ 10) \\ \rightarrow & \mathsf{with}(l_3)(q\ 10) \\ & \qquad \{\ \textit{Binding for } q \textit{ found in layer } l_3 \textit{ of } \mathsf{with}(l_3)[\,].\ \} \\ \rightarrow & \mathsf{with}(l_3)((\lambda x.5)\ 10) \\ \rightarrow^* & 5 \end{array}$$

This shows that with this approach, parameters are now (correctly) dynamically scoped.

4.3 Capture Layer Environment and Use It to Interpret proceed: Variant II

The approach above performs the search in the current layer environment augmented with the captured layers. Another variant performs the search exclusively in the original layer environment. Thus we would have the two rules:

$$\begin{array}{lcll} E[\mathsf{with}(l)E'[p]] & \rightarrow & E[\mathsf{with}(l)E'[e\{^{p_E}/_{\mathsf{proceed}}\}]] & \\ & & \text{if } p \notin \mathsf{BP}(E') \text{ and } e = \Delta(l)(p) & \\ & & & (\textsc{Disp4a}) \\ E[p_{E_0}] & \rightarrow & E[e\{^{p_{E_2}}/_{\mathsf{proceed}}\}] & \\ & & \text{if } E_0[\,] \equiv E_2[\mathsf{with}(l)E_3[\,]], & \\ & & \quad p \notin \mathsf{BP}(E_3) \text{ and } e = \Delta(l)(p) & \\ & & & (\textsc{Disp4b}) \end{array}$$

The first rule describes the initial dispatch to a parameter. The second rule describes dispatch corresponding to proceed. The environment in which the search is performed is $E_0[\,]$, rather than $E[E_0[\,]]$. For our example, the results are the same as above.

4.4 Build Composite Expression

In this approach, when a parameter is dispatched, the entire expression to which it corresponds, with all the proceed calls expanded in place, is computed:[5]

$$E[p] \rightarrow E[build(p, \overline{E})] \qquad \text{(DISP5)}$$

$$build(p, [\,]) = error$$

$$build(p, \mathsf{with}(\bar{l}, l_0)) = \begin{cases} e\{^{build(p,\mathsf{with}(\bar{l}))}/_{\mathsf{proceed}}\} \\ \quad \text{if } e = \Delta(l_0)(p) \\ build(p, \mathsf{with}(\bar{l})) \\ \quad \text{otherwise} \end{cases}$$

This approach is illustrated using layers Δ_2:

$$\begin{array}{ll} & (\lambda f.\mathsf{with}(l_3)(f\ 10))\ \mathsf{with}(l_1, l_2)p \\ \left\{\begin{array}{ll} & build(p, \mathsf{with}(l_1, l_2)) \\ = & (\lambda x.\mathsf{proceed}\ x)\{^{build(p,\mathsf{with}(l_1))}/_{\mathsf{proceed}}\} \\ = & (\lambda x.\mathsf{proceed}\ x)\{^{q}/_{\mathsf{proceed}}\} \\ = & (\lambda x.q\ x) \end{array}\right\} \\ \rightarrow & (\lambda f.\mathsf{with}(l_3)(f\ 10))\ \mathsf{with}(l_1, l_2)(\lambda x.q\ x) \\ \rightarrow & (\lambda f.\mathsf{with}(l_3)(f\ 10))\ (\lambda x.q\ x) \\ \rightarrow & \mathsf{with}(l_3)((\lambda x.q\ x)\ 10) \\ \rightarrow & \mathsf{with}(l_3)(q\ 10) \\ & \qquad \left\{\ build(q, \mathsf{with}(l_3)) = \lambda x.5\ \right\} \\ \rightarrow & \mathsf{with}(l_3)((\lambda x.5)\ 10) \\ \rightarrow^* & 5 \end{array}$$

This approach is actually semantically the same as the approach in Section 4.3 (we conjecture). They differ in the way they perform the computation. One approach (§ 4.3) performs the computation of the 'generic function' on-the-fly, by reusing the previously saved layer to perform dispatch. The other (§ 4.4) determines the complete generic function whenever dispatching to a parameter. This second approach corresponds to the precomputation of applicable methods in generic functions in CLOS, and hence ContextL.

4.5 Discussion

Table 1 summarizes the different approaches described until now. For each proposal, the layer environment in which the lookup of a binding is performed is specified (E_{look}), as well as the layer environment in which execution continues (E_{eval}). As before, E_0 refers to the remaining definition-time layer environment above the layer where proceed was captured, and E is the current layer environment. Also, $E_{0,\text{full}}$ is the complete definition-time layer environment, including the layers below and the layer where proceed was captured.

From this table, we can rule out several approaches that have clear drawbacks. Looking up the binding in the current environment $E[\,]$ is unsatisfactory because it may fail to find an appropriate layer. Also, continuing evaluation in an environment where the definition-time layer environment is at the bottom, such as $E[E_0[\,]]$ (4.1), destroys dynamic binding of parameters, by allowing definition-time bindings to shadow the currently-active ones. Using simply $E_0[\,]$ or $E_{0,\text{full}}[\,]$[6] to continue in would be similarly wrong, since current dynamic bindings would not be available at all. However, $E_{0,\text{full}}[E[\,]]$ seems to be a reasonable choice for continuing evaluation, since both dynamic binding of parameters is respected *and* the definition-time layer environment is present to fulfil the expectations of original definitions.

For lookup, we have three remaining possibilities:

- $E_0[\,]$ lookup can fail even though a parameter binding is available in the current layer environment. This seems to contradict the purpose of dynamically-activated layers.
- $E_0[E[\,]]$[7] A binding in the current layer environment can shadow a binding in the definition-time environment.
- $E[E_0[\,]]$ A binding in the definition-time environment shadows bindings in the current environment.

It is obvious that $E_0[\,]$ should be present, but we currently do not agree whether or how $E[\,]$ should be included.

5. Obtaining More Control

The previous strategies are meant to be possible defaults: the same semantics for all lambdas and for all layers. First the appropriate layer is found in an environment E_{look}, then the evaluation is continued in environment E_{eval}.

It is also possible to give programmers more fine-grained control. For instance, at the lambda level, we can distinguish between a lambda that captures E_0 and one which does not (resulting in $E_0 = [\,]$). We explore this in Section 5.1. In Section 5.2 we explore the dual approach, where fine-grained control is given at the layer level: we can specify that some layers *stick* to an escaping closure (and are therefore part of E_0) so that they are reinstated when the closure is applied, whereas others do not.

5.1 Control at the Lambda Level

Here, we annotate λ-abstractions to indicate whether or not they capture the layer environment they escape. Assume that λ° denotes the λ which does not capture its context and $\lambda^\bullet$ be the λ which does. We have the following rules:

$$E[\mathsf{with}(l)\lambda^\circ x.e] \rightarrow E[\lambda^\circ x.e] \qquad \text{(ESC-}\circ\text{)}$$

$$E[\mathsf{with}(l)\lambda^\bullet x.e] \rightarrow E[\lambda^\bullet x.\mathsf{with}(l)e] \qquad \text{(ESC-}\bullet\text{)}$$

The same β-reduction rule applies to both λs.

λ° behaves as before, discarding the surrounding layer activations. $\lambda^\bullet$ behaves differently, wrapping the body of the λ-abstraction with the layer environment, so that when the λ-abstraction is applied, the captured layers will be reactivated.

[5] This is similar to linearization of class hierarchies in, for example, Scala.

[6] Both not described in this paper.

[7] Not described in this paper.

Proposal	E_{look}	E_{eval}	Comments
Naïve	$E[\,]$	$E[\,]$	May fail to find appropriate layer (see Section 3)
4.1	$E[E_0[\,]]$	$E[E_0[\,]]$	Sacrifices dynamic binding of parameters
4.2	$E[E_0[\,]]$	$E[\,]$	Dynamic binding is back
4.3	$E_0[\,]$	$E[\,]$	Same as previous, though lookup fails even if current environment has a binding
4.4	$E_0[\,]$	$E[\,]$	Same as previous, though generic function is computed when called

Table 1. Comparison of Approaches: E_{look} is where the lookup of a binding is performed, and E_{eval} is where the execution continues. E_0 is the environment in which the closure containing proceed was defined and E is the current layer environment.

Let

$$\Delta_3 = \left\{ \begin{array}{lcl} l_1 & = & \{p = q, q = \lambda^\circ x.4\} \\ l_2 & = & \{p = \lambda^\bullet x.\mathsf{proceed}\ x\} \\ l_3 & = & \{p = \lambda^\circ x.3, q = \lambda^\circ x.5\} \end{array} \right\}.$$

Now, adapting the example above:

$$\begin{array}{ll} & (\lambda^\circ f.\mathsf{with}(l_3)(f\ 10))\ \mathsf{with}(l_1, l_2)p \\ \rightarrow & (\lambda^\circ f.\mathsf{with}(l_3)(f\ 10))\ \mathsf{with}(l_1, l_2)(\lambda^\bullet x.q\ x) \\ \rightarrow^* & (\lambda^\circ f.\mathsf{with}(l_3)(f\ 10))\ (\lambda^\bullet x.\mathsf{with}(l_1, l_2)(q\ x)) \\ \rightarrow & \mathsf{with}(l_3)((\lambda^\bullet x.\mathsf{with}(l_1, l_2)(q\ x))\ 10) \\ \rightarrow & \mathsf{with}(l_3, l_1, l_2)(q\ 10) \\ & \qquad \left\{ \ build(q, \mathsf{with}(l_3, l_1, l_2)) = \lambda^\circ x.4 \ \right\} \\ \rightarrow & \mathsf{with}(l_3)((\lambda^\circ x.4)\ 10) \\ \rightarrow & \mathsf{with}(l_3)4 \\ \rightarrow^* & 4 \end{array}$$

5.2 Control at the Layer Level

Inspired by the work of Tanter on expressive scoping of aspects [7], we can consider (at least) two kinds of layer activations: a *fluid* activation such that a layer does not stick to closures, and a *sticky* activation such that a layer is captured in the definition-time layer environment of a closure ($E_0[]$).

$$\begin{array}{rcll} E[\mathsf{with\text{-}fluid}(l)\lambda x.e] & \rightarrow & E[\lambda x.e] & (\textsc{EscFl}) \\ E[\mathsf{with\text{-}sticky}(l)\lambda x.e] & \rightarrow & E[\lambda x.\mathsf{with\text{-}sticky}(l)e] & (\textsc{EscSt}) \end{array}$$

Assume that we are using layers Δ_2 and the final semantics presented in the last section. Then:

$$\begin{array}{ll} & (\lambda f.\mathsf{with\text{-}fluid}(l_3)(f\ 10)) \\ & \qquad \mathsf{with\text{-}sticky}(l_1)\mathsf{with\text{-}fluid}(l_2)p \\ \rightarrow & (\lambda f.\mathsf{with\text{-}fluid}(l_3)(f\ 10)) \\ & \qquad \mathsf{with\text{-}sticky}(l_1)\mathsf{with\text{-}fluid}(l_2)(\lambda x.q\ x) \\ \rightarrow & (\lambda f.\mathsf{with\text{-}fluid}(l_3)(f\ 10))\ \mathsf{with\text{-}sticky}(l_1)(\lambda x.q\ x) \\ \rightarrow & (\lambda f.\mathsf{with\text{-}fluid}(l_3)(f\ 10))\ (\lambda x.\mathsf{with\text{-}sticky}(l_1)(q\ x)) \\ \rightarrow & \mathsf{with\text{-}fluid}(l_3)((\lambda x.\mathsf{with\text{-}sticky}(l_1)(q\ x))\ 10) \\ \rightarrow & \mathsf{with\text{-}fluid}(l_3)\mathsf{with\text{-}sticky}(l_1)(q\ 10) \\ & \qquad \left\{ \ build(q, \mathsf{with}(l_3, l_1)) = \lambda x.4 \ \right\} \\ \rightarrow & \mathsf{with\text{-}fluid}(l_3)((\lambda x.4)\ 10) \\ \rightarrow & \mathsf{with\text{-}fluid}(l_3)4 \\ \rightarrow & 4 \end{array}$$

6. Conclusion

We argued that it is not clear how to give semantics to a closure containing a call to proceed which subsequently escapes its defining layer environment. We offer a number of possible solutions to this problem, along with mechanisms for better handling the layer environment associated with an escaping closure. By exploring various possibilities, we want to offer programmers both a predictable semantics so that they can reason about their code, and flexibility so that they can design the code to do exactly what they want it to. We offer no definitive answer. Furthermore, we conjecture that delimited dynamic bindings [5] will provide even more flexibility.

References

[1] Daniel Bobrow, Linda DeMichiel, Richard Gabriel, Sonya Keene, Gregor Kiczales, and David Moon. Common Lisp Object System Specification. *Lisp and Symbolic Computation*, 1(3-4):245–394, 1989.

[2] Dave Clarke and Ilya Sergey. A Semantics for Context-oriented Programming with Layers. Submitted to COP2009, April 2009.

[3] Pascal Costanza and Robert Hirschfeld. Language Constructs for Context-oriented Programming. In *ACM Dynamic Languages Symposium 2005*. ACM Press, 2005.

[4] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented Programming. *Journal of Object Technology*, March/April 2008.

[5] Oleg Kiselyov, Chung chieh Shan, and Amr Sabry. Delimited Dynamic Binding. In *International Conference on Functional Programming (ICFP)*, volume 41 of *SIGPLAN Notices*, pages 26–37. ACM, 2006.

[6] Luc Moreau. A Syntactic Theory of Dynamic Binding. *Higher-Order and Symbolic Computation*, 11(3):233–279, 1998.

[7] 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.