Contextual values

Conference Paper · January 2008

DOI: 10.1145/1408681.1408684 · Source: DBLP

1 author:

Éric Tanter
University of Chile
188 PUBLICATIONS **2,236** CITATIONS

Contextual Values

Éric Tanter *

PLEIAD Laboratory
Computer Science Department (DCC)
University of Chile – Santiago, Chile
etanter@dcc.uchile.cl

Abstract

Context-oriented programming proposes to treat execution context explicitly and to provide means for context-dependent adaptation at runtime. There are many mechanisms that can be used to make either variable bindings or application code adapt dynamically, like dynamically-scoped variables, dynamic layer activation, and contextual dispatch. There are no means however, to make actual *values* be context-dependent. This means that side effects engendered by dynamically-activated adaptations are potentially global. We propose *contextual values*: values that actually depend on the context in which they are looked at and modified. We explore how contextual values can be provided, either as a simple library or directly into the language through different designs, for which we provide the operational semantics in the form of Scheme interpreters. Being able to scope side effects to certain contexts is a step forward for context-oriented programming, and can have interesting applications in other areas like security and dynamic software evolution.

1. Introduction

Context awareness [1, 8] is the ability of a program to behave differently depending on its surrounding execution context. The need for this context-dependent behavior has been recognized in many areas, from ubiquitous computing [29], self-adaptive [18] and autonomic systems [14], to more standard business applications, for instance where personalization is a key concern. With traditional programming techniques and infrastructures, addressing context awareness requires the development of unnecessarily complex and tangled solutions.

Several approaches to address context awareness have been proposed at different levels, including middleware [3, 4, 20] and programming languages [7, 13, 23, 26]. Taking context into account at the level of the programming language allows for simpler development, evolution and reuse of software [26]. For instance, ContextL [7] introduces several linguistic mechanisms to support context-dependent behavior, by allowing code structures (classes) to be context-dependent: a class can be defined in several layers, and layers can be dynamically activated. At a lower level of abstraction, dynamic binding mechanisms, like special variables in Common Lisp [21], allow variable bindings to be redefined for certain dynamic extents.

We observe that none of the proposals for context-oriented linguistic constructs makes it possible for *values* to be context-dependent. As a result, any side effect that occurs in a given context cannot easily be made local to that context: it is visible from all the parts of the system that have access to the value, independently of their execution context. As far as we have been able to determine, the only kind of context-dependent value that has been identified, and is indeed widely used, is thread-local values: values that are specific to the executing thread.

This paper proposes *contextual values*, as a simple generalization of thread-local values to any execution context information. Starting from this generalization in the form of explicit contextual values provided as a library (Section 2), we proceed to explore actual language support for contextual values (Section 3). Finally, we examine a new language construct, based on contextual values, for systematically scoping side effects performed in certain regions of a program execution (Section 4). We describe the operational semantics of our proposals by extending a small Scheme interpreter. We discuss related proposals in Section 5, showing that, by focusing on the issue of *shared state*, contextual values are indeed complementary to existing approaches for context-dependent behavior adaptation. Section 6 concludes.

2. Explicit Contextual Values

In their simplest form, contextual values are a trivial generalization of *thread-local* values. We first give a brief recall of what thread locals are, and then discuss their generalization.

2.1 Thread-local Values

The most well-known kind of contextual values are thread-local values: values that are specific to the executing thread. More precisely, a thread local is a cell whose content is, when de-referenced, specific to the executing thread. Thus it may contain different values for different threads. Most languages with concurrency support provide thread-local values of some sort, including POSIX threads, C++ libraries, C# thread static attributes, Java, Python, Delphi, various Scheme, Lisp, and Smalltalk dialects, and more.

For instance, in Java a thread local is an instance of the `ThreadLocal` class, and its content is accessed with `get` and `set` methods:

```
ThreadLocal<Integer> counter = new ThreadLocal<Integer>();
counter.set(1); int c = counter.get();
```

The initial state of the thread local is specified in the method `initialValue` that can be redefined in subclasses. PLT Scheme [10] adopts a slightly different design, where a thread cell is created with a default value that is used for all existing threads:

* Partially financed by the Millenium Nucleus Center for Web Research, Grant P04-067-F, Mideplan, Chile, and FONDECYT project 11060493.

DLS'08 July 8, 2008, Paphos, Cyprus

```
(define counter (make-thread-cell 0))
(thread-cell-set! counter 1)
(thread-cell-ref counter)
```

In addition, in PLT Scheme the value of a thread cell can be *preserved*, meaning that when a new thread is created, the current value of the cell for the creator thread is used as the initial value for the new thread.

Beyond these differences mostly related to initialization, the essence of thread-local values is the same. A straightforward implementation of a thread local is as a wrapper around a table that associates thread identifiers to values. A getter does a look up in the table using the current thread identifier as key, while a setter updates the table.

2.2 Contextual Values

In essence, our proposal boils down to a trivial generalization of the idea of thread-local values. Considering that the current thread is but one kind of execution context information, we propose to generalize the idea to any kind of context.

From a programming language point of view, the general understanding of context is as *any information that is computationally accessible*. In this light, a contextual value is like a thread local, except that any computationally-accessible value can be used to discriminate amongst possible values, not only the current thread.

For instance, let us create a contextual value in Scheme, bound to the variable `background-color`; the value is black by default, and its *context function* returns the value bound to `user`:

```
(define background-color
   (make-cv-init (lambda () user) black))
```

For a certain context –here, a certain user name–, any side effect on `background-color` is effective:

```
(define user "Totoro")
(cv-set! background-color red)
(cv-ref background-color) ;;--> red
```

Conversely, if the context changes –that is, if the user changes–, the previous side effect is *not* effective:

```
(set! user "Bilbo")
(cv-ref background-color) ;;--> black
```

2.2.1 Supporting contextual values

Not surprisingly, introducing contextual values in Scheme, Java, or any language with structured data types, is not very challenging. A contextual value consists of a context function ctx of no arguments, and a value mapping $vals$, mapping possible values of the context function to actual values. $vals$ is a function that returns a default value whenever there is no explicit mapping for a given context:

$$cv = \langle ctx, vals \rangle$$

Setting a contextual value means applying the context function ctx, and using the returned value as a key to update the definition of $vals$ (either by creating a new mapping or by updating an existing one). Similarly, getting the actual value of a contextual value means applying ctx and passing the returned value to $vals$.

Listing 1 presents the definition of contextual values in PLT Scheme. `define-struct` defines a structure of three fields: the context function, the value mappings (an association list), and the default value. The structure definition generates accessor functions, like `cv-ctx` to access the context function. `extend-vals` (the code of which is omitted) extends the association list with a new mapping. The constructor `make-cv-init` is a convenience function to create a contextual value initialized with the mapping of the current value of the context function to the given default value.

Listing 1 Definition of contextual values in PLT Scheme.

```
;; data type definition
(define-struct cv (ctx vals default))

;; read a cv
(define (cv-ref cv)
  (let ((key ((cv-ctx cv))))
    (let ((val (assoc key (cv-vals cv))))
      (if val (cdr val) (cv-default cv)))))

;; set a cv
(define (cv-set! cv nval)
  (let ((key ((cv-ctx cv))))
    (let ((val (assoc key (cv-vals cv))))
      (if val (set-cdr! val nval)
              (extend-vals cv key nval)))))

;; simplified constructor
(define (make-cv-init ctx val)
  (make-cv ctx (list (cons (ctx) val)) val))
```

Listing 2 Thread-local values as contextual values.

```
;; constructor of thread local
(define (make-tl val)
  (make-cv-init current-thread val)))

;; accessors
(define tl-ref cv-ref)
(define tl-set! cv-set!)
```

Listing 3 Playing with thread locals.

```
(define counter (make-tl 0))
(tl-ref counter) ;; --> 0

(thread (lambda ()
          (tl-set! counter 42)
          (let loop ()
            (display (tl-ref counter)) ;; --> 42 for ever
            (loop))))

(tl-ref counter) ;; --> 0
(tl-set! counter 1)
(tl-ref counter) ;; --> 1
```

2.2.2 Expressing thread-local values

Providing a simple thread cell library on top of contextual values is trivial: a thread cell is a contextual value whose context function is the primitive used to obtain the current thread. In Java this is done using `Thread.currentThread`, and in PLT Scheme, `current-thread`. Listing 2 shows the implementation of a thread cell library using contextual values. Listing 3 illustrates the use of this library.

2.2.3 Contextual values and dynamic scope

To further illustrate the kind of applications of contextual values, we show their use in another toy example, where we make use of dynamic bindings as supported by `fluid-let` in Scheme, similar to special variables in Common Lisp [21]: a variable can be rebound for a given dynamic extent.

Listing 4 demonstrates how to take advantage of dynamic binding by defining a contextual value `msg` whose context function returns the value of a `language` variable. In turn, we use dynamic bindings to alter in certain dynamic extents the binding of this vari-

Listing 4 Playing with contextual values.

```
(define language "EN")

;; contextual value msg
(define msg (make-cv-init (lambda () language) "hello"))

;; function using contextual value
(define show-message
  (lambda () (display (cv-ref msg))))

;; init for different contexts
(fluid-let ((language "SP"))
  (cv-set! msg "hola"))
(fluid-let ((language "FR"))
  (cv-set! msg "bonjour"))

(show-message) ;;--> hello

(set! language "SP")
(fluid-let ((language "FR"))
  (show-message) ;; --> bonjour

(show-message) ;;--> hola
```

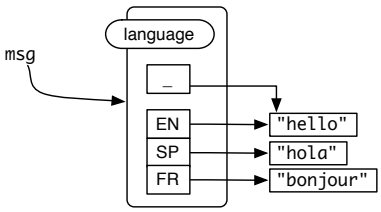

Figure 1. A contextual value holding different values depending on the language context.

able. For instance, we initialize `msg` in different language contexts using `fluid-let`. Note that the side effect on the contextual value is visible only when the execution is again in a context for which `language` is bound to the value it had when the side effect was performed.

This illustrates an important difference between dynamic binding and contextual values. While a dynamic rebinding is only valid for the extent of the execution of the nested expression, the assignment of a contextual value *survives* this extent: it is only dependent on the outcome of the context function. Fig. 1 illustrates the binding between `msg` and the contextual value. The empty box refers to the default value.

2.3 Issues: Obliviousness and Shared State

The version of contextual values we have presented in this section follows the same design principle as thread cells in Scheme or thread locals in Java: contextual values are provided by means of a library. We refer to this design as providing *explicit* contextual values. The programming language itself has no notion whatsoever of what a contextual value is. Indeed, the name of contextual (or thread local) *values* is misleading, because in effect, the value is a container, and is global. The container itself is just smart enough to implement context dependence when its content is accessed.

While this explicit approach is simple to implement and manage, it presents two important limitations. First, it forces programmers to explicitly manipulate the containers, with accessors like `cv-ref` and `cv-set!`. As a consequence, if the decision is taken to make a certain value contextual, all the parts of the program that manipulate it must be updated to use these explicit operations. In other words, a given module cannot be oblivious to the fact that it is manipulating a value that is contextual.

Listing 5 Contextual values in the language.

```
(define language "EN")
;; formatter :: val -> string
(define file-formatter (cv (lambda () language)
                           (lambda (value) ...)))

;;--rest of the program, unaware of contextual value--

;; printing library function
(define (print-to-file path val)
  (write-file (open-file path)
              (file-formatter val))) ;; standard access

;;...in a spanish module, where language is "SP"...
(set! file-formatter
      (lambda (valor) ...)) ;; standard assignment
```

A second consequence is that, being a container, a contextual value becomes a means to share mutable state between procedures in a call-by-value language. Indeed, passing a container, such as a box, a list, a vector or any other mutable data structure, as parameter in a call-by-value language ensures that mutation of the state of the container done by the callee is visible at the caller site. It seems rather odd to couple the notions of shared mutable state and contextual values. The essence of contextual values is, after all, that the actual *value* depends on the context in which it is looked at.

The rest of this paper explores *language support* for contextual values, making it possible to implicitly and transparently use contextual values, while preserving call-by-value semantics.

3. Implicit Contextual Values

We begin our exploration of language support for contextual values by introducing support for uniform access and assignment of variables, regardless of whether they refer to normal or contextual values. Denoting which values are to be treated as contextual is still done explicitly; only the *use* of these values is oblivious to context dependence. After a brief program example, we explain how to implement such a language by first exposing a small Scheme-like language, its interpreter, and how to extend it to support implicit contextual values.

3.1 Programming with Implicit Contextual Values

A programmer has to denote explicitly the values that must be treated as contextual, by using a `cv` expression that specifies the context function and default value (and possibly initial value mappings). This corresponds to the constructor we have seen previously with explicit contextual values.

The difference is now that the language processor itself takes care of the semantics of contextual values. This means that there are no *syntactic* differences in accessing or assigning a value, whether it is a standard value or a contextual one: explicit accessors like `cv-ref` and `cv-set!` have disappeared. Listing 5 illustrates a context-dependent behavior of a printing function, where the contextual value first referred to as `file-formatter` is read and set in various parts of the program like any other value. Both the printing library function and the code corresponding to a spanish module are *oblivious* to the fact that `file-formatter` is contextual.

3.2 Working out the semantics

Introducing implicit contextual values in the language actually raises a number of issues, all related to the fact that from the viewpoint of the programmer, their use is transparent.

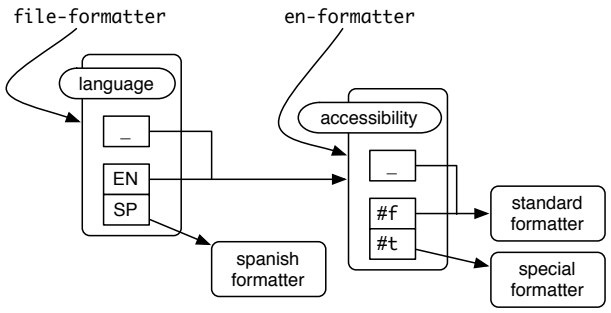

Figure 2. Nesting of contextual values.

3.2.1 Nested contextual values

With explicit contextual values, nested contextual values are also explicitly managed: if the value of a contextual value is another contextual value, then the client has to explicitly `cv-ref` it. With implicit contextual values, this can no longer be the case. Because contextual values are not visible from the language point of view, the interpreter itself must take care of recursively reducing a contextual value to an actual value whenever needed.

For instance, suppose that instead of using an anonymous default formatter in Listing 5, we use the English formatter bound to the variable `en-formatter`. It may be the case that this formatter is already a contextual value, *e.g.* such that, depending on an "accessibility" context, the formatter is either a standard one, or a special formatter that uses larger symbols.

The definition of `file-formatter` as a contextual value depending on the language context results in *nested* contextual values; we say that `file-formatter` is bound to a *compound* contextual value (Figure 2). Therefore, obtaining the actual value of `file-formatter` at a given point in time implies first dispatching based on the language context, and if the current language context is English, then a second dispatch must be performed, based on the accessibility context. Nesting is clearly not commutative.

3.2.2 Procedural abstraction boundaries

A call-by-value language ensures that assignments to variables in a called procedure do not affect variable bindings in the caller context. Avoid such aliasing of values is an important point to ensure good abstraction boundaries between units of modularity. If the callee is intended to mutate a value, then it should consume a container data structure. This is then part of the explicit interface between the caller and the callee. Supporting contextual values implicitly in the language requires the call-by-value semantics to be preserved, even for contextual values.

For instance, with explicit contextual values, the mutation done in the `foo` function is visible for the caller, because the contextual value is a container:

```
(define (foo x) (cv-set! x 3))
(let ((a (make-cv ctx 10)))
  (foo a)
  (cv-ref a))  --> 3
```

Conversely, with implicit contextual values, the assignment ought not affect the value of `a` at the caller site:

```
(define (foo x) (set! x 3))
(let ((a (cv ctx 10)))
  (foo a)
  a)  --> 10
```

3.2.3 Call-by-"value"?

In a call-by-value language, the arguments to a function are, as the name suggests, reduced to values before being bound in the environment used to evaluate the function body. A naive implementation of implicit contextual values may overlook this and result in a language where as soon as a contextual value is passed as parameter, it is reduced to an actual value and is therefore not contextual anymore. This would –least to say– greatly reduce the interest of contextual values.

For instance, going back to the example of Listing 5, suppose that the `print-to-file` function is defined in a lexical scope where `file-formatter` is not bound; rather it takes the formatter to apply as a parameter:

```
(define (print-to-file path val formatter)
  (write-file (open-file path)
              (formatter val)))
```

If a context-dependent formatter is passed as parameter to this function, possibly through different intermediate steps, potentially distant in time, we want to be sure that the actual formatter used is determined by the context at the time the formatter is effectively applied.

3.3 Interpretation of Contextual Values

We now dive into the details of the semantics of implicit contextual values by exposing their interpretation. We do by first introducing a small Scheme language and its traditional interpretation (Listing 6). Then, we extend it with implicit contextual values (Listing 7), explaining how we address the different semantic issues raised in the previous section.

3.3.1 A small Scheme and its interpretation

The small Scheme language we will extend is very similar to the core Scheme with mutation used by Matthews and Findler [17]. It supports literals (numbers, strings, booleans), variables, sequencing, `if`, `let`, arbitrary arity procedures (either primitives or first-class anonymous functions), and, of course, assignment.

```
<expr> ::= <lit>
         | <id>
         | (lambda ({<id>}*) {<expr>}*)
         | (if <expr> <expr> <expr>)
         | (let ({(<id> <expr>)}*) {<expr>}*)
         | (<expr> {<expr>}*)
         | (<prim> {<expr>}*)
         | (set! <id> <expr>)
```

The interpreter is written in the classical environment-passing style [11]. The `eval` procedure receives both the expression to evaluate and the lexical environment, and is structured as a type case on the expression to evaluate. The core of this interpreter is given on Listing 6 for reference. We do not give the interpretation of `let` expressions as they are equivalent to immediate application of an anonymous function.

The important parts for this work are the way variable and assignment expressions are interpreted, as well as parameter passing in function application. In order to allow mutation of variables, the environment contains bindings of variables (symbols) to addresses in the store, where value reside. These addresses are represented as structures called references. The interface of this data type is the function `deref` that, given a reference, returns the associated value; and the function `setref!` that, given a reference and a value, changes the reference so that it refers to the given value.

The use of references can be seen on Listing 6. The function `env-lookup` looks up an identifier in the environment, and returns a reference to the associated value. A variable expression

Listing 6 Scheme interpreter of a small Scheme-like language with mutation.

```
;; evaluate expression exp in lexical environment env
(define (eval exp env)
  (cond ((lit-exp? exp)    (lit-exp-value exp))
        ((lambda-exp? exp) (make-closure (lambda-exp-params exp) (lambda-exp-body exp) env))
        ((prim-exp? exp) (let ((args (eval-args (prim-exp-args exp) env))) (apply (prim-exp-prim exp) args)))
        ((if-exp? exp)   (if (eval (if-exp-test exp) env) (eval (if-exp-then exp) env) (eval (if-exp-else exp) env)))
        ((var-exp? exp)  (deref (env-lookup (var-exp-name exp) env)))
        ((set!-exp? exp) (setref! (env-lookup (set!-exp-name exp) env) (eval (set!-exp-nval exp) env)))
        ((app-exp? exp)  (let* ((cl   (eval (app-exp-fun exp) env))
                                (args (eval-args (app-exp-args exp) env))
                                (env  (extend-env (closure-params cl) args (closure-env cl))))
                           (eval-body (closure-body cl) env)))))
;; data type for values
(define-struct value ())
(define-struct (closure value) (params body env)) ;; only one variant, closures
```

(`var-exp`) is interpreted by de-referencing the returned reference; similarly, an assignment (`set!-exp`) sets the referenced value to the new value.

When a function is applied (`app-exp`), its body is evaluated in an environment that extends the definition-time environment of the function with the bindings of the formal to the actual parameters. The extended environment is obtained by applying `extend-env`. To ensure call-by-value semantics, this function creates new store cells to hold the values of the actual parameters (hence avoiding aliasing).[1]

3.3.2 Introducing contextual values

To extend the language with contextual values, the only syntactic extension is the new expression to define a contextual value:

```
<expr> ::= ... | (cv <expr> <expr>)
```

Internally, the interpreter must now manipulate contextual values as a new kind of values, as shown on Listing 7. The data type definition is the same as that of explicit contextual values (Listing 1). Interpreting a `cv` expression simply instantiates the a contextual value structure, evaluating the context function and the default value (and initial mappings if provided).

3.3.3 Evaluating a variable

Contextual values, as all values, live on the store. This means that when looking up a variable in the environment, we can get a reference to a contextual value. As discussed in Section 3.2.3, when we pass a contextual value as parameter, we do not want to reduce it to an actual value: it should remain contextual. So the question arises of when a contextual value should in fact be reduced.

This issue is reminiscent of lazy evaluation: in a language with lazy evaluation like Haskell, argument expressions are not evaluated until they are *really* needed. Instead, *expression closures* are passed as arguments (to preserve static scoping). The precise points at which expression closures are reduced to actual values are called the *strictness* points of the language [16].

Implicit contextual values deserve a similar treatment. Interestingly, the strictness points of the interpreter with contextual values include those of a typical language with lazy evaluation: top-level evaluation, arguments of a primitive application, test part of an `if`, and function position of a function application.

Therefore, to reduce contextual values to actual values, we use a strictness function. To properly deal with compound contextual values, `strict` recursively applies `cv-val` until a non-contextual value is reached:

```
;; strict :: value -> non-contextual value
(define (strict val)
  (if (cv? val)
      (strict (deref (cv-val val)))
      val))
```

The `cv-val` procedure in the interpreter is essentially the same as that of Listing 1: it determines the corresponding value depending on the outcome of the context function. The only difference is that it now deals with store references (as in the rest of the evaluator) and not association lists.

Listing 7 highlights the particular points where `strict` is applied. First, we add a top-level evaluation function, `eval-prog`: it is a strictness point because users are not interested in getting back a contextual value structure as the result of the execution of a program. For primitive application, we use a new `strict-eval-args` function, that evaluates strictly each argument. This is because a primitive function, like `+` or `<`, does not know how to deal with contextual values. Similarly, the result of the evaluation of the test part of an `if` must be reduced, for the interpreter to be able to determine which branch to evaluate. Finally, function application requires the function expression to be reduced to an actual value (a closure) in order to proceed.

3.3.4 Assignments

Assignments must now discriminate on the kind of value that is referenced. To do so, in the interpreter we call a `setval!` function instead of using `setref!` directly:

```
(define (setval! r nval)
  (let ((val (deref r)))
    (if (cv? val)
        (cv-update! val nval)
        (setref! r nval))))
```

If the referenced value is non-contextual, the reference is simply set. If the referenced value is contextual, it is updated with the new value. Updating a contextual value means determining in which context the execution currently is; if an existing value is present for this context, replace it, otherwise, add a new binding for the new context value. Note that to properly deal with compound contextual values, `cv-update!` recursively applies all context functions in order to reach either a normal value or no value at all.

[1] For more details, we refer the reader to the reference textbook [11] and our implementations at: http://pleiad.dcc.uchile.cl/research/scope

Listing 7 Scheme interpreter of a small Scheme-like language with implicit contextual values. *(changes highlighted)*

```
;; top-level evaluation
(define (eval-prog exp)
 (strict (eval exp (empty-env))))

;; evaluate expression exp in lexical environment env
(define (eval exp env)
  (cond ((lit-exp? exp)  ...)
        ((lambda-exp? exp)  ...)
        ((cv-exp? exp)   (make-cv (eval (cv-exp-ctx exp) env ...))
        ((prim-exp? exp) (let ((args (strict-eval-args (prim-exp-args exp) env))) (apply (prim-exp-prim exp) args)))
        ((if-exp? exp)   (if (strict (eval (if-exp-test exp) env))
                             (eval (if-exp-then exp) env)
                             (eval (if-exp-else exp) env)))
        ((var-exp? exp)  (deref (env-lookup (var-exp-name exp) env)))
        ((set!-exp? exp) (setval! (env-lookup (set!-exp-name exp) env) (eval (set!-exp-nval exp) env)))

        ((app-exp? exp)  (let* ((cl   (strict (eval (app-exp-fun exp) env)))
                                (args (eval-args (app-exp-args exp) env))
                                (env  (extend-env (closure-params cl) (by-val args) (closure-env cl))))
                           (eval-body (closure-body cl) env)))))
;; data type for values
(define-struct value ())
(define-struct (closure value) (params body env))
(define-struct (cv value) (ctx vals default))
```

3.3.5 Applying the context function

A difference between explicit and implicit contextual values lies in the way the context function is applied. In Listing 1 we simply apply the function held in the contextual value structure: `((cv-ctx cv))`. Moving contextual values into the interpreter makes this approach incorrect. The context function in the structure represents a function of the base level, which must be evaluated at the base level. The interpreter must therefore evaluate the function syntactically:

```
(define (cv-evalctx cv)
 (let ((thunk (strict (cv-ctx cv))))
   (strict (eval-body (closure-body thunk)
                      (closure-env thunk)))))
```

Nothing prevents a context function to be itself a contextual value, so `cv-evalctx` reduces the value of the context function using `strict`, in order to be able to evaluate its body. Similarly, since the value returned by `cv-evalctx` is immediately used to look up in the value mapping of a contextual value, it needs to be an actual value. This is why the result of the context function application is also reduced[2].

3.3.6 Contextual values and parameter passing

The small Scheme language we consider has a call-by-value semantics, as explained in Section 3.3.1. This is ensured by `extend-env`, which allocates fresh store locations for holding the values of the arguments passed to a function. We have explained in Section 3.2.3 that we want to maintain call-by-value semantics even for implicit contextual values.

However, in the interpreter, contextual values are containers. So a naive implementation of parameter passing will overlook this fact and result in a language with call-by-reference semantics for contextual values (while keeping call-by-value semantics for actual values). To address this issue, we need to ensure that upon passing a contextual value as parameter, the *structure* of the contextual value is not shared between the caller and callee sites.

In other words, we need to deep copy the structure of a contextual value: the actual values, however, do not have to be copied. Considering a contextual value as a particular kind of tree, the copy operation consists in a deep copy of the internal nodes of the tree, and shallow copy of the outer nodes, hence sharing the leaves. This operation is performed by the `by-val` function, which is called when extending the definition-time environment of a function with bindings for the parameters (Listing 7).

3.4 Discussion

The presented language design adopts contextual values as real values in their own right from the point of view of the interpreter. This allows base programs to manipulate values without having to be aware that they are contextual. This mechanism therefore has the advantage to allow the provision of a restricted power over mutation to libraries and external programs, such that *some of their side effects* are restricted to a given context. However, the design still requires contextual values to be explicitly designated as such when they are first defined. The next section explores a more radical integration by which side effects on previously non-contextual values can be made local to certain contexts.

4. Implicit Scoping of Side Effects

This section explores a different approach to the integration of contextual values in a programming language: introducing a language construct for scoping all the side effects that occur in a given dynamic extent to a certain context. This can be used for instance to introduce an untrusted component in a system and ensure that it does not produce side effects visible to the rest of the system.

[2] The simple implementation of explicit contextual values of Listing 1 does not take into account the cases where the context function or its returned values are contextual.

4.1 A Language Construct for Scoped Assignments

We introduce a new construct to specify that side effects occurring during the execution of some expressions should be scoped according to a given context specification.

```
(scoped ctx-function exprs)
```

We say that the `scoped` construct defines a *scoped assignment region*. The boundaries of the region are dynamic since the region is the dynamic extent of the `scoped` block. Furthermore, a scoped assignment region is parametrized by a context function. The intuitive semantics is that, during the execution of *exprs*, all side effects are treated in a way that makes them local to the context specified by *ctx-function*. Internally, this means that all mutated values are turned into contextual values. If a value is already contextual, it is refined through nesting of contextual values.

Consider a simple operating system API for untrusted client programs. We can make sure that a client program will have no side effects visible to other users of the system:

```
(define (run-program p)
  (scoped current-user
    (load-n-run p)))
```

Here we assume that `current-user` is a function that returns the user that is currently requesting to run a program.

This toy example shows the benefit of the proposed construct: as a provider of the API with `run-program`, we have no idea of which accessible values (*e.g.* system properties) are not only read by the user program, but also mutated. If we were to ensure isolation of these side effects with explicit contextual values, we would have to explicitly define these values as contextual. In addition, it would be necessary to make sure the untrusted client program actually knows that these values are containers that should be accessed by means of accessor functions. Otherwise, any direct assignment would ruin the design. Using implicit contextual values, we would avoid the pitfalls of explicit contextual values, but we would still have to anticipate which values have to be defined as contextual. Conversely, here all values that are side effected by the program, and only those, are automatically made contextual, on a by-need basis.

4.2 Working out the Semantics

The introduction of the `scoped` construct raises a number of semantic issues that deserve special attention. First, if every single assignment in a scoped region is made contextual, mutating a value used in the computation of the context function of the region may potentially yield an infinite regression. Second, since `scoped` is an expression, scoped regions can naturally be defined in sequence, as well as nested (if not syntactically, at least dynamically). Thus we need to clarify the semantics of assignments in successive and nested regions. We now analyze these issues in more details using simple examples.

4.2.1 Avoiding infinite regression

Consider the following program:

```
(let ((user "Bilbo") (counter 0))
  (let ((user-ctx (lambda () user)))
    (scoped user-ctx
      (set! user "Totoro")
      (set! counter 1)
      counter))) ;; infinite loop
```

The context function `user-ctx` is used to scope side effects in a region of the program that happens to do an assignment to both the `user` and `counter` variables. These side-effects must therefore be scoped by making the associated values contextual. To this end, the actual value of `counter` must be determined: applying `user-ctx` should yield the value used to lookup its value for the current context. In this process the value associated to the `user` variable is needed, but it is also a contextual value, which happens to depend on... `user-ctx`! This results in an infinite loop.

This highlights that any value used in the evaluation of a context function application had better not be a contextual value on the same context function. Beyond throwing an error when this situation is detected, a possible approach is to define some values as *protected*: protected values can never be made contextual implicitly, their mutation is visible to all the referents of the value.

An alternative to explicitly denoting each and every protected value consists in providing a construct to delimit a protected region in the store, such that all values in that region are protected. While both designs are entirely legitimate, we opt for the latter. To do that, we introduce a `with-scope` construct that activates scoped assignments for the dynamic extent of its inner block, ensuring that all previously-defined values are protected during its execution:

```
(let ((user "Bilbo")) (let ((user-ctx (lambda () user)))
  (with-scope    ;; scoped assignments from here only
    (let ((counter 0))
      (scoped user-ctx
        (set! user "Totoro")
        (set! counter 1)
        counter)))))  ;;--> 1
```

The values associated to `user` and `user-ctx` are defined *above* the `with-scope` expression, so they are protected from being made contextual by any assignment performed in any scoped region below. Note that in order to ensure that the value of `counter` can be made contextual, we moved its definition inside the `with-scope` expression. `with-scope` expressions can be nested, and placed inside scoped regions as well. The semantics remains the same: values defined prior to it are protected from being made contextual during the dynamic extent of the body.

This design assumes a certain discipline according to which values that are used in determining a given context are defined prior to other values that are deemed to be dependent on this context.

4.2.2 Sequence of scoped regions

We now analyze the issues associated with multiple scoped regions, first considering sequences of non-nested scoped regions. Consider the following example:

```
(let ((counter 0))
  ...
  (scoped current-thread
    (set! counter (inc counter)))
  (scoped current-user
    (set! counter 100))
  ...
  counter)
```

After the first assignment, the value bound to `counter` is a thread-contextual value, with value 1 for the thread that executed the assignment (`t1`) and 0 otherwise. The second assignment makes the counter value be dependent on the user context: say, 100 if the user is “Totoro”. The question arises of how to compose the possible values of the counter and their associated contexts.

Since we are dealing with side effects, the reasonable semantics is that the most recent assignment has priority. This means that, regardless of which thread looks at the counter, if the current user is “Totoro”, then the value is 100. If not, then this means that the counter should be what it was prior to the last contextual assignment, that is, a thread local value. Fig. 3 illustrates the resulting contextual value.

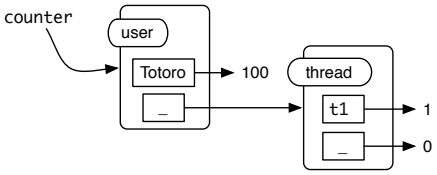

Figure 3. Sequence of scoped regions: the most recent assignment goes ahead.

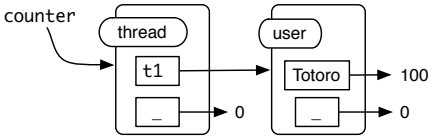

Figure 4. Nested scoped regions: the structure of the value reflects the nesting relation of regions.

Note that this is indeed a simple generalization of the semantics of scoped assignments. An assignment in a scoped region updates a value with a contextual value whose default is the previous value (whether it be contextual or not).

4.2.3 Nested scoped regions

As we have seen, when an assignment occurs in a scoped region, its effect is to *refine* the existing value for the considered context: creating a contextual value with the context function, apply the function and set the new value for the context value.

This refinement operation must be extended to support multiple nested contexts. Consider the following example, a variation of the previous one, where the scoped regions are now nested:

```
(let ((counter 0))
  ...
  (scoped current-thread
    (scoped current-user
      (set! counter 100)))
  ...
  counter)
```

Similarly to sequencing, nesting also defines a notion of precedence, but in a different manner: the region is first and foremost a region where assignments are made thread local. Within that scope, there is a further refinement that applies to the assignment of `counter`, that makes the new value 100 only visible depending on the current user context. This means that the above program returns 100 if and only if the current thread is the thread that executed the assignment *and* the current user is the one that was active at that time. In any other case, the value is 0. Fig. 4 illustrates how the nesting of contextual values reflects the nesting of scoped regions.

4.2.4 Mutating contextual values

Up to now, we have omitted the fact that when a mutation occurs within a scoped region, and the mutated value is already a contextual value, it may be that the context function of the outermost region is the same as the context function of the contextual value[3]. In this case, the contextual value should be updated (recall that this can imply either overwriting an existing mapping, or adding a new one), otherwise we would end up chaining contextual values with the same context function; this in turn would mean having to apply the same context function several times for the same value.

[3] Two functions are deemed equal either if they are the same actual value (this is the simple equality our interpreters use), or if they have the same source location and close over the same lexical environment (as implemented by the `eq?` primitive of AspectScheme [9]).

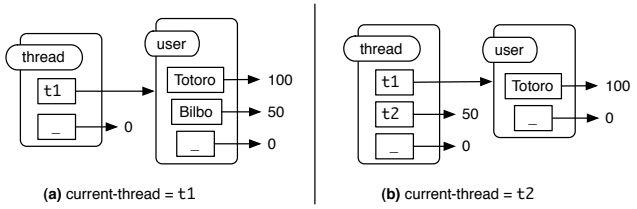

Figure 5. Updating an already-contextual value in a scoped region.

Furthermore, both sequencing and nesting of scoped regions can lead to contextual values being *compound*: a contextual value is compound if one of its value mapping contains another contextual value. Fig. 3 and 4 illustrate two different kinds of compound contextual values. This means that the assignment mechanism must recursively consider inner contextual values. For instance, consider the following extension to the previous example:

```
(let ((counter 0))
  ...
  (scoped current-thread
    (scoped current-user
      (set! counter 100)))
  ;; extension:
  (set! user "Bilbo")
  (scoped current-thread
    (set! counter 50))
  ...
  counter)
```

The contextual value associated to `counter`, depicted on Fig. 4, is already thread-contextual, and the assignment occurs in a scoped region on the `current-thread` context function. We should not treat that new assignment like we did for a sequence of scoped regions, because this would result in putting a new thread-contextual value in front of the already thread-contextual value. Instead, we should update the definition of the thread-contextual value accordingly. If the current thread is also `t1`, then the corresponding value is the user-contextual value, so we should update it recursively. Since the current user has changed, we add a new mapping for the "Bilbo" user (Fig. 5(a)). On the contrary, if the assignment is run in a different thread `t2`, then we must create a new mapping in the thread-contextual value for `t2` (Fig. 5(b)).

4.3 Interpretation

Now we have clarified different semantic issues of scoped assignment regions with contextual values, we turn to the actual interpretation of the proposed constructs.

4.3.1 New expressions

The core Scheme interpreter shown on Listing 6 is extended with two cases for the two new expressions we introduce, namely `scoped` and `with-scope`:

```
<expr> ::= ...
         | (scoped <expr> {<expr>}*)
         | (with-scope {<expr>}*)
```

To support scoped assignment regions, the interpreter maintains a list of currently-active scoped regions, in the form of a list of context functions. Evaluating a `scoped` expression means extending the list of currently-active scoped regions (except if the context function is the same as that of the last-entered region). To support protected store regions, the interpreter also maintains a reference to the environment that is active when a `with-scope` expression is

Listing 8 Interpretation of the new expressions.

```
 ... in the body of eval (Listing 6)...

((scoped-exp? exp)
 (let ((ctx (eval (scoped-exp-ctx exp) env)))
  (if (reentering-scope? ctx)
      (eval-body (scoped-exp-body exp) env)
      (fluid-let ((*scope* (cons ctx *scope*)))
        (eval-body (scoped-exp-body exp) env)))))

((with-scope-exp? exp)
 (fluid-let ((*protected* env))
  (eval-body (with-scope-exp-body exp) env)))
```

Listing 9 Interpretation of scoped assignment.

```
;; sets ref to nval in nested scope regions cs
(define (setval-s! ref nval cs)
 (let ((val (deref ref)))
   (if (and (cv? val) (eq? (cv-ctx val) (car cs)))
       (setval! (cv-to-update! val) nval (cdr cs))
       (setref! ref (refine val nval cs)))))

;; returns cv that refines v0 with nval in scopes cs
(define (refine v0 nval cs)
 (if (null? cs) nval
     (let ((cv (make-cv (car cs) '() (vector #f v0))))
       (cv-update! cv (refine v0 nval (cdr cs)))
       cv)))
```

entered: all values in the store that are accessible from this environment are considered protected during the extent of the block.

Both the list of active scoped regions and protected environment frames are associated to dynamic extents. Therefore, a typical design would be to pass them around in the `eval` procedure. To avoid too much cluttering however, we opt for the use of two global structures, `*scope*` and `*protected*`, which are redefined for dynamic extents using the `fluid-let` rebinding mechanism of PLT Scheme, which does precisely the dynamic binding we need. The interpretation of these expressions is shown on Listing 8.

4.3.2 Scoped assignment semantics

The semantics of reading a value is unchanged by the introduction of scoped assignment regions; it is as discussed in Section 3. The actual value of a contextual value at a given point in time is obtained by successively applying contextual functions and looking up in the mapping until a non-contextual value is reached.

The part of the semantics that is most affected by the introduction of scoped assignment is, unsurprisingly, the assignment function `setval!`. First of all, `setval!` now takes an extra argument, the list of context functions that represent the active scoped regions. In the rest of this text, for conciseness, we refer to this list as the *list of scopes*. The list is passed in reversed order of its construction, so that the context function of the outermost region is the first element in the list. Setting a value begins by discriminating whether the assignment to the given reference should be scoped:

```
(define (setval! ref nval cs)
 (if (not-scoped? ref cs)
     (setval-ns! ref nval)
     (setval-s! ref nval cs)))
```

The assignment should be scoped only if the list of scopes `cs` is not empty and if the reference `ref` is not pointing at a protected value. If the assignment is not scoped, the standard semantics is used (`setval-ns!` is identical to `setval!` presented in Section 3).

Setting a value in a scoped region is defined by `setval-s!` (Listing 9 top). First, if the referenced value is *not* a contextual value, then we do a refinement of the value for the list of scopes. The refinement function `refine` is a recursive procedure over the list of scopes (Listing 9 bottom). If the list is empty, the new value is itself its refinement. Otherwise, `refine` creates a new contextual value whose default is the current value, and whose value for the current context is the refinement of the value in the rest of scopes. This recursively chains contextual values in the order illustrated in Fig. 4.

If the referenced value given to `setval-s!` is a contextual value, then we need to check if the context function of the first contextual value is the same as the first function in the list of scopes. If this is the case, we call `setval!` recursively with the rest of the list of scope, passing it a reference to the value to update in the current contextual value. The `cv-to-update!` function applies the context function to get the current context value, and if there is a mapping for this context, it returns a reference to the associated value. If not, it adds a new mapping to the contextual value, associating the context to the default value, and returns a reference to that value.

This corresponds to the two cases illustrated in Fig. 5: in the first case (a), there is already a mapping in the contextual value for thread `t1`, so the associated value is recursively updated, ending up in a new mapping in the nested contextual value; in the second case (b), the current thread is different, so a new mapping is added to the first contextual value.

4.4 Discussion

This section has presented a much deeper and interesting level of integration of contextual values in the programming language. Conceptually, what we obtain is a context-dependent store of values, which to our knowledge is something unavailable in any language that supports mutation. The particularity of the design based on contextual values in that the contextual store is built on a by-need basis: only the values that are mutated within a scoped assignment region are made contextual.

The question can be raised of how many values have to be made contextual and what the performance impact of such an approach might be. This first of all depends on how much side effects are relied upon in the considered programming language or style. For instance, in impure functional languages like Scheme, which have side effects, good programming practice is to maximize purely functional procedures. Also, to be fair, any comparison would have to be between otherwise-equivalent implementations in terms of functionality: if context dependence is needed, it has to be implemented in one way or another. Another question to consider is the kind of optimizations that can be made. For instance, it would be possible to avoid making contextual all the values that have been first defined in the scoped region itself, and to use information about contexts that can never occur again to garbage collect some values. The implementation we have presented is on purpose fairly naive to be as close as possible to a clean operational semantics description.

5. Related Work

Several approaches to obtain dynamic bindings have been proposed. The foremost example is that of dynamically-scoped "special" variables of Common Lisp [21]: a global variable introduced by `defvar` can be rebound for a certain dynamic extent in a `let` form. The `fluid-let` macro of PLT Scheme [10] simulates this local dynamic rebinding by side effecting the binding; as a consequence issues can occur in multi-threaded programs. Conversely, virtually all multi-threaded implementations of Common Lisp perform dynamic local rebinding of special variables in a thread-local way. The `dletf` framework of Costanza [6] introduces thread-local

dynamic rebinding for any variable, by hooking into the special variable mechanism of Common Lisp.

While the above approaches work on bindings, other approaches to dynamic binding rely on particular *values* that are containers simulating the dynamic binding, at the cost of explicit accessor operations to be used by clients. An example is the fluid bindings (`fluid` and `let-fluid`) of Scheme 48 [15]. This is similar to the design of explicit contextual values we have presented. Thread-local versions of explicit fluid bindings have been proposed in several languages under different names: thread fluids in scsh [12], parameters in PLT Scheme [10], contextual variables in PyContext [25]. All these are containers whose encapsulated values are both thread-local and re-definable for a dynamic extent.

All these approaches to dynamic binding target at most two notions of what "context" is: either a given dynamic extent, or the executing thread. Contextual values are more general: a context function is a first-class function that can use any computationally-accessible information.

There are several proposals that are more explicitly targeted to context-dependent adaptation. Some deal with adaptation of program structure, such as ContextL with layered classes [7], *i.e.* classes whose structure can be adapted for certain dynamic extents. Others deal with adaptation of key elements of the behavioral properties of a program, such a method dispatch in object-oriented systems. A general approach in this category is predicate dispatch [24, 19], and more specifically context-oriented proposals like contextual dispatch [26], or more recently, the subjective multi-methods of the Ambiance language [13]. Aspect-oriented programming using pointcuts and advices [27] deals with execution events, and can also be used to address context-dependent adaptation [23].

Finally, several approaches exist that provide interesting scoping strategies for the refinements and adaptations they permit to express. For instance, Classboxes [2] is a module system that provides open classes [5, 28] whose refinements are scoped to certain client modules; ContextL [7] provides mixin layers that can be activated only for a certain dynamic scope. While these approaches deal with program code, there also exists several proposals for controlling the application of behavioral changes in aspects [9, 22].

However, in all the approaches we have mentioned, *side effects* caused by the execution of the refinements are visible *outside* their scope of activation. Scoped assignment regions as we proposed with contextual values make it possible to effectively restrict the visibility of these side effects to certain contexts. For instance, if a dynamic layer in ContextL is deployed within a `scoped` expression, its side effects will only be visible in a given context (which can be based on the layer identity, or any other criteria). Therefore, we believe our proposal complements the above proposals.

6. Conclusion and Perspectives

We propose a new language construct for context-oriented programming: contextual values, that is, values that change depending on the context in which they are looked at and modified. Starting from a trivial generalization of thread-local values –a particular kind of contextual value that is well known and widely used–, we have explored more integrated language support for contextual values. We have discussed the issues associated with providing implicit contextual values in a call-by-value language. The culmination of this exploration is the proposal of a construct for scoped assignment regions, whereby side effects performed in a certain dynamic extent can be made local to certain contexts. We have illustrated our proposal and its semantics through Scheme examples and interpreters. By focusing on the issue of shared state, contextual values complement existing approaches for context-dependent behavior adaptation.

As future work, we want to specify contextual values more formally using an extension of the recently-proposed operational semantics of Scheme [17], and explore their formulation in an object-oriented context. This work exposes the design of contextual values, but does not go far enough in their applications. In this line, it is also interesting to look at how implicit contextual values can be integrated in mature languages like Scheme and Common Lisp. In particular, studying the complementarity of contextual values with other context-oriented constructs in ContextL seems attractive. Also, there are various interesting applications of contextual values that deserve more attention, such as security (*e.g.* sandboxing the store), and, in languages where code itself is a value, program definitions can be made context-dependent.

Acknowledgments. We would like to thank Pascal Costanza for interesting discussions and references, Thomas Cleenewerck, Johan Fabry, Robby Findler, Guillaume Pothier, and the anonymous DLS reviewers for their comments on this work.

Availability. The code of the interpreters presented in this paper, along with examples, is available at:
`http://pleiad.dcc.uchile.cl/research/scope`

References

[1] M. Baldauf and S. Dustdar. A survey on context-aware systems. Technical Report TUV-1841-2004-24, Technical University of Vienna, 2004.

[2] Alexandre Bergel, Stéphane Ducasse, and Oscar Nierstrasz. Classbox/J: Controlling the scope of change in Java. In *Proceedings of the 20th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2005)*, pages 177–189, San Diego, California, USA, October 2005. ACM Press. ACM SIGPLAN Notices, 40(11).

[3] Licia Capra, Wolfgang Emmerich, and Cecilia Mascolo. Reflective middleware solutions for context-aware applications. In Akinori Yonezawa and Satoshi Matsuoka, editors, *Proceedings of the 3rd International Conference on Metalevel Architectures and Advanced Separation of Concerns (Reflection 2001)*, volume 2192 of *Lecture Notes in Computer Science*, pages 126–133, Kyoto, Japan, September 2001. Springer-Verlag.

[4] Keith Cheverst, Christos Efstratiou, Nigel Davies, and Adrian Friday. Architectural ideas for the support of adaptive context-aware applications. In *Workshop on Infrastructure for Smart Devices - How to Make Ubiquity an Actuality*, Bristol, UK, September 2000.

[5] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava: Modular open classes and symmetric multiple dispatch in java. In *Proceedings of the 15th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2000)*, pages 130–145, Minneapolis, Minnesota, USA, October 2000. ACM Press. ACM SIGPLAN Notices, 35(11).

[6] Pascal Costanza. How to make Lisp more special. In *Proceedings of International Lisp Conference*, Stanford, CA, USA, June 2005.

[7] 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.

[8] A. K. Dey and G. D. Abowd. Towards a better understanding of context and context-awareness. In *Workshop on the What, Who, Where, When, and How of Context-Awareness, as part of the 2000 Conference on Human Factors in Computing Systems (CHI 2000)*, The Hague, The Netherlands, April 2000.

[9] 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.

[10] Matthew Flatt. PLT MzScheme: Language manual, 2007. Version 372.

[11] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages (2nd ed.)*. The MIT Press, 2001.

[12] Martin Gasbichler and Michael Sperber. Processes vs. user-level threads in scsh. In *3rd Workshop on Scheme and Functional Programming*, October 2002.

[13] Sebastián González, Kim Mens, and Patrick Heymans. Highly dynamic behaviour adaptability through prototypes with subjective multimethods. In *Proceedings of the ACM Dynamic Languages Symposium (DLS 2007)*, pages 77–88, Montreal, Canada, October 2007. ACM Press.

[14] J. Kephart. A vision of autonomic computing. In *Onward! Track at OOPSLA 2002*, pages 13–36, Seattle, WA, USA, 2002.

[15] Richard A. Kesley and Jonathan A. Rees. A tractable Scheme implementation. *Lisp and Symbolic Computation*, 7(4):315–335, 1995.

[16] Shriram Krishnamurthi. *Programming Languages: Application and Interpretation*. 2007. Version 2007-04-26.

[17] Jacob Matthews and Robert Bruce Findler. An operational semantics for Scheme. *Journal of Functional Programming*, 18(1):47–86, January 2008.

[18] P. K. McKinley, S. M. Sadjadi, and B. H Kasten, Cheng. Composing adaptive software. *IEEE Computer*, 37(7):56–64, July 2004.

[19] Todd Millstein. Practical predicate dispatch. In *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2004)*, pages 345–364, Vancouver, British Columbia, Canada, October 2004. ACM Press. ACM SIGPLAN Notices, 39(11).

[20] Kurt Schelfthout, Tom Holvoet, and Yolande Berbers. Views: Middleware abstractions for context-aware applications in manets. In *5th International Workshop on Software Engineering for Large-scale Multi-Agent Systems*, 2005.

[21] Guy Steele. *Common Lisp the Language, 2nd Edition*. Digital Press, 1990.

[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, Kris Gybels, Marcus Denker, and Alexandre Bergel. Context-aware aspects. In Welf Löwe and Mario Südholt, editors, *Proceedings of the 5th International Symposium on Software Composition (SC 2006)*, volume 4089 of *Lecture Notes in Computer Science*, pages 227–242, Vienna, Austria, March 2006. Springer-Verlag.

[24] Aaron Mark Ucko. Predicate dispatching in the Common Lisp Object System. Technical Report AITR-2001-006, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Cambridge, MA, 2001.

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

[26] Robert J. Walker and Gail C. Murphy. Implicit context: Easing software evolution and reuse. In *Proceedings of the 8th International ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE-8)*, pages 69–78, San Diego, CA, USA, 2000. ACM Press.

[27] 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.

[28] Daniel Weinreb and David Moon. Flavors: Message passing in the Lisp machine. A.I. Memo 602, Massachussetts Institute of Technology, Artificial Intelligence Laboratory, 1980.

[29] Mark Weiser. Some computer science issues in ubiquitous computing. *Communications of the ACM*, 36(7):75–84, July 1993.