Open Closures

Disclosing lambda's inner monomaniac object!

Stefan Monnier
monnier@iro.umontreal.ca
Université de Montréal
Département d'Informatique et Recherche Opérationnelle
Montréal, QC, Canada

ABSTRACT

While folklore teaches us that closures and objects are two sides of the same coin, they remain quite different in practice, most notably because closures are opaque, the only supported operation being to call them.

In this article we discuss a few cases where we need functions to be less opaque, and propose to satisfy this need by extending our beloved λ so as to expose as sorts of record fields some of the variables it captures. These *open closures* are close relatives of CLOS's *funcallable objects* as well as of the *function objects* of traditional object-oriented languages like Java, except that they are functions made to behave like objects rather than the reverse.

We present the design and implementation of such a feature in the context of Emacs Lisp.

CCS CONCEPTS

• **Software and its engineering → Data types and structures**; **Procedures, functions and subroutines**; **Functional languages**; *Object oriented languages*; Integrated and visual development environments.

KEYWORDS

Functional programming, Function objects, Translucent functions, Emacs Lisp

ACM Reference Format:
Stefan Monnier. 2022. Open Closures: Disclosing lambda's inner monomaniac object!. In *Proceedings of the 15th European Lisp Symposium (ELS'22).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.5281/zenodo.6228797

1 INTRODUCTION

Undergraduate programming language courses will often point out that one can implement objects (in the *object-oriented* meaning of the term) as functions, e.g. by making them take a "method name" as a first argument and dispatching to different behaviors based on that argument. Yet if we try to take the idea seriously, one quickly encounters significant drawbacks, whether it's because of efficiency concerns, or because of the difficulty to give static types to the resulting code, or the inability to determine if a function obeys this convention before calling it, or any number of other issues that may come up.

The reverse is a somewhat simpler story: an object can be used to implement a function, by simply arranging for that object to have just one method, variously called `run`, `call`, `exec`, or `apply`. Depending on the language, this can be syntactically cumbersome and verbose, and may sometimes require to explicitly specify the captured variables, but in terms of efficiency at least not much is lost by treating a function as an object limited to a single method (often called a *function object*).

So while in the world of object-oriented languages, it is very common to add support for functions by encoding them as *function objects*, in the world of functional programming languages objects are usually not encoded as functions but as tuples. Disregarding issues of aesthetics, the result may appear to be just as good since we get both functions and objects in either case. Yet, *function objects* actually provide a bit more flexibility because they are *simultaneously* functions and objects, which has no equivalent in the world of functional programming languages.

A notable difference between functions and objects in this respect is that functions are opaque: the only non-trivial operation allowed on a function is to call it, but calling a function is a very risky business if we don't know what kind of function we're dealing with. This is usually not a problem because the responsibility is traditionally on the code that provides the function to provide one that works adequately, not on the code that calls it. But *function objects* can offer more flexibility since they may come with a type and may also expose object attributes that can be read via accessors, so while most attributes as well as the code of their sole method may be just as opaque as that of a function, the object itself can reveal extra information when desired.

In this article, we will discuss some situations where this kind of information is needed, and based on those we show the design of *open closures* which are an extension of the usual functions with extra information exposed in the form of a type and a set of *slots* that can be reached via accessors. Good old λ can then be redefined as a bare-bones open closure whose type is trivial, with an empty set of slots. Note that the types used to classify those open closures fundamentally constrain the set of slots exposed. These can be seen as a constraint on the captured environment of closures, and should not be confused with the notion of type used more traditionally to classify functions according to their signature, i.e. the set of arguments that the function accepts and the values it returns. Those two notions of type are orthogonal and in this article we will not discuss the types in the sense of function signatures.

ELS'22, March 21–22, 2022, Porto, Portugal

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.5281/zenodo.6228797

```
(defun compose-function (function where orig-fun)
  (cond
   ((eq where :override) function)
   ((eq where :before)
    (lambda (&rest args)
      (apply function args)
      (apply orig-fun args)))
   ((eq where :after)
    (lambda (&rest args)
      (apply orig-fun args)
      (apply function args)))
   ((eq where :around)
    (lambda (&rest args)
      (apply function orig-fun args)))))

(defun add-function (function where var)
  (set var (compose-function function where
                             (symbol-value var))))
```

Figure 1: Adding functions to a variable

2 MOTIVATING EXAMPLE

In this section we will see the main example that will help explain the design of our open closures.

In Emacs, we have many variables holding functions that are called in various circumstances, in order to be able to customize the behavior of commands. We generally call them *hooks*, but you can just as well think of them as callbacks. They take various forms, but the form of interest here is when a global variable (or an object's slot) holds a single function.

When a *package* (the name we give to plugins, in Emacs) wants to affect the corresponding behavior, it will want to modify the function stored in this variable by composing the old and the new function. We could provide that functionality as shown in Figure 1.

This would work fine but comes with an annoyance and a serious problem. The annoyance is that when we try to debug this code, the composed function will not show us what it is made of, even if the provided `function` and the original `orig-fun` are named functions. It requires trained eyes looking at the innards of the closure to decipher what it is made of and reverse engineer where it may come from.

But the more serious problem comes when the package decides it does not want to modify that variable any more and hence wants to undo its changes. The easy solution is to stash the old value somewhere so we can restore it afterwards, but that only works if all the packages add and remove their modifications in a properly nested order, which is neither enforced nor desirable. For example, after:

```
(defvar my-var #'A)
(add-function #'B :after 'my-var)
(add-function #'C :after 'my-var)
```

`my-var` will hold an anonymous function which first calls `A`, then `B`, then `C`. And if the package that added `B` has buyer's remorse, it would like to be able to do:

```
(remove-function #'B 'my-var)
```

$$
\begin{array}{llll}
\textit{(index)} & i & \in & \mathbb{N} \\
\textit{(expressions)} & e & ::= & c \mid x \mid (\text{olam } \overrightarrow{(x\ e)}\ (x_a)\ e_b) \\
 & & \mid & (\text{oapp } e_1\ e_2) \mid (\text{oref } e\ i)
\end{array}
$$

Figure 2: Syntax of the *open lambda calculus*

After this call, we would like `my-var` to hold a function which calls `A` and then `C`, but there is no mechanism which would let `remove-function` extract the necessary information from `my-var` to construct this new function, because the function stored there is opaque.

Ideally, we would like to be able to test whether a given function is one of the wrappers built by `add-function`, and if so, we would like to be able to extract the "`function`" and the "`orig-fun`" from which they were built, as well as "`where`" they were composed.

In current functional programming language, this can only be achieved with a significant amount of extra work, and often with additional runtime costs when calling the function, such as an additional indirection if one uses a CLOS-style *funcallable object* [Kiczales et al. 1991]. This is particularly frustrating considering that the most common internal representation of those closures makes the corresponding information readily available, if only one were given a way to access it.

3 OPEN CLOSURES

We propose to solve the previous problem by opening up our lambda abstractions such that some of the captured variables can also be accessed from outside, like the slots of a tuple. The result is fundamentally a combination of a tuple and a function. It can be seen as a function with slots, or as a tuple with code.

Figure 2 shows the syntax of an *open lambda calculus* which exposes the core idea in a minimalist way. We use the convention that $\overrightarrow{m}$ is a shorthand for $(m_1 \ldots m_n)$, and $\overrightarrow{(a\ b)}$ is a shorthand for $((a_1\ b_1) \ldots (a_n\ b_n))$.

- c stands for a builtin constant.
- x is the usual variable reference.
- $(\text{oapp } e_1\ e_2)$ is your usual function application, limited to a single argument without loss of generality.
- $(\text{olam } \overrightarrow{(x\ e)}\ (x_a)\ e_b)$ constructs a function, where (x_a) is the list of arguments, again limited to a single argument, e_b is the body, and $\overrightarrow{(x\ e)}$ is the (ordered) list of slots. The slots are accessible both internally and externally. For internal access, e_b can refer to the value of those slots using the slot's name as a variable.
- $(\text{oref } e\ i)$ fetches the value of the i^{th} slot of the function e. The index is an immediate value rather than an expression only for the purpose of simplifying the static semantics of the language: in a dynamically typed language, i can be generalized to an arbitrary expression evaluating to an integer.

This calculus is a superset of the standard λ-calculus since we can encode "$\lambda x.e$" as $(\text{olam } ()\ (x)\ e)$ which are those functions that expose no slots, and hence upon which we cannot apply any "oref".

And while we can of course encode the usual tuples $(e_1, \ldots, e_n)$ using a Church-style encoding, we can also encode them more

$$
\begin{array}{rrcl}
\textit{(values)} & v & ::= & c \mid x \mid (\text{olam } \overrightarrow{(x\ v)}\ (x_a)\ e) \\
\textit{(ctxts)} & E & ::= & \bullet \mid (\text{oapp } E\ e) \mid (\text{oapp } v\ E) \mid (\text{oref } E\ i) \\
& & \mid & (\text{olam } ((x_1\ v_1) \ldots (x_i\ E) \ldots (x_n\ e_n))\ (x_a)\ e_b)
\end{array}
$$

$e \rightsquigarrow e'$	Small-step reduction of e to e'

$$\frac{}{(\text{oapp } (\text{olam } \overrightarrow{(x\ v)}\ (x_a)\ e)\ v_a) \rightsquigarrow e[\overrightarrow{v}, v_a/\overrightarrow{x}, x_a]}\ (\beta)$$

$$\frac{}{(\text{oref } (\text{olam } \overrightarrow{(x\ v)}\ (x_a)\ e)\ i) \rightsquigarrow v_i}\ (\pi) \qquad \frac{e \rightsquigarrow e'}{E[e] \rightsquigarrow E[e']}$$

$$\frac{x' \notin \text{fv}(e)}{(\text{olam } \overrightarrow{(x\ e)}\ (x_a)\ e_b) \rightsquigarrow (\text{olam } \overrightarrow{(x\ e)}\ (x')\ e_b[x'/x_a])}\ (\alpha_1)$$

$$\frac{x' \notin \text{fv}(e) \qquad x' \notin \overrightarrow{x}, x_a}{\begin{array}{l}(\text{olam } ((x_1\ e_1) \ldots (x_i\ e_i) \ldots (x_n\ e_n))\ (x_a)\ e_b) \\ \rightsquigarrow (\text{olam } ((x_1\ e_1) \ldots (x'\ e_i) \ldots (x_n\ e_n))\ (x_a)\ e_b[x'/x_i])\end{array}}\ (\alpha_2)$$

Figure 3: Dynamic semantics of the *open lambda calculus*

directly as functions of the form $(\text{olam } ((\_\ e_1) \ldots (\_\ e_n))\ (x)\ x)$ where the traditional projection operation "$e.i$" is just $(\text{oref } e\ i)$.

3.1 Dynamic semantics

Figure 3 shows the corresponding dynamic semantics, with a call-by-value reduction strategy. The top of the figure defines the syntax of values v, which are a subset of valid expressions, as well as the syntax of evaluation contexts E which define where evaluation can take place in an expression. The semantics is defined as the small step relation $e \rightsquigarrow e'$. Rule β shows hows the slot values are substituted into the body of a function, making them available internally, while rule π shows how oref accesses a slot's value from outside. The contexts E together with the congruence rule Cong show where primitive reductions can take place and define a left-to-right evaluation order. The two α renaming rules, where fv returns the free variables of a term, are only intended to give further details about the intended semantics.

Notice that the access to slots is done by position rather than by name. In other words, slot names are only meaningful internally when accessing them from within the body of the function and are not exposed outside of the function, so they obey the usual α-renaming of variable bindings as evidenced by the α_2 rule. This simplifies the metatheory and lets us rely on the usual conventions to avoid issues linked to name capture [Urban et al. 2007].

Note also the absence of an η rule $(\text{olam ?}\ (x)\ (\text{oapp } e\ x)) \rightsquigarrow e$ because what to put into "?" depends on the slots exposed by e. In other words, while olam encodes the usual λ, it does not enjoy the same η-reduction rule.

3.2 Typing rules

While open closures were developed in the context of a dynamically typed language, they would work just as well in a statically typed context.

$$
\begin{array}{rrcl}
& \tau & ::= & \text{Int} \mid \ldots \mid (\text{oarw } \overrightarrow{\tau}\ \tau_a\ \tau_r) \\
& \Gamma & ::= & \bullet \mid \Gamma, x:\tau
\end{array}
$$

$\Gamma \vdash e : \tau$	e has type τ in environment Γ

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\ (\textsc{Var}) \qquad \frac{\Gamma \vdash e : (\text{oarw } \overrightarrow{\tau}\ \tau_a\ \tau_r)}{\Gamma \vdash (\text{oref } e\ i) : \tau_i}\ (\textsc{Ref})$$

$$\frac{\Gamma \vdash e_1 : (\text{oarw } \overrightarrow{\tau}\ \tau_a\ \tau_r) \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash (\text{oapp } e_1\ e_2) : \tau_r}\ (\textsc{App})$$

$$\frac{\Gamma, \overrightarrow{x:\tau}, x_a:\tau_a \vdash e_b : \tau_r \qquad \forall i.\ \ \Gamma \vdash e_i : \tau_i}{\Gamma \vdash (\text{olam } \overrightarrow{(x\ e)}\ (x_a)\ e_b) : (\text{oarw } \overrightarrow{\tau}\ \tau_a\ \tau_r)}\ (\textsc{Lam})$$

Figure 4: Static semantics of the *open lambda calculus*

To make that clear, Figure 4 shows a possible simple type system for our calculus. Even without a need for static types, those rules can be helpful to clarify the intended semantics. The top of the figure defines the syntax of type environments Γ and of types τ, which can include any number of builtin types plus the new type of open closures that we denote as $(\text{oarw } \overrightarrow{\tau}\ \tau_a\ \tau_r)$ which is the type of functions that take an argument of type τ_a, return a result of type τ_r, and expose slots of type $\overrightarrow{\tau}$. Just like open closures are a fusion of a function and a tuple, these function types are a fusion of the traditional function types and the traditional tuple types.

The typing judgment has the form $\Gamma \vdash e : \tau$. The typing rules reflect the dual nature of our open closures as both functions and tuples: the App rule is the same as the corresponding rule in the simply typed λ-calculus, expect for the extra $\overrightarrow{\tau}$ annotation in $(\text{oarw } \overrightarrow{\tau}\ \tau_a\ \tau_r)$ which is simply ignored, and the Ref rule similarly matches the classic rule for the operation that projects a specific slot from a tuple, except for the extra τ_a and τ_r annotations on $(\text{oarw } \overrightarrow{\tau}\ \tau_a\ \tau_r)$ which are similarly ignored. The more interesting rule is Lam: the right part of the premises corresponds to the usual premise for the construction of tuples, but the left part does not quite match the premise for the typing rule of the usual λ constructor because the body e_b is now typed in an environment that includes not only the argument x_a but also all the open closure's slots $\overrightarrow{x}$.

3.3 Compilation

Looking at the syntax and semantics of the open lambda calculus, one may wonder why it makes sense to introduce these open closure objects with their dual tuple/function nature, since both the dynamic and the static semantics suggest that the result is not much simpler than if we had introduced tuples and functions separately.

The real motivation becomes apparent only once we consider the usual implementation of closures via closure conversion. Closure conversion turns closures into tuples which contains a reference to the code of the function plus the values of all the variables captured by the function. Our open closures take advantage of this representation to store their extra slots alongside the values of the captured variables. This way, a degenerate open closure with zero slots ends up represented exactly as a normal λ would, and the only

$$
\begin{array}{l}
C[\![x]\!]_\sigma \quad\quad = \sigma(x) \\
C[\![e_1\ e_2]\!]_\sigma \quad = (\text{let } x\ C[\![e_1]\!]_\sigma\ (\text{call } (\text{ref } x\ 0)\ x\ C[\![e_2]\!]_\sigma)) \\
C[\![\lambda x_a.e_b]\!]_\sigma = (\text{tuple } (\text{code } (x_c\ x_a)\ C[\![e_b]\!]_{\sigma_e}) \\
\qquad\qquad\qquad\qquad\quad \sigma(y_1)\ \ldots\ \sigma(y_m)) \\
\quad \text{where } \overrightarrow{y} = \text{fv}(e_b) - \{x_a\} \\
\qquad\quad x_c \text{ is fresh} \\
\qquad\quad \sigma_e = \{x_a \mapsto x_a, \\
\qquad\qquad\quad y_1 \mapsto (\text{ref } x_c\ 1), \ldots, y_m \mapsto (\text{ref } x_c\ m)\}
\end{array}
$$

$$
\begin{array}{l}
C[\![x]\!]_\sigma \quad\quad\quad\quad\quad\quad = \sigma(x) \\
C[\![(\text{oref } e\ i)]\!]_\sigma \quad\quad\quad = (\text{ref } C[\![e]\!]_\sigma\ i) \\
C[\![(\text{oapp } e_1\ e_2)]\!]_\sigma \quad\quad = (\text{let } x\ C[\![e_1]\!]_\sigma\ (\text{call } (\text{ref } x\ 0)\ x\ C[\![e_2]\!]_\sigma)) \\
C\left[\!\!\left[(\text{olam } \overrightarrow{(x\ e)}\ (x_a)\ e_b)\right]\!\!\right]_\sigma = (\text{tuple } (\text{code } (x_c\ x_a)\ C[\![e]\!]_{\sigma_e}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad e_1\ \ldots\ e_n\ \sigma(y_1)\ \ldots\ \sigma(y_m)) \\
\quad \text{where } \overrightarrow{y} = \text{fv}(e_b) - \{x_a, x_1, \ldots, x_n\} \\
\qquad\quad x_c \text{ is fresh} \\
\qquad\quad \sigma_e = \{x_a \mapsto x_a, \\
\qquad\qquad\quad x_1 \mapsto (\text{ref } x_c\ 1), \ldots, x_n \mapsto (\text{ref } x_c\ n), \\
\qquad\qquad\quad y_1 \mapsto (\text{ref } x_c\ n{+}1), \ldots, y_m \mapsto (\text{ref } x_c\ n{+}m)\}
\end{array}
$$

Figure 5: Example of closure conversion
On the left, the algorithm for a plain λ-calculus and on the right the algorithm for our *open lambda calculus*.

cost of adding slots to an open closure is to increase the size of the tuple. It does not introduce any extra indirection nor add any extra cost when the function is called.

Let's write $C[\![e]\!]_\sigma$ the closure conversion of expression e where σ is a substitution used to remember how to access the free variables of e. And let's assume the following lower-level language for the target of the closure conversion:

$$
\begin{array}{lrcl}
\textit{(exps)} & e & ::= & c \mid x \mid (\text{call } e_1\ e_2\ e_3) \mid (\text{code } (x_1\ x_2)\ e) \\
& & \mid & (\text{let } x\ e_1\ e_2) \mid (\text{tuple } \overrightarrow{e}) \mid (\text{ref } e\ i)
\end{array}
$$

In this language (code $(x_1\ x_2)\ e$) denotes a chunk of closed code that could hence be represented as a pointer to piece of machine code. Without loss of generality, we limited this language to have only functions (and functions calls) of exactly two arguments.

Figure 5 shows what the closure conversion algorithm may look like, first for the plain $\lambda - calculus$, and then for open closures, where we highlighted the parts that are affected by the slots of open closures. As you can see, in both cases a function is converted to something of the form (tuple (code ...) ...) and the only significant change is the addition of one extra value per slot into the tuple.

We place the extra slots of the open closure at the beginning of the tuple, which thus push the values of captured variables to later slots of the tuple. This is done to make it easy to find the exposed slots of the open closure since it is independent of the number of captured variables. This choice is not the only one, of course, but it is the simplest here and makes for an efficient implementation of (oref e i). An alternative would be to place the captured variables first and the extra slots later, in which case (oref e i) would need to know the number of captured variables of the closure e, which could be stored for example just before the beginning of the code, so as not to increase the size of the tuples.

More generally, there are many other ways to represent closures than the *flat closures* used in this algorithm, and open closures do not impose the use of flat closures. The only requirement is that the extra slots's values be stored somewhere and that (oref e i) be able to find those values, potentially with the help of some extra information stored somewhere in the closure e, such as alongside its code.

4 OCLOSURES IN EMACS LISP

The calculus in the previous section shows the core idea of open closures in a minimalist setting, but we have designed and implemented open closures in the context of the Emacs Lisp language, so we discuss here what such a functionality can look like in a real-life setting.

Emacs Lisp is a programming language that lacks any namespace management features, so every globally-visible definition is instead given a name which includes a "package prefix". In the case of open closures, we chose the prefix "`oclosure-`" for its definitions and we call its (open) closures "OClosures".

The most important difference between the previous calculus and OClosures is that additionally to carrying values in slots, OClosures come with a type. This can be thought of as forcing every open closure to have an extra slot, placed first, which contains that runtime type information. In practice it is implemented differently, because for technical reasons we decided to store that type in a different place than the first slot.

So the constructor of OClosures has the following form:

$$(\texttt{oclosure-lambda}\ (\textit{type}\ .\ \overrightarrow{(x\ e)})\ \textit{args}\ e_b)$$

Where *args* follows the usual format of Emacs Lisp formal arguments. The type can be retrieved with `oclosure-type`. We added this type information so as to be able to perform type tests and type-based dispatch, by integrating the feature with the rest of our CLOS-inspired object system.

For example, in the case of the composed functions presented in Section 2, we called the type of those OClosures `advice`. This is useful in `remove-function` where can now distinguish the case where `myvar` contains an `advice`, so we know we can look at its slots to find its component functions. It also lets us change the printer by defining a new method which dispatches on the specializer `advice` so as to print those functions in a more human-friendly way.

4.1 OClosure types

Of course, before using a new type, we need to define it. While the types of open closures in Section 3.2 constrain both the set of slots and the signature of the function, OClosure types leave the arity and return types unconstrained, and only specify their slots. While positional access to slots was convenient for our little calculus, it

```
(defun uncompose-function (function listfunc)
  (if (not (eql (oclosure-type listfunc) 'advice))
      listfunc                ;; Nothing to remove.
    (let ((sva (slot-value listfunc 'car))
          (svd (slot-value listfunc 'cdr))
          (svw (slot-value listfunc 'where)))
      (if (eql sva function)
          svd                 ;; Found it!
        (compose-function
         sva svw
         (uncompose-function function svd))))))

(defun remove-function (function var)
  (set var (uncompose-function function
                               (symbol-value var))))
```

Figure 6: Removing a function from a variable

is more convenient in real life to be able to access slots by name. Type definitions thus indicate the list of slots that are to be included for OClosures of that type. They work very much like Common Lisp's `defstruct` and `defclass`. The syntax is loosely based on `defstruct`:

(`oclosure-define` (*name* . *props*) . *slots*)

Where *slots* is the list of slots included in this type, where each slots can come with some extra information, the only such extra information currently used is whether it's mutable or not, the default being for slots to be immutable.

The type's properties specified in *props* can include a list of parents, which allows subtyping, including multi-inheritance.

4.2 OClosure copies

Going back to our motivating example from Section 2, the function `add-function` can now create OClosures which work just as well as the old functions, but with extra information easily available. We can define the type `advice` of those OClosures as follows:

```
(oclosure-define (advice) car cdr where)
```

We chose `cdr` and `car` as the name of the slots holding resp. the original function and the added function because the repeated addition of functions creates a list structure.

The type information and the now exposed slots make it now possible for `remove-function` to do its job, by finding out the new set of functions to compose and reconstruct a new function after removing some element, as shown in Figure 6.

While this does work, we can do better if we consider that the last 4 lines of `uncompose-function` construct the same function as `listfunc`, except with a different `cdr`. For such use-cases, we have added the ability to perform functional updates of OClosures. We call them copiers. For example the previous type definition can be changed to:

```
(oclosure-define (advice
                  (:copier advice-with-cdr (cdr)))
  car cdr)
```

This defines a new function `advice-with-cdr` which will take an `advice` as first argument and any function as second argument and will return a new `advice` identical to the first except that its `cdr` slot will contain the function provided as second argument. With this function, we can simplify `uncompose-function` to:

```
(defun uncompose-function (function listfunc)
  (if (not (eql (oclosure-type listfunc) 'advice))
      listfunc                ;; Nothing to remove.
    (let ((sva (slot-value listfunc 'car))
          (svd (slot-value listfunc 'cdr)))
      (if (eql sva function)
          svd                 ;; Found it!
        (advice-with-cdr
         listfunc
         (uncompose-function function svd))))))
```

Notice that we did not need the `where` slot any more nor did we have to call `compose-function` any more. A side effect is that this code is more efficient because it can blindly copy all the bits of `listfunc` and then just change the `cdr` slot, although this was not the motivation since speed of `remove-function` is not a concern.

OClosure copiers offer a second way to construct OClosures (besides `oclosure-lambda`) and they offer a limited way in which one can access the still opaque content of a closure, in the sense that they read the slots of the tuple containing the reference to the code and the values of captured variables that are not directly exposed as OClosure slots.

It should be noted that they impose an additional constraint on the system, in the sense that in order to be able to perform such a functional update, it is imperative that we be able to find all the places where the content of a slot are stored in the closure. In most closure representations, this is not a problem since the value of each captured variable is only stored in a single place, but there are exceptions such as when using run-time code generation to specialize the code of a closure to the particular values of the variables it captures [Lee and Leone 1996], or when the compiler notices that a captured variable always has the same value and decides to apply constant propagation to it.

4.3 Mutability

As mentioned earlier, when defining a type, each slot can be specified as being either mutable or immutable and that the default is for slots being immutable. Emacs Lisp is a language that is usually not in the business of preventing users from shooting themselves in the foot (preferring to merely try and make it easier for the users not to shoot themselves in the foot), so the choice of immutability deserves some explanation.

When a variable is both mutated and captured, the closure conversion will apply a *store conversion* to turn the variable into an immutable variable pointing to a "box" in which the real value is kept. This extra indirection can be avoided in some cases, but in the general case it is indispensable in order to handle a variable captured by several closures that need to share its state.

For this reason, when accessing the content of a slot, we need to know if that slot has been store-converted or not. One could store this auxiliary information alongside the code, inside a closure, but in order to make slot access more efficient, and to avoid having

to store that auxiliary information alongside the code, we decided instead to make this choice ahead of time in `oclosure-define`: if a slot is defined as mutable we simply force store-conversion on it.

Another, probably better option would be to never perform store-conversion on OClosure slots. Instead, such mutable slots would "live" in the OClosure object and any other closure that wants to refer to it will just have to keep a reference to the whole OClosure. This requires more changes in the closure conversion algorithm, so we decided to put it in the wishlist for now.

Another important reason to declare beforehand when a slot is mutable is that the evaluation of a λ usually does not guarantee it returns a fresh new object. This is a problem for example with Guile's `set-procedure-property!` [Guile 2021] which may end up affecting more functions than intended. But if one of the slots is declared mutable, then `oclosure-lambda` will know that it needs to return a fresh new object, avoiding these unpredictable semantics.

4.4 Implementation

OClosures are currently implemented as a set of functions and macros that are loaded fairly early on during the bootstrap, but they are not implemented as a core data-structure. Most importantly, Emacs Lisp's `lambda` is not defined as a special case of `oclosure-lambda` as it arguably should. It's rather the reverse.

As far as we know, the only reliable way to implement something like `oclosure-lambda` involves defining it as a new *special form* of the language. Yet, introducing new special forms in Emacs Lisp is tricky because it can break existing packages which rely on code-walkers in their macros. So, instead we decided to implement it as a macro, and make it rely on cooperation from the closure conversion phase of the compiler.

At its simplest that macro looks like:

```
(defmacro oclosure--lambda
          (type bindings args &rest body)
  `(let ,(reverse bindings)
     (lambda ,args
       (:documentation ,type)
       (if t nil ,@(mapcar #'car bindings))
       ,@body)))
```

The name has two hyphens, because this is an internal macro, used by the real `oclosure-lambda` macro, among other things because it takes its args in a slightly different form.

The way this macro works is as follows: it adds the desired slots as "normal" variables in the context of a normal lambda and then arranges two things: first it makes sure that those variables will be captured into the closure, and then it controls the placement of those variables into the closure.

For both of those, it relies on knowledge about the way the code will be compiled, so the macro itself does not tell the whole story, and it requires cooperation from the compiler. You can see that it arranges for the variables to be captured by adding a piece of dummy code (wrapped in an `if` test to make sure it's never executed). To control the placement of the slots, it relies on the closure conversion which places the captured variables according to their position in the environment (one could say they are ordered by increasing de Bruijn index), which is why it uses reverse on the bindings so that the first slot gets added last to the environment.

The type information is handled specially, stashed as if it were the docstring of the function. A more obvious choice might have been to store that information in the first slot of the closure, except that we need to be able to distinguish reliably an OClosure from a normal closure that happens to have captured a variable holding a type information and placed it in its first slot.

The real macro is a bit more complex in order to handle the case of mutable slots, on which we want to force store conversion. This is obtained very simply by changing the dummy code that's never executed so that instead of only referring to the variable it performs an update on it. This relies on the fact that the current closure conversion naively performs store conversion on any variable that is both captured and mutated.

Clearly, the current state of implementation is not ideal, but it works well enough for now. It will likely be replaced by something cleaner when (or if) OClosures are made into a core data structure such that `lambda` is defined as a special case of `oclosure-lambda`, but there are various backward compatibility hurdles along the way, which will take some years to iron out.

5 EXPERIENCE

OClosures were developed in response to a growing set of use cases collected over the years. Here are the highlights, showing cases where the alternatives had significant shortcomings.

5.1 Advice

While there are various ways to solve the problem presented in Section 2, we did not want to pay the corresponding run time price of incurring an additional indirection or storing the extra information in a separate `eq`-indexed hash table. So the preexisting implementation of those `advice` functions relied on manually constructed closures. It worked well enough but made for rather obscure code.

The use of OClosures made the code much cleaner, removing all the low-level implementation-dependent tricks from it. It also made it possible to implement the pretty printing with a normal `defmethod` rather than the previous ad-hoc test which intruded into the more generic part of the pretty printer. Other than that, the actual runtime representation of those objects ends up being virtually identical.

5.2 `next-method-p`

CLOS defines `next-method-p` to return a non-nil if there is a next method (which `call-next-method` will invoke when called) and nil otherwise. These two functions can only be called from within methods. Internally, the code of methods can be implemented in various ways, but as far as I can tell, they are usually implemented as in Figure 7 which shows the relevant code used in Closette. In that code, *form* is the actual body of the method received by the `defmethod` macro. As you can see, the method is compiled to a function that takes the actual arguments *args* that were passed to the generic function, of course, and it takes an additional argument *next-emfun* which holds the next method to call. This argument is `nil` when there is no next method, so `next-method-p` is trivial and efficient, but in return for that `call-next-method` has to test *next-emfun* with an `if` before it can call it. This is the

```
`(lambda (args next-emfun)
   (flet ((call-next-method (&rest cnm-args)
            (if (null next-emfun)
                (error "No next method for the~@
                        generic function ~S."
                 (method-generic-function ',method))
              (funcall next-emfun (or cnm-args args))))
          (next-method-p ()
            (not (null next-emfun))))
     (apply #'(lambda ,(kludge-arglist lambda-list)
                ,form)
            args))))))
```

Figure 7: Implementation of a method in Closette

wrong trade-off since `next-method-p` is used much less often than `call-next-method`.

Now, arguably, this `if` test is fairly minor: `call-next-method` is not called very often and most of the performance issues have to do instead with the cost of creating the various closures and the layers of function calls. So more efficient implementations, such as PCL spend a fair deal of efforts optimizing this code but they still leave this `if` test untouched.

To remove this `if` we need to replace the `nil` representation of the error case with a function which will signal the error when called. This makes the `call-next-method` code simpler and more efficient, but it introduces a problem in `next-method-p`: how can we tell if *next-emfun* is one of those functions representing the "no next method" case?

In Emacs Lisp, we used to do just that with a really gross and brittle hack which dug into the innards of the closures to compare them against a sample. With OClosures we now simply defined a trivial type with no slot, (`oclosure-define cl-generic-nnm`), then use `oclosure-lambda` when building those functions, and finally replaced the 20 line monster of magic incantations with just:

```
(eq (oclosure-type cnm) 'cl--generic-nnm)
```

5.3 Keyboard macros

Emacs's keyboard macros are not macros in Lisp's sense but are simply a sequence of key presses recorded by the user so they can replay them later at will. Originally, they were represented as a simple vector of key presses and still several parts of Emacs support this form, but then the `kmacro` package extended that functionality and needed more info for that, making it unable to use the built in support to treat a mere vector as a kind of executable object. Instead it represented keyboard macros as a sort of object implemented as a list holding a vector of key presses, plus 2 other pieces of information, and in order to make it executable, it then wrapped it into a function.

The nasty part was when `kmacro` needed to look at such a function, in order to extract the 3-element list from it, either to print it in a human-friendly way, or even to let the user edit it. Contrary to the previous two examples, those functions constructed by `kmacro` did not need to run fast and could use more or less any calling convention they wanted, so where able to implement in a less hideous way, by arrange for the function to return its contents when called with a special argument, and simply using a special docstring to recognize those functions (which was needed simply to know that it's safe to call it with that special argument).

The new code uses an OClosure to replace both the list of 3 elements and the wrapper function, making most of the code significantly cleaner. Contrary to the previous two cases, this is a use case where something like *funcallable objects* would have worked almost as well since the extra indirection it would have imposed would be of no consequence.

5.4 Commands

Emacs Lisp functions are actually not quite as opaque as the λ-calculus wants them to be. We can not only get to know a function's arity but we can also query a bit more information about it: Emacs Lisp functions can carry and expose a *docstring* as well as an *interactive form*. The first is used for documentation purposes only (except for exceptional cases as in `kmacro`), while the second makes it possible to use function names as interactive commands: an interactive form is a chunk of code which constructs the list of arguments to pass to the function when the user invokes the command.

These are basically ad-hoc forms of OClosure slots. Emacs also defines a subtype of functions, called *commands* which corresponds to those functions which have an interactive form.

The current OClosure code makes these ad-hoc forms of function slots and function subtypes obsolete, by defining the type of `oclosure-command` containing an `interactive-form` slot, and making it possible to use OClosure slots to carry a function's docstring and interactive form. Nevertheless, the obsolete support is still in very heavy use because of the subtle incompatibilities that are introduced when using the new code.

5.5 Threesomes

Another circumstance where we have found a need to look inside a function is when trying to avoid accumulating function wrappers. These accumulations can typically occur for wrappers implementing coercions, as in type-directed unboxing [Leroy 1992] or in gradual typing [Siek and Taha 2006]. A solution to those accumulations consists in collapsing those wrappers by recognizing that some of them inevitably cancel others [Minamide and Garrigue 1998]. Siek and Wadler [2010] provide such a solution for the case of gradual typing. In the calculus they use to solve the problem, they introduce *threesomes* which are coercions written $\langle T \overset{R}{\Leftarrow} S \rangle s$, where a the term s of type S is coerced to type T via type R. The way they avoid accumulating coercions is by having a rule which reduces $\langle T \overset{R_1}{\Leftarrow} U \rangle \langle U \overset{R_2}{\Leftarrow} S \rangle s$ to $\langle T \overset{R_1 \& R_2}{\Leftarrow} S \rangle s$, so coercions can never accumulate.

In their calculus, those $\langle T \overset{R}{\Leftarrow} S \rangle s$ don't reduce to functions when s itself is a function, instead they are part of the possible runtime values, which means that function calls have to handle the case of a λ differently than the case of a coercion. The other option when implementing such a system is to make those coercions (when applied to functions) reduce to functions implemented using wrappers. This can simplify and speed up the all important functions calls. But it is only an option if there is still a way to recognize those

wrapper functions so we can combine them when we try to apply a wrapper on top of another. If all you have is a plain λ, there is no other option than to get your hands dirty and look under the abstraction barrier. With OClosures instead, you can have your cake and eat it: simple and efficient function calls, with efficient wrappers, while still able to quickly recognize those wrappers and extract whichever information is needed in order to collapse them.

6 RELATED WORK

The idea of treating functions as objects is quite old.

As mentioned, the AMOP [Kiczales et al. 1991] uses *funcallable objects* which are somewhat like OClosures with a single slot which do nothing more than pass their arguments to that slot's value, which should be another function. They suffer from the fact that they tend to introduce an indirection between the *funcallable object* and the actual underlying function, and the fact that the code of the function cannot directly access the funcallable object's slots. In return for that, the contained function can be changed by side-effect, whereas it would be difficult to allow changing the code of an OClosure.

MIT Scheme [MIT-Scheme 2020] provides similar functionality under the name *application hooks*. The more interesting of them are called *entities* which contain a function and another object. When called, an entity calls its contained function, passing it the arguments it received *plus itself*. This somewhat reduces the problem mentioned above that the function cannot directly access the funcallable object's slots.

GNU Kawa [Kawa 2020] and GNU Guile [Guile 2021] allow functions to carry extra properties, called *procedure properties* that can be added via side effect and queried. Again, the function itself does not have any direct access to those properties, limiting their applicability.

The Lisp Machine Lisp [Stallman et al. 1984] did not really support lexical scoping like we now have in Scheme and Common Lisp, but it had a `closure` operator that took a list of (dynamically scoped) variables and a function and returned a new function which called its argument function with the vars temporarily re-bound to the value they had when you created that "closure". The relevant part here is that you could access the list of closed-over variables and extract their values, just as we do in OClosures. Going even further, Lisp Machine Lisp had the `entity` operator which worked almost identically, except that it made it possible to assign a type to the returned function, which was typically used to allow specialized pretty printing output for those *entities*.

More recently, Scheme's SRFI 229 suggests the notion of tagged procedure, which is a procedure that comes with one extra immutable slot (called its tag) holding an arbitrary value. Beside the fact that it is limited to a single slot, it is also more limited than open closures in the sense that the tagged procedure's body cannot directly refer to the tag, so when that is needed, the tag value will probably end up duplicated in the object: one copy in the tag slot and another among the captured variables.

Of course, OClosures correspond to objects limited to a single method, used quite widely in OO-style languages that do not have a separate notion of function. They differ a bit in the sense that they conflate `oclosure-define` and `oclosure-lambda` and force every function with a different body to have a different type (since the method is associated with the class).

The function objects of Python are also similar: one can get the list of captured variables of a Python function as well as query (and modify) their values. But this is mostly a result of its introspection facilities, offering no way for the programmer to control which captured variables are exposed and which aren't.

Siskind and Pearlmutter [2007] propose to make closures more transparent by providing a `map-closure` function which is like `mapcar` but for closures, applying a given function to each of the values captured within the closure. The name of the captured variables is not made available, so this cannot be used to extract targeted information such as the value of a particular slot, and in this sense their functions remain quite opaque (in a sense analogous to security through obscurity, maybe).

7 CONCLUSION

We have presented the idea of making functions a bit less opaque in the form of open closures, then shown a design and implementation of this feature in Emacs Lisp under the name of OClosures, and given a sense of how they can be applied in a variety of circumstances where using either tuples or functions or a combination of both is not quite satisfactory.

ACKNOWLEDGMENTS

The author would like to thank the readers of `emacs-devel` for their naming suggestions and in particular Qiantan Hong who proposed the name of "open closures".

This work was supported by the Natural Sciences and Engineering Research Council of Canada grants № 298311/2012 and RGPIN-2018-06225. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the NSERC.

REFERENCES

Guile 2021. *GNU Guile Reference Manual* (3.0.7 ed.). https://www.gnu.org/software/guile/manual/

Kawa 2020. *The Kawa Scheme Language – Reference Documentation* (3.1.1 ed.). https://www.gnu.org/software/kawa/pt01.html

Gregor Kiczales, Jim Des Rivières, and Daniel G. Bobrow. 1991. *The Art of the Metaobject Protocol*. MIT Press.

Peter Lee and Mark Leone. 1996. Optimizing ML with Run-Time Code Generation. In *Programming Languages Design and Implementation*. ACM Press, Philadelphia, PA, 137–148.

Xavier Leroy. 1992. Unboxed Objects and Polymorphic Typing. In *Symposium on Principles of Programming Languages*. 177–188.

Yasuhiko Minamide and Jacques Garrigue. 1998. On the runtime complexity of type-directed unboxing. In *International Conference on Functional Programming*. ACM Press, 1–12.

MIT-Scheme 2020. *MIT/GNU Scheme Reference* (11.2 ed.). https://www.gnu.org/software/mit-scheme/documentation/stable/mit-scheme-ref/index.html

Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *Scheme Workshop*. 81–92.

Jeremy G. Siek and Philip Wadler. 2010. Threesomes, with and without blame. In *Symposium on Principles of Programming Languages*. 365–376. https://doi.org/10.1145/1707801.1706342

Jeffrey Mark Siskind and Barak A. Pearlmutter. 2007. First-class Nonstandard Interpretations by Opening Closures. In *Symposium on Principles of Programming Languages*. 71–76. https://doi.org/10.1145/1190216.1190230

Richard Stallman, Daniel Weinreb, and David Moon. 1984. *Lisp Machine Manual* (6th ed.). MIT. https://hanshuebner.github.io/lmman/frontpage.html

Christian Urban, Stefan Berghofer, and Michael Norrish. 2007. Barendregt's Variable Convention in Rule Inductions. In *International Conference on Automated Deduction*. 35–50. https://doi.org/10.1007/978-3-540-73595-3_4