Continuation Sensitive Compilation *

submitted to LASC special issue on continuations, also appears in "Les Écrits d'ICSLA", volume 7

Christian Queinnec†
École Polytechnique & INRIA-Rocquencourt

Abstract

This paper presents a compilation technique for Scheme-like languages where functions may look at their continuation before pushing frames onto it. This allows for some optimizations when the frame to be pushed and the frame on top of the continuation can be combined into a single and simplified frame. Among possible simplifications are: intermediate data structure elimination and removal of redundant calculations. Functions can therefore be compiled with respect to their near future and reorganize it when appropriate.

The compilation technique is based on an improved CPS-like transformation that makes continuation (i.e. stack) frames explicit. Shape of continuations is approximated to determine which frames would gain by being combined together then partial evaluation is used to determine the behavior of combined frames. Our main results cover local deforestation-like effect as well as iterative compilation of associatively wrapped recursions.

This paper presents a compilation technique for Scheme-like languages where functions may look at their continuation before pushing frames onto it. Basically, when a frame is about to be pushed onto a continuation and, if it is possible to combine this very frame with the frames with which the continuation begins, then the top of the continuation is reorganized. The simplest reorganization is to combine the frame to push with the frame on top of the continuation so these two frames can be replaced by a single one. The resulting frame might have a substantially simpler behavior since some simplifications can be performed like removing intermediate data structures produced by one frame for the other, removing needless computations or, precomputing ready redexes.

The technique crucially depends on a program transformation which is the direct and immediate counterpart of abstract continuations [FWFD88]. Abstract continuations were invented to denote elaborate control operators. Abstract Continuation Passing Style represents continuations by lists of frames. Each frame waits for a value as well as the list of frames that are below it; when invoked it usually transforms the value into a result it sends to the other frames. We start from this semantics and turn it into a program transformation nicknamed Value Transforming Style (VTS).

VTS represents continuations in a less opaque way than CPS. Because of this richer structure, it is possible to approximate the shape of the continuations that are used in a program. This continuation shape analysis provides some hints about the frames that are likely to be pushed on top of others. Rather than building a frame φ to push onto a frame φ', it might be interesting to combine them into a single frame φ'' so that φ'' is built instead of φ and φ'' replaces φ'. There is a gain if the new frame φ'' is shorter or has a simpler behavior than the old frames φ and φ'.

We therefore statically analyze the various combinations suggested by the continuation shape analysis. Combining frames is essentially obtained through a kind of partial evaluation which is easier than in real Scheme since VTS, heir of CPS, restricts the language of value transformers. Of course, only interesting combinations of frames are retained.

---

*Revision : 1.15 of November 22, 1992 at 17:21 — Draft submitted at LASC.

†Laboratoire d'Informatique de l'École Polytechnique (URA 1437), 91128 Palaiseau Cedex, France – Email: queinnec@polytechnique.fr This work has been partially funded by Greco de Programmation.

A continuation represents the future of a computation. This reorganization of the continuation can be viewed as a simplification of this future. Our technique can be used statically in known contexts but it can also be of some use at run-time where, before pushing a frame, the frame on the top of the continuation can be analyzed to check if there is a dynamically unforeseen but statically precomputed combination to perform.

The contribution of the paper is a compilation technique that reorganizes continuations using frames combination. This technique shortens continuations, performs *weeding* (a small-scaled deforestation-like effect) and transforms some associatively wrapped recursions into tail-recursions.

Our compilation technique does not depend on the implementation of continuations. It is a local analysis since first, it does not require gobal properties the whole program must respect (as in deforestation) and second, it only needs to know the behaviors of the frames that are to be combined. Moreover it can be used in the context of a "mostly functional" language where pure fragments are not rare. To end with an example, the `iota` function of table 1 is compiled with no continuation growth.

```
(define (factorial n)
  (reduce * n (iota 1 n)) )
(define (iota start stop)
  (if (< start stop)
      (cons start (iota (+ start 1) stop))
      '() ) )
(define (reduce binary end list)
  (if (pair? list)
      (binary (car list) (reduce binary end (cdr list)))
      end ) )
```

Table 1: Original program

The structure of the paper is as follows: we first introduce the VTS transformation in section 1. In order to be able to inquire frames about their nature as well as to be able to extract closed variables out of closures, we use a particular λ-lifting defined in section 2. We thus obtain a first order equivalent representation for any program even containing elaborate control features.

Continuation shape analysis is then performed and how to combine value transformers is handled in section 3. We are thus able to present some applications like weeding in section 4.

Further refinements like specializing a function with respect to the frame that appears on top of the continuation is treated in section 5, use of more semantic properties such as associativity in section 6 and finally derecursivation in section 7. Related works and conclusions follow.

1 Value Transforming Style

This section introduces the VTS program transformation. CPS makes continuations explicit but represents them by unary functions which can only be applied. VTS improves on CPS since it adopts a less opaque representation for continuations: continuations are modeled by lists of value transformers. A value transformer is a binary function that waits for a continuation and a value. When given a value, it performs some computation transforming this value, the result of which is given back to the continuation. A value transformer is the functional representation of a stack frame that – first, expects a value on which to be resumed – second, will transform this value in order to pass it to the frames which are below.

Similarly to CPS, VTS adds an extra argument to functions to represent their continuation: we name this argument q. Two operators: RESUME and EXTEND[1] encapsulate continuations. Sending a value v to a

[1] New operators appear in SMALL CAPS when introduced by program transformations.

$\mathcal{VTS}[\![\nu]\!]q \rightarrow$ (RESUME q ν)
$\mathcal{VTS}[\![$(quote ε)$]\!]q \rightarrow$ (RESUME q (quote ε))
$\mathcal{VTS}[\![$(if π_0 π_1 π_2)$]\!]q \rightarrow \mathcal{VTS}[\![\pi_0]\!]$(EXTEND q ($\lambda(q'$ v') (if v' $\mathcal{VTS}[\![\pi_1]\!]q'$ $\mathcal{VTS}[\![\pi_2]\!]q'$)))
$\mathcal{VTS}[\![$(set! ν π)$]\!]q \rightarrow \mathcal{VTS}[\![\pi]\!]$(EXTEND q ($\lambda(q'$ v') (RESUME q' (set! ν v'))))
$\mathcal{VTS}[\![(\lambda(v^*)$ $\pi)]\!]q \rightarrow$ (RESUME q ($\lambda(q'$ v^*) $\mathcal{VTS}[\![\pi]\!]q'$))
$\mathcal{VTS}[\![(\pi_1$ $\pi_2)]\!]q \rightarrow \mathcal{VTS}[\![\pi_1]\!]$(EXTEND q ($\lambda(q_1$ v_1) $\mathcal{VTS}[\![\pi_2]\!]$(EXTEND q_1 ($\lambda(q_2$ v_2) (v_1 q_2 v_2)))))
$\mathcal{VTS}[\![(\pi_1 \ldots \pi_n)]\!]q_0 \rightarrow$
$\mathcal{VTS}[\![\pi_1]\!]$(EXTEND q_0 ($\lambda(q_1$ v_1) $\ldots \mathcal{VTS}[\![\pi_n]\!]$(EXTEND q_{n-1} ($\lambda(q_n$ v_n) (v_1 q_n v_2 $\ldots v_n$)))))

Table 2: VTS rules

continuation q is expressed with (RESUME q v). One extends a continuation with a value transformer φ i.e. pushes a frame on a stack, with (EXTEND q φ). The RESUME and EXTEND operators obey the following fundamental relation i.e. when a continuation receives a value, it simply applies its first value transformer on the value itself and on the rest of the value transformers:

(RESUME (EXTEND q φ) v) $\equiv$ (φ q v)

Unlike CPS, value transformers in VTS appear as VTS-transformed unary functions i.e. they take two arguments, a continuation and a value. The continuation is thus represented by a list of regular closures.

A simple implementation of RESUME and EXTEND in regular Scheme follows[2]; observe that RESUME and EXTEND also respect the interface of VTS-transformed unary functions:

```
(define (RESUME q v) ((car q) (cdr q) v))
(define (EXTEND q vt) (cons vt q))
```

The VTS transformation is defined on table 2[3]. VTS creates a lot of administrative redexes. A reformulation of VTS along the lines of [SF92] is probably possible but would slightly complicate the above rules. Alternatively, a post-processing phase can be added to remove these administrative redexes as well as others that might exist in the original program. Applications of primitives without control effect such as `cons`, `pair?` etc[4]. (trivial forms of [Rey72]) can also be simplified.

The running example of our paper will be the (odd) `factorial` appearing in table 1. Assuming a left-to-right evaluation order, VTS transforms the original program as shown in table 3. Generated value transformers are indexed to be distinguished later. Observe how the code excerpt (`cons start` ...) within `iota` has been turned into a value transformer within `VTS-iota`[5]. If `factorial` is called then during the computation of `VTS-iota`, the continuation starts by a list of λ_3 closures followed by a λ_1 closure.

2 Object Oriented λ-lifting

VTS transforms continuations into lists of closures. To combine frames requires two distinct abilities: to extract closed variables out of closures and to enquire about frames to know their nature. The simplest way to provide these capabilities is to adopt an object-oriented explicit representation for closures. We thus adapt λ-lifting [PJ87, App92] for our purpose.

λ-lifting essentially transforms abstractions, applications and free variables references. We will not sketch the complete algorithm and only the salient features appear in table 4. The target language adopts very limited features of object-oriented technology [BDG+88]. We will use the three following defining forms: `define-generic`, `define-method` and `define-class`.

`define-generic` defines a generic function to which new behaviors can be added using `define-method`. The parameters of `define-method` specify the generic function to specialize, the discriminating class and the

[2] Observe that CPS is a special case of VTS with the following definitions: `(define (extend q vt) (lambda (v) (vt q v)))` and `(define (resume q v) (q v))`.

[3] Binary and n-ary applications are both shown to give a flavor of the ... ellipsis.

[4] Observe that `car` has a control effect since it raises an error if applied on a non-dotted pair.

[5] Actually this gives a means to display the frames of a stack in terms of the original user's program.

```
(define (VTS-factorial q0 n)
  (VTS-iota (EXTEND q0 (λ1 (q1 v1) (VTS-reduce q1 VTS-* n v1)))
            1 n ) )
(define (VTS-iota q2 start stop)
  (if (< start stop)
      (VTS-iota (EXTEND q2 (λ3 (q3 v3) (RESUME q3 (cons start v3))))
                (+ start 1) stop )
      (RESUME q2 '()) ) )
(define (VTS-reduce q4 binary end list)
  (if (pair? list)
      (let ((temp1 (car list)))
        (VTS-reduce (EXTEND q4 (λ5 (q5 v5) (binary q5 temp1 v5)))
                    binary end (cdr list) ) )
      (RESUME q4 end) ) )
```

Table 3: After VTS

body of the method itself. A default method exists on the two generic functions we will use (INVOKE and EXTEND). Both of them provide a default method that can be superseded by an optimized method resulting from our analysis.

New classes are defined by means of `define-class` which requires the name of the new class, the name of its super-class (we only use inheritance in this paper to gather closure objects as instances of the virtual class `closure`) and the names of its slots. When a class c is defined, some functions automatically accompany it: `c?` to recognize instances of c, `make-`c to create immutable instances of c and c`-`f to select slot f from instances of c.

This language is equivalent to a kind of high level assembly language where data structures are defined with `define-class` and where generic functions abstract case analysis.

assume: `(define-generic (INVOKE function . arguments))`
`(define (RESUME q v) (INVOKE (car q) (cdr q) v))`

transform:

$(\pi_0\ \pi_1\ \ldots\ \pi_n) \quad \rightsquigarrow \quad$ `(INVOKE` $\pi_0\ \pi_1\ \ldots\ \pi_n$`)`

transform:

`(lambda (`ν^*`)` π`)` *with* ν'^* *free in* $\pi \quad \rightsquigarrow \quad$ `(make-closure`$_i$ ν'^*`)`

with: `(define-class closure`$_i$ `(closure)` ν'^*`)`
`(define-method (INVOKE (self closure`$_i$`)` ν^*`)`
$\pi[$`(closure`$_i$`-`ν' `self)`$/\nu'] _{\forall \nu' \in \nu'^*}$ `)`

Table 4: OO-λ-lifting rules

The functional behavior of each closure is recorded as a method of the INVOKE generic function. Note that, similar to [App92], the closure object *is* the closed environment. The free variables that appeared in the body of an abstraction are now found in the closure object itself i.e. the `self` variable of the INVOKE method. The oo-λ-lifted version of the current example appears in table 5.

The above transformation is only a minor variation of the classical λ-lifting technique with a different target language. It corresponds well to the transformations that compilers do more or less implicitly except that *(i)* we do not dictate a uniform way to represent closures but rather leave it to the implementor of `define-class`, *(ii)* we do not impose how closures are applied i.e. how the generic function INVOKE is

```
(define-class closure0 (closure))        ;no closed environment !
(define-method (INVOKE  (self closure0) q0 n)
  (INVOKE  VTS-iota (EXTEND q0 (make-closure1 n)) 1 n) )
(define-class closure1 (closure) n)
(define-method (INVOKE  (self closure1) q1 v1)
  (INVOKE  VTS-reduce q1 VTS-* (closure1-n self) v1) )
(define VTS-factorial (make-closure0))

(define-class closure2 (closure))        ;no closed environment !
(define-method (INVOKE  (self closure2) q2 start stop)
  (if (< start stop)
      (INVOKE  VTS-iota (EXTEND q2 (make-closure3 start)) (+ start 1) stop)
      (RESUME q2 '()) ) )
(define-class closure3 (closure) start)
(define-method (INVOKE  (self closure3) q3 v3)
  (RESUME q3 (cons (closure3-start self) v3)) )
(define VTS-iota (make-closure2))

(define-class closure4 (closure))        ;no closed environment !
(define-method (INVOKE  (self closure4) q4 binary end list)
  (if (pair? list)
      (let ((temp1 (car list)))
        (INVOKE  VTS-reduce (EXTEND q4 (make-closure5 binary temp1))
                            binary end (cdr list) ) )
      (RESUME q4 end) ) )
(define-class closure5 (closure) binary temp1)
(define-method (INVOKE  (self closure5) q5 v5)
  (INVOKE  (closure5-binary self) q5 (closure5-temp1 self) v5) )
(define VTS-reduce (make-closure4))
```

Table 5: After OO-λ-lifting

implemented.

Within this framework, the current state of a computation can be characterized by a single INVOKE or RESUME application accompanied by its arguments. The first argument, the continuation, is a list of frames which may be allocated in a stack or a heap depending on the features of the language (whether it offers `call/cc`, for example) or on the chosen implementation. There is actually a single generic function: INVOKE . The next step is to remark that EXTEND can also be a generic function.

3 Combining frames

The EXTEND operator extends a continuation with a value transformer. The main idea of the paper is to take advantage of this extension to reorganize the continuation. From an implementation point of view, EXTEND is nothing but an operator that pushes frames onto a continuation whether implicitly (in a stack-based implementation) or explicitly with some links adjustment (in a heap-based implementation). Alternatively EXTEND can be considered as a generic function which may have specialized methods to push some frames depending on the nature of the frames on top of the continuation. Suppose EXTEND to be defined as:

```
(define-generic (EXTEND q vt))
(define-method (EXTEND q (vt closure))  ;default behavior:
```

```
(cons vt q) )                                   ;push the frame.
```

EXTEND does not depend on any method other than the default one to work but, as a generic function, EXTEND offers prospect of improvement if specialized methods can be devised.

We first need to approximate the shape of continuations. We therefore analyze all INVOKE methods to examine how continuations are handled. The analysis is particularly simple since INVOKE methods are written in λ-lifted VTS-style language where all computations are trivial except RESUME and INVOKE forms. Let us take the single example of `VTS-iota` independently of any other functions. A call to `closure2` (call `VTS-iota`) may extend its q_2 continuation only with an instance of `closure3` (build the result of `VTS-iota`) before calling again a `closure2`[6]. An invocation of `closure3` simply resumes its continuation. Therefore when a `closure2` is called, the continuation has a prefix of zero or more instances of `closure3`. We can thus deduce that instances of `closure3` are likely to be pushed on top of instances of `closure3`. It is therefore interesting to analyze the combination of two `closure3` frames.

Suppose that we are about to push φ on top of φ', both being instance of `closure3`, we must look for a possible reorganization q' of the continuation such that

(RESUME (EXTEND (EXTEND q φ') φ) v) $\equiv$ (RESUME q' v)

One can easily show that the left term, after unfolding RESUME and the INVOKE method of `closure3` twice, is equivalent to:

(RESUME q (cons (closure3-start φ') (cons (closure3-start φ) v)))

To combine the two instances of `closure3` into a single frame thus requires the invention of a new sort of frame, say `closure11` and to specialize EXTEND when pushing instances of `closure3`. The INVOKE method of `closure11` strictly corresponds to the above form except that we abstract on the slots of the frames we want to combine. We thus obtain:

```
(define-class closure11 (closure) start1 start2)  ;Combination of closure3/closure3
(define-method (INVOKE  (self closure11) q v)     ;Combined behavior
  (RESUME q (cons (closure11-start1 self)
                  (cons (closure11-start2 self) v) )) )
(define-method (EXTEND q (vt closure3))           ;specialized push
  (if (closure3? (car q)) ;top frame is (car q)
      (EXTEND (cdr q)     ;pop it then push:
              (make-closure11 (closure3-start (car q))
                              (closure3-start vt) ) )
      (cons vt q) ) )     ;the default behavior
```

The above combination of frames does not seem to be interesting since a new sort of frame was necessary and its number of slots is not decreasing compared to the number of slots present in the two frames that were combined. To adopt this specialization would also force us to analyze the combination of a `closure3` on top of a `closure11` as well as the combination of two `closure11` frames which does not seem to be attractive either.

Not unlike partial evaluation, the problem is to determine where to stop inventing and combining new frames. Practically, we only invent new frames that (directly or after being themselves combined) decrease the number of slots i.e. closed values. A more thorough comparison with partial evaluation is delayed until section 8.

4 Weeding

Taking into account interprocedural results of the above continuation shape analysis reveals new possible combinations as well as avoiding the analysis of all possible pairs of frames[7]. From `VTS-factorial`, we know that a `closure3` (build `iota` result) can be pushed on a `closure1` instance (apply `reduce`) so it might be interesting to analyze this interaction. Let us name φ the instance of `closure3` which is about to be pushed on top of φ', the instance of `closure1`. Resuming them is expressed as:

[6] We assume `VTS-iota` to be immutable.

[7] A strongly typed language would restrict the analysis to type compatible couples of frames.

```
(RESUME (EXTEND (EXTEND q φ') φ) v)
```

Simple unfolding leads to

```
(INVOKE VTS-reduce q VTS-* (closure1-n φ') (cons (closure3-start φ) v))
```

If `VTS-reduce` is an immutable global binding and since the structure (or the value) of some of the arguments to pass to `VTS-reduce` is known, we can try to propagate this knowledge and unfold the previous application into:

```
(let ((self VTS-reduce)
      (q4 q)
      (binary VTS-*)
      (end (closure1-n φ'))
      (list (cons (closure3-start φ) v)) )
  (if (pair? list)
      (let ((temp1 (car list)))
        (INVOKE VTS-reduce (EXTEND q4 (make-closure5 binary temp1))
                           binary end (cdr list) )
      (RESUME q4 end) ) ) )
```

Accordingly to the program transformations offered by Similix [Bon91], the previous expression can be simplified since *(i)* `pair?` yields a known result, *(ii)* `car` and `cdr` forms can be simplified, *(iii)* non duplicated variables may be substituted by their value if side-effect-free, *(iv)* unused `let` variables can be scavenged [FH89] provided their value is a terminating expression (if we want to respect termination properties). These improvements yield:

```
(INVOKE VTS-reduce
        (EXTEND q (make-closure5 VTS-* (closure3-start φ)))
        VTS-* (closure1-n φ') v )
```

We can decide to stop our simplification here. We do not need to invent a new kind of frame since we can recognize an existing one: `closure1` ! We can thus specialize EXTEND such that pushing a `closure3` on a `closure1` is like pushing the same `closure1` on a `closure5`.

```
(define-method (EXTEND q (vt closure3))
  (if (closure1? (car q))
      (EXTEND (EXTEND (cdr q) (make-closure5 VTS-* (closure3-start vt)))
              (make-closure1 (closure1-n (car q))) )[8]
      (cons vt q) ) )
```

This specialized EXTEND method removes the need to construct a pair which is immediately destructured. This achieves a deforestation-like effect but on a small scale. This weeding effect is also able to remove useless computations. An example would be to analyze the expression `(begin (factorial n) ...)` where it is clear that the value of `(factorial n)` is useless. The computation of `(factorial n)` is done above a frame, say `closure9`, which ignores the value it will be given back. To push a `closure5` on a `closure9` is therefore equivalent to directly discarding the `closure5` frame[9] since the combination is equivalent to the sole `closure9` frame.

Two first results were shown in this section: *(i)* a non trivial reorganization of the continuation and, *(ii)* a successful elimination of intermediate data. But still more can be done !

5 Specializing frames

The `make-closure5` form in the previous EXTEND method can still be refined since one of its closed variables is known to be `VTS-*`. We can thus invent a new frame: `closure5clone` based on `closure5` where we also

[8] If continuations are heap-based, one can remark that `(make-closure1 (closure1-n (car q)))` is nothing else but `(car q)` since frames are immutable objects. One can alternatively imagine in a stack-based implementation (and in absence of continuation capture) to bit-blit the `closure1` frame.

[9] Note that the `(factorial n)` computation is not removed since this would not preserve termination equivalence. Only the final multiplications are eliminated.

simplify the call to the (immutable) global variable `VTS-*` into a direct (inlined) call to the multiplication operator `*`:

```
(define-class closure5clone (closure) start)
(define-method (INVOKE (self closure5clone) q5 v5)
  (RESUME q5 (* (closure5clone-start self) v5)) )
```

We can now improve the previous EXTEND method to use `closure5clone`:

```
(define-method (EXTEND q (vt closure3))
  (if (closure1? (car q))
      (EXTEND (EXTEND (cdr q) (make-closure5clone (closure3-start vt)))
              (make-closure1 (closure1-n (car q))) )
      (cons vt q) ) )
```

We thus shorten the size of the continuation at the expense of the creation of a new kind of frame. This will nevertheless prove to be useful in the next section.

6 Using associativity

Whenever a `closure2` is called, the continuation now starts with a `closure1` followed by a series of `closure5clone`. We thus have to examine the combination of two `closure5clone` frames. Suppose that we want to push φ onto φ':

```
(RESUME (EXTEND (EXTEND q φ') φ) v)
```

Simple unfolding leads to

```
(RESUME q (* (closure5clone-start φ') (* (closure5clone-start φ) v)))
```

Were we to invent a new frame for this combination, there would be no gain in grouping multiplications two by two unless, we use the associativity of the multiplication. Note that all previous reorganizations never use such properties, they were pure applications of fold/unfold/ specialize/generalize strategies not beyond regular compilers ability. Note also that more clever transformation schemes are not forbidden; [SJ87] showed, for example, how some recursions à la `f91`, can be iteratively interpreted with a dynamically introduced local counter.

If we rearrange the above multiplication, we can group the two multiplicands and obtain:

```
(RESUME q (* (* (closure5clone-start φ') (closure5clone-start φ)) v))
```

We already know a frame which has this behavior: `closure5clone`[10]! Therefore to push a `closure5clone` frame onto a `closure5clone` frame is similar to replacing the two by a single `closure5clone` frame enclosing their combined product. Therefore we can add a new method to the EXTEND generic function.

```
(define-method (EXTEND q (vt closure5clone))
  (if (closure5clone? (car q))
      (EXTEND (cdr q) (make-closure5clone (* (closure5clone-start (car q))
                                             (closure5clone-start vt) )))
      (cons vt q) ) )
```

Depending on the properties continuations have in the language, their possible capture, their lifetime, their possible (multiple) use etc. an implementation may choose to implement the previous EXTEND method with an update-in-place effect. Since a `closure5clone` is replaced with another instance of `closure5clone`, one can modify *in situ* the existing instance rather than replace it with a new one. Observe that the validity of this side-effect depends on non-local properties allowing us to decide if the to-be-patched frame is shared or not.

So far we simply add two methods to the generic function EXTEND (approximatively 10 lines) which achieve the computation of `factorial` with a constant continuation depth and entirely remove the need for intermediate data. These improvements are actually dynamic since EXTEND is generic and discriminates on its second argument in every call. Since all EXTEND forms have a second argument with a known nature,

[10] It is important to share frames to reduce their number.

one can easily specialize these calls and avoid the cost of invoking a generic function. Moreover certain EXTEND forms can be replaced by the default EXTEND behavior (push i.e. `cons`), where the gain is unsure. For instance, the following does not prevent `iota` to be evaluated in constant depth (nor a normally written factorial function) but precludes the further squashing of pending multiplications, if any, on top of `(cdr q)`:

```
(define-method (EXTEND q (vt closure5clone))  ;New
  (if (closure5clone? (car q))
      (cons (make-closure5clone (* (closure5clone-start (car q))
                                   (closure5clone-start vt) ))
            (cdr q) )                          ;no more generic now !
      (cons vt q) ) )
```

7 Integrating the top frame

Due to VTS, the current state is characterized by the closure which is currently processed and its arguments which include the continuation. The original `VTS-factorial` invokes `closure2` (recursive call to `VTS-iota`) with a continuation containing many `closure3` frames (constructing the pairs returned by `VTS-iota`) over a single `closure1` frame (call to `VTS-reduce`). The previous section transforms this continuation into a single `closure1` frame over a single `closure5clone` (pending multiplication) frame. We can thus easily specialize `VTS-iota`, into `VTS-iota-clone`, to assume that the top frame is a `closure1`. But at that time, it is no use building this top frame, one can simply add its slots as supplementary arguments to `VTS-iota-clone`.

In other terms we specialize `closure2` into `closure2clone`, in a manner reminiscent of [Wan80], such that:

```
(INVOKE  VTS-iota (EXTEND  q  (make-closure1  n))  i  j)
          ⇝  (INVOKE  VTS-iota-clone  q  i  j  n)
```

We accordingly modify the behavior of `closure2` and thus define the `closure2clone` behavior to be:

```
(define-method (INVOKE  (self closure2) q2 start stop)  ;New!
  (if (< start stop)
      (if (closure1? (car q2))
          (INVOKE  VTS-iota-clone (EXTEND  (cdr q2) (make-closure5clone start))
                   (+ start 1) stop (closure1-n (car q2)) )
          (INVOKE  VTS-iota (EXTEND  q2 (make-closure3 start))
                                     (+ start 1) stop ) )
      (RESUME  q2 '()) ) )
(define-class closure2clone (closure))
(define-method (INVOKE  (self closure2clone) q2 start stop n)
  (if (< start stop)
      (INVOKE  self                              ; VTS-iota-clone
               (EXTEND  q2 (make-closure5clone start))
               (+ start 1) stop n )
      (INVOKE  VTS-reduce q VTS-* n '()) ) )  ;may be further simplified...
(define VTS-iota-clone (make-closure2clone))
```

The computation now becomes a call to `closure2clone` with a continuation that contains a single `closure5clone` frame. We can thus reiterate the same process and specialize `VTS-iota-clone`, let it be `VTS-iota-other-clone`, with respect to a `closure5clone` frame and obtain the final code of table 6.

The final code implements the three original functions (`factorial`, `iota` and `reduce`), and specializes `iota` when called from `factorial` to be evaluated with no neat growth of the continuation. Code related to `closure4clone` and `closure5clone` could have been removed since it is not used, nevertheless it represents a by-product of our technique. `closure5clone` had been explained before, `closure4clone` corresponds to the combination of a `closure3` upon a `closure4` frame, an optimization that can be used in other contexts.

Observe that the technique that integrates the top frame applied to the regular recursive factorial turns it automatically into the equivalent of the iterative factorial [FWH92, Chap. 10].

We can sum up the main points of our compilation technique as follows: *(i)* perform VTS, *(ii)* perform Object-Oriented λ-lifting, *(iii)* analyze continuations shape, *(iv)* combine frames, *(v)* specialize frames on values, *(vi)* integrate top frame in INVOKE methods, *(vii)* occasionally use semantic properties such as associativity or other program transformations.

8 Related works

Many works are related to ours. We are first indebted to the work of Greussay who presented in his PhD [Gre77], run-time techniques allowing proper tail-recursion in the framework of an interpreter for a dynamically-scoped Lisp. Interpreter functions, such as `eval`, `evlis` etc. were allowed to analyze (by pattern-matching) the top of the stack in order to detect some configurations where optimizations were possible. This work has been continued by Saint-James who improved it to a very sophisticated level [SJ84, SJ87, SJ90]. Our first motivation was to bring similar improvements to compiled Scheme.

To use simultaneously VTS and the ability for functions to inspect their continuation was therefore a simple idea, at least tackled by Wand in [Wan80]. A quite similar idea can be found in [CD91] who advocates the idea that partial evaluation of CPS-transformed programs improves the quality of residual programs since CPS brings nearer the producers and consumers of intermediate data structures. Hence the idea of combining frames together.

The rest of the paper is an instance of a kind of partial evaluation in a language aimed at the implementation of frames, specializing frame behaviors on the structure of a prefix of the continuation. Due to the restricted language produced by VTS, many optimizations are simpler than in direct style. For example, weeding is possible if a compound data is produced and immediately destructured and not given back to a deeper continuation. Our approach is local and does not require global properties to hold. For example and contrasting with deforestation [Wad88], combining two frames is a local transformation that is not affected by the presence of side-effects elsewhere. In a "mostly functional" language such as Scheme, many fragments are purely functional and can benefit from our approach.

Deforestation technique has been recently extended by Chin [Chi92]. His technique is quite different since it is based on an annotation scheme on the source code where expressions are annotated as safe/unsafe producers/consumers. Though our techique is very different our results are quite similar and all the examples of his paper (even the high-orer one) can be handled by our technique i.e., analyzed so they do not consume intermediate data structures. We even have a somewhat better result on the following example: `(length (append x y))`. Chin introduced some extra information (a law) to convert this program into: `(+ (length x) (length y))` whereas our technique converts it into a number of `plus1` frames something that could be expressed at the source level as:

```
(define (main l1 l2) (append-clone l1 l2) ) (define (plus1 n) (+ 1 n))
(define (append-clone l1 l2) (if (pair? l1) (plus1 (append-clone (cdr
l1) l2)) (length l2) ) )
```

It seems natural to remove this number of `plus1` frames into a single addition frame. Our analysis is unable to do so presently and this is a point of future work. Were we able to do that and if we invent a sort of inverse VTS (see [DL92, SF92]), we transform the original program into:

```
(define (main l1 l2)
  (append-clone l1 l2 0) )
(define (append-clone l1 l2 n)
  (if (pair? l1)
      (append-clone (cdr l1) l2 (+ 1 n))
      (length-clone l2 n) ) )
(define (length-clone l n)
  (if (pair? l)
      (length-clone (cdr l) (+ 1 n))
      n ) )
```

Our OO language to represent frames is a precise implementation language since it expresses the details of allocation or field selection. It is for example possible to express the optimization where a frame accesses a value in some frame below. This is similar in spirit with some of the goals of Wand and Oliva [WO92].

From an implementation point of view, the OO representation of frames can be followed encoding frames as records tagged with their class. However our approach multiplies the number of frames that can appear in continuations since most of the frames exist in multiple variants specialized for particular contexts. New frames can be invented when compiling a function, compiling a group of functions, or compiling a whole application. Since functions are usually grouped in files or modules, common frames combinations can be found out.

9 Conclusions

The paper presented a technique nicknamed "Continuation sensitive compilation" based on the idea that some optimizations might be performed if functions are allowed to look at the continuation. The various existing frames that can appear in a continuation are analyzed to see if there is some gain to combine them. This allows the removal of useless computations and intermediate data structures and enable some kinds of recursion to be made iterative. Combining frames is an application of partial evaluation within the framework of an implementation language specialized in frames handling.

A real compiler based on these ideas is currently under progress. The language being compiled is a concurrent and distributed extension of Scheme [Que92a] where it is important to have explicit (heap-based) continuations that can be migrated through a network.

Acknowledgments: I wish to thank David de Roure who followed the maturation of this paper, Matthias Felleisen and Olivier Danvy for their comments and encouragement.

Bibliography

[App92] Andrew Appel. *Compiling with continuations*. Cambridge Press, 1992.

[BDG+88] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. Common lisp object system specification. *SIGPLAN Notices*, September 1988.

[Bon91] Anders Bondorf. Similix manual, system version 4.0. Technical report, DIKU, University of Copenhagen, Denmark, September 1991.

[CD91] Charles Consel and Olivier Danvy. For a better support of static data flow. In John Hughes, editor, *FPCA '91 – Functional Programming and Computer Architecture*, Lecture Notes in Computer Science523, pages 496–519, Cambridge (Massachusetts, USA), August 1991. Springer-Verlag.

[Chi92] Wei-Ngan Chin. Safe fusion of functional expressions. In *LFP '92 – ACM Symposium on Lisp and Functional Programming*, pages 11–20, San Francisco (California USA), June 1992.

[DL92] Olivier Danvy and Julia Lawall. Back to direct style II: First-class continuations. In *LFP '92 – ACM Symposium on Lisp and Functional Programming*, pages 299–310, San Francisco (California USA), June 1992. ACM Press. Lisp Pointers V(1).

[FH89] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. Computer Science Technical Report No. 100, Rice University, June 1989.

[FWFD88] Matthias Felleisen, Mitchell Wand, Daniel P. Friedman, and Bruce Duba. Abstract continuations: a mathematical semantics for handling functional jumps. In *Proceedings of the 1988 ACM Symposium on LISP and Functional Programming*, Salt Lake City, Utah., July 1988.

[FWH92] Daniel P Friedman, Mitchell Wand, and Christopher Haynes. *Essentials of Programming Languages*. MIT Press, Cambridge MA and McGraw-Hill, 1992.

[Gre77] Patrick Greussay. *Contribution à la définition interprétative et à l'implémentation des Lambda-langages*. Thèse d'état, Université Paris VI, November 1977. Rapport LITP 78-2.

[PJ87] Simon L. Peyton-Jones. *The Implementation of Functional Programming Languages*. International Series in Computer Science. Prentice-Hall, 1987.

[QS91] Christian Queinnec and Bernard Serpette. A Dynamic Extent Control Operator for Partial Continuations. In *POPL '91 – Eighteenth Annual ACM symposium on Principles of Programming Languages*, pages 174–184, Orlando (Florida USA), January 1991.

[Que92a] Christian Queinnec. A concurrent and distributed extension to scheme. In D. Etiemble and J-C. Syre, editors, *PARLE '92 – Parallel Architectures and Languages in Europe*, pages 431–446, Paris (France), June 1992. Lecture Notes in Computer Science 605, Springer-Verlag.

[Que92b] Christian Queinnec. Value transforming style. Research Report LIX RR 92/07, Laboratoire d'Informatique de l'École Polytechnique, 91128 Palaiseau Cedex, France, May 1992.

[Rey72] John Reynolds. Definitional interpreters for higher order programming languages. In *ACM Conference Proceedings*, pages 717–740. ACM, 1972.

[SF92] Amr Sabry and Matthias Felleisen. Reasoning about continuation-passing style programs. In *LFP '92 – ACM Symposium on Lisp and Functional Programming*, pages 288–298, San Francisco (California USA), June 1992. ACM Press. Lisp Pointers V(1).

[SJ84] Emmanuel Saint-James. Recursion is more efficient than iteration. In *LFP '84 – ACM Symposium on Lisp and Functional Programming*, pages 228–234, 1984.

[SJ87] Emmanuel Saint-James. *De la Méta-Récursivité comme Outil d'Implémentation.* Thèse d'état, Université Paris VI, December 1987.

[SJ90] Emmanuel Saint-James. Transformations de programmes à l'exécution : puissance et efficience. In Pierre Cointe, Philippe Gautron, and Christian Queinnec, editors, *Actes des JFLA 90 – Journées Francophones des Langages Applicatifs*, pages 110–117, La Rochelle (France), January 1990. Revue Bigre+Globule 69.

[Wad88] Philip Wadler. Deforestation: Transforming programs to eliminate trees. In H Ganzinger (ed), editor, *ESOP '88 – European Symposium on Programming*, volume 300 of *Lecture Notes in Computer Science*, pages 344–358, 1988.

[Wan80] Mitchell Wand. Continuation-based program transformation strategies. *Journal of the ACM*, 27(1):164–180, 1980.

[WO92] Mitchell Wand and Dino P Oliva. Proving the correctness of storage representations. In *LFP '92 – ACM Symposium on Lisp and Functional Programming*, pages 151–160, San Francisco (California USA), June 1992. ACM Press. Lisp Pointers V(1).

```
(define-class closure0 (closure))       ;no closed environment !
(define-method (INVOKE (self closure0) q0 n)
  (INVOKE VTS-iota-other-clone q0 1 n n 1) )
(define-class closure1 (closure) n)
(define-method (INVOKE (self closure1) q1 v1)
  (INVOKE VTS-reduce q1 VTS-* (closure1-n self) v1) )
(define VTS-factorial (make-closure0))

(define-class closure2 (closure))       ;no closed environment !
(define-method (INVOKE (self closure2) q2 start stop)
  (if (< start stop)
      (INVOKE VTS-iota                  ;self !
              (EXTEND q2 (make-closure3 start)) (+ start 1) stop )
      (RESUME q2 '()) ) )
(define-class closure3 (closure) start)
(define-method (INVOKE (self closure3) q3 v3)
  (RESUME q3 (cons (closure3-start self) v3)) )
(define VTS-iota (make-closure2))

(define-class closure4 (closure))       ;no closed environment !
(define-method (INVOKE (self closure4) q4 binary end list)
  (if (pair? list)
      (let ((temp1 (car list)))
        (INVOKE VTS-reduce              ;self !
                (EXTEND q4 (make-closure5 binary temp1))
                binary end (cdr list) ) )
      (RESUME q4 end) ) )
(define-class closure5 (closure) binary temp1)
(define-method (INVOKE (self closure5) q5 v5)
  (INVOKE (closure5-binary self) q5 (closure5-temp1 self) v5) )
(define VTS-reduce (make-closure4))

(define-class closure2otherclone (closure))
(define-method (INVOKE (self closure2otherclone) q2 start stop n product)
  (if (< start stop)
      (INVOKE self                      ; VTS-iota-other-clone
              q2 (+ start 1) stop n (* product start) )
      (RESUME q2 (* product n)) ))
(define VTS-iota-other-clone (make-closure2otherclone))

(define-class closure4clone (closure) binary end)
(define-method (INVOKE (self closure4clone) q4 v4)
  (INVOKE VTS-reduce q4 (closure4clone-binary self)
                        (closure4clone-end self) v4 ) )
(define-method (EXTEND q (vt closure3))
  (if (closure4? (car q))
      (EXTEND (EXTEND (cdr q) (make-closure5 (closure4clone-binary (car q))
                                             (closure3-start vt) ))
              (make-closure4clone (closure4clone-binary (car q))
                                  (closure4clone-end (car q)) ) )
      (cons vt q) ) )

(define-class closure5clone (closure) start)
(define-method (INVOKE (self closure5clone) q5 v5)
  (RESUME q5 (* (closure5clone-start self) v5)) )
(define-method (EXTEND q (vt closure5clone))
  (if (closure5clone? (car q))
      (EXTEND (cdr q) (make-closure5clone (* (closure5clone-start vt)
                                             (closure5clone-start (car q)) )))
      (cons vt q) ) )
```

Table 6: Final code