Continuation Conscious Compilation

Christian Queinnec*
École Polytechnique & INRIA-Rocquencourt

Abstract

This paper proposes some (unimplemented) ideas for the compilation of Scheme-like languages where functions may be specialized with respect to some of the continuations with which they are invoked. This allows for some optimizations, for instance, when a frame to be pushed and the frame on top of the continuation can be combined into a single and simplified frame. Among possible improvements are: intermediate data structure elimination and removal of useless calculations. Functions can therefore be compiled with respect to their near future and reorganize it when appropriate.

The compilation technique is based on a program transformation named Abstract Continuation Passing Style 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 converting, for example, a recursive unary factorial into an iterative binary one.

This paper presents some ideas to improve the compilation of Scheme-like languages. The main idea is to explore what can be gained if functions were allowed to 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 named Abstract Continuation Passing Style (ACPS) [FWFD88]. Abstract continuations were invented to denote elaborate control operators [FWFD88, Que92b, QD93]. Despite the rich underlying theory, we only use the fact that ACPS represents continuations by lists of frames. Each frame waits for a value as well as the list of frames that are below it; when a frame is invoked it usually transforms the value into a result it sends to the other frames. ACPS represents continuations in a less opaque way than CPS. Because of this richer structure, it is easier to approximate the shape of the continuations that might appear 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 *(i)* φ'' is built instead of φ and *(ii)* 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'. Let us give a short example where φ_1 is a frame that waits for a list v, prefixes it with the number 3 and sends this new list to the frame below. Let φ_2 be another frame that waits for a value and pair it with the number 3. Finally let φ_3 be a frame waiting for a list out of which is extracted its first term. Less formally but more visually:

$$\begin{aligned}\varphi_1 &= \lambda v.cons(3, v)\\ \varphi_2 &= \lambda v.cons(v, 3)\\ \varphi_3 &= \lambda v.car(v)\end{aligned}$$

With these frame behaviors, to push φ_1 over φ_3 is like removing φ_3 then pushing instead $\lambda v.3$. Alternatively to push φ_2 over φ_3 is simply to remove φ_3 (and avoid pushing φ_2 at all).

*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 first analysis, called continuation shape analysis, suggests which frames are likely to be combined and which functions might be specialized with respect to their continuation. Combining frames is essentially obtained through a kind of partial evaluation [Bon91] which is easier than in real Scheme since ACPS, heir of CPS, restricts the language. Of course, our program transformation suffers from the same problems as partial evaluation does regarding size of produced code or termination.

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 global 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. We will show, by hand, how our technique handles the examples (even the higher order ones) presented in [Chi92, HJ91].

The structure of the paper is as follows: we first introduce the ACPS transformation in Section 1. Continuation shape analysis is explained in Section 2. Section 3 handles combination of frames and presents examples of weeding. Section 4 treats the specialization of functions with respect to continuations. Other examples related to the litterature on deforestation are discussed in Section 6; related works and conclusions follow.

1 Abstract Continuation Passing Style

This Section introduces the ACPS program transformation. CPS makes continuations explicit but represents them by unary functions which can only be applied. ACPS improves on CPS since it adopts a less opaque representation for continuations: continuations are modeled by lists of frames. A frame is functionally represented by 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.

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

Table 1: ACPS rules for plain Scheme

Similarly to CPS, ACPS adds an extra argument to functions to represent their continuation: we name this argument q. Two operators, `resume` and `extend` encapsulate continuations. Sending a value v to a continuation q is expressed with `(resume` q v`)`. One extends a continuation q with a frame 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 frame on the value itself and on the rest of the frames:

Rule 1: `(resume (extend` q φ`)` v`)` $\equiv$ `(`φ q v`)`

A simple implementation of `resume` and `extend` in regular Scheme, where the continuation is represented by a list of regular closures, follows[1]. Observe that `resume` and `extend` also respect the interface of ACPS-transformed unary functions.

```
(define (resume q v) ((car q) (cdr q) v))
(define (extend q frame) (cons frame q))
```

The ACPS transformation is defined on table 1[2] for all the special forms of Scheme. The order of evaluation is left to right. ACPS creates a lot of administrative redexes. A reformulation of ACPS along the lines of [SF92] is probably possible but would 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. (trivial forms of [Rey72]) can also be simplified according to rule 2:

Rule 2: `(ACPS-`*primitive* `q` $v_1 \ldots v_n$`)` $\equiv$ `(resume q (`*primitive* $v_1 \ldots v_n$`))`

Let us give an example of ACPS transformation inspired from [HJ91] where the regular factorial of n is computed by first building the list of all positive numbers from 1 upto n then making the product of all of them. This program will serve as a running example for a large part of the paper.

```
(define (factorial n)
  (product (upto 1 n)) )
(define (product numbers)  ; (product '(v1 ... vn)) = ∏_{i=1}^n v_i
  (if (pair? numbers)
      (* (car numbers) (product (cdr numbers)))
      1 ) )
(define (upto start stop)  ; (upto 1 n) = (1 2 ... n)
  (if (<= start stop)
      (cons start (upto (+ 1 start) stop))
      '() ) )
```

The ACPS transformation of these functions appears below. To make it more readable, we assume `pair?`, `cons`, `car`[3] and other similar functions to be primitive and therefore to be left as they are by ACPS. We also made some cosmetical changes: *(i)* forms are simplified, whenever possible, according to rules 1 and 2, *(ii)* `let` forms were introduced instead of `((`λ`...)...)` and local variables are hygienically renamed, *(iii)* global variables have a name prefixed by `ACPS-`, *(iv)* all `lambda` forms are indexed with a number to distinguish them later, *(v)* forms like `(lambda (q v) (`f` q v))` are η-simplified into f,

```
(define ACPS-factorial
  (λ0 (q n) (ACPS-upto (extend q ACPS-product) 1 n)) )
(define ACPS-product
  (λ1 (q numbers)
    (if (pair? numbers)
        (let ((tmp1 (car numbers)))
          (ACPS-product (extend q (λ2 (q result) (resume q (* tmp1 result))))
                        (cdr numbers) ) )
        (resume q 1) ) ) )
(define ACPS-upto
  (λ3 (q start stop)
    (if (<= start stop)
        (ACPS-upto (extend q (λ4 (q result) (resume q (cons start result))))
                   (+ 1 start) stop )
        (resume q '()) ) ) )
```

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

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

[3] To consider car as a primitive is similar to inline it as in a real compiler. If an inlined car is applied on a non dotted pair, it is not possible to reify its exact continuation so this kind of error is not continuable. Therefore car has no perceivable control effect.

λ index	calls	after pushing
λ_0 (= `ACPS-factorial`)	λ_3	λ_1
λ_1 (= `ACPS-product`)	λ_1	λ_2
λ_3 (= `ACPS-upto`)	λ_3	λ_4

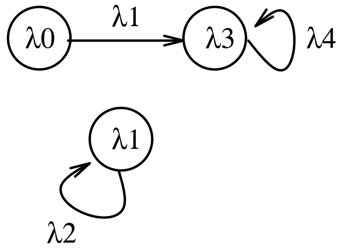


Figure 1: Results of continuation shape analysis

2 Continuation shape analysis

The `extend` operator extends a continuation with a frame. The paper proposes 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 link adjustment (in a heap-based implementation). Alternatively `extend` can be considered, from an Object Oriented point of view[4], as a generic function which may possess specialized methods to push some kinds of frames depending on the nature of the frame on top of the continuation. The nature of a frame is specified by the body of the abstraction that functionally represents this frame and is materialized, in the `factorial` example, by the λ-indices.

The continuation shape analysis proceeds as follows: for each non trivial application i.e. application that calls a function which is not a primitive, nor a `resume` nor an `extend` operator, we record, if known[5], the invoked function and the shape of the continuation i.e. the nature of the first frame(s). These are known when the continuation argument is an `extend` form with a frame explicitly specified by a `lambda` form.

The analysis is particularly simple in the case of the above `factorial` example: there are only three non trivial applications. Assuming that all global variables are immutable, the following table and figure show the results of the analysis:

This table does not exclude these functions to be called with other continuations. Conversely, if one of these functions, say `ACPS-product`, is known not to be called from elsewhere (for example as a result of an exportation directive within a module facility [QP91]) then more aggressive transformations can be easily imagined.

The results of the continuation shape analysis can be exploited in two ways.

1. The analysis reveals possible combinations of frames that might appear at run-time. A strongly typed language would restrict even more these possible combinations to type-compatible couples of frames. In our preceding example we see that: *(i)* λ_4 frames might be pushed over λ_4 or λ_1 frames, *(ii)* λ_2 frames might be pushed over λ_2 frames.

2. The analysis reveals functions that can be specialized with respect to their continuation i.e. that are called with some known frames on top of their continuation. Still from the example, we see that *(i)* λ_3 may be called over a λ_1 or a λ_4 frame, *(ii)* λ_1 may be called over a λ_2 frame.

These two aspects are instances of partial evaluation applied, in the first case, to `extend` forms and in the second case, to `lambda` forms where both are specialized with respect to the continuation. These transformations must be driven not to produce optimized dead code. These three aspects will be respectively handled in the three next Sections. The whole transformation appears as follows:

3 Combining frames

To combine frames φ and φ' (where φ is about to be pushed over φ') is similar to the invention of a new frame φ'' such that, for all continuation `q` and value `v`:

[4] this point of view is developped in [Que92a].
[5] Some analyzes, [Shi90] for instance, may improve this knowledge.

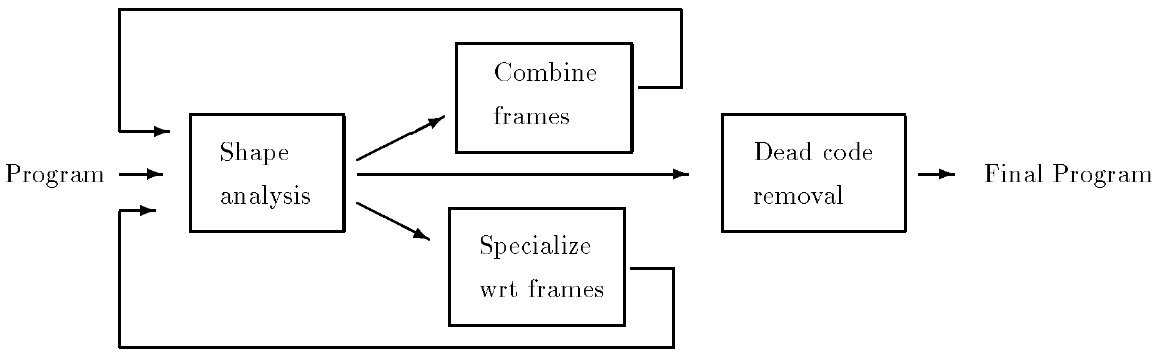


Figure 2: Continuation Conscious Compilation

```
(resume (extend (extend q φ') φ) v) ≡ (resume (extend q φ'') v)
```

The combined frame φ'' can be precisely defined as `(lambda (q v) (φ (extend q φ') v))`.

Not unlike partial evaluation, the problem is to determine where to stop inventing and combining new frames. Practically, we only invent new frames that decrease the size of the body of the associated `lambda` form compared to the sum of the sizes of the body of the combined `lambda` forms. This ensures a finite number of new frames but also promises some increase of the size of the code.

Not all combinations are interesting but when they are i.e. when the partial evaluation of φ'' leads to some gain, then we can transform all `extend` forms that pushes a φ frame on top of a φ' frame so they analyze their continuation and if appropriate, remove that φ' frame and push instead a φ'' frame. In more programming terms, we replace `(extend q φ)` with:

```
(cond ((is a φ'? (car q)) (extend (cdr q) φ''))
      (else (extend q φ)) )
```

To build the combined frame requires to be able to recognize the nature of frames and to extract values of closed variables from closures. To make this paper more readable, we express our ideas directly on ACPS code and not on λ-lifted code as done in [Que92a] since this would expose far too much details. We will therefore suppose λ_i`?` to be a predicate recognizing if its argument is a closure with a λ_i nature and $\lambda_i\downarrow$`v` to be a selector that extracts the value of the closed variable `v` out of a λ_i closure.

Returning to our running example, we can easily see that combining a (just consing) λ_4 frame with a (just consing) λ_4 frame is not interesting since no simplification can be performed on the combined frame.

3.1 Weeding

Among the suggested combinations by figure 1 was the combination of a (just consing) λ_4 over a (destructure for multiplying) λ_1 frame. To combine them is similar to partially evaluate (modulo rules 1 and 2):

```
(lambda (q v) ;(λ4 (extend q λ1) v)
  ((λ4 (q result) (resume q (cons start result)))
   (extend q (λ1 (q numbers)
               (if (pair? numbers)
                   (let ((tmp1 (car numbers)))
                     (ACPS-product (extend q (λ2 (q result)
                                                 (resume q (* tmp1 result)) ))
                                   (cdr numbers) ) )
                   (resume q 1) ) ))
   v ) )
```

This is easily simplified into the following. Note that we can here use rules as `(car (cons` x y`))=`x since due to ACPS, forms x and y are trivial, terminate and have no control effects.

```
(lambda7 (q v)
  (ACPS-product (extend q (λ2 (q result) (resume q (* start result))))
```

```
v ) )
```

It is important to limit the number of different frames since when new ones are invented, the continuation shape analysis must be resumed to take these new frames into account. Therefore before inventing new frames and even if this test is expensive, we check if they already exist. For instance, in the above expression the `lambda` form can be recognized as an instance of a λ_1 frame i.e. a call to `ACPS-product`. Therefore we see that pushing a λ_4 over a λ_1 frame can be performed by removing the λ_1 frame then pushing a λ_7 frame. This last action `(extend q` λ_7`)` is easily simplified into `(extend (extend q` λ_2`)` λ_1`)` so, finally, to push a λ_4 over a λ_1 frame can be performed by removing the λ_1 frame then sequentially pushing a λ_2 and again a λ_1.

This combination of frames has an immediate gain which is to remove the need to construct a pair which is immediately destructured. This achieves a deforestation-like effect but on a small scale, something more akin to weeding. Weeding removes useless computations, these useless computations can concern data allocations as well as really useless computations. Consider,for example, the following function

```
(define (main n)
  (factorial n)
  n )
```

which is ACPS-translated into:

```
(define ACPS-main
  (λ10 (q n)
    (ACPS-factorial (extend q (λ11 (q v) n)) n) ) )
```

It is easy to see that λ_{11} ignores the value it will receive, it is an absorbing frame that swallows any frame that `resumes` it and in particular λ_2 frames that will occur during the computation of `ACPS-factorial`. A λ_{11} frame cannot of course be combined with a frame which is not known to terminate, for example, an `ACPS-product` frame. Weeding there allows to remove all the multiplying λ_2 frames produced by `ACPS-factorial` but leaves the frames enumerating the numbers from `n` to `1`, a potentially looping activity if `n` is negative.

3.2 Using associativity

Another possible combination, suggested by the continuation shape analysis, was to push a (multiplying) λ_2, say φ, over a (multiplying) λ_2 frame φ'. The same partial evaluation as above leads to:

```
(lambda (q v) ;(φ (extend q φ') v)
  (resume q (* (λ2↓tmp1 φ') (* (λ2↓tmp1 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. All previous reorganizations never use such properties, they were pure applications of fold/unfold/ specialize/generalize strategies not beyond regular compiler ability.

If we rearrange the above multiplications using associativity, we can group the two multiplicands and obtain:

```
(lambda (q v) (resume q (* (* (λ2↓tmp1 φ') (λ2↓tmp1 φ)) v)))
```

We already know a frame which has this behavior: λ_2[6]! Therefore, given the associativity of the multiplication, two λ_2 frames can be combined into a single one.

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 frames combination with an update-in-place effect. Since a λ_2 is replaced with another instance of λ_2, 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.

Let us return to our running example to show what we achieved so far. If we retain all the analyzed combinations described so far, then we obtain the following code:

```
(define ACPS-factorial
```

[6] It is beneficial to share frames to reduce their number.

```
    (λ0 (q n) (ACPS-upto (extend q ACPS-product) 1 n)) )
  (define ACPS-product
    (λ1 (q numbers)
      (if (pair? numbers)
          (let ((tmp1 (car numbers)))
            (ACPS-product
             (cond ((λ2? (car q)) ; when called from ACPS-product
                    (extend (cdr q)
                            (let ((tmp2 (λ2↓tmp1 (car q))))
                              (λ2 (q result)
                                (resume q (* (* tmp2 tmp1) result)) ) ) ) ) )
                   (else (extend q (λ2 (q result) (resume q (* tmp1 result)))) ) )
             (cdr numbers) ) )
          (resume q 1) ) ) )
  (define ACPS-upto
    (λ3 (q start stop)
      (if (<= start stop)
          (ACPS-upto
           (cond ((λ1? (car q)) ; when called from ACPS-factorial
                  (extend (cond ((λ2? (cadr q))
                                 (extend (cddr q)
                                         (let ((tmp2 (λ2↓tmp1 (cadr q))))
                                           (λ2 (q r)
                                             (resume q (* (* tmp2 start) r)) ) ) ) ) )
                                (else (extend (cdr q)
                                              (λ2 (q result)
                                                (resume q (* start result)) ) )) ) )
                          ACPS-product ) )
                 (else (extend q (λ4 (q result) (resume q (cons start result))))) )
           (+ 1 start) stop )
          (resume q '()) ) ) )
```

Although oversized, this transformed program is able to compute the `factorial` of a number without any allocation of pairs and with a continuation containing at most two more top frames, one λ_2 and one λ_1 frames. The transformation not only improves the evaluation of `factorial`, it also improves the `product` function as well since it now computes its final result with a continuation containing at most one λ_2 top frame.

Perhaps we can make this behavior clearer showing the state of the computation while computing `(factorial n)`. In the figure below, the state is represented by the current form over the continuation; closed values appear between parentheses just after the name of the closure.

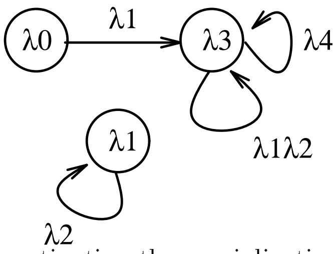
$$\frac{\mathit{factorial}\ n}{\ } \rightarrow \frac{\mathit{upto}\ 1\ n}{\lambda_1} \rightarrow \frac{\mathit{upto}\ 2\ n}{\begin{array}{c}\lambda_1\\ \lambda_2(1)\end{array}} \rightarrow \frac{\mathit{upto}\ 3\ n}{\begin{array}{c}\lambda_1\\ \lambda_2(1*2)\end{array}} \rightarrow \ldots \rightarrow \frac{\mathit{upto}\ n{+}1\ n}{\begin{array}{c}\lambda_1\\ \lambda_2(1*2*\ldots*n)\end{array}} \rightarrow \frac{\mathit{product}\ ()}{\lambda_2(n!)} \rightarrow \frac{1*n!}{\ }$$

Two problems still affect this code: it is oversized and all these improvements are dynamic since frames are recognized at run-time. It is then time to exploit the second aspect of the continuation shape analysis which is to now specialize functions with respect to their continuation.

4 Specialization with respect to the continuation

Due to the previous transformation, the continuation shape analysis has slightly evolved, new frames were created that modify the shape of involved continuations. The analysis now produces:

λ index	calls	with top frames
λ_0 (= `ACPS-factorial`)	λ_3	λ_1
λ_1 (= `ACPS-product`)	λ_1	λ_2
λ_3 (= `ACPS-upto`)	λ_3	λ_4
λ_3 (= `ACPS-upto`)	λ_3	λ_1 λ_2



We can easily extract from the above table that it is worth investigating the specialization of λ_1 with a top frame of λ_2 as well as the specializations of λ_3 with a top frame of λ_4 or two top frames: λ_1 and λ_2. These are possibilities of specialization that do not prevent these functions to be called with other continuations. The continuation shape analysis just suggests some specializations. There again, not all possibilities lead to interesting specializations. To be interesting, a function to be specialized must be called more than once as well as offer some gain i.e. be simpler than its non specialized version. This is a kind of polyvariant partial evaluation since a same function may be specialized with respect to different continuations. The suggested specializations of the `factorial` example seem to be particularly interesting since they all lead to recursive specializations.

Let us first consider the case of specializing a function with respect to a single frame on top of the continuation, we call this operation: "integrating the top frame". Specializing a function with respect to more than one frame can be done by successive top frames integrations. The operation is a two phase transformation: a raw specialization of all the functions that might benefit from the knowledge of the top frame followed by a general retrofit of all invocations to these functions.

4.1 Raw specialization

Suppose a function f with variables `q`, $v_1 \ldots v_n$ to be specialized with respect to a frame λ_i, say φ, with free variables $c_1 \ldots c_p$. The specialized version will be a function, say f`-over`λ_i, such that:

$$(f\ \texttt{(extend q}\ \lambda_i\texttt{)}\ v_1\ \ldots\ v_n\texttt{)} \equiv \texttt{(}f\texttt{-over}\lambda_i\ \texttt{q}\ v_1\ \ldots\ v_n\ \lambda_i{\downarrow}c_1(\varphi)\ \ldots\ \lambda_i{\downarrow}c_j(\varphi)\texttt{)}$$

The definition of f`-over`λ_i is obtained as follows. Its list of variables is the list of variables of f plus the variables $c_1 \ldots c_p$. Its body is the simplified expression `(`f `(extend q` λ_i`)` $v_1 \ldots v_n$`)` where the values of `(`$\lambda_i{\downarrow}c_j$ `(car q))` are renamed c_j, this is just a sort of λ-lifting modulo rules 1 and 2.

For instance, this is the raw result of this transformation on `ACPS-product` with respect to λ_2:

```
(define ACPS-product-overλ2
  (λ5 (q numbers tmp1)
    (if (pair? numbers)
        (let ((tmp3 (car numbers)))
          (ACPS-product ; We know we are over a λ2
           (extend q (let ((tmp2 tmp1))
                       (λ2 (q result)
                         (resume q (* (* tmp2 tmp3) result)) ) ))
           (cdr numbers) ) )
        (resume (extend q (λ2 (q result) (resume q (* tmp1 result))))
                1 ) ) ) )
```

4.2 General retrofit

The second phase of the transformation consists in changing all invocations to f with a top frame of λ_i i.e. a form like `(`f `(extend` q `(`λ_i `...))` `...)`, into the appropriate call to f`-over`λ_i. This transformation is performed everywhere even in the newly generated functions. So the raw (and still to be simplified) definition of `ACPS-product-over`λ_2 becomes:

```
(define ACPS-product-overλ2
  (λ5 (q numbers tmp1)
    (if (pair? numbers)
```

```
            (let ((tmp3 (car numbers)))
              (ACPS-product-overλ2
               q (cdr numbers) (let ((tmp2 tmp1)) (* tmp2 tmp3)) ) )
            (resume (extend q (λ2 (q result) (resume q (* tmp1 result))))
                    1 ) ) ) )
```

The result is particularly interesting since it leads to a tail-recursive function. An ACPS inverse transformation [SF92, DL92b] would rewrite the above function back to direct style into:

```
(define (product-overλ2 numbers tmp1)
  (if (pair? numbers)
      (let ((tmp3 (car numbers)))
        (product-overλ2
         (cdr numbers) (let ((tmp2 tmp1)) (* tmp2 tmp3)) ) )
      (* tmp1 1) ) )
```

Observe that the technique that integrates the top frame applied to the unary recursive `product` function turns it into its binary iterative (with accumulator) equivalent. The same effect can be observed on the regular recursive factorial which is turned into its iterative equivalent [FWH92, Chap. 10].

5 Dead code removal

Let us summarize the results obtained so far in two tables. The first one shows the results of combining frames, the second the results of top frame integration.

pushing	
λ_4 over λ_4	uninteresting
λ_4 over λ_1	(weeding) push instead λ_1 on top of λ_2
λ_2 over λ_2	(associativity) push instead λ_2

calling	
λ_1 over λ_2	call instead `ACPS-product-over`λ_2
λ_3 over λ_4	not interesting
λ_3 over $\lambda_1\lambda_2$	call instead `ACPS-upto-over`λ_1
`ACPS-upto-over`λ_1 over λ_2	call instead `ACPS-upto-over`$\lambda_1\lambda_2$

There is still room for improvements (as well as other analyses such as closure analyses) since lot of administrative redexes have been produced. Another effect is that, specializing functions, we multiply their number so we must refine our continuation shape analysis to take into account these new possible interactions. Fortunately, combining frames or specializing functions allows to suppress some type of λforms since they can no more be built so we can remove all the conditional code relative to them.

Another source of simplification is whether the transformation is applied on a whole program or a bunch of functions. If we consider a whole program and the initial call is an invocation to `factorial` then we can just keep `ACPS-fatorial`, `ACPS-upto-over`λ_1 and `ACPS-upto-over`$\lambda_1\lambda_2$. If we allow as well direct calls to `product` and `upto` then we must add `ACPS-upto`, `ACPS-product` and `ACPS-product-over`λ_2.

All the previous analyses were static and yield static improvements. But the information acquired so far also allows to install some code that will, at run-time, try to check whether some statically undetected but dynamically possible optimizations occur. These various levels of code removal appears in appendix for the `factorial` example.

6 Other examples

We discuss in this Section how our technique handles other examples taken from the literature on deforestation and mainly [HJ91, Chi92]. The following are classical and come from [HJ91], `main1` also appears in [Chi92]:

```
(define (append x y)
  (if (pair? x)
```

```
        (cons (car x) (append (cdr x) y))
        y ) )
  (define (reverse2 x y)
    (if (pair? x)
        (reverse2 (cdr x) (cons (car x) y))
        y ) )
  (define (main1 x y z)
    (append (append x y) z) )
  (define (main2 x y z)
    (append (reverse2 x y) z) )
  (define (main3 x y z)
    (reverse2 (append x y) z) )
```

Examples `main1` and `main3` do not pose any problems. It is easy to see that the inner call to `append` will have to push a consing frame that can be fused with the outer destructuring `append` or `reverse2` waiting frame. The case of `main2` is interesting since, as noted in [HJ91], regular deforestation [Wad88] cannot handle it but a more complex analysis, creation analysis, can take care of this. Our technique is unable to improve anything since `reverse2` is tail-recursive and as such does not push any frame on the continuation. Unless frames exist that can be combined or serve to specialize functions, our analysis is impotent.

The following example comes from [Chi92], it takes two lists and counts the number of their terms (in a rather inefficient manner (but these sort of transformations are at their best when programs are not efficiently coded):

```
  (define (size l1 l2)
    (length (append l1 l2)) )
  (define (length l)
    (if (pair? l)
        (+ 1 (length (cdr l)))
        0 ) )
```

The ACPS transformation yields:

```
  (define ACPS-size
    (λ20 (q l1 l2) (ACPS-append (extend q ACPS-length) l1 l2)) )
  (define ACPS-length
    (λ21 (q l)
      (if (pair? l)
          (ACPS-length (extend q (λ22 (q r) (resume q (+ 1 r))))
                       (cdr l) )
          (resume q 0) ) ) )
  (define ACPS-append
    (λ23 (q x y)
      (if (pair? x)
          (ACPS-append (extend q (λ24 (q r) (cons (car x) r)))
                       (cdr x) y )
          (resume q y) ) ) )
```

Because `append` is a producer of pairs (using frame λ_{24}) that are consumed by `length` i.e. frame λ_{21}, our technique works well, combines these frames into, say λ_{25} and therefore removes intermediate data allocation automatically. Our technique rediscovers that:

```
  (length (size (cons x y) z)) = (+ 1 (size y z))
```

We thus do not need to explicitly introduce, as Chin does, a law such as:

```
  (length (append l1 l2)) = (+ (length l1) (length l2))
```

The invention of frame λ_{25} leads to a computation that pushes as many λ_{22} frames as they are terms in the two arguments of `size`. The continuation shape analysis suggests to try to combine λ_{22} frames but unfortunately the combination is not interesting since, even using associativity of the addition, to combine

`1+` with `1+` only yields `(lambda (q r) (resume q (+ 2 r)))`. We cannot invent such a frame for each natural number. What is needed to combine these frames is to be able to generalize this kind of frame into one which waits for a value and will add it to a closed value. This ability is related to the problem of sharing frames to limit the invention of new frames when parameterizing old ones may suffice. This is an instance of a generalization process, a difficult and complex task.

7 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 ACPS 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.

It is of course possible to make the same analysis on a non ACPS-transformed program but we thought ACPS is more convenient to relate small parts of code together. Whenever two frames are known to be possibly contiguous, their combination can be analyzed. These frames do not need to appear at the beginning of the continuation, they might appear deeply inside. An interesting alley seems to improve programs according to our technique then to translate them back to direct style [DL92a, SF92] to take benefit of regular Scheme compilers.

We compare with some details our technique to deforestation in the previous Section. To summarize, our technique *(i)* is purely local: it 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; two side-effect free frames can be combined even if there are side-effects elsewhere. *(ii)* is able to handle full Scheme with higher order functions, *(iii)* improves programs by weeding intermediate data allocations, useless computations and thus shortens continuations. For the sole point of deforestation, it is inferior to the technique of [HJ91] as shown in Section 6. Compared to [Chi92], it needs less additional laws but requires the ability of generalizing frames.

8 Conclusions

The paper presented a program transformation which crucially depends on the shape of the continuations. The various possible 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 associative recursion to be made iterative. Combining frames and specializing functions with respect to such frames are applications of partial evaluation.

We presently have no figures of the usefulness of these ideas. But it is sure that efficiently coded programs will not gain from this technique since they already avoid to allocate short-lived objects as well as recursions that are not tail-recursive.

Bibliography

[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.

[DL92a] 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).

[DL92b] Olivier Danvy and Julia L. Lawall. Back to direct style II: First-class continuations. In *LFP '92 – ACM Symposium on Lisp and Functional Programming*, LISP Pointers, Vol. V, No. 1, pages 299–310, San Francisco, California, June 1992. ACM Press.

[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 LITPy 78-2.

[HJ91] G W Hamilton and S B Jones. Extending deforestation for first order functional programs. In *Procedings of the 1991 Glasgow Workshop on Functional Programming*, pages 134–145, Portree, Isle of Skye (United Kingdom), August 1991.

[QD93] Christian Queinnec and David De Roure. Design of a concurrent and distributed language. In Robert H Halstead Jr and Takayasu Ito, editors, *Parallel Symbolic Computing: Languages, Systems, and Applications, (US/Japan Workshop Proceedings)*, volume Lecture Notes in Computer Science 748, pages ??–??, Boston (Massachussetts US), October 1993.

[QP91] Christian Queinnec and Julian Padget. Modules, Macros and Lisp. In *Eleventh International Conference of the Chilean Computer Science Society*, pages 111–123, Santiago (Chile), October 1991. Plenum Publishing Corporation, New York NY (USA).

[Que92a] Christian Queinnec. Continuation sensitive compilation. Research Report LIX RR 92/14, Laboratoire d'Informatique de l'École Polytechnique, 91128 Palaiseau Cedex, France, November 1992.

[Que92b] Christian Queinnec. Value transforming style. In M Billaud, P Castéran, MM Corsini, K Musumbu, and A Rauzy, editors, *WSA '92—Workshop on Static Analysis*, number 81-82 in Revue Bigre+Globule, pages 20–28, Bordeaux (France), September 1992.

[Que92c] 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).

[Shi90] Olin Shivers. Data-flow analysis and type recovery in scheme. Technical Report CMU-CS-90-115, CMU School of Computer Science, Pittsburgh, Penn., March 1990.

[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.

Appendix: The final `factorial` example

If the whole program is nothing but a call to **`factorial`**, then it can be transformed into:

```
(define ACPS-factorial
  (λ0 (q n) (ACPS-upto-overλ1 q 1 n)) )
(define ACPS-upto-overλ1
  (λ6 (q start stop)
    (if (<= start stop)
        (ACPS-upto-overλ1λ2 q (+ 1 start) stop start)
        (resume q 1) ) ) )
(define ACPS-upto-overλ1λ2
  (λ7 (q start stop tmp1)
    (if (<= start stop)
        (ACPS-upto-overλ1λ2 q (+ 1 start) stop (* tmp1 start))
        (resume q tmp1) ) ) )
```

If we still allow direct calls to functions `product` and `upto` we have to add the following definitions:

```
(define ACPS-product
  (λ1 (q numbers)
    (if (pair? numbers)
        (let ((tmp1 (car numbers)))
          (ACPS-product-overλ2 q (cdr numbers) tmp1) )
        (resume q 1) ) ) )
(define ACPS-product-overλ2
  (λ5 (q numbers tmp1)
    (if (pair? numbers)
        (let ((tmp3 (car numbers)))
          (ACPS-product-overλ2 q (cdr numbers) (* tmp1 tmp3)) )
        (resume q tmp1) ) ) )
(define ACPS-upto
  (λ3 (q start stop)
    (if (<= start stop)
        (ACPS-upto
         (extend q (λ4 (q result) (resume q (cons start result))))
         (+ 1 start) stop )
        (resume q '()) ) ) )
```

Finally, if we wish to allow dynamic optimizations to occur, then we have to replace `upto` (the sole function to push frames on an unknown continuation) by:

```
(define ACPS-upto
  (λ3 (q start stop)
    (if (<= start stop)
        (cond ((λ1? (car q))
               (cond ((λ2? (cadr q))
                      (let ((tmp (λ2↓tmp1 (cadr q))))
                        (ACPS-upto-overλ1λ2
                         (cddr q) (+ 1 start) stop (* tmp start) ) ) )
                     (else (ACPS-upto-overλ1
                             (cdr q) (+ 1 start) stop start )) ) )
          (else (ACPS-upto (extend q (λ4 (q result)
                                       (resume q (cons start result)) ))
                           (+ 1 start) stop )) )
        (resume q '()) ) ) )
```

In all these examples, some obvious transformations were performed such as simplifying `(*` v `1)` into v.