Regions as Continuation Marks

Paulette Koronkevich
University of British Columbia
Vancouver, Canada
pletrec@cs.ubc.ca

William J. Bowman
University of British Columbia
Vancouver, Canada
wjb@williamjbowman.com

Abstract

Region-based memory management started as a compiler pass to optimize Standard ML programs. Garbage collection was expensive, manual memory management was unsafe, and so instead one infers a stack of *memory regions* to enable the compiler to insert explicit memory allocation and deallocation instructions. The result of region inference is a ML program with lexically-scoped regions. Unfortunately, the stack-based region inference interferes with the correct behaviour of tail recursion: that tail recursion must run in constant space, and much work was required to correct their behaviour in practice.

We show a formal connection between these lexical regions and *continuation marks*, which elucidates the behaviour of regions with respect to tail calls, and provide insights into possible implementation techniques and new semantics. Continuation marks enable placing a value on the dynamic program frame, and were designed and implemented to behave correctly with respect to tail recursion. We provide a translation from regions to continuation marks and theorize what this connection could entail for both languages with region-based memory management and our favourite Schemes (and Racket) with continuation marks.

CCS Concepts: • **Software and its engineering → Functional languages**; • **Theory of computation → Operational semantics**.

Keywords: Continuation marks, Memory regions

ACM Reference Format:
Paulette Koronkevich and William J. Bowman. 2026. Regions as Continuation Marks. In *Proceedings of the 27th ACM SIGPLAN International Workshop on Scheme and Functional Programming (Scheme '26), August 24–29, 2026, Indianapolis, IN, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3830440.3832303

1 Introduction

Region-based memory management was originally used as an memory optimization technique for Standard ML [8, 9], and now has made it into the mainstream with languages like Rust. The idea is that garbage collection in functional languages is expensive, but manual memory management is tedious, unsafe, and not why we use and love functional languages. The original region based memory management explicated the memory behaviour of a functional program into a stack of *memory regions*, so that the memory usage of functional programs could be implemented by explicit allocations by the compiler rather than relying on an external garbage collector. The original analysis annotated ML programs with an explicit **letregion** construct to lexically indicate the regions of memory allocated and deallocated.

As an example, consider the following ML program.

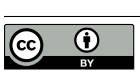
$$\textbf{let } \mathsf{x} = (2,3) \textbf{ in } \boldsymbol{\lambda}\ \mathsf{y}.(\textbf{fst } \mathsf{x}, \mathsf{y}) \textbf{ end } 5$$

To make explicit the memory usage of the program, Tofte and Talpin [8] introduced two new main constructs into the target language: **letregion** ρ **in** e **end** and e **at** ρ. The first introduces a new region name ρ that can be used within the body e, in particular, using the e **at** ρ construct to allocate the value of an expression e into the region ρ. The region ρ is then deallocated after **end**. The above example then translates to the following explicitly region annotated program, copied from Tofte and Talpin [8, 9].

$$\begin{array}{l}\textbf{letregion } \mathsf{r}_1, \mathsf{r}_2, \mathsf{r}_3 \textbf{ in} \\ \quad (\textbf{letregion } \mathsf{r}_4, \mathsf{r}_5 \textbf{ in} \\ \quad\ \textbf{letregion } \mathsf{r}_6 \textbf{ in} \\ \quad\quad \textbf{let } \mathsf{x} = (2 \textbf{ at } \mathsf{r}_2, 3 \textbf{ at } \mathsf{r}_6) \textbf{ at } \mathsf{r}_4 \textbf{ in} \\ \quad\quad (\boldsymbol{\lambda}\ \mathsf{y}.(\textbf{fst } \mathsf{x}, \mathsf{y}) \textbf{ at } \mathsf{r}_1) \textbf{ at } \mathsf{r}_5 \textbf{ end end} \\ \quad 5 \textbf{ at } \mathsf{r}_3) \textbf{ end end}\end{array}$$

The final value of the program is stored in regions $\mathsf{r}_1, \mathsf{r}_2, \mathsf{r}_3$, assuming that the program uses these regions to communicate the value to the outside world. The program first introduces the regions $\mathsf{r}_1, \mathsf{r}_2, \mathsf{r}_3$. Then, we have the intermediate regions $\mathsf{r}_4, \mathsf{r}_5, \mathsf{r}_6$ that are deallocated after their respective dynamic extents. These intermediate regions hold the values not needed for the final value of the program, in particular, the second component of the pair x has 3 **at** r_6, and the pair x itself **at** r_4. Likewise, the closure $(\boldsymbol{\lambda}\ y.(\textbf{fst } x, y))$ is allocated in r_5, a region deallocated after it is applied to 5. Note that in this model all values, even word-sized values, are boxed values on the heap, though additional optimizations by Birkedal et al. [2] can change values to be stack or register allocated.

Unfortunately, the stack-based memory management of the original inference algorithm interferes with the expected behaviour of functional tail recursive calls [2, 8]. In particular, follow-up work finds that additional complex analyses



Scheme '26, Indianapolis, IN, USA

ACM ISBN 979-8-4007-2870-9/2026/08
https://doi.org/10.1145/3830440.3832303

are needed in addition to the region inference pass [2]: an analysis called *storage mode analysis* is required for tail recursive functions to run in constant space. We can see this in small example programs.

First, consider the previous example. If we are not careful with the interpretation and implementation of the expression **letregion** r **in** e, the inference can transform a tail call into a non-tail call by wrapping it in **letregion**, which conceptually performs allocation before the call and deallocation after the call. This would be particularly problematic in a tail recursive call: **letrec** *tailcall* = ...**letregion** r **in** (*tailcall e*)... would allocate regions linear in number of tail calls. Ensuring that region introduction around tail call uses constant space requires destructive updates to regions, an equivalence rule roughly like:

$$\textbf{letregion } r \textbf{ in letregion } r \textbf{ in } (\mathit{tailcall}\ e) \quad = \quad \textbf{letregion } r \textbf{ in } (\mathit{tailcall}\ e)$$

The **letregion** construct is not the only problem. The introduction of **at** annotations on tail calls causes a similar problem. Consider the following tail recursive factorial program. This particular program takes in a pair where the first element is the argument to factorial, and the second element is the accumulator. The tail recursive function produces the pair with the values of the final argument and accumulator.

```
letrec fact-tr = λ (p). let n = fst p, a = snd p in
                        if n < 1
                         then p
                         else (fact-tr (n − 1, n ∗ a))
 in (fact-tr (5, 1)) end
```

Consider the same function, now annotated with regions from the analysis, copied from Birkedal et al. [2].

```
letregion r1 in
 letrec fact-tr at r1 =
  λ [r2, r3, r4] (p). let n = fst p, a = snd p in
                      if n < 1
                       then p
                       else    letregion r5 in
                               fact-tr[r2, r3, r4] at r5
                                (letregion r6 in
                                (n − (1 at r6)) at r3
                                end , n ∗ a at r4) at r2
                              end
 in ... end end
```

Phew! That's quite the region annotated program. The expected behaviour of this program is as follows: first, the recursive closure of `fact-tr` is allocated into the freshly introduced region r_1. The function `fact-tr` is polymorphic in three regions, so within the scope of the function the regions r_2, r_3, r_4 are available for use. This means that any future call sites of `fact-tr` can use any region in scope to substitute in for r_2, r_3, r_4. The region r_2 is for the region where the single parameter `p` is allocated The next region r_3 is for the region where the first element of the pair `p` is allocated, and finally the last region r_4 is where the second element of the pair `p` is allocated. The tail recursive call in the **else** branch introduces an intermediate regions r_5, r_6 for allocating the value of the closure and the value 1 respectively. Finally, the polymorphic regions r_2, r_3, r_4 are used for allocating the pair and its elements for the recursive call.

We note one major problem with the inference algorithm, which required additional analyses by Birkedal et al. [2] to recover the constant space of tail recursion. If **at** expressions are to be interpreted as "copy" into a region, then the analysis has forced *each* argument pair to be allocated in the regions represented as ρ_2, ρ_3, ρ_4. To recover the constant space of tail calls, each tail-recursive argument should be overwrite the last tail recursive argument, essentially occurring at the bottom of the call stack every time. However, what actually occurs is that each pair is allocated in the same region, which causes tail recursion to take space proportional to the number of recursive calls. This problem also occurs with the inference allocating of the tail recursive closure **at** r_5, where the tail recursive closures are re-allocated each time into ρ_5 rather than overwritten.

How we expect the semantics of regions to work with respect to tail recursion, by the dynamic semantics of the region annotated language and by the source language itself, does not match the actual compiler implementation and analysis. Despite the wins that regions may attain with intermediate values and non-tail calls, prior to the additional storage mode analysis, a program consisting of two tail-recursive functions (one a tail recursive loop over a number until 0, the other a tail recursive minus) on an input of 2000 runs of memory, instead of using constant space [2]. As we consider the constant space of tail recursion as a "must-have" for functional language implementations, regions are quite disappointing as an initial abstraction.

Meanwhile, in PLT Scheme and Racket, *continuation marks* were developed, enabling adding a dynamic value to a program's frame inside using a **wcm** expression. Continuation marks were originally used to model an algebraic stepper [3], as they could record each context as a continuation mark, and still obey the dynamic semantics of tail calls. Consider the following evaluation rules of continuation marks.

$$\begin{array}{rcl} E\,[(\textbf{wcm}\ v_1\ (\textbf{wcm}\ v_2\ e))] & \rightarrow & E\,[(\textbf{wcm}\ v_2\ e)] \\ & & E \neq E'[(\textbf{wcm}\ v_3\ [])] \\ E\,[(\textbf{wcm}\ v_1\ v_2)] & \rightarrow & E\,[v_2] \end{array}$$

Here we see that, given two nested **wcm** expressions, the outermost value is deleted. The key idea takeaway is that **wcm** expressions preserve the dynamic behaviour of tail calls as they are prevented from stacking on one another. Later, PLT Scheme and Racket extended the usage of these continuation marks to associate a key with each value that

could be updated [5, 7], as well as be combined with other forms of delimited control.

Another key implementation detail of continuation marks is that after the body is evaluated, the value on the dynamic frame is thrown away. This behaviour is similar to regions, where we would like to introduce a new binding to be used in the program that is deallocated after the construct, but still respects tail calls.

In this paper, we present a formal connection between the regions from Standard ML and continuation marks from Scheme and Racket. This connection makes the problem with regions and tail calls more clear, and possibly provides an easy implementation technique for regions. We formalize a translation and an evaluation for a new **at** expression in the context of Scheme in Section 3. We provide some more examples and comparisons to regions, and show how continuation marks can represent regions, but the implementation of region-polymorphic functions is what causes the trouble with constant space of tail calls. We propose an alternative to the implementation of region-polymorphic functions based on our connection to continuation marks. We then conclude in Section 4 with some future ideas based on our connection.

2 Regions and Continuation Marks

In this section, we review some key properties of the evaluation of region-based languages and continuation marks, and show a short example comparing the evaluation of both systems. We see that by translating regions to continuation marks, where the keys are the region names and the values are the store, we can achieve the same lifetimes as the original regions.

Region-based memory management is a type-based translation to statically determine where in the program memory is allocated and deallocated. The original idea was to create a translation that inferred the lifetimes of values in a statically typed higher-order programming language, in order to allocate and deallocate through the compiler rather than relying on garbage collection. The target language with regions included a dynamic semantics, and was used to prove the language sound with respect to the static semantics and to prove the inference translation correct. We copy the dynamic semantics for region introduction, value allocation into a region, polymorphic function definition, and polymorphic application, directly from Tofte and Talpin [9] as these are the main rules concerned with regions. Each rule consists of the form $s, VE, R \vdash e \longrightarrow v, s$, where s is the store mapping addresses to values, VE is the value environment, and R is the region environment mapping abstract region variables to concrete region variables.

$$\frac{r \notin \mathrm{Dom}(s) \qquad s + \{r \mapsto ()\}, VE, R + \{\rho \mapsto r\} \vdash e \longrightarrow v, s'}{s, VE, R \vdash (\textbf{letregion } \rho \textbf{ in } e \textbf{ end }) \longrightarrow v, s' \setminus \{r\}}$$

$$\frac{r = R(\rho) \qquad o \notin \mathrm{Dom}(s(r))}{s, VE, R \vdash (c \textbf{ at } \rho) \longrightarrow (r, o), s[(r, o) \mapsto c]}$$

For region introduction, the region ρ is mapped to a concrete region name r and recorded in the region environment for the evaluation of the body e, with an updated store that shows the region r initially stores no values. When allocating into a region, the allocation steps to a fresh address in the store (a pair of a region name r and offset o, or also represented as a), and updates the store accordingly.

$$\frac{\begin{array}{c} r = R(\rho) \qquad o \notin \mathrm{Dom}(s(r)) \qquad VE' = VE + \{f \mapsto (r, o)\} \\ s + \{(r, o) \mapsto \langle \rho_1, \ldots \rho_k, x, e_1, VE', R\rangle\}, VE', R \vdash e_2 \longrightarrow v, s' \end{array}}{s, VE, R \vdash (\textbf{letrec } f[\rho'_1, \ldots \rho'_k](x) \textbf{ at } \rho = e_1 \textbf{ in } e_2 \textbf{ end }) \longrightarrow v, s'}$$

$$\frac{\begin{array}{c} VE(f) = a \qquad s(a) = \langle \rho_1, \ldots \rho_k, x, e, VE_0, R_o\rangle \\ r = R(\rho) \qquad o \notin \mathrm{Dom}(s(r)) \\ sv = \langle x, e, VE_0, R_0 + \{\rho_i \mapsto R(\rho_i); 1 \le i \le k\}\rangle \end{array}}{s, VE, R \vdash f[\rho_1, \ldots \rho_k] \textbf{ at } \rho \longrightarrow (\rho, o), s[(r, o) \mapsto sv]}$$

For region polymorphic function introduction, the store is updated with a fresh address that stores the region polymorphic closure, containing the region variables, parameters, body, and environment. The value environment is then updated with the mapping of the function name to the fresh address. When applying a region polymorphic function to region names, the stored value of the closure is updated to contain mappings from the prior region parameters to the current region names in the region environment.

Continuation marks were introduced in Scheme, an untyped functional language, originally to model the algebraic stepper in Dr. Scheme [3]. The main idea is to associate a value with a dynamic program frame instead of a lexical context. To implement the dynamic behaviour of the program correctly, continuation marks must preserve the frame overwriting behaviour of tail calls. The key insight is that the evaluation of continuation marks must explicitly disallow the use of nesting that would cause stack frames to build up. Here, we reproduce the evaluation rules of the original paper.

$$\begin{array}{lcl} E & ::= & F \mid (\textbf{wcm } w\ F) \\ F & ::= & [] \mid (E\ \overline{e}) \mid \ldots \end{array}$$

$$\begin{array}{rcl} E\,[(\textbf{wcm } v_1\ (\textbf{wcm } v_2\ e))] & \longrightarrow & E\,[(\textbf{wcm } v_2\ e)] \\ & & E \neq E'[(\textbf{wcm } v_3\ [])] \\ E\,[(\textbf{wcm } v_1\ v_2)] & \longrightarrow & E\,[v_2] \\ & & E \neq E'[(\textbf{wcm } v_3\ [])] \\ & \ldots & \end{array}$$

Any occurence of a nested **wcm** deletes the outermost value v_1 on the frame. These evaluation rules preserve the expected behaviour of tail recursive calls, where each call follows another. Since non-tail calls must return, a **wcm** is not directly nested inside another **wcm** but rather would be embedded in another surrounding context.

```
(define (fact n)
  (letrec
    ([! (lambda (n)
          (if (= n 0)
              (begin
                (display (c-c-m))
                (newline)
                1)
              (w-c-m n
                (* n (! (- n 1))))))])
    (! n)))

> (fact 4)
(1 2 3 4) ; displayed output
24
```

```
(define (fact-tr n)
  (letrec
    ([! (lambda (n a)
          (if (= n 0)
              (begin
                (display (c-c-m))
                (newline)
                a)
              (w-c-m n
                (! (- n 1) (* a n)))))])
    (! n 1)))

> (fact 4)
(1) ; displayed output
24
```

Figure 1. The example used by Clements et al. [3] to demonstrate how continuation marks behave with tail calls.

As an example, consider the two factorial functions using continuation marks in Figure 1, taken from the paper by Clements et al. [3]. One factorial is non-tail-recursive and the other is tail-recursive. The functions use **wcm** to place the current value of the argument n on the program frame, and when the functions return, they display the current continuation marks, concatenated as a list. In the non-tail recursive *fact*, we see that all values of the argument to fact are printed, while only the last argument of *fact-tr* is printed, indicating that the previous frames of the tail-recursive calls were overwritten.

In Racket and newer versions of Scheme, continuation marks instead associate a *key* with each value on the dynamic program frame and use an explicit **call/cm** expression rather than **wcm**, which is used internally for evaluation. A dynamic value on the program frame is initialized and updated through its associated key using a **call/cm** expression [5, 7]. A **call/cm** expression includes a key, a value, and a thunk to be evaluated. Additionally, multiple keys and values can be associated dynamically into a list w. The evaluation rules of continuation marks with keys are as follows, where the value of a key can be updated within a **call/cm** context, but **wcm** still cannot be directly nested. When additional keys are added, the key value pair are added to the list of keys and values on the nearest **wcm** frame.

$$E\,[(\textbf{wcm}\ w\ v)] \to E\,[v]$$

$$E\,[(\textbf{wcm}\ (\ldots(k\ v)\ldots)\ (\textbf{call/cm}\ k\ v'\ (\boldsymbol{\lambda}\ ()\ e)))] \to E\,[(\textbf{wcm}\ (\ldots(k\ v')\ldots)\ e)]$$

$$E\,[(\textbf{wcm}\ w\ (\textbf{call/cm}\ k'\ v'\ (\boldsymbol{\lambda}\ ()\ e)))] \to E\,[(\textbf{wcm}\ (w\ (k'\ v'))\ e)] \quad \text{where } k' \notin w$$

$$E\,[(\textbf{call/cm}\ k\ v\ (\boldsymbol{\lambda}\ ()\ e))] \to E\,[(\textbf{wcm}\ ()\ (\textbf{call/cm}\ k\ v\ (\boldsymbol{\lambda}\ ()\ e)))] \quad \text{where } E \neq E'\,[(\textbf{wcm}\ w\ [])]$$

$\ldots$

Since continuation marks are a syntactic construct to dynamically add a value to the program frame and respect the constant space of tail calls, one question then arises: can regions can be directly translated to an implementation using continuation marks? The main property of regions we need to preserve in the translation is the lifetime of the region, i.e., we would not like to introduce any correctness issues by collecting a region before it is dead, or unnecessarily extend the lifetime of regions. We could have a major win with the constant space of tail calls, but we would also like to get the best of all worlds and still have the wins provided by the previous stack-like allocation behaviour of regions.

Let's revisit our first example of region-based memory management from Section 1, and see how the translation still has the expected allocation behaviour in a simple case.

```
letregion r1, r2, r3 in
 (letregion r4, r5 in
  letregion r6 in
   let x = (2 at r2, 3 at r6) at r4 in
   (λ y.(fst x, y) at r1) at r5 end
  end
  5 at r3) end end
```

This example uses intermediate regions $\mathsf{r}_4, \mathsf{r}_5, \mathsf{r}_6$ that are deallocated before the final value.

Our idea for representing regions as continuation marks is to use a continuation mark as a region name, and initialize the value as the empty store. The initializaion matches the initialization of a **letregion** construct. For brevity in this example, we directly translate each **letregion** into a

wcm construct instead of a **call/cm** with a thunk. We represent the empty store as · (center dot). To avoid mixing syntax, we rewrite the ML pairs to **cons** pairs, and use the traditional Scheme s-expression syntax.

```
(wcm ((r1 ·)(r2 ·)(r3 ·))
  (wcm ((r4 ·)(r5 ·))
    ((wcm (r6 ·)
       (let ([x (at (cons  (at 2 r2) (at 3 r6)) r4)])
         (at (λ (y) (at (cons (fst x) y) r1)) r5)))
     (at 5 r3))))
```

While **wcm** seems to match the lexical structure, does the **wcm** actually semantically preserve the lifetimes of the regions, as **wcm** behaves differently in a dynamic context?

Looking at a sketch of the full evaluation of both programs, which we present in Figure 2, we find **call/cm** preserves the lifetimes of regions. For brevity, we retain the (**at** e r) syntax, but one can envision (for now) that each (**at** e r) is an update to the mutable store associated with each key. The region annotated program evaluation is displayed on the left, while the continuation marked program evaluation is displayed on the right. The evaluation of both programs proceeds downwards for space.

We focus on what regions are currently allocated and deallocated at which program evaluation points, and for now ignore what values are allocated inside each region. Stepping through the evaluation, we see the regions on the left and continuation marks on the right are allocated and deallocated in lock-step. Regions are live for exactly as long as continuation marks are live. Let's make this connection formal, and study what it means for the semantics of regions and tail calls.

3 Translation and Dynamic Semantics

In this section, we present a translation from a typed, explicit region annotated standard ML like language to a Scheme-like, untyped language with continuation marks. The main idea is to translate each region into a **call/cm** with an empty store. To update the store (allocate into a region), we introduce a new (**at** e ρ) syntax and evaluation for brevity, and to greatly simplify the translation and updates to the values (stores) associated with keys.

The translation is shown in Figure 3. We present only the rules for region introduction, allocation, recursively polymorphic function introduction and application, as the rest is standard, *e.g.,* addition in ML translates to Scheme's addition, etc. Region introduction is translated to **call/cm** with the region ρ becoming the continuation mark key, and the value being the empty store · (center dot). A region polymorphic function is converted to an untyped Scheme function, where we create a closure expecting first the region names (now continuation mark keys) and then the formal parameters. And finally, a region polymorphic function application turns into a usual function application nested within an **at** expression, as the syntax of region polymorphic application is always expected to be allocated at a particular region.

Of note in this translation is the fact that allocation into a region is translated to a new syntax (**at** e ρ) that does not exist in Scheme or Racket. Nor is there evaluation rules for a region location, represented as ℓ_ρ. We show special evaluation rules for **at** and locations in Figure 4. We could instead translate allocation using **call/cm** as follows.

$$E[e \textbf{ at } \rho] \rightarrow \begin{array}{l}(\textbf{let } ([\ell \ (\texttt{fresh})]) \\ \quad (\textbf{call/cm } \rho \ (\texttt{update } \rho \ \ell \ e) \ (\lambda \ () \ E[\ell])))\end{array}$$

The `update` then uses `current-marks` with the key ρ to obtain all the current values (locations allocated) and adds another, which is then used in the context E.

However, to avoid cluttering the translation, we simply present additional evaluation rules for **at** inside within a **wcm** expression. Recall that the **wcm** expression is the internal representation of evaluation: the translation only produces **call/cm** expressions. The evaluation rule for **at** introduces a fresh location ℓ, and updates the nearest current store value for ρ (indicated by no other surrounding **wcm** context). We annotate the location ℓ with the region ρ for brevity, rather than representing addresses as a pair of a region name and an offset (r, o) as is done in the region version of ML. All other evaluation rules are standard for **wcm** and **call/cm** given by Flatt et al. [5].

Next, we work through two examples to ensure the behaviour of regions-as-continuation-marks work correctly in both tail calls and non-tail calls.

3.1 Non-tail Calls

The first example is the non-tail recursive factorial, shown in Figure 5, where the region annotated program is on the left, omitting the details for the initialization of regions for the call. The corresponding translated version of non-tail recursive factorial to continuation marks is on the right. We omit the initialization of all the regions to the empty store, and use **wcm** for brevity instead of a **call/cm** expression with a thunk.

We sketch the evaluation of both in Figure 6. Evaluation proceeds the same as the region annotated version: first, the continuation mark r_1 is initialized to the empty store, the mark is then updated to contain the store with closure of `fact`. Then, when calling `fact`, we initialize the first two arguments with the keys representing the marks initialized for the call of factorial of 5. Evaluation then proceeds to introduce the mark r_4 to store the value of the recursive factorial call, then marks r_5, r_6 to store the values of the argument 4 to factorial and factorial's closure. The temporary mark r_7 is deallocated after each subtraction. Continuation marks continue to stack until factorial returns, as they are interspersed between different contexts that expect to return. The marks are then deallocated in a stack-based manner, just like regions.

```
letregion r1, r2, r3 in
  letregion r4, r5 in
    (letregion r6 in
      let x = (2 at r2, 3 at r6) at r4 in
      (λ y.(fst x, y) at r1) at r5 end end
    5 at r3) end end
⇓
{r1, r2, r3} ⊢  letregion r4, r5 in
                  (letregion r6 in
                    let x = (2 at r2, 3 at r6) at r4 in
                    (λ y.(fst x, y) at r1) at r5 end
                  end
                  5 at r3) end
⇓
{r1—r5} ⊢  letregion r6 in
             let x = (2 at r2, 3 at r6) at r4 in
             (λ y.(fst x, y) at r1) at r5 end
           end
⇓
{r1—r5, r6} ⊢  let x = (2 at r2, 3 at r6) at r4 in
               (λ y.(fst x, y) at r1) at r5 end
⇓
...
{r1—r5, r6} ⊢ (ℓ (5 at r3))
⇓
r6 is deallocated after closure is allocated in r5 at ℓ
{r1—r5} ⊢ (ℓ (5 at r3))
⇓
r4, r5 are deallocated after the function application
results in a value
→ {r1, r2, r3} ⊢ ℓp
r1, r2, r3 are deallocated after accessing the pair at ℓp
```

```
[] [ (wcm ((r1 ·)(r2 ·)(r3 ·))
     (wcm ((r4 ·)(r5 ·))
       ((wcm (r6 ·)
          (let ([x (at (cons  (at 2 r2) (at 3 r6)) r4)])
            (at (λ (y) (at (cons (fst x) y) r1)) r5)))
        (at 5 r3)))) ]
⇓
(wcm r1, r2, r3 []) [ (wcm ((r4 ·)(r5 ·))
                        ((wcm (r6 ·)
                           (let ([x (at (cons  (at 2 r2) (at 3 r6)) r4)])
                             (at (λ (y) (at (cons (fst x) y) r1)) r5)))
                         (at 5 r3)))) ]
⇓
(wcm r1—r5 []) [ ((wcm (r6 ·)
                    (let ([x (at (cons  (at 2 r2) (at 3 r6)) r4)])
                      (at (λ (y) (at (cons (fst x) y) r1)) r5)))
                  (at 5 r3)) ]
⇓
(wcm r1-r5 ([] ...)) [ (wcm (r6 ·)
                         (let ([x (at (cons  (at 2 r2) (at 3 r6)) r4)])
                           (at (λ (y) (at (cons (fst x) y) r1)) r5))) ]
⇓
(wcm r1-r5 ((wcm r6 []) ...)) [ (let ([x (at (cons  (at 2 r2) (at 3 r6)) r4)])
                                  (at (λ (y) (at (cons (fst x) y) r1)) r5))) ]
⇓
...
(wcm r1—r5 ((wcm r6 []) ...)) [ℓ]
⇓
The closure is allocated in r5 at ℓ, and is a value,
so wcm with r6 is deleted (deallocated).
(wcm r1—r5 (ℓ [])) [(at 5 r3)]
⇓
...
→ (wcm r1—r5 []) [ℓp]
Then deallocate r1, r2, r3, r4, r5 after accessing the final value at ℓp
```

Figure 2. Evaluation sketch of both regions and the translation to continuation marks. The region evaluation is on the left, the continuation mark version is on the right. Evaluation proceeds downwards.

$\boxed{\mathbf{W}[\![e]\!] = e}$

$$
\begin{array}{rcl}
 & \cdots & \\
\mathbf{W}[\![\textbf{letregion } \rho \textbf{ in } e \textbf{ end}]\!] & = & (\textbf{call/cm } \rho \cdot (\boldsymbol{\lambda} () \mathbf{W}[\![e]\!])) \\
\mathbf{W}[\![e \textbf{ at } \rho]\!] & = & (\textbf{at } \mathbf{W}[\![e]\!] \ \rho) \\
\mathbf{W}[\![\textbf{letrec } f[\overline{\rho}](\overline{x}) \textbf{ at } \rho' = e_1 \textbf{ in } e_2 \textbf{ end}]\!] & = & (\textbf{letrec } ([f \ (\textbf{at } (\boldsymbol{\lambda} \ (\overline{\rho}) \ (\boldsymbol{\lambda} \ (\overline{x}) \ \mathbf{W}[\![e_1]\!])) \ \rho')]) \ \mathbf{W}[\![e_2]\!]) \\
\mathbf{W}[\![f[\overline{\rho}] \textbf{ at } \rho']\!] & = & (\textbf{at } (f \ \overline{\rho}) \ \rho)
\end{array}
$$

Figure 3. The translation from ML-style regions to Scheme-style continuation marks.

3.2 Tail Calls

The second example is tail recursive factorial, presented in Figure 7 and features the region annotated tail recursive factorial from Section 1. The corresponding program with continuation marks is on the left. We again omit the details for the initialization of the regions for the inital call to factorial, and use **wcm** for brevity instead of a **call/cm** expression with a thunk.

We have now encountered the issue with the interpretation of allocation into a region, the **at** expression, combined with region polymorphic functions. Notice that despite factorial being tail recursive, the dynamic regions storing the

Expression	e	::=	$\ldots \mid (\textbf{call/cm}\ \rho\ s\ (\boldsymbol{\lambda}\ ()\ e))$
Value	v	::=	$\ldots \mid \ell_\rho$
Store	s	::=	$\cdot \mid s[l \mapsto v]$
Mark Mappings	w	::=	$((\rho\ s)\ldots)$
Continuation Mark Context	W	::=	$E \mid (\textbf{wcm}\ w\ E)$
Evaluation Context	E	::=	$[] \mid (W\ \overline{e}) \mid (v\ W\ \overline{e}) \mid (v\ \overline{v}\ W) \mid (\textbf{letrec}\ ([x\ W])\ e) \mid (\textbf{if}\ W\ e\ e) \mid (\textbf{at}\ W\ \rho)$

$\boxed{E \rightarrow E}$

$$
\begin{array}{rcl}
E\,[(\textbf{wcm}\ w\ v)] & \rightarrow & E\,[v] \\
E\,[(\textbf{wcm}\ (\ldots(\rho\ s)\ldots)\ (\textbf{call/cm}\ \rho\ s'\ (\boldsymbol{\lambda}\ ()\ e)))] & \rightarrow & E\,[(\textbf{wcm}\ (\ldots(\rho\ s')\ldots)\ e)] \\
E\,[(\textbf{wcm}\ w\ (\textbf{call/cm}\ \rho'\ s'\ (\boldsymbol{\lambda}\ ()\ e)))] & \rightarrow & E\,[(\textbf{wcm}\ (w\ (\rho'\ s'))\ e)] \\
 & & \text{where } \rho' \notin w \\
E\,[(\textbf{wcm}\ (\ldots(\rho\ s)\ldots)\ E'\,[(\textbf{at}\ v\ \rho)])] & \rightarrow & E\left[(\textbf{wcm}\ (\rho\ s[\ell \mapsto v])\ E'\left[\ell_\rho\right])\right] \\
 & & \text{where } E' \neq E''\,[(\textbf{wcm}\ w\ [])] \wedge \ell \text{ fresh in } (\rho\ s) \\
E\left[(\textbf{wcm}\ (\rho\ s)\ (E'\left[\ell_\rho\right]))\right] & \rightarrow & E\,[(\textbf{wcm}\ (\rho\ s)\ (E'\,[v]))] \\
 & & \text{where } E' \neq E''\,[(\textbf{wcm}\ w\ [])] \wedge \ell \mapsto v \in s \\
E\,[(\textbf{call/cm}\ \rho\ s\ v)] & \rightarrow & E\,[(\textbf{wcm}\ ()\ (\textbf{call/cm}\ \rho\ s\ v))] \\
 & & \text{where } E \neq E'\,[(\textbf{wcm}\ w\ [])]
\end{array}
$$

Figure 4. Continuation marks dynamic semantics with allocation into regions.

```
letregion r1 in
 letrec fact at r1 =
  λ [r2, r3] (n). if n < 1
                  then 1 at r3
                  else letregion r4 in
                         (letregion r5, r6 in
                         (fact[r5, r4] at r6
                          letregion r7 in
                          (n + (−1 at r7)) at r5
                          end )
                         ∗ n) at r3
                        end
 in ...(fact[r2, r3] 5)... end end
```

```
(wcm r1
 (letrec [(fact (at  (λ (r2 r3) (λ (n)
                      (if (< n 1)
                        (at 1 r3)
                        (wcm r4
                         (at (∗ (at (wcm r5, r6
                                  ((at (fact r5 r4) r6)
                                   (wcm r7
                                     (at (+ n (at − 1 r7))
                                         r5)))) r4)
                                  n) r3)))))
                    r1))]
 ...((fact r2 r3) 5)...))
```

Figure 5. Non-tail recursive factorial with regions and the continuation mark version on the right.

arguments and the closures keep growing with each subsequent tail recursive call. As a result, the tail recursive function does not take constant space. Since we have taken the evaluation with a stack and explicited the stack as regions passed to the function, we are essentially forced to treat the stack itself as arguments that must be saved across calls. It becomes difficult to recognize the connection between the explicated stack and the arguments that can be reset.

With continuation marks, since each time we recursively call the function, we introduce a new **call/cm** expression with an empty store, each previous mark with a value is destroyed, at least in the closure case. Unfortunately though, the stores associated with keys $\mathtt{r}_2, \mathtt{r}_3, \mathtt{r}_4$ are updated with each argument pair. Because the **at** expression introduces a fresh location at each allocation, the previous allocation of the arguments in $\mathtt{r}_2, \mathtt{r}_3, \mathtt{r}_4$ are not erased.

Instead we could re-introduce the regions $\mathtt{r}_2, \mathtt{r}_3, \mathtt{r}_4$ for the arguments on each tail call, replacing the previous argument with the next argument at each tail call. Then, in the tail-recursive case, we would have the expected stack of **wcm** such that repeated allocations of the arguments are overwritten. For example, in the body of `fact-tr`, we might have the following expression instead:

```
(wcm r2, r3, r4, r5,
((at (fact-tr r2, r3, r4) r5)
      (at (cons (wcm r6
        (at (− n (at 1 r6)) r3))
        (at (∗ n a) r4)) r2)))))))))
```

```
letregion r1 in
 letrec fact at r1 =
  λ [r2, r3] (n). if n < 1
                  then 1 at r3
                  else letregion r4 in
                         (letregion r5, r6 in
                         (fact[r5, r4] at r6
                          letregion r7 in
                          (n + (−1 at r7)) at r5
                          end )
                         ∗ n) at r3
                        end
  in ... (fact[r2, r3] 5) ... end end
⇓
r1, r2, r3 ⊢ letregion r4 in
               (letregion r5, r6 in
               (fact[r5, r4] at r6
                letregion r7 in
                (5 + (−1 at r7)) at r5
                end )
               ∗ 5) at r3
              end
⇓
r1−r6 ⊢ letregion r7 in
         (5 + (−1 at r7)) at r5
         end
⇓
r1−r6, r7 ⊢ (5 + (−1 at r7)) at r5
⇓
r7 is deallocated after subtraction from 5
⇓
r1−r6 ⊢ ((fact[r5, r4] at r6) ℓval4)
⇓
regions pile up until return and popped
like r5, r6 on return for fact of 4
⇓
r1−r4 ⊢ (ℓfact4 ∗ 5)
⇓
r4 is deallocated after the return and multiplication
⟶ {r1, r2, r3} ⊢ ℓfact5
r1, r2, r3 are deallocated after final value
```

```
(wcm r1
 (letrec [(fact (at (λ (r2 r3) (λ (n)
                     (if (< n 1)
                       (at 1 r3)
                       (wcm r4
                        (at (∗ (at (wcm r5, r6
                                 ((at (fact r5 r4) r6)
                                  (wcm r7
                                   (at (+ n (at − 1 r7))
                                       r5)))) r4)
                                 n) r3)))))
                    r1))]
  ... ((fact r2 r3) 5) ...))
⇓
(wcm r1−r3 [...]) [ (wcm r4
                     (at (∗ (at (wcm r5, r6
                              ((at (fact r5 r4) r6)
                               (wcm r7
                                (at (+ 5 (at − 1 r7))
                                    r5)))) r4)
                              5) r3)) ]
⇓
(wcm r1−r6 [...]) [ (wcm r7
                     (at (+ 5 (at − 1 r7))
                         r5)) ]
⇓
(wcm r1−r6 ... (wcm r7 [])) [(at (+ 5 (at − 1 r7)) r5)]
⇓
r7 is deallocated after subtraction from 5
⇓
(wcm r1−r6 (∗ [] 5)) [((at (fact r5 r4) r6) ℓval4)]
⇓
marks pile up until return and popped
like r5, r6 on return for fact of 4
⇓
(wcm r1... (wcm r5, r6 ... [])) [(∗ ℓfact4 5)]
⇓
r4 is deallocated after the return and multiplication
⟶ (wcm r1... (wcm r4 [])) [ℓfact5]
r1, r2, r3 are deallocated after final value
```

Figure 6. Evaluation sketch of both regions and the translation to continuation marks for non-tail recursive factorial. The region evaluation is on the left, the continuation mark version is on the right. Evaluation proceeds downwards.

Such an expression would redefine the value of the keys r_2, r_3, r_4 to the empty store at each tail-recursive call, and so each argument is only allocated once until the store is redefined again. The final value would be stored in r_2, without any previous pair allocated, ensuring tail recursive functions take constant space again. This proposed alternative translation however needs more study to see how such a translation intereferes with the expected behaviour of region-polymorphic functions and whether may likely cause too many allocations in non-tail recursive functions.

4 Future Directions

Our vision for what this connection may hold for the future are two-fold: benefiting both languages with region-based memory systems and Scheme-like languages with delimited control.

```
letregion r1 in
 letrec fact-tr at r1 =
  λ [r2, r3, r4] (p). let n = fst p, a = snd p in
                      if n < 1
                       then p
                       else    letregion r5 in
                               fact-tr[r2, r3, r4] at r5
                                (letregion r6 in
                                (n − (1 at r6)) at r3
                                end , n ∗ a at r4) at r2
                              end
 in ... (fact[r2, r3, r4] (5, 1)) end end
```

```
(wcm r1
 (letrec [(fact-tr (at  (λ (r2 r3 r4) (λ (p)
                         (let ([n (car p)][a (cdr p)])
                          (if (< n 1)
                            p
                            (wcm r5
                            ((at (fact-tr r2, r3, r4) r5)
                                  (at (cons (wcm r6
                                     (at (− n (at 1 r6)) r3))
                                     (at (∗ n a) r4)) r2)))))))))
                    r1))]
 ... ((fact-tr r2 r3 r4) (cons 5 1)) ...))
```

Figure 7. Tail recursive factorial with regions and the continuation mark version on the right.

While other languages have found ways to enable efficient non-lexical or non-stack-based regions, such as ReML [4], Spegion [6], Rust, and Verona [1], we still believe the connection to be beneficial for region-based languages. Such a connection can elucidate the formal semantics of languages with regions, especially with respect to tail calls. Another potential avenue is to be able to integrate more constructs for delimited control in these languages with region-based memory management, and with this connection to continuation marks, the semantics of such a combination can become more clear.

For Scheme and Racket like languages who already enjoy delimited control constructs, we now can perhaps integrate some ideas from region-based memory management into their implementations. Some implementations, such as the Chez Scheme compiler, have already achieved great success in efficient implementation, but perhaps could benefit from a new view on memory management. Alternatively, there may be a desire to integrate regions at the user-level, perhaps to allow broader control over memory regions rather than individual boxes in memory.

Our connection between regions and continuation marks really goes to show how wonderful many functional languages are!

letregion r_1 **in**
letrec fact-tr **at** r_1 =
λ [r_2, r_3, r_4] (p). **let** n = **fst** p, a = **snd** p **in**
if n < 1
then p
else **letregion** r_5 **in**
fact-tr[r_2, r_3, r_4] **at** r_5
(**letregion** r_6 **in**
(n − (1 **at** r_6)) **at** r_3
end , n ∗ a **at** r_4) **at** r_2
end
in ... (fact[r_2, r_3, r_4] (5, 1)) **end end**

⇓

r_1–r_4 ⊢ **letregion** r_5 **in**
fact-tr[r_2, r_3, r_4] **at** r_5
(**letregion** r_6 **in**
(n − (1 **at** r_6)) **at** r_3
end , n ∗ a **at** r_4) **at** r_2

⇓

r_1–r_4, r_5 ⊢ **letregion** r_6 **in**
(5 − (1 **at** r_6)) **at** r_3
end

⇓

r_1–r_5, r_6 ⊢ (5 − (1 **at** r_6)) **at** r_3

⇓

r_6 is deallocated after subtraction from 5

⇓

r_1–r_5 ⊢ ((**at** fact[r_2, r_3, r_4] r_5) (**at** (**cons** ℓ_{val4} ℓ_a) r_2))

⇓

the regions r_2–r_4 grows with pairs and
the region r_5 grows with closures c_1, ... in the
number of tail calls, until recursion returns

⇓

$r_1, r_2 = \{(4, 5), (3, 20), ...\}, ..., r_5 = \{c_1, c_2, ...\}$ ⊢ (1, 120)
and only now we can deallocate the pairs

(**wcm** r_1
(**letrec** [(fact-tr (**at** (λ (r_2 r_3 r_4) (λ (p)
(**let** ([n (**car** p)][a (**cdr** p)])
(**if** (< n 1)
p
(**wcm** r_5
((**at** (fact-tr r_2, r_3, r_4) r_5)
(**at** (cons (**wcm** r_6
(**at** (− n (**at** 1 r_6)) r_3))
(**at** (∗ n a) r_4)) r_2)))))))))
r_1))]
... ((fact-tr r_2 r_3 r_4) (**cons** 5 1)) ...))

⇓

(**wcm** r_1–r_4 [...]) $\left[\begin{array}{l}\text{(wcm } r_5 \\ \text{((at (fact-tr } r_2, r_3, r_4\text{) } r_5\text{)} \\ \quad \text{(at (cons (wcm } r_6 \\ \quad\quad \text{(at (− n (at 1 } r_6\text{)) } r_3\text{)) } \\ \quad\quad \text{(at (∗ n a) } r_4\text{)) } r_2\text{)))}\end{array}\right]$

⇓

(**wcm** r_1–r_4, r_5 [...]) $\left[\begin{array}{l}\text{(wcm } r_6 \\ \quad \text{(at (− n (at 1 } r_6\text{)) } r_3\text{))}\end{array}\right]$

⇓

(**wcm** r_1–r_4, r_5 ... (**wcm** r_6 [])) [(**at** (− n (**at** 1 r_6)) r_3)]

⇓

r_6 is deallocated after subtraction from 5

⇓

(**wcm** r_1–r_5 ... []) $\left[\begin{array}{l}\text{((at (fact-tr } r_2, r_3, r_4\text{) } r_5\text{)} \\ \quad \text{(at (cons } \ell_{\text{val4}}\ \ell_a\text{) } r_2\text{))}\end{array}\right]$

⇓

at next recursive call, we reinitialize r_5 *with the empty store!*
which allows us to take up constant space with closures
however, since r_2–r_4 are passed into the function as
arguments, each pair of arguments are allocated in the stores
instead of reinitializing the marks to empty stores

⇓

(**wcm** $r_1, r_2 = \{(4, 5), (3, 20), ...\}, ..., r_5 = \{c_4\}$ ... []) [(**cons** 1 120)]

Figure 8. Evaluation sketch of both regions and the translation to continuation marks for tail recursive factorial. The region evaluation is on the left, the continuation mark version is on the right. Evaluation proceeds downwards.

Acknowledgments

We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC), funding reference number RGPIN-2019-04207. Cette recherche a été financée par le Conseil de recherches en sciences naturelles et en génie du Canada (CRSNG), numéro de référence RGPIN-2019-04207.

References

[1] Ellen Arvidsson, Elias Castegren, Sylvan Clebsch, Sophia Drossopoulou, James Noble, Matthew J. Parkinson, and Tobias Wrigstad. 2023. Reference Capabilities for Flexible Memory Management. *Proceedings of the ACM on Programming Languages (PACMPL)* Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) (2023). doi:10.1145/3622846

[2] Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. 1996. From region inference to von Neumann machines via region representation inference. In *Symposium on Principles of Programming Languages (POPL)*. doi:10.1145/237721.237771

[3] John Clements, Matthew Flatt, and Matthias Felleisen. 2001. Modeling an Algebraic Stepper. In *European Symposium on Programming (ESOP)*. doi:10.1007/3-540-45309-1_21

[4] Martin Elsman. 2024. Explicit Effects and Effect Constraints in ReML. *Proceedings of the ACM on Programming Languages (PACMPL)* Symposium on Principles of Programming Languages (POPL) (2024). doi:10.1145/3632921

[5] Matthew Flatt, Gang Yu, Robert Bruce Findler, and Matthias Felleisen. 2007. Adding delimited and composable control to a production programming environment. In *International Conference on Functional Programming (ICFP)*. doi:10.1145/1291151.1291178

[6] Jack Hughes, Michael Vollmer, and Mark Batty. 2025. Spegion: Implicit and Non-Lexical Regions with Sized Allocations. In *European Conferene on Object-Oriented Programming (ECOOP)*. doi:10.4230/LIPICS.ECOOP.2025.15

[7] The Racket Language Reference. 2026. Continuation Marks. Accessed on 2026-06-01. https://docs.racket-lang.org/reference/contmarks.html

[8] Mads Tofte and Jean-Pierre Talpin. 1994. Implementation of the typed call-by-value lambda-calculus using a stack of regions. In *Symposium on Principles of Programming Languages (POPL)*. doi:10.1145/174675.177855

[9] Mads Tofte and Jean-Pierre Talpin. 1997. Region-Based Memory Management. *Information and Computation* (1997). doi:10.1006/inco.1996.2613

Received 2026-06-05; accepted 2026-07-01