Recycling Continuations

Jonathan Sobel Daniel P. Friedman*

Computer Science Department
Indiana University
Bloomington, Indiana 47405
{jsobel, dfried}@cs.indiana.edu

Abstract

If the continuations in functional data-structure-generating programs are made explicit and represented as records, they can be "recycled." Once they have served their purpose as temporary, intermediate structures for managing program control, the space they occupy can be reused for the structures that the programs produce as their output. To effect this immediate memory reclamation, we use a sequence of correctness-preserving program transformations, demonstrated through a series of simple examples. We then apply the transformations to general anamorphism operators, with the important consequence that all finite-output anamorphisms can now be run without any stack- or continuation-space overhead.

1 Introduction

The runtime architecture for a language implementation keeps track of the continuations of procedure calls using either a stack of call frames or a linked chain of heap-allocated continuation structures. One advantage that may be claimed for the stack approach is that deallocation of frames is inexpensive, and it happens as soon as a procedure returns. Heap-allocated continuations, on the other hand, typically take up memory until the space is reclaimed by the runtime memory manager (e.g., a garbage collector). We show how, for a certain class of procedures, using the continuation-based approach can lead not only to immediate reclamation of the space used by the continuations, but also to the elimination of most of the memory overhead incurred by both of the aforementioned architectures.

Link-inversion algorithms (more generally, Deutsch-Schorr-Waite algorithms) are a standard means of traversing data structures with polynomial (sum-of-products) types using little or no memory overhead. The problem with such algorithms is that they mutate the structures during the traversal. Thus, link inversion is generally unsafe, except in "critical sections," during garbage collections (the context of the original published description [14]), or over uniquely referenced objects. We demonstrate a series of correctness-preserving program transformations that takes a functional structure-generating procedure (including anamorphisms [8, 9]) and produces one that uses a safe variation of link inversion to reduce or eliminate the memory overhead of recursive procedure calls.

Some of our algorithms for traversing and constructing data structures are variations on techniques first published twenty or thirty years ago; some special cases might have been in use since the 1950s. They are now standard fare in undergraduate data structures texts [6, 15]. The contribution here is to *derive* these algorithms through a sequence of correctness-preserving program transformations. This exercise may seem at first to be more entertaining than useful, but there are two real practical benefits to the transformation-based approach. First, a programmer using this technique can begin with a formal mathematical algorithm that parallels the recursive structure of the data and then tune the program for efficiency. Second, the transformations could be automated, allowing them to be included in a compiler or source-to-source translator/optimizer.

We begin with a simple, concrete procedure definition and follow it through a sequence of transformations. The first example is a procedure that copies a list. After working through this example, we consider a procedure that copies binary trees. At the end, we generalize from the examples.

2 History

How to implement recursion (and, more generally, procedure calls) efficiently has been a subject of great scrutiny in programming language design and compiler construction. Early on, the stack-based runtime architecture was developed, with call frames being pushed onto the stack for each procedure call. Stack frames take up memory, though, and programmers have always been on the lookout for ways to reduce memory consumption. One technique for traversing trees and other inductively defined structures, while using very little memory overhead, is to invert the pointers on the way "down" into the structure, leaving a trail to follow on the way back out. As each recursive call returns, instead of popping a stack frame, it re-inverts a pointer back to its original orientation. During such a traversal, however, if another procedure needed to start from the "top" of the structure, it would not be able to perform a correct traversal until all the pointers were returned to their original states. Not surprisingly, then, the first publication of such a technique [14] specified a context in which it is not only essential to use

*This work was supported in part by the National Science Foundation under grant CCR-9633109.

Appeared at ICFP '98 (the International Conference on Functional Programming) in September, 1998 in Baltimore, Maryland.

little or no memory, but also safe to assume (at the time) that no other traversal could happen concurrently: garbage collection. Schorr and Waite's algorithm was developed at the same time that Deutsch developed a similar one independently; thus, link-inversion algorithms are now known collectively as Deutsch-Schorr-Waite algorithms.[1]

The idea that programs could be transformed so that no return information needed to be saved for procedure calls is first recorded in a talk by Van Wijngaarden and a transcript of the ensuing discussion [18]. Using explicit continuations to represent computational contexts moved center-stage beginning with Reynolds's seminal paper [13]. As implementors began to explore the possibility of replacing the stack-based architecture with a continuation-based one, a variety of trade-offs were exposed.

The trade-off of immediate interest is between the simplicity of a procedure call in continuation-based systems and the simplicity of space-reclamation upon procedure return in stack-based systems. When a program has been written in (or transformed into) continuation-passing style (CPS), every call is a tail call and may be implemented as a simple jump (or `goto`), but the information about what still needs to be done after the call "returns" is encapsulated into a structure that is usually heap-allocated. Barring the user-level reification of continuations, each continuation could be disposed of after it has been used (thus, the stack architecture), but if continuations are treated just like any other heap structures, they continue to occupy space until the run-time memory manager recognizes that they are unreachable. In a directly recursive, stack-based style, each call requires some extra work in the creation and setup of a new stack frame, but returning from a call is usually a simple matter of adjusting the value of a single register. The memory used by the frame is instantly available for reuse.

In an attempt to eliminate recursion overhead altogether, Minamide demonstrates a new technique for representing "unfinished" data structures [10]. Using this technique, programs that would otherwise have required the growth of a stack or the construction of a chain of continuations can instead run as simple loops. The style in which the programs must be written in order to benefit from Minamide's representation is closely related to CPS, as the author mentions. This relationship led us to question more deeply the connection between the continuations that appear in certain programs and the data structures that the programs construct. What follows is the result of that inquiry.

3 Lists

The code in the following examples is in Scheme, but in order to maintain continuity between the list and tree examples, the use of built-in Scheme lists (e.g., `car`, `cdr`, `cons`) will be avoided. Instead, we explicitly define a list datatype:

⟨*list datatype*⟩≡
```
(datatype List
  (Empty-List)
  (Pair head tail))
```
The `datatype` construct is not a standard Scheme form.[2] It is intended to feel similar to the homonymous form in ML. In short, defining a datatype provides us with constructors for the variants, as well as a "case" form for de-structuring objects of the datatype. Thus, the procedure for copying lists is:

⟨*list copy*⟩≡
```
(define list-copy
  (lambda (l)
    (List-case l
      ((Empty-List) (Empty-List))
      ((Pair h t) (Pair h (list-copy t))))))
```
Using `list-copy` as an example may seem a bit odd, since it is functionally equivalent to the identity function. The intent of these examples is to convey the nature of the program transformations, though, and `list-copy` is among the simplest of procedures that construct lists. Procedures that actually perform interesting work, such as `subst`, `remove`, or `iota` (which produces an increasing list of numbers), are straightforward generalizations of `list-copy`, as long as their non-varying parameters are abstracted out of the recursion.

3.1 (Lists) Continuation-Passing Style

The first transformation to be applied to `list-copy` is the standard call-by-value conversion to continuation-passing style (CPS). All the "administrative" CPS reductions have been left out, and constructors have been treated as primitive, atomic operations.

⟨*list copy in CPS*⟩≡
```
(define list-copy-cps
  (lambda (l k)
    (List-case l
      ((Empty-List) (k (Empty-List)))
      ((Pair h t) (list-copy-cps t
                    (lambda (v)
                      (k (Pair h v))))))))
```
The original one-argument version of `list-copy` can be recovered with a simple "wrapper" for `list-copy-cps`.

⟨*list copy in CPS*⟩+≡
```
(define list-copy
  (lambda (l)
    (list-copy-cps l (lambda (v) v))))
```
The two versions of `list-copy`—direct and continuation-passing style—compute the same result, but the work done along the way is slightly different. In the direct-style version, the language implementation maintains the control context. In the CPS version, every call is a tail call (equivalent to a jump or `goto`); as far as the compiler can tell, there is nothing left to do after copying the tail. Instead, the continuation is maintained explicitly in the program.

As `list-copy` "walks down" the list, it constructs a sequence of nested closures. With the display-closure representation [5, 2, 3] in mind (in which a closure is a sequence of memory locations containing a code pointer and the values of the procedure's free variables), it becomes evident that the continuation is really a linear, linked structure. Upon reaching the end of the list, the procedure starts a chain reaction by invoking the outermost (most recently linked) of these closures. The first one constructs a new pair and then invokes the next one, each constructing another pair until the identity (empty, initial) continuation is reached, and the copied list is returned.

[1] Schorr and Waite used link inversion during the "mark" phase of a mark-sweep collector, so they had to chase arbitrary pointers through a graph structure. Our focus is on functional, recursive structures, not arbitrary graphs.

[2] A definition for (and explication of) the `datatype` macro appears at `http://www.cs.indiana.edu/~jsobel/`.

3.2 (Lists) Representation Independence

The explicit treatment of the continuations in the CPS version of `list-copy` is, in fact, a bit *too* explicit: the continuations-as-closures representation was "hard-wired" into the code. The next step is to make `list-copy` treat the continuation more abstractly [13], so that the representation of the continuation can be modified at will. The procedures will retain their prior form:

⟨*list copy in representation-independent CPS*⟩≡

```
(define list-copy-cps
  (lambda (l k)
    (List-case l
      ((Empty-List) (invoke_L k (Empty-List)))
      ((Pair h t) (list-copy-cps t
                    ⟨construct pairing continuation⟩)))))

(define list-copy
  (lambda (l)
    (list-copy-cps l ⟨construct initial continuation⟩)))
```

Here the continuations will be constructed and invoked through a more abstract interface. For invocation, the $\texttt{invoke}_L$ operator suffices to eliminate the assumption that continuations are procedures. If, in spite of the more abstract interface, continuations really are still represented as procedures, $\texttt{invoke}_L$ can be defined as

⟨*invoking list continuations (procedure representation)*⟩≡

```
(define invoke_L
  (lambda (k v)
    (k v)))
```

Having committed to a procedural representation, the continuation constructors had better produce the same closures as in the original CPS version (with the bodies transformed where necessary). Of course, this implies that the values of the free variables must be passed as arguments to the constructors.

⟨*list continuation constructors (procedure representation)*⟩≡

```
(define Initial-k
  (lambda ()
    (lambda (v) v)))

(define Pairing-k
  (lambda (h k)
    (lambda (v)
      (invoke_L k (Pair h v)))))
```

Thus, the missing part of `list-copy-cps` is

⟨*construct pairing continuation*⟩≡

```
(Pairing-k h k)
```

and the missing part of `list-copy` is just

⟨*construct initial continuation*⟩≡

```
(Initial-k)
```

3.3 (Lists) Representing Continuations as Records

Freed from any dependence on a particular representation, we can now switch from procedural to record-based representation of continuations without changing the definition of `list-copy`. What changes is the construction and invocation of the continuations: instead of explicitly defined constructors, a datatype is used.

⟨*list continuation type*⟩≡

```
(datatype List-k (Initial-k) (Pairing-k head k))
```

Thus, the constructors defined in the preceding section are replaced by trivial record constructors. The work is shifted to the $\texttt{invoke}_L$ procedure. In order to preserve the semantics of the invocation, we can simply copy the code from the bodies of the old continuation procedures into the different cases of the $\texttt{invoke}_L$ procedure.

⟨*invoking list continuations (record representation)*⟩≡

```
(define invoke_L
  (lambda (k v)
    (List-k-case k
      ((Initial-k) v)
      ((Pairing-k h k)
       (invoke_L k (Pair h v))))))
```

The linked structure of continuations described in Section 3.1 is now completely explicit. Each continuation has an explicit link (in the form of a record field) to the next continuation in the chain. Consider how a call to `list-copy` works now:

1. First, `list-copy-cps` walks down the list, creating a `Pairing-k` record for each `Pair` in the list.
2. At the end of the list, the base case of `list-copy-cps` calls $\texttt{invoke}_L$ with the chain of `Pairing-k`s and an `Empty-List` as arguments.
3. Then, $\texttt{invoke}_L$ constructs a new `Pair` for each `Pairing-k`, using the saved head value for the head and the value of the `v` parameter for the tail.
4. When $\texttt{invoke}_L$ reaches the initial continuation, the value of `v` (which is by now a complete copy of the original list) is returned.

Thus, three data structures are involved in a call to `list-copy`: the input list, the chain of continuations, and the output list. In addition, the property that every call is a tail call has been preserved from the original CPS transformation.

3.4 (Lists) Link Inversion

There are two crucial observations to be made about the last definition of $\texttt{invoke}_L$. First, we have "accidentally" used the name `k` as part of the pattern on the `Pairing-k` line, shadowing the `k` parameter of $\texttt{invoke}_L$. Hiding the `k` parameter did no harm, though; it was only used to choose a case and bind the pattern variables. *The continuation passed to* $\texttt{invoke}_L$ *is never used on the right-hand side of a case clause.* Since there are no free occurrences of `k` in the clauses, its value is no longer reachable (i.e., it is garbage) as soon as the pattern is matched [12]. In other terminology, $\texttt{invoke}_L$ obeys a linear type discipline, which has important implications about how the data structure bound to `k` can be treated [4].

The second observation is that the `Pair` constructed in the `Pairing-k` clause is similar in form and contents to the `Pairing-k` itself. Both have two fields, and both have the same value in their first fields. The only difference is that the `k` in the `Pairing-k` is replaced with `v` in the `Pair`.

Taking advantage of the linearity of `k`, why not take the "replacement" of `k` with `v` more literally? The input continuation will not be needed again, so why not use it as the result? Suppose that, in addition to regular constructors and the case form, the `datatype` form is extended to define "recycling constructors" for each variant. For instance, when the `List` datatype is defined, we get not only the `Pair` constructor, but also a recycle-as-Pair constructor, which is a new syntactic form that looks like this:

⟨*form of* recycle-as-Pair⟩≡
```
(recycle-as-Pair record-expr [fieldname expr] ...)
```

Evaluating this form is roughly equivalent to evaluating the following pseudo-expression:

⟨*pseudo-expansion of* recycle-as-Pair⟩≡
```
(let ((record record-expr))
  (coerce-to-Pair! record)
  (set-fieldname! record expr)
  ...
  record)
```

where coerce-to-Pair! is an imagined low-level operation that requires its argument to be the same size record as a Pair and "re-tags" the record as a Pair. The also-imagined set-head! and/or set-tail! update the appropriate fields of the record with the values of their second arguments. If no update is specified for a field, its value remains unchanged after recycling. Thus, we could rewrite invoke$_L$ like this:

⟨*recycling list continuations (record representation)*⟩≡
```
(define invoke_L
  (lambda (c v)
    (List-k-case c
      ((Initial-k) v)
      ((Pairing-k h k)
       (invoke_L k
         (recycle-as-Pair c [tail v]))))))
```

This version of invoke$_L$ does exactly what has been suggested. It reuses the memory formerly used by the Pairing-k[3] to construct a Pair whose contents are the same as those of the Pairing-k, except that the tail field contains the value of v, instead of whatever used to be there. In other words, the Pairing-k is "re-tagged" to be a Pair, and the tail field is updated to contain a new value. No new memory is allocated.

A call to list-copy still produces the same answer as before, but now the behavior along the way has changed quite a bit:

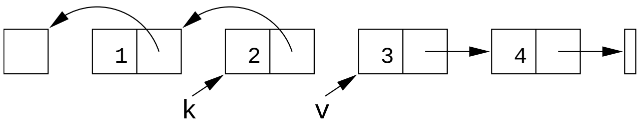
1. First, list-copy-cps walks down the list, creating a Pairing-k record for each Pair in the list.
2. At the end of the list, the base case calls invoke$_L$ with the chain of Pairing-ks and the Empty-List as arguments.
3. Then, invoke$_L$ inverts the "backward-pointing" links (k) that connect each continuation to the next, making them point "forward" to the tail (v) instead.
4. When invoke$_L$ reaches the initial continuation, the v (which is by now a complete copy of the original list) is returned.

Only *two* data structures are involved in a call to list-copy now: the input list and the continuations/output list. No extra memory is used beyond that which is necessary for creating the result.[4]

This diagram represents the situation as invoke$_L$ is called for the third time when copying a list containing the numbers 1, 2, 3, and 4.

[3]Ironically, exploiting linearity requires its violation.

[4]Actually, a constant amount of memory is used for allocating the Initial-k, so it is more accurate to claim that the memory use of list-copy is constant, not counting the memory used for the output.



The narrow box on the far right is an Empty-List. The square on the left is the initial continuation. The records containing 1 and 2 are Pairing-ks. The records containing 3 and 4 are Pairs.

After this last transformation, list-copy works almost exactly like the standard link-inversion algorithm for stackless list traversal. The difference is that, whereas most presentations of link inversion treat it as a means of traversing an existing list, list-copy uses link inversion to traverse its freshly allocated result list. Since no external references to the result can exist yet, the usual danger created by link inversion goes away: two traversals cannot happen at once.

The first version of list-copy, at the beginning of Section 3, had the desirable property that its recursive structure closely matched the inductive type structure of the lists being copied. That correspondence is now less clearly present in the code, which is why it is preferable to begin by writing the direct-style version and to convert to the more space-efficient version through correctness-preserving transformations. After four transformations, the program still produces the same answer, but it uses no extra space, whereas the original program requires extra space proportional to the length of the list in order to compute its result. To complete the path toward a more standard link-inversion algorithm for lists, one can observe that maintaining the distinction between List and List-k is not necessary. Thus, the List-k type could be eliminated in favor of Lists everywhere, including the representation of continuations, obviating the record coercion.

3.5 Other List Examples

The transformation technique demonstrated in the preceding sections never depended on the form of the input to the procedure being transformed. A certain control flow was necessary in order to produce the desired output, and a certain form for the continuations was induced by the control flow.

To clarify that the form of the output is the determining factor, consider that from having seen only list-copy, it would be easy to conclude that this sequence of transformations only works for programs that take a list of length n and return a new list, also of length n. In fact, exactly the right number of continuations are allocated for procedures in which the lengths are two *different* numbers n and m. For instance, the procedure remove returns a list whose length is less than or equal to the length of its input, eliminating the elements that match its first argument.

⟨*remove*⟩≡
```
(define remove
  (lambda (x l)
    (letrec
       ((rem (lambda (l)
               (List-case l
                 ((Empty-List) (Empty-List))
                 ((Pair h t)
                  (if (equal? h x)
                      (rem t)
                      (Pair h (rem t))))))))
     (rem l))))
```

After completely transforming this program, it becomes:

⟨*remove with recycling*⟩≡
```
(define remove
  (lambda (x l)
    (letrec
        ((rem-cps (lambda (l k)
                    (List-case l
                      ((Empty-List)
                       (invoke_L k (Empty-List)))
                      ((Pair h t)
                       (if (equal? h x)
                           (rem-cps t k)
                           (rem-cps t
                             (Pairing-k h k))))))))
      (rem-cps l (Initial-k)))))
```

where $\texttt{invoke}_L$ is defined exactly as in the recycling version of `list-copy`. (In fact, $\texttt{invoke}_L$ will be the same for almost any program that constructs lists. A counterexample is the procedure `double`, which creates a list with two elements for every one in the input list, so that the output list is twice as long as the input list.) Here we see that a `Pairing-k` is created just in case a `Pair` is needed in the output list. No space is wasted by the continuations.

The input need not even be a list, though. A program need only construct a list as its output to be suitable for our transformation. For example, the `iota` procedure, which takes a natural number `n` and produces a list that contains the numbers from `0` up to (but not including) `n`, is also a suitable candidate for eliminating stack overhead.

⟨*iota*⟩≡
```
(define iota
  (lambda (n)
    (letrec
        ((up (lambda (i)
               (cond
                ((= i n) (Empty-List))
                (else (Pair i (up (+ i 1))))))))
      (up 0))))
```

Since `iota` returns a list, it needs to construct a `Pair` at each recursive step. After the transformations, we get:

⟨*iota with recycling*⟩≡
```
(define iota
  (lambda (n)
    (letrec
        ((up-cps (lambda (i k)
                   (cond
                    ((= i n)
                     (invoke_L k (Empty-List)))
                    (else
                     (up-cps (+ i 1)
                       (Pairing-k i k)))))))
      (up-cps 0 (Initial-k)))))
```

The construction of the `Pair` has been replaced by the construction of the pairing continuation, which will be converted to a `Pair` in $\texttt{invoke}_L$, after the base case is reached.

4 Trees

Lists are sometimes beguilingly simple. Their definition has a single recursive reference, which makes it straightforward to use them as their own continuations, or even to traverse them iteratively. This is not possible for arbitrary, inductively-defined structures. In the following sections, we attempt to apply the same sequence of transformations to a tree-copying procedure that we applied to the list-copying procedure. This attempt reveals subtleties (clarified in Section 4.4) that did not arise in the preceding section.

We begin, as before, with a datatype definition. For simplicity and without loss of generality, there will be no distinct leaf type; a leaf is a node with two empty subtrees.

⟨*tree datatype*⟩≡
```
(datatype Tree (Empty-Tree) (Node datum left right))
```

The direct-style copying procedure induced by this type is:

⟨*tree copy*⟩≡
```
(define tree-copy
  (lambda (t)
    (Tree-case t
      ((Empty-Tree) (Empty-Tree))
      ((Node d l r)
       (Node d (tree-copy l) (tree-copy r))))))
```

It is the presence of *two* recursive calls to `tree-copy`—and the accompanying two input and output subtrees—that will act as the crucible for our sequence of transformations, helping us to refine and generalize them.

4.1 (Trees) Continuation-Passing Style

Even the standard CPS conversion raises issues that were hidden before. CPS is inherently sequential, but the two recursive calls to `tree-copy` have no specified ordering in the direct-style programs in Scheme. We arbitrarily choose a left-to-right evaluation ordering.

⟨*tree copy in CPS*⟩≡
```
(define tree-copy-cps
  (lambda (t k)
    (Tree-case t
      ((Empty-Tree) (k (Empty-Tree)))
      ((Node d l r)
       (tree-copy-cps l
         (lambda (vl)
           (tree-copy-cps r
             (lambda (vr)
               (k (Node d vl vr))))))))))

(define tree-copy
  (lambda (t)
    (tree-copy-cps t (lambda (v) v))))
```

As in the list examples, converting the program to CPS makes explicit the still-to-be-done computation at any point. The procedure walks down the leftmost path in the tree, creating a sequence of nested closures formed from the `(lambda (vl) ...)` continuation. Upon reaching the end of the left branch, the last of these closures is invoked, and the procedure begins to walk down a right branch. When both left and right branches are completed, the next continuation in the chain is invoked on a newly created node. Even though the data structure being traversed is now a tree, the continuation always remains a linear structure. This is precisely what makes a stack-based runtime architecture feasible.

4.2 (Trees) Representation Independence

The `list-copy` procedure creates two different kinds of continuations; the tree version creates three. The first is the omnipresent empty continuation. The second is created when descending into left branches, and the third—for right branches—is created only when the second is invoked. Here is a version of the CPS tree copy expressed more abstractly:

⟨*tree copy in representation-independent CPS*⟩≡
```
(define tree-copy-cps
  (lambda (t k)
    (Tree-case t
      ((Empty-Tree) (invoke_T k (Empty-Tree)))
      ((Node d l r)
       (tree-copy-cps l
         ⟨construct left continuation⟩)))))

(define tree-copy
  (lambda (t)
    (tree-copy-cps t (Initial-k))))
```
Given the procedural definition of $\texttt{invoke}_T$,

⟨*invoking tree continuations (procedure representation)*⟩≡
```
(define invoke_T
  (lambda (k v)
    (k v)))
```
the constructor for the initial continuation is the same as the one for lists:

⟨*tree continuation constructors (procedure representation)*⟩≡
```
(define Initial-k
  (lambda ()
    (lambda (v) v)))
```
Constructors for nested continuations are easiest to write starting with the innermost and working outward.

⟨*tree continuation constructors (procedure representation)*⟩+≡
```
(define Right-k
  (lambda (d vl k)
    (lambda (v)
      (invoke_T k (Node d vl v)))))
```
This clarifies that the constructor for left continuations must be:

⟨*tree continuation constructors (procedure representation)*⟩+≡
```
(define Left-k
  (lambda (d r k)
    (lambda (v)
      (tree-copy-cps r (Right-k d v k)))))
```
Thus, the missing part of `tree-copy` can now be completed only one way:

⟨*construct left continuation*⟩≡
```
(Left-k d r k)
```

4.3 (Trees) Representing Continuations as Records

Section 3.3 ended with the observation that the types `List-k` and `List` were isomorphic and that the distinction between them could be dropped. This was clearly a special case, because `Tree` and `Tree-k` are dissimilar:

⟨*tree continuation type*⟩≡
```
(datatype Tree-k
  (Initial-k)
  (Left-k datum right k)
  (Right-k datum left k))
```
A fact that may not be readily apparent is that the `right` field of `Left-k` (the `r` parameter above) and the `left` field of `Right-k` (the `vl` parameter above) point into two different structures, the old tree and the new tree, respectively. The `right` field of `Left-k` always refers to the old tree, the one being copied. The `left` field of `Right-k` refers to the newly constructed left subtree, which will become a part of the output tree.

As before, the burden of behavior is lifted from the continuations and falls onto the $\texttt{invoke}_T$ procedure. Copying the bodies of the continuation procedures from the preceding section, we get:

⟨*invoking tree continuations (record representation)*⟩≡
```
(define invoke_T
  (lambda (k v)
    (Tree-k-case k
      ((Initial-k) v)
      ((Left-k d r k)
       (tree-copy-cps r (Right-k d v k)))
      ((Right-k d l k)
       (invoke_T k (Node d l v))))))
```
The linked structure of the continuations is completely explicit again. If we were to observe the dynamic behavior of the chain of continuations, though, we would see something that we did not observe in the list example. The chain of list continuations grew (monotonically) until it reached its maximum length. Then it shrank (monotonically) until the initial continuation was invoked. The chain of tree continuations alternately grows and shrinks, matching the growth pattern that a stack would exhibit during a traversal of the tree being copied. The *total* number of continuation records created (which is greater than the maximum length of the chain) is the same as the number of nodes in the tree, again inviting us to reuse the space they occupy.

4.4 (Trees) Link Inversion

Comparing the tree $\texttt{invoke}_T$ with the list $\texttt{invoke}_L$, we observe that the first case clause is identical to the list version. The last case clause is so similar to what we saw in the list example that it is evident what to do with it: simply replace `k` with `v`. The middle clause is new, but it is still fairly obvious that the `Right-k` being constructed is nearly identical to the `Left-k` it replaces. The only difference is that the `r` in the `Left-k` is replaced with `v` in the `Right-k`. Thus, it is again possible to recycle the continuations and avoid allocating new records.

⟨*recycling tree continuations (record representation)*⟩≡
```
(define invoke_T
  (lambda (c v)
    (Tree-k-case c
      ((Initial-k) v)
      ((Left-k d r k)
       (tree-copy-cps r
         (recycle-as-Right-k c [left v])))
      ((Right-k d l k)
       (invoke_T k
         (recycle-as-Node c [right v]))))))
```
Let us walk through the sequence of events that begins with a call to `tree-copy` on some tree.

1. As `tree-copy-cps` walks down the leftmost branch of the tree, it creates for each node a `Left-k` record. That record contains the node's datum and a pointer to its right subtree, which still needs processing.

2. At the first empty left subtree, `tree-copy-cps` calls $\texttt{invoke}_T$ with the chain of `Left-k`s and an `Empty-Tree` as arguments.

3. Next, $\texttt{invoke}_T$ turns the `Left-k` into a `Right-k`, saving the new left subtree (an `Empty-Tree`, for the first in the chain of `Left-k`s), and redirects `tree-copy-cps` to the old node's right subtree.

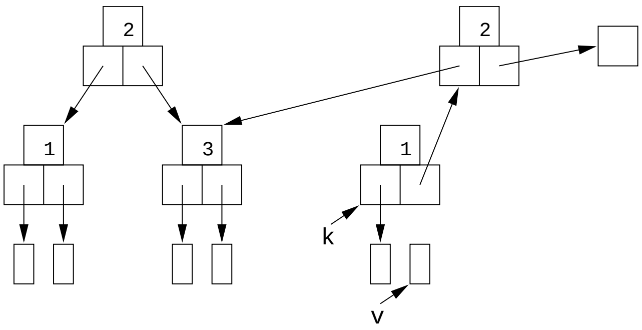
4. The whole process starts over with the right subtree, until a node with no children is reached. Then,

tree-copy-cps calls $\texttt{invoke}_T$, and the first continuation in the chain is now a Right-k.

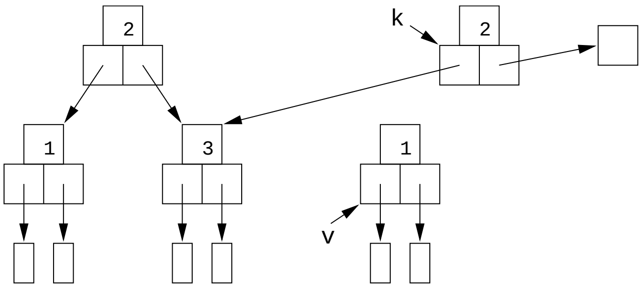
5. Later, $\texttt{invoke}_T$ turns the Right-k into a Node, inverting the "upward-pointing" link (to the next continuation in the chain) so that it points "down" to the right subtree instead, and moves on to the next continuation.

6. When $\texttt{invoke}_T$ reaches the initial continuation, the whole new tree is returned.

Suppose tree-copy is called on a balanced 3-node tree. This diagram represents the situation as $\texttt{invoke}_T$ is called for the second time:



The tree on the left is the tree being copied. The current continuation is a Right-k. In the next step, the continuation is recycled as a Node, with the right field updated to point to the current v (an Empty-Tree). Then, $\texttt{invoke}_T$ is called again, with new values for k and v:



Now the current continuation is a Left-k, and the value of v is the Node that was just completed.

As with the list copying program, only two data structures are involved now: the input tree and the chain of continuations, which gradually becomes the output tree. *It is not possible, however, to claim that* tree-copy *has no memory overhead.* The continuations are not isomorphic to trees because it is necessary to distinguish between left and right continuations, whereas trees have only one kind of node. In link inversion terminology, a tag bit is necessary to determine which pointers have already reached their final state, and which are being used to save data that still need to be processed. Since there are only two alternatives, the distinction can be encoded in a single bit. The length of the chain of continuations is equal to the depth of the recursion at any moment, so the maximum amount of memory needed is a number of bits equal to the maximum depth (i.e., the height) of the tree. At least one undergraduate data structures textbook comes very close to expressing the connection between the tag bit and continuations:

> This method ... does not completely eliminate the memory used by the stack of the recursive algorithm, though it does reduce it to a single bit per cell. In essence, this bit encodes the same information as is needed in the recursive version ... to indicate, when a recursive call finishes, which of the two recursive calls in the body of the program caused that invocation, and hence where in the body of the program execution should continue [6, page 114].

In comparison with the prior versions of tree-copy, a great deal of memory overhead has been eliminated, but it is not possible to remove *all* the overhead. In binary trees, the remaining overhead is merely one bit per level of recursion, but the situation grows worse for more general recursive structures. For a structure with n recursive fields, there will be n different kinds of continuations to distinguish, leading to an overhead of $\lg n$ bits per level of recursion. In practice, record length is constrained by memory word length, so the overhead is bounded to one word per level of recursion.[5]

The preceding discussion notwithstanding, most *implementations* of user-defined types and variants actually take an entire word to tag each variant and distinguish among them. Thus, even though the left and right continuations theoretically occupy more space than the tree node, they typically take up *exactly the same amount of space* in practice. Otherwise, it would not really be a fair trick to recycle a Right-k as a Node, as was done in the last definition of $\texttt{invoke}_T$.

5 General Polynomial Types

The transformations in the preceding sections generalize to types other than lists and trees. In fact, most procedures that act as generators of any polynomially typed data (sums of product or record types [7]) can use link inversion, instead of stack space. In particular, our sequence of transformations works for every finite-output *anamorphism* [8], to be defined more precisely in Section 6.

The transformations in the following three sections are based on those in the sources already cited in the examples, but we have reconstructed and extended them to suit the language of the examples and the needs of the fourth transformation, which is entirely our own.

5.1 Continuation-Passing Style

The CPS algorithm we use is standard, with the following exception: constructors and most language-defined primitives are treated as atomic operations, like variable reference. Procedures that run user code, such as map, are not treated atomically; they need to be rewritten by the user (e.g., map-cps). Also, we insist on defining the "wrapper" procedures that explicitly call the CPS-transformed versions with initial continuations. For example, it would have been difficult to discuss the complete behavior of list-copy-cps without referring to the list-copy wrapper.

Most complete CPS algorithms define translations for call/cc; however, in order for it to be possible to recycle the continuations, they must be treated linearly (in the

[5] Another way to implement the algorithm is with an extra pointer in each record, which identifies the next field that still needs to be processed. This also takes one memory word per record. Still another implementation uses plain tree nodes plus a stack of tag bits [17].

sense of linear logic). The translation for `call/cc` creates two references to—and multiple potential uses of—the continuation, and so we disallow it in the programs for which recycling is desired.

5.2 Representation Independence

This transformation can be briefly described as follows:

1. Replace all continuation invocations `(k E)` with `(invoke k E)`.
2. Beginning with the innermost continuation constructions (`lambda`-expressions), transfer each continuation to a new definition, using an outer `lambda`-expression to bind free variables.
3. At the site of the original continuation constructions (the ones that were just moved), insert calls to the newly defined constructors, passing the values of the free variables.
4. Define `invoke` as `(lambda (k v) (k v))`.

The ordering of the free-variable parameters (i.e., the parameters of the outer `lambda` of each constructor) provides some room for optimization. The goal is to do as little shifting as possible at the recycling stage. The most general solutions to this problem are equivalent to register allocation strategies in compilers. A simple heuristic that works well, though, is based on the fact that the free variables usually match the fields of the records over which the recursion is defined. The value of one field is the argument to the continuation; the rest of the fields should be listed in the order they appear in the record. Thus, there will be one field fewer mentioned in the free-variable parameters of the continuation constructors than there are in the variant being handled by the continuation. Every continuation except the initial one also needs to refer to exactly one other continuation. If the other continuation is passed as the last argument to the constructor, the total number of arguments will be the same as the number of fields in the variant being handled, and the order will simplify its recycling later.

5.3 Representing Continuations with Records

In this transformation, we eliminate all the constructor definitions produced in the preceding transformation and replace them with a single datatype definition. The fields of each variant should be exactly the same as the free-variable parameters in the corresponding procedural constructor. The `invoke` procedure is rewritten to use the case form of the continuation datatype. Each clause contains the body of the constructor procedure from which the variant was derived. If the interpretive (dispatch) overhead of `invoke` is an important concern, the kinds of jump-table optimization performed by Smalltalk-style object systems can be used to recover most of the speed of direct procedure calls.

5.4 Link Inversion

After completing the three preceding transformations, it should be the case that each clause of `invoke` constructs a new variant. The clauses that correspond to the innermost continuations will construct result types, and the other clauses (with the exception of the initial-continuation clause) will match and construct continuation types. As we demonstrated in the list and tree examples, the standard constructors for these types can be replaced with recycling constructors,[6] updating only one field at a time. In fact, when each continuation is first constructed (and allocated), all of its fields will refer to the input data structure. The fields will be updated, one at a time, to refer to continuations that have been completely recycled as output data structures. Finally, when the last field (the `k` field) is updated, and the continuation is recycled as an output type, the next continuation in the chain is invoked. One of its fields is updated to point "down" to the newly completed node, and thus, the link is inverted.

A potential cause for concern at this point is that the record lengths of the continuations might not match the record lengths of the appropriate output records. If there is a mismatch in the lengths, either the continuation record has fewer slots than the output structure, or it has more. In order for the continuation to have too few slots, it must be filling the output structure with data that does not come from free variables. For instance, if the continuation were `(lambda (v) (k (Pair 88 v)))`, it would have only one free-variable slot (for `k`). In truth, it is surprisingly hard to find realistic examples of this problem. (The `iota` procedure of Section 3.5 would at first seem likely to fit into this problem class, but it does not.) If a real case does arise, the values (like `88`) can be passed to the continuation constructors as if they were free (as in `(Pairing-k 88 k)`), or an alternative translation can be used. (The "list of 88's" program could have been written iteratively from the start.)

What about continuations with *more* free variables than there are slots in the output structure? For example, what if a continuation looks like `(lambda (v) (k (Pair (+ x y) v)))` where `x`, `y`, and `k` are free? Clearly, it would be possible to construct this continuation with `(Pairing-k (+ x y) k)`, instead of `(Pairing-k x y k)`. More generally, it is always possible to fold the free values into fewer slots, either by shifting computations to an earlier time (as in the preceding example) or by storing values in an additional record, and adding an indirection. This last solution is obviously less desirable, because it creates more memory overhead, which is precisely what we are trying to eliminate!

Finally, we claim that in practice, the preceding "problem" cases arise very rarely. In fact, we assert—but do not prove here, since the following section will make it clear—that *in all procedures expressed as finite-output anamorphisms* (to be defined shortly)*, the length of the continuation records always matches the length of the output records.*

6 Anamorphisms

We have already argued that what enables the recycling of continuations as output structures is not the *algorithm* being implemented, but a similarity of *form* between the continuations and the output data in structure-generating procedures. One canonical class of generators of inductively typed data is the set of procedures defined as *anamorphisms* [8]. An anamorphism is a recursive function that can be defined using the "lens" operator $[\![\;]\!]$, a generalized version of the list `unfold` functional [1]. For those not acquainted with

[6] In some settings, it may be more appropriate for recycling constructors (e.g., `recycle-as-Pair`) to be a low-level operation that may not always be available to the end user, but may be available to the compiler or to users operating in some privileged mode.

anamorphisms, we define them briefly here, but it is not essential to understand these definitions in order to appreciate the remaining programs.

Definition 1 *A* functor F *is a homomorphism between categories, mapping objects* A *to objects* $F(A)$ *and arrows* $A \xrightarrow{f} B$ *to arrows* $F(A) \xrightarrow{F(f)} F(B)$*, such that identity and composition are preserved:*

$$\begin{aligned} F(1_A) &= 1_{F(A)} \\ F(g \circ f) &= F(g) \circ F(f) \end{aligned}$$

Definition 2 *An* endofunctor *is a functor from a category back to itself.*

Definition 3 *Given an endofunctor* F *and an arrow* $A \xrightarrow{\psi} F(A)$*, an* anamorphism *is an arrow* $A \xrightarrow{[\![\psi]\!]} \mathrm{fix}(F)$ *such that the following diagram commutes:*

$$\begin{tikzcd} A \arrow[r, "\psi"] \arrow[d, "{[\![\psi]\!]}"'] & F(A) \arrow[d, "{F([\![\psi]\!])}"] \\ \mathrm{fix}(F) \arrow[r, equal] & F(\mathrm{fix}(F)) \end{tikzcd}$$

In other words, an anamorphism over ψ *is defined recursively as*

$$[\![\psi]\!] = F([\![\psi]\!]) \circ \psi$$

The preceding definitions allow ψ to be such that no closed-form solution exists for the anamorphism. In lazy languages or languages with streams, it can be useful to admit such anamorphisms. In our setting, however, where we wish to produce finite structures, we require anamorphisms over ψ to terminate (strictly), producing finite data structures. (Otherwise, there would never be an opportunity to "invert the links.")

In programming, the category of interest is usually **Type**, with types as objects and functions as arrows. Functors are a pair of a type constructor and a higher-order function (usually named `map` in the purely functional programming community). For example, the type constructor whose fixpoint is the lists with elements of type α is

$$F = \lambda\tau\,.\,(\texttt{Empty-List}) \mid (\texttt{Pair}\ \alpha\ \tau)$$

The higher-order "map" function is

$$\begin{aligned} F = \lambda f : \tau \to \sigma\,. \\ &\lambda x : F(\tau)\,. \\ &\quad \texttt{case}\ x\ \texttt{of} \\ &\qquad (\texttt{Empty-List}) \Rightarrow (\texttt{Empty-List}) \\ &\qquad (\texttt{Pair}\ a\ t) \Rightarrow (\texttt{Pair}\ a\ f(t)) \end{aligned}$$

or, in Scheme ("un-curried"),

⟨*list map*⟩≡
```
(define List-map
  (lambda (f l)
    (List-case l
      ((Empty-List) (Empty-List))
      ((Pair h t) (Pair h (f t))))))
```

The "lens" brackets $[\![\ ]\!]$ are written out as `ana`. Here is the corresponding `ana` for lists:

⟨*list lens*⟩≡
```
(define List-ana
  (lambda (psi)
    (letrec ((f (lambda (l)
                  (List-map f (psi l)))))
      f)))
```

These allow us to write `list-copy`, `remove`, and `iota` as anamorphisms.

⟨*list copy anamorphism*⟩≡
```
(define list-copy
  (List-ana (lambda (l) l)))
```

⟨*remove anamorphism*⟩≡
```
(define remove
  (lambda (x l)
    (letrec ((psi (lambda (l)
                    (List-case l
                      ((Empty-List) l)
                      ((Pair h t)
                       (if (equal? x h) (psi t) l))))))
      ((List-ana psi) l))))
```

⟨*iota anamorphism*⟩≡
```
(define iota
  (lambda (n)
    ((List-ana (lambda (i)
                 (cond
                  ((= i n) (Empty-List))
                  (else (Pair i (+ i 1))))))
     0)))
```

But now, instead of transforming the individual procedures, why not transform the `ana` operator itself (along with the appropriate datatype functors)? If we use these completely transformed versions:

⟨*list map with recycling*⟩≡
```
(define List-map-cps
  (lambda (f l k)
    (List-case l
      ((Empty-List) (invoke_L k (Empty-List)))
      ((Pair h t) (f t (Pairing-k h k))))))
```

⟨*list lens with recycling*⟩≡
```
(define List-ana-cps
  (lambda (psi)
    (letrec ((f (lambda (l k)
                  (List-map-cps f (psi l) k))))
      f)))

(define List-ana
  (lambda (psi)
    (lambda (l)
      ((List-ana-cps psi) l (Initial-k)))))
```

and use the recycling definition of $\texttt{invoke}_L$ from Section 3.4, then the definitions of `list-copy`, `remove`, and `iota` can remain untouched, but they now run without any stack or continuation space overhead! Since many of the code-transformation passes of a compiler or language preprocessor can be written as anamorphisms, the recycling technique appears to be very widely applicable. In fact, the CPS transformation itself can be implemented this way, inviting self-application of the optimization.

7 Future Directions

We have demonstrated a method for transforming both specific structure-producing programs and general anamorphism operators to eliminate the stack-space usage of recursion. For lists, it is clearly possible to go further and eliminate the link-inversion stage by inverting the links forward as the pairs/continuations are constructed. This makes list construction fully iterative, as in Minamide's paper [10]. There should be some way to extend the iterative property to one spine of trees and other polynomial types, but we have not yet discovered an elegant transformation (to follow the link-inversion transformation) to produce this property.

Wand [16] demonstrates an elegant sequence of transformations[7] that exploits the connection between CPS and accumulator-passing style. He makes a different, but related, set of observations about the free variables in continuation terms, enabling recursive procedures to be converted to iterative ones with accumulators. Where Wand performs his conversion by proving that the accumulated value correctly represents the continuation, we simply switch to another—obviously equivalent—representation. Our transformations are presented rather informally, but Wand's technique for formal equivalence proofs should be very easily applicable to all but the last transformation. We expect the introduction of side effects to complicate somewhat the equivalence proof for the last transformation.

Meijer and Hutton extend anamorphisms to exponential types (function spaces) [9]. Their work should fit naturally into our setting, but we have not explored the implications of this mix. Furthermore, since anamorphisms and catamorphisms are duals we wonder whether there is a dual to our technique, which might provide some sort of optimization for catamorphisms.

The relationship between continuation-passing style and monadic style [11] invites the extension of this work to a monadic setting. The CPS monad is but one monad that incurs a cost in terms of space. We plan to explore the extension of these transformations to other monads or some general monadic framework. (For related work, see Chen and Hudak's discussion of translating functional, linear abstract datatypes into monadic ones with updatable state [4].)

Also, one of the great powers of monads is their provision for the reification and reflection of monadic meta-information. In order to retain this power, the precise implications of the presence of reflective operators like `call/cc` must be explored in the context of recycling.

Acknowledgments

Our thanks to Erik Hilsdale for hours of discussion and help with the `datatype` macro. Thanks to Erik, Matthias Felleisen, Mitch Wand, Jon Rossie, Steve Ganz, and an anonymous referee for thorough readings and insightful comments to guide us on our way. We also appreciate the comments of Michael Levin, Anurag Mendhekar, Oleg Kiselyov, and the participants of the Friday Morning Programming Languages Seminar at Indiana University.

References

[1] Richard Bird and Philip Wadler. *An Introduction to Functional Programming*. Prentice-Hall, 1988.

[2] Luca Cardelli. The functional abstract machine. Technical Report TR-107, Bell Labs, 1983. Bell Labs Technical Memorandum TM-83-11271-1.

[3] Luca Cardelli. Compiling a functional language. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, pages 208–217. ACM Press, 1984.

[4] Chih-Ping Chen and Paul Hudak. Rolling your own mutable ADT: A connection between linear types and monads. In *Conference Record of POPL '97: The 24TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 54–66, Paris, France, January 1997. ACM Press.

[5] R. Kent Dybvig. *Three Implmentation Models for Scheme*. PhD thesis, University of North Carolina at Chapel Hill, 1987.

[6] Harry R. Lewis and Larry Denenburg. *Data Structures and Their Algorithms*. HarperCollins, 1991.

[7] Grant Malcolm. Algebraic data types and program transformation. *Science of Computer Programming*, 14(2–3):255–280, September 1990.

[8] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes, and barbed wire. In *FPCA '91: 5^{th} International Conference on Functional Programming Languages and Computer Architecture*, number 523 in Lecture Notes in Computer Science. Springer-Verlag, 1991.

[9] Erik Meijer and Graham Hutton. Bananas in space: Extending fold and unfold to exponential types. In *FPCA '95: 7^{th} International Conference on Functional Programming Languages and Computer Architecture*, pages 324–333, La Jolla, June 1995. ACM Press.

[10] Yasuhiko Minamide. A functional representation of data structures with a hole. In *Conference Record of POPL '98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Diego, January 1998. ACM Press.

[11] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, July 1991.

[12] Greg Morrisett, Matthias Felleisen, and Robert Harper. Abstract models of memory management. In *FPCA '95: 7^{th} International Conference on Functional Programming Languages and Computer Architecture*, pages 66–77, La Jolla, June 1995. ACM Press.

[13] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM National Conference*, pages 717–740. ACM Press, 1972.

[14] H. Schorr and W. M. Waite. An efficient machine-independent procedure for garbage collection in various list structures. *Communications of the ACM*, 10(8):501–506, August 1967.

[15] Jeffrey D. Smith. *Design and Analysis of Algorithms*. PWS-KENT, Boston, 1989.

[16] Mitchell Wand. Continuation-based program transformation strategies. *Journal of the Association for Computing Machinery*, 27(1):164–180, January 1980.

[17] Benjamin Wegbreit. A space-efficient list structure tracing algorithm. *IEEE Transactions on Computers*, C21:1009–1010, 1972.

[18] A. Van Wijngaarden. Recursive definition of syntax and semantics. In *Formal Language Description Languages*, pages 13–24. North-Holland, 1964.

[7] In his words, "In this paradigm, one writes a clear, correct, though possibly inefficient, program, and then transforms it via correctness-preserving transformations into a program which is more efficient although probably less clear."