January 1995

Using compact data representations for languages based on catamorphisms

Leonidas Fegaras

Andrew Tolmach

Using Compact Data Representations for Languages Based on Catamorphisms

OGI, Tech-report #95-025

Leonidas Fegaras
Department of Computer Science
Oregon Graduate Institute
fegaras@cse.ogi.edu

Andrew Tolmach
Department of Computer Science
Portland State University
apt@cs.pdx.edu

Abstract

We describe a new method for improving the performance of functional programs based on catamorphisms. The method relies on using a compact vector representation for the recursive structure over which the catamorphism operates. This saves space and allows catamorphisms to be implemented in tail-recursive fashion even in cases where the standard linked structure representation requires non-tail-recursive evaluation. Preliminary experimental measurements show substantial improvements are possible with our approach.

Keywords: program transformation, compilation methods, data representations, catamorphisms.

1 Introduction

Most functional languages provide higher-order library functions that capture common computation patterns over recursive data structures. These operators allow algorithms to be expressed at a higher level of abstraction than explicitly recursive programs that manipulate the data structure "one piece at a time." Perhaps the most useful of these operators is the *catamorphism* (literally: down-former) that expresses "regular recursion" over data structures. For lists, this operator is known as fold in ML and foldr in Haskell. Catamorphisms are quite powerful: for example, any primitive recursive function can be written as a catamorphism [13]. Catamorphisms may be specified by the user or, under some conditions, generated from ordinary recursive function code [7]. While catamorphisms can easily be expressed in terms of ordinary recursive functions, there has been much recent interest in exploiting a "higher-level" view of catamorphisms to improve program performance, in particular by performing program fusion [13, 3, 7, 1].

In this paper, we explore a new method for improving the performance of program with catamorphisms over recursive structures. We optimize the *representation* of the structures, taking advantage of the fact that catamorphisms act on an entire structure as a unit in a highly stereotyped manner. Specifically, we use a compact vector representation for structures, built using imperative update operations. The vectors contain the actual data of the recursive structure but no internal links; data are stored in a specified flattened order. These vector representations reduce the space requirements for the structure by up to 50% over the conventional linked representation. More importantly, using vector representations allows the structure to be accessed either "top-down" or "bottom-up." The ability to read bottom-up allows us to encode many catamorphisms into tail-recursive functions that run much faster than the top-down versions required by the conventional linked representation. Our technique is complementary to catamorphism fusion; it is suitable in cases where fusion fails and some intermediate data structure must be built.

Our main ideas apply to arbitrary tree-like algebraic datatypes. In this paper, however, we concentrate on lists, both because these are the most common recursive structure and because vector forms of lists are especially flexible. In particular, we can always read vector lists efficiently, either top-down (from head to tail) or bottom-up (from tail to head). Of course, the vector form of lists can be very inefficient to construct, because, unlike linked representations, it does not allow for tails to be shared between lists. Thus, naively adopting vectors everywhere would introduce arbitrary amounts of extra copying compared to the linked representation. Any practical system must support both linked and compact representations, and be able to switch between them at appropriate points with minimal overhead.

We describe a translation algorithm that converts list-processing programs into selectively vectorized form. Our source language is a pure strict functional language with list constructors (essentially as in SML [10]), extended with catamorphisms viewed as primitive forms. The target language adds arrays and imperative update operations. Our analysis and translation works at the level of individual list-building expressions. We use abstract interpretation to identify which expressions that can be profitably modified to produce vector representations. Vectorized expressions should not introduce copying that was not implied by the original expression and their output size should be accurately predictable. The abstract interpretation can deliver useful results on catamorphisms without recourse to fixed points; this makes it much simpler than would be required for a language relying exclusively on general recursion.

To glue together the translated expressions, we use runtime tags to distinguish between linked and compact representations. The checks on these tags can often be elided by use of partial evaluation.

We give preliminary performance measurements illustrating the speedup obtainable for individual expressions, using hand-compiled target code in imperative SML. The translation methods described here could be easily integrated into existing functional language compilers. Some additional benefits could be realized by making the garbage collector aware of vectorized structures as a distinct kind of heap record.

Finally, we sketch the extension of these ideas to arbitrary tree data structures. It is possible to obtain good gains in reading efficiency (though, unlike for lists, some read operations are more expensive on the compact form); the analysis of writing is more complex.

2 Background

The List data type may be defined as

```
datatype List(α) = Nil | Cons of α × List(α)
```

The list catamorphism [9, 13] captures functions with regular recursion over lists. It is defined as follows:

```
fun cataL(f,g) Nil           = f()
|   cataL(f,g) (Cons(a,r))   = g( a, cataL(f,g) r )
```

That is, cataL(f,g) x replaces the Nil constructor in list x by f() and the Cons constructors by g, i.e., if x=Cons(a_1,Cons(a_2, ...,,Cons(a_n,Nil))), then cataL(f,g) x computes g(a_1,g(a_2, ...,,g(a_n,f()))). The type of cataL is $\forall\alpha,\beta: \ ((){\rightarrow}\beta)\times(\alpha\times\beta{\rightarrow}\beta){\rightarrow}\mathsf{List}(\alpha){\rightarrow}\beta$. A list catamorphism can always be written in the form cataL(fn () ⇒ e_1, fn (a,r) ⇒ e_2). The variable r is called the *inductive variable* of the cataL since it holds the intermediate result of the accumulation at each inductive step.

The following are simple list catamorphisms:

```
fun sum x = cataL( fn () ⇒ 0, fn (a,r) ⇒ r+a ) x
fun map(h) x = cataL( fn () ⇒ Nil, fn (a,r) ⇒ Cons(h(a),r) ) x
fun append(x,y) = cataL( fn () ⇒ y, fn (a,r) ⇒ Cons(a,r) ) x
fun find(p) x = cataL( fn () ⇒ None, fn (a,r) ⇒ if p(a) then Some(a) else r ) x
fun reverse x = cataL( fn () ⇒ Nil, fn (a,r) ⇒ append(r,Cons(a,Nil)) ) x
```

sum x computes the sum of the elements of the list x, map(h) x maps the function h onto the elements of the list x, append(x,y) appends y to x, find(p) x finds the first element of x that satisfies p, and reverse is the quadratic list reverse.

The list catamorphism cataL(f,g) x gains most of its expressiveness when it is higher-order, that is, when it computes a function to be applied to some value. Consider, for example, the following function that tests whether the list x is equal to the list y:

```
fun eql(x,y) = cataL(fn () ⇒ fn w ⇒ (w=Nil),
                     fn (a,r) ⇒ fn w ⇒ case w of
                                          Nil ⇒ False
                                        | Cons(b,s) ⇒ (a=b) andalso (r s)) x y
```

This catamorphism inducts over the first input x. The inductive variable r is now a function that deconstructs the second argument y, since each time r is called in (r s), it gets the tail of w as argument, which becomes the new value for w. Since w is initialized to y, the list y is deconstructed one level deeper at each inductive step. Many functions that induct over multiple inputs can be captured using higher-order list catamorphisms. For example, nth(x,n) returns the nth element of the list x:

```
fun nth(x,n) = cataL(fn () ⇒ fn k ⇒ None,
                     fn (a,r) ⇒ fn k ⇒ if k=0 then Some(a) else r(k-1)) x n
```

Higher-order catamorphisms can also be used to accumulate values starting from some initial seed. One example is a linear-time version of list reverse:

```
fun rev x = cataL( fn () ⇒ fn w ⇒ w, fn (a,r) ⇒ fn w ⇒ r(Cons(a,w)) ) x Nil
```

The variable w here plays the role of an accumulator. Its initial value is Nil. At each inductive step, the current element a of the list x is consed onto w, thus augmenting the accumulator. At the end, the accumulated reversed list is returned.

3 The Basic Idea

3.1 Our List Representation

In a typical implementation, lists are represented as linked cells, where each cell contains a list element and a link to the next cell. In our framework, lists can be compacted into vectors. Our compact representations require half as much space as the linked list representations. More specifically, a value of type List(α) is translated into a value of type Vec(α), where

$$\text{datatype Vec}(\alpha) = \text{Vector of int} \times \text{array}(\alpha)$$

where array has the semantics the vector type in SML that supports random access (function sub), random update (function update), and vector construction (function array). The construction Vector(n,a) represents a list in which the vector a contains the list in vector form in reverse order, and the integer n is the length of the list (always less than the size of a). For example, the list Cons(1,Cons(2,Cons(3,Nil))) is represented by Vector(4,[|3,2,1|]). Lists are laid down in a vector in reverse order so that consing a new element at the beginning of a list is done by simply writing this element at the end of the vector (if the vector size is greater than the list length). Under this encoding, we can traverse a list in vector form in two ways: from the head to the tail and from the tail to the head.

Traversing a list of type List(α) in vector form is captured by a vector reader of type VecReader(α):

$$\text{datatype VecReader}(\alpha) = \text{VReader of int} \times \text{Vec}(\alpha)$$

where the integer is the location of the current element of the list during a list traversal.

3.2 Translating Catamorphisms with Simple Output

The compact representation of lists offers many opportunities for program optimization. In this section we demonstrate that any list catamorphism can be translated into a tail recursive function. To illustrate our method, we translate the function

```
fun sum x = cataL( fn () ⇒ 0, fn (a,r) ⇒ r+a ) x
```

such that it accepts the vector representation of the list x instead of x. That is, we want to translate sum into

```
fun Sum x = cataL'( fn () ⇒ 0, fn (a,r) ⇒ r+a ) x
```

```
datatype maybe(α) = None | Some of α
fun lenV(Vector(n,_)) = n
fun elemV(VReader(i,Vector(_,a))) = sub(a,i)
fun initR(v) = VReader(0,v)
fun initL(v as Vector(n,_)) = VReader(n-1,v)
fun newV(n) = Vector(0,array(n,(cast 0)))
fun writeV(e,Vector(n,a)) =
    ( update(a,n,e); Vector(n+1,a) )
fun nextR(VReader(i,v as Vector(n,a))) =
    if i≥n then None
    else Some(sub(a,i),VReader(i+1,v))
```

```
fun nextL(VReader(i,v as Vector(n,a))) =
    if i<0 then None
    else Some(sub(a,i),VReader(i-1,v))
fun appendV(Vector(n1,a1),Vector(n2,a2)) =
    let fun F -1 = ()
          | F i  = (update(a2,n2+i,sub(a1,i)); F (i-1))
    in F(n1-1); Vector(n1+n2,a2) end
fun tailV(VReader(i,Vector(n,a))) =
    VReader(i-1,Vector(i,a))
fun nullV(VReader(_,Vector(n,_))) = (n=0)
```

Figure 1: Semantics of Vector Manipulation Primitives

where cataL' is a catamorphism over the vector representation of lists. As a first attempt, we define a vector catamorphism just as on the usual linked form:

```
fun cataL'(f,g) x = let fun F x = case nextL(x) of
                                      None ⇒ f()
                                    | Some(a,x') ⇒ g( a, F x' )
                    in F(initL x) end
```

where initL (defined in Figure 1) prepares the input vector x to be read from right to left (i.e., from the beginning of the list). The expression nextL(x) returns the current element of the vector x and a new vector reader whose cursor is advanced one position left. To see how cataL' works, consider the list x=Cons(a_1,Cons(a_2,...,,Cons(a_n,Nil))) represented by a vector y. Function F in cataL'(f,g) y reads the elements of y in the order $a_1, a_2, \ldots, a_n$. Consequently, F computes g(a_1,g(a_2,...,g(a_n,f()))), which is equal to cataL(f,g) x.

If we unfold the cataL' definition in Sum (unfolding catamorphisms is always meaning preserving since they always terminate if their constituent functions terminate), we get:

```
fun SUM x = let fun F x = case nextL(x) of
                              None ⇒ 0
                            | Some(a,x') ⇒ (F x')+a
            in F(initL x) end
```

Function F is not a tail-recursive function; in fact, cataL' never unfolds to a tail-recursive function for non-trivial g. But the vector representation of lists offers another alternative, namely to read the list in reverse order, i.e., from tail to head. Consider the following encoding of sum:

```
fun SUM' x = let fun F x res = case nextR(x) of
                                   None ⇒ res
                                 | Some(a,x') ⇒ F x' (res+a)
             in F (initR x) 0 end
```

where initR prepares a vector to be read from left to right, and nextR returns the current element of the vector x and a new vector reader whose cursor is advanced one position right. Notice that F is now a tail recursive function.

In general, any cataL'(f,g) x can be translated into

```
let fun F x res = case nextR(x) of
                      None ⇒ res
                    | Some(a,x') ⇒ F x' (g(a,res))
in F(initR x) (f()) end
```

To see why this is always possible, consider the list x=Cons(a_1,Cons(a_2, . . .,,Cons(a_n,Nil))) represented by a vector y. Function F reads the elements of y in the order $a_n, a_{n-1}, \ldots, a_1$. Consequently, F is called with the following values for res:

$$\mathsf{f()},\ \mathsf{g}(a_n,\mathsf{f()}),\ \mathsf{g}(a_{n-1},\mathsf{g}(a_n,\mathsf{f()})),\ \ldots,\ \mathsf{g}(a_1,\ldots,\mathsf{g}(a_{n-1},\mathsf{g}(a_n,\mathsf{f()})))$$

That is, the last value of res is equivalent to the computation of cataL(f,g) x.

Even though the above translation of cataL(f,g) x is always tail recursive, it may not be the most efficient one. For instance, list catamorphisms that do not need to traverse the entire input list, such as finding the first element of a list that satisfy a predicate, may be more efficiently coded using a top-down evaluation. Worse, the function g may ignore its second argument (the intermediate result), as in cataL(fn () ⇒ None,fn (a,r) ⇒ Just(a)), which computes the head of a list. In those cases, top-down evaluation of lists seems more appropriate. More importantly, cataL(f,g) x may be of higher-order, as in the linear-time reverse function rev. In that case, the parameter res in the bottom-up evaluation is a higher-order value. That is, even though the tail recursive evaluation does not require building a call stack for F, it still requires constructing length(x) different closures. Higher-order catamorphisms are considered in detail in Section 3.5.

3.3 Translating Catamorphisms that Construct Lists

We have shown how catamorphisms can consume vectors efficiently; now we must describe how to generate vectors efficiently. Any list-producing expression could be made to generate a vector, but often copying would be required. Although vector copying is cheap on most hardware architectures, we rule out translations that introduce extra copying compared to the linked representation. Thus, we need to analyze which forms of expressions can produce a vector without unnecessary copying. To simplify the analysis, we focus our attention to list-processing catamorphisms. Consider for example the map function:

```
fun map(h) x = cataL( fn () ⇒ Nil, fn (a,r) ⇒ Cons(h(a),r) ) x
```

Here we not only want to vectorize the input x, but the output of the cataL as well. To do so, we need to allocate a vector for the output and replace the Nil and Cons constructors with vector primitives:

```
fun Map(h) x = let val vec = newV(lenV x)
               in cataL'( fn () ⇒ vec, fn (a,r) ⇒ writeV(h(a),r) ) x end
```

Expression newV(n)[1] constructs a new Vec with a vector of size n and lenV(x) returns the size of x. The length of vec is initially set to zero. Expression writeV(a,vec) writes the element a at the end

[1]The semantics of newV given in Figure 1 allow to return a Vec(α) for any α; this is not actually possible in the type system of ML, but it would be a harmless element because the vector initialization values are never used.

of the vector vec and increments the vector length. That is, vec is used as a stack and consing an element is done by pushing this element into the stack.

Using the same method as before, we derive the following tail recursive definition for Map:

```
fun MAP(h) x = let val vec = newV(lenV x)
                   fun F x res = case nextR(x) of
                                     None ⇒ res
                                   | Some(a,x') ⇒ F x' (writeV(h(a),res))
               in F (initR x) vec end
```

3.4 Condition for the Vectorization of Output

Functions that induce sharing in the linked representation are poor candidates for vectorization. One such function is list append:

```
fun append(x,y) = cataL( fn () ⇒ y, fn (a,r) ⇒ Cons(a,r) ) x
```

If vectorized, this would becomes:

```
fun Append(x,y) = let val vec = newV((lenV x)+(lenV y))
                  in cataL'(fn () ⇒ appendV(y,vec),
                            fn (a,r) ⇒ writeV(a,r)) x end
```

where appendV(y,vec) copies the vector y at the end of vector vec (it modifies vec). We reject Append since it requires some extra copying overhead, not needed for append.

In general, a list expression e is vectorizable iff the *tail term* of e is Nil. We call this condition the *vectorization condition.* The tail term of an expression of type List is either a free variable or the Nil value that constitutes the tail of the resulting list. For example, the tail term of Cons(1,Cons(2,r)) is the variable r. The output of map is vectorizable because the tail term of its cataL is Nil, whereas the output of append is not vectorizable since the the tail term of its cataL is y.

One case worth handling specially occurs when the the intermediate result associated with the inductive variable of a catamorphism is itself a list. Often it is possible to *recycle* this intermediate result without copying. For example, the function calculating the element-wise pairing of two lists x and y:

```
fun pair x y = cataL( fn () ⇒ Nil, fn (a,r) ⇒ cataL( fn () ⇒ r, fn (b,s) ⇒ Cons((a,b),s) ) y ) x
```

should build just one vector. On the other hand, in some cases, the intermediate result cannot be recycled, and a new vector must be allocated. For example, in the quadratic reverse function

```
fun reverse x = cataL(fn () ⇒ Nil,
                      fn (a,r) ⇒ cataL(fn () ⇒ Cons(a,Nil),
                                       fn (b,s) ⇒ Cons(b,s)) r) x
```

the outer cataL cannot just cons onto the intermediate result r, since it must generate a list with tail Cons(a,Nil). It turns out to be easy to distinguish these two cases: if the tail term of g(a,r) in cataL(f,g) is not r, then we need an extra vector to store the new intermediate result. For example, the tail term of cataL(fn () ⇒ Cons(a,Nil), fn (b,s) ⇒ Cons(b,s)) in reverse is Nil, since the last list construction is Cons(a,Nil), whereas the tail term of pair's inner cataL is r.

3.5 Second-order Catamorphisms

Some second-order list catamorphisms can be translated into tail recursive forms. One such example is the linear list reverse:

```
fun rev x = cataL(fn () ⇒ fn w ⇒ w,
                  fn (a,r) ⇒ fn w ⇒ r(Cons(a,w))) x Nil
```

Its vectorized form is

```
fun Rev x = let val vec = newV(lenV x)
            in cataL'(fn () ⇒ fn w ⇒ w,
                      fn (a,r) ⇒ fn w ⇒ r(writeV(a,w))) x vec end
```

In contrast to first-order catamorphisms, it would not be wise to capture this function as a bottom-up traversal. Instead, a top-down traversal is preferable:

```
fun REV x = let val vec = newV(lenV x)
                fun F x w = case nextL(x) of
                              None ⇒ w
                            | Some(a,x') ⇒ F x' (writeV(a,w))
            in F (initL x) vec end
```

Function REV has a tail recursive form, even though it reads the list x from head to tail.

In general, if a second-order catamorphism has the form

```
cataL(fn () ⇒ fn w ⇒ h(w),
      fn (a,r) ⇒ fn w ⇒ f(a,r(g(a,w)))) x b
```

where f and g do not depend on a, r, and w, then it is equivalent to:

```
cataL(fn () ⇒ cataL(fn () ⇒ fn w ⇒ h(w),
                    fn (a,r) ⇒ fn w ⇒ r(g(a,w))) x b,
      fn (c,s) ⇒ f(c,s)) x
```

since it computes $\mathsf{f}(a_1, \mathsf{f}(a_2, \ldots, \mathsf{f}(a_n, \mathsf{h}(\mathsf{g}(a_n, \mathsf{g}(a_{n-1}, \ldots, \mathsf{g}(a_1, b)))))))$ for a list $x = [a_1, \ldots, a_n]$. Consequently, the outer cataL can be put into a tail recursive function that reads x from tail to head and the inner cataL into a tail recursive function that reads x from head to tail.

One example of a second-order catamorphism that cannot be put into tail recursive form is zip:

```
fun zip(h) (x,y) = cataL(fn () ⇒ fn w ⇒ Nil,
                         fn (a,r) ⇒ fn w ⇒ case w of
                                              Nil ⇒ Nil
                                            | Cons(b,s) ⇒ Cons(h(a,b),r(s))) x y
```

Function zip is translated into

```
fun Zip(h) (x,y) = let val vec = newV(lenV x)
                   in cataL'(fn () ⇒ fn w ⇒ vec,
                             fn (a,r) ⇒ fn w ⇒ if nullV(w) then vec
                                                else writeV(h(a,elemV w),r(tailV w))) x (initL y) end
```

where the case expression is compiled into an if-then-else expression and the case patterns into calls to elemV and tailV that compute the head and the tail of a list in vector form.

4 The Translation Process

The compact representation is undesirable for lists with shared tails. A practical system should support both linked and compact representations in the same program. We suggest a simple solution based on "wrapping" lists with headers that allow vectorized and linked representations to be distinguished at runtime [11]. Wrapping and unwrapping are expensive operations, but are required only when passing lists to and from functions. Moreover, partial evaluation can be used to remove pairs of complementary wrap and unwrap operations for known functions.

Conceptually, our scheme works by replacing the standard list datatype by a datatype LIST:

```
datatype LIST(α) = COMPACT of Vec α | LINKED of List α
```

The translation begins by identifying the *list-building* expressions of the program. These are exactly the list-valued expressions that form right-hand sides of let bindings, actual arguments to functions and constructors (except the second argument to Cons), and the tested expression in a case.

The algorithm first attempts to translate each list-building expression into a version that constructs a vector. If this attempt is successful, the translated expression is wrapped in a COMPACT constructor. If the vectorization condition is false for the expression, the vectorizing translation attempt will fail, and an alternative translation returns code that builds a linked list, which is wrapped in a LINKED constructor.

The outcome of the vectorization condition test, and the translated expression itself, may depend on the representations of the list-valued free variables. In a higher-order setting, it is sometimes impossible to know these representations statically. To handle this problem, we generate separate translations for each possible collection of free-variable representations. The entire list-building expression is then replaced by a case statement that dispatches (at runtime) on the actual representations of the free variables. Thus, an expression with n list-valued free variables will produce a case statement with 2^n choices.

For example, the function fun cons12 x = Cons(1,Cons(2,x)) is translated to:

```
fun cons12 x = case x of
                   COMPACT xvec  ⇒ COMPACT(writeV(1,writeV(2,xvec)))
                 | LINKED xlink  ⇒ LINKED(Cons(1,Cons(2,xlink)))
```

The code space and execution time overheads introduced by these case dispatches can be large. Fortunately, it should be possible to eliminate many of these tests by partial evaluation. In particular, any recursive functions acting on linked list representations should be able to avoid wrapping and unwrapping around recursive calls.

Other authors have attacked the problem of static dispatch from the other direction; they propose extended type systems (e.g., refinement types [12] or 2nd-order polymorphic λ-calculus [4]) that allow many checks to be performed statically, but fall back on runtime checks when necessary. We plan to consider these schemes in relation to ours more carefully.

The vectorization condition is evaluated by means of an abstract interpretation on expressions. The interpretation produces precise results for catamorphisms without the need for fixed-point calculations (but gives up immediately when given a recursive function). The ability to write efficient and terminating abstract interpretations is a very useful characteristic of catamorphism-based languages.

In all our examples we have assumed that the output vectors are allocated with the correct size. To vectorize programs automatically, we need a good method of estimating vector sizes. The estimation should be conservative, that is, it should be no less than the real sizes of vectors so no overflow occurs during run time. (Our experiments indicate that dynamically expanding vectors when they overflow is computationally too expensive.) A copying garbage collector could trim oversized list vectors when it moves them, if they were specially identified.

To attack this problem, we use another abstract interpration over list-building expressions. It produces a term, possibly containing catamorphisms over the natural numbers, that evaluates to an integer length guaranteed to bound the size of the output vector. These terms can often be simplified statically into finite sums with closed-form solutions that can be evaluated at compile time. List-building expressions for which size estimation fails are not vectorized.

The formal presentation of key elements of the translation algorithm is in Appendix A.

5 Performance Experiments

To illustrate the potential benefits of our representation, we hand-compiled a variety of common catamorphisms into vector-based code within Standard ML of New Jersey (version 0.93), and compared their execution times against those of standard recursive function definitions. Both versions are coded as tightly as possible, with all operations inlined; the vector versions use the "unsafe" versions of the SML/NJ array primitives to avoid the overhead of bounds checking. Some results are in Table 1.

The catamorphisms tested include **rev** (linear reverse of an integer list using an accumulator), **reverse** (quadratic reverse of an integer list without an accumulator), **add1** (adds one to each element of an integer list), **zipadd** (generates a list of pairwise sums from a pair of integer lists), and **sum** (sums the nodes of a binary tree). Except for **zip** (discussed below), the timings show substantial speedups for all the catamorphisms when using vectors. In some cases a substantial minimum list size must be reached before much benefit is obtained; we assume this is due to the overhead of creating a new array, including the (minor) cost of initializing its contents, which is required for the garbage collector.

Of course, these figures represent best cases for our method, since real programs might not spend much time doing catamorphisms of this sort, but they do suggest the potential for performance gain. For some catamorphisms we measured the performance of an implementation using vectors but *not* tail-recursion. The results clearly indicate that most of the performance improvement is due to tail-recursion elimination; without it, vectorization is slow to recoup its overheads. Interestingly, the one tree example, **sum**, runs up to twice as fast in the vector version even though processing the tree bottom-up converts only one of the two recursive calls present in the standard top-down

Cata	Nodes	T.R. Vect/ Linked	Cata	Nodes	Vect/ Linked	T.R. Vect/ Linked
rev	10	1.75	add1	10	1.36	0.99
	100	1.05		100	0.93	0.50
	1000	0.66		1000	0.80	0.39
	10000	0.62		10000	0.78	0.39
	100000	0.54		100000	0.69	0.31
reverse	10	0.92	zip	10	1.90	1.82
	50	0.58		100	1.65	1.19
	200	0.50		1000	1.50	1.03
	1000	0.45		10000	1.64	1.03
				100000	1.56	0.82
sum	8	0.94	zipadd	10	1.34	0.78
(tree)	128	0.47		100	1.06	0.45
	1024	0.51		1000	0.95	0.37
	8192	0.49		10000	0.99	0.37
	65536	0.51		100000	0.91	0.29

Table 1: Ratios of user time (including garbage collection time) for vectorized (Vect) and tail-recursive vectorized (T.R. Vect) implementation over time for ordinary (Linked) implementation, for various catamorphisms discussed in this paper. Nodes indicates size of list or tree.

algorithm into a tail-recursion.

The zip cata fails to show much improvement; in fact, just switching to vectors without tail-recursion gives a substantial performance *degradation*. This is because zip generates a list of pointers to pairs rather than a simple integer list. The costs of pair allocation, which are the same for both methods, are a significant share of the total cost, lowering the opportunity for speedup. More importantly, SML/NJ imposes heavy overhead for updates to pointer arrays, because it logs the updated addresses to be used as potential live data roots during the next generational garbage collection. If we were able to inform the collector about the status of our target vectors, a more efficient marking algorithm could be used.

6 Extensions

Most recursive datatypes can be vectorized in a manner similar to lists. Binary trees illustrate all the important points. The Tree data type is defined as follows:

$$\mathsf{datatype\ Tree}(\alpha,\beta) = \mathsf{Leaf\ of}\ \alpha \mid \mathsf{Node\ of\ Tree}(\alpha,\beta) \times \beta \times \mathrm{Tree}(\alpha,\beta)$$

The tree catamorphism cataT(f,g) x replaces the Leaf (resp., Node) constructors in a tree with f (resp., g):

```
fun cataT(f,g) (Leaf(a))      = f(a)
|   cataT(f,g) (Node(l,b,r)) = g( cataT(f,g) l, b, cataT(f,g) r )
```

The following are examples of tree catamorphisms:

```
fun tree_map(f,g) x = cataT( fn a ⇒ Leaf(f(a)), fn (l,b,r) ⇒ Node(l,g(b),r) ) x
fun reflect(x) = cataT( fn a ⇒ Leaf(a), fn (l,b,r) ⇒ Node(r,b,l) ) x
fun flatten(x) = cataT( fn a ⇒ fn w ⇒ Cons(a,w), fn (l,b,r) ⇒ fn w ⇒ l(r(Cons(b,w))) ) x Nil
```

A value of **Tree(t)** can be represented in our framework as **Vec(Tree'(t))**, where

```
datatype Tree'(α,β) = Vleaf of α | Vnode of β
```

That is, we remove all recursive references from **Tree**. Trees are vectorized in left-to-right postorder form. For instance, **Node(Node(Leaf 1,2,Leaf 3),4,Leaf 5)** is vectorized as

```
Vector(5,[|Vleaf 1,Vleaf 3,Vnode 2,Vleaf 5,Vnode 4|])
```

Tree constructions are translated as follows:

```
Leaf(a)       ⟶ fn w ⇒ writeV(Vleaf a,w)
Node(l,b,r)   ⟶ fn w ⇒ writeV(Vnode b,r(l(w)))
```

A tree catamorphism **cataT(f,g)** can be translated into the top-down form **cataT'(f,g)**, where

```
fun cataT'(f,g) x = let fun F x = case nextL(x) of
                                      Some(Vleaf(a),x') ⇒ (f(a),x')
                                    | Some(Vnode(b),x') ⇒ let val (r,x1) = F x'
                                                              val (l,x2) = F x1
                                                          in (g(l,b,r),x2) end
                    in π1(F(initL x)) end
```

For example, **tree_map** is computed as follows:

```
fun Tree_Map(f,g) x = let val vec = newV(lenV x)
                      in cataT'(fn a ⇒ fn w ⇒ writeV(Vleaf(f(a)),w),
                                fn (l,b,r) ⇒ fn w ⇒ writeV(Vnode(g(b)),r(l(w)))) x vec end
```

There is an alternative translation into a bottom-up form, using an auxiliary stack **res**, in which one of the two recursive calls is guaranteed to become a tail-recursive call:

```
fun cataT'(f,g) x = let fun F x res = case nextR(x) of
                                          None ⇒ hd(res)
                                        | Some(Vleaf(a),x') ⇒ F x' ((f a)::res)
                                        | Some(Vnode(b),x') ⇒ case res of
                                                                  r::l::res' ⇒ F x' (g(l,b,r)::res')
                    in F(initR x) [ ] end
```

As for list catamorphisms, writing a vectorized tree is not as easy as reading it. The vectorization condition for any catamorphism is similar to the one for list catamorphisms: a tree expression e is vectorizable if its tail term is a **Leaf**. Unlike lists, it can also be expensive to read a vectorized tree even top-down, since inspecting a given node requires inspecting all its right neighbors as well as its ancestors.

7 Related Work and Conclusions

The idea of using compact representation for lists is old. It can be traced back to the cdr-coding schemes for Lisp [5, 8, 6]. Under these schemes, lists are represented as linked vectors of fixed size. Consing an element to a list is done by filling in the first vector in the vector list, if there is space to do so, or by allocating a new vector to fill. All these methods rely on extra cdr-coding bits to identify how data are organized in each vector. These bits need to be checked at each node at run-time, which is quite expensive. Shao, Reppy, and Appel [12] propose a system that exploits compile-time analysis to eliminate most run-time checks. Their representation of a list of length n uses filled vectors of size k with no coding bits for all but the first $(n \bmod k)$ list elements. The first list elements are stored in a vector that includes coding bits (captured in their system by k different value constructors). Thus, the only run-time overhead is when reading the first list elements. Some of these tests too can be eliminated statically. A similar approach is by Hall [4]. Even though her list representation requires the same run-time overhead as cdr-coding, she describes a system in which both linked and compact representations can be used in the same system. Her system assigns local constraints on list structures, in the form of selector fields attached to the list type, that are propagated using a Hindley-Milner type system.

Our work on compact data representation is complementary to program fusion [2] and deforestation (the elimination of intermediate data structures) [14]. Catamorphisms have already been used as a good intermediate representation of programs that supports fusion and deforestation [13, 3, 7, 1]. Even though these methods do a good job on eliminating the unnecessary intermediate structures, some data structures do serve a true computational role and cannot be eliminated. In that case, a compact representation of these structures should improve system performance. A further project is to integrate both methods in a single compiler.

References

[1] R. Bird and O. de Moor. Solving Optimisation Problems with Catamorphisms. In *Mathematics of Program Construction*, pp 45–66. Springer-Verlag, LNCS 669, June 1992.

[2] W. Chin. Safe Fusion of Functional Expressions. *Proceedings of the ACM Symposium on Lisp and Functional Programming, San Francisco, California*, pp 11–20, June 1992.

[3] L. Fegaras, T. Sheard, and T. Zhou. Improving Programs which Recurse over Multiple Inductive Structures. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, Orlando, Florida*, pp 21–32, June 1994.

[4] C. Hall. Using Hindley-Milner Type Inference to Optimize List Representation. *ACM conference on Lisp and Functional Programming, Orlando, Florida*, pp 162–172, June 1994.

[5] W. J. Hansen. Compact List Representation: Definition, Garbage Collection, and System Implementation. *Communications of ACM*, 12(9):499–507, September 1969.

[6] L. Harrison and D. Padua. PARCEL: Project for the Automatic Restructuring and Concurrent Evaluation of Lisp. In *ACM International Conference on Supercomputing*, pp 527–538, June 1988.

[7] J. Launchbury and T. Sheard. Warm Fusion. *Seventh Conference on Functional Programming Languages and Computer Architecture, La Jolla, California*, pp 314–323, June 1995.

Source Language

$$
\begin{array}{lll}
t ::= & \text{basic} \mid \text{List}(t) \mid t_1 \times t_2 \mid t_1 \rightarrow t_2 & \\
 & & \\
e ::= & v & \text{(variable)} \\
 & \mid c & \text{(constant)} \\
 & \mid \lambda v.\, e & \text{(abstraction)} \\
 & \mid e_1\, e_2 & \text{(application)} \\
 & \mid (e_1, e_2) & \text{(pair)} \\
 & \mid \text{Cons} \mid \text{Nil} & \text{(constructors)} \\
 & \mid \pi_1 \mid \pi_2 & \text{(projections)} \\
 & \mid \text{cataL}(\lambda().\, e_1,\ \lambda(v_1, v_2).\, e_2) & \text{(catamorphism)} \\
 & \mid \textbf{let}\ v = e_1\ \textbf{in}\ e_2 & \text{(binding)} \\
 & \mid \textbf{let fun}\ v_1\ v_2 = e_1\ \textbf{in}\ e_2 & \text{(recursive fun.)} \\
 & \mid \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 & \text{(if)} \\
 & \mid \textbf{case}\ e_1\ \textbf{of}\ \text{Nil} \Rightarrow e_2 & \text{(case)} \\
 & \qquad\qquad [\!]\ \text{Cons}(a, r) \Rightarrow e_3 &
\end{array}
$$

Target Language

$$
\begin{array}{ll}
t ::= & \text{basic} \mid \text{List}(t) \mid \text{Vec}(t) \\
 & \mid t_1 \times t_2 \mid t_1 \rightarrow t_2 \\
 & \\
vp ::= & \text{lenV} \mid \text{elemV} \mid \text{writeV} \mid \text{newV} \\
 & \mid \text{appendV} \mid \text{tailV} \mid \text{nullV} \mid \text{initR} \\
 & \mid \text{nextL} \mid \text{initL} \mid \text{nextR} \\
 & \\
e ::= & v \mid c \mid \lambda v.\, e \mid e_1\, e_2 \mid (e_1, e_2) \\
 & \mid \text{Cons} \mid \text{Nil} \mid \pi_1 \mid \pi_2 \mid vp \\
 & \mid \text{cataL}(\lambda().\, e_1,\ \lambda(v_1, v_2).\, e_2) \\
 & \mid \text{cataL}'(\lambda().\, e_1,\ \lambda(v_1, v_2).\, e_2) \\
 & \mid \text{cataN}(\lambda().\, e_1,\ \lambda v.\, e_2) \\
 & \mid \textbf{let}\ v = e_1\ \textbf{in}\ e_2 \\
 & \mid \textbf{let fun}\ v_1\ v_2 = e_1\ \textbf{in}\ e_2 \\
 & \mid \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \\
 & \mid \textbf{case}\ e_1\ \textbf{of}\ \text{Nil} \Rightarrow e_2 \\
 & \qquad\qquad [\!]\ \text{Cons}(a, r) \Rightarrow e_3
\end{array}
$$

Figure 2: The Source and the Target Languages

[8] K. Li and P. Hudak. A New List Compaction Method. *Software – Practice and Experience*, 16(2):145–163, February 1986.

[9] E. Meijer, M. Fokkinga, and R. Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts*, pp 124–144. Springer-Verlag, LNCS 523, August 1991.

[10] L. Paulson. *ML for the working programmer.* Cambridge University Press, 1991.

[11] S. Peyton Jones and J. Launchbury. Unboxed Values as First Class Citizens in a Non-strict Functional Language. *Fifth Conference on Functional Programming Languages and Computer Architecture, Cambridge, MA*, pp 636–665, August 1991.

[12] Z. Shao, J. Reppy, and A. Appel. Unrolling Lists. *ACM conference on Lisp and Functional Programming, Orlando, Florida*, pp 185–195, June 1994.

[13] T. Sheard and L. Fegaras. A Fold for All Seasons. *Sixth Conference on Functional Programming Languages and Computer Architecture, Copenhagen, Denmark*, pp 233–242, June 1993.

[14] P. Wadler. Deforestation: Transforming Programs to Eliminate Trees. *Proceedings of the 2nd European Symposium on Programming, Nancy, France*, pp 344–358, March 1988.

A Formal Framework

In this section, we formally define our translation algorithm that transforms a pure strict functional language based on list catamorphisms into the same language extended with vector operations. The source and target languages are defined in Figure 2. The let-fun expression allows recursive function

$$
\begin{array}{lcl}
\mathcal{T}\mathit{ail}\llbracket x \rrbracket\, \rho & = & \text{if } x \in \rho \text{ then } \rho(x) \text{ else } \{x\} \\
\mathcal{T}\mathit{ail}\llbracket \lambda x.\, e \rrbracket\, \rho & = & \lambda z.\, \mathcal{T}\mathit{ail}\llbracket e \rrbracket\, \rho[z/x] \\
\mathcal{T}\mathit{ail}\llbracket e_1\, e_2 \rrbracket\, \rho & = & (\mathcal{T}\mathit{ail}\llbracket e_1 \rrbracket\, \rho)\, (\mathcal{T}\mathit{ail}\llbracket e_2 \rrbracket\, \rho) \\
\mathcal{T}\mathit{ail}\llbracket (e_1, e_2) \rrbracket\, \rho & = & (\mathcal{T}\mathit{ail}\llbracket e_1 \rrbracket\, \rho, \mathcal{T}\mathit{ail}\llbracket e_2 \rrbracket\, \rho) \\
\mathcal{T}\mathit{ail}\llbracket \text{Cons} \rrbracket\, \rho & = & \lambda(a, r).\, r \\
\mathcal{T}\mathit{ail}\llbracket \text{Nil} \rrbracket\, \rho & = & \{\text{Nil}\} \\
\mathcal{T}\mathit{ail}\llbracket \text{cataL}(\lambda().\, e_1,\ \lambda(a,r).\, e_2) \rrbracket\, \rho & = & \lambda x.\, (\mathcal{T}\mathit{ail}\llbracket e_1 \rrbracket\, \rho)\, \overline{\cup}\, (\mathcal{T}\mathit{ail}\llbracket e_2 \rrbracket\, \rho[(\mathcal{T}\mathit{ail}\llbracket e_1 \rrbracket\, \rho)/r]) \\
\mathcal{T}\mathit{ail}\llbracket \textbf{let } x = e_1 \textbf{ in } e_2 \rrbracket\, \rho & = & (\lambda z.\, \mathcal{T}\mathit{ail}\llbracket e_2 \rrbracket\, \rho[z/x])\, (\mathcal{T}\mathit{ail}\llbracket e_1 \rrbracket\, \rho) \\
\mathcal{T}\mathit{ail}\llbracket \textbf{let fun } f\, x = e_1 \textbf{ in } e_2 \rrbracket\, \rho & = & (\lambda g.\, \mathcal{T}\mathit{ail}\llbracket e_2 \rrbracket\, \rho[g/f])\, (\lambda z.\, \mathcal{T}\mathit{ail}\llbracket e_1 \rrbracket\, \rho[z/x, \bot/f]) \\
\mathcal{T}\mathit{ail}\llbracket \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \rrbracket\, \rho & = & (\mathcal{T}\mathit{ail}\llbracket e_2 \rrbracket\, \rho)\, \overline{\cup}\, (\mathcal{T}\mathit{ail}\llbracket e_3 \rrbracket\, \rho) \\
\begin{array}{l} \mathcal{T}\mathit{ail}\llbracket \textbf{case } e_1 \textbf{ of } \text{Nil} \Rightarrow e_2 \\ \qquad\qquad [\!] \ \text{Cons}(a, r) \Rightarrow e_3 \rrbracket\, \rho \end{array} & = & (\mathcal{T}\mathit{ail}\llbracket e_2 \rrbracket\, \rho)\, \overline{\cup}\, ((\lambda r.\, \mathcal{T}\mathit{ail}\llbracket e_3 \rrbracket\, \rho)\, (\mathcal{T}\mathit{ail}\llbracket e_1 \rrbracket\, \rho))
\end{array}
$$

Figure 3: The Tail Term of an Expression

definitions. The cataN(f,g) n operation is a catamorphism over the natural number n, defined as follows:

```
fun cataN(f,g) 0 = f()
  | cataN(f,g) n = g(cataN(f,g) (n-1))
```

It is used for size estimation.

We separate the compilation into two stages:

- **Phase I**: Vectorization of lists; translation of cataL into cataL';
- **Phase II**: Translation of cataL' into tail recursive function definitions.

Phase II was explained through examples in Section 3. The rest of this section is focused on Phase I.

A.1 The Tail Term of an Expression

The operation $\mathcal{T}\mathit{ail}\llbracket e \rrbracket\, \rho$ in Figure 3 computes the tail terms of the expression e. That is, if e is of type $\text{List}(\alpha)$, then it returns a set that contains free variables or the Nil value, which constitutes the possible tails of the resulting list. The computation of $\mathcal{T}\mathit{ail}$ fails if it derives $\bot$ at any point. The definition of the function $\overline{\cup}$ depends on the type of its input: if x and y are of type $t_1 \to t_2 \to \cdots \to t_n \to \text{set}(t)$, then $x \overline{\cup} y = x \cup_n y$, where $x \cup_0 y = x \cup y$ and $f \cup_n g = \lambda x.\, (f(x)) \cup_{n-1} (g(x))$. The environment ρ maps variables to variables and it is used for renaming lambda variables. There is another environment σ, which is carried through all the abstract interpretations and is omitted in order to make our transformation rules easier to illustrate. It maps variables to the interpretation domains. For example, when our translator finds a binding let x = e in u in an expression, it performs all the abstract interpretations to e and augments the environment σ with the binding from x to the values computed by the interpretations. That way, when x is found in u, its abstract interpretation is directly derived from σ.

The following is an example of computing the $\mathcal{Tail}$ of a term:

$$\begin{aligned}
&\mathcal{Tail}[\![(\lambda z.\, \mathrm{Cons}(1, z))\ (\mathrm{cataL}(\lambda().\, y,\ \lambda(a, r).\, \mathrm{Cons}(a, r))\ x)]\!]\, \rho \\
=\quad &(\lambda w.\, \mathcal{Tail}[\![\mathrm{Cons}(1, z)]\!]\, \rho[w/z])\ (\mathcal{Tail}[\![\mathrm{cataL}(\lambda().\, y,\ \lambda(a, r).\, \mathrm{Cons}(a, r))\ x]\!]\, \rho) \\
=\quad &(\lambda w.\, w)\ (\mathcal{Tail}[\![y]\!]\, \rho)\ =\ \{y\}
\end{aligned}$$

The vectorizing condition for a list expression e is $\mathcal{Tail}[\![e]\!]\, \rho = \{\mathrm{Nil}\}$. This condition implies that the above example is not vectorizable.

From the way it is defined, the abstract interpretation $\mathcal{Tail}$ preserves beta reduction. But the tail terms of two equivalent terms are not necessarily equivalent, as we can see from the following example:

$$\{\mathrm{Nil}\} = \mathcal{Tail}[\![\mathrm{cataL}(\lambda().\, \mathrm{Nil},\ \lambda(a, r).\, \mathrm{Cons}(a, r))\ x]\!]\, \rho \neq \mathcal{Tail}[\![x]\!]\, \rho = \{x\}$$

A.2 Term Translation

The operation $\mathcal{C}[\![e]\!]\rho$ in Figure 4 translates the expression e in the source languages into an expression in the target language. If e is not a variable, $\mathcal{C}$ allocates space to store the result of the computation of e. It is called by $\mathcal{T}[\![e]\!]$ in the following cases:

1. in a let binding, to store the result of binding;
2. in an application, to store the operand;
3. in a case expression, to store the cased value;
4. in the head of a list construction.

The boolean b in $\mathcal{T}[\![e]\!](b, v)\rho$ is true when e is a vectorizable list. The variable v is the location to store the output.

The most important rule of our translator is the rule for translating $\mathrm{cataL}(\lambda().\, e_1,\ \lambda(a, r).\, e_2)$ for a vectorizable output. It checks whether it should allocate a vector R for the intermediate result r by checking the condition for recycling the intermediate result, i.e., by comparing the tail term of e_2 with $\{r\}$.

A.3 Size Estimation

A crucial part of our translation algorithm is allocating vectors for the output values. The algorithm for estimating vector sizes is given in Figure 5. If e is of type List, then $\mathcal{Size}[\![e]\!]\, \rho$ estimates the size of the list computed by e. Function $\overline{\max}$ is defined in terms of max in a manner similar to the way $\overline{\cup}$ is defined in terms of $\cup$. For example, the size of append(x, y) is

$$\begin{aligned}
&\mathcal{Size}[\![\mathrm{cataL}(\lambda().\, y,\ \lambda(a, r).\, \mathrm{Cons}(a, r))\ x]\!]\, \rho \\
=\quad &\mathrm{cataN}(\lambda().\, \mathcal{Size}[\![y]\!]\, \rho,\ \lambda n.\, \mathcal{Size}[\![\mathrm{Cons}(a, r)]\!]\, \rho[n/r,\ \perp/a])\ \mathcal{Size}[\![x]\!]\, \rho \\
=\quad &\mathrm{cataN}(\lambda().\, \mathrm{lenV}(y),\ \lambda n.\, 1 + n)\ (\mathrm{lenV}(x))
\end{aligned}$$

$$
\begin{aligned}
\mathcal{C}[\![x]\!]\rho &= \rho(x) \\
\mathcal{C}[\![e]\!]\rho &= \textbf{let } v = \mathrm{newV}(\mathcal{S}ize[\![e]\!]\,\rho)\textbf{ in } \mathcal{T}[\![e]\!](\mathbf{T},v)\rho \quad \text{if } \mathcal{T}ail[\![e]\!]\,\rho = \{\mathrm{Nil}\} \text{ and } e : \mathrm{List}(t) \\
\mathcal{C}[\![e]\!]\rho &= \mathcal{T}[\![e]\!](\mathbf{F},\bot)\rho \quad \text{otherwise}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{T}[\![x]\!](b,v)\rho &= \begin{cases} v & \text{if } \rho(x) = v \\ \mathrm{initL}(\rho(x)) & \text{if } b = \mathbf{T} \text{ and } \rho(x) : \mathrm{List}(t) \\ \rho(x) & \text{otherwise} \end{cases} \\
\mathcal{T}[\![\lambda x.\, e]\!](b,v)\rho &= \lambda z.\, \mathcal{T}[\![e]\!](b,v)\rho[z/x] \\
\mathcal{T}[\![e_1\, e_2]\!](b,v)\rho &= (\mathcal{T}[\![e_1]\!](b,v)\rho)\,(\mathcal{C}[\![e_2]\!]\rho) \\
\mathcal{T}[\![(e_1,e_2)]\!](b,v)\rho &= (\mathcal{T}[\![e_1]\!](b,\pi_1(v))\rho, \mathcal{T}[\![e_2]\!](b,\pi_2(v))\rho) \\
\mathcal{T}[\![\mathrm{Cons}]\!](\mathbf{F},v)\rho &= \mathrm{Cons} \\
\mathcal{T}[\![\mathrm{Cons}]\!](\mathbf{T},v)\rho &= \lambda(a,r).\, \mathrm{writeV}(\mathcal{C}(a), \mathcal{T}(r)(\mathbf{T},v)\rho) \\
\mathcal{T}[\![\mathrm{Nil}]\!](\mathbf{F},v)\rho &= \mathrm{Nil} \\
\mathcal{T}[\![\mathrm{Nil}]\!](\mathbf{T},v)\rho &= v \\
\mathcal{T}[\![\mathrm{cataL}(\lambda().\, e_1,\ \lambda(a,r).\, e_2)]\!](\mathbf{F},v)\rho &= \mathrm{cataL}(\lambda().\, \mathcal{T}[\![e_1]\!](\mathbf{F},v)\rho,\ \lambda(c,s).\, \mathcal{T}[\![e_2]\!](\mathbf{F},v)\rho[c/a,s/r]) \\
\mathcal{T}[\![\mathrm{cataL}(\lambda().\, e_1,\ \lambda(a,r).\, e_2)]\!](\mathbf{T},v)\rho &= \begin{cases} \left.\begin{array}{l} \mathrm{cataL}'(\lambda().\, \mathcal{T}[\![e_1]\!](\mathbf{T},v)\rho, \\ \qquad \lambda(c,s).\, \mathcal{T}[\![e_2]\!](\mathbf{T},v)\rho[c/a,s/r]) \end{array}\right\} \text{if } \mathcal{T}ail[\![e_2]\!]\,\rho = \{r\} \\ \left.\begin{array}{l} \mathrm{cataL}'(\lambda().\, \mathcal{T}[\![e_1]\!](\mathbf{T},v)\rho, \\ \qquad \lambda(c,s).\, \textbf{let } R = \mathrm{newV}(\mathcal{S}ize[\![e_2]\!]\,\rho[c/a,s/r]) \\ \qquad\qquad \textbf{in } \mathcal{T}[\![e_2]\!](\mathbf{T},R)\rho[c/a,s/r]) \end{array}\right\} \text{otherwise} \end{cases} \\
\mathcal{T}[\![\textbf{let } x = e_1 \textbf{ in } e_2]\!](b,v)\rho &= \textbf{let } z = \mathcal{C}[\![e_1]\!]\rho \textbf{ in } \mathcal{T}[\![e_2]\!](b,v)\rho[z/x] \\
\mathcal{T}[\![\textbf{let fun } f\, x = e_1 \textbf{ in } e_2]\!](b,v)\rho &= \textbf{let fun } g\, z = \mathcal{C}[\![e_1]\!]\rho[z/x, g/f] \textbf{ in } \mathcal{T}[\![e_2]\!](b,v)\rho[g/f] \\
\mathcal{T}[\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!](b,v)\rho &= \textbf{if } \mathcal{T}[\![e_1]\!](\mathbf{F},v)\rho \textbf{ then } \mathcal{T}[\![e_2]\!](b,v)\rho \textbf{ else } \mathcal{T}[\![e_3]\!](b,v)\rho \\
\begin{array}{l} \mathcal{T}[\![\textbf{case } e_1 \textbf{ of } \mathrm{Nil} \Rightarrow e_2 \\ \qquad\qquad [\!]\ \mathrm{Cons}(a,r) \Rightarrow e_3]\!](b,v)\rho \end{array} &= \begin{cases} \left.\begin{array}{l} \textbf{case } \mathcal{C}[\![e_1]\!]\rho \textbf{ of } \mathrm{Nil} \Rightarrow \mathcal{T}[\![e_2]\!](b,v)\rho \\ \qquad [\!]\ \mathrm{Cons}(a,r) \Rightarrow \mathcal{T}[\![e_3]\!](b,v)\rho \end{array}\right\} \text{if } \mathcal{T}ail[\![e_1]\!]\,\rho \neq \{\mathrm{Nil}\} \\ \left.\begin{array}{l} \textbf{let } x = \mathcal{C}[\![e_1]\!]\rho \\ \textbf{in if } \mathrm{nullV}(x) \textbf{ then } \mathcal{T}[\![e_2]\!](b,v)\rho \\ \quad \textbf{else let } c = \mathrm{elemV}(x),\ s = \mathrm{tailV}(x) \\ \qquad \textbf{in } \mathcal{T}[\![e_3]\!](b,v)\rho[c/a,s/r] \end{array}\right\} \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 4: Term Translation (Phase I)

$$
\begin{array}{lcl}
\mathcal{S}ize[\![x]\!]\,\rho & = & \text{if } x \in \rho \text{ then } \rho(x) \text{ else lenV}(x) \\
\mathcal{S}ize[\![\lambda x.\, e]\!]\,\rho & = & \lambda x.\, \mathcal{S}ize[\![e]\!]\,\rho \\
\mathcal{S}ize[\![e_1\, e_2]\!]\,\rho & = & (\mathcal{S}ize[\![e_1]\!]\,\rho)\,(\mathcal{S}ize[\![e_2]\!]\,\rho) \\
\mathcal{S}ize[\![(e_1, e_2)]\!]\,\rho & = & (\mathcal{S}ize[\![e_1]\!]\,\rho, \mathcal{S}ize[\![e_2]\!]\,\rho) \\
\mathcal{S}ize[\![\text{Cons}]\!]\,\rho & = & \lambda(a,r).\, 1 + \mathcal{S}ize(r)\,\rho \\
\mathcal{S}ize[\![\text{Nil}]\!]\,\rho & = & 1 \\
\mathcal{S}ize[\![\text{cataL}(\lambda().\, e_1,\ \lambda(a,r).\, e_2)]\!]\,\rho & = & \text{cataN}(\lambda().\, \mathcal{S}ize[\![e_1]\!]\,\rho,\ \lambda n.\, \mathcal{S}ize[\![e_2]\!]\,\rho[n/r, \perp/a]) \\
\mathcal{S}ize[\![\textbf{let } x = e_1 \textbf{ in } e_2]\!]\,\rho & = & (\lambda z.\, \mathcal{S}ize[\![e_2]\!]\,\rho[z/x])\,(\mathcal{S}ize[\![e_1]\!]\,\rho) \\
\mathcal{S}ize[\![\textbf{let fun } f\, x = e_1 \textbf{ in } e_2]\!]\,\rho & = & (\lambda z.\, \mathcal{S}ize[\![e_2]\!]\,\rho[z/x, \perp/f])\,(\mathcal{S}ize[\![e_1]\!]\,\rho) \\
\mathcal{S}ize[\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!]\,\rho & = & \overline{\max}(\mathcal{S}ize[\![e_1]\!]\,\rho, \mathcal{S}ize[\![e_2]\!]\,\rho) \\
\mathcal{S}ize[\![\textbf{case } e_1 \textbf{ of } \text{Nil} \Rightarrow e_2 & = & \overline{\max}(\mathcal{S}ize[\![e_1]\!]\,\rho, \mathcal{S}ize[\![e_2]\!]\,\rho[(\mathcal{S}ize[\![e_1]\!]\,\rho - 1)/r]) \\
\qquad\qquad [\!]\ \text{Cons}(a,r) \Rightarrow e_3]\!]\,\rho & &
\end{array}
$$

Figure 5: Size Estimation

But there are cases where we cannot estimate the size, such as

$$
\begin{array}{rcl}
\mathcal{S}ize[\![\text{flatten}(x)]\!]\,\rho & = & \mathcal{S}ize[\![\text{cataL}(\lambda().\, \text{Nil},\ \lambda(a,r).\, \text{cataL}(\lambda().\, r,\ \lambda(b,s).\, \text{Cons}(b,s))\ a)\ x]\!]\,\rho \\
& = & \text{cataN}(\lambda().\, 1,\ \lambda n.\, \text{cataN}(\lambda().\, r,\ \lambda m.\, 1 + m)\ \perp)\ (\text{lenV}(x)) \ = \ \perp
\end{array}
$$

where $\perp$ was substituted for a.

There are two integer operations used in Figure 5, namely addition and integer catamorphism. In many cases, each integer catamorphism can take the following form:

$$\text{cataN}(\lambda().\, b,\ \lambda i.\, k \times i + m)\ n$$

when we factor out all occurrences of i, where k and m may be integer expressions that contain other cataN or calls to lenV. Let $a_n = \text{cataN}(\lambda().\, b,\ \lambda i.\, k \times i + m)\ n$. Then we obtain the finite series a_n with $a_0 = b$ and $\forall i :\ 0 < i < n :\ a_i = k \times a_{i-1} + m$. The solution of this finite series is

$$
a_n \ = \ \begin{cases} b + n \times m & \text{if } k = 1 \\ k^n \times b + \frac{m \times (k^n - 1)}{k - 1} & \text{otherwise} \end{cases}
$$

For example, for the size of append(x,y) we have $k = 1$, $m = 1$, $b =$ length(y), and $n =$ length(x). Thus, $\mathcal{S}ize[\![\text{append(x,y)}]\!]\,\rho = \text{length(y)} + \text{length(x)}$.