# Generalized Recursive Structure Combinators

Tim Sheard
Computer and Information Science Department
University of Massachusetts

COINS Technical Report 89-26

April 3, 1989

# Generalized Recursive Structure Combinators *

Tim Sheard

Department of Computer and Information Science
University of Massachusetts, Amherst, MA 01003

April 3, 1989

**Abstract**

A recursive type describes a data structure that contains substructures of the same type as itself. Such types are often given the semantics of the least fixpoint of a recursive type equation. Manipulation of the instances of recursive types are often expressed as recursive functions. Such functions can be quite complex, especially for types needed to model complex objects found in many modern applications. We show how the definition of these manipulations can be simplified by the automatic definition of a class of general combinators of which the usual mapping and reduction functions of lists are special cases. These combinators are automatically defined as a byproduct of type declaration.

Languages which support the definition of recursive structures can easily be extended to define these general combinators automatically. This extension greatly simplifies the specification and implementation of complex algorithms dealing with these structures, while maintaining support for a functional programming style.

## 1 Introduction

Recursive types are those types whose instances have subcomponents of the same type as themselves. For example, binary trees are a recursive type, since the left and right subtrees of a binary tree are themselves trees.

Recursive types are usually defined as the least fixpoint of recursive type equations [1], and constitute an important class of types. Many important data types, such as lists, stacks, trees, etc., can be defined as recursive data types. Such structures are generally manipulated by recursive functions. The definition of these

*This research was supported by NSF grants DCR-8503613 and IST-8606424 and by the Office of Naval Research University Research Initiative contract number N00014-86-K-0764.

functions can be tedious and error prone, especially if a type has many recursive subcomponents occurring in different contexts.

In this paper we describe an implementation of recursive types that, unlike any other implementation we are aware of, automatically generates a class of functions (tailored to each recursive type) which provide for very simple specification and implementation of complex operations on values of these types. We call such functions *generalized recursive structure combinators.*

Consider for a moment a type for complex machine descriptions. A machine is made of parts. Every part has a name, a weight and a cost. Some parts are themselves composed of parts. Thus part descriptions will be the building blocks for composing machine descriptions. Part descriptions will be of two basic forms: descriptions for base parts having components name, cost, and weight; and for composite parts with name, assembly cost, and a list of subparts. We assume that the assembly cost of a composite part includes only the cost to put the subparts together, not the cost of the subparts themselves.

How does one write functions which perform the following queries and update?

1. What is the total cost of a part?
2. What are all the subparts (both composite and base) of part P?
3. What is the increased cost of replacing parts named N, with part Y, in part P?
4. Update every subpart of P, having name N, by adding a new subpart S to each of them.

Each of these can be realized by a recursive function which answers the query or performs the update. These functions are naturally expressed recursively since the computation for any part requires the same computation to be performed on that part's subparts (if it has any).

What is the nature of the recursion? Detailing the correct form for the recursion is complicated because the subparts component is a list of parts rather than just a part. The form of the recursion would be even more complex if a part description had several other components that were records, arrays, or other composite types containing part descriptions. Expressing such recursion correctly can be tedious, time consuming, and error prone. Fortunately the recursive equation that defines the type uniquely determines the form of the recursion. Function templates that express this recursion can be automatically generated.

This paper identifies a class of function templates (in terms of combinators which take other functions as input) which are particularly useful in that they are widely applicable, (the list mapping and reduction functions are special cases),

can be automatcially generated, and provide a concise yet surprisingly transparent and understandable mechansim for describing a wide class of recursive structure manipulations.

These function templates can be most easily understood by considering that every instance of a recursive type could be modelled as a tree. Each node in the tree represents the non-recursive components of the type. Each recursive component corresponds to an arc from node to node. To perform a computation on such a structure a "walk" of the tree must be performed. The control mechanism to coordinate such a walk is different for every recursive type, but is determined by the recursive type equation defining the type.

In general a walk computes some value from a node, N, and then combines that value with the values obtained by recursively walking all the nodes reachable through the arcs originating at N. Reiterating, there are three basic components to such a walk.

1. Computing the value from each node.

2. Coordinating the recursive walks from every arc originating at the node.

3. Combining the computed value with the values from the recursive walks.

The system we describe provides automatic assistance for specifying some or all of the three components. Our general recursive structure combinators coordinate the recursive walks and are automatically generated from a type's defining equation.

If the combining function combines the values into another instance of the same type as the original tree, we call such a walk a *map*. The combining functions for maps are also automatically generated.

If the combining function combines the values into a scalar value or any other type other than the type of the original tree, we call that walk a *reduction*. The user must specify the combining function for reductions.

Computing the value from each node differs depending upon what the particular walk was intended to compute. Our system simplifies this by providing an expressive pattern directed interface.

The rest of this paper expands upon these ideas. Section 2 describes the notation for recursive type equations used to define recursive types. Section 3 describes the use of *patterns* to specify simple structure manipulations. Section 4 describes the functions and combinators we will use to define function templates and describes the algorithms we use to generate the templates from the type equations. Section 5 gives examples in which the structure combinators are used to express the answers to complex queries and updates clearly and concisely.

## 2 Notation

Recursive types are those types that have as their semantics the least fixpoint of recursive type equations. For example lists satisfy the recursive type equation:

```
list(alpha) = nil | alpha × list(alpha)
```

Where | is the disjunctive operator between types, × is the conjunctive operator (or cross product) amongst types, and alpha is a universally quantified type variable.

A *list(alpha)* is then the disjunction of *nil* and the cross product of types *alpha* and *list(alpha)*. Because of the type variable, *alpha*, the above equation actually represents a large class of recursive type equations, one for each instantiation of the type variable *alpha*. Thus *list(number)* and *list(string)* define discrete types, where the types of the items in the lists are *numbers* and *strings* respectively.

In parameterized type equations where the lefthand side of the equation contains a type variable, we will assume the type variable is universally quantified, (in much the same way one assumes the formal parameter of a function definition can take on any value).

$Function(t_1 \times t_2 \times ...t_n) \rightarrow t(n+1)$ denotes the type of a function from the cross product of ($t_1$, $t_2$, ... $t_n$) to $t(n+1)$. $Function() \rightarrow t_n$ is one way to type a constant in type $T_n$.

A *record* type is a labelled cross product type with named constructor and named selectors. The record with type $(t_1 \times t_2 \times ...t_n)$, that has named constructor $c$, and named selectors $s_1, s_2...s_n$ is denoted as $C[s_1 : t_1, s_2 : t_2, ...s_n : t_n]$. Note that such a expression also denotes that $C$ is a function with type $function(t_1 \times t_2 \times ...t_n) \rightarrow t_1 \times t_2 \times ...t_n$, and that each $s_i$ is a function with type $function(t_1 \times t_2 \times ...t_n) \rightarrow t_i$.

A *discriminated union* type is a labeled disjunction. The discriminated union with type $(t1|t2|...tn)$, that has named discriminants $r_1, r_2...r_n$ is denoted as $\{r_1 : t1, r_2 : t2, ...r_n : t_n\}$. Note that such an expression also denotes that each $r_i$ is a function with type $function(t_1|t_2|...t_n) \rightarrow boolean$, which is used to determine the particular alternative type of an instance of the discriminated union.

A function definition is denoted by an equation of the form *f(x) = value-expression*, and a type definition is denoted by an equation of the form *T(x) = type-expression.*

Using this notation we can refine the recursive type equation for lists which we call *linearlists* to distinguish them from LISP lists as follows:

```
linearlist(alpha) = { lnull:lnil,
                      lconsp:lcons[lfirst:alpha,
                                   lrest:linearlist(alpha)] }
```

`Lnil` is the constructor function with type *function*() → *list*(*alpha*). It is a nullary function and hence a constant. In our implementation we will define a constant object as well as a nullary function for bottom objects. Hence `lnil()` the nullary function call, and `lnil` the constant denote the same value. `Lcons` is the constructor function with type *function*(*alpha*×*list*(*alpha*)) → *list*(*alpha*), `Lfirst` and `Lrest` are the selector functions with types *function*(*list*(*alpha*)) → *alpha*, and *function*(*list*(*alpha*)) → *list*(*alpha*). The discriminants are `lnull` and `lconsp` and have type *function*(*list*(*alpha*)) → *boolean*.

Some recursive types may not be parameterized types, in which case the parameter on the lefthand side of the equality is omitted.

In general the right-hand side of a recursive type equation does not have to be a union of types. But a union (where at least one of the alternatives is not recursive) must occur somewhere in the type equation or the type will have no finite instances. In a non-lazy evaluation environment this creates problems. We will not consider the case where the right-hand side of the equation is not a union. Also, in our implementation, a recursive type may not refer to an undefined type (other than its type parameter) in its defining equation. This disallows mutually recursive type definitions.[1]

Recursive types may refer to previously defined recursive types. We will illustrate this by defining the internal representations of statements of a small imperative language. Such representations might be useful in a compiler or interpreter, and later we will show how they can be used in symbolic processing of expressions in the language. A statement can either be an *if* statement, a *while* statement, an *assignment* statement, or a *compound* statement. Several of these (if, while, and compound) are records having components that are themselves statements. Because a compound statement is not a record type our implementation requires the user to supply the name of a *coercion* function which is used to coerce an ordinary linearlist of statements into the compound alternative of the statement type.[2]

```
stmt(exp) = { compoundp: linearlist(stmt(exp)) COERCE BY begin,
              assignp:   assign[leftvar:symbol, rightval:exp],
              ifp:       ifc[iftest:exp, then:stmt(exp), else:stmt(exp)],
              whilep:    while[wtest:exp, body:stmt(exp)] }
```

Here `begin` is a coercion function with type *function*(*list*(*stmt*(*exp*))) → *stmt*(*exp*). In this definition we have chosen the expression type to be a parameter to the stmt type definition.

[1]Mutual recursion can generally be avoided by constructing a single type comprised of an alternative for each of the original mutually recursive types.

[2]For record types the constructors play the role of both constructor and coercion function.

Each alternative to the union is implemented by defining its recognizer function. The discriminant of each alternative is used to name these functions. In the statement example the recognizers are `compoundp`, `assignp`, `ifp`, and `whilep`. Given any stmt, the user can test what alternative form it is by using these functions. Each alternative of the recursive structure type that is a record type has its constructor function and selector functions defined as well. Non record types have their coercion functions defined. Thus statements are built by the functions `begin`, `assign`, `ifc` and `while`. Given an `ifp` alternative, `x` we can decompose it into its subcomponents using its selectors `iftest`, `then` and `else`, e.g., `iftest(x)` returns the test subcomponent of the if statement `x`.

# 3 Manipulating Recursive Structures

In addition to constructing and decomposing operations, recursive structure objects require some mechanism for taking different actions depending upon which of the alternatives the object is. This can be done with a case-like statement, or by some sort of pattern-based mechanism.

## 3.1 The Recursive Case Expression

Our implementation provides the *rec-case* expression. This expression takes different actions for each of the type's alternatives. The syntax for a rec-case statement is.

```
rec-case form
       { discriminant1 =>  action1;  ... ;
         discriminantN =>  actionN }
```

`Form` is any expression with a value that is a recursive structure having discriminants `discriminant1` to `dicriminantN`. The discriminant `others` can also be used, the action for `others` will be repeated for all the missing alternatives. For example, if `x` has type *stmt(exp)*, the rec-case

```
rec-case x
  compoundp => length(x);
  assignp   => rightval(x);
  others    => 4
  end
```

transforms into something equivalent to the following (though the actual form may be optimized).

```
if compoundp(x)       then length(x)
   elseif assignp(x) then rightval(x)
   elseif ifp(x)      then 4
   elseif whilep(x)   then 4
   else               ERROR
```

## 3.2 Pattern Directed Manipulation

The recursive case statement can be awkward to use when the object being inspected is complicated. For example, consider a recursive type that models expressions. An expression is either a binary operator with expressions as operands, a unary operator, a constant, or a variable.

```
expr = { variablep  string COERCE BY var;
         constp     number COERCE BY const;
         binaryp    pair[left:expr, bin-op:string, right:expr];
         unaryp     prefix[unary-op:string, unary-operand:expr] }
```

Using this recursive type the infix expressions on the left could be represented by the value of the expressions on the right.

```
 (X+Y)              pair( var("x"), "+", var("y") )

 (z*y+3)            pair( pair(var("z"),"*",var("y")), "+", const(3) )

 (0 + (x * 1))      pair( const(0), "+", pair(var("x"), "*", const(1))
)

 (0 * (X + 0))      pair( const(0), "*", pair(var(x), "+", const(0)) )
```

Suppose we wished to determine if some expr, x, had the form (0 + Y) where Y could be any other legal expression, (0 + (Z + 2)) or (0 + A) for example. The recursive case statement for this is

```
rec-case x
   { binaryp => if bin-op(x) = "+"
                then rec-case left(x)
                       { consp => (= x 0);
                         others => false }
                else false;
     others  => false }
```

A rather clumsy statement for a very simple idea. Pattern directed manipulation allows a pattern to be matched against a value and different actions to

be taken depending upon whether the pattern matches the value. This provides a simple means of specifying alternate actions. These actions can be based upon values determined during the pattern matching. If a pattern contains a variable, then that variable is bound to the corresponding part of the value being matched. This variable can be used in the action with its pattern-determined value. This simplifies the expression of complex case based computations even further.

Many functional languages such as ML[2] supply pattern directed manipulation of objects. Here we will use pattern directed manipulation with pattern expressions of the form:

```
pattern form
  { pattern1 => action1;
    pattern2 => action2;
    ...
    patternN  => actionN }
```

The semantics of the pattern statement is to match the patterns from left to right, against `form`, returning the value of the action for the first pattern that matches. If no pattern matches a runtime error occurs. Remember pattern matching is also a binding construct. Variables are bound in a successful pattern match, and may be referenced in the pattern's matching action. Patterns obey the following rules.

1. Any constant is a pattern. Constants of the primitive types string and number are patterns. For example `"A string"`, and 45, are patterns. Bottom objects such as `Lnil` are also patterns. Constant patterns match equal constant objects.

2. Any constructor or coercion function call, where all of its arguments are legal patterns is a pattern. For example `lcons(4, lnil)` is a pattern.

3. A variable is a pattern. Variables match any value and are used to bind values that are subpatterns in other patterns. The pattern `lcons(4, x)`, for example, matches any list with lfirst component equal to 4. The value of the lrest component can be anything and is bound to x. The bindings of variables are available in a pattern's matching action.

4. The wildcard pattern is a pattern. * is the wildcard. Wildcards are like variable patterns except that no binding is done. * matches any expression and the value it matches is thrown away, i.e. not available for reference in the action paired with the pattern.

5. A value pattern is a pattern. The value pattern format is `@form`. A value pattern matches if the object matching against it is equal to the value of form. For example `@y` is a pattern. It matches values that are equal to the

value of the variable, y, in the environment where the match takes place. For example if y had value 7 then @y matches 7. Value patterns do not bind values.

Patterns are very useful when asking questions about recursive structures. A statement that tests if an expr, x, has the form (0 + X) would be written as follows:

```
pattern x
  { pair(const(0), "+ ", x) => true;
    *                       => false }
```

Note the catchall pattern * as the pattern of the last pattern action pair of the pattern statement above. Since a variable pattern matches any expression this makes sure some value is returned for every input. Since this is also a wildcard pattern the value it matches is thrown away, i.e. not available in the corresponding action expression.

It might be assumed that languages that use pattern directed manipulation of structures will need to carry the run-time overhead of unification in order to determine whether patterns unify with the values they are matched against. Fortunately this is not the case. Patterns built by the rules above can be completely resolved and translated at compile time to very efficient code [3]. For example the pattern action pair below which would appear in a *pattern* statement.

```
assign(var, pair(x, "+", @n))  => g(x,var)
```

translates to the clause:

```
IF assignp(z) and
   LET z-3 = rightval(z)
       IN
       ( binaryp(z-3) and
         bin-op(z-3) = "+" and
         right(z-3) = n )
   THEN g(left(z-3),leftvar(z))
   ELSE fail
```

Suppose we are matching some value, z, against the pattern above, then z must be an assignp structure. Its rightval must be a binaryp structure whose bin-op is +, and whose right operand must be equal to the value in the variable n (from the value pattern). Z's leftvar (var in the pattern), and rightval(z)'s left operand (x in the pattern) can be any value, but they must be "bound". If all these things are true, the value the function g applied to the left operand and the leftvar is returned. This value is returned due to g(x,var) being the action part of the pair. Note that

in the translation variables are not actually bound, but that the correct values are substituted in the action. If any of the pattern tests fail the unique value *fail* is returned which causes the *pattern* statement to move onto the next alternative.

Patterns are concise specifications for matching complex structures and have efficient compilations. They are natural for use with recursive structures, and we will use them in the rest of this paper to describe our computations.

# 4 Generalized Structure Combinators

Each recursive structure definition determines a set of generalized structure combinatorss. As mentioned in the introduction generalized structure combinators come in two classes, maps and reductions. Maps take a recursive structure and return another recursive structure. Reductions take a recursive structure and return (most often) a single unstructured value. We have divided reductions and maps themselves into two varieties, parameter and recursive. In a broad sense a parameter map or reduction accesses only those subcomponents of each node of the "tree" that have the parameter type, while a recursive map or reduction accesses all the information at each node. For non-parameterized recursive types parameter combinators are not possible, and are not defined.

## 4.1 Parameter Combinators

Suppose we have a parameterized recursive structure like *linearlist(alpha)*, where *alpha* is the type parameter. A parameter combinator on a value of this type would act on every component having type *alpha*. For example *mapcar*[4], the LISP map that returns a list (where every element in the input list is transformed by the mapped function), is a parameter map because the only subcomponents transformed are of parameter type. The recursive structure parameter map corresponding to mapcar is the linearlist parameter map. It accepts two inputs a recursive structure of type *linearlist(alpha)* and a function with type *function(alpha)* $\rightarrow$ *beta* and returns an object of type *linearlist(beta)*. It is automatically defined when the linearlist type is defined. It has the definition below.

```
linearlist-parameter-map(x,tf) =
PATTERN x
 {              lnil => x;
   lcons(first,rest) => lcons(tf(first),
                              linearlist-parameter-map(rest,tf)) }
```

The function decomposes each linearlist node into its subnodes, transforms the parameter component (matched by the pattern variable `first`), using the transform

function, `tf`. Then it transforms the recursive component (matched by the pattern variable `rest`) using recursive calls, and then reconstructs a new node using the constructor `lcons`. When linearlists are defined, *linearlist-parameter-map* is also automatically defined. It is equivalent to the function above (though it may be optimized). It is the linearlist equivalent to LISP's mapcar.

The algorithm for generating the parameter map from the recursive type equation can be stated as follows. Each alternative of the union in the type equation generates a pattern action pair. For alternatives with no recursion and no parameter, the action is simply the unchanged object. For each recursive component generate a recursive call, for each parameter object generate an application of the transform function, all other components are unchanged. Note the correspondence between the recursive equation and the parameter map.

```
linearlist(alpha) = { lnull:lnil,
                      lconsp:lcons[lfirst:alpha,
                                   lrest:linearlist(alpha)] }


linearlist-parameter-map(x,tf) =
PATTERN x
 {              lnil => x;
   lcons(first,rest) => lcons(tf(first),
                              linearlist-parameter-map(rest,tf)) }
```

Not all recursive typea are so simple. In the general case a recursive structure may

1. have more than one alternative that is recursive,
2. have alternatives containing more than one subcomponent of parameter type,
3. have alternatives containing many subcomponents of recursive type.

In the first case each alternative is reconstructed after its transformation by its own constructor (or coercion) function. In the second case, all parameter subcomponents are transformed by the transform function, `tf`, and in the third case every recursive subcomponent is transformed by a recursive call.

## 4.2 Recursive Maps

While the parameter map for lists, *mapcar*, is well known, the more general map which maps over the "whole node" rather than just the parameter is relatively unknown. A recursive map transforms each "node" in the recursive structure into another form. For each node this transformation can be done before its recursive subcomponents are transformed, (in which case we call it a *pre-map*) or after its

recursive subcomponents are transformed, (in which case we call it a *post-map*). In the parameter map for linearlists the transform function, `tf`, is applied to the `lfirst` component since it has parameter type.

A recursive map for linearlists would apply the transform function to the "whole node". The linearlist recursive pre-map has the following definition.

```
linearlist-rec-pre-map(x,f) =
let new = f(x) in
  PATTERN new
   { lcons(first,rest) => lcons(first,
                                linearlist-rec-pre-map(rest,f))
                  lnil => new }
```

Here `x` has type *linearlist(alpha)* and `f` has type *function(linearlist(alpha))* $\rightarrow$ *linearlist(alpha)*. Note that the application of the transform function `f`, is done before the recursive mapping, by using the let clause. The result, `new`, is then decomposed into its parts, the recursive components being transformed by recursive calls, the other parts being left alone. The whole node is then reconstructed and returned. Note that it is possible for the transform function, `f`, to return an alternative form with no recursive subcomponents (such as `lnil`) and hence short circuit some of the recursion.

In a post-map the transform function, `f` is applied after the recursive mapping. Thus the short circuiting is not possible.

```
linearlist-rec-post-map(x,f) =
f(PATTERN x
   { lcons(first,rest) => lcons(first,
                                linearlist-rec-post-map(rest,f));
                  lnil => x })
```

In the general case both a pre- and post-transform function should be supplied. If only a pre-transform is needed the post-transform can be the identity function, or vice-versa. The general recursive map function actually defined by the linearlist recursive type definition is

```
linearlist-recursive-map(x,f,g) =
g(let new = f(x) in
   PATTERN new
    { lcons(first,rest) => lcons(first,
                                 linearlist-recursive-map(rest,f,g));
                  lnil => new })
```

Here, f is the pre-transform, and g is the post-transform.

Recursive maps greatly simplify the implementation of functions that operate on recursive types. Consider a function that reduces an expression to a simpler form using the identities 0+x = x, 1*x = x, and 0*x = 0. (This might be useful in simplifying the output of a symbolic differentiation function, for example.)

Note that in applying the identity 0+x = x, it would be necessary to recursively simplify the term represented by x, since this term might be further reducible by the identity. For example 0 + (0 + y) reduces in two steps to y. But for the simplification of the term 0*y, simplification of y is wasted since the result of the reduction is 0 no matter what y simplifies to. Thus a function that applied the rule 0*y = 0 is a good candidate for a pre-transform function, and the rule 0+x = x is a good candidate for both a pre and post-transform.

```
zm(x) =
      PATTERN x
         pair(const(0),"*",*) => const(0);
                              y  => y
      end
```

The function zm (for zero multiply) checks to see if its argument, x, is a binaryp alternative of the expr type (i.e. built with the pair constructor), if its operator is a multiply (*), and if the left term is the constant 0. If all these conditions hold then zm returns the constant 0 without consulting the right term (hence the use of the wildcard pattern *), otherwise it returns x unchanged by using a variable, y, as a pattern (which matches anything) and returning y which has been bound to x's value as part of the matching.

```
za(x) =
      PATTERN x
       { pair(const(0),"+",x) => x;
                              y => y }
```

The function za (for zero add), similiar to zm, applies the rule 0+X = X. Some example applications of these functions and the result (to the right of the ==>) are:

```
zm(pair(const(0),"*",pair(var("x"),"+",const(0)))) ==>
        const(0)

zm(pair(const(0),"+",pair(var("x"),"*",const(1)))) ==>
   pair(const(0),"+",pair(var("x"),"*",const(1)))

za(pair(const(0),"+",pair(var("x"),"*",const(1)))) ==>
                     pair(var("x"),"*",const(1))
```

To recursively apply these two rules to all levels of an expression we can now use the expr-recursive-map defined when the expr type is defined.

```
expr-recursive-map(pair(const(0),"*",pair(var("x"),"+",const(0))),
                   zm,
                   za)   ==>      const(0)
```

The expr-recursive-map maps zm (the pre-map transform) first over the input (0 * (X + 0)), returning 0, without ever looking deeper into the structure at (X + 0). Suppose we had instead used (0 + (0 * X)) as the actual parameter.

```
expr-recursive-map(pair(const(0),"+",pair(const(0),"*",var("x"))),
                   zm,
                   za)   ==>      const(0)
```

The pre transform, zm, does not initially apply, but on the recursive call to the right operand of the +, (0 * X), it applies and returns 0. The post-transform then deals with (0 + 0), and returns 0.

Finally, consider ((0 * X) * (Y + 0)). Here the pre transform for 0*X=0 is not initially applicable. The recursive transform on (0 * X) gives us (0 * (Y + 0)). The 0 on the left of the multiply (*) is available only for the post transformation.

Another problem manifests itself in the subterm (Y+0) because the 0+X=0 transform does not apply since the right and lefthand sides are reversed. An implementation that deals with both the commutativity and late appearance of 0 is as follows.

```
simplify(x) =
   PATTERN x
    { pair(const(0),"*",*) => const(0);
      pair(*,"*",const(0)) => const(0);
      pair(const(0),"+",x) => x;
      pair(x,"+",const(0)) => x;
                         y => y }

expr-recursive-map(pair(pair(var("x"),"*",const(0)),
                        "*",
                        pair(var("y"),"+",const("0"))),
                   simplify,
                   simplify)   ==>  const(0)
```

Using the same function as both the pre- and post-transform is not uncommon. This represents a rather concise, elegant solution to a rather difficult problem. This solution is easy to modify as well. The addition of the rule (1*X) = X, can be added by simply adding two more patterns to the simplify function.

### 4.2.1 Recursive Maps With Recursive Components

A recursive type is often defined in terms of a previously defined recursive type. The stmt type is an example, since it has a simple alternative defined in terms of linearlists. The parameter and recursive maps for these types have slight complications that need to be discussed. Recall the definition of the stmt type:

```
stmt(exp) = { compoundp: linearlist(stmt(exp)) COERCE BY begin
              assignp: assign[leftvar:symbol, rightval:exp],
                  ifp: ifc[iftest:exp, then:stmt(exp), else:stmt(exp)],
               whilep: while[wtest:exp, body:stmt(exp)] }
```

The parameter map for stmts is as follows:

```
stmt-parameter-map(x,tf) =
   PATTERN x
     {       begin(n) => begin(linearlist-parameter-map
                                 (n,lambda(z)stmt-parameter-map(z,tf)));
          assign(*,*) => x;
  ifc(test,then,else) => ifc(tf(test),
                             stmt-parameter-map(then,tf),
                             stmt-parameter-map(else,tf));
     while(test,body) => while(tf(test),stmt-parameter-map(body,tf)) }
```

Note that in the definition of stmt-parameter-map, the action for the pattern with the coercion function `begin` includes a call to `linearlist-parameter-map`. The stmt parameter map is supposed to map the transform function, `tf`, over every subcomponent of parameter type. Since the compoundp alternative is a linearlist of stmts, each stmt in that list needs to be transformed. The control mechanism to accomplish that mapping is the linearlist parameter map function defined when the recursive type *linearlist* was defined. The function that `linearlist-parameter-map` will map over each of the elements in the list is `stmt-parameter-map` (the function being defined). Since the transform function must be a function of one argument, we lamdba abstract it by fixing its transform map to `tf`.

For *recursive* maps on types defined in terms of previously defined recursive types, a similiar strategy must be employed. The transform on each "node" must transform the whole node rather than just the parameter components. Consider the following definition.

```
stmt-recursive-map(x,f,g) =
      g(PATTERN f(x)
        {   begin(n) => begin(linearlist-parameter-map
                                (n,lambda(z)stmt-recursive-map(z,f,g)));
```

```
      assign(*v,*e) => assign(*v *e);
    ifc(test,th,el) => ifc(test,
                           stmt-recursive-map(th,f,g),
                           stmt-recursive-map(el,f,g));
   while(test,body) => while(test,stmt-recursive-map(body,f,g))     }
)
```

In this example each stmt in the compoundp list must be transformed, so the linearlist parameter map is again used to map the function being defined (or at least an appropriate lambda abstraction of it) over each stmt in that list. Note that the recursive map uses the linearlist parameter map, not the linearlist recursive map. Parameter maps seem to be "more basic" than recursive maps in this regard, perhaps the reason that list parameter maps such as mapcar are well known, while the recursive map for lists are not.

## 4.3 Recursive Structure Reductions

A *reduction* reduces a complex structured object to a simpler object, often a scalar value. The process of reduction works by using some accumulator[3] function to successively accumulate an answer. For example consider the linearlist of integers.

```
lcons(2,lcons(5,lcons(6,lcons(1,lnil))))
```

We could reduce this using addition (+) by successively applying + to the lfirst of the list and reducing the the lrest of the list, eventually getting the answer (2+(5+(6+(1+0)))). Note that we need some default (or bottom) value to return when the list is the degenerate `lnil` case, in this case 0. Every recursive structure defines two types of reductions. For each node in the recursive structure the *parameter reduction* adds a value to the accumulation from the parameter substructures, while the *recursive reduction* gets its values from the whole node. In both cases the the recursive substructures need to be recursively reduced. Since the whole structure contains the parameter substructure, parameter reductions could be defined in terms of recursive reductions. The parameter reductions are separately defined because they are often used and simpler to apply.

### 4.3.1 Parameter Reductions

Consider the linearlist parameter reduction.

[3]Traditionally a binary accumulator has been used and we continue this trend, but with complicated recursive types a n-ary accumulator might be of some utility. We have yet to work out the details.

```
linearlist-parameter-reduce(x,acc,app,bottom) =
   PATTERN x
    { lcons(first,rest) => acc(app(first),
                           linearlist-parameter-reduce(rest,
                                                       acc,
                                                       app,
                                                       bottom));
                  lnil => bottom }
```

Here x has type *linearlist(alpha)*. Acc is the binary accumulator with type $function(beta, gamma) \rightarrow gamma$. App is a unary application function that selects a value from each parameter substructure and feeds it to the accumulating function as its first parameter. It has type $function(alpha) \rightarrow beta$. Finally bottom is an object of type *gamma*. Parameter reduction functions work as follows. For each alternative without a parameter substructure (the lnil pattern above for linearlists) the function returns the bottom object. For each alternative that contains a parameter substructure, the application function computes a value for that substructure (app(first) in the linearlist example above). Finally the accumulating binary operator acc is used to accumulate the result of the call to app and the result of recursively reducing the recursive subcomponents.

The use of an application function is not strictly necessary since a more complicated binary accumulator could do the transformation on its first argument. Since the binary accumulators are often well known operators such as addition, it is strictly for the ease of the user that the application function is a parameter. This frees the user from having to make complicated lambda abstractions as the accumulating binary operator.

For recursive structures with more than one parameter subcomponent to an alternative, several calls to the accumulating binary operator are nested one inside the other. Structures with more than one recursive subcomponent to an alternative, nest the recursive calls themselves, where the base object for the outermost call is a recursive call, and the innermost call uses the actual base object. For example, note in the definition of stmt-parameter-reduce below, how the base object for the recursive call for the then part of an ifp clause is a recursive call for the else part.

```
stmt-p-reduce(x,acc,app,bottom) =
   PATTERN x
          begin(n) => linearlist-parameter-reduce
                        (n,
                         lambda(x,y)stmt-p-reduce(x,acc,app,y),
                         identity,
                         bottom);
```

```
    assign(*,rv) => acc(app(rv),bottom);
  ifc(test,th,el) => acc(app(test),
                         stmt-p-reduce(th,acc,app,
                                       stmt-p-reduce(el,acc,app,bottom)));
 while(test,body) => acc(app(test),stmt-p-reduce(body,acc,app,bottom))
  end
```

The stmt parameter reduction also nicely illustrates reductions on recursive types with subcomponents that have types that are previously defined recursive types. The previously defined type's *parameter* reduction is used to reduce those subcomponents. See the action of the begin pattern in the example above. Since this is a reduction we need to specify both an accumulator and an application function. We use an appropriate lambda abstraction of the function being defined as the accumulating binary operator, and the identity function as the application function. This strategy will have to be slightly modified when we consider recursive reductions.

A linearlist of integers, x, can be reduced to the sum of all its elements by the the following call.

```
linearlist-parameter-reduce(x, +, identity, 0)
```

A stmt of expressions, x, can be reduced to a linearlist of all its expressions by the call.

```
stmt-p-reduce(x, lappend, identity, lnil)
```

### 4.3.2 Recursive Reductions

Recursive reductions are similar to parameter reductions, except that the application function is a function from the recursive type (the whole node rather than just the parameter subcomponents) to the accumulator's first parameter's type. Consider:

```
linearlist-recursive-reduce(x,acc,app,bottom) =
acc(app(x),PATTERN x
            lcons(*,rest) => linearlist-recursive-reduce
                                 (rest,acc,app,bottom);
                     lnil => bottom
          end)
```

Here x has type *linearlist(alpha)*. Acc is the binary accumulator with type *function*(*beta*, *gamma*) $\rightarrow$ *gamma*. App is a unary application with type *function*(*linearlist*(*alpha*)) $\rightarrow$ *beta*. Finally bottom is an object of type gamma.

Recursive reduction works as follows: the application function is applied to the whole node to get a value (the app(x) term above). That value is then accumulated with the result of recursively reducing each recursive subcomponent (the

pattern statement above). Alternatives with no recursive subcomponent return the bottom object as their contribution to the reduction (the `lnil` pattern action pair).

Recursive types with previously defined recursive subcomponents or multiple recursive subcomponents in one alternative follow the same pattern as the parameter reduction functions, as explained above. For example consider:

```
stmt-recursive-reduce(x,acc,app,bottom) =
acc(app(x),PATTERN x
            begin(n) => linearlist-parameter-reduce
                          (n,acc,
                           lambda(x)stmt-recursive-reduce
                                       (x,acc,app,bottom),
                           bottom);
          assign(*,*) => bottom;
         ifc(*,th,el) => stmt-recursive-reduce
                           (th,acc,app,stmt-recursive-reduce
                                       (el,acc,app,bottom));
        while(*,body) => stmt-recursive-reduce(body,acc,app,bottom)
        end)
```

Again the `begin` pattern's action uses a call to the linearlist parameter reduction, (note that we do not use the linearlist recursive reduction even though we are defining the stmt recursive reduction). Since this is a reduction we need both an accumulating and an application function. The accumulating function is the original accumulating function, `acc`, and a lambda abstracted version of the function being defined is used as the application function. This lambda extracted version is used to extract a value from each stmt in the linearlist of stmts, and `acc` is used to accumulate these to a single value.

Recursive reductions allow easy specification of complex computations. Recall the definition of expr.

```
expr = { variablep  string COERCE BY var;
         constp     number COERCE BY const;
         binaryp    pair[left:expr, bin-op:string, right:expr];
         unaryp     prefix[unary-op:string, unary-operand:expr] }
```

It generates the recursive reduction

```
expr-recursive-reduce(x,acc,app,bottom) =
acc(app(x), PATTERN x
```

```
        var(*) => bottom;
      const(*) => bottom;
   pair(l,*,r) => expr-recursive-reduce
                     (l,acc,app,expr-recursive-reduce(r,acc,app,bottom));
  prefix(*,u) => expr-recursive-reduce(u,acc,app,bottom)
     end)
```

Suppose we wanted to compute a list of all the symbols used as variables in a large expr. We could do this by reducing using the linearlist *union* function as the accumulator, with the following as the application function.

```
symbols(x) =
   PATTERN x
      var(x) => list(x);
           y => lnil
   end

expr-recursive-reduce(x,union,symbols,lnil)
```

The `symbols` function returns a list with a single element for the `var` alternative and `lnil` for the empty set for other alternatives. The call to `expr-recursive-reduce` then `unions` all these singleton sets up, by recursively getting the variables from each "nodes" subexpressions.

An implementation could take one final step to provide easy user access to structure maps and reductions. It should provide a uniform method of accessing the maps and reductions defined for each recursive structure definition. The user should not need to remember the long names given to the functions by the implementation. A good implementation should supply macro like statement which deduces the type of an expression, and expands into a call to the correct function. For example the reduction of x which has type *expr* to obtain all variables in x could be expressed as:

```
R-REDUCE x, union, nil BY { var(x) ==> list(x); * => nil }
```

The r-reduce statement's first argument is the object to be reduced. We deduce its type and expand to a call of the correct recursive reduction. The second and third parameter are the accumulator function and the bottom object respectively. The pattern action pairs after the keyword `BY` specify patterns to construct the application function. The above call would expand to

```
expr-recursive-reduce(x,union,
      lambda(x)PATTERN x { var(y) => list(y); * => nil },nil)
```

Macro statements r-map (for recursive map), p-reduce (for parameter reduction), and p-map (for parameter map), work in a similiar manner. for example

```
P-MAP y IN  x BY (+ 1 y)
```

expands to

```
linearlist-parameter-map(x,lambda(y)(+ 1 y))
```

Note that parameter maps do not use patterns since the parameter type is not always a recursive type. We specify a term, `(+ 1 y)`, and a variable, `y`, to lambda abstract from the term to create a function. `lambda(y)(+ 1 y)`.

The calling sequence for each of our macro-like statements follow:

```
P-MAP       lambdavar IN x BY lambdabody

P-REDUCE    [ lambdavar IN ] x , acc , bottom [ BY lambdabody ]

R-REDUCE    x , acc , bottom BY pattern

R-MAP2      x BEFORE pre-pattern AFTER post-pattern

R-MAP       x BY pattern
```

`R-MAP` and `R-MAP2` differ in that `R-map2` allows the specification of two patterns. One for each of the pre- and post-transform functions. `R-map` uses only one pattern which it uses for both transforms.

Note that parameter macro statements, like `P-map` and `P-reduce`, take a term and a variable (to lambda abstract from it), rather than a pattern to define the application function. In the *p-reduce* statement they are optional. If they are omitted the default values for the lambda-term and the lambda-var cause the application function to abstract to the identity function.

```
P-REDUCE x, +, 0
```

expands to

```
linearlist-parameter-reduce(x,+,lambda(x)x,0)
```

# 5 The Part Subpart Example

We illustrate the use of our techniques by posing and solving several queries and updates on a sample structure. The sample structure is the simple part subpart structure, where some parts are made from other parts (subparts) which we first saw

in the introduction. The example is meant to elicit comparisons with structures in a system modeling the design of complex objects. Each part is typed by the recursive structure below.

```
part = { basep: base[base-name:string,
                     base-cost:number,
                     base-mass:number];
    compositep: composite[comp-name:string,
                          assembly-cost:number,
                          subparts:linearlist(part)] }
```

What follows is a list of examples, each comprising an English language query or update, and a function definition. In the case of a query, the result of calling the function is the answer to the query. In the case of an update, the result of the function is a new value that will replace the original object being updated. We hope to show that the expression of these queries and updates using our recursive structure combinators are both concise (i.e. it specifies complex ideas with little notation) and (unlike the infamous APL one-liners) easy to understand.

What is the total cost of a part?

```
totalcost(p) =
R-REDUCE part, +, 0
   BY {    base(*,cost,*) => cost;
        composite(*,ac,*) => ac    }
```

This solution reduces the part structure by addition. Base parts contribute their costs, composite parts contribute the cost of their assembly. There is no need for the action paired with the composite pattern to attempt to deal with the cost of that composite parts subparts since these are automatically added in by the recursion handled by the `R-reduce` function.

Return a linearlist of all the names of every subpart of part p.

```
all-subpart-names(p) =
   R-REDUCE p, lappend, lnil
      BY {      base(name,*,*) => lcons(name,lnil);
           composite(name,*,*) => lcons(name,lnil) }
```

Some part, with name N, has been redesigned. It now has an additional subpart, S. Write a function that updates every subpart of a part, P, with name N, by adding S, to its subparts list.

```
add-new-subpart(p,n,s) =
 R-MAP p
 BY { composite(@n,ac,sub) => composite(n,ac,lcons(s,sub));
                         x => x }
```

Given a part structure, flatten it into a two dimensional (part subpart) list, that contains all pairs (x y), where x is the name of a composite part in P, and y is the name of a subpart in x's subparts list. First we use patterns to define a polymorphic name function that returns the name of a part regardless of whether it is a base or composite part.

This is an interesting example since it uses two recursive structure combinators. R-reduce is used to reduce the structure into a list of part, subpart pairs, and P-map is used to create a list of pairs for each composite parts subparts list. In the example below pair is a function which creates a pair[4].

```
pname(y) =
PATTERN y base(n,*,*) => n; composite(n,*,*) => n END

flattenpart(p) =
   R-REDUCE p lappend lnil
      BY { composite(name,ac,sub) => P-MAP y IN sub BY
                                        pair(name,pname(y));
                                * => lnil }
   END
```

How many pieces is a part composed of. Do not count composite parts since they are composed of other pieces.

```
countpieces(p) =
   R-REDUCE p, +, 0
       BY { composite(*,*,*) => 0; base(*,*,*) => 1 }
```

[4]A definition for pair might be `pair(x,y) = lcons(x,lcons(y,lnil))`

All the sub parts of part P, that cost more than 1.00

```
expensive-subparts(p) =
   R-REDUCE p, lappend, lnil
       BY { x => IF totalcost(x) > 1.00
                    THEN lcons(name(x),lnil)
                    ELSE lnil }
```

What is the assembly cost of a part. Do not include the cost of the pieces only the cost of assembling it.

```
cost-of-assembly(part) =
   R-REDUCE part, +, 0
       BY {   base(*,*cost,*) => 0;
            composite(*,ac,*) => ac }
```

What is the increased cost of replacing part with name N, with part Y, in part P.

```
extra-cost(p,n,y) =
let yc = totalcost(y) in
    R-REDUCE p, +, 0
        BY { composite(@n,ac,s) => yc - totalcost(composite(n,ac,s));
                    base(@n,c,*) => yc - c;
                               * => 0          }
```

Replace all occurrences of part, old, with part, new, in part P.

```
new-part(p,old,new) =
   R-MAP p
      BY { @old => new;
              x => x    }
```

# 6 Conclusion

Functional programming makes it possible to use functions that apply other functions to complex structures. The classical example of this is the LISP *mapcar*

function. While complex versions of such combinators are expressible in any language which allows recursive types, these functions are difficult to write and to understand. In such languages there are natural combinators that can be defined automatically based on the type structure, and furthermore, these combinators are both useful and reasonably easy to understand.

In this paper we have illustrated the definiton and use of generalized recursive combinators. Functions comprising four different kinds of combinators based on the recursive structures of the defined types are automatically generated as a side effect of the type definitions. This approach could be added naturally to any language with recursive type definition capabilities.

These combinators provide precise specifications for complex computations on recursive types. These specifications are surprisingly easy to understand because they abstract away the details of performing the recursions in the correct manner.

The understanding and use of these functions is greatly facilitated by the use of pattern matching in function definitions, such as is present in languages such as ML. Pattern matching allows the complicated case analysis necessary for robust object manipulation to be specified in a parsimonious fashion.

The ability to traverse complex data in a structured manner has become an issue of great importance in the efforts to design and implement sophisticated systems such as are found in design and knowledge-based applications. Database systems featuring complex objects can make use of generalized recursive structure maps both in the specification of general traversal schemes and the manipulation of complex objects as was shown by our examples.

# References


[1] Hoare, C.A.R. "Recursive Data Structures", International Journal of Computer and Information Science, Vol. 4, No. 2, 1975, pp. 105-132.

[2] Witsrom, Ake. *Functional Programming Using Standard ML*, Prentice Hall International Series in Computer Science, C.A.R. Hoare series editor, Prentice Hall, N.Y. 1987.

[3] Peyton Jones, Simon L. *The Implementation of Functional Programming Languages*, Prentice Hall International Series in Computer Science, C.A.R. Hoare series editor, Prentice Hall, N.Y. 1987.

[4] McCarthy, John. "History of Lisp", Sigplan Notices, vol. 13, no. 8, August 1978.