Uniform Traversal Combinators: Definition, Use and Properties [⋆]

Leonidas Fegaras[1], *Tim Sheard*[2], *and David Stemple*[1]

[1] Department of Computer and Information Science, University of Massachusetts, Amherst, MA 01003. (*email:* {fegaras,stemple}@cs.umass.edu)
[2] Department of Computer Science and Engineering, Oregon Graduate Institute, Beaverton, OR 97006. (*email:* sheard@cse.ogi.edu)

Abstract

In this paper we explore ways of capturing well-formed patterns of recursion in the form of generic reductions. These reductions, called *uniform traversal combinators*, can substantially help the theorem proving process by eliminating the need for induction and can also be an aid in achieving effective program synthesis.

1 Introduction

Recursive structures, such as lists and trees, can be defined inductively in most functional languages [6]. The recursive types of these structures can be formalized using axiom sets generated automatically from their type definition, which are basically equivalent to Hoare's axioms for recursive data structures [5]. Programs that operate on instances of these types can be expressed as recursive functions in a pure applicative language. Theorems about these functions can be proved using induction principles on the structure of the parameter types of these functions. The Boyer-Moore theorem prover [3], for example, proves theorems about recursive functions mechanically by using axioms, definitions, and previously proved theorems, along with powerful induction mechanisms on recursive structures. Program synthesis needs techniques similar to those used in theorem proving. However it is more difficult, partially because induction methods cannot be applied directly for synthesizing the recursive definition of a function.

Proving theorems about computations over recursive structures can be made easier by requiring that functions be expressed in stereotyped ways. One kind of stereotyping is the systematic use of higher order functions that carry out all the traversal of recursive structures. Such traversal functions can capture common patterns of recursion that occur often during programming and, therefore, minimize the explicit use of recursion, which now becomes encapsulated by these functions. The well-known map function that applies a function to each element of a list is an example of a higher order function that encapsulates a traversal. By proving theorems about such traversal functions, some properties of functions using them can be proven by shallower reasoning than would be required

[⋆] This paper is based on work supported by the National Science Foundation under grants IRI-8606424 and IRI 8822121 and by the Office of Naval Research University Research Initiative contract, number N00014-86-K-0764 if the traversals were not "pre-analyzed" in isolation. In particular, the use of induction proofs can be substantially diminished.

Reductions [12, 11] are convenient abstractions for expressing manipulations of bulk data types represented by recursive structures. They accumulate results as they traverse a structure and can be used for more computations than are expressible using a mapping traversal. Reductions tailored to particular recursive types can be generated automatically by a compiler by examining the type details. Reductions over lists and finite sets are expressive enough to directly simulate all primitive recursive functions [7]. The work reported here extends these reductions to cover a larger set of recursion patterns and is motivated by the desire to use them as an aid both in theorem proving and program synthesis.

In this paper we explore a broad class of traversal functions and prove their fundamental properties. We introduce a family of generic functions, called **traversal combinators**, that capture a large family of type-safe primitive recursive functions. Most functions expressed as recursive programs where only one parameter becomes smaller at each recursive call are members of this family. This restriction excludes some valid functions, such as structural equalities and ordering, because they require their two input structures to be traversed simultaneously.

Our generic functions are combinators as each takes functions as inputs and return a new function as output (the one that performs the actual reduction). The most important contribution of this paper is our treatment of the class of traversal combinators resulting from restricting their input functions to be themselves traversal combinators. We call these functions **uniform traversal combinators**. This offers a disciplined and uniform treatment of functions. This uniformity introduces some nice properties, such as these combinators being closed under composition, that aid in theorem proving and program synthesis. In order to prove equality theorems it was necessary to extend our language to include structural equality as a special primitive. Programs expressed in this algebra can be tested for functional equivalence in a systematic and complete way, based on the fact that there is a unique way for expressing a function as a traversal.

Our algebra is at least equivalent to the first order logic. It cannot capture some interesting functions, such as transitive closure and integer exponentiation. Nevertheless, our system can express and prove complex theorems. We envision a system where all theorems expressible in our algebra are proved using the efficient algorithms presented in this paper, while the rest are tested by a theorem prover based on heuristics, such as the Boyer-Moore theorem prover.

2 Related Work

The work reported here is a new method for theorem proving with structural induction. Even though there is some research on analyzing the properties of some highly stereotyped recursions similar to our traversals, there is no work reported on defining a systematic method for applying these properties to theorem proving.

The most influential work on realizing the importance of capturing recursions into a few powerful patterns was by Richard Bird [1]. Even though his work was focused on specific types, such as lists, it suggested ways of extending these methods to other types. The work of Grant Malcolm on homomorphisms [8] generalized these methods for all types. His paper introduced a generic form of homomorphisms, very similar to our traversal combinators, stated without proof the promotion theorem for any homomorphism, and used it for proving some general properties of recursive types. Another work with similar definitions of recursive patterns, called iterative functions, was by Bohm and Berarducci [2], based on the second order lambda calculus. The most complete work on analyzing the properties of stereotyped recursions was by Meijer, Fokkinga, and Paterson [9]. They presented four classes of generic functions: catamorphisms (similar to our traversal combinators), anamorphisms, hylomorphisms, and paramorphisms. They proved a large number of generic theorems for each class, such as the fusion law (similar to our promotion theorem) and the uniqueness property for catamorphisms (for proving equalities of two functions).

3 Definitions

The language used in this and subsequent sections is ADABTPL [4], which is a strongly-typed functional programming language [6]. All types described in this paper are **canonical types**. The set of canonical types is a restricted subset of all the types that can be expressed in ADABTPL. They are constructed exclusively using 1) parameterization, 2) recursion, 3) the singleton type constructor, 4) the tuple type constructor, and 5) the union type constructor. For purposes of explanation, in the definition of a type T we restrict recursion to be a direct reference to type T, not to any type expression $t(T)$ that depends on T. For example, we do not permit the definition of the part-subpart tree structure where a subpart consists of a list of parts. In Section 8 we will remove this restriction to allow any recursively defined type.

An example of a tuple type definition is:

```
person = struct make_person ( name: string, address: string );
```

The tuple value constructor here is `make_person` of type `[string,string]->person`, that is, it takes two strings as input and returns a new object of type `person` as output. Examples of polymorphic union types are lists and trees (they have one type parameter `alpha` that can be instantiated to any canonical type):

```
list(alpha) = union
    ( null: singleton nil,
      consp: struct cons ( head: alpha, tail: list(alpha) ) );
tree(alpha) = union
    ( emptytree: singleton empty,
      fulltree: struct node
                 ( info: alpha, left: tree(alpha), right: tree(alpha) ) );
```

The singleton type constructor creates a type with only one value. The value is constructed by the nullary constructor function named following `singleton`, such as `nil` and `empty`. Lists and trees have the following constructors:

```
nil: []->list(alpha)
```

```
cons: [alpha,list(alpha)]->list(alpha)
empty: []->tree(alpha)
node: [alpha,tree(alpha),tree(alpha)]->tree(alpha)
```

In general, any canonical type T has a number of constructors $C_1, \ldots, C_n$. More specifically, a tuple type has only one constructor whose input types are not recursive. Each alternative of a union type has one constructor and therefore this union type has all these constructors. We assume that any union type has at least one constructor with no recursive input types. Nullary constructors of type $C_i : () \rightarrow T$ are considered to be constant values of type T. For example, `nil` is a constructor of type `[]->list(alpha)` but it is used as a constant of type `list(alpha)`. To make our notation simpler, we assume that each constructor C_i has the variables of type T separated from the other variables: we write $C_i(\overline{x_i}, \overline{y_i})$ to indicate that the variables $\overline{x_i} = x^i_1, \ldots, x^i_{i_r}$ are of any type other than T and the variables $\overline{y_i} = y^i_1, \ldots, y^i_{i_s}$ are of type T.

One generic function over lists is the list reduction function `tc_list(fnil,fcons)`:

```
function(alpha,beta)
   tc_list ( fnil:  []->beta,
             fcons: [alpha,list(alpha),beta]->beta ): [list(alpha)]->beta;
[x]->case x
     { nil -> fnil();
       cons(a,r) -> fcons(a,r,tc_list(fnil,fcons)(r)) };
```

where $[x_1, \ldots, x_n] \rightarrow \text{exp}$ is a lambda abstraction with variables $x_1, \ldots, x_n$ and body exp (expressed as $\lambda x_1 \ldots \lambda x_n.\text{exp}$ in lambda calculus) and `case` matches `x` with one of the list constructor patterns. The list of variables following the keyword `function`, that is, `(alpha,beta)` in our example, denotes free type variables of a polymorphic function definition. Note that the body of `tc_list` is a lambda abstraction, that is, this function returns a closure. In other words, since the inputs are functions this is a combinator. For this reason, applying `tc_list` of the functions `f1` and `f2` to a list `l` is written as `tc_list(f1,f2)(l)`. For example, the list length function is computed by `tc_list([]->0,[?,?,i]->i+1)`, where `?` is a don't-care parameter. The list append function `append(x,y)` is computed by `tc_list([]->y,[a,?,l]->cons(a,l))(x)`, while the list reverse is computed by `tc_list([]->nil,[a,?,l]->append(l,cons(a,nil)))`.

Reductions can be generalized to cover all canonical types. We call these generalized reductions *traversal combinators*. They are combinators because they accept functions as inputs, such as `fnil` and `fcons` in `tc_list`, and return a new function as output. The 'traversal' part of the name is justified because the output function of the combinator traverses the hierarchical structure of its input object.

Definition 1 (Traversal combinator) *Let T be a canonical type with constructors $C_i(\overline{x_i}, \overline{y_i})$. A traversal combinator $\mathcal{H}_T(f_1, \ldots, f_n) : T \rightarrow b$, where b is any type, is defined as follows:*

$$
\begin{array}{l}
[x] \rightarrow \textit{case } x \\
\quad \{ \quad \ldots \\
\quad\quad C_i(\overline{x_i}, \overline{y_i}) \ \rightarrow\ f_i(\overline{x_i}, \overline{y_i}, \mathcal{H}_T(f_1, \ldots, f_n)(y^i_1), \ldots, \mathcal{H}_T(f_1, \ldots, f_n)(y^i_{i_r})); \\
\quad\quad \ldots \\
\quad \}
\end{array}
$$

Variables z^i_k in $f_i(\overline{x_i}, \overline{y_i}, \overline{z_i})$, where $z^i_k = \mathcal{H}_T(f_1, \ldots, f_n)(y^i_k)$, are called **accumulative result variables** because they accumulate the results of the recursive calls to $\mathcal{H}_T(f_1, \ldots, f_n)$, while variables from $\overline{x_i}$ and $\overline{y_i}$ are called non-accumulative result variables. Note that if $C_i(\overline{x_i}, \overline{y_i})$ has k variables of a type other than T (these are the $\overline{x_i}$ variables) and m variables of type T (these are the $\overline{y_i}$ variables), then f_i has $k + 2m$ variables: $k + m$ non-accumulative and m accumulative result variables. For example, in the expression `tc_list([]->y,[a,l,r]->cons(a,r))(x)` variable `r` is accumulative while `a`, `l` and `x` are not.

From Definition 1 we can see that if f is the traversal combinator $\mathcal{H}_T(f_1, \ldots, f_n)$ then:

$$f(C_i(\overline{x_i}, \overline{y_i})) = f_i(\overline{x_i}, \overline{y_i}, f(y^i_1), \ldots, f(y^i_{i_r}))$$

In addition, if $\forall i: \; f_i(\overline{x_i}, \overline{y_i}, \overline{z_i}) = C_i(\overline{x_i}, \overline{z_i})$ then $f = \lambda x.x$; that is, f is the identity function for T.

In ADABTPL, a traversal combinator $\mathcal{H}_T$ is written as `tc_T`. For example, the integer type `int` has two constructors: `zero: int` and `succ: [int]->int`. The traversal combinator over integers is `tc_int(fz,fs)`:

```
function(beta) tc_int ( fz: []->beta, fs: [int,beta]->beta ): [int]->beta;
[x]->case x
     { zero -> fz();
       succ(i) -> fs(i,tc_int(fz,fs)(i)) };
```

If `beta=int` then this combinator can simulate all primitive recursive functions for integers [10]. For example, `tc_int([]->y,[?,i]->succ(i))(x)` computes $x + y$; `tc_int([]->zero,[?,i]->i+y)(x)` computes $x * y$; `tc_int([]->true,[?,i]->(i=false))(x)` computes the predicate $even(x)$; `tc_int([]->true,[i,r]->(i=y) or r)(x)` computes $x \leq y$; `tc_int([]->zero,[i,r]->succ(if (i=y) then zero else r))(x)` computes $x - y$; and `tc_int([]->succ,[?,f]->tc_int([]->f(succ(zero)),[?,i]->f(i)))(m)(n)` computes the Ackermann function $\mathrm{Ack}(m, n)$ [2].

The boolean type is just the union of the singletons `true` and `false`. The boolean traversal combinator `tc_boolean` is the thinly disguised if-then-else function:

```
function(beta) tc_boolean
   ( ftrue: []->beta, ffalse: []->beta ) : [boolean]->beta;
[x]->case x { true -> ftrue(); false -> ffalse() };
```

A theorem, very useful for proving properties about traversal combinators is the promotion theorem [8] (or fusion law [9]). It states the condition for the composition of a function with a traversal combinator to be a traversal combinator too. It says that the composition of a function g with a traversal combinator $\mathcal{H}_T(f_1, \ldots, f_n)$ is also a traversal combinator if the composition of g with each f_i promotes the g call only to the accumulative result variables of f_i. This theorem is used in Section 4 as a reduction rule for composing traversal combinators.

Theorem 1 (Promotion theorem)
if $\forall i \forall \overline{x_i} \forall \overline{y_i} \forall \overline{z_i}: \; g(f_i(\overline{x_i}, \overline{y_i}, \overline{z_i})) = \phi_i(\overline{x_i}, \overline{y_i}, g(z^i_1), \ldots, g(z^i_{i_r}))$ *then*

$$g \circ \mathcal{H}_T(f_1, \ldots, f_n) = \mathcal{H}_T(\phi_1, \ldots, \phi_n)$$

Proof by structural induction: We will prove that $\forall x : g(f(x)) = \phi(x)$, where $f = \mathcal{H}_T(f_1, \ldots, f_n)$ and $\phi = \mathcal{H}_T(\phi_1, \ldots, \phi_n)$. If x is a construction $C_i(\overline{x_i})$ (that is, C_i has no arguments of type T), then $g(f(C_i(\overline{x_i}))) = g(f_i(\overline{x_i})) = \phi_i(\overline{x_i}) = \phi(C_i(\overline{x_i}))$. Let $x = C_i(\overline{x_i}, \overline{y_i})$. We assume the theorem is true for all $y : T$ that are subtrees of the tree x. Then $g(f(x)) = g(f_i(\overline{x_i}, \overline{y_i}, f(y^i_1), \ldots, f(y^i_{i_r}))) = \phi_i(\overline{x_i}, \overline{y_i}, g(f(y^i_1)), \ldots, g(f(y^i_{i_r})))$. But each y^i_k is a subtree of x and thus $g(f(y^i_k)) = \phi(y^i_k)$. Therefore, $\phi_i(\overline{x_i}, \overline{y_i}, g(f(y^i_1)), \ldots, g(f(y^i_{i_r}))) = \phi_i(\overline{x_i}, \overline{y_i}, \phi(y^i_1), \ldots, \phi(y^i_{i_r})) = \phi(C_i(\overline{x_i}, \overline{y_i})) = \phi(x)$.□

For example, for lists we have:

$$\left.\begin{array}{rl} g(f_1()) & = \phi_1() \\ \forall a \forall l \forall s : g(f_2(a, l, s)) & = \phi_2(a, l, g(s)) \end{array}\right\} \Rightarrow g \circ \text{tc\_list}(f_1, f_2) = \text{tc\_list}(\phi_1, \phi_2)$$

The promotion theorem for integers is:

$$\left.\begin{array}{rl} g(f_1()) & = \phi_1() \\ \forall i \forall r : g(f_2(i, r)) & = \phi_2(i, g(r)) \end{array}\right\} \Rightarrow g \circ \text{tc\_int}(f_1, f_2) = \text{tc\_int}(\phi_1, \phi_2)$$

The following corollary says that there is a unique way for expressing a function as a traversal combinator [9]. It is used in Section 5 for testing the functional equality of two traversal combinators:

Corollary 1 (Uniqueness property)

$$\forall i \forall \overline{x_i} \forall \overline{y_i} : \; g(C_i(\overline{x_i}, \overline{y_i})) = \phi_i(\overline{x_i}, \overline{y_i}, g(y^i_1), \ldots, g(y^i_{i_r})) \Longleftrightarrow g = \mathcal{H}_T(\phi_1, \ldots, \phi_n)$$

Proof: $\Rightarrow$: From the promotion theorem with $f_i(\overline{x_i}, \overline{y_i}, \overline{z_i}) = C_i(\overline{x_i}, \overline{z_i})$.
$\Leftarrow$: From Definition 1. □

Structural equalities for all but the trivial canonical types cannot be captured as traversal combinators. We need to define a special combinator instead. In the following definition of structural equality we do not separate the arguments of C_i of type T from the others in order to make the notation more readable:

Definition 2 (Structural Equality) *Let $T(\overline{\alpha})$ be a canonical type, where $\overline{\alpha}$ is a sequence of type parameters $\alpha_1, \ldots, \alpha_r$, $r \geq 0$. A structural equality $\mathcal{EQ}_T(\varepsilon_1, \ldots, \varepsilon_r) : T(\overline{\alpha}) \times T(\overline{\alpha}) \rightarrow$ boolean, where $\varepsilon_i : \alpha_i \times \alpha_i \rightarrow$ boolean, over the canonical type $T(\overline{\alpha})$ is defined as:*

```
$[x, y] \rightarrow$ case $x$, $y$
        {   ...
            $C_i(x_1, \ldots, x_{k_i})$, $C_i(y_1, \ldots, y_{k_i})$ $\rightarrow$ $\bigwedge_{j=1}^{k_i} \mathcal{R}_{i,j}(x_j, y_j)$;
            ...
            other $\rightarrow$ false
        }
```

where each x_j and y_j are of type $T_{i,j}(\overline{\alpha})$ and

$$\mathcal{R}_{i,j} = \begin{cases} \varepsilon_s & \text{if } \exists s : T_{i,j}(\overline{\alpha}) = \alpha_s \\ \mathcal{EQ}_{T_{i,j}}(\varepsilon_1, \ldots, \varepsilon_r) & \text{otherwise} \end{cases}$$

In ADABTPL, a structural equality $\mathcal{EQ}_T$ is written as equal_T. For example, the list equality has only one parameter ea that corresponds to the type parameter alpha:

```
function(alpha) equal_list
   ( ea: [alpha,alpha]->boolean ) : [list(alpha),list(alpha)]->boolean;
[x,y]->case x, y
       { nil, nil -> true;
         cons(a,l), cons(b,r) -> ea(a,b) and equal_list(ea)(l,r);
         other -> false };
```

We will now enhance the definition of structural equality $\mathcal{EQ}_T$ in such a way that the composition algorithm in the next section is true. This is done by taking the two inputs of type T of the output function of $\mathcal{EQ}_T$, such as the two lists of equal_list, as nullary functions and by adding a continuation that maps the result of the equality to a type β (typically this continuation is expressed as a tc_boolean combinator). The reason behind this enhancement is that when we compose a traversal combinator with an equality combinator we want to yield another equality combinator so that our language is closed under composition. This is achieved with the extra continuation ϕ.

Definition 3 (Equality Combinator) *Let T be a canonical type with structural equality $\mathcal{EQ}_T$, f and g functions of type $()\rightarrow T$, and ϕ a function of type boolean $\rightarrow\beta$. The equality combinator for T is $\mathcal{E}_T : ()\rightarrow\beta$, defined as:*

$$\mathcal{E}_T(f,h,\phi) = \phi(\mathcal{EQ}_T(\varepsilon_1,\ldots,\varepsilon_r)(f,h))$$

Parameters ε_k are instantiated to equalities whenever the type parameters of T are instantiated to types. From now on we will ignore them because they do not affect our analysis. We will assume that they are hidden in $\mathcal{EQ}_T$. The promotion theorem for equality combinators is simply:

$$g \circ \mathcal{E}_T(f,h,\phi) = \mathcal{E}_T(f,h,g\circ\phi)$$

In ADABTPL, an equality combinator $\mathcal{E}_T$ is written as eq_T. For example:

```
function(beta) eq_list
  ( f: []->list(int), h: []->list(int), c: [boolean]->beta ) : beta;
c(equal_list(equal_int)(f(),h()));
```

4 Uniform Traversal Combinators

A very interesting class of traversal combinators results from restricting all functions f_i in $\mathcal{H}_T(f_1,\ldots,f_n)$ to be traversal combinators themselves. This restriction is very important because any theorem or algorithm that refers to such combinators can also work on each f_i recursively. This strict discipline of the function form offers us a more uniform treatment of functions. Functions have now a tree-like form, where each traversal combinator $\mathcal{H}_T(f_1,\ldots,f_n)$ is a node and each f_i is a child. This structured view of functions is aimed at simplifying theorem proving and facilitating program synthesis.

Definition 4 (Uniform Traversal combinator) *A uniform traversal combinator from T to b, where T and b are canonical types and all variables z and $\overline{z}$ are bounded variables, is one of the following:*

- **projection***: a lambda abstraction $\lambda\overline{x}.\, z$, where z is a variable of type b;*
- **construction***: an expression $\lambda\overline{x}.\, C(h_1(\overline{z}), \ldots, h_s(\overline{z}))$ where C is a constructor of b and each h_i is a uniform traversal combinator;*
- **traversal***: a traversal combinator $\lambda\overline{x}.\, \mathcal{H}_T(f_1, \ldots, f_n)(z)$ from T to b where each f_i is a uniform traversal combinator and variable z is a non-accumulative result variable;*
- **equality***: an equality combinator $\lambda\overline{x}.\, \mathcal{E}_T(f, h, \phi)$, where f, h, and ϕ are uniform traversal combinators.*

For example, integer multiplication is computed by the uniform traversal combinator:

```
[x,y]->tc_int([]->zero,[?,i]->tc_int([]->i,[?,j]->succ(j))(y))(x)
```

The function `reverse(x)` computed by:

```
[x]->tc_list([]->nil,[a,?,l]->tc_list([]->cons(a,nil),
                                      [c,?,s]->cons(c,s))(l))(x)
```

is not a uniform traversal combinator, because the inner `tc_list` is on `l` which is an accumulative result variable. The following is a uniform traversal combinator that when applied over a list of lists `x` it returns the list of lengths of `x`:

```
[x]->tc_list([]->nil,[a,?,r]->cons(tc_list([]->zero,
                                           [?,?,i]->succ(i))(a),r))(x)
```

Here the inner `tc_list` is over `a`, which is not an accumulative result variable. Integer subtraction `x-y` can be computed by the following uniform traversal combinator:

```
[x,y]->tc_int([]->zero,
              [i,r]->succ(eq_int([]->i,[]->y,
                                 [z]->tc_boolean([]->zero,[]->r)(z))))(x)
```

Two points need to be clear in the definition of uniform traversal combinators. First, traversals are over variables, not over expressions. In addition, this definition does not say that the composition of uniform traversal combinators is also a uniform traversal combinator. But we will prove next that this is true for any such composition. Second, the variable z of a traversal must not be an accumulative result variable. This is a necessary condition for having these combinators closed under composition. The intuition behind this is that we cannot traverse the values that are accumulated during the traversal of a structure (but we can pass them as whole values to constructions or to equalities). This restriction is very important because it substantially limits the expressiveness of our algebra (it makes our language deterministic logspace instead of polynomial time). Even though there is an alternative way of expressing the reverse function, there are some interesting functions, such as the transitive closure and the integer exponentiation, that cannot be captured in our algebra.

4.1 Composition of Uniform Traversal Combinators

Suppose that we have the composition $g(h(\overline{x}))$, where g and h are uniform traversal combinators. We can synthesize the traversal ϕ equal to $g \circ h$ by applying the promotion theorem to break it down into simpler cases. These cases are also made simpler by applying the promotion theorem again. The following constructive proof composes any uniform traversal combinators.

Theorem 2 (Composition of Uniform Traversal Combinators)
The composition of uniform traversal combinators is a uniform traversal combinator.

Constructive proof: First we will present the algorithm for composing combinators and then we will prove that the algorithm is correct. We denote $\Phi(g, [h_1, \ldots, h_r], \rho)$ the application of the function g over $h_1 \ldots h_r$, where all g, $h_1, \ldots, h_r$ are uniform traversal combinators. Variable ρ contains bindings from combinators to combinators. To find this composition, the g call is pushed inside the expressions h_i by applying the promotion theorem. The promotion theorem says that for $f = \mathcal{H}_T(f_1, \ldots, f_n)$: $g \circ f = \mathcal{H}_T(\phi_1, \ldots, \phi_n)$ if $g(f_i(\overline{x_i}, \overline{y_i}, \overline{z_i})) = \phi_i(\overline{x_i}, \overline{y_i}, g(z_1^i), \ldots, g(z_{i_r}^i))$. If $w_k^i = g(z_k^i)$ then $\phi_i(\overline{x_i}, \overline{y_i}, w_1^i, \ldots, w_{i_r}^i)$ is the composition $g \circ f_i$, provided that all references to z_k^i are eliminated (as it will be proved below). For that reason we pass a binding list ρ to Φ that contains the bindings $g(z_k^i) = w_k^i$ and $z_k^i = f(y_k^i)$ (from Definition 1). The algorithm consists of the following rules:

Algorithm 1 (Composition Algorithm)

$\Phi(\lambda\overline{x}.x_k, [\ldots, h_i, \ldots], \rho)$	$\rightarrow h_k$
$\Phi(g, [\lambda\overline{x}.x_i], \rho)$	$\rightarrow$ if $\rho \vdash g(x_i)/e$ then $\lambda\overline{x}.e$ else $\lambda\overline{x}.g(x_i)$
$\Phi(\lambda\overline{x}.C(\ldots, e_i(\overline{x}), \ldots), [\ldots, h_i, \ldots], \rho)$	$\rightarrow \lambda\overline{x}.C(\ldots, \Phi(e_i, [\ldots, h_i, \ldots], \rho), \ldots)$
$\Phi(\lambda\overline{x}.\mathcal{H}_T(\ldots, f_i, \ldots)(z), [C_i(\overline{u}, \overline{w})], \rho)$	$\rightarrow \Phi(f_i, [\overline{u}, \overline{w}, \ldots, \Phi(\lambda\overline{x}.\mathcal{H}_T(\ldots, f_i, \ldots)(z), [w_k], \rho), \ldots], \rho)$
$\Phi(\lambda\overline{x}.\mathcal{E}_T(f, h, \phi), [\ldots, h_i, \ldots], \rho)$	$\rightarrow \lambda\overline{x}.\mathcal{E}_T(\Phi(\lambda\overline{x}.f, [\ldots, h_i, \ldots], \rho), \Phi(\lambda\overline{x}.h, [\ldots, h_i, \ldots], \rho), \phi)$
$\Phi(g, [\lambda\overline{x}.\mathcal{E}_T(f, h, \phi)], \rho)$	$\rightarrow \lambda\overline{x}.\mathcal{E}_T(f, h, \Phi(g, [\phi], \rho))$
$\Phi(g, [\lambda\overline{x}.\mathcal{H}_T(\ldots, f_i, \ldots)(z)], \rho)$	$\rightarrow \lambda\overline{x}.\mathcal{H}_T(\ldots, \Phi(g, [\lambda\overline{x_i}\lambda\overline{y_i}\lambda\overline{w_i}.f_i(\overline{x_i}, \overline{y_i}, \overline{z_i})], \rho'), \ldots)(z)$
	where $\rho' = \rho[\ldots, g(z_k^i)/w_k^i, z_k^i/\mathcal{H}_T(\ldots, f_i, \ldots)(y_k^i), \ldots]$

Expression $\rho[u/w]$ extends ρ with the binding from the combinator u to the combinator w, while expression $\rho \vdash u/w$ returns true if there is a binding in ρ from u to w. The last rule comes from the promotion theorem. It renames the variables of f_i from $\overline{w_i}$ to $\overline{z_i}$ but it binds each w_k^i in $\overline{w_i}$ to $g(z_k^i)$. So if later a call $g(y_k^i)$ is found, it is replaced with w_k^i.

It is obvious that the above reductions are correct and they always terminate, because one of the two parameters of Φ becomes smaller in each recursive call to Φ. The resulting expression has the form of a uniform traversal combinator but possibly with some variables unbound. These are the z_k^i variables from the last rule. The only thing that remains to prove is that there are no such unbound variables z_k^i at the end. But the last rule applies whenever g is a traversal. In that case Φ is called recursively with g as the first argument. This is true for the fourth and sixth rule too. This recursion terminates whenever we find a call $\Phi(g, [h], \rho)$ that does not match these recursive rules. Therefore, this call must match the second rule. Then, if x_i in $\lambda\overline{x}.x_i$ is one of the z_k^i in ρ then $g(x_i)$ is replaced with the variable w_k^i. If z_k^i does not appear in a call to g, then z_k^i is replaced with $f(y_k^i)$, where $f = \mathcal{H}_T(\ldots, f_i, \ldots)$. The only other place that z_k^i could appear is in z in the last rule. But this is not permitted because traversals cannot be done over accumulative result variables. □

For example, suppose that we want to find a traversal combinator `tc_list(h1,h2)(x)` equivalent to the composition `length(append(x,y))`, where `append` and `length` are:

```
append(x,y) = tc_list([]->y,[a,?,s]->cons(a,s))(x)
length(x) = tc_list([]->zero,[?,?,i]->succ(i))(x)
```

We apply the promotion theorem with g =`length`:

```
1) h1() = length(y) = tc_list([]->zero,[?,?,i]->succ(i))(y)
2) h2(a,?,length(s)) = length(cons(a,s)) = succ(length(s))
     => h2(?,?,u)=succ(u)   where u=length(s)
```

Therefore, the composition `length(append(x,y))` is:

```
tc_list([]->tc_list([]->zero,[?,?,i]->succ(i))(y),[?,?,u]->succ(u))(x)
```

Let `length(x)+length(y)` be equal to `tc_list(h1,h2)(x)`, where

```
x+y = tc_int([]->y,[?,j]->succ(j))(x)
```

We apply the promotion theorem for lists with

```
g(x) = tc_int([]->tc_list([]->zero,[?,?,i]->succ(i))(y),[?,j]->succ(j))(x)
```

to compose `g(length(x)) = tc_list(h1,h2)(x)`:

```
1) h1() = g(zero) = tc_list([]->zero,[?,?,i]->succ(i))(y)
2) h2(?,?,g(i)) = g(succ(i)) = succ(g(i))
     => h2(?,?,w)=succ(w)    where w=g(i)
```

Therefore, `length(x)+length(y)` is:

```
tc_list([]->tc_list([]->zero,[?,?,i]->succ(i))(y),[?,?,w]->succ(w))(x)
```

From these two examples we can see that

```
length(append(x,y)) = length(x)+length(y)
```

This is an example of a theorem proved without using induction explicitly. Here testing the equality of `length(append(x,y))` and `length(x)+length(y)` was trivial. In Section 5 we will present a complete method for testing functional equalities.

We will compose now `g(f(x))`, where:

```
f(x) = tc_list([]->nil,[a,?,r]->cons(tc_list([]->zero,
                                              [?,?,i]->succ(i))(a),r))(x)
g(y) = tc_list([]->zero,[b,?,j]->tc_int([]->j,[?,k]->succ(k))(b))(y)
```

(if `x` is a list of lists then `f(x)` returns the list of lengths of `x` and if `y` is a list of integers then `g(y)` returns the sum of all these integers). The promotion theorem for `g(f(x))` equal to `tc_list(f1,f2)(x)` gives:

```
1) f1() = g(nil) = zero
2) f2(a,?,g(r)) = g(cons(tc_list([]->zero,[?,?,i]->succ(i))(a),r))
      = tc_int([]->g(r),[?,k]->succ(k))
        (tc_list([]->zero,[?,?,i]->succ(i))(a))
```

Let `h(x) = tc_int([]->u,[?,k]->succ(k))(x)`, where `u=g(r)`, and `h(tc_list([]->zero,[?,?,i]->succ(i))(a)) = tc_list(h1,h2)(a)`, then:

```
1) h1() = u
2) h2(?,?,h(i)) = succ(h(i)) => h2(?,?,w)=succ(w)
```

Therefore, `g(f(x))` is

```
tc_list([]->zero,[a,?,u]->tc_list([]->u,[?,?,w]->succ(w))(a))(x)
```

5 Equality of Uniform Traversal Combinators

The following algorithm tests whether any two uniform traversal combinators compute the same function. It is based on the uniqueness property that says that there is a unique way for expressing a function as a traversal combinator. We will make use of this algorithm in Section 6 for proving equality theorems.

Theorem 3 (Equality theorem) *Let $\lambda\overline{x}.f(\overline{x})$ and $\lambda\overline{x}.g(\overline{x})$ be two uniform traversal combinators. Then $\forall\overline{x}: \ f(\overline{x}) = g(\overline{x})$ iff $\mathcal{E}(\lambda\overline{x}.f(\overline{x}), \lambda\overline{x}.g(\overline{x}), []) =$ true, where $\mathcal{E}$ is defined as: (for purposes of explanation we ignore variable renaming)*

Algorithm 2 (Equality of Combinators)

```
$\mathcal{E}(\lambda\overline{x}.z, \lambda\overline{x}.z, \rho)$    $\rightarrow$ true
$\mathcal{E}(g, \lambda\overline{x}.z, \rho)$    $\rightarrow \rho \vdash z/g$
$\mathcal{E}(\lambda\overline{x}.C(\ldots, e_i(\overline{z}), \ldots), \lambda\overline{x}.C(\ldots, h_i(\overline{z}), \ldots), \rho)$ $\rightarrow \forall i: \ \mathcal{E}(e_i, h_i, \rho)$
$\mathcal{E}(\lambda\overline{x}.C_1(\ldots), \lambda\overline{x}.C_2(\ldots), \rho)$    $\rightarrow$ false
$\mathcal{E}(g, \mathcal{E}_T(f, h, \phi), \rho)$    $\rightarrow$ if $\mathcal{E}(f, h, \rho)$ then $\mathcal{E}(g, \phi(\text{true}), \rho)$
                                                  else $\mathcal{E}(g, \phi(\text{false}), \rho)$
$\mathcal{E}(\lambda\overline{x}.g(z), \lambda\overline{x}.\mathcal{H}_T(\ldots, \phi_i, \ldots)(z), \rho)$    $\rightarrow \forall i: \ \mathcal{E}(g(C_i(\overline{x_i}, \overline{y_i})), \lambda\overline{x_i}\lambda\overline{y_i}.\phi_i(\overline{x_i}, \overline{y_i}, \overline{z_i}), \rho')$
                                              where $\rho' = \rho[\ldots, z^i_k/g(y^i_k), \ldots]$
```

Proof: From the uniqueness property we have:

$$\forall i \forall\overline{x_i}\forall\overline{y_i}: \ g(C_i(\overline{x_i}, \overline{y_i})) = \phi_i(\overline{x_i}, \overline{y_i}, g(y^i_1), \ldots, g(y^i_{i_r})) \ \Leftrightarrow \ g = \mathcal{H}_T(\phi_1, \ldots, \phi_n)$$

That is, g is equal to $\mathcal{H}_T(\ldots, \phi_i, \ldots)$ if and only if for all i: $g(C_i(\overline{x_i}, \overline{y_i}))$ is equal to $\phi_i(\overline{x_i}, \overline{y_i}, \overline{z_i})$, where $z^i_k = g(y^i_k)$ (this is the last rule). In that case, ρ is extended to include all bindings $z^i_k = g(y^i_k)$ so that if later we need to test whether $z^i_k = g(y^i_k)$ then this is true (this is the second rule). □

For example, suppose that we want to prove the commutativity law for integer addition `x+y==y+x`, where `==` is the structural equality `equal_int` for integers. This is expressed as:

```
tc_int([]->y,[?,i]->succ(i))(x) == tc_int([]->x,[?,j]->succ(j))(y)
```

Let `g(y)=tc_int([]->y,[?,i]->succ(i))(x)` then we apply the uniqueness property for the equality `g(y)==tc_int([]->x,[?,j]->succ(j))(y)`:

```
1) y=zero:    (g(zero)==x) = (tc_int([]->zero,[?,i]->succ(i))(x)==x) = true
2) y=succ(j): (g(succ(j))==succ(g(j))) =
              (tc_int([]->succ(j),[?,i]->succ(i))(x)==succ(g(j)))
```

Let `f(x)=succ(g(j)))`. We apply the uniqueness property again:

```
1) x=zero:    (f(zero)==succ(j)) = (succ(j)==succ(j)) = true
2) x=succ(i): (f(succ(i))==succ(f(i))) = (succ(f(i))==succ(f(i))) = true
```

6 Theorem Proving

The algorithms for composing uniform traversal combinators and for testing their functional equalities can be applied for proving theorems about uniform traversal combinators. Suppose that we want to prove that an expression $e(\overline{x})$ is always true. First we need to find all uniform traversal combinators associated with each function call in e. Then we use the composition algorithm to synthesize the uniform traversal combinator $f(\overline{x})$ equivalent to $e(\overline{x})$. Then we are left with the simpler task of proving whether $f(\overline{x})$ is always true (a tautology).

Algorithm 3 (Tautology) *Let $f(\overline{x})$ be a uniform traversal combinator of type boolean. Then $\forall \overline{x} : f(\overline{x}) = c$, where c is either true or false, iff $\mathcal{T}(f(\overline{x}), c)$, where $\mathcal{T}$ is defined as:*

$\mathcal{T}(c, c)$	$\rightarrow$ *true*	
$\mathcal{T}(\lambda\overline{x}\lambda\overline{y}.y_i, c)$	$\rightarrow$ *true*	(y_i *is an accumulative result variable*)
$\mathcal{T}(\lambda\overline{x}.\mathcal{H}_T(f_1, \ldots, f_n)(z), c)$	$\rightarrow \forall i : \mathcal{T}(f_i, c)$	
$\mathcal{T}(\lambda\overline{x}.\mathcal{E}_T(f, h, \phi), c)$	$\rightarrow$ *if* $\mathcal{E}(f, h, [])$ *then* $\mathcal{T}(\phi(\mathit{true}), c)$ *else* $\mathcal{T}(\phi(\mathit{false}), \neg c)$	
otherwise	$\rightarrow$ *false*	

Suppose that we want to prove the associativity law for integer multiplication, that is: `(x*y)*z==x*(y*z)`, where multiplication is computed by:

```
x*y = tc_int([]->zero,[?,i]->tc_int([]->i,[?,j]->succ(j))(y))(x)
```

We start by composing `(x*y)*z`. We apply the promotion theorem with

```
g(u) = u*z = tc_int([]->zero,[?,i]->tc_int([]->i,[?,j]->succ(j))(z))(u)
```

Let `g(tc_int([]->zero,[?,i]->tc_int([]->i,[?,j]->succ(j))(y))(x))` be equal to `tc_int(f1,f2)(x)`. Then from the promotion theorem we have:

```
1) f1() = g(zero) = zero
2) f2(?,g(i)) = g(tc_int([]->i,[?,j]->succ(j))(y)) = tc_int(h1,h2)(y)
```

Let `m=g(i)`. We apply the promotion theorem again:

```
1) h1() = g(i) = m
2) h2(?,g(j)) = g(succ(j)) = tc_int([]->g(j),[?,k]->succ(k))(z)
    => h2(?,w) = tc_int([]->w,[?,k]->succ(k))(z)
```

Therefore, `(x*y)*z` is

```
tc_int([]->zero,
       [?,m]->tc_int([]->m,[?,w]->tc_int([]->w,[?,k]->succ(k))(z))(y))(x)
```

We will compose now `x*(y*z)`:

```
x*(y*z) = tc_int([]->zero,[?,i]->tc_int([]->i,[?,j]->succ(j))(y*z))(x)
```

Let `g(u) = u+i = tc_int([]->i,[?,j]->succ(j))(u)`. Expression `x*(y*z)` becomes:

```
g(y*z) = tc_int(f1,f2)(y)
       = tc_int([]->zero,[?,u]->tc_int([]->u,[?,w]->succ(w))(z))(y)
```

and the promotion theorem gives:

```
1) f1() = g(zero) = i
2) f2(?,g(u)) = g(tc_int([]->u,[?,w]->succ(w))(z)) = tc_int(h1,h2)(z)
```

Let `k=g(u)`. We apply the promotion theorem again:

```
1) h1() = g(u) = k
2) h2(?,g(w)) = g(succ(w)) = succ(g(w))  => h2(?,m) = succ(m)
```

Therefore, `x*(y*z)` is

```
tc_int([]->zero,
       [?,i]->tc_int([]->i,[?,k]->tc_int([]->k,[?,m]->succ(m))(z))(y))(x)
```

Finally, we can see that `(x*y)*z` is equal to `x*(y*z)`.

7 Program Synthesis

Let g and ϕ be uniform traversal combinators. The following algorithm synthesizes all uniform traversal combinators f such that $g(f(\overline{x})) = \phi(\overline{x})$ is true. We express that as: $f \in \mathcal{S}(g, \phi)$, that is, $\mathcal{S}(g, \phi)$ returns the set of uniform traversal combinators f that satisfy $g(f(\overline{x})) = \phi(\overline{x})$. If ϕ is a traversal, then from the promotion theorem we have $\forall i : \; g(f_i(\overline{x_i}, \overline{y_i}, \overline{z_i})) = \phi_i(\overline{x_i}, \overline{y_i}, g(z_1^i), \ldots, g(z_r^i))$. If all such f_i are found, then f is $\mathcal{H}_T(f_1, \ldots, f_s)$. But $\phi_i(\overline{x_i}, \overline{y_i}, g(z_1^i), \ldots, g(z_r^i))$ is the composition of the uniform traversal combinators ϕ_i and g and thus it can be derived using the composition algorithm. Therefore, we need to synthesize all f_i that satisfy the equation $g(f_i(\overline{x_i}, \overline{y_i}, \overline{z_i}))$ = a known traversal. This is achieved by calling the same algorithm for synthesizing combinators recursively. The detailed algorithm is the following:

Algorithm 4 (Synthesis algorithm)

$$\begin{array}{ll}
\mathcal{S}(\lambda y.y, \phi) & \rightarrow \{\phi\} \\
\mathcal{S}(\lambda y.C(\ldots), \lambda\overline{z}.z_i) & \rightarrow \mathcal{S}(\lambda y.C(\ldots), \lambda\overline{x_i}\lambda\overline{y_i}.C(\overline{x_i}, \overline{y_i})) \\
\mathcal{S}(\lambda y.C_1(\ldots), C_2(\ldots)) & \rightarrow \emptyset \\
\mathcal{S}(\lambda y.C(\ldots, g_i, \ldots), C(\ldots, h_i, \ldots)) & \rightarrow mgu(\ldots, \mathcal{S}(g_i, h_i), \ldots) \\
\mathcal{S}(\mathcal{H}_T(\ldots), \lambda\overline{z}.z_i) & \rightarrow \mathcal{S}(\mathcal{H}_T(\ldots), \lambda\overline{z}.\mathcal{H}_T(\ldots, C_i, \ldots)) \\
\mathcal{S}(g, \mathcal{H}_T(\ldots, \lambda\overline{x_i}\lambda\overline{y_i}\lambda\overline{z_i}.\phi_i(\overline{x_i}, \overline{y_i}, \overline{z_i}), \ldots)) & \rightarrow \{\mathcal{H}_T(\ldots, f_i, \ldots) / \\
 & \quad f_i \in \mathcal{S}(g, \Phi(\phi_i, [\overline{x_i}, \overline{y_i}, \ldots, g(z_k^i), \ldots], []))\} \\
\mathcal{S}(\mathcal{H}_T(\ldots), C(\ldots)) & \rightarrow \bigcup_i \mathcal{S}(\Phi(\mathcal{H}_T(\ldots), C_i, []), C(\ldots)) \\
 & \quad \textit{where } C_i \textit{ is a constructor of } T
\end{array}$$

where mgu($g_1, \ldots, g_n$) returns the most general unifier for all g_k. The last rule tries every constructor C_i of T as the output value of $f(\overline{x})$

For example, suppose that there is a uniform traversal combinator g that satisfies:

```
length(g(x,y)) = tc_list([]->tc_list([]->zero,[?,?,j]->succ(j))(y),
                         [?,?,i]->succ(i))(x)
```

where `length` is `tc_list([]->zero,[?,?,i]->succ(i))`. We want to find every g that satisfies the above equation. Let g be `tc_list(g1,g2)(x)`. From the promotion theorem we have:

```
1) length(g1) = tc_list([]->zero,[?,?,j]->succ(j))(y)
   let g1=tc_list(h1,h2)(y) then
   1.1) length(h1) = zero
          []->zero is the only component of length that returns zero => h1 = nil
   1.2) length(h2(c,s,j)) = succ(length(j))
          [?,?,i]->succ(i) is the only component of length that returns succ.
          Let h2(c,s,j)=cons(x1,x2) then length(cons(x1,x2))=succ(length(j))
          => succ(length(x2))=succ(length(j)) => h2(c,s,j) = cons(x1,j)
2) length(g2(b,r,i)) = succ(length(i)) => g2(b,r,i) = cons(x3,i)
```

Therefore, `g(x,y)` is:

```
tc_list([]->tc_list(nil,[?,?,j]->cons(x1,j))(y),[?,?,i]->cons(x3,i))(x)
```

where `x1` and `x3` are universally quantified variables that yield a class of solutions for `g(x,y)`.

8 Model Extensions

We can extend the definition of canonical types to include all recursively defined types. In that case, if a type T has a constructor $C(\ldots, x_i, \ldots)$, where x_i is of type $t(T)$, that is a type expression that refers to T, then instead of calling $\mathcal{H}_T$ recursively in the definition of $\mathcal{H}_T$ we call $\mathcal{M}_{T'}(\mathcal{H}_T)$, where $T'(\alpha) = t(\alpha)$, α is a type parameter, and $\mathcal{M}_{T'}$ is the generic map over T' [12]. A generic map of a canonical type $T(\alpha_1, \ldots, \alpha_n)$ is the function $\mathcal{M}_T(m_1, \ldots, m_n)$ that assigns a mapping m_i to each type parameter α_i. All theorems and algorithms for combinators are still valid because any generic map $\mathcal{M}_T(m_1, \ldots, m_n)$ is a uniform traversal combinator if all m_i are uniform traversal combinators. For example, the map over lists is `map_list(m)` and it is equal to `tc_list([]->nil,[a,?,r]->cons(m(a),r))`.

We can also extend our model to include higher order expressions: Let $T = T_1 \times \ldots \times T_n \rightarrow T_0$ be a function type. The uniform traversal combinator for T is $\mathcal{H}_T(c, r_1, \ldots, r_n) : T \rightarrow \beta$, where $c : T_0 \rightarrow \beta$ and $r_i : \rightarrow T_i$, and it is defined as $\lambda f.\ c(f(r_1(), \ldots, r_n()))$. Again variable f is not permitted to be an accumulative result variable. The promotion theorem for T is simply $g \circ \mathcal{H}_T(c, r_1, \ldots, r_n) = \mathcal{H}_T(g \circ c, r_1, \ldots, r_n)$. The composition algorithm is still correct because if h is a uniform traversal combinator so is $\mathcal{H}_T(c, r_1, \ldots, r_n)(h)$. That way we can prove high-order theorems, that is, for any function of a specific type.

In ADABTPL, a combinator for the type $T = T_1 \times \ldots \times T_n \rightarrow T_0$ is written as `tc_T1x...xTn_T0`. For example, the following proves that the list map distributes over append:

```
map_list(f)(append(x,y)) = append(map_list(f)(x),map_list(f)(y))
```

where `map_list(f)=tc_list([]->nil,[b,?,r]->tc_a_b([z]->cons(z,r),[]->b)(f))`. We apply the promotion theorem for `g(append(x,y))=tc_list(f1,f2)(x)`, where `g(x)=map_list(f)(x)`:

```
f1() = g(y) = map_list(f)(y)
f2(b,?,g(r)) = g(cons(b,r)) = tc_a_b([z]->cons(z,g(r)),[]->b)(f)
  => f2(b,?,u) = tc_a_b([z]->cons(z,u),[]->b)(f))
```

Therefore, `map_list(f)(append(x,y))` is

```
tc_list([]->map_list(f)(y),[b,?,u]->tc_a_b([z]->cons(z,u),[]->b)(f))(x)
```

which is the same with `append(map_list(f)(x),map_list(f)(y))`.

9 Acknowledgments

We are grateful to Neil Immerman and Sushant Patnaik for their help in analyzing the expressiveness of our language.

References

1. R. S. Bird. The Promotion and Accumulation Strategies in Transformational Programming. *ACM Transactions on Programming Languages and Systems*, 6(4):487–504, October 1984.
2. C. Bohm and A. Berarducci. Automatic Synthesis of Typed Λ-Programs on Term Algebras. *Theoretical Computer Science*, 39:135–154, 1985.
3. R. S. Boyer and J. S. Moore. *A Computational Logic*. Academic Press, New York, 1979.
4. L. Fegaras, T. Sheard, and D. Stemple. The ADABTPL Type System. In *Proceedings of the Second International Workshop on Database Programming Languages, Salishan, Oregon*, pages 243–254, 1989.
5. C. A. Hoare. Recursive Data Structures. *Journal of the ACM*, 4(2):105–132, June 1975.
6. P. Hudak. Conception, Evolution, and Application of Functional Programming Languages. *ACM Computing Surveys*, 21(3):359–411, September 1989.
7. N. Immerman, S. Patnaik, and D. Stemple. The Expressiveness of a Family of Finite Set Languages. *Proceedings of the Tenth ACM Symposium on Principles of Database Systems, Denver, Colorado*, pages 37–52, May 1991.
8. G. Malcolm. Homomorphisms and Promotability. In *Mathematics of Program Construction*, pages 335–347. Springer-Verlag, June 1989.
9. E. Meijer, M. Fokkinga, and R. Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *fifth ACM Conference on Functional Programming Languages and Computer Architecture, Cambridge Massachusetts*, pages 124–144, August 1991.
10. H. Rogers. *Theory of Recursive Functions and Effective Computability*. McGraw-Hill, 1967.
11. T. Sheard. Generalized Recursive Structure Combinators. Technical report, University of Massachusetts at Amherst, 1989. COINS Technical Report 89-26.
12. T. Sheard. Automatic Generation and Use of Abstract Structure Operators. *ACM Transactions on Programming Languages and Systems*, 19(4):531–557, October 1991.