Generating Programs by Reflection[*]

James Hook, Richard B. Kieburtz, and Tim Sheard[†]
Pacific Software Research Center
Oregon Graduate Institute of Science & Technology
19600 N.W. von Neumann Drive
Beaverton, OR 97006-1999
{hook,kieburtz,sheard}@cse.ogi.edu

CS/E 92-015

July 22, 1992

Abstract

Many algorithms derive their control from the inductive structure of a datatype. Polymorphic functions that realize *map*, generalized iteration and primitive recursion over sum-of-products datatypes can be generated by meta-functions applied to a type constructor's signature. Subject to a few easily checked restrictions, the type constructor of a freely constructed, sum-of-products datatype has the categorical structure of a monad. Functions derived from monadic datatypes have additional compositional properties useful in programming.

Meta-functions to calculate the monadic functions of sum-of-products datatypes have been implemented in a reflective language, TRPL, and are duplicated in Standard ML. The language extensions needed to support type-safe, compile-time reflection in a typed language are discussed.

As an example of program development, monads are used to derive an implementation of λ-calculus using de Bruijn indexing. In the derivation, a *map* function for the type of lambda-terms is modified with a "policy function" that captures the effects of variable binding in the lambda calculus. The resulting algorithms depend only on the monad operations of the datatype and are independent of details of the data structure, up to the final step of term contraction.

[*]A version of this paper has been submitted to the *Journal of Functional Programming.*

[†]The authors are supported in part by grants from the NSF (CCR-9101721), Tektronix, and OACIS.

1 Introduction

A great deal of programming consists of defining types for data structures, then writing programs that traverse these structures either to transform them or to calculate values of other types. If these types are defined carefully, so as to possess a few general properties, then their traversal operations can be generated automatically, from very compact specifications. There are three classes of operations that fit this mold.

1. **Control Structures** are given as abstract recursion schemes over types. An algorithm may be specified in terms of a parametric recursion scheme, independently of a particular datatype for which it may be instantiated. Certain properties of algorithms, such as termination, can be inferred from the parametric recursion scheme used. Two such control structures that we will describe are generalized *primitive recursion* and *bounded iteration.*

2. **Monads** provide a structuring mechanism that allows us to synthesize well behaved programs. A monad consists of a type constructor, T, and a fixed set of polymorphic operations related by a few equational laws. These operations are generic building blocks for well typed programs. Monads have been found to precisely describe various programming notions such as datatypes, exceptions, state, and continuations [Mog91]. The monad operations for a large class of types can be automatically generated.

3. **Distributive Morphisms** translate multi-structured objects into related forms. Given two type constructors, S and T, it is often convenient to have a polymorphic function *dist* with type $T(S(\alpha)) \rightarrow S(T(\alpha))$ which obeys certain laws. For some useful data types such distribution functions can be automatically generated.

In addition, the polymorphic traversal operators can provide control templates for data structure traversal functions that are not polymorphic.

4. **Policy Functions** are type-specific functional parameters that specialize polymorphic functions. Programs for many real application functions follow the same patterns of recursion found in generic control structures, but introduce local, type-specific, modifications that cannot be expressed in terms of polymorphic control structures. A mechanism that automates the specialization of these generic control structures can be extremely useful.

By programming in terms of generic operations that can be automatically generated, one can write program specifications that are considerably smaller and more abstract than customary executable specifications. Furthermore, such specifications are surprisingly insulated from the details of the types supporting the data structures in the program.

A programmer using this method will carefully define the data types required by the application, specify which operations over these types will be needed, and then "push a button" to generate these operations. In addition, a "generate and specialize" paradigm is used to capture those details not expressible as polymorphic functions. These operations are then composed to build the application program.

Monads and distributive morphisms are particularly helpful in constructing these compositions. The function-lifting operators of the monad can construct new functions whose types are agreeable to composition where an original function was not. The distributive morphisms provide "glue" for compositions of functions generated for monads corresponding to compound types.

1.1 Monads and program structure

Monads, a class of structures found in category theory, have attracted the interest of an enthusiastic group of advocates whose primary interests lie in programming methodology. When applied to functional program specification, monads provide uniform, natural rules for semantic extensions and composition of program components. Monads can characterize entire classes of parameterized types. The utility of monads for computer science was independently discovered by Spivey, who found use for them in providing a uniform framework for operators on datatypes and for control exceptions [Spi90], and by Moggi, who finds in them a framework capable of describing a diverse variety of notions of computation[Mog91]. We find the monad structure useful because it provides a formal characterization of the algebra of functions over parametric types [Wad89, Mai91].

Wadler has argued that monads can be used directly in programming. He claims the polymorphic functions which are part of any monad can be used to advantage as the building blocks of many programs[Wad92]. Use of these functions affords a high degree of design reuse. We seek to take the idea of monadic functions as building blocks one step further, by generating instances of these generic functions as needed.

In this paper we focus on monads that characterize a particular class of types: the freely generated, sum-of-products datatypes that are found in programming languages such as Standard ML, Miranda, Hope and Haskell. These types have inductive rules for term introduction. Corresponding to these rules are inductive control schemes for iterating over the structure of terms, or more generally, recursion schemes for these types. From a monadic characterization of such types, we shall define schematic functions that can be instantiated for any such type to yield a (terminating) recursion scheme for the type.

1.2 Monads defined

Monads are defined most generally in terms of categorical concepts. For our purposes, we shall fix the category to be a cartesian-closed one whose objects are types and whose arrows are functions. Having said that, we shall say no more about categories here.

A monad is a quadruple comprised of a type constructor T and three polymorphic functions:

$$(T, \mathit{map}, \mathit{unit}, \mathit{mult})$$

The three functions have typings

$$\begin{aligned} map^T &: (\alpha \to \beta) \to T(\alpha) \to T(\beta) \\ unit^T_\alpha &: \alpha \to T(\alpha) \\ mult^T_\alpha &: T(T(\alpha)) \to T(\alpha) \end{aligned}$$

Here, we have introduced a notational convention that will be used throughout the paper. When one of these function names is used apart from the monad definition, it may be superscripted with the name of the monad's type constructor and subscripted with a type name that (through the formulas above) indicates its typing. The superscript and subscript may be omitted when no confusion could arise.

These functions are further required to obey certain equational laws. The two conditions determining *map* are that it must preserve identities and compositions of functions:

$$\begin{aligned} map^T\, id_\alpha &= id_{T(\alpha)} \\ map^T\,(f \circ g) &= map^T\, f \circ map^T\, g \end{aligned}$$

Three more equations, called the *monad laws*, relate the three functions.

$$\begin{aligned} mult^T_\alpha \circ unit^T_{T(\alpha)} &= id_{T(\alpha)} \\ mult^T_\alpha \circ (map^T\ unit^T_\alpha) &= id_{T(\alpha)} \\ mult^T_\alpha \circ mult^T_{T(\alpha)} &= mult^T_\alpha \circ (map^T\ mult^T_\alpha) \end{aligned}$$

in which the subscripts indicate the types of particular instances of the polymorphic functions. These equations can be obtained by reading them from the following commutative diagrams:

$$\begin{tikzcd} T(\alpha) \arrow[d, "unit^T"'] \arrow[dr, equal] & \\ T^2(\alpha) \arrow[r, "mult^T"'] & T(\alpha) \end{tikzcd} \qquad \begin{tikzcd} T(\alpha) \arrow[d, "map^T\ unit^T"'] \arrow[dr, equal] & \\ T^2(\alpha) \arrow[r, "mult^T"'] & T(\alpha) \end{tikzcd}$$

$$\begin{tikzcd} T^3(\alpha) \arrow[r, "map^T\ mult^T"] \arrow[d, "mult^T"'] & T^2(\alpha) \arrow[d, "mult^T"] \\ T^2(\alpha) \arrow[r, "mult^T"'] & T(\alpha) \end{tikzcd}$$

There are several other, equivalent definitions of monads, which we shall not explore here[Wad92, DKM91].

The *unit* is a function that, given a value of type α, constructs a 'singleton' value of type $T(\alpha)$. The *map* is a function that calculates a pointwise extension of any function $f : \alpha \rightarrow \beta$ to a function over the elements of the constructed type, $T(\alpha)$. The multiplier, *mult*, 'flattens' a value whose type is that of the constructor applied twice. The prototypical example of a multiplier for datatypes is *reduce-append* for a list type. It flattens a list of lists by catenating all of the component lists into a single one.

In fact, one's intuition about monads never goes wrong when *List* is taken as an example of a type constructor. Its *unit* is the singleton list constructor, its *map* is the familiar `mapcar` of Lisp, and its multiplier has been described above. However, by framing these functions in the structure of a monad, we are led to realize that they are just instances of a more general principal that can be applied to many other type constructors, as well as to *List*.

2 Programs from Datatypes

This section describes how control structures, monad operations, and distributive morphisms can be generated from the details of a sum-of-products type definition.

2.1 Sum-of-Products types

A sum-of-products data type is defined by a recursive equation. Its left hand side consists of a type constructor applied to a p-tuple of distinct type variables, and its right hand side consists of an explicitly tagged disjoint sum. Each disjunct of the sum is a product of types tagged by a unique data constructor. The tuple of types $(t_{i,1} \times \ldots \times t_{i,m_i})$ which denotes the domain of the i^{th} data constructor, C_i, is called the *type signature* of C_i. The types we consider are freely generated by their data constructors, which is to say, there is no equational theory equating terms of the type that are tagged with distinct data constructors.

Let T be a type constructor, $\{\alpha_1, \ldots, \alpha_p\}$ be a finite set of type variables, and $\{C_1, \ldots, C_n\}$, be a finite set of data constructors. To define the type constructor T we write a declaration of the form

$$T(\alpha_1, \ldots, \alpha_p) = C_1(t_{1,1} \times \ldots \times t_{1,m_1}) \mid \ldots \mid C_n(t_{n,1} \times \ldots \times t_{n,m_n})$$

in which the type variables are universally quantified. The form of such a declaration is restricted to assure that a type constructor application, $T(t_1, \ldots, t_p)$ denotes a well-founded type. Each of the types $t_{i,j}$ in the type signature of C_i is required to be either: (1) $T(\alpha_1, \ldots, \alpha_p)$, or (2) one of $\{\alpha_1, \ldots, \alpha_p\}$, or (3) a type which does not depend upon T or on any of the type variables $\alpha_1, \ldots, \alpha_p$. [1]

[1]These conditions are more restrictive than necessary. However, composite type constructions require additional mechanism, some of which will be introduced in a later section, and mutually recursive type equations would require more complex restrictions to guarantee well-foundedness.

Two examples of such type definitions, which are used as running examples, follow:

$$\mathit{List}(\alpha) = \mathit{Cons}(\alpha \times \mathit{List}(\alpha)) \mid \mathit{Nil}$$

$$\mathit{Term}(\alpha,\beta) = \mathit{Var}(\alpha) \mid \mathit{App}(\mathit{Term}(\alpha,\beta) \times \mathit{Term}(\alpha,\beta)) \mid \mathit{Abs}(\beta \times \mathit{Term}(\alpha,\beta))$$

The second defines a type which represents terms in the pure lambda calculus[2]. In $\mathit{Term}(\alpha,\beta)$, α is the type of names for free variables, and β is the type of abstracted names.

2.2 Control structures

Recursive control structures can be derived from the inductive definitions of sum-of-products types. We consider two such schemes here, iteration and primitive recursion. The iteration recursion scheme generalizes iteration over the natural numbers. We call such a recursion scheme a *reduction*. The primitive recursion scheme is slightly more general.

2.2.1 The reduce operator

Let T be a sum-of-products type constructor with n data constructors. The reduce function for T is a function of 2 arguments. The first argument is a tuple of n functional arguments, called accumulators, $(f_1, \ldots, f_n)$. The curried application $\mathit{red}^T\ (f_1, \ldots, f_n)$ has type $T(\alpha_1, \ldots, \alpha_p) \rightarrow \omega$. The reduce function can be defined by n equations that give the mapping on terms constructed by each of the n data constructors. The general form is:

$$\mathit{red}^T\ (f_1, \ldots, f_n)\ (C_i(x_1, \ldots, x_{m_i})) \ = \ f_i(e_1, \ldots, e_{m_i})$$

$$\text{where } e_j \ = \ \begin{cases} \mathit{red}^T\ (f_1, \ldots, f_n)\, x_j & \text{if } x_j \text{ has type } T(\alpha_1, \ldots, \alpha_p) \\ x_j & \text{if } x_j \text{ has any other type} \end{cases}$$

In case a constructor, C_i, is nullary, then f_i takes the empty tuple as an argument.

The types of the accumulator functions, $(f_1, \ldots, f_n)$, required by red^T are determined by the following rules. For each data constructor, C_i, in the definition for T, if C_i has the type $(t_{i,1} \times \ldots \times t_{i,m_i}) \rightarrow T(\alpha_1, \ldots, \alpha_p)$ then the corresponding accumulator has the typing

$$f_i \ : \ (\sigma_{i,1} \times \ldots \times \sigma_{i,m_i}) \rightarrow \omega$$

$$\text{where } \sigma_{i,j} \ = \ \begin{cases} \omega & \text{if } t_{i,j} = T(\alpha_1, \ldots, \alpha_p) \\ t_{i,j} & \text{otherwise} \end{cases}$$

Returning to the examples, the reduce function for *List* has a recursion scheme defined by two equations,

$$\begin{aligned} \mathit{red}^{\mathit{List}}(f_{\mathit{Cons}}, f_{\mathit{Nil}})\, \mathit{Nil} \ &= \ f_{\mathit{Nil}}() \\ \mathit{red}^{\mathit{List}}(f_{\mathit{Cons}}, f_{\mathit{Nil}})\, (\mathit{Cons}(x, xs)) \ &= \ f_{\mathit{Cons}}(x, \mathit{red}^{\mathit{List}}(f_{\mathit{Cons}}, f_{\mathit{Nil}})\, xs) \end{aligned}$$

[2] Its derivation is given in section 4.3.

and the reduce function for *Term* satisfies the three equations

$$\begin{aligned}
red^{Term}(f_V, f_{Ap}, f_{Ab})\,(Var(x)) &= f_V(x) \\
red^{Term}(f_V, f_{Ap}, f_{Ab})\,(App(x,y)) &= f_{Ap}(red^{Term}(f_V, f_{Ap}, f_{Ab})\,x,\ red^{Term}(f_V, f_{Ap}, f_{Ab})\,y) \\
red^{Term}(f_V, f_{Ap}, f_{Ab})\,(Abs(x,y)) &= f_{Ab}(x,\ red^{Term}(f_V, f_{Ap}, f_{Ab})\,y)
\end{aligned}$$

2.2.2 The primitive recursion operator

The primitive recursion control structure is more general than the reduction control structure in that its accumulating functions may access both the original and recursively transformed versions of its recursive arguments. We use the convention that if a data constructor, C_i, has m arguments, r of which have type $T(\alpha_1, \ldots, \alpha_p)$, then the corresponding accumulator has m arguments, r of which are pairs.

To specify pr^T we use the template:

$$pr^T\,(f_1, \ldots, f_n)\,(C_i(x_1, \ldots, x_{m_i})) = f_i(e_1 \ldots e_{m_i})$$

$$\text{where } e_j = \begin{cases} (x_j,\ pr^T\,(f_1, \ldots, f_n)\,x_j) & \text{if } x_j \text{ has type } T(\alpha_1, \ldots, \alpha_p) \\ x_j & \text{otherwise.} \end{cases}$$

For example, the primitive recursion function for *List* is defined by the two equations:

$$\begin{aligned}
pr^{List}(f_{Cons}, f_{Nil})\,Nil &= f_{Nil}() \\
pr^{List}(f_{Cons}, f_{Nil})\,(Cons(x, xs)) &= f_{Cons}(x, (xs, pr^{List}(f_{Cons}, f_{Nil})\,xs))
\end{aligned}$$

2.3 Monad operations

Not every sum-of-products type supports the structure of a monad. A few restrictions on the recursive type equation defining a type are needed. This section describes these restrictions and gives rules for generating the monad operations for a type meeting them. Appendix A contains a proof that the operations generated from these rules obey the monad laws.

2.3.1 The map operator

Let $T(\alpha_1, \ldots, \alpha_p)$ be a sum-of-products type of n data constructors $\{C_1, \ldots, C_n\}$. The *map* for $T(\alpha_1, \ldots, \alpha_p)$ on α_k (designated as map_k^T) is a function of two arguments with type $(\alpha_k \rightarrow \beta) \rightarrow T(\alpha_1, \ldots, \alpha_p) \rightarrow T(\alpha_1, \ldots, \alpha_{k-1}, \beta, \alpha_{k+1}, \ldots, \alpha_p)$. It can be realized as a reduction by supplying the proper accumulating function arguments.

$$map_k^T\ f\ x = red^T\,(g_1, \ldots, g_n)\,x$$

$$\text{where } g_h(y_i, \ldots, y_{m_h}) = C_h(e_1, \ldots, e_{m_h})$$

$$\text{in which } e_j \;=\; \begin{cases} f\ y_j & \text{if } y_j \text{ has type } \alpha_k \\ y_j & \text{if } y_j \text{ has any other type} \end{cases}$$

A function template defining map_k^T by a set of recursive equations can be derived from the reduction template by using properties of red^T:

$$map_k^T\ f\ (C_i(x_1, \ldots, x_{n_i})) \;=\; C_i(e_1, \ldots, e_{n_i})$$

$$\text{where } e_j \;=\; \begin{cases} map_k^T\ f\ x_j & \text{if } x_j \text{ has type } T(\alpha_1, \ldots, \alpha_p) \\ f\ x_j & \text{if } x_j \text{ has type } \alpha_k \\ x_j & \text{if } x_j \text{ has any other type} \end{cases}$$

The map for *List* satisfies two equations:

$$\begin{aligned} map^{List}\, f\ Nil &= Nil \\ map^{List}\, f\ (Cons(x, xs)) &= Cons(f\ x, map^{List}\ f\ xs) \end{aligned}$$

For $Term(\alpha, \beta)$ there are two maps, one on α, and one on β.

$$\begin{aligned} map_1^{Term}\ f\ (Var(x)) &= Var(f\ x) \\ map_1^{Term}\ f\ (App(x, y)) &= App(map_1^{Term}\ f\ x,\ map_1^{Term}\ f\ y) \\ map_1^{Term}\ f\ (Abs(x, y)) &= Abs(x,\ map_1^{Term}\ f\ y) \end{aligned}$$

$$\begin{aligned} map_2^{Term}\ f\ (Var(x)) &= Var(x) \\ map_2^{Term}\ f\ (App(x, y)) &= App(map_2^{Term}\ x,\ map_2^{Term}\ f\ y) \\ map_2^{Term}\ f\ (Abs(x, y)) &= Abs(f\ x,\ map_2^{Term}\ f\ y) \end{aligned}$$

2.3.2 Zero and unit constructors

If the data constructor, C_z, is nullary then C_z is called a *zero constructor* or a zero of T. In the examples above, *Nil* is the only zero constructor.

If the type signature $(t_1 \times \ldots \times t_{m_u})$ of a data constructor, C_u, has ***exactly one*** type, t_i, equal to the k^{th} type variable bound in the declaration of the type constructor, α_k, then C_u is called a *unit constructor* of T on the k^{th} type variable. We shall abbreviate this name as "a unit of T on k" and when T has only a single type parameter we will often omit the k. In the case that C_u is unary (there is no type other than α_k in C_u's type signature), then C_u is called a *perfect unit constructor* of T on k. Note that in general, there may be multiple unit constructors for any of the parameters of a type constructor T.

In the examples, *Cons* is a non-perfect unit constructor for *List*. For the type constructor *Term*, the data constructor *Var* is a perfect unit constructor on 1, and *Abs* is a non-perfect unit constructor on 2.

2.3.3 Zero-based and unitary types

A type constructor T is *zero-based* if there is exactly one zero constructor and every term of the type embeds at least one instance of that zero. Equivalently, if $T(\alpha_1, \ldots, \alpha_p)$ has a unique zero constructor C_z, and each of the non-zero constructors, C_i, has a type signature $(t_1 \times \ldots \times t_{n_i})$, in which at least one of the component types t_j is $T(\alpha_1, \ldots, \alpha_p)$, then T is zero-based. In the examples above, *List* is zero-based since every list term embeds one occurrence of *Nil*.

A type constructor T is *unitary* on k if there exists a unique polymorphic function of type $\alpha_k \rightarrow T(\alpha_1, \ldots, \alpha_p)$ that is linear in its argument. This requires that T has exactly one unit constructor, C_u on k, and can be satisfied in one of two ways.

1. Either C_u is a perfect unit constructor on k, or
2. C_u is a non-perfect unit constructor of T on k and
   (a) T is zero-based, and
   (b) each type component, $t_{u,j}$ in the type signature of C_u is either α_k or is $T(\alpha_1, \ldots, \alpha_p)$.

Note that in general, a type may be unitary on each of its type variables.

2.3.4 The unit operator

If T is a unitary type on k then the unit function for T, of type $\alpha_k \rightarrow T(\alpha_k)$, is defined to be:

$$unit_k^T(x) = C_u(x) \text{ if } C_u \text{ is a perfect unit constructor for } T,$$

or if C_u is not a perfect unit constructor, then

$$unit_k^T(x) = C_u\ (y_1, \ldots, y_{n_u})$$

$$\text{where } y_j \ = \ \begin{cases} x & \text{if } y_j \text{ has type } \alpha_k \\ C_z & \text{if } y_j \text{ has type } T(\alpha_1, \ldots, \alpha_p) \end{cases}$$

When T has only one type variable the subscript, k, will often be omitted.

For the *List* example the non-perfect unit constructor, *Cons*, forms a unit function with the zero constructor, *Nil*, i.e. $unit^{List}(x) = Cons(x, Nil)$. *Term* is unitary on 1 because *Var* is a perfect unit constructor. On the other hand, *Abs* cannot be used to construct a unit function for terms on 2, since *Abs* is not a perfect unit constructor and there is no zero constructor for *Term*.

2.3.5 Zero replacement

The third monad operator, *mult*, flattens 2-level monad structured values to 1-level monad structured values. For a unitary type with a perfect unit constructor, the flattening operation is obvious, but if the unit constructor is non-perfect, care must be taken that the flattening operation preserves structure as it embeds the second level monadic value as a substructure of the first. The function that does this is interesting in its own right and has many important properties. We call it a *zero replacement* function.

Let $T(\alpha_1, \ldots, \alpha_p)$ be a zero-based sum-of-products type. The right-biased, zero replacement for T is designated by the infix operator, $\oplus_R$. We call $(x \oplus_R y)$ a zero replacement, since it replaces the rightmost zero in x with y. The type of $\oplus^T_R$ is $(T(\alpha_1, \ldots, \alpha_p) \times T(\alpha_1, \ldots, \alpha_p)) \rightarrow T(\alpha_1, \ldots, \alpha_p)$ and it is defined by the equations:

$$C_z \oplus^T_R y = y$$

$$(C_i(x_1, \ldots, x_{n_i})) \oplus^T_R y = C_i(e_1, \ldots, e_{n_i})$$

$$\text{where } e_j = \begin{cases} x_j \oplus^T_R y & \text{if } x_j \text{ is the rightmost parameter with type } T(\alpha_1, \ldots, \alpha_p) \\ x_j & \text{otherwise} \end{cases}$$

That is, for the rightmost argument of type $T(\alpha_1, \ldots, \alpha_p)$ there is a recursive call to the section $(\oplus^T_R y)$, while all other arguments remain unchanged.

In a similar fashion the left-biased zero replacement, $\oplus^T_L$, is defined by

$$y \oplus^T_L C_z = y$$

$$y \oplus^T_L (C_i(x_1, \ldots, x_{n_i})) = C_i(e_1, \ldots, e_{n_i})$$

$$\text{where } e_j = \begin{cases} y \oplus^T_L x_j & \text{if } x_j \text{ is the leftmost parameter with type } T(\alpha_1, \ldots, \alpha_p) \\ x_j & \text{otherwise} \end{cases}$$

Picking the leftmost or rightmost argument of C_i with type $T(\alpha_1. \ldots, \alpha_p)$ builds zero replacement functions that combine their arguments in a linear fashion. This is an important property necessary for the multiplier (which will be built with zero replacements) to meet the second monad law.

Of our two examples, only *List* is zero-based and has a zero replacement function. Since the unit constructor *Cons* has only a single parameter of type *List*(α), the rightmost such parameter and the leftmost are the same. and thus $a \oplus^{List}_R b = b \oplus^{List}_L a$.

$$\begin{aligned} Nil \oplus^{List}_R ys &= ys \\ (Cons(x, xs)) \oplus^{List}_R ys &= Cons(x, xs \oplus^{List}_R ys) \end{aligned}$$

We recognize from these equations that $\oplus^{List}_R$ is the list append operator. Note that for natural numbers defined by the type equation $N = 0 \mid S(N)$, the addition operator is also a zero replacement. Important properties of zero replacement functions are that they are associative, and that they have the zero, C_z, for both a left and right identity. These properties are proved in Appendix A.

2.3.6 The multiplier operator

Let $T(\alpha_1, \dots, \alpha_p)$ be a freely constructed sum-of-products type unitary on k. The *multiplier* for $T(\alpha_1, \dots, \alpha_p)$ on the k^{th} type variable is a function of one argument, with type $T(\alpha_1, \dots, \alpha_{k-1}, T(\alpha_1, \dots, \alpha_p), \alpha_{k+1}, \dots, \alpha_p)) \rightarrow T(\alpha_1, \dots, \alpha_p)$ which converts a 2-level monadic value to a simpler 1-level monadic value.

The multiplier may be realized as a reduction. The tuple of accumulating functions required for the reduction is comprised of the corresponding data constructors, except that a *linearizing* function, *link*, substitutes for the unit constructor. The multiplier can be implemented by an equation of the form:

$$mult_k^T\ x \ = \ red^T\ (C_1, \dots, C_{u-1}, link^T, C_{u+1}, \dots, C_n)\ x$$

where $link^T$ is described below.

If $C_u(t_{u,1}, \dots, t_j, \dots, t_{u,m_u})$ is a unit constructor, in which t_j is the unique argument with type α_k, and for all $i \neq j$, (if there are any such) t_i has type $T(\alpha_1, \dots, \alpha_p)$, then $link^T$ can be defined by

$$link^T\ (x_1, \dots, x_j, \dots, x_{m_u}) \ = \ x_1 \oplus_L^T \dots \oplus_L^T\ x_j \oplus_R^T \dots \oplus_R^T\ x_{m_u}$$

All the arguments to the left of the j^{th} index are linked to the j^{th} argument with the left-biased zero replacement operator, and those arguments to the right of the j^{th} index are linked to the j^{th} argument with the right-biased zero replacement operator. If x_j is the only argument, (i.e., C_u is a perfect unit constructor), then $link^T$ is the identity function.

The linearizing function, $link^T$, was contrived to get a new $T(\alpha_1, \dots, \alpha_p)$ object from m objects of this type, where m is the arity of the unit constructor, C_u. Think of $link^T$ as an m-ary combining function for $T(\alpha_1, \dots, \alpha_p)$ objects, which combines them in a *linear*, order-preserving fashion. The property that C_z is an identity for $\oplus_L^T$, and $\oplus_R^T$, and the linearization property of $\oplus_L^T$ and $\oplus_R^T$ are necessary to prove the second monad law.

Using properties of the *reduce* operator we can derive explicit equations for $mult_k^T$. On terms tagged with the unit constructor for k,

$$mult_k^T\ C_z \ = \ C_z$$

$$mult_k^T\ C_u(x_1, \dots, x_j, \dots, x_{m_u}) \ = \ (mult_k^T\ x_1) \oplus_L^T \dots \oplus_L^T\ x_j \oplus_R^T \dots \oplus_R^T\ (mult_k^T\ x_{m_u})$$

where x_j is the unique parameter with type α_k, the k^{th} type parameter of T. On terms tagged with other constructors,

$$mult_k^T\ (C_i(x_1, x_2, \dots, x_{m_i})) = C_i(e_1, \dots, e_{m_i})$$

$$\text{where } e_j \ = \ \begin{cases} mult_k^T\ x_j & \text{if } x_j \text{ has type } T(\alpha_1, \dots, \alpha_p) \\ x_j & \text{otherwise} \end{cases}$$

Since *List* does not have a perfect unit we first define $link^{List}$, then use it to define $mult^{List}$.

$$link^{List}(x,y) = x \oplus_R^{List} y$$

$$mult^{List}\, x = red^{List}(\, link_R^{List}, (\lambda() \,.\, Nil)\,)\, x$$

Since *Term* has a perfect unit, its multiplier uses the identity function as a linearizing function.

$$mult_1^{Term}\, x = red^{Term}(id, App, Abs)\, x$$

Using the properties of red^{Term}, we calculate an explicit, recursive definition of $mult_1^{Term}$:

$$\begin{aligned} mult_1^{Term}\,(\,Var(x)) &= x \\ mult_1^{Term}\,(App(x,y)) &= App(mult_1^{Term}\, x,\, mult_1^{Term}\, y) \\ mult_1^{Term}\,(Abs(x,y)) &= Abs(x,\, mult_1^{Term}\, y) \end{aligned}$$

2.3.7 Datatypes and monads

Theorem 1 *Let T be a freely constructed, sum-of-products type constructor. If T is unitary on its k^{th} type argument then the quadruple $(T, map_k^T, unit_k^T, mult_k^T)$ has the structure of a monad.*

Proof is given in Appendix A.

2.4 Distributive morphisms and composite monads

Under certain conditions, monads can be composed to form new monads. For the monads that correspond to sum-of-products datatypes, the conditions needed for monad composition are that certain distributive morphisms exist. These morphisms are functions whose types are reminiscent of distributive laws for algebras. They provide the "glue" that joins the *unit* and *mult* of individual monads into the *unit* and *mult* of a composite monad. However, the laws governing distributive morphisms are quite stringent.

Let the type $T(\alpha) = R(S(\alpha))$, where $(R, map^R, unit^R, mult^R)$ and $(S, map^S, unit^S, mult^S)$ are the monads of sum-of-products type constructors. In general R and S can be type constructors of more than one type variable; this simply clutters the notation but does not affect the result in any material way. Then the S-distribution function for R, π_R^S, is a function of one argument with type $S(R(\alpha)) \rightarrow R(S(\alpha))$, and which satisfies[BW85]:

$$\begin{aligned} \pi_R^S \circ unit^S &= map^R\, unit^S & (1) \\ \pi_R^S \circ (map^S\, unit^R) &= unit^R & (2) \\ \pi_R^S \circ mult^S &= (map^R\, mult^S) \circ \pi_R^S \circ (map^S\, \pi_R^S) & (3) \\ \pi_R^S \circ (map^S\, mult^R) &= mult^R \circ (map^R\, \pi_R^S) \circ \pi_R^S & (4) \end{aligned}$$

Proposition 2 *Given monads* $(R, map^R, \mathrm{unit}^R, \mathrm{mult}^R)$ *and* $(S, map^S, \mathrm{unit}^S, \mathrm{mult}^S)$ *with a distribution morphism* π_R^S *satisfying the above equations, the quadruple*

$$(RS,\ map^R \circ map^S,\ unit^R \circ unit^S,\ (map^R\, mult^S) \circ mult^R \circ (map^R\, \pi_R^S))$$

is a monad.

Proof is given in Appendix B.

Unfortunately, it is not obvious how to construct a distributive morphism satisfying equations (1–4) unless R is a particularly simple sum-of-products type[3]. We do not even know if such a function is computable, in general. The following section gives a construction for a distribution over products that is useful in the construction of a type distribution morphism for a restricted type constructor, R.

2.4.1 Product distributions

An n-ary product distribution function for $T(\alpha_1, \ldots, \alpha_p)$ on k, τ_n^T, is a function with type $(T(\alpha_1 \ldots \alpha_{k-1}, \beta_1, \alpha_{k+1} \ldots \alpha_p) \times \ldots \times T(\alpha_1 \ldots \alpha_{k-1}, \beta_n, \alpha_{k+1} \ldots \alpha_p)) \to T(\alpha_1 \ldots \alpha_{k-1}, (\beta_1 \times \ldots \times \beta_n), \alpha_{k+1} \ldots \alpha_p)$. It maps an n-tuple of T-objects to a T-object of n-tuples.

Let $T(\alpha_1, \ldots, \alpha_p)$ be unitary on k, with operations $unit_k^T$, map_k^T, and $mult_k^T$, then the *comprehension* notation[Wad90] makes it particularly easy to express the the n-ary product distribution function for T on k, τ_n^T.

$$\tau_n^T(x_1 \ldots x_n) \;=\; \{\, (a_1 \ldots a_n) \mid a_1 \leftarrow x_1;\ \ldots\ ; a_n \;\leftarrow x_n\}^T$$

The comprehension notation is defined in terms of the familiar monad operations.

$$\begin{aligned}
\{\, t\,\}^T &= unit^T t \\
\{\, t \mid x \leftarrow u\,\}^T &= map^T\, (\lambda\, x \,.\, t)\, u \\
\{\, t \mid (p; qs)\}^T &= mult^T\ \{\ \{\, t \mid qs\}^T \mid p\}^T
\end{aligned}$$

The notation $(p; qs)$ denotes a sequence of $(x_i \leftarrow u_i)$ expressions, the first of which is p, with the rest being designated by qs.

Note that τ_n^T could have been defined differently, as:

$$\tau_n^T(x_1 \ldots x_n) \;=\; \{\, (a_1 \ldots a_n) \mid a_n \leftarrow x_n; \ldots ; a_1 \;\leftarrow x_1\}^T$$

or by any other equation which places the $(a_i \leftarrow x_i)$'s in a different order. Note also that $\tau_1^T \;=\; map^T\, (\lambda a \,.\, a)$ which is the identity function.

For example, let the type constructor *Maybe*[Spi90] be defined by the equation

$$Maybe(x) \;=\; Nothing \mid Just(x)$$

[3]Similar observations have been made by the Charity group [Fuk92].

The binary product distribution for *Maybe*, with type $(Maybe(a) \times Maybe(b)) \rightarrow Maybe(a \times b)$, can be defined as:

$$\tau_2^{Maybe}(x_1,\, x_2) \;=\; \{\, (a_1, a_2) \mid a_1 \leftarrow x_1;\; a_2 \;\leftarrow x_2 \}^{Maybe}$$

Translating the comprehension expression using the rules above, we get:

$$\tau_2^{Maybe}(x_1,\, x_2) \;=\; mult^{Maybe}\,(map^{Maybe}\,(\lambda\, a_1 \,.\, (map^{Maybe}\,(\lambda\, a_2 \,.\, (a_1, a_2))\, x_2))\, x_1)$$

Using the definitions of map^{Maybe} and $mult^{Maybe}$, calculated using the rules of Sec. 2.3, this definition simplifies to four explicit equations.

$$\begin{array}{rcl}
\tau_2^{Maybe}(Nothing, Nothing) & = & Nothing \\
\tau_2^{Maybe}(Just(x_1), Nothing) & = & Nothing \\
\tau_2^{Maybe}(Nothing, Just(x_2)) & = & Nothing \\
\tau_2^{Maybe}(Just(x_1), Just(x_2)) & = & Just((x_1, x_2))
\end{array}$$

2.4.2 Type composition distributions

Definition: A sum-of-products data type $R(\alpha)$ such that every data constructor is at most unary in either of α or $R(\alpha)$ (but cannot have both types in its signature) is said to be *linearly-constructed.*

Let R be a linearly-constructed type constructor with a product distribution[4] and let the type $T(\alpha) \;=\; S(R(\alpha))$, where S is any sum-of-products type (not necessarily unitary). (For a linearly constructed type, the arbitrary order in which elements are generated from type $R(\alpha)$ values in the product distribution function is moot, as each constituent of type $R(\alpha)$ contributes at most one element.) Then there is a straightforward construction of the S-distribution for R, In particular, if both R and S are also unitary, and hence correspond to monads, then there is also a monad corresponding to T. In spite of the severe restriction on the form of R, there is at least one interesting example of such a type constructor, namely *Maybe.*

A construction for π_R^S can be given in terms of the function red^S,

$$\pi_R^S\, x \;=\; red^S\,(f_1 \ldots f_n)\, x$$

if an accumulating function, f_i, can be found for each data constructor, C_i, of S. If C_i is a zero constructor, C_z, then $f_i\;() \;=\; unit^R\; C_z$. If C_i is not a zero constructor, but has type $(\sigma_1 \ldots \sigma_{m_i}) \rightarrow S(\alpha)$ then the the corresponding accumulating function, f_i, can be defined as

$$f_i(x_1 \ldots x_{m_i}) \;=\; map^R\; C_i\;\; (\tau_{m_i}^R(e_1 \ldots e_{m_i}))$$

[4]There are monadic types such as state transformers, which possess product distribution functions, but cannot be expressed as a sum-of-products type because they require a function space (exponential) type.

$$\text{where } e_j \;=\; \begin{cases} x_j & \text{if } \sigma_j \text{ is } S(\alpha) \quad \text{i.e. } e_j : R(S(\alpha)) \\ x_j & \text{if } \sigma_j \text{ is } \alpha \quad\quad\;\; \text{i.e. } e_j : R(\alpha) \\ unit^R\ x_j & \text{if } \sigma_j \text{ is any other type} \end{cases}$$

For example, the type composition distribution function, π^{List}_{Maybe}, is a function with type $List(Maybe(\alpha)) \rightarrow Maybe(List(\alpha))$, and can be defined by using the reduction for lists, red^{List}.

$$\pi^{List}_{Maybe}\ x \;=\; red^{List}\,(f_{Nil}, f_{Cons})\ x$$

$$\text{where } \begin{cases} f_{Cons}(x, xs) & = \; map^{Maybe}\ Cons\ \ (\tau_2^{Maybe}(x, xs)) \\ f_{Nil}() & = \; unit^{Maybe} Nil \;=\; Just(Nil) \end{cases}$$

Appealing to the definition of τ_2^{Maybe} from section 2.4.1 we see we can simplify the definition of f_{Cons} to the four equations

$$\begin{aligned} f_{Cons}(Nothing, Nothing) &= Nothing \\ f_{Cons}(Just(x_1), Nothing) &= Nothing \\ f_{Cons}(Nothing, Just(x_2)) &= Nothing \\ f_{Cons}(Just(x_1), Just(x_2)) &= Just(Cons(x_1, x_2)) \end{aligned}$$

Intuitively π^{List}_{Maybe} yields *Nothing* if there are any *Nothing*s in its argument, and otherwise yields $Just(l)$, where l is a list composed of all the unexceptional elements in its argument.

3 Meta-level Programming

The theoretical framework developed in Section 2 offers results on the algebraic structure of data types that can be directly applied to transform specifications into software. For example, given any data type constructor T, the polymorphic function map^T can be synthesized algorithmically by analyzing its signature [MFP91]. In a programming language that incorporates both expression and type representations as first-class values, one can write a single meta-function, *MAP*, that when applied to a type constructor T, generates the function map^T.

Here is how *MAP* is defined for a free, sum-of-products data type. A type constructor can be defined by a signature, Σ^T, with a set of free type variables, $\{\alpha_1, \ldots, \alpha_p\}$ (where $p \geq 1$), as

$$T \;=\; [\alpha_1, \ldots, \alpha_p]\Sigma^T$$

The signature corresponds directly to a data type definition in Standard ML, for instance. Meta-functions use the syntactic representations of definitions as data and produce new definitions as results. The meta-function

$$MAP\ :\ Signature \rightarrow Ident \rightarrow Int \rightarrow Expr$$

is applied to a signature, an identifier and an integer index, k, to produce a program defining a function map^T_k. Here, *Expr* is the type of syntactic expressions in a programming language.

The identifier argument is an internal name that will be used to indicate the recursion variable in the program scheme. When an application of *MAP* is evaluated to an expression, identifiers introduced in the expression are named so as to avoid conflict with the identifier given for the recursion variable. The integer index, k, indicates which of the p type variables is the object of the *map*.

To get the meaning of a recursive program, interpret it in the standard semantics of the programming language, obtaining a function of the specified type,

$$map^T =_{\text{def}} [\![MAP([\alpha_1,\ldots,\alpha_p]\Sigma^T)\text{“}m\text{”}\, k]\!] : (\alpha_k \to \beta) \to T(\alpha_1,\ldots,\alpha_k,\ldots,\alpha_p) \to T(\alpha_1,\ldots,\beta,\ldots,\alpha_p)$$

3.1 Specializing recursive programs

Programs for real applications use type-specific functions as well as generic ones. A great many type-specific functions appear to follow nearly the same recursion scheme as a generic *map*, *reduce*, or primitive recursion for some data type, but differ in some detail. For instance, a function that filters elements from a list by applying a test predicate q follows a recursion scheme much like that of map^{List}. Comparing

$$\begin{aligned} map^{List} f\, Nil &= Nil \\ map^{List} f\,(Cons(x, xs)) &= Cons(f\,x,\ map^{List} f\, xs) \end{aligned}$$

with

$$\begin{aligned} filter^{List} q\, Nil &= Nil \\ filter^{List} q\,(Cons(x, xs)) &= \text{if } q\,x \text{ then } Cons(x,\ filter^{List} q\, xs) \\ &\quad\ \text{else } filter^{List} q\, xs \end{aligned}$$

we notice that $filter^{List}$ resembles map^{List} in its recursive control scheme except for the conditional expression that applies the policy predicate $q : \alpha \to \text{Bool}$ to each list element. The predicate, unless trivial, is necessarily a type-specific function, thus $filter^{List} q$ cannot be polymorphic in α.

Ordinarily, textual modification of an existing program scheme to obtain a new program is done with a text editor. All semantic properties of the old program, from its typing to its termination, must be formally reestablished for the new one. We should like to be able to restrict editing so that a generic function could be modified by embedding a policy function, yet at the same time, be assured that certain of its properties will be preserved.

When generic functions are generated from schematic meta-functions applied to signatures, there are many ways that modifications may be constrained. To provide a mechanism, modifications are made with a syntactic difference operator, $\Delta : (Expr \times (Expr \times Expr)) \to Expr$. The left component of its argument is the generic expression to be modified. The right component of its argument is a pattern-activated rewrite rule to be applied to the left component. The difference operator incorporates restrictions that determine applicability of the rewrite rule.

For a rewrite rule to be acceptable, both its sides must belong to a common phylum in the syntax of the programming language, so that the syntactic correctness of an expression will be preserved under the rewrite. The difference operator, like other meta-functions, uses an abstract syntax representation of its arguments. With such a representation it is easy to verify this syntactic constraint. The difference operator also invokes the type-inference mechanism of the programming language to assure that well-typing of the recursion scheme is preserved by the rewrite. The type of the target expression of the rewrite must be an instance of the principal type of its pattern.

The difference operator that we illustrate here applies only to expressions that have the form of `let` definitions. It adds the identifier defined in the `let` form to the current lexical scope. Any other binding operators that may occur in the expression to be modified are treated as local bindings. The identifiers they introduce are renamed to avoid possible capture of free variables that may occur in the target of the rewrite rule argument.

However, neither syntactic correctness nor type preservation is a strong enough restriction to provide a sense in which a modified program uses a recursion scheme similar to the original one. To capture that aspect, we further require that the rewrite rule used by the difference operator must satisfy two conditions:

(a) in any application of the recursion variable that occurs in the target, the $T(a_1, \ldots, a_p)$-typed arguments of the application must be exactly those bound in the pattern, and

(b) the resulting program will evaluate no more applications of the recursion variable than does the original.

Condition (a) requires, for instance, that in a specialization of map^{List}, if an application of the recursion variable in the pattern, $(m\ f\ xs)$ (where xs has type $List(\alpha)$), is replaced by an expression E in the target, that each occurrence of the recursion variable in E must be in an application of the form $(m\ f'\ xs)$, differing from the form in the target only in that f' may differ from f.

Condition (b) tells us that a recursion scheme can undergo weakening by the rewriting, but that its formal structure cannot be otherwise modified. Note that condition (b) has not been stated as a syntactic restriction on the number of occurrences of applications of the recursion variable. A target expression may duplicate such applications on the separate arms of a conditional, for instance, which increases the number of syntactic occurrences but will not increase the number of recursive applications that can be evaluated. The difference operator can enforce a syntactic restriction on the rewrite rule that is sufficient to assure these two conditions. These conditions guarantee that if the original program terminates, so will the modified program.

To obtain a new program scheme that applies a policy function, we need to specify a rewrite rule $P : Expr \rightarrow Expr \times Expr$ and an expression for a policy function, $z : Expr$. The left-hand member of a rewrite rule is a pattern, possibly containing single occurrences of designated expression variables, and the right-hand member is a replacement expression that may contain occurrences of the expression variables introduced in the pattern. A modification of a target term occurs by replacing a subterm of the target matching the pattern with an instance of the replacement term, in which bindings to the expression variables are determined by the pattern match. If more than one subterm of the target is matched by the pattern, the modification is undefined.

To calculate a more specialized function from a *map*, for instance, a modification rule is

$$\frac{m \;:\; \mathit{Identifier} \qquad P \;:\; \mathit{Expr} \rightarrow \mathit{Expr} \times \mathit{Expr} \qquad z \;:\; \mathit{Expr}}{(\mathit{MAP}\,([\alpha_1, \ldots, \alpha_p]\Sigma^T)\; m\; k)\; \Delta\; P(z) \;:\; \mathit{Expr}}$$

Upon abstracting on the policy function variable and taking the semantics of the resulting expression, we get

$$\frac{m \;:\; \mathit{Identifier} \qquad P \;:\; \mathit{Expr} \rightarrow \mathit{Expr} \times \mathit{Expr}}{\mathit{modified\_map}^T_P \;=_{\mathrm{def}}\; [\![(\lambda z\, .\, (\mathit{MAP}\,([\alpha_1, \ldots, \alpha_p]\Sigma^T)\; m\; k)\; \Delta\; P(z))]\!]}$$

As an example, this scheme can be applied to the *List* type constructor, specializing the recursion scheme for map^{List} to yield the function that filters a list with a test predicate. The recursion scheme for the *List* map is generated from an application of *MAP* to the *List* signature,

$$\begin{array}{rcl} \mathit{MAP}\,([\alpha]\Sigma^{List})\; m\; 1 & = & \text{let val rec } m \;=\; \\ & & \lambda f\, .\; \lambda xs\, .\quad \text{case } xs \text{ is} \\ & & \qquad\qquad \mathit{Nil} \;\Rightarrow\; \mathit{Nil} \\ & & \qquad\quad \mid\;\; \mathit{Cons}(x, xs') \;\Rightarrow\; \mathit{Cons}(f\,x,\; m\; f\; xs') \\ & & \text{in } m \end{array}$$

The text generated by elaboration of $\mathit{MAP}\,([\alpha]\Sigma^{List})\; m\; 1$ provides a programmer with a basis on which to formulate a rewrite rule to effect a modification. A function to filter a list can be defined in terms of the specialization scheme for *modified_map*,

$$\begin{array}{rcl} \mathit{filter}^{List} & = & \mathit{modified\_map}^{List}_{P()} \\ & & \text{where } P() \;=\; \mathit{Cons}(f\,x,\; m\; f\; xs) \\ & & \qquad\qquad\quad \leadsto\; \text{if } f\,x \;\; \text{then } \mathit{Cons}(x,\; m\; f\; xs) \\ & & \qquad\qquad\qquad\qquad\quad\;\; \text{else } m\; f\; xs \end{array}$$

in which the wavy arrow specifies rewriting and the parameter $f \;:\; \alpha \rightarrow \mathit{Bool}$ is a test predicate. In this particular example, the modification is to the formation of a term in the *Cons* case, and does not require an additional policy function.

It is worthwhile noting that although the mechanism of specializing a generic recursion scheme with a syntactic difference operation provides an intuitive way to specify type-dependent functions, it is not the only way. For instance, the function used for our example could also be expressed directly in terms of the primitives of the monad of lists as

$$\mathit{filter}^{List}\; q \;=\; \mathit{mult}^{List} \circ \mathit{map}^{List}\,(\lambda x\, .\; \text{if } q\, x \text{ then } \mathit{unit}^{List}\, x \text{ else } \mathit{Nil})$$

although this form will be unfamiliar to almost all programmers. It is worth pointing out, however, that this definition is generic. If the zero-constructor were named uniformly for all types that have one, so that *Nil* in the above formula was replaced by zero^{List}, then the formula would define a generic function filter^T for any unitary type constructor T that has a unique zero constructor, simply by replacing the superscript *List* by T.

We have illustrated a mechanism for pattern-activated, constrained program editing. This mechanism offers several advantages over unconstrained text editing as a means for formulating recursive programs.

- It allows extensive reuse of schematic meta-functions for program generation, such as *MAP, PR, RED.*
- It affords a strict separation between control mechanisms, which include the intrinsic, polymorphic recursion schemes of inductively defined data types, and the type-specific policies that customize program schemes to applications. During program maintenance, intrinsic control schemes are seldom changed, but policy functions are changed frequently.
- Although the Δ-application of patterns modifies the intensional representation of a generic function, it can enforce syntactic constraints sufficient to guarantee preservation of important general properties under the standard semantic interpretation of programming languages. These properties include typing in an ML type system and relative termination.

4 Example: The Lambda Calculator

This section illustrates the synthesis methods presented in the previous sections by using them to develop algorithms to implement a λ-calculus interpreter. In this development the program calculation steps are done by hand in Standard ML. Later, in Section 5, the same development will be used to illustrate the automatic synthesis of programs.

4.1 The problem

The terms of the pure λ-calculus are either variables, applications or abstractions. They are described by the grammar:

$$\begin{array}{lcll} M & ::= & x & \text{variable} \\ & | & MN & \text{application} \\ & | & \lambda x \,.\, M & \text{abstraction} \end{array}$$

Intuitively, the abstraction $\lambda x \,.\, M$ represents that function of x that returns M. Function application is computed via substitution with the rule:

$$(\beta) \qquad (\lambda x \,.\, M)N \rhd M[N/x]$$

Where $\rhd$ is a binary relation (reduction) and $M[N/x]$ denotes the substitution of N for x in M. The relation $\rhd$ is extended to allow any subterm to be replaced by use of the β rule.

Two λ-terms are *congruent* if they differ only in the names of bound variables. For example, $\lambda x \,.\, x$ and $\lambda y \,.\, y$ are congruent. This is summarized by the rule

(α) $\quad (\lambda x \,.\, M) \cong \lambda y \,.\, M[y/x]$ provided y has no free occurrences in M

The traditional treatments of the λ-calculus used these syntactic rules and a subtle definition of substitution to calculate in the formal calculus[CF58, HS86]. These methods can be implemented directly, but are complex and must do computations to avoid variable name clashes. Even the simple test for congruence of two terms requires use of a binding environment (or symbol table).

In the AUTOMATH project, de Bruijn developed a new way to represent λ-terms using indexes rather than variables[Bru72, Bru78, Bru80]. The bound variable names are eliminated from abstractions; each variable occurrence is replaced by the number of λ's between the occurrence and the λ that binds it.[5] For example, $\lambda x \,.\, \lambda y \,.\, xy$ is represented $\lambda \,.\, \lambda \,.\, 1\ 0$. In this representation congruence is simply identity. The complexity of substitution is also reduced since it is no longer necessary to compare names of bound variables or compute new names. Implementations of de Bruijn's scheme are subtle since they require the delicate adjustment of indexes.

This section presents algorithms for computing the de Bruijn representation from a more traditional representation and for substitution using the de Bruijn representation. This development illustrates the use of monads in program development.

4.2 Maybe types

The monadic type constructor `Maybe` defined in Section 2.4 was introduced originally by Spivey to simulate exceptions in a pure functional language[Spi90]. Under the translation, a function that either returns an `int` or raises an exception is converted to a function that always returns an `int Maybe` (note that in Standard ML type constructor application is written in postfix). If the function would have raised the exception then it returns the value `Nothing`. If it would have returned an integer, say `17`, it returns `Just 17`.

When using a monad, T, to develop algorithms it is often natural to lift functions from $\alpha \to T(\beta)$ to $T(\alpha) \to T(\beta)$ so that they can be composed. This is called the natural extension (or "Kleisli star") of a function; it can be defined in terms of map^T and the multiplier. In SML this is written:

```
fun extension f = mult o (map f);
```

The blackboard syntax for extension is a superscripted $*$, e.g. f^*. An alternative characterization of monads may be given directly in terms of the natural extension.

Although it is not strictly necessary, we will include the natural extension in the monads generated in this section. An implementation of `Maybe` types is given as an SML structure definition in Figure 1. Note that functions in this structure can be mechanically calculated from the data type declaration.

4.3 The Term datatype

The problem is to produce an implementation of terms, and ultimately terms with de Bruijn indexes. The first thing we need is a *parametric* type constructor—monads over a constant

[5]The de Bruijn index may be thought of as an "environment pointer" index in a stack based interpreter.

```
structure Maybe =
    struct
        datatype 'a Maybe = Nothing | Just of 'a;
        val unit = Just
        fun map f Nothing = Nothing
          | map f (Just a) = Just(f a);
        fun mult Nothing = Nothing
          | mult (Just a) = a;
        fun extension f = mult o (map f);
    end;
```

Figure 1: An SML implementation of the Monad operations for Spivey's **Maybe** type.

---

type are not of obvious utility. Since variables seem to be "where the action is" we use the following as our first approximation:

```
datatype 'a Term_0 = Var of 'a
                   | App of 'a Term_0 * 'a Term_0
                   | Abs of 'a * 'a Term_0
```

This structure is sufficiently general to include both the surface syntax (**string Term_0**) and the de Bruijn representation (int Term_0). Since Term_0 is a sum-of-products type it is straightforward to calculate map and reduce combinators. The type fails to be unitary, however, since both Var and Abs are unit constructors. Thus this formulation of Term_0 does not support the monadic operators.

To recover the monadic structure, we separate the types of free and bound variables, yielding the **('a,'b)Term** type introduced in Section 2:

```
datatype ('a,'b)Term = Var of 'a
                     | App of ('a,'b)Term * ('a,'b)Term
                     | Abs of 'b * ('a,'b)Term
```

This supports an even more faithful de Bruijn encoding, **(int,triv)Term**[6], as well as the surface syntax **(string,string)Term**.

In the program development the names **map_free** and **map_bound** are used for the combinators map_1^{Term} and map_2^{Term} defined in Section 2.3.1.

[6]Because we are using **unit** to name an operation of the monad we have used **triv** as the name of the one element type in ML. This differs from the official definition where this type is called **unit**. This is done strictly for readability; Standard ML would not be confused because type identifiers and value identifiers come from distinct name spaces.

```
fun map_free f (Var a) = Var (f a)
  | map_free f (App(a,b)) = App(map_free f a,map_free f b)
  | map_free f (Abs(x,a)) = Abs(x,map_free f a);

fun map_bound f (Var a) = Var a
  | map_bound f (App(a,b)) = App(map_bound f a,map_bound f b)
  | map_bound f (Abs(x,a)) = Abs(f x,map_bound f a);

val unit = Var;

fun mult (Var t) = t
  | mult (App(a,b)) = App(mult a, mult b)
  | mult (Abs(x,a)) = Abs(x,mult a);

fun extension f = mult o (map f);
```

Figure 2: Map and monad functions for `('a,'b)Term`.

The map and natural extension functions are given in Figure 2. Note that the argument type of the natural extension, `'a -> ('a,'b)Term`, is suggestive of a substitution function, i.e. it associates terms with variables. We will ultimately use a version of the natural extension to compute the action of a substitution on a term.

4.4 Policy functions

Even though `('a,'b)Term` is a monad as a function of `'a`, the straightforward monad operations are not helpful in operating on λ-terms. The problem is that they do not reflect the critical role of variable binding in terms. To remedy this, we make a small change to the **`map_free`** function to specialize it to the domain. We introduce a policy function, `Z`, that transforms the function being mapped, based on the value of the function and the name of the bound variable. This transformation is applied to the function being mapped every time map is invoked on the body of an abstraction. That is, the function, `f`, is transformed by `Z` whenever it is to be applied in the scope of a new bound variable.

The new function, `map_with_policy`, is defined below. Note that this function is essentially the original `map_free` function augmented with an additional functional parameter, `Z`, that is applied to `f` and `x` in the application case.

```
fun map_with_policy Z f (Var a) = Var (f a)
  | map_with_policy Z f (App(a,b)) = App(map_with_policy Z f a,
                                         map_with_policy Z f b)
  | map_with_policy Z f (Abs(x,a)) = Abs(x,map_with_policy Z (Z f x) a);
```

Given a policy function, Z, of type:

```
Z : ('a -> 'b) -> 'c -> 'a -> 'b
```

the function `map_with_policy Z` has the same type as `map_free`, i.e.

```
('a -> 'b) -> ('a,'c)Term ->  ('b,'c)Term
```

The derivation of `map_with_policy` from `map_free` can be expressed with the Δ operation discussed in Section 3. The rewrite rule, P, is:

$$P(\mathtt{Z}) = \mathtt{Abs(x, m\,f\,a)} \rightsquigarrow \mathtt{Abs(x, m\,(Z\,f\,x)\,a)}$$

The function definition would then be:

$$\mathtt{map\_with\_policy\,Z} = [\![(\mathrm{MAP}\,([\alpha,\beta]\Sigma^{\mathrm{Term}})\ \text{“}m\text{”}\,1)\,\Delta\,P(\mathtt{Z})]\!]$$

The specialized map function may be transformed into a specialized natural extension function, `extension_with_policy`, by imitating the original definition of `extension`. In SML we can define this by:

```
fun extension_with_policy Z f = mult o (map_with_policy Z f);
```

It seems clear that for arbitrary `Z` the monad axioms will not hold for this operation. This does not appear to be necessary since we do not exploit the monad *laws* explicitly in the following development.

4.5 Conversion to de Bruijn representation

The first illustration of the use of these techniques is the construction of a function that yields the de Bruijn representation of a term given in surface syntax. The algorithm walks the argument term using `map_with_policy`. During the traversal it constructs a function associating variable names with indexes.

The initial association must be everywhere undefined. To represent this we use the `Maybe` monad of Figure 1. The initial association is:

```
fn _ => Nothing : string -> int Maybe
```

When the map enters an abstraction, say `Abs("x",a)`, with an existing association function, f, we want to extend the association by defining it to be 0 on `"x"`. Since there is now one more abstraction separating the current term from the definitions of all previously defined variables, the association should yield $1 + f(y)$ for all identifiers distinct from `"x"`. This transformation is expressed in the function `extend_association` in the code fragment below.

```
fun extend_association f x
    = fn y => if x = y then 0 else 1+(f y);

val extend_association : (string -> int) -> string -> (string -> int)
```

However, to be consistent with the type of the initial association function we must lift the result to `int Maybe` instead of `int`. For the constant 0 this is trivial, we just use the `unit` for the `Maybe` type. Otherwise we use the `map` from the `Maybe` type[7] to lift the successor function, making it applicable to `(f y)`.

```
fun extend_association f x
    = fn y => if x = y then unit 0 else map (fn x => 1+x) (f y);

val extend_association : (string -> int Maybe) -> string -> (string -> int Maybe)
```

This policy function, together with the initial association function, is used with `map_with_policy` to give the core of the conversion:

```
val var_to_index = map_with_policy extend_association (fn _ => Nothing)
```

This fragment has the type `(string, string)Term -> (int Maybe, string)Term`.[8] This has two problems: (1) bound variables are still strings, and (2) we have `int Maybe` instead of `int`.

The first problem is easily solved by composing `var_to_index` on the left with `map_bound (fn _ => ())` to remove all traces of variable names. The second problem requires a special function to "distribute" the type constructors, i.e. to convert `(int Maybe, triv)Term` to `(int, triv)Term Maybe`. Intuitively this transformation yields `Nothing` if there are any `Nothing`'s in the argument and otherwise yields `Just t` for a term `t: (int, triv)Term`. The function witnessing this distributive property is called `distribute_Maybe`; it is calculated using the techniques of Section 2.4. This construction is applicable because `Term` is a sum-of-products type and `Maybe` is a linearly-constructed monad possessing a product distribution function. The code for `distribute_Maybe` is given in Figure 3.

The final code for the conversion to de Bruijn form is:

```
val de_Bruijn = distribute_Maybe
                o (map_bound (fn _ => ()))
                o var_to_index

val de_Bruijn : (string,string)Term -> (int,triv)Term Maybe
```

Note that the details of the data structure are only referred to in the generic functions for manipulating the monad. They do not appear in the code specific to computing the de Bruijn indexes.

[7] In the actual code this dependence on `Maybe` types is made explicit by referencing the `Maybe` structure.

[8] Actually the type is not this specific. The three occurrences of `string` are replaced by `''a`, denoting an arbitrary equality type.

```
functor Enrich_Maybe (M:Maybe) =
    struct
        open M;
        val tau_1 = map (fn x=>x);
        fun tau_2(x1,x2)
            = extension (fn a1 => map (fn a2 => (a1,a2)) x2) x1;
    end;

structure E_Maybe = Enrich_Maybe (Maybe);

val distribute_Maybe
    = let fun var_f x = E_Maybe.map Var (E_Maybe.tau_1 x)
          fun abs_f (y, M)
              = E_Maybe.map Abs (E_Maybe.tau_2 (E_Maybe.unit y,M))
          fun app_f (M, N)
              = E_Maybe.map App (E_Maybe.tau_2 (M,N))
      in reduce (var_f, app_f, abs_f)
      end
```

Figure 3: Standard ML code for distribute Maybe.

4.6 Substitution

A *substitution* is a function from variables to terms. The *application* of a substitution to a term (or the *action* of a substitution on a term) is the natural extension of a substitution. That is, it is the function from terms to terms obtained by applying the substitution to the free variables occurring within a term while otherwise preserving its structure.

Since contraction is our motivating example, we illustrate the substitution application operation (`apply_substitution`) with the substitution used in β-contraction. The definition of substitution application developed below is parametric in the substitution applied.

To implement β-contraction it is necessary to replace a variable of index 0 by a term in any context. For example, consider:

$$\lambda y\,.\,(\lambda z\,.\,zy(\lambda v\,.\,z))(\lambda u\,.\,y)$$

which is represented:

$$\lambda\,.\,(\lambda\,.\,0\,1\,(\lambda\,.\,1))(\lambda\,.\,1)$$

The subexpression $(\lambda z . zy(\lambda v . z))(\lambda u . y)$ is a redex, it contracts to $(\lambda u . y)y(\lambda v . \lambda u . y)$, which is represented $(\lambda\,.\,1)0(\lambda\,.\,\lambda\,.\,2)$. Note that the two occurrences of z are represented by both a 0 and a 1 before the contraction, the y in $\lambda u\,.\,y$ becomes both a 1 and a 2 in the representation of the contracted term, and the index of the other occurrence of y decreases from 1 to 0. This decrement of the index of y is required because the lambda binding z is no longer present in the term.

Since the initial goal is to replace 0 by N and decrement all indexes of free variables we define the initial substitution as follows:

```
val sigma_0 = fn x => if x = 0 then N else unit x-1;

val sigma_0 : int -> (int, triv)Term
```

Because substitution application interacts with abstractions, the extension of the substitution must be modified appropriately. This requires `extension_with_policy`. The most general type of `extension_with_policy` is given in Section 4.4. Here we specialize its type to de Bruijn substitutions:

```
val extension_with_policy :
      ((int -> (int,triv)Term) -> triv -> (int -> (int,triv)Term))
      -> (int -> (int,triv)Term) -> (int,triv)Term -> (int,triv)Term
```

The key to the development is the construction of the first argument to `extension_with_policy`, the policy function `transform_substitution`. For the second argument we use the initial substitution `sigma_0`. This gives the initial outline:

```
extension_with_policy transform_substitution sigma_0
: (int,triv)Term -> (int,triv)Term
```

4.6.1 Transforming the substitution

The policy function `transform_substitution` must have the type:

```
transform_substitution : (int -> (int,triv)Term) -> triv -> (int -> (int,triv)Term)
```

It will generate a series of substitutions, $\sigma_0, \sigma_1, \ldots$, which will be applied in the contexts corresponding to the nested abstractions in the term. The basic properties of this series are:

$$\begin{aligned} \sigma_{i+1} 0 &= \mathtt{Var}\, 0 \\ \sigma_{i+1}(n+1) &\approx \sigma_i\, n \end{aligned}$$

That is, the variable bound in the current context is not involved in the substitution; the variables with non-zero index should be treated as their predecessors were in the surrounding context.

Note, however, that the correspondence indicated by $\approx$ is not exact. Since the term being substituted is now in the context of an additional abstraction, all indexes representing bindings outside the term need to be incremented. This suggests[9]:

$$\sigma_{i+1}(n+1) = \mathtt{map\_free}\ succ(\sigma_i(n))$$

But this use of `map_free` would increment variables bound within $\sigma_i(n)$ as well as the global occurrences, so a policy specific version of map must be used instead. In this context the specific type of `map_with_policy` is:

```
map_with_policy : ((int -> int) -> triv -> (int -> int))
                  -> (int -> int)
                  -> (int,triv)Term -> (int,triv)Term
```

We call the policy function that specializes this map `transform_index`. The successor function is the initial function to be mapped. Assuming this transformation, the final form of the constraint on σ_{n+1} is:

$$\sigma_{i+1}(n+1) = \mathtt{map\_with\_policy}\,\mathtt{transform\_index}\ succ(\sigma_i(n))$$

This sequence is generated by the function:

$$\lambda\sigma\,.\,\lambda n\,.\,\mathtt{map\_with\_policy}\,\mathtt{transform\_index}\ succ(\sigma_i(n))$$

To adapt this function to the type scheme required by `extension_with_policy` a dummy parameter of type `triv` is added, yielding the SML definition of `transform_substitution`:

```
fun transform_substitution sigma (():triv)
    = fn n => if n = 0 then unit n
              else map_with_policy transform_index (fn n => n+1) (sigma (n-1));

fun transform_substitution : (int -> (int,triv)Term) -> triv
                             -> int -> (int,triv)Term
```

[9]The function *succ* represents the successor function, `fn x => x+1` in SML.

```
fun apply_substitution sigma_0 M
    = let fun succ x = x+1
            fun transform_index f (():triv)
                = fn n => if n = 0 then n else 1+f(n-1)
            fun transform_substitution sigma (():triv)
                = fn n => if n = 0 then unit 0
                                    else map_with_policy transform_index succ (sigma (n-1))
        in extension_with_policy transform_substitution sigma_0 M
        end;
```

Figure 4: SML code for substitution function using monadic operators. Note that there is no reference to the specific constructors of the Term datatype.

4.6.2 Transforming indexes

Like the substitution operation, the index adjustment is achieved by a family of functions, $f_0, f_1, \ldots$. They satisfy:

$$\begin{aligned} f_0 n &= n+1 \\ f_{(i+1)} n &= \begin{cases} n & \text{if } n \leq i \\ n+1 & \text{otherwise} \end{cases} \end{aligned}$$

This is equivalent to the recursive pattern:

$$\begin{aligned} f_0 n &= n+1 \\ f_{i+1} 0 &= 0 \\ f_{i+1}(n+1) &= 1 + f_i(n) \end{aligned}$$

This recurrence suggests the SML definition of `transform_index` below. The "dummy" argument of type `triv` is dictated by the type scheme.

```
fun transform_index f (():triv)
    = fn n => if n = 0 then 0 else 1 + f(n-1);

val transform_index : (int -> int) -> triv -> int -> int
```

These pieces are combined in the definition of `apply_substitution` in Figure 4. The substitution application function is expressed without any explicit mention of the constructor functions for the `Term` datatype.

4.7 Contraction

We complete the derivation of the lambda calculator by defining a function, `contract`, which reduces the leftmost outermost redex in a term, i.e. it performs *exactly one step* of a normal order reduction. Many other reductions can be specified. Since `contract` explicitly identifies reducible expressions (*redexes*) it will make explicit reference to the datatype constructors.

The first design issue we address is what control combinator we need. Consider the leftmost outermost contraction of MN. Suppose that both M and N contain redexes and that M' and N' are their respective reduced terms. To build the reduced term for MN we use M' but ignore N'. (This is because we specified that we should do exactly one step and M is to the left of N.) The only recursion combinator that gives access to both the original N and the value of the recursive function on M (M') is the primitive recursion operator, pr^{Term}, defined in Section 2.2.2; here it is written `primitive_recursion`.

In the `Term` type, to define a function of type `('a,'c)Term -> 'b` by primitive recursion we need functions of the following types:

```
var_f  :  'a -> 'b
app_f  :  (('a,'c)Term * 'b) * (('a,'c)Term * 'b) -> 'b
abs_f  :  'c * (('a,'c)Term * 'b) -> 'b
```

In our specific task of defining reduction, the first cut assigns `int` to `'a`, `(int,triv)Term` to `'b`, and `triv` to `'c`. However, as soon as we consider `var_f` we are forced to ask "What do we return for normal forms?" Here again it is natural to use the `Maybe` type, interpreting `Nothing` as "the argument is normal" and `Just t` as "the argument reduces in one step to `t`." This requires `'b` to be an `(int,triv)Term Maybe`. Since variables are always irreducible, `var_f` is simply the constant function returning `Nothing`.

The complete definition of `contract` is given in Figure 5.

4.8 Reuse

In the development of the substitution and conversion functions we did not use any details of the `Term` datatype. This allows extensive code reuse whenever the datatype is extended, so long as the monadic properties are not changed[Wad92]. To illustrate this, consider extending the λ-calculus terms with boolean constants and the conditional. The following productions would be added to the syntax:

$$\begin{array}{lcll} M & ::= & \mathit{true} & \text{Boolean constant} \\ & | & \mathit{false} & \\ & | & \mathit{if}\, M \;\mathit{then}\; N_0 \;\mathit{else}\; N_1 & \text{conditional} \end{array}$$

This is reflected in the `Term` datatype by the declarations:

```
datatype ('a,'b)Term =  ...
                      | True
                      | False
                      | Cond of ('a,'b)Term * ('a,'b)Term
                                    * ('a,'b)Term
```

```
fun contract t
    = let fun var_contract _ = Nothing
          fun app_contract ((Abs(x,M),_),(N,_))      (* explicit redex *)
              = let fun sigma_0 n = if n = 0 then N   (* 0 |-> N *)
                                    else unit (n-1)   (* decrement globals *)
                in
                    Just (apply_substitution sigma_0 M)
                end
            | app_contract ((_,Just a),(b,_))        (* function term reduced *)
              = Just (App(a,b))
            | app_contract ((a,Nothing),(_,Just b))  (* only arg reduced *)
              = Just (App(a,b))
            | app_contract ((_,Nothing),(_,Nothing)) (* Normal subterm *)
              = Nothing
          fun abs_contract (x,(_,a))
              = Maybe.extension (fn a => Just(Abs(x,a))) a
      in primitive_recursion (var_contract, app_contract, abs_contract) t
      end;
```

Figure 5: Contraction function for beta reduction

Using the techniques of Section 2 we can verify that this is still a sum-of-products type and a monad in `'a`. New reduction functions, map functions, multipliers and units can be automatically generated. Applying the method of Section 3.1, the same Δ-modification may be applied to the new `map_free` function to yield `map_with_policy` and `extension_with_policy` functions for the new *Term* type. These automatically generated functions encapsulate all of the information about the type needed by the substitution and conversion functions. Thus, the substitution and conversion functions can be reused without modification.

The contraction function, however, cannot be reused directly because it is dependent on the primitive recursion combinator, the type of which reflects the details of the datatype. Since the contraction function is the only place where the semantics of terms is expressed, it is natural that it would require change.

Adding *let* to the language illustrates another interesting point. As for the Boolean constants and conditional, the relevant properties of the *Term* type are preserved and all combinators can be automatically generated. The rewrite rule used to define `map_with_policy`, however, cannot be reused because *let* is a binding operator. Just as it is necessary to modify the code expressing the semantics, it is necessary to specify nontrivial binding structure.

In both of these examples, monadic programming has distilled the non-automatable programming tasks to specifying just that information relevant to the abstraction that is captured by the program. Wadler presents many more examples illustrating this point. His λ-calculus implementation is significantly different from ours because it is based on an environment model rather than rewriting. The approach taken here was selected because it illustrates the use of policy functions, which are not required in the environment model.

5 Program generation using reflection

We have argued that from the structure of datatypes we can infer equational properties of polymorphic functions that provide the control schemes for programs over these types. In the last section, a program was derived by following this principle. In the present section, we shall demonstrate how *compile-time reflection* can be used to build program generation capability into a language such as SML, thereby enabling automatic generation of many of the functions needed for an application.

Reflection is the "magic" that turns data into programs. Compile-time reflection allows user written functions to access data calculated during compilation to construct program representations. These representations are then transformed, by reflection, into the programs they represent. Essentially, compile-time reflection allows data calculated by compile-time evaluation to be type-checked and submitted to the compiler itself, to be turned into object code and integrated with the rest of the compiler's output.

To use the generation paradigm, a program consists of a series of top-level declarations some of which may contain *reflection* directives. Reflection directives recursively apply the compiler to data (in the form of abstract syntax) produced by compile-time evaluation.

For example, the monad operations for a *Maybe* type could be incorporated in an application program by including the declaration for the *Maybe* type constructor and providing reflection directives to generate functions realizing these operations, as shown below.

```
Maybe(x)  =  Nothing | Just(x);

val reduce_maybe = Reflect(e) => (REDUCE (Sigma e "Maybe"));
val unit_maybe = Reflect(e) => (UNIT (Sigma e "Maybe")  0);
val map_maybe = Reflect(e) => (MAP (Sigma e "Maybe") 0);
val mult_maybe = Reflect(e) => (MULT (Sigma e "Maybe") 0);
```

The reflect directive is one of several interfaces between compile-time functions and the compiler that we have experimented with in the language TRPL (Typed, Reflective Programming Language)[She90]. It has the form `Reflect(`*e*`) =>` *exp*, where *e* is the name of a variable, which is bound by the compiler to the current compiler environment at the time the reflect directive is executed. The scope of this binding is the body, *exp*, which can be any expression with type abstract syntax. The variable bound by reflect has type **`environment`** which is an abstract data type. Using this environment, compile time operations may access type information computable by the type checker, or stored in the compiler's symbol table.

In the example above each directive expands, as a predefined macro, into the appropriate function. The function **`Sigma`** (which corresponds to Σ^T in Section 3) extracts from the environment, `e`, a representation of the type definition of the *Maybe* type.

Normal SML declaration processing goes through three stages: *parsing*, *elaboration*, and *evaluation* [MTH90]. Parsing constructs abstract syntax from the textual input. Elaboration performs type checking and performs symbol table updates. Evaluation obtains the value of the construct and updates the store.

The elaboration of a reflection directive is special, and involves three steps. First the body of the reflection is elaborated in a new environment where the bound variable has type **`environment`**. The system expects the body to have the type of abstract syntax. The second step involves the evaluation of a successfully elaborated body. This is done in a new environment where the bound variable is bound to the *current compiler environment*. This evaluation produces new abstract syntax. The third and final step is the elaboration of the new abstract syntax, which (like a macro) replaces the reflection directive.

In this section we describe the details of this process. There are three necessary ingredients for supporting compile-time reflection in a language.

- *Self representation.* There must be a standard representation *in the language* for each facet of the language. We do this by making public the predefined types that comprise the abstract syntax trees of all program elements. Users can then build representations of program facets like types, expressions, and declarations just as they can with any other datatype.

- *Reflection.* Reflection is the ability to calculate representations of new functions, type definitions, or other declarations and interpret them as if they were written directly by the programmer. This allows generator-based systems to generate definitions or declarations, supplementing the ones supplied by application programmers, and to use them as part of the application solution.

- *Statically checkable and reifiable types.* A meta-programming system for a typed language must have the ability to statically infer the types of expressions, to extract type information from the compiler's symbol table, and to manipulate this type information as data. This allows type information to guide the generation process[10].

5.1 Self representation in a reflective language

Let us introduce several sum-of-products types used to represent abstract syntax trees for parts of SML programs. These types are a simplification of the types one might actually use in a complete implementation but are sufficiently rich to explain the generation paradigm without introducing unnecessary complication.

A simplified abstract syntax for expressions is:

```
datatype ('a,'b)erep =
     Id of 'a
|    Iconst of int
|    Bconst of bool
|    Sconst of string
|    App of ('a,'b)erep * ('a,'b)erep
|    Tuple of (('a,'b)erep) list
|    Abs of 'b * ('a,'b)erep
|    Letrec of 'b * ('a,'b)erep * ('a,'b)erep
|    Case of ('a,'b)erep * (('a,'b)erep * ('a,'b)erep) list;

type exp_rep = (string,string)erep;
```

For example, a case expression consists of an argument expression and a list of (pattern, action) pairs to match against the argument. As an example of the construction of representations consider the expression (here given in concrete syntax):

```
case x of  Nil => 0 | Cons(a,m) => 1
```

and its representation in abstract syntax:

```
Case( Id("x"),
      [ ( Id("Nil"),Iconst(0) ),
        ( App(Id("Cons"),Tuple([Id("a"),Id("m")])), Iconst(1) ) ] )
```

Types can also be represented by a sum-of-products type. Consider the (simplified) type representation for SML types below.

[10]Lisp-like languages accommodate self representation by list structures and reflection via the eval function [RS84], but unfortunately they do not provide static type information. This severely limits the use of the generation paradigm in Lisp-like languages unless users explicitly supply type information.

```
datatype 'a trep =
     Freevrep of 'a
|    Intrep
|    Boolrep
|    Stringrep
|    Tuplerep  of ('a trep) list
|    Funrep of 'a trep * 'a trep
|    Parametricrep of 'a trep * string
|    Unionrep of (string * ('a trep) list) list;

type type_rep = string trep;
```

Declarations of types and functions also have representations. We give a sum-of-products type for a simplified declaration representation:

```
datatype decl_rep =
     FunDecl of string * string list * exp_rep
   | TypeDecl of string list * string * type_rep;
```

The type declaration given in concrete syntax as:

```
datatype 'a list = Nil | Cons of 'a * 'a list
```

can be represented in the abstract syntax as:

```
TypeDecl(["a"],"list",
         Unionrep([("nil",[]),
                   ("cons",[Freevrep("a"),
                            Parametricrep(Freevrep("a"),"list")])]));
```

Readers will recognize that these representations are merely the compiler's abstract syntax types made public.

5.2 Reflection: using representations to generate code

To illustrate the generation of code we develop a generator that takes a representation of a type declaration (our encoding of a type's signature) as input and produces the representation of a the map function for that type as output. We provide a concrete realization of the algorithms from Section 2.3.1 as compile-time functions.

Recall the template for the map function:

$$map_k\ f\ (C_i(x_1 \ldots x_{n_i})) \;=\; C_i(e_1 \ldots e_{n_i})$$

$$\text{where } e_j \;=\; \begin{cases} map_k\ f\ x_j & \text{if } x_j \text{ has type } T(\alpha_1 \ldots \alpha_f) \\ f\ x_j & \text{if } x_j \text{ has type } \alpha_k \\ x_j & \text{if } x_j \text{ has any other type} \end{cases}$$

The compile-time algorithm must provide a functional equation for each of the constructors in the type declaration. The implementation described below encodes this set of equations as a stylized recursive function definition. For example, the map generator applied to the **list** datatype would generate the representation of the following expression.

```
let val rec list_map = (fn f1 => (fn y2 => case y2 of
        nil => nil
      | cons (x1,x2) => cons (f1 x1,list_map f1 x2)))
in list_map
```

We say that such a representation is a function definition in *generated form.*

As specified in the template, given an expression, **x_k** an argument to the constructor, C_i, we must compute a new expression which is either an application of the mapping function **f_name**, a recursive call to **map_name**, or the expression, **x_k** unchanged, depending upon the type of the **x_k** argument, **x_ktyp**. This can be encoded concretely by the ML function:

```
fun maprule map_name f_name a_ktyp rectyp (x_k,x_ktyp) =
if x_ktyp = a_ktyp
   then App(f_name,x_k)
   else (if x_ktyp = rectyp
             then App(App(map_name,f_name),x_k)
             else x_k);
```

Given a constructor, **constr**, and a list, **nametype_pairs**, of (name,type) pairs, it is possible to construct a concrete representation of a single equation in the map's definition by a pair of **exp_rep**'s. The type component in the i^{th} pair is the type of the i^{th} argument of **constr**, specified in its declaration. The name component of the i^{th} pair is an arbitrary unique name representing the corresponding pattern variable. Such a pair is turned into a clause in a case expression. This is done with the function map_eqn.

```
fun map_eqn(constr,nametype_pairs,mapname,fname,a_ktyp,rectyp) =
    ( constr_call (constr,(map (fn (a,_) => a) nametype_pairs)),
      constr_call(constr, map (maprule mapname fname a_ktyp rectyp)
                                   nametype_pairs) );
```

where the **constr_call** function builds an application of the data constructor to its arguments if the list of arguments is not null, or returns an identifier representing a nullary data constructor if it is.

```
fun constr_call (cname,[]) = cname
|   constr_call (cname,l) = App(cname,Tuple(l));
```

The final step in the process is to construct a generated form expression defining the map for a type, given that types declaration representation as input. This is a concrete realization of the meta-function *MAP* from Section 3.

```
fun MAP (TypeDecl(free,name,Unionrep(l))) mapname pos =
let val mapname_exp = Id(mapname);
    val fname = newname "f";
    val f_exp = Id("f");
    val argname = newname "y";
    val freev = Freevrep (nth(free,pos));
    val rectyp = Parametricrep(tupletype(map Freevrep free),name);
    val eqnfun = (fn (constr,typs) =>
                 let val args = iota 1 (length typs)
                                    (fn n => Id("x" ^ (makestring n)));
                     val pairs = zip args typs;
                     val constr_exp = Id(constr)
                 in map_eqn(constr_exp,pairs,mapname_exp,f_exp,freev,rectyp) end)
in Letrec(mapname,
          Abs(fname,Abs(argname,Case(Id(argname),map eqnfun l))),
          Id(mapname))
end;
```

Given a type declaration, with free variable list, **free**, a name for the new map function, **mapname**, and a parameter pos, indicating the position of the the free variable for which the map is to be constructed, the algorithm proceeds by generating an expression with that name, **mapname_exp**. It then builds type representations for the free variable type, **freev**, and for the recursive type, **rectype**, and builds a function which will be mapped over each pair of constructors and type lists in the type declaration. This function, **eqnfun**, will return a pair of expressions representing the pattern-action pair of a case clause. The function, **eqnfun** builds a list of pairs each consisting of an arbitrary name (a pattern variable) paired with a type from that data constructor's signature, and passes this list to the **map_eqn** function defined earlier.

As described in Section 3 the final step in the the generation paradigm is to interpret the output of the MAP meta-operator in the standard semantics of the language. This is done by the compiler directive *Reflect*, which applies reflection. Thus the declaration:

```
val map_list = Reflect(e) => (MAP (Sigma e "Maybe") "m" 0);
```

is equivalent to the generated form:

```
val map_list = let val rec m =
   (fn f1 => (fn y2 =>
       case y2 of
         nil => nil
       | cons (x1,x2) => cons (f1 x1,m f1 x2)))
in m;
```

5.3 Patterns and representations

Representations are tedious to construct since they require the use of data constructors, whose names only the most devoted user will remember. In addition, the representation of a program is, in general, much larger than the program itself. To alleviate these problems a reflective language should provide pattern based access to the self representation available in the language. The operator, EXP, provides such pattern based access. In TRPL, EXP(x) is an expression of type **exp_rep**, whose value is the representation of x. For example:

```
          EXP(5)  ≡  Iconst(5)
     EXP( (3,x) )  ≡  Tuple([Iconst(3),Id("x")])
 EXP(fn x => g x)  ≡  Abs("x",App(Id("g"),Id("x")))
```

The EXP operator can be used to build all representations built only from ground terms. To provide more expressive power it is desirable to use compile-time variables in syntactic patterns. Such variables can be placed in expression patterns using the tilde (~) as a prefix to indicate a syntax variable.

Variables in patterns can be used to construct new representations from other representations, or to distinguish representations in a pattern matching construct, such as **case**. For example if the variable, x, had as its value the representation of some expression, then EXP(f(~x)) has as its value the the representation of f applied to that representation.

```
     EXP( (3,~x) )  ≡  Tuple([Iconst(3),x])
 EXP(fn x =>(~g) x)  ≡  Abs("x",App(g,Id("x")))
```

Patterns with variables can be used in case statements to specify program transformations in a compact way. For example

```
case e of
  EXP((~x) + 0) => x
| EXP(0 + (~x)) => x
| EXP((~x) * 1) => x
| EXP(1 * (~x)) => x
| other => other;
```

encodes the transformation embodying the identities $x + 0 = x$ and $x * 1 = x$.

To simplify notation we use the abbreviation:

```
PAT( p1 ==> a1, ... , pn ==> an )  =
  ( fn x => case x of EXP(p1) => EXP(a1) | ... | EXP(pn) => EXP(an) | other => x )
```

Thus the transformation embodying the identities $x + 0 = x$ and $x * 1 = x$ is abbreviated as follows:

```
PAT( ~x + 0 ==> x,
     0 + ~x ==> x,
     ~x * 1 ==> x,
     1 * ~x ==> x )
```

The EXP and PAT operators delineate the boundary between the meta-language used to encode program representations, and the object language.[11].

5.4 Policy functions

As outlined in Section 3.1, polymorphic functions can be specialized by applying syntactic difference operators to their representations. The patterns introduced in the previous section make it easy to specify such syntactic difference operators as rewrite rules.

The delta operator takes an expression in generated form and a rewrite rule that tells how to transform it, returning the transformed expression in generated form.

```
fun delta (Letrec(v,e,b)) tf = Letrec(v,e,transform b tf);
```

where (transform b tf) applies the transformation specified in tf to b in a hygienic manner, renaming bound variables where appropriate to avoid variable capture [KFFD86].

The transformation of the map for lists into the filter function for lists can be specified as:

```
val filter = Reflect(e) =>
        (delta (MAP (Sigma e "list") "m" 1)
              PAT(Cons(~f ~x, m ~f ~xs)
                  ==> if ~f ~x then Cons(~x, m ~f ~xs) else m ~f ~xs));
```

in which the operator symbol "==>" separates the two sides of the rewrite rule. The term (MAP (Sigma e "list") "m" 1) will expand into generated form, as described earlier, and the delta operator will transform it to a representation of:

```
let val rec m =
   (fn f1 => (fn y2 =>
       case y2 of
         nil => nil
       | cons (x1,x2) => if f1 x1 then cons(x1, m f1 x2) else m f1 x2 ))
in m;
```

The function *map_with_policy* of Section 4.4 can be generated in a similar manner from the specification:

[11]In TRPL there are representation patterns for all the self representation constructs. In fact pattern based access to self representation is implemented using abstract syntax macros, one of the other interfaces, and is not a builtin feature.

```
val map_with_policy = (fn z => Reflect(e) =>
      (delta (MAP (Sigma e "Term") "m" 1)
             PAT(Abs(~x, m ~f ~y)
                  ==> Abs(~x, m (z ~f ~x) ~y))));
```

The policy function parameter, `z`, which is bound in the definition of **`map_with_policy`**, is a free variable of the `delta` transformation.

6 Conclusions

Recursive definition and higher order functions provide powerful mechanisms for specifying programs but they do not impose much structure on the form of programs that may be expressed. In this paper, we have explored some aspects of monads associated with datatypes as a means for structured program synthesis. The control structure most obviously associated with a datatype is its structural induction, or slightly more generally, its primitive recursion. When these concepts are generalized from the type of natural numbers to arbitrary sum-of-products datatypes, they account for the control in a wide range of algorithms.

While such observations are not new, they suggest another idea. An internal representation of a datatype signature, in the form of abstract syntax, provides the data needed by a generic algorithm that calculates the definition of a recursive function realizing generalized induction (red^T) or primitive recursion for the given datatype, T. Other researchers have recently advocated the concept of "categorical programming" [Hag87, CS92], in which control schemes, represented as combinators, are derived from the structure of free datatypes. This is analogous to "extracting" the computational content of datatypes as they are defined in a second-order logic such as System F [Gir71]. Here we have taken this concept in a slightly different direction. Rather than programming directly in terms of the combinators derived from specific datatypes, we specify families of programs, using a small number of type-parametric meta-functions. These meta-functions are themselves programs that analyze the intensional representations of datatypes (their signatures) to generate concrete, type-specific programs. This mode of specification helps to structure the development of programs. Similar ideas for program development are implicit in the work of Meijer, Fokkinga and Paterson [MFP91], although less emphasis is given to monadic structure.

The recursive control schemes represented in programs generated from datatypes have wider application than the generic functions for which they are derived. Similar control schemes are found in more specialized algorithms that also analyze values of a given datatype, T. This has led us to propose a new technique for program scheme modification by pattern-directed rewriting. It allows well-understood control schemes to be applied to develop specialized algorithms that are not strictly instances of a generic function. This is a new form of program-scheme reuse. To be able to edit program text in such a way that specific semantic properties are preserved is a goal we have long wished to attain. This formalized editing technique deserves additional trials to determine what improvements are needed in the program designer's interface, and tools must be developed to support it. Also remaining to be explored is how to link the incremental change of programs by restricted term rewriting to a programming logic.

We should like to be able to make and verify assertions about programs expressed as generic functions that have been altered by incremental modification.

To implement the program generation that has been described, compile-time reflection is essential. This capability allows text generated by partial evaluation (including meta-function applications) to be compiled into program components. In particular, it allows datatype definitions to be interpreted for their computational content as well as templates for storage structures. Unlike reflection in untyped languages, compile-time reflection in a typed language is type-safe, as the type-checking phase of a compiler is never bypassed.

There are reasons to believe that type-parametric program generation can be accompanied by type-parametric program transformation to produce efficient, automatically-generated software. Parametricity results, such as the promotion theorem for proving equalities of homomorphisms [Mal89], can also be interpreted to yield term rewriting rules (directed equations) that can accomplish program transformations such as fusion and deforestation [Wad88]. This is a topic of further research.

The reader may ask how the structure of monads is essential to the program derivation methodology that has been illustrated here, as it has not been overtly used by others. The answer is that monad structure leads to a calculus of programs in each datatype. The calculus of lists has been extensively used in the systematic derivation of programs [Bir86, Bir88] but has only recently been generalized [MFP91]. The essentials of a program calculus are the monoidal algebras induced by the function composition operator and the (left and right) identity function. When we have a monad for a particular type constructor, T, there is a synthetic composition and identities in which the structure imposed by any constructed type, $T(\alpha)$, is accounted for implicitly, and does not need to be specified explicitly. These synthetic components can be calculated for each new datatype that satisfies the restrictions needed to satisfy the monad laws. Further, Wadler [Wad92] points out that closely related structures share similarly related algorithms. When an algorithm is parameterized on a monad, it can have multiple instances, each obtained by binding a new monad as its parameter (as the `struct` argument of an SML `functor`, for instance).

There is additional program structure to be gotten from the analysis of datatypes. Generalized terminating recursion schemes are analogous to course-of-values induction schemes for particular datatypes. These too, may be amenable to calculation by the application of suitable meta-functions. However, termination cannot be guaranteed by the type safety of such functions (in a decidable type system); instead, there will be proof obligations to be fulfilled to assure termination. This too, is a topic of current research.

Finally, we point out the challenge to determine whether there are more general conditions than the one we have proposed that will allow type-distribution morphisms to be calculated, and thus support the composition of datatype monads.

A Appendix: Proofs of Properties

In this section we prove some properties of the functions we have defined. We reiterate here a set of numbered equations for our templates.

$$reduce\,(f_1 \ldots f_n)\; C_i(x_1 \ldots x_{n_i}) = f_i(e_1 \ldots e_{n_i}) \tag{5}$$

$$\text{where } e_k = \begin{cases} reduce\,(f_1 \ldots f_n)\, x_k & \text{if } x_k \text{ has type } T(a_1 \ldots a_f) \\ x_k & \text{if } x_k \text{ has any other type} \end{cases}$$

$$map\ f\ C_z = Cz \tag{6}$$

$$map\ f\ C_i(x_1 \ldots x_{n_i}) = C_i(e_1 \ldots e_{n_i}) \tag{7}$$

$$\text{where } e_k = \begin{cases} map\ f\ x_k & \text{if } x_k \text{ has type } T(a_1 \ldots a_f) \\ f\ x_k & \text{if } x_k \text{ has type } a_k \\ x_k & \text{if } x_k \text{ has any other type} \end{cases}$$

$$y \oplus_L C_z = y \tag{8}$$

$$y \oplus_L C_i(x_1 \ldots x_{n_i}) = C_i(e_1 \ldots e_{n_i}) \tag{9}$$

$$\text{where } e_k = \begin{cases} y \oplus_L x_k & \text{if } x_k \text{ is the leftmost parameter with type } T(a_1 \ldots a_f) \\ x_k & \text{otherwise} \end{cases}$$

$$C_z \oplus_R y = y \tag{10}$$

$$C_i(x_1 \ldots x_{n_i}) \oplus_R y = C_i(e_1 \ldots e_{n_i}) \tag{11}$$

$$\text{where } e_k = \begin{cases} x_k \oplus_R y & \text{if } x_k \text{ is the rightmost parameter with type } T(a_1 \ldots a_f) \\ x_k & \text{otherwise} \end{cases}$$

$$link\ x_1 \ldots x_i \ldots x_m = x_1 \oplus_L \ldots x_i \ldots \oplus_R x_m \tag{12}$$

where x_i is the unique parameter with type a_k

$$mult\ x = reduce\ C_1 \ldots link_u \ldots C_n\ x \tag{13}$$

$$mult\ C_z = C_z \tag{14}$$

$$mult\ C_u(x_1 \ldots x_i \ldots x_{m_u}) = (mult\ x_1) \oplus_L \ldots x_i \ldots \oplus_R (mult\ x_{m_u}) \tag{15}$$

where x_i is the unique parameter with type a_k

$$mult\ Ci(x_1, x_2 \ldots x_{m_i}) = Ci(e_1 \ldots e_{m_i}) \tag{16}$$

$$\text{where } C_i \neq C_u \text{ and } e_k = \begin{cases} mult\ x_k & \text{if } x_k \text{ has type } T(a_1 \ldots a_f) \\ x_k & \text{otherwise} \end{cases}$$

A.1 Proof that $\oplus_L$ and $\oplus_R$ are associative

To show $w \oplus_R (x \oplus_R y) = (w \oplus_R x) \oplus_R y$ perform a proof by induction on the structure of w. Either w is the zero, C_z, or it is constructed by some other constructor C_i.

- Base case: $w = C_z$
  Prove: $Cz \oplus_R (x \oplus_R y) = (Cz \oplus_R x) \oplus_R y$

  $Cz \oplus_R (x \oplus_R y)$
  $= (x \oplus_R y)$ by (10)
  $= ((C_z \oplus_R x) \oplus_R y)$ by (10)

- Induction step: $w = C_i(z_1 \ \ldots \ z_{m_i})$
  Assume: $z_i^R \oplus_R (x \oplus_R y) = (z_i^R \oplus_R x) \oplus_R y$
  where z_i^R is the rightmost parameter with type $T(a_1 \ \ldots \ a_f)$
  Prove: $C_i(z_1 \ \ldots \ z_{m_i}) \oplus_R (x \oplus_R y) = (C_i(z_1 \ \ldots \ z_{m_i}) \oplus_R x) \oplus_R y$

  $C_i(z_1 \ \ldots \ z_{m_i}) \oplus_R (x \oplus_R y)$
  $= C_i(z_1 \ldots z_i^R \oplus_R (x \oplus_R y) \ldots z_{m_i})$ by (11)
  $= C_i(z_1 \ldots (z_i^R \oplus_R x) \oplus_R y \ldots z_{m_i})$ by hypothesis
  $= C_i(z_1 \ldots (z_i^R \oplus_R x) \ldots z_{m_i}) \oplus_R y$ by (11)
  $= (C_i(z_1 \ \ldots \ z_{m_i}) \oplus_R x) \oplus_R y$ by (11)

In a similar fashion to show $w \oplus_L (x \oplus_L y) = (w \oplus_L x) \oplus_L y$ perform a proof by induction on the structure of y. The proof is similar and is omitted.

A.2 Proof that C_z is left and right identity for $\oplus_L$ and $\oplus_R$

For $\oplus_L$ the zero constructor C_z is a right identity by definition. To show $C_z \oplus_L y = y$ proceed by induction on y.

- Base case: $y = C_z$

  $C_z \oplus_L C_z = C_z$
  $C_z = C_z$ by (8)

- Induction step: $y = C_i(z_1 \ \ldots \ z_{m_i})$
  Assume: $C_z \oplus_L z_i^L = z_i^L$
  where z_i^L is the leftmost parameter of type $T(a_1 \ \ldots \ a_f)$
  Prove: $C_z \oplus_L C_i(z_1 \ \ldots \ z_{m_i}) = C_i(z_1 \ \ldots \ z_{m_i})$

  $C_z \oplus_L C_i(z_1 \ \ldots \ z_{m_i})$
  $= C_i(z_1 \ldots (C_z \oplus_L z_i^L) \ldots z_{m_i})$ by (9)
  $= C_i(z_1 \ldots z_i^L \ldots z_{m_i})$ by hypothesis
  $= C_i(z_1 \ldots z_{m_i})$

The proof that C_z is a left and right identity for $\oplus_R$ is similar and is omitted.

A.3 Proof that the multiplier distributes over zero replacements

Assume the j^{th} argument, z_j, is the unique argument of the unit constructor $C_u(z_1 \ldots z_j \ldots z_{m_u})$ with type a_k, and that their exists at least one argument to the right of z_j with type $T(a_1 \ldots a_f)$, and the rightmost of these is called z_{m_u}, then $mult(w \oplus_R b) = (mult\ w) \oplus_R (mult\ b)$. This is proved by induction over the structure of w. The proof will have two induction steps, the first when w is constructed by the unit constructor, and the second when w is constructed by a non-unit constructor.

- Base Case: $w = Cz$
  Prove: $mult(C_z \oplus_R b) = (mult\ C_z) \oplus_R (mult\ b)$

  $mult(C_z \oplus_R b)$
  $= mult\ b$ by (10)
  $= C_z \oplus_R (mult\ b)$ by (10)
  $= (mult\ C_z) \oplus_R (mult\ b)$ by (14)

- Induction Step **Case 1**: w is constructed by a non-unit constructor, and is written as: $C_i(z_1\ \ldots z_{m_i})$, where no z_j has type a_k. Let z_i^R be the rightmost parameter with type $T(a_1\ \ldots\ a_f)$.
  Assume: $mult\,(z_i^R \oplus_R b) = (mult\ z_i^R) \oplus_R (mult\ b)$
  Prove: $mult\,(C_i(z_1\ \ldots z_{m_i}) \oplus_R b) = (mult\ C_i(z_1\ \ldots z_{m_i})) \oplus_R (mult\ b)$

  $mult(C_i(z_1\ \ldots z_{m_i}) \oplus_R b)$
  $= mult(C_i(z_1\ \ldots (z_i^R \oplus_R b)\ \ldots z_{m_i}))$ by (11)
  $= C_i(e_1\ \ldots (mult\,(z_i^R \oplus_R b))\ \ldots e_{m_i})$ by (16)
  $= C_i(e_1\ \ldots ((mult\ z_i^R) \oplus_R (mult\ b))\ \ldots e_{m_i})$ by hyp
  $= C_i(e_1\ \ldots (mult\ z_i^R)\ \ldots e_{m_i}) \oplus_R (mult\ b)$ by (11)
  $= (mult\ C_i(z_1\ \ldots z_{m_i})) \oplus_R (mult\ b)$ by (16)

- Induction step **Case 2**: w is constructed by the unit constructor C_u, and is written as: $C_u(z_1\ \ldots z_j\ \ldots z_{m_u})$, where z_j has type a_k, and z_{m_u} is the rightmost parameter of type $T(a_1\ \ldots\ a_f)$.
  Assume: $mult(z_{m_u} \oplus_R b) = (mult\ z_{m_u}) \oplus_R (mult\ b)$
  Prove: $mult(C_u(z_1\ \ldots z_j\ \ldots z_{m_i}) \oplus_R b) = (mult\ C_u(z_1\ \ldots z_j\ \ldots z_{m_i})) \oplus_R (mult\ b)$

$$\begin{array}{lr}
mult(C_u(z_1 \ldots, z_j \ldots z_{m_u}) \oplus_R b) & \\
= mult(C_u(z_1 \ldots z_j \ldots (z_{m_u} \oplus_R b)) & \text{by (11)} \\
= (mult\ z_1) \oplus_L \ldots z_j \ldots \oplus_R (mult\ (z_{m_u} \oplus_R b)) & \text{by (15)} \\
= (mult\ z_1) \oplus_L \ldots z_j \ldots \oplus_R ((mult\ z_{m_u}) \oplus_R (mult\ b)) & \text{by hyp} \\
= ((mult\ z_1) \oplus_L \ldots z_j \ldots \oplus_R (mult\ z_{m_u})) \oplus_R (mult\ b) & \text{by assoc } \oplus_R, \oplus_L \\
= (mult\ C_u(z_1 \ldots z_j \ldots z_{m_u})) \oplus_R (mult\ b) & \text{by (15)}
\end{array}$$

The theorem that $mult(w \oplus_L b) = (mult\ w) \oplus_L (mult\ b)$ assumes the existence of at least one parameter of the unit constructor, $C_u(z_1 \ldots z_j \ldots z_{m_i})$, of type $T(a_1 \ldots a_f)$ to the left of z_j. It proceeds by induction over b. It is similar and is omitted.

A.4 Proof that the Multiplier distributes over *link*

To show that $mult(link\ x_1\ \ldots x_m) = (link\ (mult\ x_1)\ \ldots (mult\ x_m))$ perform a case analysis on the structure of the unit constructor C_u.

- If C_u is a perfect unit, then *link* is the identity function, so:

$$\begin{array}{l}
mult(link\ x_1) = link(mult\ x_1) \\
mult(id\ x_1) = id(mult\ x_1) \\
mult\ x_1 = mult\ x_1
\end{array}$$

- If C_u is not a perfect unit, then T must have a zero, C_z, and support zero replacement functions, $\oplus_L$,and $\oplus_R$
  Prove: $mult(link\ z_1\ \ldots z_j\ \ldots z_{m_u}) = link\ (mult\ z_1)\ \ldots (mult\ z_j)\ \ldots (mult\ z_{m_u})$

$$\begin{array}{lr}
mult(link\ z_1\ \ldots z_j\ \ldots z_{m_u}) & \\
= mult(z_1 \oplus_L \ldots z_j \ldots \oplus_R z_{m_u}) & \text{by (12)} \\
= (mult\ z_1) \oplus_L \ldots (mult\ z_j) \ldots \oplus_R (mult\ z_{m_u}) & \text{by mult distributes over } \oplus_R, \oplus_L \\
= link\ (mult\ z_1) \ldots (mult\ z_j)\ \ldots\ (mult\ z_{m_u}) & \text{by (12)}
\end{array}$$

A.5 The monad laws

A monad is characterized by the three laws

$$\begin{array}{rcl}
mult \circ unit & = & id \\
mult \circ (map\ unit) & = & id \\
mult \circ mult & = & mult \circ (map\ mult)
\end{array}$$

We will prove each in turn to demonstrate that the triple (*map*,*unit*,*mult*) on a_k imposes the structure of a monad on any unitary on a_k sum-of-products type.

1. To prove the first monad law, $mult \circ unit \ = \ id$, we will show $mult(unit\ x) = x$.

$$\begin{array}{ll}
mult(unit\ x) = x & \\
= mult(Cu(C_z \ldots x \ldots C_z)) & \text{by definition of } unit \\
= (mult\ C_z) \oplus_L \ldots x \ldots \oplus_R (mult\ C_z) & \text{by (15)} \\
= C_z \oplus_L \ldots x \ldots oplus^R C_z & \text{by (14)} \\
= x \ldots \oplus_R C_z & \text{by } C_z \text{ is identity of } \oplus_L \\
= x & \text{by } C_z \text{ is identity of } \oplus_R
\end{array}$$

2. To prove the second monad law, $mult \circ (map\ unit) \ = \ id$, we will show $mult(map\ unit\ x) \ = \ x$ by cases on the structure of the unit constructor for T.

   (a) T has a perfect unit, C_u, then the *link* function is identity. We will do a proof by induction on structure of x. The base case in this induction will be x constructed by the unit constructor.

   - Base case: $x = C_u(m)$
     Prove: $mult(map\ unit\ C_u(m)) = C_u(m)$

$$\begin{array}{ll}
mult(map\ unit\ C_u(m)) & \\
= mult(C_u(unit\ m)) & \text{by (7)} \\
= (unit\ m) & \text{by (15)} \\
= C_u(m) & \text{by definition of } unit
\end{array}$$

   - Induction step: $x = C_i(z_1 \ldots z_{m_i})$
     Assume: for all z_i with type $T(a_1 \ldots a_f)$, $mult(map\ unit\ z_i) \ = \ z_i$
     Prove: $mult(map\ unit\ C_i(z_1 \ldots z_{m_i})) = C_i(z_1 \ldots z_{m_i})$

$$mult(map\ unit\ C_i(z_1 \ldots z_{m_i}))$$

$$= mult(C_i(e_1 \ldots e_{m_i}))$$

$$\text{where } e_k \ = \ \begin{cases} map\ unit\ z_k & \text{if } z_k \text{ has type } T(a_1 \ldots a_f) \\ z_k & \text{otherwise} \end{cases} \qquad \text{by (7)}$$

$$= C_i(e_1 \ldots e_{m_i})$$

$$\text{where } e_k \ = \ \begin{cases} mult\ (map\ unit\ z_k) & \text{if } z_k \text{ has type } T(a_1 \ldots a_f) \\ z_k & \text{otherwise} \end{cases} \qquad \text{by (16)}$$

$$= C_i(e_1 \ldots e_{m_i})$$

$$\text{where } e_k \ = \ \begin{cases} z_k & \text{if } z_k \text{ has type } T(a_1 \ldots a_f) \\ z_k & \text{otherwise} \end{cases} \qquad \text{by hyp}$$

$$= C_i(z_1 \ldots z_{m_i})$$

   (b) T does not have a perfect unit. Thus T must have a zero, C_z, and support zero replacement functions, $\oplus_L$,and $\oplus_R$. We will prove $mult(map\ unit\ x) \ = \ x$ by induction on x. The base case will be x constructed by the zero C_z. There will be two induction steps, one if x is constructed by the unit constructor, and second if x is constructed by any non-unit, non-zero constructor.

- Base case: $x = C_z$
  Prove: $mult(map\, unit\, Cz) = C_z$

  $mult(map\, unit\, Cz)$
  $= mult\ C_z$ by (6)
  $= C_z$ by (14)

- Induction step **Case 1**: x is constructed by the unit constructor. We will express x as $C_u(z_1 \ldots z_k \ldots z_{m_u})$ where z_k is the unique argument of type a_k, and for all $j \neq k$, z_j has type $T(a_1 \ldots a_f)$.

  Assume: $mult(map\, unit\, z_j) = z_j$
  Prove: $mult(map\ unit\ C_u(z_1 \ldots z_k \ldots z_{m_u})) = C_u(z_1 \ldots z_k \ldots z_{m_u})$

  $mult(map\ unit\ C_u(z_1 \ldots z_k \ldots z_{m_u}))$
  $= mult(C_u((map\ unit\ z_1) \ldots (unit\ z_k) \ldots (map\ unit\ z_{m_u})))$ by (7)
  $= (mult(map\ unit\ z_1)) \oplus_L \ldots (unit\ z_k) \ldots \oplus_R (mult(map\ unit\ z_{m_u}))$ by (15)
  $= z_1 \oplus_L \ldots (unit\ z_k) \ldots \oplus_R z_{m_u}$ by hyp
  $= z_1 \oplus_L \ldots C_u(C_z \ldots z_k \ldots C_z) \ldots \oplus_R z_{m_u}$ by def unit
  $= C_u((z_1 \oplus_L \ldots C_z) \ldots z_k \ldots C_z) \ldots \oplus_R z_{m_u}$ by (9)
  $= C_u((z_1 \oplus_L \ldots C_z) \ldots z_k \ldots (C_z \ldots \oplus_R z_{m_u}))$ by (11)
  $= C_u(z_1 \ldots z_k \ldots (C_z \ldots \oplus_R z_{m_u}))$ by (8)
  $= C_u(z_1 \ldots z_k \ldots z_{m_u})$ by (10)

- Induction step **Case 2**: x is not constructed by the unit constructor thus it can be expressed as: $C_i(z_1 \ldots z_{m_i})$
  Assume: for all z_i with type $T(a_1 \ldots a_f)$, $mult(map\, unit\, t_i) = t_i$
  Prove: $mult(map\, unit\, C_i(z_1 \ldots z_{m_i})) = C_i(z_1 \ldots z_{m_i})$

$$\begin{aligned}
&mult(map\ unit\ C_i(z_1 \ldots z_{m_i})) \\
&= mult(C_i(e_1 \ldots e_{m_i})) \\
&\quad \text{where } e_k = \begin{cases} map\ unit\ z_k & \text{if } z_k \text{ has type } T(a_1 \ldots a_f) \\ z_k & \text{otherwise} \end{cases} && \text{by (7)} \\
&= C_i(e_1 \ldots e_{m_i}) \\
&\quad \text{where } e_k = \begin{cases} mult\ (map\ unit\ z_k) & \text{if } z_k \text{ has type } T(a_1 \ldots a_f) \\ z_k & \text{otherwise} \end{cases} && \text{by (16)} \\
&= C_i(e_1 \ldots e_{m_i}) \\
&\quad \text{where } e_k = \begin{cases} z_k & \text{if } z_k \text{ has type } T(a_1 \ldots a_f) \\ z_k & \text{otherwise} \end{cases} && \text{by hyp} \\
&= C_i(z_1 \ldots z_{m_i})
\end{aligned}$$

3. To prove the third monad law, $mult \circ mult = mult \circ (map\ mult)$, we will prove $mult\ (mult\ x) = mult\ (map\ mult\ x)$ by a case analysis on the structure of the unit constructor for T.

   (a) If T has a perfect unit, C_u, then it has exactly one argument, and the *link* function for T is the identity function. In this case let $x = C_u(a)$.

   Prove: $mult\ (mult\ C_u(a)) = mult\ (map\ mult\ C_u(a))$

   $$\begin{aligned}
   &mult\ (mult\ C_u(a)) \\
   &= mult\ (a) && \text{by (15)} \\
   &= mult\ C_u(mult\ a) && \text{by (15)} \\
   &= mult\ (map\ mult\ C_u(a)) && \text{by (7)}
   \end{aligned}$$

   (b) If T is not a perfect unit, then there exists a zero, C_z, and a zero replacement functions, $\oplus_L$, $\oplus_R$, and we will prove $mult\ (mult\ x) = mult\ (map\ mult\ x)$ by induction on x. We will have a base case and two induction step cases, one for when x is constructed by the unit constructor, and one for when x is constructed by any other non-unit constructor.

   - Base case $x = C_z$

     Prove: $mult\ (mult\ C_z) = mult\ (map\ mult\ C_z)$

     $$\begin{aligned}
     &mult\ (mult\ C_z) \\
     &= mult\ (\ C_z\ ) && \text{by (14)} \\
     &= mult\ (map\ mult\ C_z\ ) && \text{by (6)}
     \end{aligned}$$

- Induction step, **Case 1:** x is not constructed by the unit constructor. let $x = C_i(z1 \ldots z_{m_i})$ Assume: for all z_k with type $T(a_1 \ldots a_f)$, $mult\ (mult\ z_k) = mult\ (map\ mult\ z_k)$
  Prove: $mult\ (mult\ C_i(z1 \ldots z_{m_i})) = mult\ (map\ mult\ C_i(z1 \ldots z_{m_i}))$

$$mult\ (mult\ C_i(z1 \ldots z_{m_i}))$$
$$= mult(C_i(e_1 \ldots e_{m_i})) \quad \text{where } e_k = \begin{cases} mult\ z_k & \text{if } z_k \text{ has type } T(a_1 \ldots a_f) \\ z_k & \text{otherwise} \end{cases} \qquad \text{by (16)}$$

$$= C_i(e_1 \ldots e_{m_i}) \quad \text{where } e_k = \begin{cases} mult\ (mult\ z_k) & \text{if } z_k \text{ has type } T(a_1 \ldots a_f) \\ z_k & \text{otherwise} \end{cases} \qquad \text{by (16)}$$

$$= C_i(e_1 \ldots e_{m_i}) \quad \text{where } e_k = \begin{cases} mult\ (map\ mult\ z_k) & \text{if } z_k \text{ has type } T(a_1 \ldots a_f) \\ z_k & \text{otherwise} \end{cases} \qquad \text{by hyp}$$

$$= mult\ C_i(e_1 \ldots e_{m_i}) \quad \text{where } e_k = \begin{cases} (map\ mult\ z_k) & \text{if } z_k \text{ has type } T(a_1 \ldots a_f) \\ z_k & \text{otherwise} \end{cases} \qquad \text{by (16)}$$

$$= mult\ (map\ mult\ C_i(e_1 \ldots e_{m_i})) \quad \text{where } e_k = \begin{cases} z_k & \text{if } z_k \text{ has type } T(a_1 \ldots a_f) \\ z_k & \text{otherwise} \end{cases} \qquad \text{by (7)}$$

$$= mult\ (map\ mult\ C_i(z_1 \ldots z_{m_i}))$$

- Induction step, **Case 2:** x is constructed by the unit constructor, C_u, let x be expressed as $C_u(z_1 \ldots z_k \ldots z_{m_u})$ where z_k is the unique argument with type a_k, and for all $j \neq k$, z_j has type $T(a_1 \ldots a_f)$.

  Assume: $mult\ (mult\ t_j) = mult\ (map\ mult\ t_j)$
  Prove: $mult\ (mult\ C_u(z_1 \ldots z_k \ldots z_{m_u})) = mult\ (map\ mult\ C_u(z_1 \ldots z_k \ldots z_{m_u}))$

$$\begin{array}{ll}
mult\ (mult\ C_u(z_1 \ldots z_k \ldots z_{m_u})) & \\
mult\ ((mult\ z_1) \oplus_L \ldots z_k \ldots \oplus_R(mult\ z_{m_u})) & \text{by (15)} \\
(mult\ (mult\ z_1)) \oplus_L \ldots (mult\ z_k) \ldots \oplus_R(mult\ (mult\ z_{m_u})) & \text{by distributivity} \\
(mult\ (map\ mult\ z_1)) \oplus_L \ldots (mult\ z_k) \ldots \oplus_R(mult\ (map\ mult\ z_{m_u})) & \text{by hyp} \\
mult\ C_u((map\ mult\ z_1) \oplus_L \ldots (mult\ z_k) \ldots \oplus_R(map\ mult\ z_{m_u})) & \text{by (15)} \\
mult\ C_u((map\ mult\ z_1) \oplus_L \ldots (mult\ z_k) \ldots \oplus_R(map\ mult\ z_{m_u})) & \text{by (15)} \\
mult\ (map\ mult\ C_u(z_1 \oplus_L \ldots z_k \ldots \oplus_R z_{m_u})) & \text{by (7)}
\end{array}$$

B Proof of the composite monad construction

Given monads $(\mathcal{R},\ R,\ \eta^R,\ \mu^R)$ and $(\mathcal{S},\ S,\ \eta^S,\ \mu^S)$[12] with a distribution morphism, π^S_R, with type $\mathcal{S}(\mathcal{R}(\alpha)) \rightarrow \mathcal{R}(\mathcal{S}(\alpha))$, satisfying the following equations

$$
\begin{aligned}
\pi^S_R \circ \eta^S &= R(\eta^S) && (17)\\
\pi^S_R \circ S(\eta^R) &= \eta^R && (18)\\
\pi^S_R \circ \mu^S &= R(\mu^S) \circ \pi^S_R \circ S(\pi^S_R) && (19)\\
\pi^S_R \circ S(\mu^R) &= \mu^R \circ R(\pi^S_R) \circ \pi^S_R && (20)
\end{aligned}
$$

the quadruple $(\mathcal{RS},\ R \circ S,\ \eta^R \circ \eta^S,\ R(\mu^S) \circ \mu^R \circ R(\pi^S_R))$ is a monad. To prove this, the three monad laws:

$$
\begin{aligned}
\mu^T \circ \eta^T &= id\\
\mu^T \circ T(\eta^T) &= id\\
\mu^T \circ \mu^T &= \mu^T \circ T(\mu^t)
\end{aligned}
$$

instantiated with the definitions for the composite maps, units, and multipliers, need to be proven. For reference, in the proofs below, we state a number of facts, all of which are consequences of the naturality of μ, and π^S_R.

$$
\begin{aligned}
R(\eta^S) \circ \eta^R &= \eta^R \circ \eta^S && (21)\\
R(\mu^S) \circ \mu^R &= \mu^R \circ RR(\mu^S) && (22)\\
RS(\pi^S_R) \circ \mu^R &= \mu^R \circ RRS(\pi^S_R) && (23)\\
RS(\mu^S) \circ \mu^R &= \mu^R \circ RR(\mu^S) && (24)\\
R(\pi^S_R) \circ \mu^R &= \mu^R \circ RR(\pi^S_R) && (25)\\
RS(\pi^S_R) \circ \pi^S_R &= \pi^S_R \circ SR(\pi^S_R) && (26)\\
R(\mu^S) \circ \pi^S_R &= \pi^S_R \circ SR(\mu^S) && (27)\\
R(\mu^S) \circ \mu^R &= \mu^R \circ RR(\mu^S) && (28)
\end{aligned}
$$

Because the maps for $\mathcal{R}$ and $\mathcal{S}$ (R,S) are functors, they preserve both identities, and compositions. Thus $R(id) = id$, and $R(f) \circ R(g) = R(f \circ g)$. In the proofs below when one of these laws is used, it is justified by *the functoriality of R*. If we have a law, called N, say $f \circ g = h$, then we justify $R(f) \circ R(g) = R(h)$ by invoking *N under R*.

1. Proof of first monad law for composite monads: $\mu^{RS} \circ \eta^{RS} = id$

$$R(\mu^S) \circ \mu^R \circ R(\pi^S_R) \circ \eta^R \circ \eta^S = id$$

[12]In this section we use R, η^R, and μ^R for map^R, $unit^R$, and $mult^R$.

$$\begin{array}{rll}
& R(\mu^S) \circ \mu^R \circ R(\pi^S_R) \circ \eta^R \circ \eta^S & \\
= & R(\mu^S) \circ \mu^R \circ R(\pi^S_R) \circ R(\eta^S) \circ \eta^R & \text{by (21)} \\
= & R(\mu^S) \circ \mu^R \circ R(R(\eta^S)) \circ \eta^R & \text{by (17) under R} \\
= & \mu^R \circ RR(\mu^S) \circ RR(\eta^S) \circ \eta^R & \text{by (28)} \\
= & \mu^R \circ RR(id) \circ \eta^R & \text{by 1st monad law for S under RR} \\
= & \mu^R \circ id \circ \eta^R & \text{by functoriality of RR} \\
= & \mu^R \circ \eta^R & \\
= & id & \text{by 1st monad law for R}
\end{array}$$

2. Proof of second monad law for composite monads: $\mu^{RS} \circ RS(\eta^{RS}) = id$

$$R(\mu^S) \circ \mu^R \circ R(\pi^S_R) \circ RS(\eta^R \circ \eta^S) = id$$

$$\begin{array}{rll}
& R(\mu^S) \circ \mu^R \circ R(\pi^S_R) \circ RS(\eta^R \circ \eta^S) & \\
= & R(\mu^S) \circ \mu^R \circ R(\pi^S_R) \circ RS(\eta^R) \circ RS(\eta^S) & \text{by functoriality of RS} \\
= & R(\mu^S) \circ \mu^R \circ R(\pi^S_R \circ S(\eta^R)) \circ RS(\eta^S) & \text{by functoriality of R} \\
= & R(\mu^S) \circ \mu^R \circ R(\eta^R) \circ RS(\eta^S) & \text{by (18)} \\
= & R(\mu^S) \circ id \circ RS(\eta^S) & \text{by 2nd monad law for R} \\
= & R(\mu^S) \circ RS(\eta^S) & \\
= & R(\mu^S \circ S(\eta^S)) & \text{by functoriality of R} \\
= & R(id) & \text{by 2nd monad law for S} \\
= & id & \text{by functoriality of R}
\end{array}$$

3. Proof of third monad law for composite monads: $\mu^{RS} \circ \mu^{RS} = RS(\mu^{RS})$

$$\begin{array}{rll}
& \mu^{RS} \circ \mu^{RS} & \\
= & R(\mu^S) \circ \mu^R \circ R(\pi^S_R) \circ R(\mu^S) \circ \mu^R \circ R(\pi^S_R) & \text{by definition} \\
= & R(\mu^S) \circ \mu^R \circ R((R\mu^S) \circ \pi^S_R \circ S(\pi^S_R)) \circ \mu^R \circ R(\pi^S_R) & \text{by (19) under R} \\
= & R(\mu^S) \circ \mu^R \circ (RR\mu^S) \circ R(\pi^S_R) \circ RS(\pi^S_R) \circ \mu^R \circ R(\pi^S_R) & \text{by functoriality of R} \\
= & R(\mu^S) \circ R(\mu^S) \circ \mu^R \circ R(\pi^S_R) \circ RS(\pi^S_R) \circ \mu^R \circ R(\pi^S_R) & \text{by (22)} \\
= & R(\mu^S) \circ RS(\mu^S) \circ \mu^R \circ R(\pi^S_R) \circ RS(\pi^S_R) \circ \mu^R \circ R(\pi^S_R) & \text{by 3rd monad law for S under R} \\
= & R(\mu^S) \circ RS(\mu^S) \circ \mu^R \circ R(\pi^S_R) \circ \mu^R \circ RRS(\pi^S_R) \circ R(\pi^S_R) & \text{by (23)} \\
= & R(\mu^S) \circ RS(\mu^S) \circ \mu^R \circ \mu^R \circ RR(\pi^S_R) \circ RRS(\pi^S_R) \circ R(\pi^S_R) & \text{by (25)} \\
= & R(\mu^S) \circ RS(\mu^S) \circ \mu^R \circ R(\mu^R) \circ RR(\pi^S_R) \circ RRS(\pi^S_R) \circ R(\pi^S_R) & \text{by 3rd monad law for R} \\
= & R(\mu^S) \circ \mu^R \circ RR(\mu^S) \circ R(\mu^R) \circ RR(\pi^S_R) \circ RRS(\pi^S_R) \circ R(\pi^S_R) & \text{by (24)} \\
= & R(\mu^S) \circ \mu^R \circ RR(\mu^S) \circ R(\mu^R) \circ RR(\pi^S_R) \circ R(RS(\pi^S_R) \circ \pi^S_R) & \text{by functoriality of R} \\
= & R(\mu^S) \circ \mu^R \circ RR(\mu^S) \circ R(\mu^R) \circ RR(\pi^S_R) \circ R(\pi^S_R \circ SR(\pi^S_R)) & \text{by (26)} \\
= & R(\mu^S) \circ \mu^R \circ RR(\mu^S) \circ R(\mu^R) \circ RR(\pi^S_R) \circ R(\pi^S_R) \circ RSR(\pi^S_R) & \text{by functoriality of R} \\
= & R(\mu^S) \circ \mu^R \circ RR(\mu^S) \circ R(\pi^S_R) \circ RS(\mu^R) \circ RSR(\pi^S_R) & \text{by (20) under R} \\
= & R(\mu^S) \circ \mu^R \circ R(\pi^S_R) \circ RSR(\mu^S) \circ RS(\mu^R) \circ RSR(\pi^S_R) & \text{by (27) under R} \\
= & R(\mu^S) \circ \mu^R \circ R(\pi^S_R) \circ RS(R(\mu^S) \circ \mu^R \circ R(\pi^S_R)) & \text{by functoriality of RS} \\
= & RS(\mu^{RS}) & \text{by definition}
\end{array}$$

References

[Bir86] Richard S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, volume 36 of *NATO Series F*. Springer-Verlag, 1986.

[Bir88] Richard S. Bird. Lectures on constructive functional programming. In M. Broy, editor, *Constructive Methods in Computing Science*, volume 52 of *NATO Series F*. Springer-Verlag, 1988.

[Bru72] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagaciones Mathematische*, 34:381–392, 1972. Also appeared in the Proceedings of the Koninklijke Nederlandse Akademie van Wetenschappen, Amsterdam, series A, 75(5).

[Bru78] N. G. de Bruijn. Lambda calculus with namefree formulas involving symbols that represent reference transforming mappings. In *Proceedings of the Koninklijke Nederlandse Akaemie van Wetenschappen*, pages 348–356, Amsterdam, series A, volume 81(3), September 1978.

[Bru80] N. G. de Bruijn. A survey of the project AUTOMATH. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism.* Academic Press, New York, 1980.

[BW85] M. Barr and C. Wells. *Toposes, Triples and Theories.* Springer-Verlag, New York, 1985.

[CF58] H. B. Curry and R. Feys. *Combinatory Logic*, volume I. North-Holland, Amsterdam, 1958.

[CS92] J. R. B. Cockett and D. Spencer. Strong categorical datatypes. In R. A. G. Seely, editor, *International Meeting on Category Theory, 1991.* AMS, 1992.

[DKM91] Olivier Danvy, Jürgen Koslowski, and Karoline Malmkjær. Compiling monads. Technical Report CIS-92-3, Kansas State University, Manhattan, Kansas, December 1991.

[Fuk92] Tom Fukushima. Monads in Charity. Unpublished manuscript, May 1992.

[Gir71] J.-Y. Girard. Une extension de l'interprétation fontionnelle de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In J. E. Fenstad, editor, *Second Scandinavian Logic Symposium*, pages 63–92, Amsterdam, 1971. North-Holland.

[Hag87] T. Hagino. *A Categorical Programming Language.* PhD thesis, University of Edinburgh, 1987.

[HS86] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and λ-calculus.* Cambridge University Press, Cambridge, 1986.

[KFFD86] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygenic macros. In *1986 ACM Conference on Lisp and Functional Programming*, pages 151–159, 1986.

[Mai91] Harry G. Mairson. Outline of a proof theory of parametricity. In *Proc. of 5th ACM Conf. on Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 313–327. Springer-Verlag, August 1991.

[Mal89] Grant Malcolm. Homomorphisms and promotability. In J. L. A. van de Snepscheut, editor, *Mathematics of Program Construction*, volume 375 of *Lecture Notes in Computer Science*, pages 335–347. Springer-Verlag, June 1989.

[MFP91] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proc. of 5th ACM Conf. on Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer-Verlag, August 1991.

[Mog91] Eugenio Moggi. Notions of computations and monads. *Information and Computation*, 93(1):55–92, July 1991.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, Cambridge, Massachusetts, 1990.

[RS84] Jim des Rivieres and Brian Cantwell Smith. The implementation of procedurally reflective languages. In *Proceedings of the 1984 Lisp and Functional Programming Conference.* ACM, 1984.

[She90] Timothy Sheard. A user's guide to TRPL: A compile-time reflective programming language. Technical Report 90-109, Computer and Information Sciences, University of Massachusetts, Amherst, 1990.

[Spi90] Mike Spivey. A functional theory of exceptions. *Science of Computer Programming*, 14:25–42, 1990.

[Wad88] Philip Wadler. Deforestation: Transforming programs to eliminate trees. In *ESOP'88*, volume 300 of *Lecture Notes in Computer Science*, pages 344–358. Springer-Verlag, March 1988.

[Wad89] Philip Wadler. Theorems for free! In *Proc. of 4th ACM Conf. on Functional Programming Languages and Computer Architecture*, pages 347–359. ACM Press, September 1989.

[Wad90] Philip Wadler. Comprehending monads. In *Proc. 1990 ACM Conference on Lisp and Functional Programming*, pages 61–78, 1990.

[Wad92] Philip Wadler. The essence of functional programming. In *Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages.* ACM Press, January 1992.