

# Programming with Algebras<sup>\*</sup>

Richard B. Kieburtz and Jeffrey Lewis

Pacific Software Research Center  
Oregon Graduate Institute  
of Science & Technology  
P.O. Box 91000  
Portland, OR 97291-1000 USA  
<http://www.cse.ogi.edu/PacSoft/>

## 1 Introduction

From the early days of computing, many individuals have recognized that algebras provide interesting mathematical models for at least some aspects of programs. In mathematics, an algebra consists of a set (called the carrier of the algebra), together with a finite set of total functions that have the carrier set as their common codomain. The algebras we learn in school, however, are usually those derived from number theory and programs are more diverse, if not richer, than operations on numbers. A somewhat more abstract notion, called signature algebras, has been used for some time to model abstract data types [GTW78]. A signature defines a set of typed operator symbols without specifying functions that would be the actual operators. Thus a signature defines a class of algebras, namely the algebras whose operators conform to the typing constraints imposed by the signature. Signature algebras have been helpful in understanding the issues involved in abstract data types, type classes, program modularity and other software interface issues.

There is an even more abstract notion of algebras that has its origins in category theory. This notion, called *structure algebra*, emphasizes properties of morphisms, or structure-preserving maps. What is meant by structure in this context is that determined by a functor on the category of sets<sup>2</sup>. Because this notion of what constitutes an algebra is so general, it encompasses a huge variety of morphisms, or functions on sets.

A particularly interesting class of structure algebras occurs when the functors that specify the structure are chosen to correspond to the signatures that

---

<sup>\*</sup> The research on which this chapter is based was supported by the USAF Materiel Command.

<sup>2</sup> Technically, we should say a functor  $Set \rightarrow Set$ , where  $Set$  is the category whose objects are sets and whose arrows are (total) functions. We trust no confusion will arise from referring to such a functor as acting “on the category of sets”.

define signature algebras. The set of functions that can be characterized as morphisms of this class of structure algebras is vast—it appears to cover the space of functions that can be computed by functional programs provably terminating in Peano arithmetic. This chapter is about how to specify computable functions in terms of structure algebras, rather than by the more traditional, but less disciplined use of recursive equations. Some of the techniques are already familiar in functional programming—the higher-order functions *fold*, *reduce* or *catamorphisms* defined by various authors construct homomorphisms of structure algebras. The algebra of lists [Bir86] is actually a class a structure algebras.

There are several important advantages to specifying functions in terms of structure algebras, rather than recursive equations:

- Patterns of control can be formally specified and explicit, rather than informal and implicit. Control is derived from a specified signature.
- Termination conditions for a function defined as a structure-algebra morphism can also be derived from a specified signature.
- Algebraic programs have a semantic interpretation over sets and do not require a cpo interpretation.
- Proof rules for algebraic programs are inductive. Verification conditions necessary to prove a hypothesized property of a computation can be automatically derived.

There are also obligations imposed upon an algebraic functional programmer to ensure that a function specified in an algebraic formalism actually exists. The most stringent obligation is to prove totality of the function over a prescribed domain. In conventional functional languages, whose programs are specified by recursive equations, the only obligation is to prove type correctness, which is usually established by an automated type reconstruction algorithm.

In this chapter the reader will be introduced to ADL—an Algebraic Design Language—in which functions are defined in terms of a family of type-parametric combinators, rather than through explicitly recursive, equational declarations. ADL is based upon the categorical notion of structure algebras and coalgebras, although only the algebraic part is discussed here [KL94]. Data types in ADL are the carriers of its free algebras. They correspond closely to the data types of conventional functional programming languages such as ML, Haskell or Miranda. Just as the structure of an algebra is determined from a signature declaration, the data types of ADL are also extracted from its signature declarations.

ADL is an implemented language. Its initial implementation has been built as an extension to Standard ML, and supports interactive use. The initial application for ADL has been in a system that calculates program generators from specifications given in a domain-specific mini-language [B<sup>+</sup>94, KB<sup>+</sup>95].

The concept of algebras as they are used in ADL is discussed informally in Section 2. The concepts are more rigorously defined in Section 3, which provides the mathematical background for the rest of the chapter. Although many of the concepts have arisen from category theory, only a superficial knowledge of categories is assumed of the reader. In Sections 4 and 5, structure algebras

become elements of programs. A catamorphism combinator provides implicitly recursive control structures and a system of proof rules is introduced. Section 6 extends the scope of algebraic control structures to include functions whose underlying structure is less obvious, functions that are not catamorphisms. With its extended scope, algebraic programming now supports the definition of a huge space of total functions, without recourse to explicit recursion. Verification conditions are also discussed in this section. The type system of ADL is given an abbreviated treatment in 7. Section 8 demonstrates how monads fit naturally into an algebraic framework. The chapter concludes by illustrating the formulation of an algorithm for contraction of lambda-calculus terms.

There are related studies of the use of higher-order combinators (including catamorphism) in theoretical programming [MFP91, Fok92], however, none has previously been incorporated into a practical system for program development. The origin of such techniques appears to lie in the work of the *Squiggol* school [Bir86, Bir88, Mee86], subsequently influenced by a thesis by Hagino [Hag87] in which morphisms of data types are generalized in a categorical framework. A categorical programming language called *Charity* [CS92] embodies inductive and coinductive control structures based upon a categorical framework. The characterization of data types as structure algebras (and coalgebras) [Mac71] can be attributed to Hagino.

## 2 Algebras, Types and signatures

ADL is a higher-order, typed functional language whose type system is inspired by concepts from the theory of order-sorted algebras, from Martin-Löf’s type theory and from the Girard-Reynolds second-order lambda calculus. While ADL does not provide the full generality of the second-order lambda calculus, it uses a constrained form of abstraction on types and it contains combinators that are indexed by the names of type constructors. Its type system is sufficiently rich that type-checking is not decidable.

Nevertheless, the ADL type system is amenable to an abstract interpretation that is similar to the Hindley-Milner system with consistent extensions. Type inference in the Hindley-Milner system, while of exponential complexity in the worst case, has been shown to be feasible in practice through years of experience with its use in several functional programming languages. The Hindley-Milner system, which embodies a structural notion of type, guarantees the slogan

“Well-typed programs don’t go wrong”.

This means that programs that satisfy the structural typing rules respect the signatures of multi-sorted algebras—integer data are never confused with floating-point numbers or with functions, for instance. ADL adds to the structural typing restrictions the further requirement that

“Well-typed programs always terminate”.

This implies that the type system accommodates the precise description of sets that constitute the domains of functions definable in ADL. Accurate type-checking in ADL requires the construction of proofs of propositions. This task is made substantially easier than it would be in an untyped linguistic framework by the underlying approximation furnished by structural typing.

In Standard ML and related languages, the Hindley-Milner type system is extended with data type declarations. A data type declaration names a type and specifies a finite set of data constructors. An ML data type name may have one or more type variables as parameters, and thus actually names a type constructor. A type variable introduced as a parameter in a data type declaration is bound by abstraction. Application of a type constructor to a type expression can be understood syntactically, as the substitution of the argument expression for all occurrences of the type variable in the data type declaration.

In ADL, data type declarations are generalized to signature declarations that specify algebraic varieties<sup>3</sup>. Following the conventions of multi-sorted algebras, we call the names of types and type constructors *sorts*. The generalization can be summarized in the following table:

Parameterized data types	Varieties
type	algebra
type constructor	sort
data constructor	operator

The *arity* is a syntactic property of a sort. The arity indicates how to form type expressions from sorts. A sort with nullary arity, designated by  $*$ , is said to be *saturated*. A sort with non-nullary arity, designated by  $* \rightarrow *$ , is said to be *unsaturated*. A saturated sort expression is either a saturated sort or an unsaturated sort applied to a saturated sort expression. A saturated sort expression denotes a type in ADL.

ADL departs significantly from functional programming languages such as ML by providing signature declarations that introduce varieties of both signature and structure algebras, not simply data types. The signature of an algebraic variety consists of a finite set of operator names, together with the type of the domain of each operator. The codomain of an operator is the carrier type for the particular algebra to which the operator belongs. In computing, we often need to work with algebras in which there are several carrier types, although such complications are rarely of interest in mathematics. To describe a variety of algebras with several carriers, the operators that have a common codomain are grouped into sets. These sets are called sorts and the sorts are named in a signature declaration.

To determine a signature algebra of a given variety, a carrier type is specified for each sort of the variety, and a well-typed function is specified for each operator of each sort. We shall sometimes refer to a particular signature algebra as being “concrete”, to distinguish it from an unspecified algebra of its variety.

---

<sup>3</sup> A variety is a class of algebras having a common signature.

If all that we got from a signature was a variety of signature algebras, then we would be hard-pressed to claim any novelty for ADL. Signature algebras have been thoroughly explored as a basis for the OBJ family of languages [GT79, GW88]. However, signatures can also be modeled abstractly by a particular class of functors in the category of sets. These functors have fixed points which are objects in the same category, i.e. sets. This fact has long been known and exploited to define (recursively) data types in functional programming languages. It is less well known that the fixed points of these functors define whole categories of algebras, called structure algebras, which have useful interpretations for programming. This is the topic of our discourse.

## 2.1 Some familiar algebras

Signature declaration in ADL generalizes data type declaration in ML. Where we would declare a *list* data type in Standard ML by writing

```
datatype 'a list = nil | cons of ('a * 'a list)
```

the corresponding declaration of a variety of *ListD*-signature-algebras is written in ADL as:

```
signature List(a){type c; list/c = {$Nil, $Cons of (a * c)}}
```

A signature algebra is characterized by a carrier and a set of typed operators. A variety of signature algebras specifies neither the carrier nor the operators, but only their typings, in terms of the carrier type variables, the type parameter of the variety, and possibly some constant types or type constructors.

The above declaration asserts *List(a)* to be the name of a variety of algebras that is parameterized on a single type (designated by the type variable *a*) and further declares a name for a single sort, *list*. The sort is unsaturated because the variety has a type parameter; thus *list* : \* → \*. The type variable *c* stands for the carrier type. The signature declares two operator symbols of sort *list*, with typings:

$$\mathit{\$Nil} : c \qquad \mathit{\$Cons} : a \times c \rightarrow c$$

To emphasize that *\$Nil* is an operator, it could have been given a function type,  $1 \rightarrow c$  (where  $1$  is the type of a singleton set), by declaring it as “*\$Nil of 1*”. Since  $1$  denotes a singleton set, every function in the type  $1 \rightarrow c$  is isomorphic to an element in  $c$ .

Operator names always begin with ‘\$’ to distinguish them from other identifiers. A concrete algebra is specified by providing bindings for the carrier type and for each operator of the algebra.

Each signature declaration implicitly defines one specific algebra. This is the algebra of free terms, whose operators are unconstrained. The equational theory of this algebra is empty; syntactically distinct terms are semantically distinct. The operators of the free term algebra are commonly called *data constructors* and the set of terms constructed by well-typed applications of these operators is called a (free) data type.

The names of the data constructors of a free term algebra definable in ADL are derived from the names of operators in a signature, by dropping the initial ‘\$’ symbol. The name of a free data type is taken from a sort name. In case the sort is unsaturated, it names a *type constructor* of free data types, rather than a data type.

Here are the declarations of some other signatures that define useful varieties of algebras in ADL:

```
signature Nat{type c; nat/c = {$Zero, $Succ of c}}
signature Tree(a){type c; tree/c = {$Tip of a, $Fork of (c * c)}}
signature Bush(a){type c; bush/c = {$Leaf of a, $Branch of list(c)}}
```

Note that *nat* is a saturated sort, while *tree* and *bush* are both unsaturated.

The use of signatures to declare free data types is a familiar aspect of typed, functional languages. However, signatures also induce control structures, and that is the main point we wish to make. We shall see in Section 4 how powerful control structures can be derived from signature declarations.

### 3 Mathematical preliminaries

Before proceeding further, we shall give some essential definitions of mathematical concepts that underlie algebraic programming.

#### 3.1 Signatures and functors

We begin with definition of a signature as a syntactic entity, sufficiently rich for our purposes.

##### Definition 1. Signatures

Let  $TC$  be a set, called type constructors. A signature is an indexed set of sets, possibly abstracted on a parameter,  $a$ ,

$$\Sigma(a) = (S, \{\Sigma_s \mid s \in S\})$$

and has the following structure:

$S$  is a finite set of identifiers called sorts.

Each  $\Sigma_s$  is a triple,  $(c_s, I_s, \{\kappa_i \text{ of } \sigma_i \mid i \in I_s\})$ , where

$c_s$  is an identifier, unique in  $\{a\} \cup \{c_t \mid t \in S\}$ ,

$I_s$  is a finite set of indices,

each  $\kappa_i$  is an identifier, unique in  $\{\kappa_j \mid j \in I_s\}$ ,

each  $\sigma_i$  is a term derived from the following grammar:

$$\begin{array}{l} \sigma ::= c_t \quad t \in S \\ \quad \mid g(\sigma) \quad g \in TC \\ \quad \mid a \\ \quad \mid \sigma * \sigma \quad (*) \text{ is associative} \\ \quad \mid \mathbf{1} \end{array}$$

□

The interpretation intended for a signature is that  $a$  and  $c_s$  (for  $s \in S$ ) denote sets (i.e. types of ADL), while the elements of  $TC$  denote functors in the category of sets (i.e. type constructors of ADL). For each sort  $s$ , the  $\kappa_j$  in  $\Sigma_s$  are operator symbols of the sort and denote functions. The domain of the function denoted by  $\kappa_i$  is given by the interpretation of  $\sigma_i$ . The codomain of  $\kappa_i$  is the carrier of sort  $s$ , which is the interpretation of  $c_s$ . The symbol  $(*)$  denotes the operator forming the product of two sets.

Given the interpretation described above, it is straightforward to associate with a signature, a sort-indexed family of multi-functors on the category of sets<sup>4</sup>. The component functor associated with a sort  $s$  is derived from the  $s$ -sorted component of the signature,

$$\Sigma_s = (c_s, [1..n_s], \{\kappa_1 \text{ of } \sigma_1, \dots, \kappa_{n_s} \text{ of } \sigma_{n_s}\})$$

The object parameters of the associated multi-functor are the several variables of the signature that have interpretations as types, namely,  $a$  and  $c_t$  for each  $t \in S$ . Let us call this set  $V$ . Then the multi-functor indexed by sort  $s$  is defined to be

$$E_s(V) = F_1(V) + \dots + F_{n_s}(V)$$

where each of the  $F_i$  is also multi-functor and  $(+)$  denotes the coproduct bifunctor on the category  $\mathcal{Set}$ . For each index  $i \in [1..n_s]$ , the term  $\sigma_i$  is of the form

$$\sigma_i = \sigma_{i,1} * \dots * \sigma_{i,m_i}$$

where none of the  $\sigma_{i,j}$  contains the operator  $(*)$ . Thus each of the  $\sigma_{i,j}$  is either a variable from  $V$  or  $1$  or an expression of the form  $g(\sigma')$ , where  $g \in TC$  is a type constructor and  $\sigma'$  is a term over  $V$ . From this expansion, we determine the form of each component multi-functor,

$$F_i(V) = \bar{\sigma}_{i,1} \times \dots \times \bar{\sigma}_{i,m_i}$$

in which  $(\times)$  denotes the set product and  $\bar{\sigma}_{i,j}$  denotes the interpretation of  $\sigma_{i,j}$ . The multi-functor  $E_s$  derived from each sort of a signature has the form of a sum of products.

*Example 1.* The signature *List* has a single sort and a parameter. The bifunctor derived from this signature by the process described above is:

$$E_{List}(a, c) = \mathbf{1} + (a \times c)$$

For the signatures *Nat*, *Tree* and *Bush* specified in Section 2.1 the derived sum-of-products functors are:

$$\begin{aligned} E_{Nat}(c) &= \mathbf{1} + c \\ E_{Tree}(a, c) &= a + (c \times c) \\ E_{Bush}(a, c) &= a + list(c) \end{aligned}$$

---

<sup>4</sup> By a multi-functor, we mean a functor parametric on multiple object variables. A bifunctor is a multi-functor of rank 2, for instance.

□

An important fact is that a sum-of-products functor has a fixed point in  $\mathcal{Set}$  [Mal90]. That is, given a sum-of-products functor  $E_s$ , there is a set  $s'$  such that  $E_s(s') \cong s'$ , where  $(\cong)$  denotes the relation of (natural) isomorphism<sup>5</sup>. In case the signature from which the functor is derived takes a parameter, the result still holds but the fixed point is indexed by the parameter. For example, if  $E_s$  is a bifunctor, then holding the first parameter constant, we look for a fixed point in the second parameter. The fixed point is then indexed by the (fixed) first parameter, i.e.  $E_s(a, s'(a)) \cong s'(a)$ . A sort-indexed family of functors also has a fixed point, which is a sort-indexed set of sets.

It is customary to designate the elements of the fixed point of the family of multi-functors derived from a signature by the sort names that index them. For example, we designate by  $nat$  the fixed point of  $E_{Nat}$ , and by  $list(a)$  the fixed point of the bifunctor  $E_{List}$  at the point  $a$ . These are categorical models of the familiar data types of the same names.

The natural isomorphism at the fixed point of a functor is designated by

$$\mathbf{in}_s : E_s(s') \rightarrow s'$$

The natural isomorphism has an obvious interpretation in the context of a functional programming language. It is the ensemble of data constructors associated with the sort  $s$ . These data constructors correspond to the operators of a free term algebra defined by the signature. Thus, for the free  $nat$ -algebra, the isomorphism is  $\mathbf{in}_{nat} = [Zero, Succ] : 1 + nat \rightarrow nat$  and for the free  $list$ -algebras it is  $\mathbf{in}_{list} = [Nil, Cons] : 1 + a \times list(a) \rightarrow list(a)$ . The inverse isomorphism corresponds to a case analysis on the freely constructed terms, discriminating terms by the data constructor whose application is outermost. Naturality of the isomorphism in the parameter,  $a$ , implies that the data constructors of the corresponding data type constructor have types that are polymorphic with respect to  $a$ .

When  $E_s$  is a bifunctor, its fixed point is also a functor. Specifically, if  $s(a)$  is the object at the fixed point, that is,  $s(a) = \mathbf{in}_s(E_s(a, s(a)))$ , then there is a higher-order function,  $map_s : (a \rightarrow b) \rightarrow s(a) \rightarrow s(b)$ , that satisfies the equations

$$\begin{aligned} map_s id_a &= id_{s(a)} \\ map_s (f \circ g) &= map_s f \circ map_s g \end{aligned}$$

where  $id_a$  denotes an identity function on the type  $a$  and  $f : b \rightarrow c$ ,  $g : a \rightarrow b$  for types  $a$ ,  $b$  and  $c$ . These properties have long been known by programmers to hold for the data type constructor  $list$ . The function  $map_s$  corresponds to the morphism mapping part of the functor  $s$ .

---

<sup>5</sup> The fixed points of interesting sum-of-products functors are unique up to isomorphism. There are, however, functors whose fixed points are not even principal. The identity functor,  $I$ , is an example, as every set is a fixed point of  $I$ . We do not have a characterization of the interesting functors.

As a consequence of the fact that the fixed point of a bifunctor (with respect to one of its arguments) is a functor in the remaining argument, the morphism mapping associated with freely constructed data type has the following characterization:

**Corollary 2.** *map*—the morphism mapping of a sort-indexed family of functors Let  $\Sigma(a) = (S, \{\Sigma_s \mid s \in S\})$  be a parameterized signature and let  $f : a \rightarrow b$ . Let

$$\Sigma_s = (c_s, [1..n_s], \{\kappa_1 \text{ of } \sigma_1, \dots, \kappa_{n_s} \text{ of } \sigma_{n_s}\})$$

for  $s \in S$ . Then for each sort  $s \in S$ ,  $\text{map}_s f : s(a) \rightarrow s(b)$  satisfies an equation:

$$\text{map}_s f = \lambda x. \text{ case } x \text{ of}$$

:

$$\kappa_i(x_1, \dots, x_{m_i}) \Rightarrow \kappa_i(y_1, \dots, y_{m_i})$$

$$\text{where } y_j = \begin{cases} f x_j & \text{if } x_j : a \\ \text{map}_{s'} f x_j & \text{if } x_j : s'(a) \quad \text{where } s' \in S \\ \text{map}_t(\text{map}_{s'} f) x_j & \text{if } x_j : t(s'(a)) \quad \text{where } t \text{ is an unsaturated} \\ & \text{sort of a signature } \Sigma' \neq \Sigma \\ & \text{and } s' \in S \end{cases}$$

□

### 3.2 Structure algebras

The fundamental concept of structure algebras is quite simple. Structure algebras comprise varieties whose structure is induced by a functor on the category  $\mathbf{Set}$  [Mac71]. The functor determines the structure. The following definitions have been specialized, for simplicity, to the case of single-sorted algebras.

**Definition 3.** Structure algebra

Let  $t$  be a functor on the category  $\mathbf{Set}$ . A *t-structure algebra* (or *t-algebra*, for short) is a pair  $(c, h)$ , where  $c$  is a type called the *carrier* of the algebra and  $h : t(c) \rightarrow c$  is called its structure function.

□

**Definition 4.** A *t-algebra morphism* is a function that maps one *t-algebra* into another. Let  $\Sigma(a)$  be a signature with a single sort,  $t$ . (Recalling that an unsaturated sort denotes a functor.) A *t-algebra morphism* is a function  $f : a \rightarrow b$  that satisfies the commuting diagram below:

$$\begin{array}{ccc} t(a) & \xrightarrow{\text{map}_t f} & t(b) \\ \downarrow h & & \downarrow k \\ a & \xrightarrow{f} & b \end{array}$$

The diagram displays the equation

$$f \circ h = k \circ \text{map}_t f$$

Note that both  $h$  and  $k$  are structure functions of  $t$ -algebras.

□

*Example 2.* A *list*-algebra morphism. Let  $\text{exp2} = \lambda n. 2^n$ , and let  $\text{sum}$  and  $\text{product}$  be the functions that reduce a list of non-negative integers to a single integer by addition and multiplication, respectively. Then the following diagram illustrates  $\text{exp2}$  as a morphism of *list*-algebras:

$$\begin{array}{ccc}
 \text{list}(\text{int}) & \xrightarrow{\text{map\_list exp2}} & \text{list}(\text{int}) \\
 \text{sum} \downarrow & & \downarrow \text{product} \\
 \text{int} & \xrightarrow{\text{exp2}} & \text{int}
 \end{array}$$

□

The following theorem summarizes an important property of varieties of signature algebras.

**Theorem 5.** Homomorphism

Let  $\Sigma$  be a signature with a single sort,  $t$ , and let  $E_t$  be the functor derived from  $\Sigma$ . There is a set,  $\bar{t}$ , and a natural isomorphism  $\mathbf{in}_t : E_t(\bar{t}) \rightarrow \bar{t}$  which together define a free algebra,  $(\bar{t}, \mathbf{in}_t)$ .

Let  $(a, h)$  be a  $E_t$ -algebra. Then there is a unique  $E_t$ -algebra morphism,  $k$ , from the free algebra to  $(a, h)$ . The conclusion is summarized in the following commuting square:

$$\begin{array}{ccc}
 E_t(\bar{t}) & \xrightarrow{\text{map}_{E_t} k} & E_t(a) \\
 \mathbf{in}_t \downarrow & & \downarrow h \\
 \bar{t} & \xrightarrow{k} & a
 \end{array}$$

□

The dotted arrows indicate that the function,  $k$ , that makes the diagram commute is uniquely determined from the other data in the diagram.

## 4 Algebras and catamorphisms

A signature algebra provides a set of operators that can be used in constructing terms that represent calculations in the carrier type, as the familiar arithmetic operators do for integers. It also provides exactly the data needed to specify a function that calculates a value from a data-structure representation of a term in the free algebra of the signature. For example, the arguments that must be furnished to a *fold* function to reduce a list are the operators specified in a *list*-signature algebra.

The *list*-algebras that we shall see in Example 3 induce functions that sum a list of integers, calculate the length of a list, and concatenate two lists, respectively. A combinator to calculate these functions from the *list*-algebra specifications will be introduced in the next section. First, we shall see how algebras are specified in ADL.

A signature introduces names for the carriers and operators of a (multi-sorted) variety of signature algebras. To define a specific algebra of the variety, the carriers must be bound to types and the operators must be bound to functions in an algebra specification. In ADL, the symbol ( $:=$ ) is used to designate the binding of either a carrier or an operator.

*Example 3.* : Three different *List*-algebras are:

```
List(int){c := int; list{ $Nil := 0, $Cons := (+) }}
List(a){c := int; list{ $Nil := 0, $Cons := \ (x,y) 1+y }}
List(a){c := list(a) -> list(a);
        list{ $Nil := id,
              $Cons := \ (x,f) \ y Cons(x,f y) }}
```

where *id* is the polymorphic identity function, here instantiated with the type *list(a)*. The operator (+) belongs to a concrete algebra of integer arithmetic, which is predefined in ADL.

□

When a signature in ADL has only a single sort, as does *List*, an algebra specification may be abbreviated by omitting the inner set of curly braces and the sort name that is prefixed to the opening brace. Thus we could abbreviate the first algebra in the list of examples above, as

```
List{c := int; $Nil := 0, $Cons := (+)}
```

The free term algebra for the sort *list* can also be specified, although this specification is redundant, as its operators are the data constructors for *list* and these are implicitly defined when its signature is declared.

```
List(a){c := list(a); $Nil := Nil, $Cons := Cons}
```

It is important to keep in mind the distinction between data constructors in the free term algebra and operators in the signature of an algebra. The operators

bound to the same operator symbol, but in different algebras of the same variety may have different types. The data constructors are a special case of the operators for a specific algebra, and their types are fixed, up to variation in the type parameter of the variety.

#### 4.1 The catamorphism combinator

If  $t$  is a sort of a parameterized signature  $\Sigma(a)$ , a structure function of the class of  $t$ -algebras is any function  $h : t(a) \rightarrow a$ . Theorem 5 asserts that if  $\Sigma(a)$  has a free term algebra, then  $h$  is also the unique  $t$ -algebra morphism from  $(\mathit{Est}(a), \mathbf{in}_t)$  to  $(a, h)$ , and we call it a *homomorphism*. (Recall that the meaning of “morphism” is “form-preserving”. Here the form that is preserved is the underlying structure of the algebra.) More generally, the composition of a  $t$ -algebra morphism  $f : a \rightarrow b$  with a homomorphism, i.e.  $g = f \circ h : t(a) \rightarrow b$ , is a  $t$ -algebra morphism from the free term algebra, and is uniquely determined by the algebra of its codomain. A function whose domain is the free term algebra of a signature is called a *catamorphism*, borrowing the prefix “cata”=“down” from Greek [MFP91].

ADL defines a combinator, *red*, that takes an algebra specification to a catamorphism<sup>6</sup> of the algebraic variety. The *red* combinator obeys a homomorphism condition for each algebra on which it is instantiated. For the algebras we have considered, the homomorphism equations are given below. The bindings of operators in the combinator expressions are required to be correctly typed.

Let  $R\_nat(z, s) = red[nat] Nat\{c := \tau; \$Zero := z, \$Succ := s\}$   
 Then  $R\_nat(z, s) Zero = z$   
 $R\_nat(z, s) (Succ\ n) = s (R\_nat(z, s) n)$

Let  $R\_list(n, f) = red[list] List\{c := \tau; \$Nil := n, \$Cons := f\}$   
 Then  $R\_list(n, f) Nil = n$   
 $R\_list(n, f) (Cons\ (x, y)) = f\ (x, R\_list(n, f) y)$

Let  $R\_tree(t, f) = red[tree] Tree\{c := \tau; \$Tip := t, \$Fork := f\}$   
 Then  $R\_tree(t, f) (Tip\ x) = t\ x$   
 $R\_tree(t, f) (Fork\ (l, r)) = f\ (R\_tree(t, f) l, R\_tree(t, f) r)$

Let  $R\_bush(l, b) = red[bush] Bush\{c := \tau; \$Leaf := l, \$Branch := b\}$   
 Then  $R\_bush(l, b) (Leaf\ x) = l\ x$   
 $R\_bush(l, b) (Branch\ y) = b\ (map\_list(R\_bush(l, b)) y)$

The function *map\_list*, referred to in the last equation above, is defined below.

**Examples of list-algebra catamorphisms** Here are some examples of *List*-algebra catamorphisms constructed with *red[list]* and the algebra specifications given in Example 3:

<sup>6</sup> Other authors [MFP91] have used “banana” brackets,  $(| \_ |)$ , to designate the catamorphism combinator.

```

sum_list = red[list] List(int){c := int; $Nil := 0, $Cons := (+)}
length = red[list] List(a){c := int; $Nil := 0, $Cons := \ (x,y) 1+y}
append = red[list] List(a){c := list(a) -> list(a);
                        $Nil := id,
                        $Cons := \ (x,f) \ y Cons(x, f y)}

```

Further examples of *List*-algebra morphisms are:

```

map_list f = red[list] List(a){c := list(b);
                        $Nil := Nil,
                        $Cons := \ (x,y) Cons(f x, y)}

```

where *f* has the type *a* -> *b*, and

```

flatten_list = red[list] List(list(a)){c := list(a);
                        $Nil := Nil,
                        $Cons := append}

```

The typings of the constants defined by these equations are:

```

sum_list : list(int) -> int
length : list(a) -> int
append : list(a) -> list(a) -> list(a)
map_list : (a -> b) -> list(a) -> list(b)
flatten_list : list(list(a)) -> list(a)

```

**Examples of *nat*-algebra catamorphisms:** Catamorphisms of *nat*-algebras are enumerations of a carrier type. The function *ntoi* translates natural numbers from a representation in the free term algebra (i.e. a *Succ* representation) to a representation as positive integers.

```

ntoi = red[nat] Nat{c := int; $Zero := 0, $Succ := \ n 1+n}
add_nat x = red[nat] Nat{c := nat; $Zero := x, $Succ := Succ}

```

with typings

```

ntoi : nat -> int
add_nat : nat -> nat -> nat

```

**Examples of *tree* catamorphisms:**

```

sum_tree = red[tree] Tree(int){c := int;
                        $Tip := id,
                        $Fork := (+)}

list_tree = red[tree] Tree(a){c := list(a);
                        $Tip := \ x Cons(x,Nil),
                        $Fork := \ (x,y) append x y}

```

```

map_tree f = red[tree] Tree(a){c := tree(b);
                               $Tip := \x Tip(f x),
                               $Fork := Fork}

flatten_tree = red[tree] Tree(tree(a)){c := tree(a);
                                         $Tip := id,
                                         $Fork := Fork}

```

with typings

```

sum_tree : tree(int) -> int
list_tree : tree(a) -> list(a)
map_tree : (a -> b) -> tree(a) -> tree(b)
flatten_tree : tree(tree(a)) -> tree(a)

```

**Examples of *bush* morphisms:**

```

sum_bush = red[bush] Bush(int){c := int;
                                $Leaf := id,
                                $Branch := sum_list}

list_bush = red[bush] Bush(a){c := list(a);
                                $Leaf := \x Cons(x,Nil),
                                $Branch := flatten_list}

map_bush f = red[bush] Bush(a){c := bush(b);
                                $Leaf := \x Leaf(f x),
                                $Branch := Branch}

flatten_bush = red[bush] Bush(bush(a)){c := bush(a);
                                         $Leaf := id,
                                         $Branch := Branch}

```

with typings

```

sum_bush : bush(int) -> int
list_bush : bush(a) -> list(a)
map_bush : (a -> b) -> bush(a) -> bush(b)
flatten_bush : bush(bush(a)) -> bush(a)

```

*Exercise 6.* Reverse of a list

- A. Specify a *list*-catamorphism to compute the reverse of a list.
- B. Specify a second *list*-catamorphism with carrier type  $list(a) \rightarrow list(a)$  to define a function  $rev : list(a) \rightarrow list(a) \rightarrow list(a)$  that satisfies the equation

$$rev\ x\ Nil = reverse\ x$$

## 4.2 Primitive recursion and case analysis

Recall Kleene's primitive recursion scheme to define functions on natural numbers:

$$\begin{aligned} f(\text{Zero}, x_1, \dots, x_n) &= g(x_1, \dots, x_n) \\ f(\text{Succ } n, x_1, \dots, x_n) &= h(\text{Succ } n, f(n, x_1, \dots, x_n), x_1, \dots, x_n) \end{aligned}$$

where  $g : t_1 \times \dots \times t_n \rightarrow a$  and  $h : \text{nat} \times a \times t_1 \times \dots \times t_n \rightarrow a$ . Although the primitive recursion scheme can be represented as a *nat*-catamorphism, the representation is unnatural and if implemented directly, can result in algorithms with worse-than-expected performance. For instance, the **case** expression for type *nat* when expressed as a *nat*-catamorphism is

```

case x of           = snd(red[nat] Nat{c := nat * a;
  Zero => g           $Zero := (Zero, g),
  | Succ(x') => h x'  $Succ := λ(x, y) (Succ x, h x)})
end

```

Evaluation of the *nat*-catamorphism explicitly traverses the entire structure of a term to construct the argument needed in the successor instance of the case analysis. This takes time linear in the size of a *nat* term, whereas the *case* primitive is a constant time function.

To avoid the problem of introducing a computation time penalty for a primitive control structure, **case** has been adopted as a control combinator in ADL, with syntax similar to that of Standard ML for the benefit of familiarity. The domain of a **case** combinator is the free data type generated by an algebraic variety.

A primitive recursive function is, however, a structure function of *nat*-algebras, one in which the carrier always has the form *nat*\**a* for some type *a*. A function definable by primitive recursion can be calculated by taking the second projection of a pair calculated by a *Nat*-catamorphism.

*Example 4.* A factorial function is easy to define by primitive recursion. However, since the primitive recursion scheme is defined over natural numbers and we would prefer to do arithmetic in the domain of integers, some conversion is needed. The factorial function defined in this way has the typing *fact* : *nat* → *int*. It can be expressed in ADL as

```

fact = snd o red[nat] Nat{c := nat * int;
  $Zero := (Zero, 1),
  $Succ := \ (m, n) (Succ m, ntoi(Succ m) * n)}

```

In a later section we shall return to this example to obtain an integer-typed factorial function.

To define a general primitive recursion scheme for natural numbers, declare a combinator,  $Pr$ , by

$$Pr(g, h) = snd \circ red[nat] Nat\{c := nat * \tau; \\ \$Zero := (Zero, g), \\ \$Succ := \langle Succ \circ fst, h \rangle\}$$

in which the angle brackets designate a functional pair,  $\langle f, g \rangle x = (f x, g x)$ . This defines a family of  $nat$  algebras, with structure functions  $Pr(g, h) : nat \rightarrow \tau$ , for each pair  $(g : \tau, h : nat * \tau \rightarrow \tau)$ . In terms of this scheme, the factorial function can be defined more succinctly:

$$fact = Pr(1, \lambda(m, n) \text{ntoi}(Succ m) * n)$$

where the codomain type has been instantiated to  $int$ .

**Generalized primitive recursion** The primitive recursive control scheme can be generalized to algebras of other varieties. There is no special combinator in ADL for primitive recursion, but a primitive recursion combinator can be composed for each variety. For example, we shall define a primitive recursion for  $List$ -algebras,

$$Pr\_List(g, h) : list(a) \rightarrow \tau \\ Pr\_List(g, h) = snd (red[list] List(a)\{c := list(a) \times \tau; \$Nil := (Nil, g), \$Cons := h\})$$

where  $g : \tau$  and  $h : (a \times (list(a) \times \tau)) \rightarrow \tau$ .

*Exercise 7. Splitting a list*

Define  $splitat : char \rightarrow list(char) \rightarrow list(char) \times list(char)$

The function  $splitat$  is specified as follows: If the list  $xs$  contains an occurrence of the character,  $c$ , then  $splitat c xs$  yields the pair of the prefix and suffix of the first occurrence of  $c$  in  $xs$ . Otherwise, it yields the pair  $(xs, Nil)$ . Hint: Use primitive recursion for  $list$ .

### 4.3 Proof rules for catamorphisms

Inference rules for reasoning about catamorphisms of varieties of signature algebras can be calculated from their signatures. The form of each rule is an induction over the structure expressed by the signature. The proof rule for  $Nat$ -algebras is natural induction. Rules for the varieties introduced in Section 2.1 are summarized below. Notice that induction is not a structural rule of the logic. Rather, an inductive proof rule is introduced for each algebraic variety to account for the computational content of its catamorphisms. This has been noted previously by Goguen [Gog80] and others.

In the following rules,  $\tau$  designates a type expression. In the last rule, the notation  $y \in ys$  is the assertion that  $y$  is an element of the list  $ys$ .

$$\frac{f : \tau \quad P(\text{Zero}, f) \quad g : \tau \rightarrow \tau \quad n : \text{nat} \quad \forall x : \tau. P(n, x) \Rightarrow P(\text{Succ } n, g \ x)}{\forall n : \text{nat}. P(n, \text{red}[\text{nat}] \text{Nat}\{c := \tau; \$\text{Zero} := f, \$\text{Succ} := g\} n)}$$

$$\frac{f : \tau \quad P(\text{Nil}, f) \quad g : a \times \tau \rightarrow \tau \quad xs : \text{list}(a) \quad \forall x : a. \forall y : \tau. P(xs, y) \Rightarrow P(\text{Cons}(x, xs), g(x, y))}{\forall xs : \text{list}(a). P(xs, \text{red}[\text{list}] \text{List}(a)\{c := \tau; \$\text{Nil} := f, \$\text{Cons} := g\} xs)}$$

$$\frac{f : a \rightarrow \tau \quad \forall x : a. P(\text{Tip}(x), f \ x) \quad g : a \times \tau \rightarrow \tau \quad u, v : \text{tree}(a) \quad \forall y, z : \tau. P(u, y) \wedge P(v, z) \Rightarrow P(\text{Fork}(u, v), g(y, z))}{\forall y' : \text{tree}(a). P(y', \text{red}[\text{tree}] \text{Tree}(a)\{c := \tau; \$\text{Tip} := f, \$\text{Fork} := g\} y')}$$

## 5 Multi-sorted signatures

It is often convenient to define a signature that has several sorts. ADL supports the declaration of signatures comprised of a finite set of sorts. The analogy in terms of ML-style data types would be a mutually recursive set of data type declarations.

For example, the types that correspond to the syntactic phyla<sup>7</sup> of a context-free grammar may be mutually recursive. When expressed as a signature, each phylum of an abstract grammar corresponds to a distinct sort. The signature of each sort is comprised of operators that correspond to the productions for a single phylum.

*Example 5.* Given an abstract grammar specifying arithmetic terms, we shall derive a variety of signature algebras that have the structure of these terms. Suppose the grammar of terms is stratified into additive and multiplicative terms. We define two phyla to represent these term classes,

**phyla:** *TERM, FACTOR*

The productions for each of these phyla consist of syntactic operators that take as arguments, terms from the indicated phyla:

Additive operators— <i>TERM</i>	
Add	<i>TERM, TERM</i>
Neg	<i>TERM</i>
Prim	<i>FACTOR</i>
Multiplicative operators— <i>FACTOR</i>	
Mpy	<i>FACTOR, FACTOR</i>
Subterm	<i>TERM</i>
Ident	<i>string</i>

<sup>7</sup> A phylum is a grouping in the top level of a classification hierarchy. By syntactic phyla we refer to a classification of terms of an abstract grammar.

The following two-sorted signature is derived from the grammar of terms by substituting a carrier for each phylum and substituting algebraic operator symbols for the syntactic operators. The primitive syntactic phylum, *string*, is generalized to an indeterminate parameter of the signature.

```
signature TermGram(a){type c, d;
    term/c = {$Add of c*c,
              $Neg of c,
              $Prim of d}
    factor/d = {$Mpy of d*d,
               $Subterm of c,
               $Ident of a}}
```

The operators *\$Prim* and *\$Subterm* coerce a value from a representation in one carrier to a representation in the other.

□

The catamorphism combinator accommodates multi-sorted signatures by using the sort names as an index set. It takes a sort and a sort-indexed signature-algebra specification as its arguments and it yields a structure function for the specified sort in the induced structure algebra. The sort given as the first argument of the combinator determines the typing of the combinator expression. For example, the two catamorphism forms of *TermGram* algebras have typings:

$$\begin{aligned} \text{red}[term] \text{ TermGram}(a)\{c := \tau_1, d := \tau_2; \text{term}\{\dots\} \text{factor}\{\dots\}\} : \text{term}(a) \rightarrow \tau_1 \\ \text{red}[factor] \text{ TermGram}(a)\{c := \tau_1, d := \tau_2; \text{term}\{\dots\} \text{factor}\{\dots\}\} : \text{factor}(a) \rightarrow \tau_2 \end{aligned}$$

*Example 6.* To define an arithmetic calculator for terms, where the parameter *a* is specialized to the type *string*, apply *red[term]* to the following *TermGram*-algebra:

```
TermGram(string){c := int; d := int;
    term{$Add := (+), $Neg := (^), $Prim := id},
    factor{$Mpy := (*), $Subterm := id, $Ident := string_to_int}}
```

where *string\_to\_int* is a conversion function that has not been defined here.

□

*Exercise 8.* Printing the image of a term

Define a function to translate values of type *term(string)* into strings of **ascii** characters, with infix operator symbols for *\$Add* and *\$Mpy*, using an implicit operator precedence and with the minimum number of parentheses necessary to avoid incorrect associations. Assume that *\$Add* and *\$Mpy* are associative operators.

## 6 Functions that are not catamorphisms

While catamorphisms are elegant constructions, they do not solve all problems of programming. There are many instances of functions whose control is determined by a structure that is not visible in the domain. For instance, a recursive-descent parser is controlled by the structure of a grammar for the language being parsed. That structure is discovered in the list of input tokens by the action of the parser; it is not manifest. A divide-and-conquer algorithm such as Quicksort is controlled by the structure of a binary tree, yet it is applied to lists, not to trees. Such examples abound.

In this section we shall examine functions that can be defined as morphisms of a structure algebra, although they are not catamorphisms. We shall introduce a combinator to construct such functions from a  $t$ -algebra specification and a new constituent, a partition relation that analyzes its argument to find a  $t$ -algebraic structure. The ability to construct such functions has led us to the idea that algebraic structures are sufficiently general to write most programs of interest. Recall the diagram in terms of which a  $t$ -algebra morphism is defined:

$$\begin{array}{ccc}
 t(a) & \xrightarrow{\text{map}_t f} & t(b) \\
 \downarrow h & & \downarrow k \\
 a & \xrightarrow{f} & b
 \end{array}$$

The catamorphisms illustrated in the diagram are  $h$ ,  $k$ ,  $\text{map}_t f$  and the diagonal composition (upper left to lower right) arrow. The homomorphism condition of the diagonal composition is the commutation condition for the diagram,

$$f \circ h = k \circ \text{map}_t f \tag{1}$$

Each catamorphism can be expressed in terms of the combinator  $\text{red}$  and the appropriate  $t$ -algebra. However,  $f$  in the diagram above is also a  $t$ -algebra morphism, and under certain conditions, it may be expressed in terms of a combinator.

Suppose there were a function  $p : a \rightarrow t(a)$  such that  $p \circ h = \text{id}_{t(a)}$ . This function need not be a right inverse for  $h$  on  $a$ , but it is a right inverse when restricted to a domain that is the  $h$ -image of  $t(a)$ ,

$$h \circ p = \text{id}_{a \downarrow h} \tag{2}$$

where the notation  $a \downarrow h$  means the set  $a$  restricted to the codomain of  $h$ . Post-composing both sides of (1) with  $p$  and making use of (2) gives an equation that is satisfied when the domain of  $f$  is restricted to the  $h$ -image of  $t(a)$ :

$$f = k \circ \text{map}_t f \circ p \tag{3}$$

This equation suggests a way to realize  $f$  as the composition of a catamorphism with a function that analyzes the domain  $a \downarrow h$ .

Let  $E\$t$  be a sum-of-products bifunctor and let  $t(a)$  be a fixed point of the functor with respect to its second argument. Let  $E\#t$  be the functor obtained by binding the second argument of  $E\$t$  at its fixed point, i.e.

$$E\#t(a) = E\$t(a, t(a))$$

Notice that  $e\#t$  is a sum-of-products functor. Thus  $E\#t(a) = \mathbf{in}_t^{-1} t(a)$  represents an explicit, one-level unfolding of the structure of terms in the set  $t(a)$ . Now suppose there is a function  $p' : a \rightarrow E\#t(a)$  that is isomorphic to a left inverse for  $h$  as described in the preceding paragraph. With this nomenclature, an isomorphic relative of equation (3) given above can be summarized in the diagram below, which reveals the structure more clearly:

$$\begin{array}{ccc}
 E\#t(a) & \xrightarrow{\text{map-}E\#t f} & E\#t(b) \\
 \uparrow p' & & \downarrow k \\
 a & \xrightarrow{f} & b
 \end{array}$$

The function  $p'$  can be expressed by a case analysis on the data constructors of its argument.

Following the suggestion outlined above, ADL provides a combinator with which to construct morphisms whose domains are  $t$ -algebras that are not free. We call this combinator  $hom$ . It takes three parameters;

1. the sort of the structure function that is to be mapped,
2. the structure algebra in the codomain of the morphism and
3. a partition relation that is “inverse” to the structure function of its domain algebra.

The partition relation is typically expressed as a conditional or a *case* expression that tests a value of type  $a$  to reveal the structure of the algebra. The codomain of the partition relation is  $E\$t(a, c)$ , where  $c$  is an unspecified type parameter. This object has the structure of a disjoint union of the domain types of the set of operators of a signature  $\Sigma$ , to which the sort  $t$  belongs.

Thus we write  $hom[t] T\{b; k\} p$ , where  $k : t(b) \rightarrow b$  and  $p : a \rightarrow E\#t(a)$ . Here is an example that illustrates a  $t$ -algebra morphism constructed using  $hom$ .

*Example 7.* : Calculate the largest power of 2 that factors a given positive integer.

Consider the *Nat*-signature algebra defined by:

$$Nat\{c := int; \$Zero := m, \$Succ := \lambda n \ 2 \times n\}$$

in which the free variable  $m$  represents an odd, positive integer. The carrier of this algebra is the set consisting of  $\{m, 2m, 4m, 8m, \dots\}$ .

From this algebra, we construct a catamorphism that defines a structure function for a *nat*-structure algebra,

$$h : nat \rightarrow int$$

$$h = red[nat] Nat\{c := int; \$Zero := m, \$Succ := \lambda n 2 \times n\}$$

To invert this structure function (up to isomorphism of types), construct a function that recovers the natural number giving the power of two that multiplies  $m$  in forming any element of the carrier. Let

$$p = \lambda n \text{ if } n \bmod 2 <> 0 \text{ then } \$Zero$$

$$\text{ else } \$Succ(n \text{ div } 2)$$

Then  $p$  is given the typing

$$p : int \rightarrow E\$nat\ int$$

where  $E\$nat$  is a derived, unsaturated type constructor. This type constructor belongs to no declared signature, thus cannot form the type of the domain or codomain of explicitly defined functions.

Notice that in the above definition, the operators of the *Nat*-algebra,  $\$Zero$  and  $\$Succ$ , assume specific types by binding the carrier as *int*. These occurrences of  $\$Zero$  and  $\$Succ$  represent the operators of the particular *Nat*-algebra that structures the *int*-typed domain of the *Nat*-algebra morphism being defined.

To complete the solution of the problem, we need to specify a *Nat*-algebra that yields an integer representation of a power of 2. To give an exponent of two, we can use the algebra in which a natural number is represented as a positive integer. This algebra was used to specify the function *ntoi* in an earlier example. (Notice that the bindings given to the operator symbols  $\$Zero$  and  $\$Succ$  in this algebra are not the same as the bindings presumed in the definition of  $p$  above. In general, they need not even have the same typings.) Thus, we get an algorithm expressed in ADL as:

$$pwr\_2 = hom[nat] Nat\{c := int; \$Zero := 0, \$Succ := \lambda n 1+n\} p$$

The equation satisfied by  $pwr\_2$  is:

$$pwr\_2\ n = \text{if } n \bmod 2 \neq 0 \text{ then } 0$$

$$\text{ else } 1 + pwr\_2(n \text{ div } 2)$$

To obtain an explicit representation of the factor that is a power of 2, the *Nat*-algebra can be modified to calculate that factor. This solution is

$$pwr\_2' = hom[nat] Nat\{c := int; \$Zero := 1, \$Succ := \lambda n 2*n\} p$$

*Example 8.* : Integer factorial. The factorial function was used in Example 4 to illustrate primitive recursion, but the type of the factorial defined there was  $nat \rightarrow int$ , rather than the more usual type,  $int \rightarrow int$ . The reason for the abnormal typing was to be able to define *fact* by a *nat*-catamorphism. Now that the combinator *hom* is available, that is no longer a requirement.

We can define a partition relation to recover the natural number structure of a non-negative integer and define the factorial function that is expected:

```
fact = snd o hom[nat] Nat{c := int * int;
    $Zero := (0,1),
    $Succ := \ (m,n) (m+1, (m+1) * n)}
(\n if n=0 then $Zero else $Succ(n))
```

*Example 9.* : Filtering a list

The function *filter p* :  $list(a) \rightarrow list(a)$  reconstructs from a list given as its argument, a list of the subsequence of its elements that satisfy the predicate function  $p : a \rightarrow bool$ . This function could be directly constructed in terms of *list*-algebra catamorphisms. Instead, we propose an algebraic variety to represent the two cases that occur in filtering—an element of the list is either to be included or omitted.

```
signature Slist(a){type c;
    slist/c={ $Nomore, $Include of a*c, $Omit of c}}
```

A definition of *filter p* can be given as a morphism of *slist*-algebras:

```
filter p = hom[slist] Slist(a){c := list(a);
    $Nomore = Nil,
    $Include = Cons,
    $Omit = id}
(\xs case xs of
    Nil => $Nomore
  | Cons(x,xs') => if p x then $Include(x,xs')
                    else $Omit xs'
end)
```

□

*Example 10.* : Quicksort

The Quicksort algorithm requires two functions, one that partitions a list,

$$part : int \rightarrow list(int) \rightarrow list(int) \times list(int)$$

and another that sorts a list,

$$sort : list(int) \rightarrow list(int)$$

The function *part* can be defined as a *list*-catamorphism:

```

part a = red[list] List(int){c := list(int)*list(int);
      $Nil := (Nil,Nil),
      $Cons := \ (b,(xs,ys)) if b<a then (Cons(b,xs),ys)
                          else (xs,Cons(b,ys))}

```

The function *sort*, however, uses a divide-and-conquer algorithm with the structure of a binary tree. It can be expressed as a morphism of the following algebraic variety:

```

signature Btree(a){type c; btree/c = {$Emptytree, $Node of c*a*c}};

sort = hom[btree] Btree(int){c := list(int);
      $Emptytree := Nil,
      $Node := \ (xs,x,ys) append xs (Cons(x,ys))}
(\xs case xs of
  Nil => $Emptytree
| Cons(x,xs') =>
  let (ys,ys') = part x xs'
  in $Node(ys,x,ys')
end)

```

Notice that although the algorithm is controlled by a tree traversal, the sort function has type  $list(int) \rightarrow list(int)$ . There is no data structure corresponding to the data type  $btree(list(int))$ . This is a “treeless” tree traversal.

*Exercise 9.* Another form of bush

Given the following signature declaration,

**signature**  $Bush'(a)\{type\ c;\ bush'/c = \{\$Leaf'\ of\ a,\ \$Branch'\ of\ nat*(nat \rightarrow c)\}\}$

construct a morphism of type  $bush(a) \rightarrow bush'(a)$  that is invertible. (Construct its inverse, too.)

*Exercise 10.* Splitting a list more efficiently

The function *splitat* of Exercise 7, when defined by primitive recursion does more computation than is necessary. It recursively evaluates the function on the tail of a list that has already been successfully split. Redefine *splitat* in terms of *hom[list]*.

*Exercise 11.* Factors of a positive integer

Give a function, *factors*, that takes a positive integer  $N$  and a list of positive integers  $M$  to a list of the factors of  $N$  by  $M$ , and which satisfies the following equations:

$$\begin{aligned}
 factors\ N\ Nil &= Cons(N, Nil) \\
 factors\ N\ (Cons(m, M')) &= \\
 &\quad Cons(m, factors\ (N/m)\ (Cons(m, M'))) \quad \text{if } m \text{ divides } N \\
 factors\ N\ (Cons(m, M')) &= factors\ N\ M' \quad \text{otherwise}
 \end{aligned}$$

Prove that your solution satisfies the equations.

## 6.1 Termination conditions for morphisms of non-free algebras

Two kinds of proof obligations are necessary to establish properties of ADL programs: (1) termination proofs establish the existence of total functions specified by combinator expressions and are associated with type correctness, and (2) verification of hypothetical propositions about type-correct (and therefore terminating) programs. We shall first address the proposition of termination of functions defined with the combinator  $hom$ , then take up the structure of proof rules to verify other propositions.

Recall that for a construction  $hom[t]T(a)\{b; k\}p$  to be a  $t$ -algebra morphism, the partition relation  $p$  must be a left inverse of the structure function of a  $t$ -algebra,  $(\tau, h)$ . Since we do not know  $h$  in general, we require a condition that can be applied directly to  $p$  itself. Note that if  $p$  is a left inverse, it is also a right inverse to  $h$  on some subset of the elements of type  $a$ . Thus a well-typed partition relation is *formally* correct in the sense that it constructs results by well-typed application of operators of the signature  $T$ . However, its application to an arbitrary element  $x : \tau$  might fail to be defined;  $x$  may not be in the codomain of  $h$ . The additional requirement can be stated in terms of a total ordering on  $a$  that must be furnished to discharge the proof obligation.

The following definition characterizes termination in terms of a predicate that can be associated with a partition relation. The idea is that elements of a domain satisfy the predicate only if they are elements of a well-founded ordering that is compatible with the partition relation. Then if the domain of the partition relation,  $p$ , is restricted to the set characterized by the predicate, termination of an expression  $hom[s]A p$  will be assured by further establishing that  $red[s]A$  is a catamorphism.

**Definition 12.** Let  $\Sigma$  be a parameterized signature,  $\Sigma(a) = (S, \{\Sigma_s \mid s \in S\})$ , where for each  $s \in S$ ,  $\Sigma_s = (c_s, [1..n_s], \{\dots \kappa_i \text{ of } \sigma_{i1} \times \dots \times \sigma_{im_i} \dots\})$ . Let  $\{\tau_s \mid s \in S\}$  be a set of types (sets) and let  $D$  be a disjoint union of this set. That is, each element of  $D$  is a pair,  $(s, x)$ , where  $s \in S$  and  $x \in \tau_s$ . Let  $P$  be a predicate over  $D$ .

Suppose that  $(\prec) \subseteq D \times D$  is a well-founded ordering on the set  $\{x : D \mid P(x)\}$ . We say that a sort-respecting function  $p : D \rightarrow \bigcup_{s \in S} t(\tau_s)$  calculates a  $\Sigma$ -*inductive partition* of the sets  $\{x : \tau_s \mid P(s, x), s \in S\}$  if

$$\begin{aligned} \forall x : \tau_s. P(s, x) \Rightarrow \\ \forall \kappa_i \in T. p(s, x) = \kappa_i(y_1, \dots, y_{m_i}) \Rightarrow \\ \forall j \in 1..m_i \begin{cases} (t, y_j) \prec (s, x) & \text{if } \sigma_{ij} = c_t \\ \forall z : \tau_t. z \text{ elt}_{s'} y_j \Rightarrow (t, z) \prec (s, x) & \text{if } \sigma_{ij} = s'(c_t) \end{cases} \end{aligned}$$

where  $t \in S$  and  $s'$  is an unsaturated sort,  $s' \notin S$ , and  $\text{elt}_{s'}$  is an infix notation for the two-place predicate defined by cases:

$$\begin{aligned} z = x &\Rightarrow z \text{ elt}_{s'} \kappa'_i(y_1, \dots, x, \dots, y_{m_i}) \\ z \text{ elt}_{s'} y &\Rightarrow z \text{ elt}_{s'} \kappa'_i(y_1, \dots, y, \dots, y_{m_i}) \end{aligned}$$

for all operators  $\kappa'_i$  in  $\Sigma_{s'}$ .

□

In the definition above, the predicate  $P$  characterizes a domain on which the morphism is well-defined. Any properties of the morphism deduced with the proof rules of the  $t$ -algebra will be valid only for points of the domain that satisfy  $P$ . When the signature is single-sorted, it is unnecessary to introduce a disjoint union of domain types and to pair each domain element with a sort to distinguish the set from which it is drawn.

In Example 7, a suitable subset and its well-ordering is the positive integers with the natural order,  $(<)$ . The partition relation  $p$  induces a *Nat*-inductive partition on this subset. In Example 8 the same ordering is used but the set is the non-negative integers.

For Example 9, a suitable ordering on  $list(int)$  is  $xs \prec ys$  iff  $length\ xs < length\ ys$ . With this ordering, the predicate “universally true” induces an *Slist*-inductive partition on  $list(a)$ .

For Example 10, the same ordering of lists by their length is compatible with the partition function used in the definition of *sort*, and induces a *Tree*-inductive partition of  $list(int)$ . The verification condition for termination of the function *sort* of this example becomes

$$(xs = Cons(x, xs')) \wedge (part\ x\ xs' = (ys, ys')) \Rightarrow (ys \prec xs) \wedge (ys' \prec xs)$$

## 6.2 Proof rules for morphisms of non-free algebras

In Section 4.3, inductive proof rules for catamorphisms of signature algebras were discussed. Although the structure of morphisms of non-free algebras, defined with the combinator *hom*, involves the added complexity of a partition relation, these morphisms also have inductive proof rules that can be calculated from signatures. We shall not present an algorithm for calculating the rules, as this is complicated and technical, but shall illustrate the principle with examples for several of the varieties used as examples in the preceding sections.

**Generalized *Nat*-induction** The conclusion of a generalized induction rule is expressed in terms of a two-place predicate that relates the argument and the result of a morphism of a *Nat*-structure algebra. The argument variable is universally quantified over a domain characterized by a termination predicate for the morphism. The termination predicate is an external condition for logical validity of the rule—it is not expressed in the rule itself. While this may be regarded as a flaw in the logic from a foundational point of view, it has an advantage from a pragmatic viewpoint. Separating the proof obligations to show termination from those required to prove other properties affords flexibility in constructing deductions, and allows deductions to be modular.

The hypotheses of a generalized induction account for analysis of the argument of a morphism by the partition relation, as well as for construction of the result by the operators of the specified signature algebra. From the signature *Nat*, we derive the rule:

$$\begin{array}{c}
z : \tau_2 \\
\forall x : \tau_1. (p \ x = \$Zero) \Rightarrow P(x, z) \\
\\
g : \tau_1 \rightarrow \tau_1 \quad s : \tau_2 \rightarrow \tau_2 \\
\forall x : \tau_1. \forall y : \tau_2. (p \ x = \$Succ(g \ x)) \Rightarrow P(g \ x, y) \Rightarrow P(x, s \ y) \\
\hline
\forall (x : \tau_1 \mid T(x)). P(x, \text{hom}[\text{nat}] \text{Nat}\{c := \tau_2; \$Zero := z, \$Succ := s\} \ p \ x)
\end{array}$$

in which  $T$  is a predicate characterizing a domain on which the morphism is defined (i.e. its computation terminates).

It is instructive to apply this rule to a simple example. Let

$$\begin{array}{c}
\tau_1 = \text{int} \quad \tau_2 = \text{int} \\
p \ x = \mathbf{if} \ x = 0 \ \mathbf{then} \ \$Zero \ \mathbf{else} \ \$Succ(x - 1) \\
z = 0 \\
s = \lambda n \ n + 1
\end{array}$$

and interpret  $P$  as equality on integers. The termination predicate for this example must restrict the domain to the non-negative integers. The hypotheses of the generalized induction become

$$\begin{array}{c}
\forall x : \text{int}. (x = 0) \Rightarrow P(x, 0) \\
\forall x : \text{int} \forall y : \text{int}. (x \neq 0) \Rightarrow P(x - 1, y) \Rightarrow P(x, y + 1)
\end{array}$$

These conditions are satisfied if  $P$  is interpreted as equality in the algebra of integer arithmetic.

**Generalized List-induction** A rule for generalized *list*-induction follows the form of that for the signature *Nat*, but with an additional analysis function. When a domain element is partitioned to match the template of the domain of the operator  $\$Cons$ , a value of the list element type is produced, as well as a value of the type of the original domain. The rule is:

$$\begin{array}{c}
n : \tau_2 \\
\forall x : \tau_1. (p \ x = \$Nil) \Rightarrow P(x, n) \\
\\
g_1 : \tau_1 \rightarrow a \quad g_2 : \tau_1 \rightarrow \tau_1 \quad f : a \times \tau_2 \rightarrow \tau_2 \\
\forall x : \tau_1. \forall y : \tau_2. (p \ x = \$Cons(g_1 \ x, g_2 \ x)) \Rightarrow P(g_2 \ x, y) \Rightarrow P(x, f(g_1 \ x, y)) \\
\hline
\forall (x : \tau_1 \mid T(x)). P(x, \text{hom}[\text{list}] \text{List}\{c := \tau_2; \$Nil := n, \$Cons := f\} \ p \ x)
\end{array}$$

*Exercise 13.* State the generalized induction rule for the signature *Btree* given in Example 10. Use this rule to derive conditions that must be satisfied by the function *part*, as defined in the example, to verify that a result calculated by the function *sort* is a sorted permutation of a list given as its argument.

## 7 The ADL type system

Logical properties of structure-algebra morphisms can be derived by inductive proof rules. Each such property can be formalized as a predicate over a set. ADL types can be interpreted as sets, although as we have seen in Section 6.2, proof obligations arise in verifying that a syntactically legal term is semantically well-founded.

Since types are sets, the restriction of a type by a predicate defines a set that may be considered to be a subtype of a structurally defined type. We call such subtypes *domain types*. An ADL domain type is expressed with set comprehension notation, as for instance,  $\{x : \tau \mid P(x)\}$ , where  $\tau$  is a structural type expression and  $P$  stands for a predicate. In the type system of ADL, domain types occur only on the left of the arrow type constructor. Domain types express restrictions in the types of functions.

---

### Syntax of type expressions

typ	::= identifier	primitive types
	typ * typ	products
	domtyp $\rightarrow$ typ	function types
	identifier(typ [, typ])	datatypes
domtyp ::= typ	{identifier : typ   Identifier(expr)}	restricted domain types

---

The Hindley-Milner type system is based upon a structural notion of type and is not expressive enough to distinguish among domain types of ADL. Thus, its type-checking algorithm is not powerful enough to ensure that a syntactically well-formed ADL expression is meaningful, but requires additional evidence as proof. Nevertheless, we find it useful to employ the Hindley-Milner type system as an approximation to ADL's type system. The Hindley-Milner type-inference algorithm is an abstract interpretation of ADL that approximates its type assignments. Whenever Hindley-Milner type checking asserts that an expression is badly typed, it cannot be well-typed in the ADL type system. When Hindley-Milner type inference assigns a type to an expression, that typing will be structurally compatible with any ADL typing of the expression.

For example, given a pair of ADL functions with typings  $f : \{x : \tau_1 \mid P(x)\} \rightarrow \tau_2$  and  $g : \{x : \tau_2 \mid Q(x)\} \rightarrow \tau_3$ , a structural (Hindley-Milner) typing approximates the ADL typings as  $f : \tau_1 \rightarrow \tau_2$  and  $g : \tau_2 \rightarrow \tau_3$ . It will judge their composition to be well-typed, with typing  $g \circ f : \tau_1 \rightarrow \tau_3$ . An ADL typing of the composition has the form  $g \circ f : \{x : \tau_1 \mid R(x)\} \rightarrow \tau_3$ , and it carries a proof obligation to show that  $R(x) \Rightarrow P(x) \wedge Q(f x)$ . To discharge the proof obligation requires a logical deduction based upon algebraic properties of the function  $f$ .

To determine whether a function application is well-typed is too complex for Hindley-Milner typing alone. To know that  $f a$  is well-typed, one must furnish evidence that  $P(a)$  holds. Function types in ADL may involve restrictions

expressed in domain types, and these restrictions might include arbitrary arithmetic formulas. For this reason, ADL does not have principal types, nor unicity of types. Domain restrictions are needed to express the termination conditions for combinators that express morphisms of non-initial structure algebras.

Domain restrictions must be expressible with first-order predicates. As a practical consequence, this implies that a domain restriction cannot assert a property of the result of applying a function-typed variable. For example, given a function  $f : \{x : \tau_1 \mid P x\} \rightarrow \tau_2$ , we can express the typing of a function that composes its argument on the left of  $f$  as

$$\lambda g. g \circ f : (\tau_2 \rightarrow \tau_3) \rightarrow \{x : \tau_1 \mid P x\} \rightarrow \tau_3$$

The type of the formal parameter,  $g$ , is only structural; it requires no domain predicate to be imposed.

If, however, we attempt to type the function  $\lambda h. f \circ h$  that composes its argument on the right of  $f$ , we find that it is impossible to do so with only a first-order domain predicate. The predicate must express that every point in the codomain of  $h$  satisfies the domain predicate  $P$ , and to express this restriction requires quantification over all points in the domain of  $h$ . The only kind of typing restriction that can be expressed of a function-typed variable is a domain restriction. Nevertheless, this can be quite powerful.

Given a proof that a function-typed variable satisfies a domain restriction at every occurrence in an expression, the variable may be abstracted from the expression and given a domain-restricted function type. For instance, suppose that in an expression  $\lambda x. e : \tau_1 \rightarrow \tau_3$ , the free variable  $f$  occurs in an applicative position and satisfies a structural typing  $f : \tau_1 \rightarrow \tau_2$ . If in addition, at every occurrence of  $f$  in  $e$  (each of the form  $f e'$ ) one can show that  $P x \Rightarrow R e'$ , then the abstraction can be given a typing  $\lambda f. \lambda x. e : (\{y : \tau_1 \mid R y\} \rightarrow \tau_2) \rightarrow \{x : \tau_1 \mid P x\} \rightarrow \tau_3$ .

An application of a function  $h : (\{y : \tau_1 \mid R y\} \rightarrow \tau_2) \rightarrow \tau_3$  to an argument  $e' : \{y : \tau_1 \mid Q y\} \rightarrow \tau_2$  is judged to be well-typed if there is a proof that  $\forall y : \tau_1. R y \Rightarrow Q y$ . This condition ensures that any expression to which  $h$ 's formal parameter might be applied will satisfy the restriction imposed by the domain predicate in the typing of  $e'$ .

## 8 Monads

Monads are mathematical structures that have found considerable use in programming. Knowing that a program is to be interpreted in a particular monad allows us to “take for granted” the structure of the monad without explicit notation. Common examples are monads of exceptions (we take for granted that exceptions are propagated, and shall only express unexceptional terms) and monads of state transformers (we take for granted that state is threaded through computations in a deterministic order).

Monads have been found useful in computer science relatively recently [Mog91, Wad90]. Monads have been used to explain control constructs such as exceptions [Spi90] and advocated as a basis for formulating reusable modules [Wad92].

A variety of monads cannot be specified with the sorted signature declarations available in ADL. Instead, there is a predefined variety, whose signature is

**signature**  $Monad(a)$  { **type**  $M(a)$ ;  
 $monad/M(a) = \{\$Unit\ of\ a,\ \$Mult\ of\ M(M(a))\}$  }

In a monad specification,  $M(a)$  can be substituted by a type expression in which the parameter  $a$  has only positive occurrences (with respect to the arrow constructor). Positive occurrences are defined in terms of predicates  $Pos_a$  and  $Neg_a$ , defined as follows:

$$\begin{aligned} Pos_a(a) &= true \\ Pos_a(b) &= true\ \text{if}\ b \neq a \\ Pos_a(X * Y) &= Pos_a(X) \wedge Pos_a(Y) \\ Pos_a(X + Y) &= Pos_a(X) \wedge Pos_a(Y) \\ Pos_a(X \rightarrow Y) &= Neg_a(X) \wedge Pos_a(Y) \\ Neg_a(a) &= false \\ Neg_a(b) &= true\ \text{if}\ b \neq a \\ Neg_a(X * Y) &= Neg_a(X) \wedge Neg_a(Y) \\ Neg_a(X + Y) &= Neg_a(X) \wedge Neg_a(Y) \\ Neg_a(X \rightarrow Y) &= Pos_a(X) \wedge Neg_a(Y) \end{aligned}$$

where  $a$  and  $b$  denote atomic type expressions. For example, the following propositions are satisfied, according to the definition:

$$\begin{aligned} Neg_a(a \rightarrow b) \\ Pos_b(a \rightarrow b) \\ Pos_a((a \rightarrow b) \rightarrow a) \end{aligned}$$

Neither  $Pos_a$  nor  $Neg_a$  holds of the expression  $a \rightarrow a$ , which contains both positive and negative occurrences.

A monad is not a free algebra; there are three equations to be satisfied:

$$mult_a^M \circ unit_{M(a)}^M = id_{M(a)} \quad (4)$$

$$mult_a^M \circ (map\_M\ unit_a^M) = id_{M(a)} \quad (5)$$

$$mult_a^M \circ mult_{M(a)}^M = mult_a^M \circ (map\_M\ mult_a^M) \quad (6)$$

There is another function that can be defined in terms of the components of a monad and it is often more convenient to use this function than  $mult^M$ . This is the natural extension,

$$\begin{aligned} ext^M &: (a \rightarrow M(b)) \rightarrow M(a) \rightarrow M(b) \\ ext^M\ f &=_{def} mult^M \circ map\_M\ f \end{aligned}$$

It is easy to prove a number of identities for  $ext$ ;

$$ext^M \text{mult}_a^M = id_{M(a)} \tag{7}$$

$$ext^M f \circ unit^M = f \tag{8}$$

$$ext^M (ext^M f \circ g) = ext^M f \circ ext^M g \tag{9}$$

$$ext^M id_{M(a)} = \text{mult}_a^M \tag{10}$$

$$ext^M (unit^M \circ f) = \text{map}_M f \tag{11}$$

A function of the form  $unit^M \circ f : a \rightarrow M(b)$  or  $ext^M (unit^M \circ f) : M(a) \rightarrow M(b)$  is said to be *proper* for the monad, whereas a function with codomain  $M(b)$  that cannot be composed in this way is said to be non-proper.

To extend a function whose domain type is a product, i.e.  $f : a \times b \rightarrow M(c)$ , the monad  $M$  must be accompanied by a *product distribution* function,  $dist^M : M(a) \times M(b) \rightarrow M(a \times b)$ . This allows us to form an extension  $(ext^M f) \circ dist^M : M(a) \times M(b) \rightarrow M(c)$  that can be composed with a pair of functions in the monad.

Generally, there is no unique way to form a product distribution function. We require only a single coherence property of such a function, namely that

$$dist^M \circ (unit_a^M \times unit_b^M) = unit_{a \times b}^M \tag{12}$$

## 8.1 Monad declarations in ADL

Monads can be declared in a declaration format resembling an algebra specification for the monad algebra,

```

monad { name [(type expr)] (type id) = type expr;
          $Unit := expression,
          $Mult := expression }

```

The square brackets are meta-syntax to indicate that the first instance of *type expr*. is optional, depending upon the particular monad. A monad declaration is valid iff the *type expr* to the right of the equals contains only positive occurrences of the *type id* and the monad equations are satisfied. An ADL translator can check the first of these conditions but will not always be able to verify the equations automatically.

## 8.2 Some useful monads

There are several structures that will be recognized as features of programming languages and which correspond to monads.

## Exceptions

```

monad {  $Ex_i(\alpha) = \mathbf{free}\{\$Just\ of\ \alpha,\ \$exc_i\};$ 
           $\$Unit := \lambda x\ Just(x),$ 
           $\$Mult := \lambda t\ \mathbf{case}\ t\ \mathbf{is}$ 
               $Just(x) => x$ 
               $| i => \$exc_i$ 
          end }

```

where  $i$  ranges over identifiers, excluding “*Just*”.

The keyword **free** is not a proper sort, but designates the carrier of the free algebra of the bracketed signature it precedes. This declaration defines an indexed family of monads that corresponds to a family of exceptions, indexed by identifiers.

For example, the type expression  $Ex_{Nothing}(term(int))$  expresses a type whose proper values are in the datatype  $term(int)$  and whose improper value is the identifier *Nothing*, an exception name. Since the type constructor of this particular monad has structure similar to that of an inductive signature, values in the monad can be analyzed by a **case** expression.

A function  $f : a \rightarrow b$  that has been defined without thought of exceptions is “lifted” into a monad  $Ex_i$  by its map function,  $map\_Ex_i\ f$ . The lifted function, which is proper for the monad, propagates the exception  $i$  but neither raises this exception nor handles it. In ADL we designate a proper function of a given monad by the use of heavy brackets,  $\llbracket f \rrbracket$ .

A distribution function for the monad of exceptions that evaluates a pair from left to right is:

```

 $dist^{Ex_i}(x, y) = \mathbf{case}\ x\ \mathbf{of}$ 
     $Just(x') => \mathbf{case}\ y\ \mathbf{of}$ 
         $Just(y') => Just(x', y')$ 
         $| i => i$ 
    end
     $| i => i$ 
end

```

Alternatively, one could define a distribution function that would evaluate pairs from right to left.

**State transformers** The monad of state transformers affords a generic, functional specification of the use of state in computing. State can be of any type and the operations on a state component are not specified in the monad.

```

monad {  $St[\beta](\alpha) = \beta \rightarrow \alpha \times \beta;$ 
           $\$Unit := \lambda a\ \lambda b\ (a, b),$ 
           $\$Mult := \lambda t\ \lambda b\ \mathbf{let}\ (s, b') = t\ b\ \mathbf{in}\ s\ b' \}$ 

```

The product distribution function specifies how a state component is threaded through the computation of a pair. Here is a left-to-right product distribution function:

$$\begin{aligned} dist^{St} = \lambda(s_1, s_2) \lambda b \mathbf{let} \ (a_1, b') = s_1 b \ \mathbf{in} \\ \quad \mathbf{let} \ (a_2, b'') = s_2 b' \ \mathbf{in} \\ \quad ((a_1, a_2), b'') \end{aligned}$$

**State readers** An important special case of state transformers occurs when a computation does not change the state. For such a case, we can use a simpler monad, the monad of state readers.

$$\mathbf{monad} \ \{ Sr[\beta](\alpha) = \beta \rightarrow \alpha; \\ \quad \$Unit := \lambda a \ \lambda b \ a, \\ \quad \$Mult := \lambda t \ \lambda b \ t \ b \}$$

The product distribution function for state readers is unbiased as to order of evaluation of the components of a pair.

$$dist^{Sr} := \lambda(s_1, s_2) \lambda b \ (s_1 b, s_2 b)$$

**The continuation-passing monad** The well-known CPS transformation used in compiler design is another instance of a familiar monad.

$$\mathbf{monad} \ \{ CPS(\alpha) = (\alpha \rightarrow \beta) \rightarrow \beta; \\ \quad \$Unit := \lambda a \ \lambda c \ c \ a, \\ \quad \$Mult := \lambda t \ \lambda c \ t \ (\lambda s \ s \ c) \}$$

in which  $\beta$  is a free variable ranging over types.

The CPS monad can be given a left-to-right product distribution function:

$$dist^{CPS} := \lambda(t_1, t_2) \ \lambda c \ t_1 \ (\lambda x \ t_2 \ (\lambda y \ c \ (x, y)))$$

It could also be given a right-to-left product distribution, but this is not usually done. The choice is completely arbitrary.

### 8.3 Interpreting an algebra in a monad

When the carrier of an algebra has the structure of a monad, we say that the algebra is interpreted in the monad. This allows us to specify functions that carry the monad operations “for free”. For instance, if a *Nat*-algebra is interpreted in a monad  $M(a)$ , and  $s : a \rightarrow a$ , we can make the binding  $\$Succ := \llbracket s \rrbracket$  to designate  $map^M s : M(a) \rightarrow M(a)$ . If  $x : a$  we could write  $\llbracket x \rrbracket$  to designate  $unit^M x$ . Interpreting an algebra in a monad affords a notational shortcut to specifying functions that are proper for the monad.

*Example 11.* Labeling a tree

Given the signature  $Btree$  of binary trees with labeled nodes, which was specified in Example 10, give an algorithm to copy a tree, replacing the labels on its nodes by a depth-first enumeration with integers beginning with 1 at the root.

If our task was simply to copy a binary tree, the  $Btree$  algebra that induces the identity catamorphism would satisfy our requirements. This algebra is given by

$$Btree(a)\{c := btree(a); \$Emptytree := Emptytree, \$Node := Node\}$$

However, an algorithm to enumerate the nodes of a binary tree must carry a count of the number of nodes already enumerated, as it traverses the tree. This count can be provided as an integer-typed state component in a  $Btree$ -catamorphism. To incorporate a state component, we shall re-interpret the identity algebra in a state-transformer monad, and modify it so that the label on each node of a tree is replaced by an integer value calculated from the state. The carrier of the algebra becomes  $St[int](btree(int))$ . The operator  $\$Emptytree$  will be bound to the injection of a data constructor,  $Emptytree$ , into the monad algebra by an application of  $unit\_St$ . We designate this standard operation by enclosing the data constructor in fat brackets.

The reinterpretation of  $\$Node$  requires slightly more thought. Of its three arguments, the first and third will be state transformers and hence must be applied to state components to produce state-value pairs. To achieve a left-to-right enumeration, the subtree arguments of  $\$Node(l, x, r)$  should act on a state component  $n$  as

$$\begin{aligned} &\mathbf{let} (l', n_1) = l\ n \ \mathbf{in} \\ &\mathbf{let} (r', n_2) = r\ n_1 \ \mathbf{in} \\ &((l', r'), n_2) \end{aligned}$$

which is just the meaning of the expression  $dist\_St(l, r)\ n$ .

The label in  $Node(l, x, r)$  is to be replaced by the current value of the enumeration count and the state passed on must be incremented by one. With this in mind, the desired binding for  $\$Node$  is

$$F = \lambda(l, x, r)\ \lambda n\ \mathbf{let} ((l', r'), n') = dist\_St(l, r)\ (n + 1) \\ \mathbf{in} (Node(l', n, r'), n')$$

and the enumeration function, specified in ADL, is

$$\lambda t\ fst(red[btree]\ Btree(int)\{c := St[int](btree(int)); \\ \$Emptytree := \llbracket Emptytree \rrbracket, \\ \$Node := F\} t\ 1)$$

□

*Exercise 14.* Breaking lines of text

Given a list of character strings representing individual words, form a list of strings representing lines of text with a length bound  $L$  given as a parameter. Fit as many words onto a line as it will contain without overflow. Separate adjacent words on a line by blank spaces counting one character. If a word is encountered whose length exceeds the bound, return an exception named *long\_word*.

*Exercise 15.* Justifying lines of text

Extend the solution of Exercise 11 to justify text on both right and left margins by inserting additional blanks between words on a line to secure spacing as nearly even as possible on each line. If only one word fits on a line, left justify it.

## 9 Algebras of lambda terms

As an example of the style of high-level programming with algebras, we shall specify an interpreter for a lambda calculus. The strategy will be to first interpret the abstract syntax of lambda terms in a related abstract syntax for deBruijn terms, then to define substitution on deBruijn terms and produce an interpretation in which the *app* operator realizes  $\beta$ -reduction.

To begin, we shall need to declare algebra classes for lambda terms and deBruijn terms.

```
signature Term(a,b){type c;
    term/c = {$Var of b, $Abs of a*c, $App of c*c}}

signature Term'(a){type d;
    term'/d = {$Var' of a, $Abs' of d, $App' of d*d}}
```

The first translation is a catamorphism of *term*-sorted terms. An environment is needed to keep track of the bindings of identifiers. The interpretation of a term is then a function from environment to deBruijn term. We observe that this is the type of a state reader monad, which allows us to refer to the interpretation of the data constructor *App'* as it is lifted into the monad as a proper morphism.

```
dB = red[term] Term(string){c := Sr[list(string)](term'(int));
    $Var := \x \e Var'(depth x e),
    $Abs := \(\x,t) \e Abs'(t(Cons(x,e)))
    $App := [|App'|]}
```

where the function *depth* counts the number of elements in the environment that precede *x*. In ADL, a function such as *depth* is not specified with recursion, but as a morphism of an appropriate algebra. For search of a list, we prefer not to use induction on the *list*-algebra itself, for that would traverse the entire list even if a successful outcome of the search had already been found. Instead, we construct a *nat*-morphism with the *hom* combinator. This allows the specification of a partition function that terminates the search when a matching element is found.

```
depth x = hom[nat] Nat{c := int;
    $Zero := 0,
    $Succ := add 1}
(\xs case xs is
    Nil => $Zero
    | Cons(x',y) => if x=x' then $Zero
                    else $Succ y
end)
```

Next, we define substitution in deBruijn terms. The function *subst* takes three arguments: the term to be substituted, the term into which the substitution is to be made, and the binding height (number of surrounding abstraction constructors) of the second term. We shall formulate *subst N* as a *term'*-algebra morphism into an algebra whose carrier is an *int* state reader monad. This allows us to refer to the interpretation of data constructors in the monad.

```
subst N = red[term'] Term' {d := Sr[int](term'(int));
  $Var' := \n \ct if n < ct then Var'(n)
          else if n = ct
          then lift N ct 0
          else Var'(n-1),
  $Abs' := [|Abs'|] ooo add 1,
  $App' := [|App'|] o dist_Sr }
```

```
dist_Sr = \ (f,g) \s (f s, g s)
```

In the definition above, we have made use of a convenient notational extension that extends function composition to curried functions. The meaning of  $\underbrace{\circ \dots \circ}_n$  is  $\circ^n$ , where  $n \geq 1$  and

$$\begin{aligned} f \circ^1 g &= f \circ g \\ f \circ^{n+1} g &= \lambda x f x \circ^n g \end{aligned}$$

Thus  $f \circ \circ g = \lambda x \lambda y f x (g y)$  and  $f \circ \circ \circ g = \lambda x \lambda y \lambda z f x y (g z)$ .

The subordinate function *lift* has the task of modifying the deBruijn indices within the substituted term to compensate for the binding height of the context into which each substitution is made. Indices within *N* that correspond to local bindings are unchanged. Indices that correspond to bindings in the term's environment must be incremented by the binding height of the context. Thus there are two integer state components given as arguments of *lift*. The first represents the binding height of the context into which a substitution is made, and the second represents the height of local bindings within a subterm of *N*.

```
lift = red[term'] Term' {d := Sr[int](Sr[int](term'(int)));
  $Var' := \n \ct \m if n < m then Var'(n)
          else Var'(n+ct),
  $Abs' := [|Abs'|] ooo add 1,
  $App' := [|App'|] o dist_Sr_Sr }
```

Contraction of a deBruijn term is accomplished by the following function:

```
contract = red[term'] Term' {d := term'(int);
  $Var' := Var'
  $Abs' := Abs'
  $App' := \ (M,N) case M is
            Var'(_) => App'(M,N)
```

```

| Abs'(y) => subst N y 0
| App'(_,_) => App'(M,N)}

```

The function *contract* performs a single contraction step but does not generally produce a term in weak head normal form, because the substitution of an abstraction for a variable in a term may produce new redexes. To weak-head-normalize a deBruijn term, the contraction step must be iterated along the normal-order spine of a term [Pey87]. An iterative control construct could be programmed in the coalgebraic part of ADL, although that topic is outside the scope of this chapter. However, we note that to verify that an iteration was well defined in ADL, one would be required to furnish a proof of its termination. For the example at hand, no such proof is possible, for termination of the iteration would imply normalizability of terms of an untyped lambda calculus, which is impossible in general.

## 10 Further work

In this chapter we have tried to convince the reader that there are compelling advantages to using the algebraic structure of functional programs in an explicit way. The advantages include better understanding of a program's control structure and the availability of explicit, detailed induction rules for reasoning about each structure that is used. Monads integrate naturally with algebras, providing explicit structure to the carrier type.

We have told only half the story, however. In the other half, there is an interesting type system for algebraic programs. Algebras also provide units of modularity for composing programs at a larger granularity. And there are dual structures, the so-called coalgebras, that lead to equally interesting families of control structures, related more naturally to iteration than to recursion.

Another untold benefit of algebraic program construction is the opportunity it presents to use powerful program transformation techniques. When the underlying control structure is manifest, less costly program analysis is needed to determine when transformation strategies may apply. Many transformations are in fact, simply directed instances of theorems provable by parametricity [SF93]. There are many opportunities that have not yet been exploited.

## Acknowledgements

We would like to thank our colleagues, Jeff Bell, Jim Hook, Tim Sheard and Lisa Walton who have served as guinea-pigs in programming experiments with ADL and have delivered valuable criticism and valued encouragement. We have also benefited from stimulating discussions with Erik Meijer and John Launchbury.

## A Syntax of ADL

The syntax  $foo^{\{,\}}^*$  means  $foo$  may be repeated zero or more times, with a  $,$  in between each instance. The  $^+$  is similar to  $^*$ , but the item may be repeated *one* or more times, and  $^{++}$  indicates *two* or more times. The angle brackets  $\langle \rangle$  indicate optional items.

### A.1 Base syntax

The non-literal terminals are  $id$  (identifiers) and  $const$  (special constants such as integers and strings).

```
decl ::= signature id  $\langle (id^{\{,\}})^+ \rangle$  { tvars ; sortsig+ }
      cosignature id  $\langle (id^{\{,\}})^+ \rangle$  { tvars ; cosortsig+ }
      val valbind
      prefix int id+
      infix left int id+
      infix right int id+

tvars ::= type id\{,\}+

sortsig ::= idsort / idcarrier = { opdecl\{,\}+ }

opdecl ::= id
           id of type

cosortsig ::= idsort / idcarrier = { coopdecl\{,\}+ }

coopdecl ::= id : type

type ::= id  $\langle (type^{\{,\}})^+ \rangle$ 
        type\{*\}++
        type1 -> type2
        ( type )

valbind ::= pat = expr

pat ::= apat
        id as pat
```

```

apat ::= _
        id
        ( pat{,}*  )

rulepat ::= _
            id
            const
            id pat
            id as pat
            ( pat{,}*  )

expr ::= aexpr
         appl
         \ apat expr
         let valbind in expr

appl ::= expr expr
         expr id expr

aexpr ::= id
         $id
         const
         map[id]
         red[id] algebra
         hom[id] algebra
         gen[id] algebra
         cohom[id] algebra
         case expr of rule{1}+ end
         ( expr{,}*  )

rule ::= rulepat => expr

algebra ::=  $\langle id \langle (id_{typaram}^{\{, \}+}) \rangle \rangle \{ tybind ; opbind^{\{, \}+} \}$ 
            $\langle id \langle (id_{typaram}^{\{, \}+}) \rangle \rangle \{ tybind^{\{, \}+} ; subalg^+ \}$ 

tybind ::= id = type

subalg ::= id { opbind{,}+ }

opbind ::= id := expr

```

Notes:

- In *type*, **\*** binds more tightly than  $\rightarrow$ , and  $\rightarrow$  associates to the right.
- In *sortsig*,  $id_{carrier}$  must be declared in *tvars*, and each  $id_{carrier}$  may only appear in one *sortsig*.
- The ambiguity of application in patterns and expressions is resolved by a precedence parser. Each *id* has an associated fixity (prefix or infix), precedence and, for infix, associativity. Precedence is a positive integer, with higher value indicating higher precedence (binds more tightly). These are assigned by a **prefix** or **infix** declaration. The default is prefix with precedence 9.
- No *id* may appear twice in a *pat* and no *id* may be that of a free algebra operator. In a *rulepat*, no *id* may appear twice, unless it is a free algebra operator.
- In an *algebra*, the *ids* bound in *tybinds* must correspond to the *tvars* declared in the algebra signature.

## A.2 Derived forms

$$\begin{aligned} id\ apat_1 \dots apat_n = expr &\quad \Rightarrow id = \backslash apat_1 \dots \backslash apat_n\ expr \\ \mathbf{let}\ valbind^{\{\text{and}\}++}\ \mathbf{in}\ expr &\quad \Rightarrow \mathbf{let}\ valbind^{\{\text{in let}\}++}\ \mathbf{in}\ expr \\ \mathbf{if}\ expr_1\ \mathbf{then}\ expr_2\ \mathbf{else}\ expr_3 &\quad \Rightarrow \mathbf{case}\ expr_1\ \mathbf{of}\ \mathbf{true}\ \Rightarrow\ expr_2\ |\ \mathbf{false}\ \Rightarrow\ expr_3 \end{aligned}$$

## References

- [B<sup>+</sup>94] Jeffrey Bell et al. Software design for reliability and reuse: A proof-of-concept demonstration. In *TRI-Ada '94 Proceedings*, pages 396–404. ACM, November 1994.
- [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.
- [CS92] Robin Cockett and Dwight Spencer. Strong categorical datatypes. In R. A. G. Seely, editor, *International Meeting on Category Theory, 1991*. AMS, 1992.
- [Fok92] Maarten M. Fokkinga. *Law and Order in Algorithmics*. PhD thesis, University of Twente, Twente, The Netherlands, February 1992.
- [Gog80] Joseph A. Goguen. How to prove inductive hypotheses without induction. In W. Bibel and R. Kowalski, editors, *Proc. 5th Conference on Automated Deduction*, volume 87 of *Lecture Notes in Computer Science*, pages 356–373. Springer-Verlag, 1980.
- [GT79] Joseph A. Goguen and Joseph Tardo. An introduction to )BJ: A language for writing and testing software specifications. In *Proc. of Conference on Specification of Reliable Software*, pages 170–189. IEEE Press, 1979.
- [GTW78] Joseph A. Goguen, James Thatcher, and Eric Wagner. An initial algebra approach to the specification, classification, correctness and implementation of abstract data types. In Raymond Yeh, editor, *Current Trends in Programming Methodology IV*, pages 80–149. Prentice-Hall, 1978.
- [GW88] Joseph A. Goguen and Timothy Winkler. Introducing OBJ3. Technical Report SRI-CSL-88-9, SRI International, August 1988.
- [Hag87] T. Hagino. *A Categorical Programming Language*. PhD thesis, University of Edinburgh, 1987.
- [KB<sup>+</sup>95] Richard B. Kieburtz, Françoise Bellegarde, Jef Bell James Hook, Jeffrey Lewis, Dina Oliva, Tim Sheard Lisa Walton, and Tong Zhou. Calculating software generators from solution specifications. In *TAPSOFT'95*, volume 915 of *Lecture Notes in Computer Science*, pages 546–560. Springer-Verlag, 1995.
- [KL94] Richard B. Kieburtz and Jeffrey Lewis. Algebraic Design Language—Preliminary definition. Technical report, Pacific Software Research Center, Oregon Graduate Institute of Science & Technology, January 1994.
- [Mac71] Saunders MacLane. *Categories for the Working Mathematician*. Springer-Verlag, 1971.
- [Mal90] Grant Malcolm. *Algebraic Data Types and Program Transformation*. PhD thesis, University of Groningen, 1990.
- [Mee86] Lambert Meertens. Algorithmics—towards programming as a mathematical activity. In *Proc. of the CWI Symposium on Mathematics and Computer Science*, pages 289–334. North-Holland, 1986.
- [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.
- [Pey87] Simon Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.
- [SF93] Tim Sheard and Leonidas Fegaras. A fold for all seasons. In *Proceedings of the conference on Functional Programming and Computer Architecture*, Copenhagen, June 1993.
- [Spi90] Mike Spivey. A functional theory of exceptions. *Science of Computer Programming*, 14:25–42, 1990.
- [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*, pages 1–14. ACM Press, January 1992.