Abstract

This report constitutes a preliminary definition of a new, high-level programming language called ADL. It uses the mathematical concept of structure algebras as its unit of modularity. When algebras are used to specify programs, control structure is fixed first and data structure, or representations, second. There is no explicit recursion or iteration construct in ADL. Control is determined by combinators applied to inductively defined algebras. An intended use of ADL is to provide computational semantics of specialized software design languages.

An algebra in ADL can be interpreted in various monads, a particular variety of algebras that has been found useful in programming. ADL also makes use of coalgebras, a concept dual to that of algebras. With coalgebras, iterative control structures typical of search algorithms can be specified.

There is a strong notion of type in ADL, guaranteeing that all well-typed programs terminate. This allows us to use sets as ADL's semantic domain and to provide ADL with an equational logic. However, to check the type correctness of an expression, there can be proof obligations that cannot be discharged mechanically. A benefit of the equational logic is that an ADL program is amenable to transformation based upon the equational theories of its algebras. Transformations are not discussed in this report, however.

Algebraic Design Language
(Preliminary definition)

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

January 14, 1994

1 Introduction

ADL—Algebraic Design Language—is a higher-order software specification language in which control is expressed through a family of type-parametric combinators, rather than through explicitly recursive function definitions. ADL is based upon the mathematical concept of structure algebras and coalgebras. The declaration of an algebraic signature specifies a variety of structure algebras[1]. A signature declaration implicitly defines the terms of a particular algebra, the free term algebra of the signature, which corresponds to a datatype in a typed, functional programming language such as ML, Haskell or Miranda.

Classes of coalgebras are declared by record signatures. The free coalgebras correspond to infinite records, and have no direct analogy in most conventional programming languages, although streams, which can be created in lazy functional languages, provide one such instance.

The functions definable in ADL are the λ-definable morphisms of such algebras and coalgebras. Properties of such functions can be proved by applying rules of inductive (or co-inductive) inference dictated by the structure of the underlying signature.

There are related studies of the use of higher-order combinators for theoretical programming [MFP91, Fok92], however, none has yet 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 datatype morphisms 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 datatypes as structure algebras (and coalgebras) [Mac71] can be attributed to Hagino.

ADL has syntax similar to that of the ML language family. Like Standard ML, it consists of a core language augmented by a module structure. ADL modules, called functors, are abstracted with respect to structure algebras or coalgebras. The functor construct in ADL

[1] A *variety* is a class of algebras that have a common signature.

indeed corresponds to the categorical notion of functor, unlike the like-named construct of Standard ML.

Unlike Standard ML, ADL has no **ref** types and has no **rec** definitions. ADL can be given a simple semantics over sets. However, domain sets are subject to logically formulated restriction, and complete type-checking of ADL programs is only semi-decidable. The semantics of ADL does not rely upon fixpoints and does not require domains of CPO's as an underlying structure, although such an interpretation is certainly possible.

The computational content of ADL can be translated via the semantic equations given in its metalanguage into a first-order, call-by-value functional language with fixpoints and exceptions that we call BDL—Basic Domain Language. BDL has no higher-order functions and no explicit abstraction (i.e. it has no **fn** expressions, as SML does). However, it does allow recursive definitions of first-order functions. BDL has a conventional, denotational semantics expressed in terms of domains. It may be thought of as the "machine language" of an abstract machine capable of evaluating ADL (and other languages as well).

2 Algebras, Types and signatures

ADL is a higher-order, typed 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 does distinguish between the names of types and the semantics of types and it contains combinators that are indexed by type names. Its type system is sufficiently rich that type-checking is not known to be 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 the ML family of 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 which satisfy the structural typing rules respect the signatures of multi-sorted algebras—integer data are never used as reals or as 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 typings.

In Standard ML and related languages, the Hindley-Milner type system is extended with datatype declarations. A datatype declaration names a type and specifies a finite set of data constructors. A datatype name may have one or more type variables as parameters, and thus actually names a type former. When a type variable is introduced as a parameter in a datatype declaration, the variable is bound by abstraction, rather than universally quantified. The binding occurrence of an abstracted type variable is its occurrence in the left hand side of a datatype definition. Application of a type former to a type expression can be understood syntactically, as the substitution of the argument expression for all occurrences of the type variable in the datatype declaration.

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

Parameterized datatypes	**Varieties**
type	algebra
type former	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 $* \to *$, $(*, *) \to *$, $(*, *, *) \to *, \ldots$ is said to be *unsaturated*. An unsaturated sort, s, can form a saturated sort expression by applying it to a tuple that consists of as many saturated sort expressions as there are asterisks to the left of the arrow in the arity of s. A type name in ADL is a saturated sort expression. Type variables range over saturated sort expressions.

ADL departs significantly from functional programming languages such as SML by providing declarations of signatures that define classes of structure algebras, not simply datatypes. An algebraic signature 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 each particular algebra.

2.1 Some familiar algebras

Where we would write the declaration of a *list* datatype in Standard ML as

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

a corresponding declaration of a family of *list*-algebras in ADL is written as:

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

This declares *List*{} to be the name of a class of algebras for which there is defined a single unsaturated sort, $\mathit{list} : * \to *$. The *list*-sorted algebras have a signature parametric on a type represented by the variable a. In the signature, the type variable c is used as the name of the carrier type. The signature consists of a pair of operator names, with typing

$$\text{type } a,\, c \;\vdash\; \begin{array}{l} \$nil : c \\ \$cons : a \times c \to c \end{array}$$

Properly, the operator *\$nil* could have been given a function type, $\mathbf{1} \to c$, by declaring it as "*\$nil* **of 1**". Since $\mathbf{1}$ is a singleton set, every function in the type $\mathbf{1} \to c$ is isomorphic to an element in c.

Operator names always begin with '\$' to distinguish them from other identifiers. A concrete algebra is specified by a structure that contains bindings for the carrier type and for each operator of the algebra.

Each signature declaration implicitly defines one specific datatype. This is the type of *free terms*, whose operators are the free constructors of the signature (just as for SML datatypes) and whose elements are the terms constructed by well-typed applications of these constructors. The names of the data constructors of the datatype of free terms are derived from the names of operators in the signature by dropping the initial '\$' symbol. As a convention, we shall also capitalize the initial letter of the name of a data constructor.

2.2 Structure algebras

Definition 2.1 : Let T be a parameterized signature. 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.

□

An important special case of a T-algebra occurs when the elements of the signature are data constructors. Data constructors are unconstrained by equational laws. The set of terms generated from values of a type a by well-typed applications of the data constructors of T constitutes a type that we call $T(a)$. Under suitable constraints on the signature T, a type $T(a)$ is the carrier of a T-algebra that is unique modulo the isomorphism class of the parameter, a. This is called the *free term algebra.*

A *T-algebra morphism* is a function that maps one T-algebra to another. This notion can be made precise, but we need some notation to express it. The meaning of T as a parameterized signature can be extended to define a signature as the mapping of a function. The following definitions have been specialized to the case of a single-sorted signature.

Definition 2.2 : Let $f : a \to b$ and let t be the (single) sort and $\kappa_1, \ldots, \kappa_n$ be the operator names of the signature T. then $map\_t\, f : t(a) \to t(b)$ is defined as follows:

$$
\begin{aligned}
map\_t\, f = \lambda x.\ & \textbf{case } x \textbf{ of} \\
& \quad \vdots \\
& \kappa_i(x_1, \ldots, x_{m_i}) \Rightarrow \kappa_i(y_1, \ldots, y_{m_i})
\end{aligned}
$$

$$
\text{where } y_j = \begin{cases} f\, x_j & \text{if } x_j : a \\ map\_t\, f\, x_j & \text{if } x_j : t(a) \\ map\_s\,(map\_t\, f)\, x_j & \text{if } x_j : s(t(a)) \text{ where } s \text{ is a 1-unsaturated} \\ & \qquad\qquad\quad \text{sort of a signature } T' \\ x_j & \text{if } x_j : s' \text{ where } s' \text{ is a saturated sort} \end{cases}
$$

Definition 2.3 : Let t be the (single) sort of a signature T. A *T-algebra morphism* $f : a \to b$ satisfies the commuting diagram below:

$$
\begin{tikzcd}
t(a) \arrow[r, "map\_t\, f"] \arrow[d, "h"'] & t(b) \arrow[d, "k"] \\
a \arrow[r, "f"] & b
\end{tikzcd}
$$

Example 2.1 : A *list*-algebra morphism. Let $exp2 = \lambda n.\, 2^n$, and let *sum* and *product* be the functions that reduce a list of non-negative integers by addition and multiplication, respectively. Then the following diagram illustrates *exp2* as a morphism of *list*-algebras:

$$
\begin{tikzcd}
list(int) \arrow[r, "map^{list}\, exp2"] \arrow[d, "sum"'] & list(int) \arrow[d, "product"] \\
int \arrow[r, "exp2"'] & int
\end{tikzcd}
$$

□

When a signature declaration satisfies the unitary condition (see Definition 2.6), there exists for each type a, a free term algebra, $(t(a), \mu_a)$. (This condition is sufficient but not necessary.) It has the property that for any T-algebra (a, h), the homomorphism h is the unique T-algebra morphism from the initial term algebra to (a, h). The last statement is summarized in the following commuting square:

$$\begin{tikzcd} t^2(a) \arrow[r, dotted, "map\_th"] \arrow[d, "\mu_a"'] & t(a) \arrow[d, "h"] \\ t(a) \arrow[r, dotted, "h"] & a \end{tikzcd}$$

The dotted arrow indicates that the function that makes the diagram commute is uniquely determined from the other data in the diagram.

Definition 2.4 Let $t : * \to *$ be an unsaturated sort of a signature T. An element "κ **of** σ" of the $t(\alpha)$ component of T is called a *unit operator* if $\sigma = \sigma_1 * \cdots * \sigma_m$ and there is at least one occurrence of α among the σ_i. If $\sigma = \alpha$, then κ is said to be *perfect*.

Definition 2.5 A signature T is *zero-based* if it contains a unique element "κ_0 **of 1**".

Definition 2.6 A (single-sorted) signature T is *unitary* if it contains a unique unit operator, and either

1. the unit operator is perfect, or
2. if the unit operator is given by a signature component "κ of $\sigma_1 * \cdots * \sigma_m$" then
   - only one factor, σ_i, is α,
   - for all factors $\sigma_j \neq \alpha \Rightarrow \sigma_j = c$, where c is the name declared for the carrier of type t(α),
   - T is zero-based.

□

If a signature T is unitary, the datatype of free terms of T is the carrier of an algebra. This algebra is in fact, initial in the category of T-algebras. Thus a (redundant) ADL specification of the free term algebra for the sort *list* would be

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

It is important to remember the distinction between data constructors in the free term algebra and operators in the signature of an algebra. Different instances of an operator may have different types, depending upon the environment in which it appears, as the carrier type will differ in distinct algebras of the same family. The data constructors are a special case of the operators for one specific algebra, and their types are fixed, up to variation in the type argument of an unsaturated sort.

Example 2.2 : Three different *list*-algebras are:

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

These *List*-algebras induce homomorphisms from free *List*-algebras that represent functions that sum a list of integers, calculate the length of a list, and catenate two lists, respectively. A combinator to specify these homomorphisms will be introduced in the next section.

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 := (+)}
```

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

```
signature Nat{type c; nat/c = {$zero, $succ of c}}
signature Tree{type c; tree(a)/c = {$tip of a, $fork of (c * c)}}
signature Bush{type c; bush(a)/c = {$leaf of a, $branch of list(c)}}
```

Note that *nat* is a saturated sort, while *tree* and *bush* are both 1-unsaturated.

2.3 The reduce combinator

If t is a 1-unsaturated sort of the signature T, a structure function of the class of T-algebras is any function $h : t(a) \rightarrow a$. If T is unitary, and hence has a free term algebra, then h is also the unique T-algebra morphism from $(t(a), \mu_a)$ 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.

ADL defines a combinator, *red* , that takes an algebra specification to a free-algebra morphism. The *red* combinator obeys a morphism condition for each algebra on which it is instantiated. For the algebras we have considered, these equations are:

$$\begin{aligned}
red[nat]\ Nat\{c; \$zero, \$succ\}\ Zero &= \$zero \\
red[nat]\ Nat\{c; \$zero, \$succ\}\ (Succ\, n) &= \$succ\,(red[nat]\ Nat\{c; \$zero, \$succ\}\ n) \\
red[list]\ List\{c; \$nil, \$cons\}\ Nil &= \$nil \\
red[list]\ List\{c; \$nil, \$cons\}\ Cons(x, y) &= \$cons(x,\ red[list]\ List\{c; \$nil, \$cons\}\ ) \\
red[tree]\ Tree\{c; \$tip, \$fork\}\ (Tip\, x) &= \$tip\, x \\
red[tree]\ Tree\{c; \$tip, \$fork\}\ (Fork(l, r)) &= \$fork\,(red[tree]\ Tree\{c; \$tip, \$fork\}\ l, \\
&\qquad\quad red[tree]\ Tree\{c; \$tip, \$fork\}\ r) \\
red[bush]\ Bush\{c; \$leaf, \$branch\}\ (Leaf x) &= \$leaf\ x \\
red[bush]\ Bush\{c; \$leaf, \$branch\}\ (Branch\, y) &= \$branch\,(map_list\,(red[bush]\ Bush\{c; \$leaf, \$branch\})\, y)
\end{aligned}$$

The function *map_list*, referred to in the last equation above, will be defined below.

Here are some examples of *list*-algebra morphisms constructed with *red* [*list*] and the algebra specifications given in Example 2.2:

```
sum_list = red[list] List{c := int; $nil := 0, $cons := (+)}
length = red[list] List{c := int; $nil := 0, $cons := \(x,y) 1+y}
append = red[list] List{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{c := list(b);
                            $nil := Nil,
                            $cons := \(x,y) Cons(f x, y)}
```

where `f` has the type `a -> b` for some existing type `a`, and

```
flatten_list = red[list] List{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)
```

Some examples of *nat*-algebra morphisms are:

```
ntoi = red[nat] Nat{c := int; $zero := 0, $succ := \n 1+n}
add x = red[nat] Nat{c := int; $zero := x, $succ := Succ}
plus = red[nat] Nat{c := int -> int; $zero := id, $succ := \f \n 1 + f n}
```

with typings

```
ntoi : nat -> int
add : nat -> nat -> nat
plus : nat -> int -> int
```

Examples of *tree* morphisms are:

```
sum_tree = red[tree] Tree{c := int;
                          $tip := id,
                          $fork := (+)}
list_tree = red[tree] Tree{c := list(a);
                           $tip := \x Cons(x,Nil),
                           $fork := \(x,y) append x y}
map_tree f = red[tree] Tree{c := tree(a);
                            $tip := \x Tip(f x),
                            $fork := Fork}
flatten_tree = red[tree] Tree{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)
```

The analogous examples of *bush* morphisms are:

```
sum_bush = red[bush] Bush{c := int;
                          $leaf := id,
                          $branch := sum_list}
list_bush = red[bush] Bush{c := list(a);
                           $leaf := \x Cons(x,Nil),
                           $branch := flatten_list}
```

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

flatten_bush = red[bush] Bush{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 2.1 Reverse of a list

a. Specify a *list*-reduce to compute the reverse of a list.

b. Now specify a second *list* reduce with carrier type $list(\alpha) \rightarrow list(\alpha)$ to define a function $rev : list(\alpha) \rightarrow list(\alpha) \rightarrow list(\alpha)$ that satisfies the equation

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

2.4 Primitive recursion

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

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

where $g : t_1 \times \ldots \times t_n \rightarrow a$ and $h : nat \times a \times t_1 \times \ldots \times t_n \rightarrow a$. Although the primitive recursion scheme can be represented as a *nat*-reduce, 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*-reduce is

$$\begin{array}{lcl} \textbf{case } x \textbf{ of} & = & \mathit{red}[\mathit{nat}]\,\mathit{Nat}\{c; \\ \quad \mathit{Zero} \Rightarrow g & & \qquad \mathit{\$zero} := (\mathit{Zero},\, g), \\ \mid \quad \mathit{Succ}(x') \Rightarrow h\ x' & & \qquad \mathit{\$succ} := \lambda(x,y)\,(\mathit{Succ}\,x,\ h\ x)\} \\ \textbf{end} & & \end{array}$$

Evaluation of the *nat*-reduce 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.

A primitive recursive function is, however, a structure function of a related variety of structure algebras, one in which the carrier always has the form $nat * a$ for some type a. This motivates us to define an operator on signatures, with which to obtain new families of structure algebras whose homomorphisms are primitive recursive.

Let `delta` be the operator that takes a signature for sort t to a derived signature that has the same set of operator names, but in which every occurrence of the carrier type, c, in the typing of an operator is replaced by $t' * c$ (where t' designates a saturated instance of the sort t). Thus, for example,

$$\mathit{delta\ nat} = \{\text{type } c;\ \ \mathit{\$zero},\ \ \mathit{\$succ}\,\text{of}\ \mathit{nat} * c\}$$

We can name this signature in a declaration:

```
signature PR_nat = delta nat;
```

and use `PR_nat` in the definition of a primitive recursive reduce. In general, for a signature T with a single sort, t, the reduce of Pr_T satisfies the equations (for each free constructor, K_i):

$$\begin{array}{l} \mathit{red}[t]\ \mathit{Pr\_T}\{c;\ \ \$\kappa_1,\ldots,\$\kappa_n\}\ K_i(x_1,\ldots,x_{m_i}) \\ \qquad = \$\kappa_i(y_1,\ldots,y_{m_i}) \\ \qquad \text{where } y_j = \begin{cases} (x_j,\, \mathit{red}[t]\ \mathit{Pr\_T}\{c;\ \$\kappa_1,\ldots,\$\kappa_n\}\ x_j) & \text{if } \sigma_{ij} = c \\ (x_j,\, \mathit{map\_s}\,(\mathit{red}[t]\ \mathit{Pr\_T}\{c;\ \$\kappa_1,\ldots,\$\kappa_n\})\ x_j) & \text{if } \sigma_{ij} = s(c) \\ x_j & \text{otherwise} \end{cases} \end{array}$$

To define a factorial function, for instance, one could write

```
fact = red[nat] PR_nat{c := int;
                       zero := 1,
                       succ := \(m,n) ntoi(Succ m) * n}
```

To define a general primitive recursion scheme for natural numbers, declare a higher-order structure functor, *Pr*, by

```
type a;
Pr(g,h) = red[nat] PR_nat{c := a;
                          $zero := g,
                          $succ := h}
```

This defines a family of *PR_nat* algebras, with structure functions $Pr(g, h) : nat \rightarrow a$, for each pair $(g : a,\ h : nat * a \rightarrow a)$. In terms of this scheme, the factorial function is defined by

```
fact = Pr(1,\(m,n) ntoi(Succ m) * n)
```

where the type variable a has been instantiated to *int*.

Exercise 2.2 Splitting a list

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

$splitat\, c\, xs$ 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*.

2.5 Proof rules for algebras

Inference rules for the particular algebras introduced in the previous section are summarized below. The rule for the *Nat*-algebra is natural induction, as one would expect. For the *List*, *Tree* and *Bush* algebras, the rules are those of "structural induction" for the datatypes that correspond to the free algebras. Note that we do not have to treat induction as a special rule of the logic—the inductive proof rules account for the computational content of the algebra morphisms. This has been noted previously by Goguen [Gog80] and others.

$$\frac{c \text{ type} \qquad P(\$zero) \qquad P(n) \Rightarrow P(\$succ\, n)}{\forall n : nat.\ P(red[nat]\ Nat\{c, \$zero, \$succ\}\ n)}$$

$$\frac{c \text{ type} \qquad P(\$nil) \qquad P(y) \Rightarrow P(\$cons(x, y))}{\forall y' : list(a).\ P(red[list]\ List\{c, \$nil, \$cons\}\ y')}$$

$$\frac{c \text{ type} \qquad P(\$tip\, x) \qquad P(y) \wedge P(z) \Rightarrow P(\$fork(y, z))}{\forall y' : tree(a).\ P(red[tree]\ Tree\{c, \$tip, \$fork\}\ y')}$$

$$\frac{c \text{ type} \qquad P(\$leaf\, x) \qquad \forall y : c.\ \forall ys : list(c).\ y \ \mathbf{in}\ ys \Rightarrow P(y) \Rightarrow P(\$branch\, ys)}{\forall z : \$branch(a).\ P(red[bush]\ Bush\{c, \$leaf, \$branch\}\ z)}$$

3 Morphisms of non-initial structure algebras

Recall the diagram in terms of which a T-algebra morphism is defined:

$$\begin{tikzcd}
t(a) \arrow[r, "map_t\, f"] \arrow[d, "h"'] & t(b) \arrow[d, "k"] \\
a \arrow[r, "f"] & b
\end{tikzcd}$$

The initial algebra homomorphisms illustrated in the diagram are h, k, $map\_t\, f$ and the composite, $f \circ h = k \circ map\_t\, f$. Each of these can be expressed in terms of the combinator *red* and the appropriate T-algebra. However, f is also a T-algebra morphism, and under certain conditions, it may also be expressed in terms of a combinator. Suppose there exists a function $p : a \rightarrow t(a)$ such that $p \circ h = id_{t(a)}$. (However, p is not necessarily a right inverse for h.) Then f must satisfy the recursion equation

$$f = k \circ map\_t\, f \circ p$$

when its domain is restricted to the image of $t(a)$ under h.

Let $E\$t(a)$ designate a type isomomorphic to $t(a)$. Typically, it will be a disjoint union of alternatives including a, $t(a)$, the "unit" type, $\mathbf{1}$, and products of these. It represents an explicit one-level unfolding of the structure of terms of type $t(a)$. Then a function $p' : a \to E\$t(a)$ may be isomorphic to a left inverse for h as described in the preceding paragraph. With this nomenclature, an isomorphic relative of the recursion equation given above can be summarized in the diagram below, which reveals the structure more clearly:

$$\begin{tikzcd} E\$t(a) \arrow[d, "map\_E\$t\, f"'] & a \arrow[l, "p'"'] \arrow[d, "f"] \\ E\$t(b) \arrow[r, "k"'] & b \end{tikzcd}$$

Following the suggestion outlined above, ADL introduces a combinator with which to construct morphisms whose domains are T-algebras that are not initial. We call this combinator *hom* . It takes three parameters; the sort of the structure function that is to be mapped, the structure algebra in the codomain of the morphism and a partition relation that is the "inverse" 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 $E\$t(a)$, which is a disjoint union of the domain types of the set of operators of the signature T.

Thus we write $\mathrm{hom}[t]\, T\{b;\, k\}\; p$, where $k : t(b) \to b$ and $p : a \to t(a)$. Here is an example that illustrates the construction of a T-algebra morphism with *hom* .

Example 3.1 : Calculate the largest power of 2 that factors a given positive integer.

Consider the *Nat*-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 $\{m, 2m, 4m, 8m, \ldots\}$. To invert the structure function, construct a function that recovers the natural number giving the power of two that multiplies m in forming any element of the carrier. That is, let

```
p : int → E$nat(int)
p =def \n  if n mod 2 <> 0 then $zero
           else $succ(n div 2)
```

where *E\$nat* is a derived, unsaturated sort. This sort belongs to no declared variety, thus has no signature and cannot form the type of the domain or codomain of other, 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 is presumed to structure 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 that represents a natural number 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 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 = \n 1+n} p
```

The equation satisfied by *pwr_2* is:

$$\begin{aligned} \mathit{pwr\_2}\; n = {} & \textbf{if } n \bmod 2 \neq 0 \textbf{ then } 0 \\ & \textbf{else } 1 + \mathit{pwr\_2}\,(n \operatorname{div} 2) \end{aligned}$$

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 = \n 2*n} p
```

Example 3.2 : $\log_2$ of a positive integer.

By modifying the algebra in the domain of the partition relation in the previous example we can obtain an algorithm for the base 2 logarithm of a positive integer. Let

$$\mathtt{p}' : \text{int} \rightarrow E\$nat(\text{int})$$
$$\mathtt{p}' =_{\text{def}} \backslash\mathtt{n}\ \ \mathtt{if\ n\ div\ 2 = 0\ then\ \$zero}$$
$$\mathtt{else\ \$succ(n\ div\ 2)}$$

```
log_2 = hom[nat] Nat{c := int; $zero = 0, $succ = \n 1+n} p'
```

Example 3.3 : 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 can be directly constructed as an instance of *red* for a suitable list algebra. However, 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{type c; slist(a)/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{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 3.4 : Quicksort

A quicksort of a list of integers 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 reduce:

```
part a = red[list] List{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, is a divide-and-conquer algorithm with the structure of a binary tree. It can be expressed as a hom of the algebraic variety:

```
signature Btree{type c; btree(a)/c = {$emptytree, $node of c*a*c}}
sort = hom[btree] Btree{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 control is a tree traversal, the sort function has type $list(int) \rightarrow list(int)$. There is no data structure corresponding to the datatype $btree(list(int))$. This is a "treeless" tree traversal.

Exercise 3.1 Another form of bush

Given **signature** $Bush'\{\text{type } c;\ bush'(a)/c = \{\$leaf' \text{ of } a,\ \$branch' \text{ of } nat * (nat \to c)\}\}$ construct a morphism of type $bush(a) \to bush'(a)$ that is invertible. (Construct its inverse, too.)

Exercise 3.2 Splitting a list more efficiently

The function *splitat* 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. Reformulate the function as a $hom[list]$.

Exercise 3.3 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:

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

Prove that your solution satisfies the equations.

3.1 Proof rules for morphisms of non-initial algebras

Properties of functions constructed with *hom* can be verified by applying the proof rules of the T-algebra, as described earlier, *provided that the construction actually is a T-algebra morphism.* Recall that for a construction $hom[t]T\{b;\ k\}p$ to be a T-algebra morphism, the partition relation p must be a left inverse of the structure function of a T-algebra, (a, 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 p is necessarily *formally* correct; it constructs results by well-typed (in the Hindley-Milner system) application of operators of the signature T. However, its application to an arbitrary element $x : a$ 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 provided to discharge the proof obligation.

Definition 3.1 Let T be a signature declared by

$$\textbf{signature } T = \{\textbf{type } c;\ s(a)/c = \{\ldots \$\kappa_i \textbf{ of } t_{i1} \times \cdots \times t_{im_i} \ \ldots\}\}$$

Let P be a predicate over a. Suppose that $(\prec) \subseteq a \times a$ is a well-founded ordering on the set $\{x : a \mid P(x)\}$. We say that a function $p : a \to s(a)$ calculates a *T-inductive partition* of the set $\{x : a \mid P(x)\}$ if

$$\forall x : a.\ P(x) \Rightarrow \forall \$\kappa_i \in T.\, p\, x = \$\kappa_i(y_1, \ldots, y_{m_i}) \Rightarrow \forall j \in 1..m_i \begin{cases} y_j \prec x & \text{if } t_{ij} = c \\ \forall z.\ z \ \mathit{elt\_s'}\ y_j \Rightarrow z \prec x & \text{if } t_{ij} = s'(c) \end{cases}$$

where s' is a 1-unsaturated sort ($s' \neq s$) and $\mathit{elt\_s'}$ is an infix notation for the two-place predicate defined by:

$$\begin{array}{rcl} z = x & \Rightarrow & z\ \mathit{elt\_s'}\ \$\kappa'_i(y_1, \ldots, x, \ldots, y_{m_i}) \\ z\ \mathit{elt\_s'}\ y & \Rightarrow & z\ \mathit{elt\_s'}\ \$\kappa'_i(y_1, \ldots, y, \ldots, y_{m_i}) \end{array}$$

for all operators $\$\kappa'_i$ in the signature of sort s'.

□

In the definition above, the predicate P characterizes a subset of type a elements 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. In Example 3.1, a suitable subset and its well-ordering is the natural order, $(<)$, on positive integers. The partition relation p induces a *nat*-inductive partition. In Example 3.2, the same ordering is used but the set is the non-negative integers. In Examples 3.3 and 3.4, a suitable ordering on $\mathit{list}(\mathit{int})$ is $xs \prec ys$ iff $\mathit{length}\ xs < \mathit{length}\ ys$. The verification condition for the function *part* of the *Quicksort* example becomes

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

Definition 3.1 of T-inductive partition of a set extends without complication to algebras of a multi-sorted signature. What becomes more complicated in such a case is the well-founded order, which may need to relate terms of different sorts.

4 The ADL type system

Logical properties of morphisms of the structure algebras associated with datatypes can be derived by inductive proof rules. Each such property is formalized as a predicate over a set. ADL types can be interpreted as sets, although as we shall see later, when the *hom* combinator is introduced, proof obligations arise in verifying that a syntactically legal term is semantically valid with respect to the ADL type system.

Since types are sets, the restriction of a type by a predicate defines a set that may be considered as 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, $\{x : t \mid P(x)\}$, where t 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 $f : \{x : t_1 \mid P(x)\} \to t_2$ and $g : \{x : t_2 \mid Q(x)\} \to t_3$, a structural (Hindley-Milner) typing approximates the ADL typings as $f : t_1 \to t_2$ and $g : t_2 \to t_3$. It will judge their composition to be well-typed, with typing $g \circ f : t_1 \to t_3$. An ADL typing of the composition has the form $g \circ f : \{x : t_1 \mid R(x)\} \to t_3$, 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 algebric 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 may include 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 $f : \{x : t_1 \mid P\,x\} \to t_2$, we can express the typing of a function that composes its argument on the left of f as

$$\lambda g.\, g \circ f : (t_2 \to t_3) \to \{x : t_1 \mid P\,x\} \to t_3$$

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. However, 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 $\lambda x.\, e : t_1 \to t_3$, the free variable f occurs in an applicative position and satisfies a structural typing $f : t_1 \to t_2$. 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 $\lambda f.\, \lambda x.\, e \;:\; (\{y : t_1 \mid R\, y\} \to t_2) \to \{x : t_1 \mid P\, x\} \to t_3$.

An application of a function $h \;:\; (\{y : t_1 \mid R\, y\} \to t_2) \to \{x : t_1 \mid P\, x\} \to t_3$ to an argument $e' \;:\; \{y : t_1 \mid Q\, y\} \to t_2$ is judged to be well-typed if there is a proof that $\forall y : t_1.\, R\, y \Rightarrow Q\, y$.

4.1 Typing combinator expressions

The function composition operator is one instance of an ADL combinator whose arguments can have domain-restricted function types. The ADL combinators *red* and *hom* are further instances, and they require special typing rules. These combinators are applied to algebra specifications, so it is necessary to specify what constitutes a well-typed algebra specification. For simplicity, we illustrate the formal rules for a single-sorted algebra A, with sort symbol s and carrier (type) symbol c. Let $\mathit{Index}(\Sigma_s)$ designate the index set of the signature of sort s. Let $t, t_1, t_2, \cdots$ range over types and $f_1, f_2, \cdots$ range over expressions. Let ρ range over typing environments. (A typing environment is a finite mapping of type variables to types.) The judgement form $\rho \vdash e : t$ is read as "expression e has type t in the typing environment ρ." The rule for well-typing of an algebra specification is:

$$\frac{\begin{array}{c}\forall i \in \mathit{Index}(\Sigma_s).\; [\alpha : \text{type}], \rho \vdash f_i \;:\; t_i \to t \\ \wedge\; t_i = \rho_c[t/c](\sigma_i)\end{array}}{[\alpha : \text{type}], \rho \vdash_{Alg} A\{c := t,\, s\{\ldots \$\kappa_i := f_i, \ldots\}\}}$$

in which ρ_c is the type environment that agrees with ρ on all type variables except c, which is not in its domain.

Well-typing of an algebra specification is a hypothesis for the typing of a reduce combinator. The rule is:

$$\frac{[\alpha : \text{type}], \rho \vdash_{Alg} A\{c := t,\, s\{\ldots \$\kappa_i := f_i, \ldots\}\}}{[\alpha : \text{type}], \rho \vdash \mathit{red}[s]\, A\{c := t,\, s\{\ldots \$\kappa_i := f_i, \ldots\}\} \;:\; s(\alpha) \to t}$$

To type instances of non-initial algebra morphisms, we require a typing for partition relations. The codomain of a partition relation does not have a unique structural type, for it is only specified up to a variety. To express this, ADL provides a unique type constructor, $E\$s(\alpha)$, to correspond to each (unsaturated) sort, s. The type of a partition relation for this sort will be of the form $t' \to E\$s(t')$, where t' is a type, the type of the carrier of the domain algebra for an instance of hom$[s]$. The well-typing of a partition relation furnishes an additional hypothesis of the typing rule for hom.

$$\frac{\begin{array}{l} p \,:\, t' \to E\$s(t') \\ [\alpha : \text{type}],\, \rho \vdash_{Alg} A\{c := t,\, s\{\ldots \$\kappa_i := f_i, \ldots\}\} \end{array}}{\rho \vdash \text{hom}[s]\; A\{c := t,\, s\{\ldots \$\kappa_i := f_i, \ldots\}\}\; p \,:\, t' \to t}$$

For an example, consider typing the definition of *pwr_2* in Example 3.1. First, we check the well-typing of the *nat* algebra. For the carrier binding $c := int$, the operator typings will be $\$zero \,:\, int$ and $\$succ \,:\, int \to int$. These are satisfied by the bindings $\$zero := m$ and $\$succ := \lambda n\, n + 1$, where $m : int$. Thus the algebra specification $Nat\{c := int;\; nat\{\$zero := m,\; \$succ := \lambda n\, n + 1\}\}$ is well-typed.

Next we type the partition relation, p. In this relation, the operators $\$zero$ and $\$succ$ are considered to be unbound, and so their typings are expressed with the codomain type represented by $E\$nat(\alpha)$, Choosing $\alpha = int$ we get the specific typings $\$zero \,:\, E\$nat(int)$ and $\$succ \,:\, int \to E\$nat(int)$, which gives p the typing $p \,:\, int \to E\$nat(int)$. Applying the rule for structural typing of hom gives

$$pwr\_2 = \text{hom}[nat]\; Nat\{c := int;\; nat\{\$zero := m,\; \$succ := \lambda n\, n + 1\}\}\; p \,:\, int \to int$$

However, to get the proper ADL typing, we must provide a domain predicate under which the algorithm can be proved to terminate. A termination condition is that the operation $\lambda n\, n\ \mathbf{div}\ 2$ must be compatible with a well-ordering relation over the predicated domain. A suitable domain restriction is $\forall n : int.\ n \neq 0$. Thus a proper ADL typing is

$$pwr\_2 \,:\, \{n : int \mid n \neq 0\} \to int$$

This typing is not unique, however. Another proper typing is

$$pwr\_2 \,:\, \{n : int \mid n > 0\} \to int$$

5 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).

Recognition that monads are useful in programming is relatively recent [Mog91, Wad90]. Monads have been used to explain control constructs such as exceptions [Spi90] and advocated as a basis for formulating reusable modules [Wad92].

Monads cannot be specified with the simple, sorted signature declarations available in ADL. Instead, ADL provides a predefined variety, whose signature is

$$\textbf{signature}\ Monad\{\textbf{type}\ M(a);\ \ monad(a)/M(a) = \{\mathit{\$unit}\ \text{of}\ a,\ \ \mathit{\$mult}\ \text{of}\ M(M(a))\}\}$$

where $M(a)$ is type expression in which the parameter a has only positive occurrences (with respect to the arrow constructor). Positive occurrences are defined in terms of a predicate Pos_a, defined as follows:

$$\begin{array}{l}
Pos_a(a) = \mathit{true} \\
Pos_a(b) = \mathit{true}\ \text{if}\ b \neq a \\
Pos_a(X * Y) = Pos_a(X) \wedge Pos_a(Y) \\
Pos_a(X + Y) = Pos_a(X) \wedge Pos_a(Y) \\
Pos_a(X \rightarrow Y) = Neg_a(X) \wedge Pos_a(Y)
\end{array}$$

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

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

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

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:

$$mult^M_a \circ unit^M_{M(a)} = id_{M(a)} \tag{1}$$

$$mult^M_a \circ (map\_M\, unit^M_a) = id_{M(a)} \tag{2}$$

$$mult^M_a \circ mult^M_{M(a)} = mult^M_a \circ (map\_M\, mult^M_a) \tag{3}$$

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,

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

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

$$ext^M\ mult^M_a = id_{M(a)} \tag{4}$$

$$ext^M\ f \circ unit^M = f \tag{5}$$

$$ext^M\ (ext^M\ f \circ g) = ext^M\ f \circ ext^M\ g \tag{6}$$

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

$$ext^M\ (unit^M \circ f) = map\_M\, f \tag{8}$$

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

$$dist^M \circ (unit^M_a \times unit^M_b) = unit^M_{a \times b} \tag{9}$$

When M is a monad derived from an inductive algebraic signature, it is also sensible to have a distribution function to be used with primitive recursion,

$$dist2^M \ : \ (a \times M(a)) \times (b \times M(b)) \rightarrow M(a \times b)$$

The coherence condition required of the primitive recursive product distribution function is:

$$dist2^M \circ (\langle id_a, unit_a^M \rangle \times \langle id_b, unit_b^M \rangle) \quad = \quad unit_{a\times b}^M \tag{10}$$

5.1 Monad declarations in ADL

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

$$\begin{array}{ll}\texttt{monad}\ \{ & name\Big[(type\ expr.)\Big]\,(type\ id) = type\ expr; \\ & \$unit := expression, \\ & \$mult := expression\}\end{array}$$

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.

5.2 Some useful monads

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

5.2.1 Exceptions

$$\begin{array}{l}\textbf{monad}\ \{Ex_i(\alpha) = \textbf{free}\{\$just\ \text{of}\ \alpha,\ \$exc_i\}; \\ \qquad \$unit\ := \lambda x\ Just(x), \\ \qquad \$mult\ := \lambda t\ \textbf{case}\ t\ \textbf{is} \\ \qquad\qquad\qquad\qquad \ \ Just(x) => x \\ \qquad\qquad\qquad\quad\ \ |\ \ i => \$exc_i \\ \qquad\qquad\qquad\quad\ \ \textbf{end}\}\end{array}$$

where i ranges over identifiers, excluding "*Just*".

in which 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 correspond 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 $f : a \to b$ 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, $[\![f]\!]$.

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

$$
\begin{array}{l}
dist^{Ex_i}(x, y) = \textbf{case } x \textbf{ of} \\
\qquad Just(x') => \textbf{case } y \textbf{ of} \\
\qquad\qquad Just(y') => Just(x', y') \\
\qquad\qquad \mid\ \ i => i \\
\qquad\qquad \textbf{end} \\
\qquad \mid\ \ i => i \\
\qquad \textbf{end}
\end{array}
$$

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

There is a useful primitive recursive product distribution function for the monad of exceptions.

$$
\begin{array}{l}
dist2^{Ex_i}((u, x), (v, y)) = \textbf{case } x \textbf{ of} \\
\qquad Just(x') => \textbf{case } y \textbf{ of} \\
\qquad\qquad Just(y') => Just(x', y') \\
\qquad\qquad \mid\ \ i => Just(x', v) \\
\qquad\qquad \textbf{end} \\
\qquad \mid\ \ i => \textbf{case } y \textbf{ of} \\
\qquad\qquad Just(y') => Just(u, y') \\
\qquad\qquad \mid\ \ i => i \\
\qquad\qquad \textbf{end} \\
\qquad \textbf{end}
\end{array}
$$

Note that while *dist* uses the exception as an anihilator, *dist2* treats it more nearly as an identity element.

5.2.2 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.

$$\begin{aligned}\textbf{monad}\ \{&St(\beta)(\alpha) = \beta \to \alpha \times \beta;\\ &\$unit := \lambda a\, \lambda b\, (a, b),\\ &\$mult := \lambda t\, \lambda b\ \textbf{let}\ (s, b') = t\, b\ \textbf{in}\ s\, b'\}\end{aligned}$$

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:

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

5.2.3 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.

$$\begin{aligned}\textbf{monad}\ \{&Sr(\beta)(\alpha) = \beta \to \alpha;\\ &\$unit := \lambda a\, \lambda b\, a,\\ &\$mult := \lambda t\, \lambda b\, t\, b\}\end{aligned}$$

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

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

5.2.4 The continuation-passing monad

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

$$\begin{aligned}\textbf{monad}\ \{&\mathit{CPS}(\alpha) = (\alpha \to \beta) \to \beta;\\ &\$\mathit{unit} := \lambda a\, \lambda c\, c\, a,\\ &\$\mathit{mult} := \lambda t\, \lambda c\, t\, (\lambda s\, s\, c)\}\end{aligned}$$

in which β is a free variable ranging over types.

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

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

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

5.3 Composite monads

The monad constructions introduced above can be used in conjunction with one another to specify composite monads. However, composition of monads is a bit tricky; arbitrary compositions do not exist, nor is there an operator to compose monads. Functors compose uniformly, but they are not directly represented in ADL except in the module facility.

In specifying a composite monad, the order in which the constituents are grouped is significant. The permissible orders of grouping are described by the string below, in which a parenthesized name indicates that the constituent may be repeated. Any constituent may be omitted.

$$(\mathit{Sr})\,(\mathit{St})\ (\mathit{Sr})\ \mathit{CPS}\,(\mathit{Sr})\,(\mathit{Ex})\,(\mathit{Sr})$$

Although state readers can be placed anywhere in the composite, the normal position would be at the far left. A state reader simply indicates that every computation may depend upon a static state object, such as an environment that maps identifiers to their meanings.

When a state transformer is introduced in a composite, the state component is implicitly paired with every value resulting from a computation, and every computation is implicitly dependent upon the current state component. Thus, for instance, the type of a composite $St(int)\,(CPS(string))$ will be $int \rightarrow ((string \times int) \rightarrow \beta) \rightarrow \beta$.

The CPS monad does not form a composite with itself. The monad of exceptions could, in principle, be introduced earlier in the string of component monads but the composite would probably not be what is intended. There are also monads corresponding to many familiar datatypes, and we have not addressed the question of how to include them in composites. However, datatypes seem to be more useful in ADL to characterize algebras (emphasizing control structure) than to characterize monads (emphasizing data structure).

5.3.1 Unit and multiplier

For the composites we have considered, the rules for forming the unit are simple:

$$\begin{aligned} unit^{St(S)\,M} &= \lambda x\; unit^{M} \circ (unit^{St(S)}\, x) \\ unit^{M_1\,M_2} &= unit^{M_1} \circ unit^{M_2} \qquad \text{when } M_1 \neq St(S) \end{aligned}$$

The rules governing the multiplier of a composite monad are somewhat more complex. Given monads M_1 and M_2, a rule for deriving a composite multiplier is:

$$mult^{M_1\,M_2} = map^{M_1}\,(mult^{M_2}) \circ mult^{M_1} \circ map^{M_1}\,(dist^{M_1}_{M_2})$$

where $dist^{M_1}_{M_2} : M_2(M_1(\alpha)) \rightarrow M_1(M_2(\alpha))$ is a polymorphic function that distributes the structure of one monad over the other. Here are some examples of such monad distribution functions:

$$\begin{aligned} dist^{St(A)}_{St(B)} &= \lambda t : St(A)\,(St(B)\,(X))\,\lambda a : A\,\lambda b : B\; \mathbf{let}\; ((x, b'), a') = t\,a\,b\; \mathbf{in}\; ((x, a'), b') \\ dist^{St(A)}_{Ex_{\mathrm{Nothing}}} &= \lambda t : Ex_{Nothing}(St(A))\,\lambda a : A \\ &\qquad\quad \mathbf{case}\; t\; \mathbf{is} \\ &\qquad\qquad Just(s) \Rightarrow Just(s\,a) \\ &\qquad\quad \mid\;\; Nothing \Rightarrow Nothing \\ &\qquad\quad \mathbf{end} \\ dist^{St(A)}_{CPS} &= \lambda t : ((X \rightarrow \beta) \rightarrow \beta)\,\lambda a : A\,\lambda c : ((X \times A) \rightarrow \beta)\,t\,(\lambda s\,c\,(s\,a)) \end{aligned}$$

$$
\begin{array}{rcl}
dist^{CPS}_{Ex_{\text{Nothing}}} & = & \lambda t : Ex_{Nothing}((X \to \beta) \to \beta)\, \lambda c : (Ex_{Nothing}(X \to \beta) \\
& & \quad \textbf{case } t \textbf{ is} \\
& & \quad \quad Just(t') \Rightarrow t'\,(c \circ Just) \\
& & \quad \mid \;\; Nothing \Rightarrow c\; Nothing \\
& & \quad \textbf{end}
\end{array}
$$

Ordinarily, declarations of monads and the required distribution functions will be supplied in an ADL library and would not ordinarily be constructed "on-the-fly" by an ADL programmer.

5.4 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 $s : a \to a$, we can make the binding $\$succ := [\![\ s\ ]\!]$ to designate $map^M\ s\ :\ M(a) \to M(a)$. If $x : a$ we could write $[\![\ x\ ]\!]$ 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 5.1 : For example, we can interpret the algebra of trees with carrier $Ex_{Nothing}(tree(string))$ to specify a function that replaces Tip nodes in the tree structure if the contents of the Tip match a specified string.

```
replace_in_tree s t =
      red[tree] Pr_Tree{c:= Ex_Nothing(tree(string));
                        $tip := \x if s=x then [|t|] else Nothing,
                        $fork := [|Fork|] o dist2_Ex}
```

Using a case discrimination eliminates the disjoint union, we obtain a space-efficient algorithm for tree replacement.

```
replace x t u = case replace_in_tree x t u is
                  Just(u') => u'
                | Nothing  => u
                end
```

The algorithm is space-efficient because a tree in which no replacement is required is not copied. The value delivered by *replace* in such a case is the original data structure. Note also that if the monad $Ex_{Nothing}$ is implemented with control transfers rather than by tagged values, then the **case** discrimination and the distribution function $dist^{Ex}$ have virtually no performance cost.

□

Exercise 5.1 Labeling a tree

Given a signature of binary trees with labeled nodes,

$$\textbf{signature } Btree\{\textbf{type } c;\ \ btree(a)/c = \{nt,\ \ node \textbf{ of } c * a * c\}\}$$

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. Could you do a breadth-first enumeration as well?

Exercise 5.2 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 5.3 Justifying lines of text

Extend the solution of Exercise 5.1 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.

6 Coinductive signatures

So far, we have only considered the signatures of algebras, in which each operator has a typing of the form $op_i : t_{i,1} * \ldots * t_{i,m_i} \rightarrow c$, where c designates the type of the carrier. When the operators are free and the carrier is the set of terms they construct, the signature defines a datatype that corresponds to its free term algebra. There is a dual to this construction.

Suppose operators were given typings of the form $op_i : c \rightarrow t_{i,1} * \ldots * t_{i,m_i}$, and the collection of operators given in a signature were the projection functions of a *record* template. When the operators are free and the carrier is the set of (infinite) records from which they project field values, the signature defines a coinductive datatype that corresponds to its free term coalgebra. In general, however, the operators of a coalgebra should be thought of as witnesses of the structure imposed upon the carrier. Coalgebras play as significant a role in ADL as do algebras. They define iterative control structures.

The quintessential coinductive coalgebra has the following signature:

$$\begin{aligned}\textbf{cosig } \mathit{Stream}\{\textbf{type } c;\ \mathit{str}(\alpha)/c = \{\mathit{\$shd} &: \alpha,\\ \mathit{\$stl} &: c\}\}\end{aligned}$$

A free *Stream* coalgebra has as its carrier a type $str(\alpha)$ whose elements are infinite streams. The two functions $Shd : str(\alpha) \rightarrow \alpha$ and $Stl : str(\alpha) \rightarrow str(\alpha)$ are defined as projections on a stream whose elements are of type α. Every stream is infinite; that is, it is always meaningful to apply the projection operators to a stream, even though there is no way to witness the entire stream at once. A stream provides a good model for an incrementally readable input file. The projection *Shd* yields the value of the first element of a stream, just as a *get* operation on an open file produces a value from the file buffer. The projection *Stl* yields a stream but that stream is not manifested until projections of it are taken. The situation is familiar in languages with lazy evaluation rules, but the operational semantics of ADL involve call-by-value.

In an infinite stream, there are both finite and infinite paths. A path is expressed by a well-typed composition of the operators *Shd* and *Stl*. A finite path is one that ends in *Shd*. To generate all paths in a stream, a control structure must support repeated applications of *Stl* until there is a final application of *Shd*, which terminates the path.

6.1 Generators and coalgebra morphisms

Definition 6.1 Let t denote an unsaturated sort of a coalgebra signature. A *T-coalgebra* consists of a pair (c, k) where c is a type, the carrier of the coalgebra, and $k : c \to t(c)$ is a co-structure function.

Definition 6.2 A function $g : a \to b$ is a *T-coalgebra morphism* if there are T-algebras (a, h) and (b, k) such that the following square commutes:

$$\begin{tikzcd} a \arrow[r, "g"] \arrow[d, "h"'] & b \arrow[d, "k"'] \\ t(a) \arrow[r, "map\_t\, g"] & t(b) \end{tikzcd}$$

□

in which t is the (single) sort of the coalgebra T.

A T-coalgebra *generator* is a function of a type $a \to t(b)$ equal to the composition of a T-coalgebra structure function with a T-coalgebra morphism, i.e. it is a diagonal arrow in a diagram such as the one above. A generator is characterized by a coalgebra specification in ADL. A coalgebra specification is an instance of a coalgebra signature, and provides a type for the carrier and specific functions for the operators of the signature. However, the type parameter of an unsaturated sort, t, is not restricted to be the same as the carrier, as is the case in the mathematical definition (Definition 6.1).

The construction of a generalized co-structure function can be understood in terms of the diagram below, which represents one level of "unfolding" of the recursive definition of a free coalgebra. To express the unfolding, we require some notation for the general case of the coalgebraic structure expressed in a signature. For simplicity, we consider a single-sorted signature whose sort is unsaturated with a single parameter, i.e. $t : * \to *$. Suppose the signature consists of n operators. The i^{th} operator has a typing $c \to t_{i,1} * \ldots * t_{i,m_i}$ where c is the type variable representing the carrier, a is the type variable representing the type parameter of the sort t, and each of the $t_{i,j}$ is either c or a. We can represent such types by $c \to F_i(a,c)$, capturing with the symbol F_i the structure of the i^{th} codomain type. Note that we can use the same symbol to designate a composite function of type $F_i(a,c) \to F_i(a,t(a))$ that is obtained by the component-wise application of $f : a \to a$ to each component of type a, and $g : c \to t(a)$ to each component of type c. We designate the component-wise application by $F_i(f,g)$, for each $i \in 1..n$. This notational convention is used in the following diagram, in which $\times^n$ means the n-fold product of the indexed family of components.

$$\begin{tikzcd} a \arrow[r, "g"] \arrow[d, dotted, "k"'] & \times^n F_i(b,a) \arrow[d, "\times^n F_i(id_b, k)"] \\ t(b) \arrow[r, "\mathbf{out}"'] & \times^n F_i(b, t(b)) \end{tikzcd}$$

in which **out** is a natural (i.e. polymorphic) isomorphism. The generalized co-structure function, k, satisfies the equation

$$k = \mathbf{out}^{-1} \circ \times^n F_i(id, k) \circ g$$

The data on which k depends consists of the sort, t, and the coalgebra specified by g. In ADL, a generalized co-structure function is defined in terms of a combinator *gen* applied to these data.

Example 6.1 For example, the following expression generates a stream of ascending integers from an integer given as its argument:

$$\mathit{from} = \mathit{gen}[\mathit{str}]\ \mathit{Stream}\{c := \mathit{int};\ \$\mathit{shd} := \mathit{id},\ \$\mathit{stl} := \mathit{add}\,1\}$$

Thus $\mathit{from}\,0$ generates the sequence of non-negative integers. We have the following equalities:

$$\begin{aligned} \mathit{Shd}(\mathit{from}\,0) &= 0 \\ \mathit{Stl}(\mathit{from}\,0)\,() &= \mathit{from}\,1 \\ \mathit{Shd}(\mathit{Stl}(\mathit{from}\,0)\,()) &= 1 \\ \mathit{Stl}(\mathit{Stl}(\mathit{from}\,0)\,()) &= \mathit{from}\,2 \\ \cdots &= \cdots \end{aligned}$$

From these equalities we see that every finite path of the stream $\mathit{from}\,0$ can be witnessed. Note that the witnesses gotten by applying Stl are suspended. The typing of Stl is

$$\mathit{Stl} \;:\; \mathit{str}(\alpha) \to \mathbf{1} \to \mathit{str}(\alpha)$$

Explicit suspension is necessary because ADL is a call-by-value language.

Example 6.2 A stream constructor function can be defined in terms of the stream generator combinator and the first and second projections of a cartesian product. Cartesian products exist in ADL because all functions are total. A stream constructor $\mathit{str\_cons} : a \times \mathit{str}(a) \to \mathit{str}(a)$ is:

$$\mathit{str\_cons} = \mathit{gen}[\mathit{str}]\ \mathit{Stream}\{c := a;\ \$\mathit{shd} := \mathit{fst},\ \$\mathit{stl} := \mathit{snd}\}$$

□

6.2 Witness paths

To compose a witness function for a coalgebra, we can formulate an inductive algebra to characterize a *path grammar* for the coalgebra. A path grammar generates all instances of finite paths, or witness functions, for the coalgebra. A path grammar is a meta-language concept, and is not formally expressible in ADL. Path grammars are useful for reasoning about coalgebras, however.

For *Stream*-coalgebras, all paths are linear, formed by iteration of the Stl operator. Thus a path grammar for *Stream* may be expressed as an instance of a *Nat* algebra whose carrier is a function from $\mathit{str}(\alpha)$ to the set of terms produced by witnessing a stream. We call this set of terms $L(\mathit{str}(\alpha))$. It corresponds to the union of the codomains of all the witness functions in the signature of the coalgebra. This union of codomains is not expressible as a type in ADL.

$$\mathit{paths}[\mathit{str}] = \mathit{red}[\mathit{nat}]\ \mathit{Nat}\{c := \mathit{str}(\alpha) \to L(\mathit{str}(\alpha));\ \$\mathit{zero} := \$\mathit{shd},\ \$\mathit{succ} := \lambda s.\, s \bullet \$\mathit{stl}\}$$

where $g \bullet f = \lambda x.\, g(f\ x\ ())$.

A natural number determines a path for $\mathit{str}(\alpha)$, and hence a witness function. For example,

$$\mathit{Shd}\,(\mathit{paths}[\mathit{str}]\ \mathit{Succ}(\mathit{Succ}(\mathit{Zero}))\ (\mathit{from}\,0)) = 2$$

For other coalgebras, the path grammars have more complex structure than *Nat* algebras.

Example 6.3 A coalgebraic variety with particularly simple structure is given by the signature

$$\textbf{cosig}\ \mathit{Iter}\{\textbf{type}\ c;\ \mathit{inf}/c = \{\$\mathit{step} : c\}\}$$

A generator for this coalgebra calculates a least fixpoint of the function bound to $\$step$. Let $f : a \to a$. Then

$$\mathit{fix}\ f = \mathit{gen}[\mathit{inf}]\ \mathit{Iter}\{c := a;\ \$\mathit{step} := f\}$$

This generated object is not only infinitary, it has no finite witness paths. However, this does not necessarily mean that it is devoid of computational meaning. If $a = b \to d$, then *fix f* has an interpretation as the least fixpoint of f, in a domain of partial functions. However, *fix f* is typed as *inf* by the structural typing rules of ADL, not as $b \to d$, hence no application of *fix f* is well typed.

Example 6.4 Another interesting coalgebraic variety is expressed by the signature

$$\begin{aligned}\textbf{cosig}\ \mathit{BinTree}\{\textbf{type}\ c;\ \mathit{bintree}(\alpha)/c = \{&\$\mathit{val} : \alpha,\\ &\$\mathit{left}, \$\mathit{right} : c\}\}\end{aligned}$$

A generator for *bintree*(*int*) is:

$$\begin{aligned}\mathit{gen}[\mathit{bintree}]\ \mathit{BinTree}\{&c := \mathit{int};\\ &\$\mathit{val} := \mathit{id},\\ &\$\mathit{left} := \lambda m.\, 2m,\\ &\$\mathit{right} := \lambda m.\, 2m + 1\}\end{aligned}$$

When this generator is applied to the integer value 1, it generates the infinite, binary tree whose integer labels enumerate the tree breadth-first.

A *bintree* generator incrementally generates streams of data witnessed via a sequence of binary decisions. The generator

$$\begin{aligned}\mathit{gen}[\mathit{bintree}]\ \mathit{BinTree}\{&c := \mathit{bool};\\ &\$\mathit{val} := \mathit{id},\\ &\$\mathit{left} := \lambda s.\, \mathit{ff},\\ &\$\mathit{right} := \lambda s.\, \mathit{tt}\}\end{aligned}$$

(where tt and ff are the identifiers of the boolean constants) generates a tree whose paths correspond to finite and infinite sequences of boolean-valued labels. Thus the set of paths contains the set of rational fractions in a binary representation. Its path grammar is specified with a $list(bool)$ algebra.

$$\begin{aligned} paths[bintree] = red[list]\ List\{ & c := bintree(\alpha) \to L(bintree(\alpha)); \\ & \$nil := id, \\ & \$cons := \lambda(b, s).\ \textbf{if}\ b\ \textbf{then}\ s \bullet Right\ \textbf{else}\ s \bullet Left\} \end{aligned}$$

□

6.3 Proof rules for generators

Coinductive proof rules assert properties that can be witnessed on all paths by which values of a coinductively generated object are accessed.

For example, the single proof rule for *Stream* generators is:

$$\frac{c \text{ type} \qquad x : c \qquad P(x) \Rightarrow P(\$stl\, x)}{P(x) \Rightarrow \forall i : nat.\ P(paths[str]\ i\ (gen[str]\ Stream\{c;\ \$shd := id,\ \$stl\}\ x))}$$

For *BinTree* generators (Example 6.4), the proof rule is:

$$\frac{c \text{ type} \qquad x : c \qquad P(x) \Rightarrow P(\$left\, x) \qquad P(x) \Rightarrow P(\$right\, x)}{P(x) \Rightarrow \forall s : list(bool).\ P(paths[bintree]\ s\ (gen[bintree]\ BinTree\{c;\ \$val := id,\ \$left,\ \$right\}\ x))}$$

7 Constructing coalgebra morphisms

Generators have limited direct use as control paradigms for algorithms. However, just as is the case for algebras, the morphisms of non-free coalgebras will yield many useful control schemes.

To calculate a coalgebra morphism, it would be sufficient to have an inverse to the arrow on the right-hand side of the morphism diagram of Definition 6.2. Then an equation to be satisfied by a coalgebra morphism k can be read from the diagram below,

$$\begin{tikzcd} a \arrow[r, "g"] \arrow[d, "k"'] & \times^n F_i(a) \arrow[d, "\times^n F_i(k)"] \\ b & \times^n F_i(b) \arrow[l, "p"] \end{tikzcd}$$

in which the projection function, p, may involve a selection conditional on the data, and may even be a partial function.

7.1 The combinator *cohom*

To realize morphisms of coalgebras that are not free, ADL provides a combinator, *cohom*, that takes as arguments a coalgebra and a splitting function. The splitting function specifies a path for witness of the coalgebra's carrier. The splitting function typically involves decisions conditional on witnessed values of the carrier, and thus, recursive elaboration of the path is implied. Just as was the case with morphisms of non-free algebras, there are proof obligations to show that an instance of *cohom* is well-founded (and hence, well-defined).

In implementing such a control structure, each application of an operator whose codomain includes the carrier, such as *\$stl*, must be suspended or else the recursive elaboration of its repeated application would fail to terminate, affording no opportunity to witness finite paths.

To make effective the suspension of projections by operators whose codomain type contains an instance of the carrier, any projection operator whose codomain includes the carrier is implicitly suspended in ADL. Suspension is not necessary for projections whose codomain type does not involve the carrier. For example, the operators of $str(\alpha)$ are given the typings

$$\begin{aligned} \mathit{\$shd} &: c \to \alpha \\ \mathit{\$stl} &: c \to 1 \to c \end{aligned}$$

where 1 designates the unique type with a single element, which is designated by $()$. A type $1 \to c$ is the type of a suspended value. To obtain an actual value, an applicative expression such as $\mathit{\$stl}\, s$ must be applied to an additional argument, namely $\mathit{\$stl}\, s\, ()$.

Example 7.1 For a familiar example of a *str*-algebra morphism, consider the construction from functions $p : c \to \mathit{bool}$ and $r : c \to c$,

$$\begin{aligned} \mathit{while}_c\,(p, r) = \mathit{cohom}[\mathit{str}]\, \mathit{Stream}\{c;\ \mathit{\$shd} := \mathit{id}_c,\ \mathit{\$stl} := r\} \\ (\lambda(x, y).\ \mathbf{if}\ p\, x\ \mathbf{then}\ y\,()\ \mathbf{else}\ x) \end{aligned}$$

This is the useful unbounded iteration construct found in nearly all programming languages. It encompasses the paradigm of linear search. To be well-defined in ADL, a *while* iteration must be shown to be bounded. This question will be addressed when we consider proof rules and finiteness conditions for coalgebra morphisms.

Example 7.2 Another paradigm for the recursive generation of a stream is the following. Let $f : str(a) \rightarrow str(a)$. Define

$$\begin{array}{lcl} rec(f) & : & a \rightarrow str(a) \\ rec(f) & = & cohom[str]\; Stream\{c := a;\; \$shd := id,\; \$stl := id\} \\ & & \qquad (\lambda(x, y).\, str\_cons(x,\; f\,(y\,()))) \end{array}$$

An equation that this coalgebra morphism satisfies is

$$rec(f) = str\_cons \circ \langle id_a,\; f \circ rec(f)\rangle$$

Example 7.3 With a *bintree* morphism, we can specify binary search.

$$\begin{array}{l} bsearch(key : int) = cohom[bintree]\; BinTree\{c;\; \$val,\; \$left,\; \$right\} \\ \qquad\qquad (\lambda(n, l, r).\, \textbf{if } n = key \textbf{ then } n \\ \qquad\qquad\qquad \textbf{else if } n < key \textbf{ then } l\,() \\ \qquad\qquad\qquad \textbf{else } r\,()) \end{array}$$

Here, the coalgebra specification is incomplete, as bindings for the carrier and the operators have not been given. Binary search can be programmed for any *bintree* coalgebra whose witness function has *int* as its codomain.

Exercise 7.1 Consider an algebra of labeled, binary trees given by the following signature:

$$\begin{array}{l} \textbf{signature } Ltree\{c \text{ type}; \\ \qquad ltree(\alpha)/c = \{\$empty, \\ \qquad\qquad \$node \textbf{ of } c * a * c\}\} \end{array}$$

Give an algorithm in ADL to construct from an arbitrary instance of an *ltree* in the free term algebra, a new copy whose nodes are labeled by their enumeration in a breadth-first traversal. *Hint: Assume that there is a stream of integers that are the generators for the labels to be used at each level in the tree. Use these to label the tree, then construct the stream with the paradigm of Example 7.2.*

7.2 Typing coalgebra combinators

Like algebra specifications, coalgebra specifications also have simple structural typing rules. The rules presented in this preliminary report are restricted to a single-sorted coalgebra (with sort s and signature Σ_s).

The first rule is one for typing a coalgebra specification. The judgement form "$\rho \vdash_{\text{Coalg}} A\{\cdots\}$" can be read as "the coalgebra specification $A\{\cdots\}$ is well-typed relative to the environment ρ".

$$\frac{\begin{array}{l}\forall(\$\pi_i, \sigma_i) \in \Sigma_s.\ [\alpha : \text{type}], \rho \vdash e_i : t \to t_i \\ \qquad\qquad \wedge\, t_i = \begin{cases} 1 \to \rho[t/c](\sigma_i) & \text{if } c \text{ does not occur in } \sigma_i \\ \rho(\sigma_i) & \text{otherwise} \end{cases}\end{array}}{[\alpha : \text{type}], \rho \vdash_{\text{Coalg}} A\{c := t;\ s(\alpha)/c = \{\cdots \$\pi_i := e_i, \cdots\}\}}$$

The typing rule for a generator is then:

$$\frac{[\alpha : \text{type}], \rho \vdash_{\text{Coalg}} A\{c := t;\ s(\alpha)/c = \{\cdots \$\pi_i := e_i, \cdots\}\}}{[\alpha : \text{type}], \rho \vdash \mathit{gen}[s]\, A\{c := t;\ s(\alpha)/c = \{\cdots \$\pi_i := e_i, \cdots\}\}\ :\ t \to s(\alpha)}$$

To give a typing for *cohom*, one must type a splitting function in addition to a coalgebra specification.

$$\frac{\begin{array}{l} g\ :\ (t_1 \times \cdots \times t_n) \to t' \\ [\alpha : \text{type}], \rho \vdash_{\text{Coalg}} A\{c := t;\ s(\alpha)/c = \{\cdots \$\pi_i := e_i, \cdots\}\}\end{array}}{\rho \vdash \mathit{cohom}[s]\, A\{c := t;\ s(\alpha)/c = \{\cdots \$\pi_i := e_i, \cdots\}\}\, g\ :\ t \to t'}$$

7.3 Termination conditions for *cohom*

A function defined by a *cohom* combinator selects a particular path for access of a value from its coalgebra. Thus any property that can be inferred of all finite paths in the coalgebra will hold for any path selected by an application of a function defined by a *cohom*, provided that the path is finite. The additional proof obligation for a *cohom* is just a proof of finiteness, or a termination proof.

A finiteness proof can be formalized as a proof that the set of paths that may be selected by a *cohom* is well-ordered. Typically, such a proof will hold only when the domain of the function is restricted by a predicate. A finiteness predicate can be inductively defined through clauses that mimic the structure of a coinductive proof. An inductive proof consists of a set of implication schemas and a rule establishing that a given predicate holds of each element of a set, by a finite chain of implications drawn from instances of the schemas. Inductive proof is an argument by which to establish a property of a set in terms of its construction. The construction of morphisms of certain varieties of structure algebras help to shape the necessary finiteness arguments in terms of well-ordering relations, thus the technique is applicable to establish properties of the codomains of algebra morphisms of these varieties.

Dually, A coinductive proof consists of a set of implication schemas and a rule establishing that a given predicate holds for each witness of a set, by a finite chain of implications drawn from instances of the schema. Coinductive proof is an argument by which to establish a property of a set in terms of its witnesses. The construction of morphisms of certain varieties of coalgebras helps to shape the finiteness arguments in terms of well-orderings, thus the technique is applicable to establish properties of the domains of coalgebra morphisms of these varieties.

A finiteness predicate is the least specific assertion that can be made about the domain of a coalgebra morphism. It asserts only that all finite witnesses are defined. Thus a finiteness predicate characterizes the domain of a coalgebra morphism. The structure of such a finiteness proof is that

- a predicate, P, holds of some initial values by direct implication from facts, i.e. without use of any implication that involves P in its hypothesis, and
- for every operator, $\$\pi_i$ of the coalgebra, let $(x_1, \ldots, x_n) = \$\pi_i\, x$.
  Then $P(x) \Rightarrow P(x_1) \wedge \cdots \wedge P(x_n)$.

Example 7.4 :

For the construct $\mathit{while}(p, r)$, a finiteness predicate is the least specific that satisfies

$$\begin{aligned} p\,x = \mathit{ff} \quad &\Rightarrow \quad P(x) \\ P(r\,x) \quad &\Rightarrow \quad P(x) \end{aligned}$$

subject to the condition that there exists a well-ordering, $(\prec)$, such that $\forall x.\ P(x) \Rightarrow r\,x \prec x$. Then $P(x)$ is a termination condition for the evaluation of $\mathit{while}(p, r)\,x$.

8 Transformational development of algorithms

The ADL language has been designed to lend itself to transformational development, i.e. the improvement of algorithms by meaning-preserving, algebraic transformation of programs. Transformational development is an old idea, but the algebraic aspect of program transformation has been emphasized by Richard Bird [Bir84, Bir91] and his coworkers. The deforestation algorithms proposed by Wadler [Wad88] furnish a good example of general transformations. Wadler [Wad89] and Malcolm [Mal89] have observed that there are general classes of theorems that have instances for any inductive datatype. Such theorems are not only useful in justifying transformations, they may be automated as tactics for the application of term rewrites that actually effect the transformations. This observation is the basis for a higher-order transformation tool (HOT) currently being developed for use with ADL programs.

References

[Bir84] Richard S. Bird. The promotion and accumulation strategies in transformational programming. *ACM Transactions on Programming Languages and Systems*, 6(4):487–504, 1984. Addendum: Ibid. 7(3):490-492 (1985).

[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.

[Bir91] Richard S. Bird. Knuth's problem revisited. In B. Möller, editor, *Constructing Programs from Specifications*. North-Holland, 1991.

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

[Fok92] Maarten M. Fokkinga. *Law and Order in Algorithmics.* PhD thesis, University of Twente, Twente, The Netherlands, February 1992.

[Gog80] Joe 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.

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

[Mac71] Saunders MacLane. *Categories for the Working Mathematician.* Springer-Verlag, 1971.

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

[Mee86] Lambert Meertens. Algorithmics—towards programming as a mathematical activity. In *Proc. of the CWI Symbposium 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.

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

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

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

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

[Wad92] Philip Wadler. The essence of functional programming. In *Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–14. ACM Press, January 1992.