Revisiting Catamorphisms over Datatypes with Embedded Functions
(or, *Programs from Outer Space*)

Leonidas Fegaras   Tim Sheard

Department of Computer Science and Engineering
Oregon Graduate Institute of Science & Technology
20000 N.W. Walker Road P.O. Box 91000
Portland, OR 97291-1000
{fegaras,sheard}@cse.ogi.edu

Abstract

We revisit the work of Paterson and of Meijer & Hutton, which describes how to construct catamorphisms for recursive datatype definitions that embed contravariant occurrences of the type being defined. Their construction requires, for each catamorphism, the definition of an anamorphism that has an inverse-like relationship to that catamorphism. We present an alternative construction, which replaces the stringent requirement that an inverse anamorphism be defined for each catamorphism with a more lenient restriction. The resulting construction has a more efficient implementation than that of Paterson, Meijer, and Hutton and the relevant restriction can be enforced by a Hindley-Milner type inference algorithm. We provide numerous examples illustrating our method.

1 Introduction

Functional programmers often use catamorphisms (or fold functions) as an elegant means of expressing algorithms over algebraic datatypes. Catamorphisms have also been used by functional programmers as a medium in which programs can be calculated from specifications [7, 6, 5] and as a good intermediate representation of programs that supports optimization [14, 3, 4, 1]. It is, thus, truly ironic that these functions apply only to first order datatypes.

Until recently, it was not known how to express catamorphisms for datatypes with embedded function types. The work by Paterson [10] and Meijer & Hutton [8] finally provided a method for doing so. While elegant and theoretically sound, their solution suffers from the disadvantage of being somewhat inefficient. This paper extends their technique using a simple *trick* that results to more efficient programs and turns out to also have several other interesting applications.

Meijer and Hutton point out that in order to express a function f as a catamorphism over an arbitrary datatype it is necessary to express another function g as an anamorphism (or unfold function) which is defined mutually recursively with the catamorphism. The anamorphism is necessary if a recursive datatype definition has contravariant occurrences of the type being defined. The purpose of the anamorphism is to "undo" what the catamorphism "does". The "inverse-like" relationship of f and g is necessary to correctly handle the contravariance of the datatype definition. This approach requires a proof of the inverse-like relationship between f and g. In general there is no known automatic way to obtain such a proof.

Our trick involves inventing a particularly simple inverse-like function to take the place of the anamorphism. This method is simple, not only because it does not require an actual definition of the inverse-like function g, but also because it gives g a highly efficient implementation. The trick does not always apply, but we present a type system that determines statically when it is applicable.

In this paper we give numerous examples illustrating the usefulness of being able to define catamorphisms over datatypes with embedded functions, as well as showing how to define functions as catamorphisms.

To appear at the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida, January 1996. Also available as OGI Technical Report 95/014.

2 Structures with Functionals are Useful

In this section, we present several examples of data structures with embedded functions. We define catamorphisms over these structures and give many examples of their use. We informally introduce our method and explain how it works. Our first example is an evaluation function over a datatype that represents closed terms in a simple lambda calculus. This representation is interesting because the evaluation function does not need an environment mapping variables to values. The second example is the expression of the parametricity theorem for any polymorphic function. The third example is the expression and manipulation of circular lists in a functional language, and the last example is the expression and manipulation of graphs. All the functions in these examples can be expressed succinctly as catamorphisms.

2.1 Meta-programming

2.1.1 Lambda Calculus

Our first example of a function over a data structure with embedded functions is an evaluation function for a datatype that represents closed terms in a simple lambda calculus. We represent terms as structures with embedded functions in a manner similar to the higher-order abstract syntax representation of programs by Pfenning and Elliot [12] (all our examples are written in Standard ML (SML) [11]):

```
datatype Term = Const of int | Succ | Appl of Term × Term
                | Abs of Term → Term
```

For example, the lambda term $(\lambda x.1 + x)\,1$ is represented by the Term construction

```
Appl(Abs(fn x ⇒ Appl(Succ,x)),Const 1)
```

This term representation can be traced back to Church's seminal work on the lambda calculus [2], in which universal quantification $\forall x.A$ is modeled by the addition of a constant Π and by writing $\Pi(\lambda x.A)$. This representation is also similar to the higher-order abstract syntax representation of programs, formulas, and rules by Pfenning and Elliot [12], and is also related to the work of Nadathur and Miller [9] on higher-order logic programming.

The datatype definition for Term differs from most in that in most lambda term representations, lambda abstractions are constructed by value constructors of type variable × Term → Term, and most representations also include constructors like Var of type variable → Term. In such representations, an operation over lambda terms needs to handle variables explicitly. For example, a lambda-calculus evaluator typically needs to build and manipulate an environment to bind variables to values. In our representation the abstraction mechanism of the meta-language (SML) is used to represent abstraction in the object language of lambda terms and this completely finesses the bound-variable naming problem.

Our evaluator is thus spared the difficulties and complexity associated with the explicit representation of variables, since all the necessary variable plumbing, in beta reduction, etc., is handled implicitly by the evaluation engine of SML in which our evaluator is expressed. But the benefit of our approach is more than just pedagogical: one can experiment with various types of evaluators without worrying about the details of names and variable binding. Representations like this have been avoided because of the difficulty of expressing certain kinds of computations.

The Paterson/Meijer/Hutton Approach

We now present an evaluator for terms in the style of Paterson [10], and Meijer & Hutton [8]. We use explicit recursion instead of catamorphisms to make it clear how it works and to illustrate the inefficiencies of this approach. Our approach, which improves this method, is described in detail in the next subsection.

The value domain of our evaluator is

```
datatype Value = Num of int | Fun of Value → Value
```

The value domain itself uses embedded functions to represent the meaning of functional terms in the lambda calculus. The lambda term evaluator is a function of type Term → Value and could be expressed as:

```
fun eval(Const n)   = Num n
  | eval(Succ)      = Fun(fn Num n ⇒ Num(n+1))
  | eval(Appl(f,e)) = (case eval(f) of Fun(g) ⇒ g(eval(e)))
  | eval(Abs f)     = Fun(g?)
```

But it is not obvious what g? in the last clause should be. The type of g? should be Value → Value. The function f of type Term → Term must somehow be manipulated into a value of type Value → Value. Based on type considerations alone, one might attempt to construct g? as a composition of f with a function of type Term → Value on the left and a function of type Value → Term on the right. The obvious choice for the first function is eval. The second function, which we call reify, translates values into terms. The function reify should be the right inverse of eval on the range of eval, i.e., for all x in the range of eval the identity eval(reify(x)) $= x$ should hold. Without this restriction, eval would fail to evaluate even the simplest abstraction, namely Abs(fn x ⇒ x), into the correct result Fun(fn x ⇒ x). Since term evaluation is not an isomorphism in general, we have many different choices for reify. As a first pass at a definition of reify, we might reasonably try:

```
fun eval(Const n)   = Num n
  | eval(Succ)      = Fun(fn Num n ⇒ Num(n+1))
  | eval(Appl(f,e)) = (case eval(f) of Fun(g) ⇒ g(eval(e)))
  | eval(Abs f)     = Fun(eval ∘ f ∘ reify)
and reify(Num n)    = Const n
  | reify(Fun f)    = Abs(reify ∘ f ∘ eval)
```

Notice the symmetry between eval and reify. In particular, the way reify handles the embedded function inside Fun is the mirror image of the way eval handles the embedded function inside Abs.

It is instructive to visualize the process that occurs when an Abs term is evaluated. An abstraction is built that will be placed within a Fun constructor. This abstraction, when applied, will reify its argument to get a Term, then apply f to get another Term, and finally evaluate this term to get a Value. For example, the evaluation of the term $(\lambda x.\,x + 1)\;1$ proceeds as follows:

```
eval(Appl(Abs(fn x ⇒ Appl(Succ,x)),Const 1))
= case eval(Abs(fn x ⇒ Appl(Succ,x))) of
     Fun(g) ⇒ g(eval(Const 1))
= case Fun(fn x ⇒ eval(Appl(Succ,reify x))) of
     Fun(g) ⇒ g(eval(Const 1))
```

The abstraction in the case is eventually applied to a term, which must itself be evaluated. The above thus equals

```
eval(Appl(Succ,reify(eval(Const 1))))
= eval(Appl(Succ,reify(Num 1)))
= eval(Appl(Succ,Const 1))
= case Fun(fn Num n ⇒ Num(n+1)) of
     Fun(g) ⇒ g(eval(Const 1))
= (fn Num n ⇒ Num(n+1)) (Num 1)
= Num 2
```

We can see that there is some computational redundancy in the evaluator. Some terms, such as Const 1, are evaluated only to be reified later on. In general, if we reduce Appl(Abs(f),e), where e is a complex term, then we get eval(f(reify(eval(e)))). That is, the term e is evaluated into a value, and then this is reified into a term, then this term (after reduction by f) is evaluated again into a value by the outer occurrence of eval. This is the general scheme within the eval-reify example: reify will undo what eval has done, eval will partially redo what has been undone, and so on. This problem becomes increasingly worse with the complexity of function f. For example, to define a printer print for Term, which is a function from Term to string, it is necessary to define a parser parse from string to Term with the property print(parse(x))=x. The computational complexity of the parser is linear in the size of the input string. But, as we will see next, in many cases, an approximate right inverse of print with constant time complexity is all that is needed.

These dual mutually recursive definitions have been explored by Paterson [10] and Meijer & Hutton [8]. They use a pair of *generic* dual functions, catamorphism and anamorphism, to capture recursion schemes similar to the one in the eval-reify example: a catamorphism will reduce a value of type T into a value of type S while an anamorphism will generate a value of type T from a value of type S.

Our Approach. To avoid the computational redundancy of eval and to find a way around the problem of finding a right inverse for eval, we return to the initial problem of expressing g? as eval ∘ f ∘ h?. In particular, the crucial property of h? with type Value → Term is that it satisfies eval ∘ h? = $\lambda x.x$. In addition, this should happen with no computational overhead. One way to accomplish this is to take h? to be a value constructor and to add the defining equation eval (h? x) = x for eval. That is, we can change the domain, Term, of eval to accommodate a new constructor and then add an extra clause to the definition of eval.

With these ideas in mind, we modify the datatype definition of Term by adding a value constructor Place that implements h?:

```
datatype Term = Const of int | Succ | Appl of Term × Term
              | Abs of Term → Term | Place of Value
```

The evaluator is also extended accordingly:

```
fun eval(Const n)    = Num n
  | eval(Succ)       = Fun(fn Num n ⇒ Num(n+1))
  | eval(Appl(f,e))  = (case eval(f) of Fun(g) ⇒ g(eval(e)))
  | eval(Abs f)      = Fun(eval ∘ f ∘ Place)
  | eval(Place x)    = x
```

Under this definition of eval, the previous example evaluates as follows:

```
eval(Appl(Abs(fn x ⇒ Appl(Succ,x)),Const 1))
= case eval(Abs(fn x ⇒ Appl(Succ,x))) of
      Fun(g) ⇒ g(eval(Const 1))
= case Fun(fn x ⇒ eval(Appl(Succ,Place x))) of
      Fun(g) ⇒ g(eval(Const 1))
= eval(Appl(Succ,Place(Num 1)))
= case Fun(fn Num n ⇒ Num(n+1)) of
      Fun(g) ⇒ g(eval(Place(Num 1)))
= (fn Num n ⇒ Num(n+1)) (Num 1)
= Num 2
```

Notice that Const 1 is evaluated only once into Num 1 and it remains in that form protected by the Place constructor until it is used (this justifies the name Place, which acts as a placeholder). In a way, Place partially satisfies the requirements needed from reify. We will see that this partial satisfaction is "good enough" in many cases and its lack can be statically detected.

The technique of using a value constructor to approximate the right inverse of a function can be applied to computations over Term other than eval. To be completely general, it is necessary to generalize the type definition of Term by abstracting over the domain of Place with a new type variable α; after all, not all computations over Terms return Values:

```
datatype α Term = Const of int | Succ
                | Appl of α Term × α Term
                | Abs of α Term → α Term
                | Place of α
```

This allows any α object to be a subtype of Term, and Place plays the role of an injection function.

One appropriate generalization of eval is the catamorphism. The catamorphism operator for Term replaces each value constructor (Const, Succ, Appl, and Abs) in an instance of Term with a corresponding function (fc, fs, fp, and fa):

```
fun cataT(fc,fs,fp,fa) (Const n)   = fc n
  | cataT(fc,fs,fp,fa) Succ        = fs
  | cataT(fc,fs,fp,fa) (Appl(a,b)) = fp( cataT(fc,fs,fp,fa) a,
                                          cataT(fc,fs,fp,fa) b )
  | cataT(fc,fs,fp,fa) (Abs f)     = fa(cataT(fc,fs,fp,fa) ∘ f ∘ Place)
  | cataT(fc,fs,fp,fa) (Place x)   = x
```

Operator cataT has the following signature:

$$(\mathsf{int}\to\alpha)\times\alpha\times(\alpha\times\alpha\to\alpha)\times((\alpha\to\alpha)\to\alpha)\to\alpha\ \mathsf{Term}\to\alpha$$

That is, the abstracted type variable α (the domain of Place) is bound to the type of the result of cataT.

We may express the evaluator eval as a catamorphism as follows:

```
cataT( Num, Fun(fn Num n ⇒ Num(n+1)),
       fn (Fun(f),e) ⇒ f e, Fun )
```

A printer for a term is:

```
cataT(makestring, "succ",
      fn (a,b) ⇒ a ↑ " " ↑ b,
      fn f ⇒ let val n = newname()
             in "(fn " ↑ n ↑ " ⇒ " ↑ (f n) ↑ ")" end)
```

where newname returns a string that represents a new variable name and ↑ is string concatenation.

As another example of using a catamorphism over Term, we present a type inference algorithm. Types can be defined as follows:

```
datatype Type = Int | Arrow of Type → Type
```

A type inference algorithm for terms is a mapping from Term to Type:

```
cataT( fn _ ⇒ Int,
       Arrow(fn Int ⇒ Int | _ ⇒ type_error()),
       fn (Arrow(f),t) ⇒ f(t) | _ ⇒ type_error(),
       Arrow )
```

Type errors are reported when the inferred type is printed (since most calls to type_error are suspended in closures).

In light of code reuse issues, it is generally not advisable to extend type definitions by adding new constructors. Such extensions can cause non-exhaustive case analyses in pre-existing code. In our example, the existence of the Place constructor is problematic for additional reasons. If the user constructs a Term with Place, then the type of the term will not be fully parametric. That is, the type variable α will be bound to some type, t, and the given term would no longer be acceptable as an input to a catamorphism that produces a value of some type other than t. In addition, since Place plays the crucial role as the approximate right inverse of the catamorphism, it is not at all clear what consequences the user-constructed Terms using Place will have for the semantics of cataT.

The solution to this problem is to hide the Place constructor from the programmer. If Place does not appear in the input of cataT, then it will not appear in the output of cataT either. To see why, consider the case of cataT over Abs(f). This is the only situation in which Place is introduced by cataT, and it produces the term:

fa(**fn** x ⇒ cataT(fc,fs,fp,fa) (f (Place x)))

If f ignores its argument, then Place will disappear; otherwise cataT will eventually reduce all occurrences of Place in the input that have type Term – including occurrences in terms like (Place z), which will reduce to z. Therefore, Place will not appear in the output.

The above observation suggests that if the catamorphism operator were a primitive in the programming language, then just one Place constructor would be needed. This special constructor, **Place**, could then be used by every catamorphism, regardless of which type it traverses. The implementation of catamorphisms as primitives would guarantee that **Place** is the right inverse of each catamorphism. The **Place** constructor is completely hidden from programmers in the same way that programmers cannot access closures; it is strictly an internal implementation detail.

The Restriction on Place. The primitive place constructor, **Place**, is only an approximation to the right inverse of cataT. Where does this approximation break down? To answer this question, consider applying eval to the term

Abs(**fn** x ⇒ **case** x **of** Const n ⇒ Const n | z ⇒ Const 0)

This yields

Fun(**fn** y ⇒ eval(**case** (**Place** y) **of** Const n ⇒ Const n
| z ⇒ Const 0))

Not every instance of a Term can be traversed by eval (expressed as a catamorphism), because the function f embedded in Abs(f) does not know how to handle the **Place** constructor, as the example above shows. If f merely "pushes its argument around", things will be ok, but if f attempts to analyze its argument with a case expression to "look inside", as the example above does, things can go wrong.

Our proposed solution to this problem is to statically detect when it occurs and to report a compile-time error. We need to statically detect when a function embedded in a value construction performs a case analysis over its argument. This is not always problematic; it only becomes a difficulty if there exists a catamorphism over this particular construction. Our strategy is to extend the type system to detect such cases. We investigate such a type system further in Section 3. The type-checking algorithm is simple and can be implemented efficiently. It reports an error only for invalid terms.

This restriction is not substantial in our encoding of terms, since any term construction using regular term encoding (i.e., by representing lambda variables explicitly) can be mapped directly into our term representation without the above mentioned problem. The problem appears when we go beyond the typical encoding of terms.

2.1.2 The Parametricity Theorem

As another example of representing terms by structures with embedded functions, we construct the parametricity theorem for a polymorphic function. To understand this section, the reader must be familiar with Wadler's theorems-for-free paper [15].

Any function f of type τ satisfies a parametricity theorem, which is derived directly from the type τ. For first-order functions, this theorem states that any strict polymorphic function is a natural transformation.

Theorem 1 (Parametricity Theorem) *Any strict function $f : \tau$ satisfies $[\![\tau]\!](f,f)$, where:*

$$
\begin{array}{lll}
[\![\text{basic}]\!](r,s) & \rightarrow & r = s \\
[\![\alpha]\!](r,s) & \rightarrow & r = \alpha(s) \\
[\![\forall\alpha.\,\tau]\!](r,s) & \rightarrow & \forall\alpha : [\![\tau]\!](r,s) \\
[\![\tau_1 \times \tau_2]\!](r,s) & \rightarrow & [\![\tau_1]\!](\pi_1(r),\pi_1(s)) \wedge [\![\tau_2]\!](\pi_2(r),\pi_2(s)) \\
[\![\tau_1 \rightarrow \tau_2]\!](r,s) & \rightarrow & \forall x,y : [\![\tau_1]\!](x,y) \Rightarrow [\![\tau_2]\!](r(x),s(y)) \\
[\![T(\tau)]\!](r,s) & \rightarrow & \forall f,x : [\![\tau]\!](f(x),x) \Rightarrow r = \text{map}^T(f)\,s
\end{array}
$$

That is, for each type variable α, we associate a function α (of type $\alpha_1 \rightarrow \alpha_2$, where α_1 and α_2 are instances of α). To avoid the name capture problem, we need to pass an environment through $[\![\tau]\!](r,s)$ that maps type variable names to function names. This environment is not necessary if we use structures with embedded functions.

To illustrate Theorem 1, we derive the parametricity theorem for the list catamorphism, fold : $\forall\alpha.\forall\beta.\,(\alpha\rightarrow\beta\rightarrow\beta)\rightarrow\beta\rightarrow\text{list}(\alpha)\rightarrow\beta$ (The construction is accomplished in four simple steps):

$$
\begin{array}{ll}
\multicolumn{2}{l}{[\![\forall\delta.\forall\varepsilon.\,\delta\rightarrow\varepsilon]\!](r,s)} \\
\rightarrow & \forall\delta,\varepsilon,x,y : [\![\delta]\!](x,y) \Rightarrow [\![\varepsilon]\!](r(x),s(y)) \\
\rightarrow & \forall\delta,\varepsilon,x,y : x = \delta(y) \Rightarrow r(x) = \varepsilon(s(y)) \\
\text{or} & \forall\delta,\varepsilon : r\circ\delta = \varepsilon\circ s
\end{array}
$$

$$
\begin{array}{ll}
\multicolumn{2}{l}{[\![\forall\delta.\forall\varepsilon.\forall\eta.\,\delta\rightarrow\varepsilon\rightarrow\eta]\!](r,s)} \\
\rightarrow & \forall\delta,\varepsilon,\eta,x,y : [\![\delta]\!](x,y) \Rightarrow [\![\varepsilon\rightarrow\eta]\!](r(x),s(y)) \\
\rightarrow & \forall\delta,\varepsilon,\eta,x,y : x = \delta(y) \Rightarrow r(x)\circ\varepsilon = \eta\circ s(y) \\
\text{or} & \forall\delta,\varepsilon,\eta,y,z : r\,(\delta\,y)\,(\varepsilon\,z) = \eta(s\,y\,z)
\end{array}
$$

$$
\begin{array}{ll}
\multicolumn{2}{l}{[\![\forall\alpha.\,\text{list}(\alpha)]\!](r,s)} \\
\rightarrow & \forall\alpha,f,x : f(x) = \alpha(x) \Rightarrow r = \text{map}^{\text{list}}(f)\,s \\
\text{or} & \forall\alpha : r = \text{map}^{\text{list}}(\alpha)\,s
\end{array}
$$

$$
\begin{array}{l}
[\![\forall\alpha.\forall\beta.\,(\alpha\rightarrow\beta\rightarrow\beta)\rightarrow\beta\rightarrow\text{list}(\alpha)\rightarrow\beta]\!](r,s) \\
\rightarrow\ [\![\alpha\rightarrow\beta\rightarrow\beta]\!](\otimes,\oplus) \Rightarrow [\![\beta\rightarrow\text{list}(\alpha)\rightarrow\beta]\!](r\otimes,s\oplus) \\
\rightarrow\ (\alpha\,x)\otimes(\beta\,y) = \beta(x\oplus y) \\
\quad\Rightarrow r\otimes(\beta\,x)\,(\text{map}^{\text{list}}(\alpha)\,y) = \beta(s\oplus x\,y)
\end{array}
$$

That is, fold satisfies the following theorem:

$$
\begin{array}{l}
(\alpha\,x)\otimes(\beta\,y) = \beta(x\oplus y) \\
\quad\Rightarrow \text{fold}(\otimes)\,(\beta\,x)\,(\text{map}^{\text{list}}(\alpha)\,y) = \beta(\text{fold}(\oplus)\,x\,y)
\end{array}
$$

As was pointed out to the authors by Ross Paterson, the parametricity theorem can be easily extended to parametrize over type constructors by associating to each free type constructor $T : * \rightarrow *$ a function T of type $(\alpha_1\rightarrow\alpha_2)\rightarrow T_1(\alpha_1)\rightarrow T_2(\alpha_2)$, where T_1 and T_2 are instances of T. This extension is useful when we want to express laws about operations

```
fun parametricity tp =
All(fn fnc ⇒ cataT(fn (r,s) ⇒ Eq(r,s),
                   fn (a,b) ⇒ fn (r,s) ⇒ And( a(Apl(Pi1,r),Apl(Pi1,s)), b(Apl(Pi2,r),Apl(Pi2,s)) ),
                   fn (a,b) ⇒ fn (r,s) ⇒ All(fn x ⇒ All(fn y ⇒ Impl(a(x,y),b(Apl(r,x),Apl(s,y))))),
                   fn f ⇒ fn (r,s) ⇒ All(fn x ⇒ f(fn (r,s) ⇒ Eq(r,Apl(x,s)))(r,s)),
                   fn (n,a) ⇒ fn (r,s) ⇒ All(fn f ⇒ All(fn x ⇒ Impl( a(Apl(f,x),x), Eq(r,Apl(Map(n,f),s)) ))))
               tp (fnc,fnc))
```

Figure 1: Generation of the Parametricity Law

parametrized by type constructors. For example, the operation cata of type $\forall T \forall \alpha : (T\,\alpha \to \alpha) \to \mu T \to \alpha$ satisfies the property $\phi \circ (T\,\alpha) = \alpha \circ \psi \;\Rightarrow\; (\text{cata}\,\phi) \circ (Y\,T) = \alpha \circ (\text{cata}\,\psi)$, where the functions cata here may be parameterized over different type constructors.

The construction of the predicate of Theorem 1 requires some variable plumbing when we introduce universal quantification (to avoid name capture, etc.). In addition, a function is associated to each type variable and this binding should be carried through the whole construction. We can avoid the variable binding problems by using structures with embedded functions to capture universal quantification. Our algorithm takes a type construction of type T and returns a predicate of type E:

```
datatype T = Basic | Prod of T × T | Arrow of T × T
           | Univ of T → T | Def of string × T
datatype E = Pi1 | Pi2 | Apl of E × E | Eq of E × E
           | And of E × E | Impl of E × E
           | All of E → E | Map of string × E
```

Following the same routine as before, (i.e., by adding the new constructor Place to type T, etc.), we arrive at the following catamorphism over types:

```
fun cataT(b,p,a,u,d) Basic       = b
  | cataT(b,p,a,u,d) (Prod(x,y))  = p( cataT(b,p,a,u,d) x,
                                       cataT(b,p,a,u,d) y )
  | cataT(b,p,a,u,d) (Arrow(x,y)) = a( cataT(b,p,a,u,d) x,
                                       cataT(b,p,a,u,d) y )
  | cataT(b,p,a,u,d) (Univ f)     = u(cataT(b,p,a,u,d) o f o Place)
  | cataT(b,p,a,u,d) (Def(n,x))   = d( n, cataT(b,p,a,u,d) x )
  | cataT(b,p,a,u,d) (Place x)    = x
```

The algorithm that generates the predicate of the parametricity theorem for a function fnc of type tp is the higher-order catamorphism presented in Figure 1. When this function operates over Univ(g), it lifts g:T → T into

$$\text{f: } (\text{E} \times \text{E} \to \text{E}) \to (\text{E} \times \text{E} \to \text{E})$$

The input to f should be **fn** (r,s) ⇒ Eq(r,Apl(x,s)), that is, it should be the rule for handling the type variable x (the second rule in Theorem 1). Notice that there is no need to use a *gensym* function to generate new variable names since the variable scoping is handled implicitly by the execution engine of SML in which this function is expressed. The Paterson-Meijer-Hutton approach would have failed to capture this function, since there is no obvious function E → T that is a right inverse of the parametricity function (described in Figure 1).

2.2 Circular Structures

In this section we present two more example of datatypes with embedded functions. We are interested in expressing computations over infinite structures that always terminate. Catamorphisms over finite structures have this property. We would like to extend this property to graph-like data structures. To do this, we need to represent such structures by finite algebraic datatypes. One way to do this is to use embedded functions.

2.2.1 Circular Lists

Lazy functional languages support circular data structures. For example, the following Haskell definition

```
circ = 0:1:circ
```

constructs the infinite list $0 : 1 : 0 : 1 : \cdots$ by using a cycle. One way to construct infinite lists in SML is to use lazy lists (also known as streams), which use an explicit "thunk" for the tail of a list to obtain tail laziness [11]:

```
datatype α Clist = Nil | Cons of α × (unit → α Clist)
```

For example, the list circ=0:1:circ is expressed as follows:

```
let fun circ() = Cons(0,fn () ⇒ Cons(1,circ)) in circ() end
```

Even though many circular structures can be defined this way, many operations over them need to explicitly carry a list of visited nodes in order to avoid falling into an infinite loop. Identifying which nodes have been visited is, however, also problematic, since there is no pointer equality in the pure functional subset of SML. Furthermore, the construction of these circular structures requires the use of recursive function definitions (such as the circ above), which we want to avoid when we define catamorphisms (after all, a catamorphism is supposed to be an alternative to recursion).

A better definition for circular lists is obtained by reconsidering the previous Haskell definition, which is equivalent to Y(**fn** x ⇒ 0:1:x), where Y is the fixpoint operator of type $(\alpha \to \alpha) \to \alpha$ satisfying $Y\,f = f(Y\,f)$. This expression too does not terminate in strict languages such as SML. One solution is to suspend the application of Y and unroll the fixpoint explicitly during an operation only when this is unavoidable, as is done implicitly in lazy languages.

We can accomplish a similar effect in strict languages by encapsulating the recursion inside a value constructor, Rec, with a type similar to the type (α list → α list) → α list of the Y combinator. This leads to the following type definition for circular lists:

```
datatype α Clist =
    Nil
  | Cons of α × α Clist
  | Rec of α Clist → α Clist
```

We can now express the list $0 : 1 : 0 : 1 : \cdots$ as

```
Rec(fn x ⇒ Cons(0,Cons(1,x)))
```

Functions that manipulate such structures will need to unroll the implicit fixpoint in Rec explicitly. The following are two such functions:

```
fun head(Cons(a,r)) = a
  | head(Rec f)     = head(f(Rec f))
fun nth(Cons(a,r),0) = a
  | nth(Cons(a,r),n) = nth(r,n-1)
  | nth(Rec f,n)     = nth(f(Rec f),n)
```

For example, if circ = Rec(**fn** x ⇒ Cons(0,Cons(1,x))), then nth(circ,100) = 0 and nth(circ,101) = 1.

To express catamorphisms cataC(b,f,g) over Clist, we add an extra type variable β and a constructor Place to Clist:

```
datatype (α,β) Clist =
    Nil
  | Cons of α × (α,β) Clist
  | Rec of (α,β) Clist → (α,β) Clist
  | Place of β
```

Then cataC(b,f,g) is:

```
fun cataC(b,f,g) Nil          = b
  | cataC(b,f,g) (Cons(a,r))  = f( a, cataC(b,f,g) r )
  | cataC(b,f,g) (Rec h)      = g( cataC(b,f,g) ∘ h ∘ Place )
  | cataC(b,f,g) (Place x)    = x
```

Notice that cataC does not unroll the fixpoint in Rec h. It does not need to. Instead, it lifts h into a function of type $\beta \rightarrow \beta$ and it is up to g to decide what to do with it. To illustrate this, consider the map mapC(h) over circular lists:

```
fun mapC(h) = cataC( Nil, fn (a,r) ⇒ Cons(h(a),r), Rec )
```

In this case, g = Rec, and g will tie a new "knot" from the lifted function h, which results in a new circular list. Working through the example,

```
mapC(fn x ⇒ x+1) (Rec(fn x ⇒ Cons(0,Cons(1,x))))
```

will compute a circular list equivalent to

```
Rec(fn x ⇒ Cons(1,Cons(2,x)))
```

The Haskell definition x = 1:(map(1+) x) that computes the infinite list $1 : 2 : 3 : 4 : \cdots$ is represented by the circular list Rec(**fn** x ⇒ Cons(1,mapC(**fn** y ⇒ y+1) x)). For example,

```
nth( Rec(fn x ⇒ Cons(1,mapC(fn y ⇒ y+1) x)), 100 )
= 101
```

But consider the following evaluation:

```
mapC(fn z ⇒ 2*z)(Rec(fn x ⇒ Cons(1,mapC(fn y ⇒ y+1) x)))
= Rec(fn x ⇒ mapC(fn z ⇒ 2*z)
                  (Cons(1,mapC(fn y ⇒ y+1)(Place x))))
= Rec(fn x ⇒ Cons(2,mapC(fn z ⇒ 2*z)
                         (mapC(fn y ⇒ y+1)(Place x))))
= Rec(fn x ⇒ Cons(2,mapC(fn z ⇒ 2*z) x))
```

which represents the infinite list $2 : 4 : 8 : 16 : \cdots$. This result is incorrect. We should have computed:

```
Rec(fn x ⇒ Cons(2,mapC(fn z ⇒ z+2) x))
```

which represents the infinite list $2 : 4 : 6 : 8 : 10 : \cdots$. But why did we get this error? The problem is that the Place constructor in this example was intended to be used for the outer occurrence of mapC, not the inner. Instead it cancelled the inner occurrence of mapC. If we had followed the Paterson-Meijer-Hutton approach, we would have used mapC(**fn** z ⇒ z/2), the right inverse of mapC(**fn** z ⇒ 2*z), instead of Place. In that case, we would have derived the correct result. In a situation like this, where a function is invertible, the Paterson-Meijer-Hutton approach clearly wins over ours, since it is more expressive. Even though nth works fine over the above construction, the problem is that this construction cannot be traversed by another cataC in our model because of the case analysis implied by the application of mapC to the argument x. Fortunately, cases like this are automatically discovered by the type inference system to be discussed in Section 3.

This restriction is not as severe as it might first appear. For example, all planar graphs defined in the next section can be encoded into our representation without any problem. This is because graph cycles are captured by a construction of the form Rec(**fn** x ⇒ f(x)). If we want to construct a backward edge from a graph node inside the graph construction f(x) to the beginning of f(x), we simply reference x. That is, this construction does not need to analyze or traverse x in f(x) any time. Problems may occur when we want to capture structures whose patterns of recursion are not constant (such as the list of natural numbers).

2.2.2 Graphs

Graphs can be represented in a manner similar to the way in which circular lists are represented. Such a representation allows terminating computations over graphs – such as the computation of the spanning tree of a graph – to be expressed as catamorphisms. The most common way to represent graphs in a functional language is to use a vector of adjacency lists. But, this approach is not really different from using pointers in a procedural language, since permits ad-hoc constructions and manipulations of graphs. In [13] a graph type is defined as:

```
datatype α graph = Graph of α → α list
```

Here a graph consists of a function that computes the successors of each node. This definition requires special care while programming to guarantee program termination.

Our graph representation is based on the idea of using embedded functions in a manner similar to the way we did it for circular lists. We start with a datatype that can represent trees, which have nodes supporting arbitrary branching levels. We call such a tree a rose_tree and define it by

```
datatype α rose_tree = Node of α × α rose_tree list
```

Now, we think of a graph as a generalization of rose trees with cycles and sharing:

```
datatype α graph =
    Node of α × α graph list
  | Rec of α graph → α graph
  | Share of (α graph → α graph) × α graph
```

Here, Rec plays the role of the Y combinator in expressing cycles, while Share plays the role of a function application.

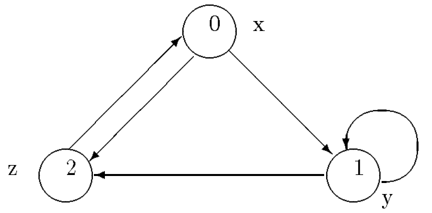

Figure 2: Graph with three nodes 0, 1, 2 and five edges

That is, Share(f,e) is unrolled as f e. All free occurrences of x in u in the term Share(**fn** x ⇒ u, e) are bound to e, i.e., all x in u share the same subgraph e.

The function

```
Rec(fn x ⇒ Share(fn z ⇒ Node(0,[z,Rec(fn y ⇒ Node(1,[y,z]))]),
                 Node(2,[x])))
```

for example, describes the graph in Figure 2. By following the same routine as for circular lists, i.e., by adding the new constructor Place, etc., the graph catamorphism cataG becomes:

```
fun cataG(f,g,k) (Node(a,r))    = f( a, map(cataG(f,g,k)) r )
 |  cataG(f,g,k) (Rec(h))       = g( cataG(f,g,k) o h o Place )
 |  cataG(f,g,k) (Share(h,n))   = k( cataG(f,g,k) o h o Place,
                                     cataG(f,g,k) n )
 |  cataG(f,g,k) (Place x)      = x
```

The following are examples of graph manipulations that use cataG:

```
val listify = cataG( fn (a,r) ⇒ a::(flatten r), fn f ⇒ f [ ],
                     fn (f,r) ⇒ f r )
val sum = cataG( fn (a,r) ⇒ cata(op +) r a, fn f ⇒ f 0,
                 fn (f,r) ⇒ f r )
fun mapG(g) = cataG( fn (a,r) ⇒ Node(g a,r), Rec, Share )
```

listify flattens a graph into a list, sum computes the sum of all node values, and mapG is the map over a graph.

The following is another datatype for graphs. Here we have merged the roles of the Share and the Rec constructors by using a list in the domain of Rec:

```
datatype α graph =
    Node of α × α graph list
  | Rec of int × (α graph list → α graph list)
```

Under such a graph representation, the graph in Figure 2 is constructed by

```
Rec( 3, fn [x,y,z] ⇒ [Node(0,[y,z]),Node(1,[y,z]),Node(2,[x])]
        | x ⇒ error )
```

In general, any set of n mutually recursive Haskell definitions of the form $x_i = f_i(x_1, \ldots, x_n)$, $1 \leq i \leq n$, can be represented by

Rec(n, **fn** $[x_1, \ldots, x_n] \Rightarrow [f_1(x_1, \ldots, x_n), \ldots, f_n(x_1, \ldots, x_n)]$
    | x ⇒ error)

The interpretation of Rec(n,f) is hd(Y f). For practical reasons, both the input and the output list of f must have the same size, n, in order for the closure Y f to execute with no run-time error. By following the same routine as before, i.e., by adding the new constructor Place, etc., the graph catamorphism becomes:

```
fun cataG(fn,fr) (Node(a,r)) = fn( a, map(cataG(fn,fr)) r )
 |  cataG(fn,fr) (Rec(m,f))
          = fr( m, map(cataG(fn,fr)) o f o (map Place) )
 |  cataG(fn,fr) (Place x) = x
```

For example, the following computes the adjacency list of a graph:

```
cataG( fn (a,r) ⇒ [(a,map(#1) (flatten r))],
       fn (n,f) ⇒ flatten(f(f(ncopies n [ ]))) )
```

where ncopies n a creates a list of n copies of a. If we had flatten(f(ncopies n [ ])) in the second parameter of cataG, we would have obtained an empty adjacency list for each node.

In each of the examples above, it was necessary to use the trick of extending the datatype definition with an additional constructor Place, adding to the datatype a type variable to hold the type of the result of a catamorphism, and writing the catamorphism function by hand. Unfortunately, in none of these cases is there a guarantee that the programs written obeyed the restriction described in Section 2.1.1.

In the next section we define a language in which our trick is implemented implicitly. That is, the Place constructor and the catamorphism function become primitives of the language, but because Place is hidden, it is impossible for the user to construct programs that use Place. The resulting language also supports a type system that enforces the above mentioned restriction. Type inference rules for this type system are also presented.

3 The Formal Framework

In this section we define a language that allows the definition of new datatypes and implicitly supplies the trick we have used in the examples of Section 2. The expression sub-language includes the catamorphism operator for any datatype as a primitive. Syntactically, the user writes cata T, where T is the name of a user-defined datatype. The semantics of the catamorphism primitive is given as an implicit case analysis over the datatype traversed by the catamorphism and need not be supplied by the user. This semantics accommodates the Place constructor only as an internal implementation detail.

In the interest of simplicity, the language does not use explicit value constructors, as do most functional languages. Instead, it uses binary sum and product types. The resulting notation renders the underlying theory easier to explain because fewer rules are needed to express our algorithms, but, unfortunately, also makes programs hard to understand. For clarity, we will explicitly describe the correspondence between functional languages and our language as we proceed through this section.

3.1 Terms

Terms e in the language are generated by the following grammar:

$$\begin{array}{lrl} \text{(term)} & e ::= & x \mid () \mid e\,e \mid \lambda x.\,e \mid (e,e) \mid \mathbf{in}^T e \mid \mathbf{inL} \mid \mathbf{inR} \\ & & \mid \text{cata}^T e \mid \lambda\mathbf{in}^T x.\,e \mid \lambda(x,x).\,e \\ & & \mid \lambda\mathbf{inL}\,x.\,e \;[]\; \mathbf{inR}\,x.\,e \end{array}$$

where x denotes a variable. The term $\mathbf{in}^T$ is the value constructor for the recursive type associated with the type definition T (to be explained in detail later). The constructors **inL** and **inR** are the left and right injectors of the sum type.

The function cata^T is the catamorphism operator for T. The *in-abstraction* $\lambda\mathbf{in}^T x.\, e$ is defined by $(\lambda\mathbf{in}^T x.\, e)\,(\mathbf{in}^T u) = (\lambda x.\, e)\, u$, and the *pair-abstraction* $\lambda(x,y).\, e$ is defined by:

$$(\lambda(x,y).\, f(x,y))(e_1,e_2) \;=\; f(e_1,e_2)$$

The *sum-abstraction* $\lambda\mathbf{inL}\, x_1.\, e_1 \,[\!]\, \mathbf{inR}\, x_2.\, e_2$, when applied to $\mathbf{inL}\, u$, computes $(\lambda x_1.\, e_1)\, u$ and, when applied to $\mathbf{inR}\, u$, computes $(\lambda x_2.\, e_2)\, u$.

As explained above, our language does not contain value constructors. The value constructors of a type can be defined in terms of other operators and these definitions can be generated automatically. For example, for SML lists defined as:

$$\textbf{datatype}\ \alpha\ \mathsf{list} = \mathsf{Nil}\ \mid \mathsf{Cons}\ \textbf{of}\ \alpha\ \times \alpha\ \mathsf{list}$$

the value constructors Nil and Cons could be defined as:

$$\mathsf{Nil} = \mathbf{in}^{\mathsf{List}}(\mathbf{inL}()) \qquad\qquad \mathsf{Cons}(a,r) = \mathbf{in}^{\mathsf{List}}(\mathbf{inR}(a,r))$$

Traditional languages use case statements to decompose values. Our term language can capture any case analysis over a value construction by using sum- and in-abstractions. For example,

$$\textbf{case}\ e\ \textbf{of}\ \mathsf{Nil}\ \Rightarrow e_1\ \mid \mathsf{Cons}(a,r)\ \Rightarrow e_2$$

can be expressed by the following composition of operators:

$$(\lambda\mathbf{in}^{\mathsf{List}} x.\, (\lambda\mathbf{inL}\, y.\, e_1 \,[\!]\, \mathbf{inR}\, y.\, (\lambda(a,r).\, e_2)\, y)\, x)\, e$$

3.2 Types

Our types are generated by the following grammar:

$$\begin{array}{lrcl}
\text{(type-definition)} & T & ::= & \Lambda(x,x).T \mid \tau \\
\text{(type)} & \tau & ::= & x \mid () \mid \tau\times\tau \mid \tau+\tau \mid \tau\to\tau \mid E \\
\text{(type-use)} & E & ::= & \mu^\omega\, T \mid E\,(\tau,\tau) \\
\text{(tag)} & \omega & ::= & x \mid \text{cased} \mid \text{folded}
\end{array}$$

A type definition consists of a number of type abstractions followed by a type. Each type abstraction $\Lambda(x,y).\,\tau$ introduces two type variables: a positive $(+)$ x and a negative $(-)$ y. A positive (resp., negative) variable should only appear in a positive (resp., negative) position in a type. This condition is implicitly checked by the rules guaranteeing well-formedness of types, given in Figure 3. Under this condition, α and δ appear in a positive position in the type $(\alpha\to\beta)\to\gamma\to\delta$, while β and γ in a negative.

The rules in Figure 3 check the well-formedness of types using the following definitions:

$$\begin{array}{lrcl}
\text{(kind)} & k & ::= & x \mid + \mid - \\
\text{(kind-assignment)} & \rho & ::= & \{\} \mid \rho\{x:k\}
\end{array}$$

A type $T = \Lambda(x_0,x_0').\ldots.\Lambda(x_n,x_n').\tau$ must satisfy $\rho\{x_0 : +, x_0' : -, \ldots, x_n : +, x_n' : -\} \vdash \tau :: +$. A type definition T must have at least one type abstraction $\Lambda(x_0,x_0')$, which is used when constructing the fixpoint $\mu^\omega\, T$ of T. The ω tag in μ^ω is used during type-checking and is hidden from programmers. The fixpoint type constructor μ^ω is a primitive and satisfies the equation

$$\begin{array}{l}
\mu^\omega\, T\,(\tau_1,\tau_1')\cdots(\tau_n,\tau_n') \\
\quad = T\,((\mu^\omega\, T\,(\tau_1,\tau_1')\cdots(\tau_n,\tau_n')),(\mu^\omega\, T\,(\tau_1',\tau_1)\cdots(\tau_n',\tau_n))) \\
\qquad (\tau_1,\tau_1')\cdots(\tau_n,\tau_n')
\end{array}$$

That is, $\mu^\omega\, T$ is the fixpoint of T, where both the arguments of the first abstraction of T are fixed to $\mu^\omega\, T$. We will see later that any type definition T that meets the well-formedness criteria of Figure 3 is a functor that is covariant in its positive arguments and contravariant in its negative arguments, that is,

$$\begin{array}{l}
(T\,(f_1,f_1')\cdots(f_n,f_n'))\circ(T\,(g_1,g_1')\cdots(g_n,g_n')) \\
\quad = T\,(f_1\circ g_1, g_1'\circ f_1')\cdots(f_n\circ g_n, g_n'\circ f_n')
\end{array}$$

The following are examples of type definitions:

$$\begin{array}{lcl}
\text{Bool} & = & \Lambda(x,y).\,()+() \\
\text{Nat} & = & \Lambda(x,y).\,()+x \\
\text{List} & = & \Lambda(x,y).\Lambda(\alpha,\alpha').\,()+\alpha\times x \\
\text{Rose_tree} & = & \Lambda(x,y).\Lambda(\alpha,\alpha').\Lambda(\beta,\beta'). \\
 & & \quad ()+(\alpha+\beta\times(\mu^\omega\text{List}\,(x,y))) \\
\text{Clist} & = & \Lambda(x,y).\Lambda(\alpha,\alpha').\,()+(\alpha\times x+(y\to x)) \\
\text{Ctype} & = & \Lambda(x,y).\Lambda(\alpha,\alpha').\,(\alpha'\to\alpha)+(y\to x)
\end{array}$$

The inductive list type definition

$$\textbf{datatype}\ \mathsf{list}(\alpha) = \mathsf{Nil}\ \mid \mathsf{Cons}\ \textbf{of}\ \alpha\times\mathsf{list}(\alpha),$$

for instance, is derived by applying the fixpoint operator to List to obtain

$$\text{list}(\alpha) \;=\; \mu^\omega\,\text{List}\,(\alpha,\alpha)$$

Note that types that do not include embedded functions – such as the familiar List, Nat and Bool – make no mention of their negative type variables.

3.3 The Semantics of Catamorphism

The semantics of the catamorphism cata^T over a type definition T can be given as an implicit case analysis over T. A type definition $T = \Lambda(x_0,x_0')\ldots\Lambda(x_n,x_n').\ \tau$ is associated with a combinator, which we will prove is a functor. The type mapping part of the functor is the polymorphic type T itself. The function mapping part of the functor is defined by $T\,(f_0,f_0')\cdots(f_n,f_n') = \mathcal{M}[\![\tau]\!]$, where each function f_i (resp., f_i') is associated with the type variable x_i (resp., x_i'). The term $\mathcal{M}[\![\tau]\!]$ is derived by a case analysis over the type τ:

$$\begin{array}{lcl}
\mathcal{M}[\![x_i]\!] & = & f_i \\
\mathcal{M}[\![()]\!] & = & \lambda x.x \\
\mathcal{M}[\![\tau_1\times\tau_2]\!] & = & \lambda(x,y).\,(\mathcal{M}[\![\tau_1]\!]\,x, \mathcal{M}[\![\tau_2]\!]\,y) \\
\mathcal{M}[\![\tau_1+\tau_2]\!] & = & \lambda\mathbf{inL}\,x.\,\mathbf{inL}(\mathcal{M}[\![\tau_1]\!]\,x) \\
 & & [\!]\,\mathbf{inR}\,y.\,\mathbf{inR}(\mathcal{M}[\![\tau_2]\!]\,y) \\
\mathcal{M}[\![\tau_1\to\tau_2]\!] & = & \lambda h.\,\lambda x.\,\mathcal{M}[\![\tau_2]\!](h(\mathcal{M}[\![\tau_1]\!](x))) \\
\multicolumn{3}{l}{\mathcal{M}[\![\mu^\omega\, S\,(\tau_1,\tau_1')\cdots(\tau_m,\tau_m')]\!]} \\
\multicolumn{3}{l}{\quad = \text{map}^S\,(\mathcal{M}[\![\tau_1]\!],\mathcal{M}[\![\tau_1']\!])\cdots(\mathcal{M}[\![\tau_m]\!],\mathcal{M}[\![\tau_m']\!])}
\end{array}$$

where for the combinator $S = \Lambda(x_0,x_0')\ldots\Lambda(x_m,x_m').\,\tau$ we have

$$\begin{array}{l}
(\text{map}^S\,(f_1,f_1')\cdots(f_m,f_m'))\circ\mathbf{in}^S \\
\quad = \mathbf{in}^S\circ(S\,(\text{map}^S\,(f_1,f_1')\cdots(f_m,f_m'), \\
\qquad\qquad \text{map}^S\,(f_1',f_1)\cdots(f_m',f_m)) \\
\qquad\quad (f_1,f_1')\cdots(f_m,f_m'))
\end{array}$$

$$\rho \vdash () :: k \qquad \rho \vdash x :: \rho(x) \qquad \frac{\rho \vdash \tau_1 :: k, \;\; \rho \vdash \tau_2 :: k}{\rho \vdash \tau_1 \times \tau_2 :: k} \qquad \frac{\rho \vdash \tau_1 :: k, \;\; \rho \vdash \tau_2 :: k}{\rho \vdash \tau_1 + \tau_2 :: k}$$

$$\frac{\rho \vdash \tau_1 :: \neg k, \;\; \rho \vdash \tau_2 :: k}{\rho \vdash \tau_1 \to \tau_2 :: k} \qquad \frac{\begin{array}{c}\rho \vdash \tau_i :: k, \;\; \rho \vdash \tau_i' :: \neg k, \;\; T = \Lambda(x_0, x_0')\ldots\Lambda(x_n, x_n').\tau, \\ \rho\{x_0 : +, x_0' : -, \ldots, x_n : +, x_n' : -\} \vdash \tau :: +\end{array}}{\rho \vdash \mu^\omega\, T\,(\tau_1, \tau_1')\cdots(\tau_n, \tau_n') :: k}$$

Figure 3: Well-formedness of Types (Definition: $\neg(+) = -$ and $\neg(-) = +$)

Theorem 2 *For $T = \Lambda(x_0, x_0')\ldots\Lambda(x_n, x_n').\tau$, the combinator $T\,(f_0, f_0')\cdots(f_n, f_n') = \mathcal{M}[\![\tau]\!]$ is a functor, i.e., it satisfies*

$$\begin{array}{l} T\,(\lambda x.x, \lambda x.x)\cdots(\lambda x.x, \lambda x.x) \;=\; \lambda x.x \\ (T\,(f_0, f_0')\cdots(f_n, f_n')) \circ (T\,(g_0, g_0')\cdots(g_n, g_n')) \\ \quad = \; T\,(f_0 \circ g_0, g_0' \circ f_0')\cdots(f_n \circ g_n, g_n' \circ f_n') \end{array}$$

This theorem can be proved by induction over the structure of τ in $\mathcal{M}[\![\tau]\!]$. Let $\mathcal{M}_1[\![\tau]\!] = T\,(f_0, f_0')\cdots(f_n, f_n')$, $\mathcal{M}_2[\![\tau]\!] = T\,(g_0, g_0')\cdots(g_n, g_n')$, and $\mathcal{M}_3[\![\tau]\!] = T\,(f_0 \circ g_0, g_0' \circ f_0')\cdots(f_n \circ g_n, g_n' \circ f_n')$. It suffices to prove that, for any type τ, if $\{\} \vdash \tau :: +$, then $\mathcal{M}_3[\![\tau]\!] = \mathcal{M}_1[\![\tau]\!] \circ \mathcal{M}_2[\![\tau]\!]$, and otherwise $\mathcal{M}_3[\![\tau]\!] = \mathcal{M}_2[\![\tau]\!] \circ \mathcal{M}_1[\![\tau]\!]$. For illustration purposes, we present only one case of this proof where $\tau = \tau_1 \to \tau_2$ and τ is positive; the proofs in the remaining cases are similar.

$$\begin{array}{ll} & \mathcal{M}_1[\![\tau]\!] \circ \mathcal{M}_2[\![\tau]\!] \\ = & (\lambda h.\, \lambda x.\, \mathcal{M}_1[\![\tau_2]\!](h(\mathcal{M}_1[\![\tau_1]\!](x)))) \\ & \circ(\lambda h.\, \lambda x.\, \mathcal{M}_2[\![\tau_2]\!](h(\mathcal{M}_2[\![\tau_1]\!](x)))) \\ = & \lambda h.\, \lambda x.\, (\mathcal{M}_1[\![\tau_2]\!](\mathcal{M}_2[\![\tau_2]\!](h(\mathcal{M}_2[\![\tau_1]\!](\mathcal{M}_1[\![\tau_1]\!](x)))))) \\ = & \lambda h.\, \lambda x.\, (\mathcal{M}_3[\![\tau_2]\!](h(\mathcal{M}_3[\![\tau_1]\!](x)))) \\ = & \mathcal{M}_3[\![\tau]\!] \end{array} \qquad \Box$$

The catamorphism over any datatype T is defined in terms of the combinator $\mathcal{E}^T$, which is equal to the functor T with all but its first two arguments fixed at the identity:

$$\mathcal{E}^T\,(f^+, f^-) \;=\; T\,(f^+, f^-)\,(\lambda x.x, \lambda x.x)\cdots(\lambda x.x, \lambda x.x)$$

According to Theorem 2, $\mathcal{E}^T$ satisfies the law

$$\mathcal{E}^T\,(f^+, f^-) \circ \mathcal{E}^T\,(g^+, g^-) = \mathcal{E}^T\,(f^+ \circ g^+, g^- \circ f^-)$$

According to our previous discussion, to define the catamorphism over T, we need to extend T with a new value constructor Place and a new type variable β. We get

$$T'\,\beta\,(\alpha_0, \alpha_0')\cdots(\alpha_n, \alpha_n') \;=\; (T\,(\alpha_0, \alpha_0')\cdots(\alpha_n, \alpha_n')) \,+\, \beta$$

Here Place is equal to $\mathbf{in}^{T'} \circ \mathbf{inR}$. The resulting catamorphism over T is

$$\begin{array}{lcl} (\mathrm{cata}^T\,\phi) \circ \mathbf{in}^{T'} & = & \lambda \mathbf{inL}\, x.\, \phi(\mathcal{E}^T\,(\mathrm{cata}^T\,\phi, \mathbf{in}^{T'} \circ \mathbf{inR})\,x) \\ & & [\!]\, \mathbf{inR}\, y.\, y \end{array}$$

But here the Place constructor is transparent to programmers. To hide it, we need to introduce the special term **Place**.

Definition 1 (Catamorphism) *The catamorphism for a type T is $\mathrm{cata}^T : (T(\alpha, \alpha) \to \alpha) \to \mu^\omega\, T \to \alpha$, defined as follows:*

$$\begin{array}{lcl} \mathrm{cata}^T\,\phi\,(\mathbf{in}^T x) & = & \phi(\mathcal{E}^T\,(\mathrm{cata}^T\,\phi, \mathbf{Place})\,x) \\ \mathrm{cata}^T\,\phi\,(\mathbf{Place}\,x) & = & x \end{array}$$

According to this definition, the catamorphism for circular lists is $\mathrm{cata}^{\mathrm{Clist}}\,\phi$, given by

$$\begin{array}{l} \mathrm{cata}^{\mathrm{Clist}}\,\phi\,(\mathbf{in}^{\mathrm{Clist}}\,x) \;= \\ \quad \phi((\lambda \mathbf{inL}\, y.\, \mathbf{inL}\, y \\ \qquad [\!]\, \mathbf{inR}\, y.\, \mathbf{inR}((\lambda \mathbf{inL}\, z.\, \mathbf{inL}((\lambda(a, r).\,(a, \mathrm{cata}^{\mathrm{Clist}}\,\phi\, r))\,z) \\ \qquad\qquad [\!]\, \mathbf{inR} h.\, \mathbf{inR}(\lambda w.\, \mathrm{cata}^{\mathrm{Clist}}\,\phi\,(h(\mathbf{Place}\,w))) \\ \qquad\quad)\,y))\,x) \\ \mathrm{cata}^{\mathrm{Clist}}\,\phi\,(\mathbf{Place}\,x) \;=\; x \end{array}$$

If we had expressed the first case of this definition in a functional language with value constructors and pattern matching, we would have

$$\begin{array}{l} \mathrm{cata}^{\mathrm{Clist}}\,\phi\,(\mathbf{in}^{\mathrm{Clist}}\,x) \\ \quad = \phi(\mathbf{case}\;\; x\; \mathbf{of} \\ \qquad\quad \mathrm{Nil}' \;\Rightarrow\; \mathrm{Nil}' \\ \qquad\; |\; \mathrm{Cons}'(a, r) \;\Rightarrow\; \mathrm{Cons}'(a, \mathrm{cata}^{\mathrm{Clist}}\,\phi\, r) \\ \qquad\; |\; \mathrm{Rec}'(f) \;\Rightarrow\; \mathrm{Rec}'((\mathrm{cata}^{\mathrm{Clist}}\,\phi) \circ f \circ \mathbf{Place})) \end{array}$$

where $\mathrm{Nil}' = \mathbf{inL}()$, $\mathrm{Cons}'(a, r) = \mathbf{inL}(\mathbf{inR}(a, r))$, and $\mathrm{Rec}'(f) = \mathbf{inR}(\mathbf{inR}(f))$. For example, the following program computes $\mathsf{mapC(g)}$, the map over Clist:

$$\begin{array}{l} \mathrm{cata}^{\mathrm{Clist}}\,(\lambda \mathbf{inL}\, y.\, \mathbf{inL}\, y \\ \qquad [\!]\, \mathbf{inR}\, y.\, \mathbf{inR}((\lambda \mathbf{inL}\, z.\, \mathbf{inL}((\lambda(a, r).\,(g\,a, r))\,z) \\ \qquad\qquad [\!]\, \mathbf{inR} h.\, \mathbf{inR}(h))\,y)) \end{array}$$

Meijer and Hutton [8] define a catamorphism cata^T in conjunction with its dual, the anamorphism ana^T, as follows:

$$\begin{array}{lcl} \mathrm{cata}^T\,\phi\,\psi\,(\mathbf{in}^T x) & = & \phi(\mathcal{E}^T\,(\mathrm{cata}^T\,\phi\,\psi, \mathrm{ana}^T\,\phi\,\psi)\,x) \\ \mathrm{ana}^T\,\phi\,\psi\,x & = & \mathbf{in}^T(\mathcal{E}^T\,(\mathrm{ana}^T\,\phi\,\psi, \mathrm{cata}^T\,\phi\,\psi)\,(\psi\,x)) \end{array}$$

That is, both cata^T and ana^T should take two functions, ϕ and ψ. The first, ϕ, is used in the catamorphism and the other, ψ, is be used in the anamorphism. This dual pair of cata-ana should satisfy the law $(\mathrm{cata}^T\,\phi\,\psi) \circ (\mathrm{ana}^T\,\phi\,\psi) = \mathrm{id}$ in order to be useful. This implies that $\phi \circ \psi = \mathrm{id}$.

To complete the explication of the relationship between our work and that of Meijer and Hutton's, we show that the notion of anamorphism is dual of that of catamorphism.

Definition 2 (Anamorphism) *The anamorphism for a type T is* $\text{ana}^T : (\alpha \to T(\alpha,\alpha)) \to \alpha \to \mu^\omega\, T$, *defined as follows:*

$$\text{ana}^T\, \psi\, x \quad = \quad \mathbf{in}^T(\mathcal{E}^T\, (\text{ana}^T\, \psi, \mathbf{Place})\, (\psi x))$$

One of the advantages of using catamorphisms instead of general recursion is that they satisfy nice properties. In particular, they satisfy an important property known as the *fusion law*, which can be used for fusing any strict function with a catamorphism to yield another catamorphism. The fusion law for a catamorphism of type T is given by the following theorem.

Theorem 3 (Fusion Law) *For any strict function g:*

$$\frac{g \circ \phi \quad = \quad \psi \circ \mathcal{E}^T\,(g, \text{id})}{g \circ \text{cata}^T\, \phi \quad = \quad \text{cata}^T\, \psi}$$

Proof: Since **Place** is internal to each cata^T, we will consider only the case where the input is $\mathbf{in}^T$ (external uses of **Place** are detected and ruled out by our type system). We have

$$\begin{array}{rcl} g \circ \text{cata}^T\, \phi \circ \mathbf{in}^T & = & g \circ \phi \circ \mathcal{E}^T\,(\text{cata}^T\, \phi, \mathbf{Place}) \\ & = & \psi \circ \mathcal{E}^T\,(g, \text{id}) \circ \mathcal{E}^T\,(\text{cata}^T\, \phi, \mathbf{Place}) \\ & = & \psi \circ \mathcal{E}^T\,(g \circ \text{cata}^T\, \phi, \mathbf{Place} \circ \text{id}) \\ & = & \psi \circ \mathcal{E}^T\,(\text{cata}^T\, \psi, \mathbf{Place}) \\ & = & \text{cata}^T\, \psi \circ \mathbf{in}^T \end{array} \qquad \Box$$

3.4 Type-checking

The grammar for the term language gives syntactic rules for valid term constructions, but not all such terms have meaning. Traditionally, a type system is used to discover invalid terms by attempting to assign types to terms. Here we will use the type system to distinguish both the ill-typed terms and terms with illegal uses of catamorphisms. Since our language support polymorphic data types, we will need a Hindley-Milner style type-inference algorithm. Here we will only give the typing rules for the type-inference system.

Figure 4 presents the typing rules for our λ-calculus. We add to the usual rules the rules (CATA), (IN), and (OUT). To simplify these rules, we assume that a type definition T has only one type abstraction, i.e., we assume that T contains one positive and one negative variable. If we ignore the ω's in the Rule (IN), then the type of $\mathbf{in}^T\, e$ is the fixpoint of the functor T and the type of e is T with both its positive and negative arguments fixed to the fixpoint of T. In addition, Rule (IN) propagates the ω flag only to the negative part of T. Rule (OUT) is in some sense the opposite of Rule (IN) since $\lambda\mathbf{in}^T\, x.\, e$ is a function from the fixpoint of T to the type of e. Rule (OUT) also sets the ω flag of the negative part of T to *cased*. Rule (CATA) sets the ω tag of μ^ω to *folded*. If a term of type $\mu^\omega\, T$ is examined by the term $\lambda\mathbf{in}^T\, x.\, e$, then Rule (OUT) sets the ω tag to *cased*, which is propagated through the negative part of T. If the ω tag reaches a catamorphism, then $\mu^{\text{cased}}\, T$ and $\mu^{\text{folded}}\, T$ will not unify and the type-checking will fail.

To illustrate how the typing rules in Figure 4 detect errors, consider the term

$$\text{cata}^{\text{Clist}}\, \phi\, (\text{Rec}(\lambda\mathbf{in}^{\text{Clist}}\, x.\, e))$$

where $\text{Rec}(f) \; = \; \mathbf{in}^{\text{Clist}}(\mathbf{inR}(\mathbf{inR}(f)))$. This term is not well-typed, since $(\lambda\mathbf{in}^{\text{Clist}}\, x.\, e)$ is of type $\mu^{\text{cased}}\, \text{Clist} \to \tau$, the term $\mathbf{in}^{\text{Clist}}$ will propagate the type of its negative input, $\mu^{\text{cased}}\, \text{Clist}$, to its output, and finally the resulting type $\mu^{\text{cased}}\, \text{Clist}$ will not unify with the type $\mu^{\text{folded}}\, \text{Clist}$ in Rule (CATA). On the other hand,

$$\text{cata}^{\text{Clist}}\, \phi\, (\text{Cons}(a, (\lambda\mathbf{in}^{\text{Clist}}\, x.\, e)\, r))$$

where $\text{Cons}(a,r) \; = \; \mathbf{in}^{\text{Clist}}(\mathbf{inR}(\mathbf{inL}(a,r)))$, is well-typed, since Rule (OUT) will bind the μ tag of the negative part of T to *cased*. Since the type of the constructor Cons does not use the negative part of T, this binding will not be propagated.

A type-checking system that is based on the typing rules in Figure 4 needs a unification algorithm. A slight variation of the usual unification algorithm can be used. The only special case it needs to consider is the case of unifying $\mu^{\omega_1}\, T$ with another $\mu^{\omega_2}\, S$, since this is the case where an error occurs if a program does not satisfy the restrictions. The type $\mu^{\omega_1}\, T$ will unify with $\mu^{\omega_2}\, T$ (denoted as $\mu^{\omega_1}\, T \equiv \mu^{\omega_2}\, T$) unless either $\mu^{\text{folded}}\, T \not\equiv \mu^{\text{cased}}\, T$ or $\mu^{\text{folded}}\, T \not\equiv \mu^{\text{folded}}\, T$. We have already presented an example in which the first case occurs and in which the type checker should therefore report an error. An example in which the second case occur was given in Section 2.2.1 by

$$\text{mapC}(\lambda x.\, 2 * x)(\text{Rec}(\lambda x.\, \text{Cons}(1, \text{mapC}(\lambda x.\, x+1)\, x)))$$

The outer mapC is a catamorphism over the infinite list $1:2:3:\cdots$. Since the inner mapC is a catamorphism too, the *folded* tag will be propagated all the way through the input of the outer mapC. As we have seen, this program is invalid and it should be ruled out by the type-checker.

4 Conclusion

We have presented a new method for defining catamorphisms over datatypes with embedded functions. Our approach can be useful even when the approach outlined by other recent proposals fails, since it does not require the existence of inverse functions. Our method has a more efficient implementation and is often easier to use and understand since it does not require programmers to explicitly develop the inverse functions. We have characterized exactly when we can trade the restrictive condition about the existence of an inverse for another, more useful, condition that we can statically test.

We have demonstrated how to use datatypes with embedded functions in two large and useful domains: metaprogramming and circular structures. We have demonstrated that structures with embedded functions provide a natural way to express meta-programming and program manipulation of languages with binding constructs like lambda abstraction, because there is no renaming problem or need for a *gensym*-like solution. Therefore, our approach is purely functional and do not require us to resort to the well-known stateful monad tricks.

The other domain of examples was on structures with cycles. It is well known that implementations of functional languages use pointers and cycles, but these are hidden implementation details. They are exactly the mechanisms programmers would like to use to implement graphs, but cannot. We have developed a mechanism that allows programmers to get a better hold of these implementation details in a manner which is still safe.

$$
\begin{array}{llll}
\text{(VAR)} & \sigma \vdash x : \sigma(x) & \text{(UNIT)} & \sigma \vdash () : () \\[1em]
\text{(INL)} & \sigma \vdash \mathbf{inL} : \tau_1 \rightarrow \tau_1 + \tau_2 & \text{(INR)} & \sigma \vdash \mathbf{inR} : \tau_2 \rightarrow \tau_1 + \tau_2 \\[1em]
\text{(APPL)} & \dfrac{\sigma \vdash e_1 : \tau_1 \rightarrow \tau_2, \quad \sigma \vdash e_2 : \tau_1}{\sigma \vdash e_1\, e_2 : \tau_2} & \text{(ABS)} & \dfrac{\sigma\{x : \tau_1\} \vdash e : \tau_2}{\sigma \vdash \lambda x.\, e : \tau_1 \rightarrow \tau_2} \\[1.5em]
\text{(PROD)} & \dfrac{\sigma \vdash e_1 : \tau_1, \quad \sigma \vdash e_2 : \tau_2}{\sigma \vdash (e_1, e_2) : \tau_1 \times \tau_2} & (\times\text{ABS}) & \dfrac{\sigma\{x_1 : \tau_1,\, x_2 : \tau_2\} \vdash e : \tau}{\sigma \vdash \lambda(x_1, x_2).\, e : \tau_1 \times \tau_2 \rightarrow \tau} \\[1.5em]
\text{(CATA)} & \dfrac{\sigma \vdash e : T\,(\tau, \tau) \rightarrow \tau}{\sigma \vdash \mathrm{cata}^T\, e : \mu^{\mathrm{folded}}\, T \rightarrow \tau} & (+\text{ABS}) & \dfrac{\sigma\{x_1 : \tau_1\} \vdash e_1 : \tau, \quad \sigma\{x_2 : \tau_2\} \vdash e_2 : \tau}{\sigma \vdash (\lambda \mathbf{inL}\, x_1.\, e_1 \,[\!]\, \mathbf{inR}\, x_2.\, e_2) : \tau_1 + \tau_2 \rightarrow \tau} \\[1.5em]
\text{(IN)} & \dfrac{\sigma \vdash e : T\,(\mu^{\omega_1}\, T, \mu^{\omega_2}\, T)}{\sigma \vdash \mathbf{in}^T\, e : \mu^{\omega_2}\, T} & \text{(OUT)} & \dfrac{\sigma\{x : T\,(\mu^{\omega}\, T, \mu^{\mathrm{cased}}\, T)\} \vdash e : \tau}{\sigma \vdash \lambda \mathbf{in}^T\, x.\, e : \mu^{\mathrm{cased}}\, T \rightarrow \tau}
\end{array}
$$

Figure 4: Typing Rules (where a type-assignment is $\sigma ::= \{\} \mid \sigma\{x : \tau\}$)

5 Acknowledgments

The authors would like to thank Patty Johann, Erik Meijer, Ross Patterson, Doaitse Swierstra, Andrew Tolmach, and the anonymous referees for extensive comments on earlier drafts of this paper. Leonidas Fegaras is supported in part by the National Science Foundation under grant IRI-9509955, and by contract by the Advanced Research Projects Agency, ARPA order number 18, monitored by the US Army Research Laboratory under contract DAAB-07-91-C-Q518. Tim Sheard is supported by the USAF Air Material Command under contract # F19628-93-C-0069.

References

[1] R. Bird and O. de Moor. Solving Optimisation Problems with Catamorphisms. In *Mathematics of Program Construction*, pp 45–66. Springer-Verlag, LNCS 669, June 1992.

[2] A. Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[3] L. Fegaras, T. Sheard, and T. Zhou. Improving Programs which Recurse over Multiple Inductive Structures. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, Orlando, Florida*, pp 21–32, June 1994.

[4] J. Launchbury and T. Sheard. Warm Fusion. *Seventh Conference on Functional Programming Languages and Computer Architecture, La Jolla, California*, pp 314–323, June 1995.

[5] G. Malcolm. Data Structures and Program Transformation. *Science of Computer Programming*, 14:255–279, 1990.

[6] G. Malcolm. Homomorphisms and Promotability. In *Mathematics of Program Construction*, pp 335–347. Springer-Verlag, LNCS 375, June 1989.

[7] E. Meijer, M. Fokkinga, and R. Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts*, pp 124–144. Springer-Verlag, LNCS 523, August 1991.

[8] E. Meijer and G. Hutton. Bananas in Space: Extending Fold and Unfold to Exponential Types. *Seventh Conference on Functional Programming Languages and Computer Architecture, La Jolla, California*, June 1995.

[9] G. Nadathur and D. Miller. Higher-Order Logic Programming. In D. Gabbay, C. Hogger, and A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*. Oxford University Press, 1995. To appear.

[10] R. Paterson. Control Structures from Types. Unpublished draft. Available by anonymous ftp from `ftp-ala.doc.ic.ac.uk/pub/papers/R.Paterson/folds.dvi`, 1994.

[11] L. Paulson. *ML for the working programmer*. Cambridge University Press, 1991.

[12] F. Pfenning and C. Elliott. Higher-order Abstract Syntax. *Proceedings of the ACM SIGPLAN '88 Symposium on Language Design and Implementation, Atlanta, Georgia*, pp 199–208, June 1988.

[13] C. Reade. *Elements of Functional Programming*. Addison Wesley, 1989.

[14] T. Sheard and L. Fegaras. A Fold for All Seasons. *Sixth Conference on Functional Programming Languages and Computer Architecture, Copenhagen, Denmark*, pp 233–242, June 1993.

[15] P. Wadler. Theorems for Free! *Fourth Conference on Functional Programming Languages and Computer Architecture, Imperial College, London*, September 1989.