Revisiting Catamorphisms over Datatypes with Embedded Functions

OGI Technical Report 95/014

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

Keywords: *functional programming, lambda calculus, type systems, infinite lists, functional graphs, parametricity theorem.*

1 Introduction

Writing functional programs based upon catamorphisms (folds), or other control mechanisms that are generated from algebraic type definitions, leads to benefits ranging from the ability to calculate programs from specifications [3] to encoding a good intermediate representation that supports optimization [8, 1, 2].

Recently Paterson [5] and Meijer & Hutton [4] extended the use of such control structures to datatypes that include embedded functions. While theoretically sound, their approach suffers from the disadvantage that to express a function, f, as a catamorphism it is necessary to express another function, g, as an anamorphism (unfold), such that the composition of f and g, $f \circ g$, is the identity function. If f has no right inverse then it cannot be expressed as a catamorphism. This places severe restrictions on which functions can be expressed as catamorphisms over datatypes with embedded functions. These authors call for increased experimentation to discover uses for these generalized operators since in their experience the number of examples for which useful catamorphisms can be expressed is quite limited.

In this paper we give numerous examples of why it is truly important to be able to define catamorphisms over datatypes with embedded functions and show how to define functions as catamorphisms even when no right inverse exists by using a *trick* that invents an *approximate inverse* instead. In order to ensure soundness of this trick we replace the restriction of the existence of a right inverse with another less demanding restriction and show how the type system can be used to statically enforce this restriction.

2 Structures with Functionals are Useful

This section presents several examples of data structures with embedded functions. We define catamorphisms over these structures and give numerous examples of their use. We informally introduce our trick and explain how it works. The first example is an evaluation function over a datatype that represents closed terms in a simple lambda calculus. This representation is quite interesting since the evaluation function does not need an environment mapping variables to values. The second example is the expression and manipulation of circular lists in a functional language. Another example, presented in Appendix A.1, is the expression of the the parametricity theorem for any polymorphic function. The final example, presented in Appendix A.2, is the expression and manipulation of graphs. Both appendices are given to support our claim that structures with functionals are useful. They need not be read to understand our ideas.

2.1 Lambda Calculus

Our first example is an evaluation function for a datatype that represents closed terms in a simple lambda calculus. In contrast to other approaches, we represent terms as structures with functionals: (all our examples are written in Standard ML (SML) [6])

$$\textbf{datatype}\ \mathsf{Term} = \mathsf{Const}\ \textbf{of}\ \mathsf{int} \mid \mathsf{Succ} \mid \mathsf{Appl}\ \textbf{of}\ \mathsf{Term} \times \mathsf{Term} \mid \mathsf{Abs}\ \textbf{of}\ \mathsf{Term} \to \mathsf{Term}$$

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

$$\mathsf{Appl}(\mathsf{Abs}(\textbf{fn}\ \mathsf{x} \Rightarrow \mathsf{Appl}(\mathsf{Succ},\mathsf{x})),\mathsf{Const}\ 1)$$

In most lambda term representations, lambda-abstractions are constructed by value constructors of type variable $\times$ Term $\to$ Term and also include constructors like Var of type variable $\to$ Term. Under 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 we use the abstraction mechanism of the meta-language (SML) to represent abstraction in the object language of lambda terms. This completely finesses the bound-variable naming problem.

Thus, our evaluator has no variables or associated complexity. All the necessary variable plumbing, when handling beta reduction, etc. is handled implicitly by the evaluation engine of SML in which our evaluator is expressed. 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.

We now present an evaluator for terms in the style of Paterson [5], and Meijer & Hutton [4]. We use explicit recursion instead of catamorphisms to make clear what exactly is going on and to illustrate why this approach is limited. Our approach, which improves this method, is described in detail in the next section.

The value domain of our evaluator is:

$$\textbf{datatype}\ \mathsf{Value} = \mathsf{Num}\ \textbf{of}\ \mathsf{int} \mid \mathsf{Fun}\ \textbf{of}\ \mathsf{Value} \to \mathsf{Value}$$

and uses embedded functions itself to represent the meaning of functional terms in the lambda calculus. The lambda term evaluator is a function of type Term $\to$ 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 $\to$ Value. The function f of type Term $\to$ Term must somehow be manipulated into a value of type Value $\to$ Value. The proper way to do this is to compose f with a function of type Term $\to$ Value on the left and a function of type Value $\to$ Term on the right. The obvious choice for the first function is eval. The second function, which we call reify, translates values into terms. Since term evaluation is not isomorphic in general, we have many different choices. Whichever function we choose, reify should be the right inverse of eval, i.e., eval $\circ$ reify $= \lambda x.x$. Without this restriction, eval would fail to evaluate even the simplest abstraction, Abs(**fn** x $\Rightarrow$ x), into the correct result: Fun(**fn** x $\Rightarrow$ x). For a first attempt, we define reify as follows:

```
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 o f o reify)
and reify(Num n)   = Const n
|    reify(Fun f)   = Abs(reify o f o 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 can be proved by structural induction that reify is the right inverse of eval:
Base case: eval(reify(Num n)) = eval(Const n) = Num n;

Induction step: eval(reify(Fun f)) = eval(Abs(reify ∘ f ∘ eval)) = Fun(eval ∘ reify ∘ f ∘ eval ∘ reify) = Fun f (from the induction hypothesis).

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. For example,

```
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))
```

This abstraction is eventually applied to a term, which needs to be evaluated.

```
= 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 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 worse for more complex functions. 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 so that 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.

In addition to the recomputation introduced by the redundant eval's and reify's, there is a more serious reason why this approach is not practical in many cases: reify must be the right inverse of eval, but such inverses may not always exist for functions other than eval.

This duality is exactly what Paterson [5] and Meijer & Hutton [4] have explored. 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 anamorphism will generate a value of type T from a value of type S.

2.1.1 Our Approach

To avoid the computational redundancy and to find a way around the problem of finding a right inverse, we return to the initial problem of expressing g? as eval ∘ f ∘ h? for a second look. In particular, the crucial property of h? with type Value → Term is that it satisfies eval ∘ h? = $\lambda x.x$. In addition, we would like this to happen with no computational overhead. One way to partially accomplish this is for h? to be a value constructor and to add the additional clause in the definition of eval: eval (h? x) = x. That is, we change the domain, Term, of eval to include a new constructor and add an extra clause to the definition of eval.

Consequently, 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 (which justifies the name Place: for 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.

This technique 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:

$$\textbf{datatype}\ \alpha\ \mathsf{Term} = \mathsf{Const}\ \textbf{of}\ \mathsf{int}\ |\ \mathsf{Succ}\ |\ \mathsf{Appl}\ \textbf{of}\ \alpha\ \mathsf{Term} \times \alpha\ \mathsf{Term}\ |\ \mathsf{Abs}\ \textbf{of}\ \alpha\ \mathsf{Term} \to \alpha\ \mathsf{Term}\ |\ \mathsf{Place}\ \textbf{of}\ \alpha$$

This allows any α object to be a subtype of Term, and Place plays the role of an indirection operator or 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 )
```

Generally it is not advisable to extend type definitions by adding new constructors for reasons of code reuse: it causes non-exhaustive case analysis in pre-existing code. In our example we have additional reasons why the existence of the Place constructor is problematic. If the user constructs a Term with Place, the type of the term will not be fully parametric. That is, the type variable α will be bound to some type, t, and that 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, if the user constructs Terms with Place, it is not clear what this will do to 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 case where Place is introduced by cataT, producing 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 places in the input that have type Term, including terms like (Place z), which will reduce to z. Therefore, Place will not appear in the output.

This suggests that if the catamorphism operator were a primitive in the programming language, then just one Place constructor is needed. This special constructor, **Place**, could be used by every catamorphism, regardless of the type it traverses. We leave it to the implementation of catamorphisms as primitives to 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.

2.1.2 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? Consider applying eval (defined in Section 2.1.1) to the following term:

$$\mathsf{Abs}(\mathbf{fn}\ \mathsf{x} \Rightarrow \mathbf{case}\ \mathsf{x}\ \mathbf{of}\ \mathsf{Const\ n} \Rightarrow \mathsf{Const\ n} \mid \mathsf{z} \Rightarrow \mathsf{Const\ 0})$$

If we evaluate it, we get

$$\mathsf{Fun}(\mathbf{fn}\ \mathsf{y} \Rightarrow \mathsf{eval}(\mathbf{case}\ (\mathbf{Place}\ \mathsf{y})\ \mathbf{of}\ \mathsf{Const\ n} \Rightarrow \mathsf{Const\ n} \mid \mathsf{z} \Rightarrow \mathsf{Const\ 0}))$$

Not every instance of a Term can be traversed by eval expressed as a primitive catamorphism, because the function, f, embedded in Abs(f) cannot know how to handle **Place** in all cases, 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 since there is no case to handle **Place**.

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 datatype performs a case analysis over its argument. This is not always problematic; it only becomes a problem if there exists a catamorphism over a particular instance in the program with this property. 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 very simple and can be implemented efficiently. It reports error only on invalid terms.

2.2 Circular Lists

In this section we present a second example of a datatype 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 these structures by finite algebraic datatypes. One way to do this is to use embedded functions. In this section we express circular lists. In Appendix A.2 we extent this method to capture general graphs.

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

$$\mathsf{circ} = 0{:}1{:}\mathsf{circ}$$

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 [6]:

$$\mathbf{datatype}\ \alpha\ \mathsf{Clist} = \mathsf{Nil} \mid \mathsf{Cons}\ \mathbf{of}\ \alpha \times (\mathsf{unit} \to \alpha\ \mathsf{Clist})$$

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

$$\mathbf{let\ fun}\ \mathsf{circ}() = \mathsf{Cons}(0{,}\mathbf{fn}\ () \Rightarrow \mathsf{Cons}(1{,}\mathsf{circ}))\ \mathbf{in}\ \mathsf{circ}()\ \mathbf{end}$$

Even though many circular structures can be defined this way, many operations over them need to explicitly carry a list of visited nodes to avoid falling into an infinite loop. Identifying which nodes have been visited is 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 an alternative to recursion).

To find a better definition for circular lists, reconsider the previous Haskell definition. This is equivalent to $Y(\mathbf{fn}\ \mathsf{x} \Rightarrow 0{:}1{:}\mathsf{x})$, where Y is the fixpoint operator of type $(\alpha \to \alpha) \to \alpha$ that satisfies $Y\,f = f(Y\,f)$. This observation while interesting has really accomplished nothing since the Haskell definition uses an implicit Y combinator. Furthermore, the expression 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 of the Y combinator, $(\alpha\ \mathsf{list} \to \alpha\ \mathsf{list}) \to \alpha\ \mathsf{list}$. This leads us 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 follows: Rec(**fn** x ⇒ Cons(0,Cons(1,x))). Functions that manipulate these structures need to unroll the implicit fixpoint in Rec explicitly. For example:

```
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 use the same trick we used earlier by adding 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 $\beta \to \beta$ and it is up to g to decide what to do with it. For example, the map mapC(h) over circular lists can be expressed by

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 example:

```
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 $2 : 6 : 8 : 10 : \cdots$. 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 mapC, not the inner. Instead it cancelled the inner 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. Even though nth works fine over the above construction, the problem is that this construction cannot be traversed over by another cataC in our model, because of the implicit case analysis implied by the application of mapC to the argument x.

Fortunately cases like this are automatically discovered by the type inference system discussed in the next section.

In each of the examples above, it was necessary to use the trick of extending the datatype definition with an additional constructor Place, an additional type variable to hold the type of the result of a catamorphism, and to write the catamorphism function by hand. In addition, there was no guarantee that the programs written obeyed the restriction explained in Section 2.1.2.

In the next section we define a language in which this trick is implemented implicitly. That is, the Place constructor and the catamorphism function are primitives of the language. Because Place is hidden, it is impossible for the user to construct programs that use Place. The language also supports a type system that enforces the 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 used above. 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 in internal implementation detail.

For reasons of simplicity, the language does not use explicit value constructors, as it is done in most functional languages. Instead it uses binary sum and product types. This notation makes the theory easier to explain because fewer rules are needed to express our algorithms, but makes programs hard to understand. As we proceed through this section, we will show the correspondence between functional languages and our language.

3.1 Terms

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

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

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). **inL** and **inR** are the left and right injectors of the sum type. cata^T is the catamorphism operator for T. The *in-abstraction* $\lambda\mathbf{in}^T\,x.\,e$ is defined by $(\lambda\mathbf{in}^T\,x.\,e)\,(\mathbf{in}^T\,u) = (\lambda x.\,e)\,u$. 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:

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

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,

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

can be expressed by the following composition of operators:

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

$$\rho \vdash () :: k \qquad \rho \vdash x :: \rho(x)$$

$$\frac{\rho \vdash \tau_1 :: k, \quad \rho \vdash \tau_2 :: k}{\rho \vdash \tau_1 \times \tau_2 :: k} \qquad \frac{\rho \vdash \tau_1 :: k, \quad \rho \vdash \tau_2 :: k}{\rho \vdash \tau_1 + \tau_2 :: k}$$

$$\frac{\rho \vdash \tau_1 :: \neg k, \quad \rho \vdash \tau_2 :: k}{\rho \vdash \tau_1 \to \tau_2 :: k}$$

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

Figure 1: Well-formedness of Types (Definition: $\neg(+) = -$ and $\neg(-) = +$)

3.2 Types

Our types are generated by the following grammar:

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

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 (negative) variable should only appear in a positive (negative) position in a type. This condition is implicitly checked by the well-formedness of types rules described in Figure 1. As an informal example, 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 1 check the well-formedness of types. They use the following definitions:

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

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

$$\mu^\omega \, T \, (\tau_1, \tau_1') \cdots (\tau_n, \tau_n') \quad = \quad T \, ((\mu^\omega \, T \, (\tau_1, \tau_1') \cdots (\tau_n, \tau_n')), (\mu^\omega \, T \, (\tau_1', \tau_1) \cdots (\tau_n', \tau_n))) \, (\tau_1, \tau_1') \cdots (\tau_n, \tau_n')$$

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 1 is a functor that is covariant in its positive arguments and contravariant in its negative arguments:

$$(T \, (f_1, f_1') \cdots (f_n, f_n')) \circ (T \, (g_1, g_1') \cdots (g_n, g_n')) \; = \; T \, (f_1 \circ g_1, g_1' \circ f_1') \cdots (f_n \circ g_n, g_n' \circ f_n')$$

The following are examples of type definitions:

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

For example, the inductive list type definition

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

is derived by applying the fixpoint operator to List:

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

Note that all 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 over a type definition T, cata^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 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/f_i' is associated with the type variable x_i/x_i'. The term $\mathcal{M}[\![\tau]\!]$ is derived by a case analysis over the type τ:

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

where for the functor $S = \Lambda(x_0, x_0') \ldots \Lambda(x_m, x_m').\ \tau$:

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

Theorem 1 *The functor $T\,(f_0, f_0') \cdots (f_n, f_n') = \mathcal{M}[\![\tau]\!]$, for $T = \Lambda(x_0, x_0') \ldots \Lambda(x_n, x_n').\tau$, satisfies*

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

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 is sufficient to prove that, for any type τ, if $\{\} \vdash \tau :: +$, then $\mathcal{M}_3[\![\tau]\!] = \mathcal{M}_1[\![\tau]\!] \circ \mathcal{M}_2[\![\tau]\!]$, otherwise $\mathcal{M}_3[\![\tau]\!] = \mathcal{M}_2[\![\tau]\!] \circ \mathcal{M}_1[\![\tau]\!]$. We present only one case of this proof where $\tau = \tau_1 \to \tau_2$ and τ is positive:

$$\begin{array}{lll}
\mathcal{M}_1[\![\tau]\!] \circ \mathcal{M}_2[\![\tau]\!] & & \\
= & (\lambda h.\,\lambda x.\,\mathcal{M}_1[\![\tau_2]\!](h(\mathcal{M}_1[\![\tau_1]\!](x)))) \circ (\lambda h.\,\lambda x.\,\mathcal{M}_2[\![\tau_2]\!](h(\mathcal{M}_2[\![\tau_1]\!](x)))) & \text{(by definition of } \mathcal{M}[\![\tau_1 \to \tau_2]\!]) \\
= & \lambda h.\,\lambda x.\,(\mathcal{M}_1[\![\tau_2]\!](\mathcal{M}_2[\![\tau_2]\!](h(\mathcal{M}_2[\![\tau_1]\!](\mathcal{M}_1[\![\tau_1]\!](x)))))) & \text{(by composition)} \\
= & \lambda h.\,\lambda x.\,(\mathcal{M}_3[\![\tau_2]\!](h(\mathcal{M}_3[\![\tau_1]\!](x)))) & \text{(by induction hypothesis)} \\
= & \mathcal{M}_3[\![\tau]\!] & \text{(by definition of } \mathcal{M}[\![\tau_1 \to \tau_2]\!])
\end{array}$$

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

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

According to Theorem 1, it 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:

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

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

$$(\text{cata}^T\ \phi) \circ \mathbf{in}^{T'} \quad = \quad \lambda \mathbf{inL}\, x.\, \phi(\mathcal{E}^T\ (\text{cata}^T\ \phi, \mathbf{in}^{T'} \circ \mathbf{inR})\ x) \parallel \mathbf{inR}\, y.\, y$$

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*

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

For example, according to this definition, the catamorphism for circular lists is $\text{cata}^{\text{Clist}}\ \phi$:

$$\begin{array}{lcl} \text{cata}^{\text{Clist}}\ \phi\ (\mathbf{in}^{\text{Clist}}\, x) & = & \phi((\lambda \mathbf{inL}\, y.\, \mathbf{inL}\, y \\ & & \quad \parallel \mathbf{inR}\, y.\, \mathbf{inR}((\lambda \mathbf{inL}\, z.\, \mathbf{inL}((\lambda (a, r).\, (a, \text{cata}^{\text{Clist}}\ \phi\, r))\ z) \\ & & \quad\quad \parallel \mathbf{inR}\, h.\, \mathbf{inR}(\lambda w.\, \text{cata}^{\text{Clist}}\ \phi\ (h(\mathbf{Place}\, w))))\ y))\ x) \\ \text{cata}^{\text{Clist}}\ \phi\ (\mathbf{Place}\, x) & = & x \end{array}$$

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

$$\begin{array}{lcl} \text{cata}^{\text{Clist}}\ \phi\ (\mathbf{in}^{\text{Clist}}\, x) & = & \phi(\text{case } x \text{ of} \\ & & \quad \text{Nil}' \ \Rightarrow\ \text{Nil}' \\ & & \quad \mid \text{Cons}'(a, r) \ \Rightarrow\ \text{Cons}'(a, \text{cata}^{\text{Clist}}\ \phi\, r) \\ & & \quad \mid \text{Rec}'(f) \ \Rightarrow\ \text{Rec}'((\text{cata}^{\text{Clist}}\ \phi) \circ f \circ \mathbf{Place})) \end{array}$$

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

$$\begin{array}{l} \text{cata}^{\text{Clist}}\ (\lambda \mathbf{inL}\, y.\, \mathbf{inL}\, y \\ \quad \parallel \mathbf{inR}\, y.\, \mathbf{inR}((\lambda \mathbf{inL}\, z.\, \mathbf{inL}((\lambda (a, r).\, (g\, a, r))\ z) \\ \quad\quad \parallel \mathbf{inR}\, h.\, \mathbf{inR}(h))\ y)) \end{array}$$

Meijer and Hutton [4] define a catamorphism cata^T in conjunction with its dual the anamorphism ana^T as follows:

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

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

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 report invalid terms by assigning 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 we 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 2 presents the typing rules for our λ-calculus. All rules are typical except the rules (CATA), (IN), and (OUT). To simplify these rules, we assume that a type definition T has only one type abstraction, i.e. 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 but 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 a way the opposite of Rule (IN): $\lambda \mathbf{in}^T\, x.\, e$ 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

$$\text{(VAR)}\quad \sigma \vdash x : \sigma(x) \qquad \text{(UNIT)}\quad \sigma \vdash () : ()$$

$$\text{(INL)}\quad \sigma \vdash \mathbf{inL} : \tau_1 \to \tau_1 + \tau_2 \qquad \text{(INR)}\quad \sigma \vdash \mathbf{inR} : \tau_2 \to \tau_1 + \tau_2$$

$$\text{(APPL)}\quad \frac{\sigma \vdash e_1 : \tau_1 \to \tau_2, \quad \sigma \vdash e_2 : \tau_1}{\sigma \vdash e_1\, e_2 : \tau_2} \qquad \text{(ABS)}\quad \frac{\sigma\{x : \tau_1\} \vdash e : \tau_2}{\sigma \vdash \lambda x.\, e : \tau_1 \to \tau_2}$$

$$\text{(PROD)}\quad \frac{\sigma \vdash e_1 : \tau_1, \quad \sigma \vdash e_2 : \tau_2}{\sigma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad (\times\text{ABS})\quad \frac{\sigma\{x_1 : \tau_1,\, x_2 : \tau_2\} \vdash e : \tau}{\sigma \vdash \lambda(x_1, x_2).\, e : \tau_1 \times \tau_2 \to \tau}$$

$$\text{(CATA)}\quad \frac{\sigma \vdash e : T\,(\tau, \tau) \to \tau}{\sigma \vdash \text{cata}^T\, e : \mu^{\text{folded}}\, T \to \tau} \qquad (+\text{ABS})\quad \frac{\sigma\{x_1 : \tau_1\} \vdash e_1 : \tau, \quad \sigma\{x_2 : \tau_2\} \vdash e_2 : \tau}{\sigma \vdash (\lambda \mathbf{inL}\, x_1.\, e_1 \,[\!]\, \mathbf{inR}\, x_2.\, e_2) : \tau_1 + \tau_2 \to \tau}$$

$$\text{(IN)}\quad \frac{\sigma \vdash e : T\,(\mu^{\omega_1}\, T, \mu^{\omega_2}\, T)}{\sigma \vdash \mathbf{in}^T\, e : \mu^{\omega_2}\, T} \qquad \text{(OUT)}\quad \frac{\sigma\{x : T\,(\mu^{\omega}\, T, \mu^{\text{cased}}\, T)\} \vdash e : \tau}{\sigma \vdash \lambda \mathbf{in}^T\, x.\, e : \mu^{\text{cased}}\, T \to \tau}$$

Figure 2: Typing Rules (where a type-assignment is $\sigma ::= \{\} \mid \sigma\{x : \tau\}$)

μ^{ω} to *folded*. If a term of type $\mu^{\omega}\, T$ is examined by the term $\lambda \mathbf{in}^T\, x.\, e$, it sets the ω tag to *cased*, which is propagated through the negative part of T. If it reaches a catamorphism, then $\mu^{\text{cased}}\, T$ and $\mu^{\text{folded}}\, T$ will not unify and the type-checking will fail.

For example, the term

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

where $\text{Rec}(f) = \mathbf{in}^{\text{Clist}}(\mathbf{inR}(\mathbf{inR}(f)))$, 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 2 needs a unification algorithm. This unification algorithm is quite typical. 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 $\mu^{\omega_1}\, T \equiv \mu^{\omega_2}\, T$) in all but the following cases:

$$\mu^{\text{folded}}\, T \not\equiv \mu^{\text{cased}}\, T \qquad\qquad \mu^{\text{folded}}\, T \not\equiv \mu^{\text{folded}}\, T$$

We have already presented an example for the first case in which the type checker should report an error. An example of the second case was given in Section 2.2:

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

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 of defining catamorphisms over datatypes with embedded functions. Our approach can be useful even when the approach outlined by other recent proposals fails, as it does not require the existence of 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 on two large and useful domains: meta-programming and circular structures.

We have demonstrated that structures with embedded functions are 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 solutions are purely functional without the need to resort to the well known stateful monad tricks. Meta-programs constructed this way always meet the type-checked condition.

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

5 Acknowledgments

The authors would like to thank Erik Meijer, Ross Patterson, Doaitse Swierstra, and Andrew Tolmach for extensive comments on earlier drafts of this paper. Leonidas Fegaras is supported 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.

References

[1] 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.

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

[3] 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, August 1991. LNCS 523.

[4] 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.

[5] R. Paterson. Control Structures from Types. *Submitted to Journal of Functional Programming*, 1994. Available from `ftp://ftp-ala.doc.ic.ac.uk/pub/papers/R.Paterson/folds.dvi`.

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

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

[8] 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.

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

A More Examples

A.1 The Parametricity Theorem

As another example of representing terms by structures with embedded functions, we construct the parametricity theorem for any polymorphic function. To understand this section, the reader must be familiar with Wadler's theorems-for-free paper [9].

Any function f of type τ satisfies a parametricity theorem, which is directly derived from the type τ. For first-order functions, this theorem basically says that any polymorphic function is a natural transformation. The theorem for $f : \tau$ is $\mathcal{F}[\![\tau]\!](f, f)$, which indicates that $(f, f) \in \tau$, where types here are considered as relations.

Theorem 2 (Parametricity Theorem) *For any strict function $f : \tau$ we have $\mathcal{F}[\![\tau]\!](f, f)$, where:*

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

Here, for each type variable α, we associate a function α (of type $\alpha_1 \to \alpha_2$, where α_1 and α_2 are instances of α). (Regularly, we would need to pass an environment through $\mathcal{F}[\![\tau]\!](r,s)$ that maps type variable names to function names.)

For example, we construct the parametricity theorem for list catamorphism, $\mathrm{cata}^{\mathrm{list}} : (\alpha \to \beta \to \beta) \to \beta \to \mathrm{list}(\alpha) \to \beta$: (the construction is done in four simple steps)

$$
\begin{array}{lcl}
\mathcal{F}[\![\forall\delta.\forall\varepsilon.\ \delta \to \varepsilon]\!](r,s) & \longrightarrow & \forall\delta,\varepsilon,x,y : \mathcal{F}[\![\delta]\!](x,y) \Rightarrow \mathcal{F}[\![\varepsilon]\!](r(x), s(y)) \\
& \longrightarrow & \forall\delta,\varepsilon,x,y : x = \delta(y) \Rightarrow r(x) = \varepsilon(s(y)) \\
& \text{or} & \forall\delta,\varepsilon : r \circ \delta = \varepsilon \circ s \\
\mathcal{F}[\![\forall\delta.\forall\varepsilon.\forall\eta.\ \delta \to \varepsilon \to \eta]\!](r,s) & \longrightarrow & \forall\delta,\varepsilon,\eta,x,y : \mathcal{F}[\![\delta]\!](x,y) \Rightarrow \mathcal{F}[\![\varepsilon \to \eta]\!](r(x), s(y)) \\
& \longrightarrow & \forall\delta,\varepsilon,\eta,x,y : x = \delta(y) \Rightarrow r(x) \circ \varepsilon = \eta \circ s(y) \\
& \text{or} & \forall\delta,\varepsilon,\eta,y,z : r\,(\delta\, y)\,(\varepsilon\, z) = \eta(s\, y\, z) \\
\mathcal{F}[\![\forall\alpha.\ \mathrm{list}(\alpha)]\!](r,s) & \longrightarrow & \forall\alpha, f, x : f(x) = \alpha(x) \Rightarrow r = \mathrm{map}^{\mathrm{list}}(f)\, s \\
& \text{or} & \forall\alpha : r = \mathrm{map}^{\mathrm{list}}(\alpha)\, s
\end{array}
$$

$$
\begin{array}{l}
\mathcal{F}[\![\forall\alpha.\forall\beta.\ (\alpha \to \beta \to \beta) \to \beta \to \mathrm{list}(\alpha) \to \beta]\!](r,s) \\
\quad \longrightarrow\ \forall\alpha,\beta,\otimes,\oplus : \mathcal{F}[\![\alpha \to \beta \to \beta]\!](\otimes,\oplus) \Rightarrow \mathcal{F}[\![\beta \to \mathrm{list}(\alpha) \to \beta]\!](r \otimes, s \oplus) \\
\quad \longrightarrow\ \forall\alpha,\beta,\otimes,\oplus,x,y : (\alpha\, x) \otimes (\beta\, y) = \beta(x \oplus y) \Rightarrow r \otimes (\beta\, x)\,(\mathrm{map}^{\mathrm{list}}(\alpha)\, y) = \beta(s \oplus x\, y)
\end{array}
$$

That is, the list catamorphism satisfies the following theorem:

$$
(\alpha\, x) \otimes (\beta\, y) = \beta(x \oplus y) \Rightarrow \mathrm{cata}^{\mathrm{list}}(\otimes)\,(\beta\, x)\,(\mathrm{map}^{\mathrm{list}}(\alpha)\, y) = \beta(\mathrm{cata}^{\mathrm{list}}(\oplus)\, x\, y)
$$

The parametricity theorem can be easily extended to parametrize over type constructors: for each free type constructor $T : * \to *$ we associate a function T of type $(\alpha_1 \to \alpha_2) \to T_1(\alpha_1) \to T_2(\alpha_2)$, where T_1 and T_2 are instances of T. This is useful when we want to express laws about operations 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: $\phi \circ (T\,\alpha) = \alpha \circ \psi \Rightarrow (\mathrm{cata}\,\phi) \circ (Y\,T) = \alpha \circ (\mathrm{cata}\,\psi)$, where these cata here may be of different type constructors.

The construction of the predicate of Theorem 2 requires some variable plumbing when we introduce universal quantification (to avoid name capture etc.) In addition, for each type variable a function is

associated, and this binding should be carried through the whole construction. We can avoid these 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
```

By following the same routine as before, (i.e. adding the new constructor Place to type T, etc.), the catamorphism over types is:

```
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) ∘ f ∘ 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 following higher-order catamorphism:

```
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))
```

When this function operates over Univ(g), it lifts g:T → T into f: (E × E → E) → (E × E → 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 2). Notice that there is no need of using 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.

A.2 Graphs

Graphs can be represented in a way similar to circular lists. This allows terminating computations over graphs, such as finding 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. This approach is not really different from using pointers in a procedural language: it permits ad-hoc constructions and manipulations of graphs. The approach described in [7] defines a graph type as:

$$\textbf{datatype}\ \alpha\ \mathsf{graph} = \mathsf{Graph}\ \textbf{of}\ \alpha \to \alpha\ \mathsf{list}$$

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 graphs are based on the idea of using embedded functions as we did for circular lists. We start with a datatype that can represent trees with nodes that support arbitrary branching levels. We call such a tree a rose_tree. It is defined as follows:

$$\textbf{datatype}\ \alpha\ \mathsf{rose\_tree} = \mathsf{Node}\ \textbf{of}\ \alpha \times \alpha\ \mathsf{rose\_tree\ list}$$

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
```

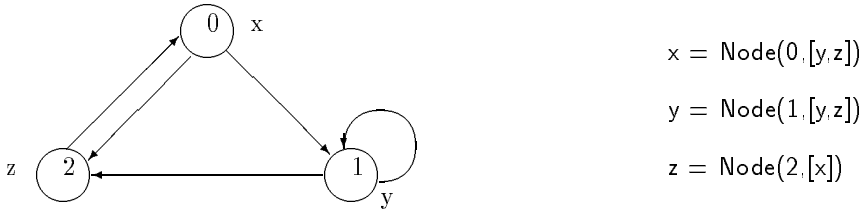

Figure 3: Graph with three nodes $0, 1, 2$ and five edges

Here, Rec plays the role of the Y combinator to express cycles, while Share plays the role of a function application. That is, Share(f,e) is unrolled as f e. Basically, all free occurrences of x in u in the term Share(**fn** x ⇒ u, e) are bound to e. That is, all x in u are sharing the same subgraph e.

For example,

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

describes the graph in Figure 3. By following the same routine as for circular lists, (i.e. adding the new constructor Place, etc.), the graph catamorphism, cataG, is:

```
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
```

Examples:

```
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)
```

For example, the graph in Figure 3 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 [x1, ..., xn] ⇒ [f1(x1, ..., xn), ..., fn(x1, ..., xn)] | 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 work. By following the same routine as before, (i.e. adding the new constructor Place, etc.), the graph catamorphism is:

```
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 gotten an empty adjacency list for each node.