Functional Programming with Structured Graphs

Bruno C. d. S. Oliveira
National University of Singapore
oliveira@comp.nus.edu.sg

William R. Cook
University of Texas, Austin
wcook@cs.utexas.edu

Abstract

This paper presents a new functional programming model for graph structures called *structured graphs*. Structured graphs extend conventional algebraic datatypes with explicit definition and manipulation of cycles and/or sharing, and offer a practical and convenient way to program graphs in functional programming languages like Haskell. The representation of sharing and cycles (edges) employs recursive binders and uses an encoding inspired by *parametric higher-order abstract syntax*. Unlike traditional approaches based on mutable references or node/edge lists, *well-formedness* of the graph structure is ensured statically and reasoning can be done with standard functional programming techniques. Since the binding structure is generic, we can define many useful generic combinators for manipulating structured graphs. We give applications and show how to reason about structured graphs.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Functional Languages; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs

General Terms Languages

Keywords Graphs, parametric HOAS, Haskell.

1. Introduction

Functional programming languages, including Haskell [31] and ML [29], excel at manipulating *tree structures*. In those languages *algebraic datatypes* describe the structure of values, and *pattern matching* is used to define functions on such tree structured values. These mechanisms provide a high-level declarative programming model, which avoids explicit manipulation of pointers or references. Additionally algebraic datatypes facilitate reasoning about functions using standard proof methods, including *structural induction*.

However, there are many kinds of data that are more naturally represented as *graph structures* rather than trees. Some examples include: typical compiler construction concerns including control/data flow graphs or grammars [2]; entity-relational data models [8]; finite state machines; or transitions systems. Sadly, functional programming languages do not have an equally good answer when it comes to manipulating graphs as they do for trees.

ICFP'12, September 9–15, 2012, Copenhagen, Denmark.

In impure functional languages, including ML or OCaml [24], a combination of algebraic datatypes and mutable references can be used to model sharing and cycles. However this requires explicit manipulation of mutable references, which precludes many of the benefits of functional programming and algebraic datatypes. For example observing sharing via pointer/reference comparison breaks *referential transparency*. Even if it is possible to encapsulate the use of mutable references under a purely functional interface, reasoning about implementations remains challenging [19, 35].

In *call-by-need* functional languages, like Haskell, it is possible to construct true cyclic structures. For example:

$$ones = 1 : ones$$

creates a cyclic list where the head contains the element 1 and the tail is a reference to itself. However, sharing is not observable and, from a purely semantic perspective, $ones$ is no different from an infinite list of 1's. A drawback of this approach is that when an operation is applied to this list sharing is lost, even if it would be possible to preserve sharing. For example,

$$twos = map\ (\lambda x \rightarrow x + 1)\ ones$$

creates an infinite list of 2's instead of a cyclic list with a single 2.

To deal with the need for observable sharing some researchers have proposed approaches that use recursive binders to model cycles and sharing [14, 15, 20]. For example, $ones$ can be expressed using a recursive binder (μ) as follows:

$$ones = \mu\ x.\ (1 : x)$$

The idea is to be able to observe and manipulate the binders (μ) and variables (x), making sharing effectively explicit.

However, several questions need to be answered for this model to become effective in practice:

1. What programming language mechanisms are needed to support convenient representation and manipulation of such binders and variables? Does the approach guarantee well-formedness of cyclic structures (no unbound variables or other types of junk)?
2. Is the model expressive enough? Can it deal with general graph edges, including the back edges and cross edges which arise in non-linear graph structures?
3. Can the model deal with operations that require special treatment of fixpoint computations? For example, is it possible to exploit monotonicity to ensure termination on all inputs for an operation that checks the *nullability* [7, 28] of a grammar or regular expression?

As far as we know no approach deals with #3. Furthermore all approaches provide only partial answers to #1 and #2 (a detailed discussion is given in Section 7), and fall short in providing a practical programming model for cyclic structures.

This paper presents a new functional programming model for graph structures, called *structured graphs*, that builds on the idea of

using recursive binders to model sharing and cycles and provides an answer for all 3 questions. Structured graphs can be viewed as an extension of algebraic datatypes that allow explicit definition and manipulation of cycles or sharing by using recursive binders and variables to explicitly represent possible sharing points. To provide a convenient and expressive programming interface, structured graphs use a binding representation based on *parametric higher-order abstract syntax* (PHOAS) [9]. This representation not only ensures well-formedness of the binding structure, but it also allows using standard proofs methods, including structural induction, in proofs for a large class of programs. To deal with cross edges we use a recursive multi-binder inspired by *letrec* expressions in functional programming. Furthermore, the expressiveness and flexibility of our PHOAS-based representation allows us to define operations that require special treatment of fixpoint computations.

Since the binding structure is generic, it is possible to define many useful generic combinators for manipulating structured graphs. By employing some lightweight datatype-generic programming [17] techniques, we also propose a datatype-generic formulation of structured graphs. This formulation enables the definitions of useful combinators like generic folds and transformations. Using such combinators it is often possible to write programs for processing graphs that are no more difficult to write than programs on conventional algebraic datatypes.

The programming model also allows for transformations requiring complex manipulation of the binding structure, which are less easily captured in combinators. Those operations can always be defined by direct pattern matching on the binding structure (variables and binders).

To summarize, our contributions are:

- **Structured graphs:** a new programming model extending the classic notion of algebraic datatypes with cycles and sharing. This model supports the same benefits as algebraic datatypes and facilitates reasoning over cyclic structures. The binding infrastructure is conveniently defined and manipulated using a PHOAS-based representation.
- **Generic combinators and infrastructure:** Additional convenience is provided through the use of generic combinators for folds and transformations. Such generic combinators can also encapsulate the use of special fixpoints for certain operations.
- **Recursive binders using PHOAS:** We also show how to define recursive binders with PHOAS. The recursive multi-binder presented in Section 3.2 is particularly relevant, since it enables the definition of cross edges.

The presentation of our work uses Haskell. Occasionally we use some common extensions implemented in the GHC compiler. The code for this paper is available online at `http://ropas.snu.ac.kr/~bruno/papers/StructuredGraphs.zip`.

2. Parametric HOAS

This section reviews the key advantages of Parametric Higher-Order Abstract Syntax (PHOAS) [9] for representing binders. There are several approaches to binding, but PHOAS has a unique combination of advantages: 1) guaranteed *well-scopedness*; 2) *no explicit manipulation of environments*; 3) *easy to define operations*. The first two advantages are due to the fact that PHOAS is a *higher-order* approach in which the function space of the meta-language is used to encode the binders of the object language. The reuse of the meta-language function space avoids common issues with first-order approaches like α-equivalence and defining the infrastructure for capture-avoiding substitution. Other higher-order approaches like classic HOAS [34] share these advantages. However,

$$
\begin{aligned}
&\textbf{data}\ PLambda\ a = \\
&\quad Var\ a \\
&\mid Int\ Int \\
&\mid Bool\ Bool \\
&\mid If\ (PLambda\ a)\ (PLambda\ a)\ (PLambda\ a) \\
&\mid Add\ (PLambda\ a)\ (PLambda\ a) \\
&\mid Mult\ (PLambda\ a)\ (PLambda\ a) \\
&\mid Eq\ (PLambda\ a)\ (PLambda\ a) \\
&\mid Lam\ (a \to PLambda\ a) \\
&\mid App\ (PLambda\ a)\ (PLambda\ a) \\
&\textbf{newtype}\ Lambda = \downarrow \{\uparrow :: \forall a.PLambda\ a\}
\end{aligned}
$$

Figure 1. PHOAS-encoded lambda calculus with integers, booleans and some primitives.

with classic HOAS (and other higher-order approaches) many operations are non-trivial to define in languages like Haskell, whereas with PHOAS various operations are generally easier to define. This unique combination of features makes PHOAS a particularly attractive foundation for our work.

To illustrate the advantages of PHOAS in more detail, we use the lambda calculus (with standard extensions) presented in Figure 1. Lambda terms are encoded by the newtype $Lambda$, which is defined in terms of the datatype $PLambda\ a$.

Well-scopedness The type argument a in $PLambda\ a$ is supposed to be *abstract*: it should not be instantiated to a concrete type when constructing lambda terms. To enforce this, a universal quantifier ($\forall a.PLambda\ a$) is used in the definition of $Lambda$. Note that, in Haskell, the following type synonym:

$$\textbf{type}\ Lambda = \forall a.PLambda\ a$$

can be problematic to encode $Lambda$. This is because in Haskell all universal quantifications are pushed to the left-most position after expansion of the type synonym. This sometimes makes types less polymorphic than expected. The use of a **newtype** circumvents this problem, at the cost of introducing explicit embedding and projection functions $\downarrow$(hide) and $\uparrow$(reveal).

Using a as an abstract type ensures that only variables *bound* by a constructor Lam can be used in the constructor Var. For example, the identity function can be defined as:

$$idLambda = \downarrow (Lam\ (\lambda x \to Var\ x))$$

However the following terms are not valid, and are rejected by the type system:

$$
\begin{aligned}
invalid_1 &= \downarrow (Var\ 1) \\
invalid_2\ y &= \downarrow (Lam\ (\lambda x \to Var\ y))
\end{aligned}
$$

The first example tries to use an integer where a value of the abstract type a is expected. The second example tries to use a variable that is not (directly or indirectly) bound by a Lam and as such has a type different from the abstract type a.

Using *parametricity* [36, 37] it is possible to prove that PHOAS-encoded terms are well-scoped and do not allow bad values to be used in variable positions [3].

No explicit manipulation of environments Functions defined over PHOAS-based representations avoid the need for explicit manipulation of environments carrying the binding information. Instead, environments are implicitly handled by the meta-language. The evaluator for our lambda calculus presented in Figure 2 illustrates this. The type of the evaluator is simply $Lambda \to Value$: there is no need for an explicitly passed environment. The definition of the evaluator is mostly straightforward, although it is worth

```
data Value = VI Int | VB Bool | VF (Value → Value)
eval   :: Lambda → Value
eval e = [[↑ e]] where
  [[·]]            :: PLambda Value → Value
  [[Var v]]        = v
  [[Int n]]        = VI n
  [[Bool b]]       = VB b
  [[If e1 e2 e3]]  = case [[e1]] of
                       VB b → if b then [[e2]] else [[e3]]
  [[Add e1 e2]]    = case ([[e1]], [[e2]]) of
                       (VI x, VI y) → VI (x + y)
  [[Mult e1 e2]]   = case ([[e1]], [[e2]]) of
                       (VI x, VI y) → VI (x * y)
  [[Eq e1 e2]]     = case ([[e1]], [[e2]]) of
                       (VI x, VI y) → VB (x ≡ y)
  [[Lam f]]        = VF ([[·]] ∘ f)
  [[App e1 e2]]    = case [[e1]] of
                       VF f → f ([[e2]])
```

Figure 2. An evaluator for the PHOAS-encoded lambda calculus.

noting that the interpreter is a partial function that can raise run-time errors from failed pattern matching. The crucial step in the interpreter is to reveal (↑) the lambda term e and instantiate the abstract type with a suitable type for defining evaluation. In the case of evaluation the obvious choice for instantiation is *Value*.

Evaluation of the lambda expression $(\lambda x \to 3 + x)\ 4$ proceeds as follows:

```
t1 = ↓ (App (Lam (λx → Add (Int 3) (Var x))) (Int 4))
test = show (eval t1)   -- returns "7"
```

The PHOAS evaluator is simpler than evaluators based on first-order binding (for example variables as strings or de Bruijn indexes), which require an explicit environment to be passed around in the evaluator. While monads [38] can encapsulate the plumbing of the environment, they also give the interpreter an imperative feel and force sequentialiality on computations that could be parallel. In contrast the PHOAS interpreter is written in a purely functional style that supports simple equational reasoning.

Easy to define operations Many operations are easier to define with PHOAS than with classic HOAS. As noted by Fegaras and Sheard [14], to evaluate a version of the lambda calculus encoded with classic HOAS, an extra function (*reify*) is needed to invert the result of evaluation:

```
data Exp = L (Exp → Exp) | A Exp Exp
evalExp :: Exp → Value
evalExp (L f)       = VF (evalExp ∘ f ∘ reify)
evalExp (A e1 e2) = case evalExp e1 of
  VF f → f (evalExp e2)
reify :: Value → Exp
reify (VF f) = L (reify ∘ f ∘ evalExp)
```

More generally, classic HOAS requires inverse functions [27], but good inverse functions do not always exist or can be significantly hard to define. For example to define a pretty print function with HOAS requires an inverse parsing function with the property $print\ (parse\ x) = x$.

```
data PLambda a = Mu1 (a → PLambda a) | ...

eval :: Lambda → Value
eval e = [[↑ e]] where
  [[·]] :: PLambda Value → Value
  ...
  [[Mu1 f]] = fix ([[·]] ∘ f)

fix :: (a → a) → a
fix f = let r = f r in r   -- f (fix f)
```

Figure 3. Extending the PHOAS-encoded lambda calculus with μ-binders.

```
data PLambda a = Mu2 ([a] → [PLambda a]) | ...

eval :: Lambda → Value
eval e = [[↑ e]] where
  [[·]] :: PLambda Value → Value
  ...
  [[Mu2 f]] = head $ fix (map [[·]] ∘ f)
```

Figure 4. Extending the interpreter with a recursive multi-binder.

3. Recursive Binders using Parametric HOAS

The traditional definition of PHOAS can be extended to model recursive binding of variables, to support single or mutual recursion.

3.1 Encoding μ-binders with Parametric HOAS

One way to support recursive functions in our lambda calculus interpreter is to extend it with a recursive binder μ. With such a μ binder the factorial function can be defined as follows.

$$\mu f.\lambda n \to \textbf{if}\ (n \equiv 0)\ \textbf{then}\ 1\ \textbf{else}\ n * f\ (n - 1)$$

Figure 3 shows the extension to the calculus and interpreter in Figures 1 and 2 that is needed to encode μ binders. This extension is not very different from a conventional λ binder: it introduces a new constructor Mu_1 with the same type has Lam. However the semantics of $Mu_1\ f$ is different from a regular lambda binder: it takes the fixpoint of the composition of f with the interpreter. The fixpoint is easily encoded in Haskell by the function *fix*. The definition of *fix* exploits the call-by-need semantics of Haskell to create sharing. Another way to define fix is as $f\ (fix\ f)$, but this does not share results.

With this extension the encoding of the factorial function is:

```
fact = Mu1 (λf → Lam (λn →
  If (Eq (Var n) (Int 0))
     (Int 1)
     (Mult (Var n)
           (App (Var f) (Add (Var n) (Int (−1)))))))
test1 = ↓ (App fact (Int 7))
```

The result of running $eval\ test_1$ is 5040 (the factorial of 7).

3.2 Encoding a recursive multi-binder

A μ binder is sufficient for expressing simple recursion, but *mutual recursion* requires some additional infrastructure. Mutually recursive definitions bind several variables at once (one for each mutually recursive definition). One way to achieve this is illustrated in Figure 4. The idea is to generalize the recursive binder type in such

a way that it takes as the input a list of variables and returns a list of lambda expressions. This form of multi-binder is sufficient to express the semantics of *letrec*. The *head* of the list computed by the fixpoint is the body of *letrec*.

Consider the following example of mutual recursive binding:

$$\begin{array}{l}\textbf{let } odd \;= \lambda n \to \textbf{if } (n \equiv 0) \textbf{ then } False \textbf{ else } even\ (n-1)\\ \quad\; even = \lambda n \to \textbf{if } (n \equiv 0) \textbf{ then } True \textbf{ else } odd\ (n-1)\\ \textbf{in } odd\ 10\end{array}$$

It is encoded with a PHOAS multi-binder as follows:

```
evenodd :: Lambda
evenodd =↓ (Mu2 (λ(∼(_ : odd : even: _)) →
  [App (Var odd) (Int 10),    -- body of letrec
   Lam (λn →                  -- definition of odd
     If (Eq (Var n) (Int 0))
        (Bool False)
        (App (Var even) (Add (Var n) (Int (−1))))),
   Lam (λn →                  -- definition of even
     If (Eq (Var n) (Int 0))
        (Bool True)
        (App (Var odd) (Add (Var n) (Int (−1)))))
  ]))
```

The recursive multi-binder specifies the set of mutually recursive definitions (in this case *even* and *odd*) and also the body of *letrec* (in this case *odd* 10). Therefore, there are three lambda expressions in the output list. The first element in the list is the expression representing the body of *letrec*. The other two definitions are the expressions representing the definitions of *even* and *odd*. The input list defines names that allow recursive references to each of the definitions. Finally, a *laziness annotation* ($\sim$) is necessary to ensure that the pattern matching of the input list is not stricter than it should be. Without that annotation, evaluation of the expression diverges.

Implicit assumptions Note that there are some implicit assumptions not captured by Haskell's type system. In particular, the list computed by the fixpoint must include at least one element. If the list has no elements then taking its *head* fails. The fixpoint function *fix* was assigned type $(a \to a) \to a$, but it can be generalized to $(b \to a) \to a$ where a is any *subtype* of b. Haskell does not have subtyping, but it is meaningful to consider subtyping relations between list types. For example, if a type $[\,T\,]_n$ is defined to mean lists of values of type T with at least n items, then $[\,T\,]_n$ can be viewed as a subtype of $[\,T\,]_m$ when $m \geqslant n$. Using this notation, the type of the Mu_2 constructor could be defined to allow any input list that is at least as long as the output list (which must still have at least one element).

$$[\,a\,]_m \to [\,PLambda\ a\,]_n \qquad \textbf{where } m \geqslant n \wedge n > 0$$

This more liberal assumption allows infinite lists as input. This often makes the algorithms simpler because it is not necessary to generate an input list of the exact size of the output list.

Note that these implicit assumptions can be enforced with a type system. For example, a dependently typed language or the Haskell extension for GADTs [33] can define types for fixed-size vectors. Here we prefer to keep the code simple and more accessible to the reader. However the reader interested in extensions of structured graphs that statically ensure such size constraints can look at recent work by Oliveira and Löh [30].

4. Structured Graphs

Structured graphs use the recursive binders introduced in Section 3 to describe cyclic structures. We consider two types of cyclic structures: *cyclic streams* and *cyclic binary trees*. These two types of structures are useful to illustrate two different types of edges that arise with structured graphs: *back edges* and *cross edges*. What is interesting about cyclic streams is that they only allow back edges, whereas most other types of structures (like cyclic binary trees) also allow cross edges. Back edges are modelled with simple μ binders, while cross edges require the recursive multi-binder introduced in Section 3.2.

4.1 Cyclic Streams and Back Edges

A datatype for cyclic streams can be defined as follows:

```
data PStream a v =
    Var v
  | Mu (v → PStream a v)
  | Cons a (PStream a v)
newtype Stream a =↓ { ↑::∀v.PStream a v }
```

This datatype of streams has the usual *Cons* constructor and also PHOAS binding constructs: the variable case and the simple recursive binder. There are two possible interpretations for this datatype: an inductive and a coinductive one[1]. In the inductive interpretation, which is the one we use for most operations on streams, this datatype represents finitely representable cyclic streams such as:

$$s_1 = \downarrow (Cons\ 1\ (Mu\ (\lambda \overbrace{v \to Cons\ 2\ (Var\ v)}^{\downarrow})))$$

$$s_2 = \downarrow (Mu\ (\lambda \overbrace{v \to Cons\ 1\ (Cons\ 2\ (Var\ v))}^{\downarrow}))$$

Acyclic (and infinite) streams such as the stream of natural numbers are not representable under this interpretation. On the other hand the inductive interpretation allows us to define several useful operations like decidable equality procedures on cyclic streams. The coinductive interpretation admits acyclic infinite streams, but some operations are no longer be valid.

Note that the only types of cycles needed in structures like streams are *back edges*: edges that point to some previous point in the structures. This stems from the fact that streams are linear structures and "pointing back" is the only option.

A final remark is that the type $Stream\ \ a$ allows values like $\downarrow (Mu\ \ Var)$, which do not represent any stream. Section 5 gives a representation that prevents such junk terms.

Folds on Streams The traditional notion of a *fold* on a list can be extended to cyclic streams. A cyclic fold *visits* each node only once. Classical imperative graph algorithms normally keep a list of visited nodes to avoid visiting a node twice. With our representation of streams such bookkeeping is not necessary. For example, the function *elems* visits all the elements in a stream exactly once and returns a list of visited elements.

```
elems :: Stream a → [a]
elems = pelems ∘ ↑ where    -- a fold
  pelems :: PStream a [a] → [a]
  pelems (Var v)      = v
  pelems (Mu g)       = pelems (g [])
  pelems (Cons x xs) = x : pelems xs
```

As in the evaluation function for the PHOAS-based interpreter, an auxiliary operation *pelems* is defined over the *PStream* type. The abstract type used for variables is instantiated to $[\,a\,]$. The *Cons* case is trivial and the variable case is easy too: it simply returns the list value v. Evaluation of a back edge must visit the elements only

[1] In Haskell it is not possible to convey to the compiler which interpretation to use. However other languages, including Coq, allow for such choice.

one time. To get access to the elements of the list, the generator function g must be applied somehow. Taking the fixpoint of g is wrong because it could generate an infinite list. Instead, the empty list is passed to g, so that when a variable (a back edge) is reached it returns that empty list.

More generally the recursion pattern of such fold-like operations can be captured by the following combinator:

```
foldStream :: (a → b → b) → b → Stream a → b
foldStream f k = pfoldStream ∘ ↑ where
  pfoldStream (Var x)      = x
  pfoldStream (Mu g)       = pfoldStream (g k)
  pfoldStream (Cons x xs) = f x (pfoldStream xs)
```

which allows writing $elems$ more compactly as:

```
elems' = foldStream (:) []
```

Cyclic Folds on Streams Another class of operations definable on cyclic streams are cyclic folds. Cyclic folds allow us to define operations that use cyclic streams as if they were represented as an infinite stream. A cyclic fold combinator can be defined as follows:

```
cfoldStream :: (a → b → b) → Stream a → b
cfoldStream f = pcfoldStream ∘ ↑ where
  pcfoldStream (Var x)      = x
  pcfoldStream (Mu g)       = fix (pcfoldStream ∘ g)
  pcfoldStream (Cons x xs) = f x (pcfoldStream xs)
```

The difference to a regular fold on streams is that there is no base case. Instead, in the case for the binder the fixpoint of the function $pcfoldStream \circ g$ is provided as an argument to g and used in the variables. Examples of cyclic folds include a $toList$ operation that computes an infinite list from a cyclic stream, or a pretty printing operation (upp) that computes an infinite string representation.

```
toList = cfoldStream (:)
upp    = cfoldStream (λx s → show x ++ " : " ++ s)
```

Sharing-preserving Transformations An example of a sharing-preserving operation is the map function ($smap$) on cyclic streams:

```
smap :: (a → b) → Stream a → Stream b
smap f s = ↓ (psmap f (↑ s)) where
  psmap f (Var v)      = Var v
  psmap f (Mu g)       = Mu (psmap f ∘ g)
  psmap f (Cons x xs) = Cons (f x) (psmap f xs)
```

In the definition of $psmap$, variables are mapped to variables and the binders are mapped to binders. In other words the structure of the original stream is preserved. Only the elements change. Another difference to functions defined previously is that, because a new stream is produced as the final result, at the end the resulting $PStream\ b\ v$ is packed into a $Stream\ b$.

Structural equality Cyclic (inductive) streams always have a finite representation, so they can be compared for structural equality without danger of nontermination:

```
instance Eq a ⇒ Eq (Stream a) where
  s1 ≡ s2 = peq 0 (↑ s1) (↑ s2)

peq :: Eq a ⇒
       Int → PStream a Int → PStream a Int → Bool
peq n (Var x)      (Var y)      = x ≡ y
peq n (Mu f)       (Mu g)       = peq (n + 1) (f n) (g n)
peq n (Cons x xs) (Cons y ys) = x ≡ y ∧ peq n xs ys
peq _ _            _            = False
```

The idea for defining structural equality is to replace each variable with a fresh label, which in this case is an integer. Then the variable case just compares whether the two labels are the same. The most interesting case is the Mu case. The idea is to pass the same label (n) to the generator functions f and g and to generate a new fresh label for the next time a new label is needed. Finally, the last two cases are standard.

```
stail :: Stream a → Stream a
stail s = ↓ (joinPStream (ptail (↑ s))) where
  ptail (Cons x xs) = xs
  ptail (Mu g)      = Mu (λx →
    let phead (Mu g)      = phead (g x)
        phead (Cons y ys) = y
    in ptail (g (Cons (phead (g x)) x)))
```

Figure 5. Tail of a stream.

A Quasi-monad Structure The $PStream\ a$ type constructor has a structure similar to a monad: it supports a $return$ and $join$ (or $concat$) operations. However, $PStream\ a$ is not a functor (that is, it does not support a functorial mapping operation), failing to be a monad for this reason. The $return$ of this quasi-monad is Var

```
retPStream :: v → PStream a v
retPStream = Var
```

and the $join$ operation has a fairly straightforward definition:

```
joinPStream :: PStream a (PStream a v) → PStream a v
joinPStream (Var v)      = v
joinPStream (Mu g)       = Mu (joinPStream ∘ g ∘ Var)
joinPStream (Cons x xs) = Cons x (joinPStream xs)
```

The $joinPStream$ stream operation is useful for defining various operations on streams. For example, consider an operation $unrollStream$ that unrolls a cycle once, as in these examples:

```
* Streams > unrollStream s2
1 : 2 : Mu (λa → 1 : 2 : a)
* Streams > unrollStream (unrollStream s2)
1 : 2 : 1 : 2 : Mu (λa → 1 : 2 : a)
```

This operation can be defined using $joinPStream$ as follows:

```
unrollStream :: Stream a → Stream a
unrollStream s = ↓ (joinPStream (punroll (↑ s)))

punroll :: PStream a (PStream a v) →
           PStream a (PStream a v)
punroll (Mu g)      = g (joinPStream (Mu g))
punroll (Cons x xs) = Cons x (punroll xs)
```

Note that $punroll$ does not define a variable case (Var). This is because $punroll$ is only called at the top-level structure and a variable cannot appear there because there is no variable that can be bound. In other words it is hard to fill the ... in an expression like $\downarrow\ (Var\ \ldots)$. For the recursive binder case, when $Mu\ g$ is found, the generator function g is called to generate one level of the structure. The argument to g is the stream itself, leading to a nested stream which is collapsed using $joinPStream$. As observed by Chlipala [9] operations like $joinPStream$ can be used to effectively implement substitution of variables in binders.

Tail of a Stream So far all the operations that have been presented have remarkably simple and high-level definitions in comparison with imperative algorithms on cyclic structures. However, certain operations are not as simple to define. For example Ghani et al. [15] consider defining the $tail$ of a cyclic stream. They observe that a possible implementation of this function should rotate the stream

when the head is part of a cycle. For example, taking the tail of s_2 should result in:

$$*\ Streams > stail\ s_2$$
$$Mu\ (\lambda a \to 2 : 1 : a)$$

The implementation of a tail of streams is presented in Figure 5. The $Cons$ case is trivial, but in the Mu case the elements in the cycle must be rotated. The basic idea is to substitute $[x \mapsto Cons\ (phead\ (g\ x))\ x]$ in g. This has the effect of putting the head of the original stream $(g\ x)$ in the last element before the variable. The new stream is formed by skipping the first element and using $joinPStream$ in a final step to perform the substitution.

4.2 Cyclic Binary Trees and Cross Edges

The datatype for cyclic binary trees is as follows:

$$\begin{aligned}&\textbf{data}\ PTree\ a\ v = \\ &\quad Var\ v \\ &\mid Mu\ ([v] \to [PTree\ a\ v]) \\ &\mid Empty \\ &\mid Fork\ a\ (PTree\ a\ v)\ (PTree\ a\ v) \\ &\textbf{newtype}\ Tree\ a = \downarrow \{\uparrow :: \forall v.PTree\ a\ v\}\end{aligned}$$

The main difference to the datatype of streams (besides the tree-specific constructors $Empty$ and $Fork$) is the need for the recursive multi-binder introduced in Section 3.2. With a simple recursive binder, it is only possible to model back edges such as:

$$\begin{aligned}&t_1 = \downarrow (Mu\ (\lambda(\sim(x{:}\ \_)) \to \\ &\quad [Fork\ 1\ (Fork\ 2\ (Var\ x)\ Empty)\ (Var\ x)]))\end{aligned}$$

In this case the reference x points *back* at the root.

Expressive Cross Edges Recursive multi-binders offer several expressiveness benefits over simple recursive binders. Namely it becomes possible to express: 1) cross edges between nodes in neighbouring trees and 2) cross edges in both directions (mutual recursion). As an example illustrating this expressiveness consider the following tree:

$$\begin{aligned}&t_2 = \downarrow (Mu\ (\lambda(\sim(x : y{:}\ \_)) \to \\ &\quad [Fork\ 1\ (Var\ y)\ (Var\ x), Fork\ 2\ (Var\ x)\ (Var\ y)]))\end{aligned}$$

This tree has two cyclic references x and y for the subtrees $Fork\ 1\ (Var\ y)\ (Var\ x)$ and $Fork\ 2\ (Var\ x)\ (Var\ y)$. In the first subtree the reference x is a back edge because it points back at itself, whereas the reference y is a cross edge because it points at the neighbouring subtree. A similar thing happens in the second subtree, only this time in reverse: the reference x is a cross edge and the reference y is a back edge. Note that this example requires mutual recursion, because the two subtrees are defined in terms of each other.

Operations on Cyclic Binary Trees Nearly all the operations defined for cyclic streams have a corresponding definition on cyclic binary trees[2]. Figure 6 shows those definitions. Most operations are defined in a similar way to the equivalent operations on streams.

The main difference to the definitions on streams lies in the treatment of Mu binders. Because trees use recursive multi-binders, operations need to be generalized to account for a list of inputs and a list of outputs. To ensure that the input list has at least as many elements as the output list, we often produce an infinite list. For example, in $foldTree$ the input to the generator function g is the infinite list $repeat\ k_1$, and in peq the generator functions are provided with the list $iterate\ succ\ n$.

The operation that needs a little more extra work, in comparison to the equivalent definition on streams, is structural equality. To

[2] An exception is the stream tail operation, which is specific to streams.

Fold:

$$\begin{aligned}&foldTree :: (a \to b \to b \to b) \to b \to b \to Tree\ a \to b \\ &foldTree\ f\ k_1\ k_2\ s = trans\ (\uparrow s)\ \textbf{where} \\ &\quad trans\ (Var\ x) = x \\ &\quad trans\ (Mu\ g) = head\ (map\ trans\ (g\ (repeat\ k_1))) \\ &\quad trans\ Empty = k_2 \\ &\quad trans\ (Fork\ x\ l\ r) = f\ x\ (trans\ l)\ (trans\ r)\end{aligned}$$

Cyclic fold:

$$\begin{aligned}&cfoldTree :: (a \to b \to b \to b) \to b \to Tree\ a \to b \\ &cfoldTree\ f\ k\ s = trans\ (\uparrow s)\ \textbf{where} \\ &\quad trans\ (Var\ x) = x \\ &\quad trans\ (Mu\ g) = head\ (fix\ (map\ trans \circ g)) \\ &\quad trans\ Empty = k \\ &\quad trans\ (Fork\ x\ l\ r) = f\ x\ (trans\ l)\ (trans\ r)\end{aligned}$$

Mapping:

$$\begin{aligned}&tmap :: (a \to b) \to Tree\ a \to Tree\ b \\ &tmap\ f\ s = \downarrow (pmap\ f\ (\uparrow s))\ \textbf{where} \\ &\quad pmap\ f\ (Var\ x) = Var\ x \\ &\quad pmap\ f\ (Mu\ g) = Mu\ (map\ (pmap\ f) \circ g) \\ &\quad pmap\ f\ Empty = Empty \\ &\quad pmap\ f\ (Fork\ x\ l\ r) = Fork\ (f\ x)\ (pmap\ f\ l)\ (pmap\ f\ r)\end{aligned}$$

Structural Equality:

$$\begin{aligned}&\textbf{instance}\ Eq\ a \Rightarrow Eq\ (Tree\ a)\ \textbf{where} \\ &\quad t_1 \equiv t_2 = peq\ 0\ (\uparrow t_1)\ (\uparrow t_2) \\ &peq :: Eq\ a \Rightarrow Int \to PTree\ a\ Int \to PTree\ a\ Int \to Bool \\ &peq\ \_\ (Var\ x)\ (Var\ y) = x \equiv y \\ &peq\ n\ (Mu\ f)\ (Mu\ g) = \\ &\quad \textbf{let}\ l_1 = f\ (iterate\ succ\ n) \\ &\qquad\ \ l_2 = g\ (iterate\ succ\ n) \\ &\quad \textbf{in}\ and\ \$\ zipWith\ (peq\ (n + length\ l_1))\ l_1\ l_2 \\ &peq\ n\ Empty\ Empty = True \\ &peq\ n\ (Fork\ x_1\ l_1\ r_1)\ (Fork\ x_2\ l_2\ r_2) = \\ &\quad x_1 \equiv x_2 \wedge peq\ n\ l_1\ l_2 \wedge peq\ n\ r_1\ r_2 \\ &peq\ \_\ \_\ \_\ \_ = False\end{aligned}$$

Quasi-monadic $join$ on $PTree\ a$:

$$\begin{aligned}&pjoin :: PTree\ a\ (PTree\ a\ v) \to PTree\ a\ v \\ &pjoin\ (Var\ v) = v \\ &pjoin\ (Mu\ g) = Mu\ (map\ pjoin \circ g \circ map\ Var) \\ &pjoin\ Empty = Empty \\ &pjoin\ (Fork\ x\ l\ r) = Fork\ x\ (pjoin\ l)\ (pjoin\ r)\end{aligned}$$

Unrolling:

$$\begin{aligned}&unrollTree :: Tree\ a \to Tree\ a \\ &unrollTree\ s = \downarrow (pjoin\ (unroll\ (\uparrow s))) \\ &unroll :: PTree\ a\ (PTree\ a\ v) \to PTree\ a\ (PTree\ a\ v) \\ &unroll\ (Mu\ g) = head\ (g\ (repeat\ (pjoin\ (Mu\ g)))) \\ &unroll\ Empty = Empty \\ &unroll\ (Fork\ x\ l\ r) = Fork\ x\ (unroll\ l)\ (unroll\ r)\end{aligned}$$

Figure 6. Operations on cyclic trees.

Mapping laws:

$$\begin{aligned} smap\ id &\equiv id \\ smap\ f \circ smap\ g &\equiv smap\ (f \circ g) \\ tmap\ id &\equiv id \\ tmap\ f \circ tmap\ g &\equiv tmap\ (f \circ g) \end{aligned}$$

Fold fusion (cyclic streams):
Assume f strict, $f\ a = b$ and $f\ (g\ x\ y) = h\ x\ (f\ y)$ for all $x\ y$, then:

$$f \circ foldStream\ g\ a \equiv foldStream\ h\ b$$

Figure 7. Some laws about operations on structured graphs.

account for the fact that recursive multi-binders Mu may bind several variables at once, the next fresh variable must be updated accordingly. Since an integer is used to produce fresh variables, and it is known how many new variable labels have been generated ($length\ l_1$), the next label is $n + length\ l_1$. The elements in the two output lists l_1 and l_2 are compared by zipping the two lists with $peq\ (n + length\ l_1)$ and then checking that all comparisons have returned $True$.

A final remark concerns the interpretation of the Mu binders. Here, the $head$ of the output list has a special role by being interpreted as the root of the tree. All other trees are auxiliary definitions to model the structure of the root tree. This interpretation is similar to $letrec$ in our interpreter in Section 3.2. There, the $head$ (which represented the body of $letrec$) was also treated specially. However, another alternative interpretation is to treat all trees equality, without preference for one of them. This interpretation uses a *forest* of trees, or a multi-rooted tree. Sometimes the later interpretation is useful for working with cyclic structures. An example of this is the model for grammars in Section 6.

4.3 Reasoning about Structured Graphs

One important benefit of structured graphs is that standard functional programming reasoning techniques can be used to reason about programs. In particular properties about several of operations defined in this section are provable by structural induction.

Figure 7 illustrates adaptations of typical laws for maps and folds to their corresponding structured graph operations. All these laws are proved by structural induction on the $PStream$ and $PTree$ datatypes. To do so, the definitions, including $smap$, $tmap$ and $foldStream$, must be unfolded to reveal the underlying definitions that operate on $PStream$ or $PTree$. The structural induction itself is standard, with the exception of the Mu case.

We illustrate the proof technique in more detail on the map fusion law for $Tree$:

$$tmap\ f \circ tmap\ g \equiv tmap\ (f \circ g)$$

This equation is proved in terms of the following property on $pmap$.

$$pmap\ f \circ pmap\ g \equiv pmap\ (f \circ g)$$

We rewrite this equation to pointwise form:

$$pmap\ f\ (pmap\ g\ x) \equiv pmap\ (f \circ g)\ x$$

Now structural induction on x applies. There are 4 cases. The Var and $Empty$ cases are trivial. The $Fork$ case is standard and not interesting. The only interesting case is the Mu case:

$pmap\ f\ (pmap\ g\ (Mu\ h))$
$\equiv$ {-Definition of $pmap$ -}
$pmap\ f\ (Mu\ (map\ (pmap\ g) \circ h))$
$\equiv$ {-Definition of $pmap$ -}
$Mu\ (map\ (pmap\ f) \circ map\ (pmap\ g) \circ h)$
$\equiv$ {-map-fusion (on lists) -}
$Mu\ (map\ (pmap\ f \circ pmap\ g) \circ h)$
$\equiv$ {-Induction hypothesis -}
$Mu\ (map\ (pmap\ (f \circ g)) \circ h)$
$\equiv$ {-Definition of $pmap$ -}
$pmap\ (f \circ g)\ (Mu\ h)$

Some proofs also need parametricity arguments. This is the case for the fold fusion law for $foldStream$. The proof requires a parametricity argument stating that the values appearing in the variable case must be the same as the values passed to the generator function. However, it is possible to avoid this parametricity argument with an alternative definition of $foldStream$:

$$\begin{aligned} &foldStream :: (a \to b \to b) \to b \to Stream\ a \to b \\ &foldStream\ f\ k = pfoldStream \circ \uparrow \mathbf{where} \\ &\quad pfoldStream\ (Var\ x) \quad = k \\ &\quad pfoldStream\ (Mu\ g) \quad = pfoldStream\ (g\ ()) \\ &\quad pfoldStream\ (Cons\ x\ xs) = f\ x\ (pfoldStream\ xs) \end{aligned}$$

By instantiating the abstract type for variables to the unit type and using k directly in the variable case, we avoid the parametricity argument. Then the proof is done with a simple structural induction proof similar to the one used in the proof for $tmap$ fusion.

Finally, note that not all operations are inductive (for example cyclic folds). Nevertheless we expect that other techniques such as coinduction or fixpoint-based reasoning techniques can be used to reason about such definitions.

5. Generic Structured Graphs

The similarity between the operations on cyclic streams and trees leads naturally to the question of whether there is a more generic way to define structured graphs. This section shows that by using some lightweight *datatype-generic programming* [17] techniques it is possible to define highly reusable combinators for manipulating structured graphs of different types. These combinators provide us with a framework which end-users can use to define their own domain-specific programs using structured graphs. Because the combinators hide most of the complexity of the PHOAS-based representation they help lowering the entry cost for users. Often, using combinators, it is possible to write programs on structured graphs that are no more complex than programs on conventional algebraic datatypes.

5.1 A Generic Representation for Structured Graphs

A generic datatype for structured graphs can be defined as follows:

$$\begin{aligned} &\mathbf{data}\ Rec\ f\ a = \\ &\quad Var\ a \\ &\quad \mid Mu\ ([a] \to [f\ (Rec\ f\ a)]) \\ &\quad \mid In\ (f\ (Rec\ f\ a)) \\ &\mathbf{newtype}\ Graph\ f = \downarrow \{\uparrow :: \forall a.Rec\ f\ a\} \end{aligned}$$

This representation separates the datatype-specific parts of structured graphs from the generic binding infrastructure (the constructors Var and Mu). The idea is to parametrize the datatype-specific parts with a type-constructor f, which is a similar to the functors used in various simple datatype-generic programming approaches [17, 23].

Streams Revisited To recover streams, the type-constructor f is instantiated as follows:

```
class Functor f where
  fmap :: (a → b) → f a → f b
class (Functor f, Foldable f) ⇒ Traversable f where
  traverse :: Applicative i ⇒ (a → i b) → f a → i (f b)
```

Figure 8. The *Functor* and *Traversable* type classes.

```
data StreamF a r = Cons a r
  deriving (Functor, Foldable, Traversable)
type Stream a = Graph (StreamF a)
```

Values of this type are defined almost as those in Section 4.1. For example the cyclic stream *onetwo* $(1 : 2 : \ldots)$ is defined as follows:

```
onetwo = ↓ (Mu (λ(∼(s: _)) →
  [Cons 1 (In (Cons 2 (Var s)))]))
```

The only difference is that additional *In* constructors are needed at each recursive step.

Trees Revisited To recover cyclic trees the functor f is instantiated as follows:

```
data TreeF a r = Empty | Fork a r r
  deriving (Functor, Foldable, Traversable)
type Tree a = Graph (TreeF a)
```

An example of a cyclic tree is:

```
tree = ↓ (Mu (λ(∼(t1 : t2 : t3: _)) → [
  Fork 1 ((In (Fork 4 (Var t2) (In Empty)))) (Var t3),
  Fork 2 (Var t1) (Var t3),
  Fork 3 (Var t2) (Var t1)]))
```

Requirements on Functors Note that the definitions of *StreamF* and *TreeF* derive some classes. In general *Functor*, *Foldable* and *Traversable* instances are required for the functors f used in *Graph* f. These classes provide useful methods to define our generic combinators. For reference Figure 8 shows (simplified versions) of the *Functor* and *Traversable* classes. Here we only define the methods *fmap* and *traverse*, which are needed in Section 6. The function *fmap* is a generalization of the map function for containers, and *traverse* is an effectful variation of *fmap* for applicative effects [26]. The definitions of the *Foldable* and *Applicative* classes are omitted. Recent versions of the GHC compiler can derive the instances of *Functor*, *Foldable* and *Traversable* mechanically using an extension of the derivable type-classes mechanism. More information about those classes can be found in work by McBride and Paterson [26] or Gibbons and Oliveira [18].

Forbidding empty cycles An additional advantage of this representation is that it prevents empty cycles. With the datatypes used for streams and trees in Section 4, empty cycles such as:

```
empty = ↓ (Mu Var)
```

were allowed. One problem with such empty cycles is that there is no (infinite) stream or tree that corresponds to that value. Such *junk* values are not desirable and should be forbidden. Fortunately, our generic representation offers a good solution for this problem. The idea is to interleave the functor f with recursive occurrences of *Rec* f a. This is used in *Mu*, which requires a function of type $[v] \to [f\ (Rec\ f\ v)]$ as an argument. Meaningless expressions, e.g.

```
empty = ↓ (Mu (λ(∼(x: _)) → [Var x]))    -- type-error
```

```
gfold :: Functor f ⇒ (t → c) → (([t] → [c]) → c) →
                      (f c → c) → Graph f → c
gfold v l f = trans ∘ ↑ where
  trans (Var x) = v x
  trans (Mu g)  = l (map (f ∘ fmap trans) ∘ g)
  trans (In fa) = f (fmap trans fa)
fold :: Functor f ⇒ (f c → c) → c → Graph f → c
fold alg k = gfold id (λg → head (g (repeat k))) alg
cfold :: Functor f ⇒ (f t → t) → Graph f → t
cfold = gfold id (head ∘ fix)
sfold :: (Eq t, Functor f) ⇒ (f t → t) → t → Graph f → t
sfold alg k = gfold id (head ∘ fixVal (repeat k)) alg
fixVal :: Eq a ⇒ a → (a → a) → a
fixVal v f = if v ≡ v' then v else fixVal v' f
  where v' = f v
```

Figure 9. Generic graph folds.

are not well typed because *Var* is a constructor of *Rec* f v and not of f (*Rec* f v). In other words, a constructor of the specific structure (given by the functor f) must always be used first.

5.2 Generic Operations

The operations defined on streams or trees can be made generic.

Generalizing Folds Figure 9 shows a little library of fold-like combinators. All folds described on the figure are an instance of *gfold*. The function *gfold* generalizes several fold-like functions presented in Section 4 in 2 dimensions:

- **Graph-generic:** Rather than depending on a particular graph structure like streams or trees, *gfold* is parametrized by a functor f, which abstracts over the particular graph structure. This type of generalization is a form of datatype-generic programming, which we call 'graph-generic' instead of 'datatype-generic' to emphasize the use of structured graphs rather than plain algebraic datatypes.
- **Fixpoint-parametrized:** As illustrated in Section 4, there are a few variations of folds (for example, regular and cyclic folds). The main difference lies on the treatment of the recursive binder (*Mu*). The function *gfold* generalizes such folds by parametrizing treatment of the fixpoint using the function l.

The functions *fold*, *cfold* and *sfold* are graph-generic variations of folds, but with specific treatments of the recursive binder. The function *fold* is the graph-generic version of folds like *foldStream* or *foldTree*. Correspondingly, the function *cfold* is the graph-generic version of cyclic folds like *cfoldStream* or *cfoldTree*. The more generic combinators support simpler definitions of the *elems* and *toList* functions:

```
elems :: Stream a → [a]
elems = fold streamf2list []
toList :: Stream a → [a]
toList = cfold streamf2list
streamf2list :: StreamF a [a] → [a]
streamf2list (Cons x xs) = x : xs
```

Finally, the *sfold* function is yet another variant of fold-like operations. It uses a special fixpoint operation *fixVal*, which works for monotonic functions and values that support a comparison operation ($\equiv$). This combinator is used in Section 6.

Generic transformations on graphs:

```
type f ⇝ g = ∀a.f a → g a

transform :: (Functor f, Functor g) ⇒
             (f ⇝ g) → Graph f → Graph g
transform f x = ↓ (hmap (↑ x)) where
  hmap (Var x) = Var x
  hmap (Mu g)  = Mu (map (f ∘ fmap hmap) ∘ g)
  hmap (In x)  = In (f (fmap hmap x))
```

Generic mapping on graph containers:

```
class BiFunctor f where
  bimap :: (a → c) → (b → d) → f a b → f c d

gmap :: (BiFunctor f, Functor (f a), Functor (f b)) ⇒
        (a → b) → Graph (f a) → Graph (f b)
gmap f = transform (bimap f id)
```

Generic quasi-monadic join:

```
pjoin :: Functor f ⇒ Rec f (Rec f a) → Rec f a
pjoin (Var x) = x
pjoin (Mu g)  = Mu (map (fmap pjoin) ∘ g ∘ map Var)
pjoin (In r)  = In (fmap pjoin r)
```

Generic unrolling:

```
unrollGraph :: Functor f ⇒ Graph f → Graph f
unrollGraph g = ↓ (pjoin (unroll (↑ g)))

unroll :: Functor f ⇒ Rec f (Rec f a) → Rec f (Rec f a)
unroll (Mu g) = In (head (g (repeat (pjoin (Mu g)))))
unroll (In r) = In (fmap unroll r)
```

Figure 10. Generic graph transformations

Generalizing Transformations Figure 10 shows a little library of transformation combinators. An important operation is the *transform* function. This function transforms a graph with a structure f into a graph with a structure g using a *natural transformation* $f \rightsquigarrow g$. Note that in categorical terms the auxiliary function *hmap* is a functorial map operation, but in a category with functors as objects and natural transformations as arrows. The function *transform* can be used, for example, to convert a tree or a stream into a graph structure *VGraph* that can be rendered into a graphical representation of the corresponding graph.

```
data VGraphF a = VNode String [a]
  deriving (Show, Functor, Foldable, Traversable)

type VGraph = Graph VGraphF

btree2vgraph :: Show a ⇒ Tree a → VGraph
btree2vgraph = transform trans where
  trans Empty        = VNode "" []
  trans (Fork x l r) = VNode (show x) [l, r]
```

Another operation that can be defined with *transform* is a generic mapping operation (*gmap*) on graph containers. The *gmap* function requires container-like type constructors such as *StreamF* or *TreeF* to be instances of the class *BiFunctor*. Note that such *BiFunctor* requirements are standard for this kind of container structures [23].

Finally, a different type of transformation is a generic version of the quasi-monadic join operation (*pjoin*). The function *pjoin* is a straightforward generalization of the corresponding function on streams and trees. A generic version of unrolling (*unrollGraph*) can be defined in terms of *pjoin*. Notably the *unrollGraph* transformation alters the graph-sharing shape: in the *Mu* case a value built using a *In* data constructor is returned. This is in contrast with operations of the *transform* family, which preserve the original graph-sharing structure.

5.3 Ad-hoc Generic Operations

While many operations can be captured with generic recursion pattern combinators like *gfold* and *transform*, some operations may require less common types of recursion patterns. While it is possible to add a large number of general purpose recursion patterns to our library, this introduces some additional end-user cost because users have to learn when and how to use the recursion patterns (which is not trivial). A less general, but more pragmatic approach consists of using type-classes to divide the generic processing parts of a specific operation from the structure specific parts of that operation. We illustrate this technique on two operations: generic structural equality and generic pretty printing.

Equality A generic version of structural equality can be defined by the *geq* function:

```
geq :: EqF f ⇒ Graph f → Graph f → Bool
geq g1 g2 = eqRec 0 (↑ g1) (↑ g2)

eqRec :: EqF f ⇒ Int → Rec f Int → Rec f Int → Bool
eqRec _ (Var x) (Var y) = x ≡ y
eqRec n (Mu g) (Mu h) =
  let a = g (iterate succ n)
      b = h (iterate succ n)
  in and $ zipWith (eqF (eqRec (n + length a))) a b
eqRec n (In x)  (In y)  = eqF (eqRec n) x y
eqRec _ _       _       = False
```

The function *eqRec* deals with the generic binding structure, while the type-class *EqF* provides equality for the structure-specific parts of the graph:

```
class Functor f ⇒ EqF f where
  eqF :: (r → r → Bool) → f r → f r → Bool
```

The type r is treated as an abstract type and the recursive call to deal with values of type r is explicitly provided. This avoids leaking implementation details of equality (dealing with fresh variables) to the code users have to write. Writing instances of equality for graphs is no more difficult than writing structural equality on conventional algebraic datatypes:

```
instance Eq a ⇒ EqF (StreamF a) where
  eqF eq (Cons x xs) (Cons y ys) = x ≡ y ∧ eq xs ys
```

Pretty Printing A generic pretty printing function can be defined as follows:

```
showGraph :: ShowF f ⇒ Graph f → String
showGraph g = showRec (iterate succ 'a') (↑ g)

showRec :: ShowF f ⇒ [Char] → Rec f Char → String
showRec _ (Var c) = [c]
showRec s (Mu f) =
  let r      = f s
      (fr, s') = splitAt (length r) s
  in "Mu (\n" ++ concat
       ["  " ++ [a] ++ " => " ++ v ++ "\n" | (a, v) ←
            zip fr (map (showF (showRec s')) r)] ++ ")\n"
showRec s (In fa) = showF (showRec s) fa
```

Like structural equality the strategy is to have an additional argument (s) which keeps track of a list of fresh variables. The *Mu* case creates the list of results based on the seed, then builds a string

that maps the fresh variables to the string encoding of the results. The class *ShowF* and the operation *showF* deal with the structure-specific behavior.

```
class Functor f ⇒ ShowF f where
  showF :: (r → String) → f r → String
```

Like in *EqF* the type r is treated as an abstract type and the recursive call for dealing with recursive occurrences is explicitly passed. Instances of this class look essentially the same as the corresponding operation on a conventional algebraic datatype:

```
instance Show a ⇒ ShowF (TreeF a) where
  showF sh Empty        = "Empty"
  showF sh (Fork x l r) = "Fork " ++ show x ++
    "(" ++ sh l ++ ") (" ++ sh r ++ ")"
```

6. Application: Grammars

This section shows a concrete application of structured graphs: grammar analysis and transformations. We discuss 3 different operations on grammars: nullability, first set and normalization. One interesting aspect of dealing with grammars is that some analyses, including nullability and first sets, require a special treatment for fixpoints to ensure termination for all grammars. Normalization is also interesting because it illustrates an example of a non-trivial transformation on graph structures.

6.1 Grammars

A grammar is a collection of mutually recursive productions, where each production has a name and a pattern, which can be a terminal, the empty string, a sequence of two patterns, or an alternative of two patterns. The pattern data type is defined as follows.

```
data PatternF a = Term String | E | Seq a a | Alt a a
  deriving (Functor, Foldable, Traversable)
```

A grammar is then a mutually recursive collection of patterns, where patterns can also refer to themselves or other patterns. The references between patterns are normally expressed by naming each pattern and allowing the names, called non-terminals, to be used as a pattern. We represent the same grammar structure as a graph, where the nodes are patterns and the edges are references between patterns. Binders take the place of explicit names.

Nullability One classical analysis of a grammar is nullability [7]. Nullability determines whether a given nonterminal can produce the empty string. The analysis is defined on each specific grammar expression node: terms are not nullable, ϵ is nullable, and sequence and alternative correspond to *and* and *or* respectively.

```
nullF :: PatternF Bool → Bool
nullF (Term s)     = False
nullF E            = True
nullF (Seq g1 g2)  = g1 ∧ g2
nullF (Alt g1 g2)  = g1 ∨ g2
```

To process a complete grammar, the *nullF* analysis is applied to each expression, such that results of analyzing a pattern are propagated to each place the pattern is used. This operation is provided by the *sfold* combinator in Section 5.2. Using *sfold*, nullability analysis on grammars is defined by applying the *nullF* transformation with starting value *False*.

```
nullable = sfold nullF False
```

Note that using *cfold* instead of *sfold* to define nullability:

```
badNullable = cfold nullF
```

is problematic, because this function does not terminate for some inputs. For example a "problematic" grammar for nullability analysis is the left-recursive grammar $a \to a \mid 'x'$, represented by:

```
g = ↓ (Mu (
  λ(∼(a: _)) → [Alt (Var a) (In (Term "x"))]))
```

Using *nullable* nullability analysis terminates, but with *badNullable* it doesn't. The reason for the non-termination of *badNullable* is that it uses the generic fixpoint combinator *fix*, but nullability analysis requires a fixpoint operation that exploits monotonicity [28].

First Set One analysis can be reused in defining another analysis. This situation arises in defining the *first set* of a pattern. The first set is the set of terminals that can start sentences produced by a pattern.

The first set analysis takes nullability and first sets as input, and returns the first set. The only interesting case is for sequences, which include the first set of both subpatterns if the left pattern is nullable.

```
firstF :: PatternF (Bool, [String]) → [String]
firstF (Term s)                = [s]
firstF E                       = []
firstF (Seq (b1, a1) (_, a2)) = if b1 then a1 ∪ a2 else a1
firstF (Alt (_, a1) (_, a2))  = a1 ∪ a2
```

To define a complete analysis, the nullability and first set analysis are composed.

```
nullFirstF :: PatternF (Bool, [String]) → (Bool, [String])
nullFirstF = compose (leftPart nullF) firstF

compose f g x = (f x, g x)

leftPart :: Functor f ⇒ (f a → a) → f (a, b) → a
leftPart alg = alg ∘ fmap fst
```

Finally, running the first/nullable analysis is similar to running nullability.

```
firstSet = sfold nullFirstF (False, [])
```

Normalization A more complex operation on grammars is a simple form of grammar normalization. A grammar is normalized if each node has a simple structure, where only one sequential/alternative composition may appear on the right hand side of a rule. For example, the normalized version of the grammar $a \to$ `'x'` $a \mid$ `'y'` a is:

$a \to b \mid c$
$b \to$ `'x'` a
$c \to$ `'y'` a

Our approach to solving this problem is to define a general mechanism for creating a new graph by writing nodes one by one. The new nodes are managed by a state monad. The state of the monad is a triple (n, i, o) where n is the number of nodes that have been defined, i is the list of referenceable node identities, and o is the list of node definitions.

```
type MGraph f a = State (Int, [a], [f (Rec f a)])
```

A helper function *addNode* creates a new node, increments the node count and returns a reference to the new node.

```
addNode x = do (pos, inn, out) ← get
               put (pos + 1, inn, out ++ [x])
               return $ Var (inn !! pos)
```

The actual work of normalization is done by *normF*, which simply copies leaf patterns (terminals and epsilons), but creates new nodes for any composite patterns.

```
normF :: PatternF (Rec PatternF a) →
         MGraph PatternF a (Rec PatternF a)
normF x@(Term s) = return $ In x
normF x@E        = return $ In x
normF x          = addNode x
```

The $normF$ function is called by $normalize$, which traverses the actual graph.

```
normalize :: Graph PatternF → Graph PatternF
normalize x = ↓ (evalState (trans (↑ x)) (0, [], []))

trans (Var x) = pure (Var x)
trans (Mu g) = pure $ Mu (λl → runIt (l, g l) (scan (g l)))
trans (In s)  = traverse trans s >>= normF
scan o        = traverse (traverse trans) o >>= addNodes
```

The definitions of the auxiliary functions $runIt$ and $addNodes$ are:

```
runIt (l, out) m = evalState m (length out, l, [])
addNodes new = do
  (_, _, nodes) ← get
  return (new ++ nodes)
```

Note that unlike nullability and first set, $normalize$ is defined by pattern matching on the binding structure using the auxiliary definition $trans$. This is because the transformation required by normalization is fairly complex and it does not fit in with common recursion schemes.

7. Related Work

Throughout the paper we have already discussed a lot of related work. In this section we make a finer comparison with the closest related work and also discuss some other related work.

Representing cyclic structures using binders In comparison to previous work, our PHOAS-based representation of binders allows a unique combination of features that:

- Ensures well-scopedeness and prevents the creation of *junk* terms;
- Allows the definition of cross edges as well as back edges;
- Makes operations easy to define and without needing to unroll cycles;
- Has fairly modest requirements from the type system;
- Can be used in dependently typed systems like Coq or Agda;
- Supports both inductive and co-inductive interpretations.

Fegaras and Sheard [14] were the first to suggest representing cyclic structures using binders. However, their mixed-variant type representation has several drawbacks that are discussed in detail by Ghani et al. [15]. The most important drawbacks, which we summarize here, are 1) their Haskell-based representation does not prevent misuses of binders and variables and there are various ways to create *junk* terms; 2) the representation forces unrolling the cycles for most operations, which significantly reduces the usefulness of the approach for preserving sharing; and 3) the representation is problematic for use in dependently typed languages like Coq or Agda, which forbid mixed-variant types. To prevent *junk*, Fegaras and Sheard propose a special-purpose type-system. This is in contrast to our PHOAS based approach, which relies in parametricity instead. Nevertheless the idea of using a placeholder constructor for variables, which (in our own representation) corresponds to Var, was first used in their approach. This placeholder constructor is important to avoid the definition of inverse functions that arise when defining functions with classic HOAS approaches to binding [27] (see also the discussion in Section 2).

Ghani et al. [15] suggest an alternative to Fegaras and Sheard's binder representation that avoids mixed-variant types. However their approach does not support cross edges and it requires *nested datatypes* [6] (which are not supported in many programming languages). The lack of support for cross edges is particularly limiting since cross edges are important for most graph structures (the exceptions are linear structures like streams). Like us, they also develop combinators and they sketch a datatype-generic programming variant of their graph library. However, the use of nested datatypes, complicates the definition of the generic combinators. Folds for cyclic stream and tree structures require *higher-ranked types* [32] (as usual for nested datatypes [5, 25]) and they suggest that in a datatype-generic version one-hole contexts [1] are also needed, adding extra complexity to the approach.

Building on Ghani et al.'s work, Hamana [20] proposes an approach that deals with cross edges. However, this representation requires a dependently typed language like Agda or, alternatively, an encoding based on *generalized algebraic datatypes* [33]. To deal with cross edges Hamana uses a *path*-based approach, where the cross edges are expressed in terms of a relative path. For example the path expression $\swarrow 11 \uparrow x$ means "go up to the node labelled x and then descent twice through the left". Dependent types are used to ensure that such paths are valid by keeping track of the shape of the structure in the types. In contrast our representation relies only on well-scoped labels to deal with cross edges. It is unclear to us that Hamana's representation extends to coinductive interpretations of cyclic structures, since this seems to require potentially infinite types to model the shapes of the structure at the level of types.

Inductive representations of unstructured graphs A different line of research concerns inductive representations of graphs in a more classical sense: *unstructured* representations of nodes and edges with no constraints on the graph structure. Erwig [13] proposes an inductive representation with two constructors: an empty graph constructor (the base case); or a graph extended with a node together with its label and edges (the inductive case). Gibbons [16] proposes an initial algebra semantics for unstructured (acyclic) graphs, but he requires 6 different types of constructors for capturing various possible configurations of nodes and edges. In contrast to structured graphs, this unstructured view does not impose strong constraints in their shape of graph structures and cannot be used to enforce constraints like: streams nodes have exactly one edge; or binary trees ($Fork$) nodes to have exactly two edges.

Binding With respect to binding our work builds on Chlipala's [9] Parametric HOAS approach. In contrast to us, Chlipala does not discuss applications of PHOAS to cyclic structures nor encodings of recursive binders. Instead he is focused on the applications of PHOAS to theorem proving. There are several other approaches to binding [11, 14, 21, 39], which are closely related and influenced the development of PHOAS. However PHOAS unique combination of features (which we discussed in detail in Section 2) make this approach particularly attractive for representing binders.

Other work Hughes proposes a functional programming language extension for *lazy memo functions* [22]. This extension allows functions like map to preserve the sharing of their inputs. Because it is a language-based approach it is convenient and transparent to use. Using generic combinators it is possible to approximate similar convenience with structured graphs. However, the convenience of lazy memo functions does come at a price in terms of flexibility: it is not possible to define functions that require explicit manipulation of cycles and sharing.

Analysis and transformations on grammars have been a hot topic recently [4, 10, 12, 28]. The analysis and transformations presented in Section 6 were inspired by Might et al. [28] work on Brzozowski's [7] derivative of regular expressions. Might et al. use

laziness, memoization and fixed points to allow simple definitions of operations on grammars and provide guarantees of termination. However pointer equality is used in the implementation of memoization. This precludes referential transparency and complicates reasoning. In contrast we exploit call-by-need for the same effect of memoization and due to our explicit representation of variables we can avoid pointer equality.

8. Conclusion

Functional programming languages have excellent mechanisms to program with tree structures, but graph structures have always been a challenge. While traditional imperative approaches can be used to work with graphs, many nice properties are lost.

Structured graphs extend the nice mechanisms available in functional programming languages to graph structures. The purely functional nature of structured graphs means that conventional reasoning techniques can be used to reason about graph structures. Ultimately, we believe that structured graphs offer a practical programming model for graph structures without giving up the benefits of functional programming.

Acknowledgements We are especially in debt to Tijs van der Storm for many insightful discussions and providing motivation for this work. During part of the time that the first author spent at Austin, Tijs was working with the second author on a better way to write interpreters in a declarative style using graphs. It was greatly due to this work and our desire to find a purely functional way to express some of those ideas that lead to the work in this paper.

We are also grateful to Alex Loh, Andres Löh, Tom Schrijvers, the members of the IFIP WG2.1 and the anonymous reviewers for several comments and suggestions. This work was funded by the UT Austin-Portugal Colab Program.

References

[1] M. Abbott, T. Altenkirch, C. McBride, and N. Ghani. δ for data: Differentiating data structures. *Fundam. Inf.*, 65:1–28, 2004.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1988.

[3] R. Atkey. Syntax for free: Representing syntax with binding using parametricity. In *TLCA'09*, 2009.

[4] A. Baars, S. Doaitse Swierstra, and M. Viera. Typed transformations of typed grammars: The left corner transform. *Electron. Notes Theor. Comput. Sci.*, 253(7), 2010.

[5] R. Bird and R. Paterson. Generalised folds for nested datatypes. *Formal Aspects of Computing*, 11:11–2, 1999.

[6] R. S. Bird and L. G. L. T. Meertens. Nested datatypes. In *MPC '98*, 1998.

[7] J. A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11, 1964.

[8] P. P. Chen. The entity-relationship model–toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, 1976.

[9] A. Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *ICFP'08*, 2008.

[10] N. A. Danielsson. Total parser combinators. In *ICFP'10*, 2010.

[11] J. Despeyroux, A. Felty, and A. Hirschowitz. Higher-order abstract syntax in Coq. In *TLCA'95*, 1995.

[12] D. Devriese and F. Piessens. Explicitly recursive grammar combinators - a better model for shallow parser DSLs. In *PADL 2011*, 2011.

[13] M. Erwig. Inductive graphs and functional graph algorithms. *J. Funct. Program.*, 11, 2001.

[14] L. Fegaras and T. Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space). In *POPL'96*, 1996.

[15] N. Ghani, M. Hamana, T. Uustalu, and V Vene. Representing cyclic structures as nested datatypes. In *TFP'06*, 2006.

[16] J. Gibbons. An initial-algebra approach to directed acyclic graphs. In *MPC '95*, 1995.

[17] J. Gibbons. Datatype-generic programming. In *Spring School on Datatype-Generic Programming*, volume 4719 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007.

[18] J. Gibbons and B. C. d. S. Oliveira. The essence of the iterator pattern. *J. Funct. Program.*, 19(3-4), 2009.

[19] A. Gill. Type-safe observable sharing in Haskell. In *Haskell'09*, 2009.

[20] M. Hamana. Initial algebra semantics for cyclic sharing tree structures. *Logical Methods in Computer Science*, 6(3), 2010.

[21] F. Honsell, M. Miculan, and I. Scagnetto. An axiomatic approach to metareasoning on nominal algebras in hoas. In *ICALP '01*, 2001.

[22] J. Hughes. Lazy memo-functions. In *FPCA'85*, 1985.

[23] P. Jansson and J. Jeuring. Polyp – a polytypic programming language extension. In *POPL'97*, 1997.

[24] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. *The OCaml system (release 3.12): Documentation and user's manual*. Institut National de Recherche en Informatique et en Automatique, 2011.

[25] C. Martin, J. Gibbons, and I. Bayley. Disciplined, efficient, generalised folds for nested datatypes. *Form. Asp. Comput.*, 16, 2004.

[26] C. Mcbride and R. Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1), 2008.

[27] E. Meijer and G. Hutton. Bananas in space: extending fold and unfold to exponential types. In *FPCA'95*, 1995.

[28] M. Might, D. Darais, and D. Spiewak. Parsing with derivatives: a functional pearl. In *ICFP '11*, 2011.

[29] R. Milner, M. Tofte, R. Harper, and D. Macqueen. *The Definition of Standard ML - Revised*. The MIT Press, 1997.

[30] B. C. d. S. Oliveira and Andres Löh. Abstract syntax graphs for domain specific languages. Unpublished. Manuscript available at `http://ropas.snu.ac.kr/~bruno/papers/ASGDSL.pdf`, 2012.

[31] S. Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, 2003.

[32] S. Peyton Jones, D. Vytiniotis, S. Weirich, and M. Shields. Practical type inference for arbitrary-rank types. *J. Funct. Program.*, 17:1–82, 2007.

[33] S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *ICFP'06*, 2006.

[34] F. Pfenning and C. Elliot. Higher-order abstract syntax. In *PLDI '88*, 1988.

[35] F. Pottier. Lazy least fixed points in ML. Unpublished. Manuscript available at `http://gallium.inria.fr/~fpottier/publis/fpottier-fix.pdf`, 2009.

[36] J. C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, pages 513–523, 1983.

[37] P. Wadler. Theorems for free! In *FPCA '89*, 1989.

[38] P. Wadler. The essence of functional programming. In *POPL'92*, 1992.

[39] G. Washburn and S. Weirich. Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism. *Journal of Functional Programming*, 18:87–140, 2008.