Cycle Therapy: A Prescription for Fold and Unfold on Regular Trees

Franklyn Turbak[*][†][‡]
Wellesley College
http://cs.wellesley.edu/~fturbak/

J. B. Wells[§][*][‡][¶]
Heriot-Watt University
http://www.cee.hw.ac.uk/~jbw/

ABSTRACT

Cyclic data structures can be tricky to create and manipulate in declarative programming languages. In a declarative setting, a natural way to view cyclic structures is as denoting *regular* trees, those trees which may be infinite but have only a finite number of distinct subtrees. This paper shows how to implement the *unfold* (*anamorphism*) operator in both eager and lazy languages so as to create cyclic structures when the result is a regular tree as opposed to merely infinite lazy structures. The usual *fold* (*catamorphism*) operator when used with a strict combining function on any infinite tree yields an undefined result. As an alternative, this paper defines and show how to implement a *cycfold* operator with more useful semantics when used with a strict function on cyclic structures representing regular trees. This paper also introduces an abstract data type (*cycamores*) to simplify the use of cyclic structures representing regular trees in both eager and lazy languages. Implementions of cycamores in both SML and Haskell are presented.

1. INTRODUCTION

1.1 Cycles are Tricky to Manipulate

Cyclic structures — collections of linked nodes containing paths from some nodes back to themselves — are ubiquitous in computer science. These structures tend to be tricky to create, process, and deallocate. The ease with which these tasks can be performed depends a great deal on the programming paradigm and particular programming language employed. To motivate our work, we illustrate these points via a sequence of simple examples.

[*]Partially supported by NSF CISE/CCR ESS grant 980674{5,7}.
[†]Partially supported by NSF grant CCR–9804053.
[‡]Partially supported by Sun grant EDUD-7826-990410-US.
[§]Partially supported by NSF grant CCR–9988529.
[¶]Partially supported by EPSRC grants GR/L 36963 and GR/L 15685.

EXAMPLE 1.1. Consider the following Haskell function:

```
altsCycle n = do putStr (show (alts!!n))
                 putStr (show (alts!!(n+1)))
  where alts = 0 : 1 : alts
```

This function creates a conceptually infinite list of alternating zeroes and ones named `alts`, and displays the nth and $n+1$th elements of this list (0-based indexing). Haskell's lazy evalution not only makes the recursive definition of `alts` well-defined, but the resulting "infinite" list is implemented particularly efficiently using just two list nodes (where the second node is the tail of the first and the first node is the tail of the second). Indeed, in standard Haskell implementations, invoking `altsCycle` requires constant space independent of the magnitude of the input. □

EXAMPLE 1.2. The `altsCycle` function is trickier to express in an eager language like Scheme or SML. One approach is to simulate the laziness of Haskell, typically via a new lazy list data type implemented using a mechanism for delaying and forcing computations [1, 18]. Not only are there conceptual and run-time overheads for this simulation, but in SML it is also problematic that the recursive binding construct (`val rec`) requires the bound expression to be a manifest function abstraction. These constraints effectively dictate that recursively defined lazy lists in SML should be represented as thunks (parameterless functions):[1]

```
datatype 'a LazyNode = znil | zcons of ('a * ('a LazyList))
withtype 'a LazyList = unit -> 'a LazyNode

fun nthThunk zlst n =
  case zlst() of
    zcons(x,xs) => if (n = 0) then x else nthThunk xs (n-1)
  | _ => raise Fail ("nthThunk of empty list")

fun altsThunk n =
  let val rec alts =
        fn () => zcons (0, fn () => zcons(1, alts))
   in (print (Int.toString(nthThunk alts n));
       print (Int.toString(nthThunk alts (n+1))))
  end
```

Unfortunately, the total number of list nodes created by a call to `altsThunk` using this representation is proportional to the magnitude of its argument `n`.[2] □

EXAMPLE 1.3. Using mutable reference cells, it is possible to implement cyclic lists in SML using only a constant num-

[1]Scheme's more lenient `letrec` construct allows lazy lists to be represented as pairs with delayed second components[1].
[2]Only a constant number of these nodes are accessible at any point of the computation.

ber of nodes, as shown below.

```
datatype 'a RefNode = rnil | rcons of ('a * ('a RefList))
withtype 'a RefList = 'a RefNode ref

fun nthRef (ref(rcons(x,xs))) n =
      if (n = 0) then x else nthRef xs (n-1)
  | nthRef (ref(rnil)) n = raise Fail ("nthRef of empty list")

fun altsRef n =
    let val alts = ref rnil
        val _ = alts := rcons(0, ref(rcons(1,alts)))
     in (print (Int.toString(nthRef alts n));
         print (Int.toString(nthRef alts (n+1))))
    end
```

The key step is using assignment (`:=`) to "tie the knot" of the cycle. Note how the elegance of defining `alts` recursively in a single declaration is lost in this approach. □

EXAMPLE 1.4. Although Haskell's laziness facilitates implementing `alts` as a cyclic list in `altsCycle`, laziness alone is often insufficient for tying cyclic knots. Consider defining `alts` in Haskell via the list-generating `unfold` function [7]:

```
unfold :: (b -> Bool) -> (b -> a) -> (b -> b) -> b -> [a]
unfold p f g x
  | p x  = []
  | otherwise = f x : unfold p f g (g x)

altsUnfold n = do putStr (show (alts!!n))
                  putStr (show (alts!!(n+1)))
  where alts = unfold (\n -> False) id (\n -> mod (n+1) 2) 0
```

Since no knots are tied, representing `alts` requires at least `n` list nodes to be resident in memory at some point during the invocation of `altsUnfold`.[3] The fact that a very compact representation for `alts` is possible in no way guarantees that the compact representation will be used! □

EXAMPLE 1.5. As a final motivating example, consider the problem of finding the set of all the integers in a cyclic list. For the case of `alts` in the above examples, the set would be $\{0, 1\}$. Intuitively, this set can be viewed as the result of accumulating the node values of the cyclic list via a kind of folding process. However, this set cannot be calculated by the classical `foldl`/`foldr` operations of SML and Haskell, which effectively require processing the infinite number of nodes in (the unwinding of) `alts`.

Calculating the set of values for an arbitrary cyclic list involves traversing the list and determining when the list starts to repeat; otherwise, the computation will not terminate. This cannot be done by examining the node values alone, but requires knowing when a list node encountered previously is encountered again. Determining if two list nodes are identical (i.e., pointer equal) is an operation that is supported in Scheme, but not in SML or Haskell.[4] So the problem of collecting the elements of `alts` is unsolvable in examples 1.1, 1.2, and 1.4. Since SML supports testing the (pointer) equality of reference cells, the problem is solvable for the representation of cyclic lists in example 1.3. However, testing set membership using only an equality predicate takes time linear in the size of the set. Other operations (e.g., ordering predicates or hashing functions) are necessary for more efficient membership tests.

There is good reason to omit node equality in a declarative language: it can reduce the set of valid program transformations [9]. For instance, if Haskell were to support a Lisp-like `eq` operator on list nodes, β-reduction would no longer be meaning-preserving. E.g., `(\x -> eq(x,x)) [1,2]` should evaluate to `True`, but `eq([1,2],[1,2])` might evaluate to `False`, because independently allocated copies of the list `[1,2]` need not be pointer-equal (unless hash-consing [9, 15] is used). □

The above examples highlight the following problems involving cyclic data structures:

1. Naively generating a conceptually cyclic structure via operations like `unfold` can lead to an unbounded representation even when bounded representations exist.

2. Naively accumulating a result over a cyclic structure via operations like `foldr` can diverge rather than returning the result of interest.

3. The presence or lack of various programming language features (e.g., laziness, side effects, recursive bindings, node equality) greatly influences how easily cyclic structures are manipulated.

1.2 Contributions

In this work, we develop a theoretical foundation for manipulating *regular trees*, mathematical entities that correspond well with typical cyclic structures. Based on this foundation, we address the three problems motivated above:

1. We define an unfold function for generating (potentially infinite) trees and present a simple condition for guaranteeing that the resulting tree is regular. In practice, this condition justifies using memoization on the input domain to construct bounded representations of regular trees.

2. We specify the meaning of a fold function for accumulating results over (potentially infinite) trees. We also introduce a novel variant of fold, which we call cycfold, that can produce useful results for regular trees when used with a strict combining function; the usual fold function will always yield an undefined result in this case. In practice, cycfold can be calculated on bounded representations of regular trees via an iterative fixed point process. For suitable arguments to cycfold, the same results will be obtained for any finite representative of a given regular tree; cycfold will not expose any details about the particular finite representative on which cycfold is invoked.

3. We introduce *cycamores*, an abstract data type for regular trees that is largely insensitive to the features of the programming language in which it is embedded. Modulo details of threading state and certain infelicities involving eagerness and laziness, cycamores (including the unfold and cycfold operations) can be implemented equally well in an eager language or lazy language. We demonstrate this by presenting implementations of cycamores in both SML and Haskell.

[3]The fact that `altsUnfold` subscripts `alts` twice is important to this argument. If `altsUnfold` subsripted `alts` only once, the production and consumption of the list nodes for `alts` could be take place in lock step in a coroutining fashion, requiring only a constant number of list nodes to be accessible at any point in time. The presence of the second subscript to `alts` implies that the list nodes comprising `alts` cannot be reclaimed early.

[4]The GHC compiler for Haskell provides an experimental non-standard `memo` function which can sometimes be used for this purpose.

1.3 Related Work

There is a vast literature on programming with fold and unfold operators on a variety of data structures; references include [14, 16, 8, 2, 7, 11]. Terminology differs widely in this literature. In the terminology of [16], fold and unfold are only given the informal names of "bananas" and "lenses" for the symbols used to write them, the results of $\mathsf{fold}(\phi)$ and $\mathsf{unfold}(\psi)$ are called, respectively, *catamorphisms* and *anamorphisms*, and the terms in and out are used for what we call make and make^{-1}. Howard's [8] it (iteration) and gen (coiteration) loosely correspond to our fold and unfold, but Howard (like many other authors in type theory) uses the names fold and unfold for type-level operations that have the same role as make and make^{-1} do here. It is especially important to avoid confusion with the terminology of the "unfold-transform-fold" program transformation methodology that involves unfolding a recursive definition, performing local simplifications, and then (re)folding to make a new recursive definition [3].

Using memoization to effectively manipulate cyclic structures is not at all new. It goes at least far back as Hughes's 1985 work on *lazy memo functions* [9]. Our contribution to this dimension are (1) developing a theoretical justification for memoization based on regular trees; (2) observing that while memoization is essential for building cyclic structures via unfold, laziness is not; (3) noting that many cyclic structures can be effectively generated by unfold [14, 7, 10], the underappreciated sibling of fold; and (4) explaining the pitfalls to be avoided and the tradeoffs that must be made when doing this.

Our approach to folding over regular trees contrasts with approaches like that of Fegaras and Sheard [5], which accumulate results over a term representing a graph and which can produce different results for different representatives.

The notion of distinguishing two different forms of fold on cyclic structures has recently been independently explored in unpublished work on graph catamorphisms by Gibbons [6] and Wile [21]. Gibbons presents Haskell definitions for two folding functions on graphs, ifold and efold, which correspond to our fold and cycfold, respectively; see sections 4 and 5 for more detailed comments comparing these functions. Gibbons does not consider expressing graph folds in an eager language, nor does he develop any theory for his two folding functions.

The theory of fold builds on classical results of fixed point theory from denotational semantics (e.g., [19]). The fixed point iteration performed by cycfold is similar to that performed in traditional compiler data flow analysis; see [17] for a nice summary of of this area.

2. THE THEORY OF UNFOLD AND FOLD ON REGULAR TREES

2.1 Mathematical De£nitions

2.1.1 Numbers, Sets, Binary Relations, Posets, Cpos

Let the usual natural numbers be given as $\mathbb{N} = \{0, 1, 2, \ldots\}$. Let i, j, and n range over $\mathbb{N}$. When ω is added to the natural numbers as in $\mathbb{N} \cup \{\omega\}$, let $i < \omega$ for all $i \in \mathbb{N}$. Given any set S, let $\mathcal{P}(S) = \{\, S' \mid S' \subseteq S \,\}$.

A binary relation R is *transitive* iff $R(x, y)$ and $R(y, z)$ imply $R(x, z)$, *antisymmetric* iff $R(x, y)$ and $R(y, x)$ imply $x = y$, and *reflexive over a set* S iff $R(x, x)$ for all $x \in S$. An object y is an *upper bound* (resp. *lower bound*) of a set S w.r.t. a transitive binary relation R iff $R(x, y)$ (resp. $R(y, x)$) for all $x \in S$. Furthermore, w.r.t. a transitive, antisymmetric binary relation R, an upper (resp. lower) bound y for set S is a *least upper bound* (resp. *greatest lower bound*) for S iff $R(y, z)$ (resp. $R(z, y)$) for any upper (resp. lower) bound z for S such that $z \neq y$. Given a transitive, antisymmetric binary relation R and a set S, the expression $\bigsqcup_R S$ (resp. $\bigsqcap_R S$) denotes the least upper (resp. greatest lower) bound of S w.r.t. R if it exists and is otherwise undefined. A least upper (resp. greatest lower) bound is called a *join* (resp. *meet*).

A binary relation R is a *partial order* over a set S iff R is reflexive over S, transitive, and antisymmetric. A *poset* is a pair $\mathcal{P} = (S, R)$ of a *carrier set* S and a partial order R over S. The restriction of a binary relation R to a set S is $R \downarrow S = R \cap (S \times S)$. Given a poset $\mathcal{P} = (S, R)$, let the notation $\bigsqcap_{\mathcal{P}}$ (resp. $\bigsqcup_{\mathcal{P}}$) stand for $\bigsqcap_{R\downarrow S}$ (resp. $\bigsqcup_{R\downarrow S}$). A poset $\mathcal{P} = (S, R)$ has a *bottom* (i.e., a *least element*) iff $\bigsqcap_{\mathcal{P}} S$ is defined. In this case the symbol $\perp$ is usually used to denote $\bigsqcap_{\mathcal{P}} S$. Given a poset $\mathcal{P} = (S, R)$, a non-empty subset $X \subseteq S$ is a *chain* in $\mathcal{P}$ iff $X \neq \emptyset$ is totally ordered by R (i.e., either $R(x, y)$ or $R(y, x)$ for every $\{x, y\} \subseteq X$). A poset $\mathcal{C} = (S, R)$ is a *complete partial order (cpo)* iff $\bigsqcup_{\mathcal{C}} X$ is defined for every chain X in $\mathcal{C}$. A cpo $\mathcal{C}$ is a *pointed cpo* iff it has a bottom.

The product of two sets S_1 and S_2 is defined as usual as $S_1 \times S_2 = \{\, (x, y) \mid x \in S_1 \text{ and } y \in S_2 \,\}$. Given binary relations R_1 and R_2, their product is the relation $R = R_1 \times R_2$ such that $R((w, x), (y, z))$ iff $R_1(w, y)$ and $R_2(x, z)$. The product of two posets $\mathcal{P}_1 = (S_1, R_1)$ and $\mathcal{P}_2 = (S_2, R_2)$ is $\mathcal{P}_1 \times \mathcal{P}_2 = (S_1 \times S_2, R_1 \times R_2)$.

Given sets S_1 and S_2, define the function space $S_1 \to S_2 = \{\, f \mid f \subseteq S_1 \times S_2 \text{ and } \{(x, y), (x, z)\} \subseteq f \Rightarrow y = z \,\}$. Given a poset (S_1, R), the extension of R to functions of type $S_2 \to S_1$ for some S_2 is the relation $S_2 \to R$ such that $(S_2 \to R)(\theta, \theta')$ iff $R(\theta(x), \theta'(x))$ for all $x \in S_2$. Given a poset $\mathcal{P} = (S_1, R)$, let $S_2 \to \mathcal{P} = (S_2 \to S_1, S_2 \to R)$. Given posets $\mathcal{P} = (S, R)$ and $\mathcal{P}' = (S', R')$, a function f from S to S' is *monotone* w.r.t. $\mathcal{P}$ and $\mathcal{P}'$ iff $R(x, x')$ implies $R'(f(x), f(x'))$ for every $x, x' \in S$. Given cpos $\mathcal{C} = (S, R)$ and $\mathcal{C}' = (S', R')$, a function f from S to S' is *continuous* w.r.t. $\mathcal{C}$ and $\mathcal{C}'$ iff $f(\bigsqcup_{\mathcal{C}} X) = \bigsqcup_{\mathcal{C}'} \{\, f(x) \mid x \in X \,\}$ for every chain X in $\mathcal{C}$. Given cpos $\mathcal{C}_1 = (S_1, R_1)$ and $\mathcal{C}_2 = (S_2, R_2)$, define the space of continuous functions from $\mathcal{C}_1$ to $\mathcal{C}_2$ as $S_1 \xrightarrow[\mathcal{C}_1,\mathcal{C}_2]{\text{cont}} S_2 = \{\, f \mid f \in S_1 \to S_2 \text{ and } f \text{ is continuous w.r.t. } \mathcal{C}_1 \text{ and } \mathcal{C}_2 \,\}$ and define the cpo of continuous functions from $\mathcal{C}_1$ to $\mathcal{C}_2$ as $\mathcal{C}_1 \xrightarrow{\text{cont}} \mathcal{C}_2 = (S_1 \xrightarrow[\mathcal{C}_1,\mathcal{C}_2]{\text{cont}} S_2, S_1 \to R_2)$.

The lifting of a set S, written $S_\perp$, is the set $S \cup \{\perp\}$ where $\perp$ is a symbol used to stand for some fresh object not in S. Given sets S and S' and a function f from S to ${S'}_\perp$, let $\mathsf{dom}(f) = \{\, x \mid x \in S \text{ and } f(x) \neq \perp \,\}$. The lifting of a poset $\mathcal{P} = (S, R)$, written $\mathcal{P}_\perp = (S, R)_\perp$, is the poset $(S_\perp, R_\perp)$, where $R_\perp$ is defined so that $R_\perp(\perp, x)$ for all $x \in S$, $R_\perp(x, \perp)$ implies $x = \perp$, and $R(x, y)$ implies $R_\perp(x, y)$.

2.1.2 Sequences

A *sequence* s over some set S is a function from $\mathbb{N}$ to $S_\perp$ such that if $s(i) \neq \perp$ and $j \leq i$ then $s(j) \neq \perp$. Let S^ω be the set of all sequences over S. Given a sequence s, let the length of s be given by $|s| = \sqcup_< \{\, i \mid s(i) \neq \perp \,\}$, where $<$ is taken to work over $\mathbb{N} \cup \omega$. Let S^* be the set of all finite sequences over S, i.e., $S^* = \{\, s \mid s \in S^\omega \text{ and } |s| < \omega \,\}$. Given a poset $\mathcal{P} = (S, R)$, the extension of the partial order R to S^ω is the partial order R^ω such that $R^\omega(s, s')$ iff $|s| = |s'|$ and $R(s(i), s'(i))$ for $i < |s|$. The sequence extension of $\mathcal{P}$ is $\mathcal{P}^\omega = (S^\omega, R^\omega)$.

Let $[x_0, \ldots, x_n]$ denote the sequence s such that $s(i) = x_i$ for $i \leq n$ and $s(i) = \bot$ otherwise. In particular, $[]$ is the empty sequence. Given $s \in S^*$ and $s' \in S^\omega$, let $s;s'$ be the sequence that begins according to s and then continues with s', i.e., $(s;s')(i) = s(i)$ if $i < |s|$ and $(s;s')(i) = s'(i - |s|)$ otherwise. In the rest of this paper, where appropriate, there is an implicit coercion from S into S^ω which maps every element $x \in S$ into the length 1 sequence $[x]$. For example, using this implicit coercion, when $s \in S^\omega$ and $x \in S$ the notation $s;x$ stands for $s;[x]$ and $x;s$ stands for $[x];s$.

For any sets S_1 and S_2, the map function of type $(S_1 \to S_2) \to {S_1}^\omega \to {S_2}^\omega$ is defined so that $\mathsf{map}(\phi)(s)(i) = \phi(s(i))$ for $i < |s|$ and $|\mathsf{map}(\phi)(s)| = |s|$.

2.1.3 Labelled Trees

Let p, q range over $\mathbb{N}^*$, the set of *positions*. Given a non-empty set L of labels, a function t from $\mathbb{N}^*$ to $L_\bot$ is a (possibly infinite) *L-labelled tree* iff t satisfies all of the following conditions:

1. $t([]) \neq \bot$. (The tree t has at least one node.)
2. If $t(p;i) \neq \bot$, then $t(p) \neq \bot$. (The tree t only has a node at a position if it has a node at the appropriate parent position.)
3. If $t(p;(i + 1)) \neq \bot$, then $t(p;i) \neq \bot$. (The children of a node in t are numbered consecutively starting with 0. This condition is not essential, but it makes things more convenient.)

Let $\mathsf{Tree}(L)$ be the set of L-labelled trees and let t range over labelled trees. A tree t is *finitely branching* iff for all $p \in \mathsf{dom}(t)$ there exists some i such that $(p;i) \notin \mathsf{dom}(t)$. A tree t is *finite* iff $\mathsf{dom}(t)$ is finite. A tree t is of *finite height*, written $\mathsf{finHt}(t)$, iff there is a $k \in \mathbb{N}$ such that $|p| \leq k$ for all $p \in \mathsf{dom}(t)$. Given a tree t and a position p, let $t[p] = \{\, (q, l) \mid t(p;q) = l \,\}$. If t is a tree and $p \in \mathsf{dom}(t)$, then $t[p]$ is not $\emptyset$ and qualifies as a tree; it is called the *subtree of t at p*. Let the set of all *subtrees* of t be $\mathsf{subtrees}(t) = \{\, t' \mid \exists p.\ t[p] = t' \text{ and } t' \text{ is a tree} \,\}$. A tree t is *regular* iff $\mathsf{subtrees}(t)$ is finite.

Given a label $l \in L$ and a sequence $s \in (\mathsf{Tree}(L))^\omega$ of L-labelled trees t_0, t_1, ..., let $\mathsf{make}(l, s)$ denote the tree t such that $t([]) = l$, $t(i;p) = s(i)(p) = t_i(p)$ for $i < |s|$, and $t(i;p) = \bot$ for $i \geq |s|$. When $t = \mathsf{make}(l, s)$, let $\mathsf{label}(t) = l$ and $\mathsf{children}(t) = s$. Observe that the make function is a bijection between $\mathsf{Tree}(L)$ and $L \times ((\mathsf{Tree}(L))^\omega)$. Let make^{-1} be the inverse of make.

REMARK 2.1. The presentation in this section avoids handling the following practical issues faced in programming languages, because the extra complexity would be orthogonal to the main issues this paper addresses. The presentation can be extended to handle these issues.

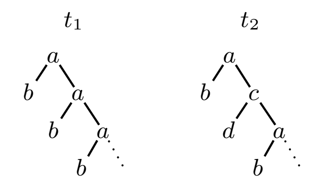
1. Although not necessary, the labels in L are often given *arities* so that a node labeled with l is required to have $\mathsf{arity}(l)$ children. More generally, the labels may have *sorts*, so that a given label produces a tree of some sort s and expects subtrees of some sorts s_0, s_1, ... Even more generally, the sorts may be parameterized. These features are needed to implement the typing restrictions on trees imposed by datatype definitions in Haskell and SML.
2. In lazy implementations (e.g., datatypes in Haskell) it is possible to have partially defined trees, where attempts to explore certain branches will diverge. The type $\mathsf{Tree}(L)$ given here does not admit partially defined trees, either finite or infinite. The presentations of unfold and fold below are simplified as a result. $\square$

REMARK 2.2. The mathematical presentation here allows infinitely branching trees. This does not make the mathematical results presented in this section any more difficult, and these results are still applicable to the finitely branching case. Due to practical difficulties implementing infinitely branching trees in eager programming languages, the SML implementation we describe later will differ on infinitely branching trees from the presentation in this section. $\square$

EXAMPLE 2.3. Let t_1 and t_2 be trees over the label set $L = \{a, b, c, d\}$ whose positions are generated by the regular expression $1^* \cdot (\epsilon|0)$ (where ϵ is the empty string, "$\cdot$" is concatenation, "$|$" is alternation, and "$*$" is Kleene closure) and whose labels are given by the rules:

$$t_1(p) = \begin{cases} a & \text{if } p \in (1^*), \\ b & \text{if } p \in (1^*0) \end{cases} \quad t_2(p) = \begin{cases} a & \text{if } p \in ((11)^*), \\ b & \text{if } b \in ((11)^*0), \\ c & \text{if } p \in ((11)^*1), \\ d & \text{if } p \in ((11)^*10) \end{cases}$$

The tree t_1 has two subtrees: t_1 itself, and $t_1[0]$, whose positions are given by the regular expression ϵ. Even though t_2 has the same positions as t_1, it has four subtrees: t_2 itself, $t_2[0]$, $t_2[1]$, and $t_2[1;0]$. As diagrams, t_1 and t_2 look like:



$\square$

2.2 Unfold

The unfold function will be treated first, because its treatment is simpler. For any non-empty sets S and L, the unfold function of type $(S \to (L \times (S^\omega))) \to S \to \mathsf{Tree}(L)$ is defined as

$$\mathsf{unfold}(\psi) = \theta \text{ where } \mathsf{isUnfold}(\psi)(\theta)$$

where the predicate $\mathsf{isUnfold}(\psi)$ is defined so that:

$$\mathsf{isUnfold}(\psi)(\theta) \Leftrightarrow \left(\theta(x) = \begin{pmatrix} \mathsf{let}\ (l, s) = \psi(x), \\ \quad s' \quad = \mathsf{map}(\theta)(s) \\ \mathsf{in}\ \mathsf{make}(l, s') \end{pmatrix} \right)$$

The definitions here do not mathematically admit the possibility for ψ or make to yield some kind of undefined value, so there is exactly one function θ satisfying $\mathsf{isUnfold}(\psi)(\theta)$. The definition of unfold is written in two parts using a separate $\mathsf{isUnfold}$ predicate in order to make clearer the symmetry with the later definition of fold.

Remark 2.4. An attempt to implement the definition of unfold given here in an eager language can easily result in an implementation of a different function. This is because (1) every type in an eager language is implicitly lifted and has an $\perp$ element indicating an undefined result which in practice is implemented by simply diverging and (2) the easy implementation of make evaluates its arguments first and is *strict*, i.e., if one of the arguments evaluates to $\perp$ then the result of make is $\perp$. The SML implementations of unfold described later in this paper have this problem for the case where the result is an irregular infinite tree, because some pragmatic issues led us to decide against using suspensions to have a lazy implementation. Our SML implementations take advantage of lemma 2.7 below in order to faithfully implement the semantics of unfold for some of the cases where the result is a regular tree. A faithful implementation of unfold in a lazy language is quite easy. However, the Haskell implementation described later is not faithful because of its approach to tying cycles in regular trees. Inspecting a subtree to the right of a subtree which our Haskell unfold fails to represent as a finite graph will diverge. □

Example 2.5. The tree t_1 in example 2.3 can be generated as follows using $S = \{0, 1\}$:

$$t_1 = \mathsf{unfold}(\psi)(0) \text{ where } \psi(i) = \begin{cases} (a, [1, 0]) & \text{if } i = 0, \\ (b, []) & \text{if } i = 1 \end{cases}$$ □

Example 2.6. The tree t_2 in example 2.3 can be generated as follows using $S = \{0, 1, 2, 3\}$:

$$t_2 = \mathsf{unfold}(\psi)(0) \text{ where } \psi(i) = \begin{cases} (a, [1, 2]) & \text{if } i = 0, \\ (b, []) & \text{if } i = 1, \\ (c, [3, 0]) & \text{if } i = 2, \\ (d, []) & \text{if } i = 3 \end{cases}$$ □

There are many interesting issues that arise in implementing unfold in various settings, but they are discussed in later sections of this paper. There is one theoretical point we wish to make. Given sets S and L, a function ψ from S to $L \times (S^\omega)$, and $x \in S$, let the *dependencies* of x w.r.t. ψ be $\mathsf{deps}(x, \psi) = S'$ where S' is the least set containing x and satisfying this closure condition:

$$(y \in S' \text{ and } \psi(y) = (l, s) \text{ and } s(i) = z \neq \perp) \Rightarrow z \in S'$$

Lemma 2.7. *Given sets S and L, a function ψ from S to $L \times (S^\omega)$, and $x \in S$, if $\mathsf{deps}(x, \psi)$ is finite, then $\mathsf{unfold}(\psi)(x)$ is a regular tree.* □

Example 2.8. The converse of lemma 2.7 does not hold. If $\mathsf{unfold}(\psi)(x)$ is a regular tree, this does not mean that $\mathsf{deps}(x, \psi)$ is finite, as illustrated by the following alternative definition of t_1 from example 2.3 using $S = \mathbb{N}$:

$$t_1 = \mathsf{unfold}(\psi)(1) \text{ where } \psi(i) = \begin{cases} (a, [0, i + 1]) & \text{if } i > 0, \\ (b, []) & \text{if } i = 0 \end{cases}$$

It should be clear that $\mathsf{deps}(1, \psi) = \mathbb{N}$, which is not finite, although t_1 is regular. □

2.3 Fold

Here we present the usual definition of the fold function on trees. In section 2.4 an alternative definition will be presented which has more useful behavior in certain situations.

Let L be a non-empty set of tree labels and let $\mathcal{C}_{\mathsf{res}} = (S, \sqsubseteq)$ be a pointed cpo intended to be used as the type of results from folding functions over trees. Let $\mathcal{C}_{\mathsf{arg}}$ be the cpo $(L, =) \times (\mathcal{C}_{\mathsf{res}}{}^{\omega})$. Let $\mathcal{C}_{\mathsf{cmb}}$ be the cpo $\mathcal{C}_{\mathsf{arg}} \xrightarrow{\text{cont}} \mathcal{C}_{\mathsf{res}}$. Let Φ be the carrier of $\mathcal{C}_{\mathsf{cmb}}$ and let ϕ range over Φ. Let $\mathcal{C}_{\mathsf{fun}}$ be the pointed cpo $(\Theta, \preceq) = (\mathsf{Tree}(L), =) \xrightarrow{\text{cont}} \mathcal{C}_{\mathsf{res}}$. Let θ range over Θ.

The fold function of type $\Phi \rightarrow \Theta$ is defined as

$$\mathsf{fold}(\phi) = \min_{\preceq}\{\, \theta \mid \mathsf{isFold}(\phi)(\theta) \,\}$$

where the predicate $\mathsf{isFold}(\phi)$ is defined so that:

$$\mathsf{isFold}(\phi)(\theta) \Leftrightarrow \left(\theta(t) = \begin{pmatrix} \mathsf{let}\ (l, s) = \mathsf{make}^{-1}(t) \\ \qquad s' \;\; = \mathsf{map}(\theta)(s) \\ \mathsf{in}\ \phi(l, s') \end{pmatrix} \right)$$

Define $\mathsf{recalc}(\phi)$ as:

$$\mathsf{recalc}(\phi)(\theta)(t) = \begin{pmatrix} \mathsf{let}\ (l, s) = \mathsf{make}^{-1}(t) \\ \mathsf{in}\ \phi(l, \mathsf{map}(\theta)(s)) \end{pmatrix}$$

It is clear that $\mathsf{isFold}(\phi)(\theta)$ iff $\mathsf{recalc}(\phi)(\theta) = \theta$, i.e., $\mathsf{isFold}(\phi)$ is true on exactly those functions that are fix points of $\mathsf{recalc}(\phi)$. It can be checked that $\mathsf{recalc}(\phi)$ is a continuous function from $\mathcal{C}_{\mathsf{fun}}$ to $\mathcal{C}_{\mathsf{fun}}$, and therefore has a least fix point because $\mathcal{C}_{\mathsf{fun}} \xrightarrow{\text{cont}} \mathcal{C}_{\mathsf{fun}}$ is a pointed cpo. Thus fold is well defined. In order to make $\mathsf{fold}(\phi)$ computable when ϕ is computable, it is sufficient to interpret the bottom of $\mathcal{C}_{\mathsf{res}}$ as divergence.

Example 2.9. This example illustrates the reason why the definition of fold needs a pointed cpo. Let $\mathsf{Bool} = \{\mathsf{true}, \mathsf{false}\}$ be the usual booleans and let b range over Bool. Let $B = \mathsf{Bool} \cup \{\star\}$. Let $L = \{x, y\}$, let $S' = L \times B$, and let $\mathcal{C}_{\mathsf{res}} = (S, \sqsubseteq) = (S', =)_{\perp}$. Let $\perp_{\mathsf{undef}}$ be the bottom added in going from S' to S. Let $\preceq$ be the partial order $\mathsf{Tree}(L) \rightarrow \sqsubseteq$. Define the function ϕ_1 as follows:

$$\begin{array}{lll} \phi_1(l, [(l', b)]) & = (l, \neg b) & \text{if } l \neq l', \\ \phi_1(l, s) & = (x, \star) & \text{otherwise, if } \forall i < |s|.\ s(i) \neq \perp_{\mathsf{undef}}, \\ \phi_1(l, s) & = \perp_{\mathsf{undef}} & \text{otherwise} \end{array}$$

Let the tree t_3 be given by this rule:

$$t_3(p) = \begin{cases} x & \text{if } p \in ((00)^*), \\ y & \text{if } p \in ((00)^*0) \end{cases}$$

Here are some of the functions satisfying $\mathsf{isFold}(\phi_1)$:

$$\begin{array}{lll} \theta_1(t_3) & = (x, \mathsf{true}), & \\ \theta_1(t_3[0]) & = (y, \mathsf{false}), & \\ \theta_1(t) & = (x, \star) & \text{otherwise} \end{array} \qquad \begin{array}{lll} \theta_2(t_3) & = (x, \mathsf{true}), & \\ \theta_2(t_3[0]) & = (y, \mathsf{false}), & \\ \theta_2(t) & = (x, \star) & \text{if } \mathsf{finHt}(t), \\ \theta_2(t) & = \perp_{\mathsf{undef}} & \text{otherwise} \end{array}$$

$$\begin{array}{lll} \theta_3(t_3) & = (x, \mathsf{false}), & \\ \theta_3(t_3[0]) & = (y, \mathsf{true}), & \\ \theta_3(t) & = (x, \star) & \text{otherwise} \end{array} \qquad \begin{array}{lll} \theta_4(t_3) & = (x, \mathsf{false}), & \\ \theta_4(t_3[0]) & = (y, \mathsf{true}), & \\ \theta_4(t) & = (x, \star) & \text{if } \mathsf{finHt}(t), \\ \theta_4(t) & = \perp_{\mathsf{undef}} & \text{otherwise} \end{array}$$

$$\theta_5(t) = (x, \star) \qquad \begin{array}{lll} \theta_6(t) & = (x, \star) & \text{if } \mathsf{finHt}(t), \\ \theta_6(t) & = \perp_{\mathsf{undef}} & \text{otherwise} \end{array}$$

Note that θ_6 is the bottom w.r.t. $\preceq$, so $\mathsf{fold}(\phi_1)(t) = \theta_6(t)$ for all t. If isFold were taken instead to be of type $((L \times (S^\omega)) \rightarrow S) \rightarrow \mathsf{Bool}$, then the only functions above satisfying $\mathsf{isFold}(\phi_1)$ would be θ_1, θ_3, and θ_5. No pair of these would be ordered by $\preceq$, so there would be no way to decide what the answers to $\mathsf{fold}(\phi_1)(t_3)$ and $\mathsf{fold}(\phi_1)(t_3[0])$ should be. □

EXAMPLE 2.10. This example illustrates that sometimes the least function satisfying $\mathsf{isFold}(\phi)$ is not the most useful one and also why the definition of fold needs the bottom of $\mathcal{C}_{\mathsf{res}}$ to be interpreted as divergence. Define the function labels to compute the set of all labels in a tree as follows:

$$\mathsf{labels}(t) = \{\, l \mid \exists t' \in \mathsf{subtrees}(t).\ \mathsf{label}(t') = l \,\}$$

The labels function seems like one that ought to be definable using fold. An attempt to do so is as follows:

$$\begin{array}{ll} \phi_{\mathsf{labels}'}(l, s) & = \{l\} \cup (\bigcup_{i<|s|} s(i)) \\ \mathsf{labels}'(t) & = \mathsf{fold}(\phi_{\mathsf{labels}'}) \end{array}$$

If $\mathcal{C}_{\mathsf{res}}$ is $(\mathcal{P}(L), \subseteq)$, this gives the correct mathematical result that $\mathsf{labels}' = \mathsf{labels}$. Unfortunately, this is not computable. An additional bottom below $\emptyset$ is needed to represent divergence.

We can instead define $\mathcal{C}_{\mathsf{res}}$ to be $(\mathcal{P}(L), =)_\perp$. Then the function $\phi_{\mathsf{labels}'}$ is of the wrong type $(L \times (\mathcal{P}(L))^\omega) \to \mathcal{P}(L)$, when instead it needs to be of type $(L \times ((\mathcal{P}(L))_\perp)^\omega) \to (\mathcal{P}(L))_\perp$. The definition of labels' can be fixed to work with fold as follows. Let $\perp_{\mathsf{undef}}$ be the element added in going from $\mathcal{P}(L)$ to $(\mathcal{P}(L))_\perp$, and make these definitions:

$$\begin{array}{ll} \phi_{\mathsf{labels}''}(l, s) = & \begin{cases} \{l\} \cup (\bigcup_{i<|s|} s(i)) & \text{if } \forall i < |s|.\ s(i) \neq \perp_{\mathsf{undef}}, \\ \perp_{\mathsf{undef}} & \text{otherwise} \end{cases} \\ \mathsf{labels}''(t) & = \mathsf{fold}(\phi_{\mathsf{labels}''}) \end{array}$$

The function labels'' is well defined and it is not hard to check that $\mathsf{labels}(t) = \mathsf{labels}''(t)$ if t is of finite height. However, $\mathsf{labels}''(t) = \perp_{\mathsf{undef}} \neq \mathsf{labels}(t)$ for *every* infinite height tree t. This is a general phenomenon: Whenever the function ϕ is strict (as $\phi_{\mathsf{labels}''}$ is), the function $\mathsf{fold}(\phi)$ will yield $\perp$ on infinite height trees. □

2.4 Cycfold

The discussion in examples 2.9 and 2.10 leads to a question: is it possible to define $\mathsf{fold}(\phi)$ in a computable way so that it returns non-trivial results on infinite-height trees when ϕ is strict? The development of an alternative to fold for this purpose is the purpose of the rest of this section. A new function cycfold will be introduced below such that $\mathsf{cycfold}(\phi)$ will coincide with $\mathsf{fold}(\phi)$ for finite trees, but for infinite regular trees cycfold may be able to find a more interesting result. The cyc part of the name refers to the fact that implementations of cycfold will take advantage of cycles in the representations of regular trees.

Before defining cycfold, some auxiliary notions are needed. Let L, $\mathcal{C}_{\mathsf{res}}$, S, $\sqsubseteq$, $\mathcal{C}_{\mathsf{arg}}$, $\mathcal{C}_{\mathsf{fun}}$, Θ, and $\preceq$ be as in the definition of fold in section 2.3. Let θ range over Θ.

The main difference is that it is required that $\mathcal{C}_{\mathsf{res}} = (S, \sqsubseteq)$ in fact is the result of lifting a pointed cpo $\mathcal{C}_{\mathsf{user}} = (S_{\mathsf{user}}, \sqsubseteq_{\mathsf{user}})$, i.e., $\mathcal{C}_{\mathsf{res}} = (\mathcal{C}_{\mathsf{user}})_\perp$. Let $\perp_{\mathsf{undef}}$ be the bottom added in going from S_{user} to S. Let $\perp_{\mathsf{user}}$ be the bottom of $\mathcal{C}_{\mathsf{user}}$. Another difference is that the combining functions are required to yield $\perp_{\mathsf{undef}}$ iff $\perp_{\mathsf{undef}}$ is one of the argument values. Let $(\Phi, R) = \mathcal{C}_{\mathsf{arg}} \xrightarrow{\text{cont}} \mathcal{C}_{\mathsf{res}}$, let $\mathsf{allowed}(\phi) \Leftrightarrow (\phi(l, s) = \perp_{\mathsf{undef}} \Leftrightarrow \exists i < |s|.\ s(i) = \perp_{\mathsf{undef}})$, and let $\Phi_{\mathsf{allowed}} = \{\, \phi \mid \phi \in \Phi \text{ and } \mathsf{allowed}(\phi) \,\}$. Let $\mathcal{C}_{\mathsf{cmb}} = (\Phi_{\mathsf{allowed}}, R)$ and let ϕ range over Φ_{allowed}.

Let $\leadsto_\phi$ be the binary relation on Θ such that:

$$\begin{array}{l} \theta \leadsto_\phi \theta' \\ \quad \Updownarrow \\ \left(\begin{array}{l} \theta' \neq \theta \\ \text{and } \exists \text{ finite } T \subset \mathsf{Tree}(L). \\ \left(\begin{array}{l} \qquad (\forall t \in (\mathsf{Tree}(L) \setminus T).\ \theta'(t) = \theta(t)) \\ \text{and } \forall t \in T. \\ \left(\begin{array}{l} \qquad (\theta'(t) = \perp_{\mathsf{user}} \text{ and } \theta(t) = \perp_{\mathsf{undef}}) \\ \text{or } \perp_{\mathsf{undef}} \neq \theta'(t) = \phi(\mathsf{label}(t), \mathsf{map}(\theta)(\mathsf{children}(t))) \end{array}\right) \end{array}\right) \end{array}\right) \end{array}$$

Let $\leadsto_\phi^*$ be the transitive, reflexive (on Θ) closure of $\leadsto_\phi$. Let $\theta_{\mathsf{undef}}(t) = \perp_{\mathsf{undef}}$ for every $t \in \mathsf{Tree}(L)$.

Lemma 2.11.

1. *$\theta_{\mathsf{undef}} \preceq \theta$ for any $\theta \in \Theta$.*
2. *If $\theta \leadsto_\phi^* \theta'$, then $\theta \preceq \theta'$.*
3. *If $\theta_{\mathsf{undef}} \leadsto_\phi^* \theta$ and $\mathsf{isFold}(\phi)(\theta')$, then for every $t \in \mathsf{Tree}(L)$ either $\theta'(t) = \perp_{\mathsf{undef}}$ or $\theta(t) \sqsubseteq \theta'(t)$.* □

Let $\mathsf{isFoldRel}(\phi)$ be the predicate such that

$$\begin{array}{l} \mathsf{isFoldRel}(\phi)(T, \theta) \\ \quad \Updownarrow \\ \forall t \in T.\ \left(\begin{array}{l} \qquad \mathsf{subtrees}(t) \subseteq T \\ \text{and } \theta(t) = \left(\begin{array}{l} \mathsf{let}\ (l, s) = \mathsf{make}^{-1}(t) \\ \mathsf{in}\ \phi(l, \mathsf{map}(\theta)(s)) \end{array}\right) \end{array}\right) \end{array}$$

The foundation has been prepared to define cycfold. Given the assumptions indicated above, the cycfold function of type $\Phi_{\mathsf{allowed}} \to \Theta$ is defined as follows:

$$\mathsf{cycfold}(\phi)(t) = \begin{cases} \theta(t) & \text{if } \theta_{\mathsf{undef}} \leadsto_\phi^* \theta, \\ & \quad \mathsf{isFoldRel}(\phi)(T, \theta), \text{ and } t \in T, \\ \perp_{\mathsf{undef}} & \text{otherwise.} \end{cases}$$

The cycfold function differs from fold only in the way that $\mathsf{cycfold}(\phi)$ picks the particular θ such that $\mathsf{isFold}(\phi)(\theta)$. The θ is found by starting with θ_{undef} and iteratively improving it until a fold is found by using ϕ to recompute the values for some number of trees using the old values for the subtrees. This approach is computable with $\perp_{\mathsf{undef}}$ given the usual interpretation of divergence and its implementation will be discussed in the rest of the paper.

EXAMPLE 2.12. Let labels and $\phi_{\mathsf{labels}''}$ be defined as in example 2.10. Let the poset $\mathcal{C}_{\mathsf{user}} = (S_{\mathsf{user}}, \sqsubseteq_{\mathsf{user}})$ required by cycfold be just $(\mathcal{P}(L), \subseteq)$. It can be checked that $\phi_{\mathsf{labels}''} \in \Phi_{\mathsf{allowed}}$. Thus, $\mathsf{labels}'''(t) = \mathsf{cycfold}(\phi_{\mathsf{labels}''})$ is well defined. It can now be checked that $\mathsf{labels}'''(t) = \mathsf{labels}(t)$ if t is a regular tree. For example, using t_2 from example 2.3, $\mathsf{labels}'''(t_2) = \{a, b, c, d\}$, the desired result. □

3. USING CYCAMORES IN SML

To investigate the practical aspects of the theory developed in the previous section, we have implemented the notions of unfold, fold, and cycfold on regular trees in both an eager language (SML) and a lazy language (Haskell). To simplify the presentation, we introduce a new abstract data type for cyclic trees which we call a *cycamore*. Discussing the interface to and implementation of what is effectively the same data type in different settings helps to highlight the essential differences between the implementations while hiding inconsequential details. It also helps us to explore the space of possible interfaces and implementations.

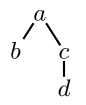
Cycamores serve a pedagogical role and are not intended as a proposal for the most elegant or efficient implementation of regular trees. To facilitate the comparison between languages, we have made design decisions that make sense in one language but seem cumbersome in the other. However, we believe that cycamores are a reasonable starting point for implementing regular tree manipulations in any language, and that the implementations presented here can effectively be specialized to particular languages.

We begin in this section by presenting the interface to cycamores in SML, and illustrating the use of this interface in various examples. In Section 4, we discuss issues in implementating the interface. In Section 5, we discuss interface and implementation issues in the context of Haskell.

3.1 SML Interface to Cycamores

Figure 1 presents the SML signature for cycamores. Conceptually, a value of the abstract type `'a Cycamore` is a potentially cyclic rose tree — i.e., a tree in which each node has two components: (1) a *label* of type `'a` and (2) a (necessarily finite) list of *children* (subtrees), each of which has type `'a Cycamore`. New cycamore nodes are constructed via `make` and are deconstructed via `view`.[5]

Example 3.1. The finite tree t_0 shown below



is built by the following expression of type `string Cycamore`:

```
val t0 = make("a", [make("b", []),
                    make("c", [make("d",[])])])
```
□

Infinite trees cannot be constructed directly via `make`, but can be constructed via `unfold`, which takes three arguments: (1) a *memoization key*; (2) a *generating function*; (3) a *seed* at which the unfolding process starts.

As we shall see in Section 4, tying structural knots in cycamores is achieved by memoizing source domain elements; the memoization key argument specifies how this is done. In the given signature, the abstract type `'a MemoKey` denotes a memoization key for type `'a`. In a typical implementation, such a memoization key might be an ordering function between two elements of type `'a`, or it might be a hashing function that maps an element of type `'a` to an integer. In the given signature, it is assumed that memoization keys are created by applying `makeMemoKey` to a comparison function of type `('a * 'a) -> order`, where `order` is a standard SML data type with elements `LESS`, `EQUAL`, and `GREATER`.

The generating function plays the role of ψ in Section 2.2. It takes an element of the source type `'a` and returns a pair `(l, ds)`, where `l` is a label of target type `'b` and `ds` is a list of *immediate dependencies* (elements of source type). For a given source value, `unfold` returns a cycamore whose label is `l` and whose children are the cycamores resulting from recursively processing `ds`.

[5]The fact that cycamores have abstract type means that we cannot use SML or Haskell's pattern matching facilities to deconstruct them – something that makes manipulating cycamores somewhat cumbersome. This problem could be addressed by using some sort of *view* mechanism (e.g., [20]).

```
signature CYCAMORE = sig
  type 'a Cycamore
  type 'a MemKey
  val make : ('a * 'a Cycamore list) -> 'a Cycamore
  val view : 'a Cycamore -> ('a * ('a Cycamore list))
  val unfold :
    'a MemKey                  (* key function *)
    -> ('a -> ('b * 'a list))  (* generating function *)
    -> 'a                      (* seed *)
    -> 'b Cycamore             (* resulting cycamore *)
  val cycfix : ('a Cycamore -> 'a Cycamore) -> 'a Cycamore
  val memofix :
     'a MemKey                 (* memoization key *)
     -> (('a -> 'b Cycamore) -> ('a -> 'b Cycamore))
                               (* function to fix over *)
     -> ('a -> 'b Cycamore)    (* resulting fixed point *)
  val fold :
    ('b -> ('a list) -> 'a)    (* combining function *)
    -> ('b Cycamore)           (* source cycamore *)
    -> 'a                      (* result *)
  val cycfold :
    'a                         (* bottom *)
    -> (('a * 'a) -> bool)     (* geq *)
    -> ('b -> ('a list) -> 'a) (* combining function *)
    -> ('b Cycamore)           (* source cycamore *)
    -> 'a                      (* result *)
  val makeMemKey : (('a * 'a) -> order) -> 'a MemKey
  val pairMemKeys : ('a MemKey) * ('b MemKey) -> ('a * 'b) MemKey
  val cycMemKey :('a Cycamore) MemKey
end
```

Figure 1: SML signature for cycamores.

Example 3.2. The infinite tree t_1 from example 2.3 can be constructed using the following function `psi_t1` that implements the function ψ in example 2.5:

```
fun psi_t1 0 = ("a", [1,0])
  | psi_t1 1 = ("b", [])
  val t1 = unfold (makeMemKey Int.compare) psi_t1 0
```
□

As a consequence of lemma 2.7, The `unfold` function is guaranteed to terminate and return a cycamore with a finite number of nodes if the transitive closure of the immediate dependencies from the original seed yields a finite set, where equality of elements in the set is determined by the comparison function that serves as the memoization key. In the case where the transitive closure of immediate dependencies yields an unbounded set, `unfold` will not terminate in SML.

Example 3.3. The translation into SML of example 2.8 is an expression which diverges upon evaluation:

```
unfold (makeMemKey Int.compare)
       (fn i => if i = 0 then ("b", []) else ("a", [0, i+1]))
       1
```
□

The `cycfix` function is handy for computing fixed points over cycamores. It takes a single function argument of type `'a Cycamore -> 'a Cycamore` and returns a cycamore that is a fixed point of this function.

Example 3.4. Another way of creating the infinite tree t_1 from example 2.3 is to use `cycfix` as follows:

```
val t1' = cycfix (fn c => make ("a", [make ("b", []), c]))
```
□

The result of `cycfix f` is well-defined as long as each use of the value of the argument of `f` within the body of `f` is guarded by a `make`. If this condition is violated, the semantics of `cycfix` is unspecified.

Example 3.5. The value of these expressions is unspecified:

```
cycfix (fn c => c)

cycfix (fn d => let val (x,[y]) = view(make ("a", [d]))
                 in y
                end)
```

In the second expression, although the parameter `d` is guarded by `make`, the alias `y` for the same value is not guarded. □

The `memofix` function is a third way to create cycles in cycamores. Given (1) a memoization key (as in `unfold`) and (2) a function f of type `(('a -> 'b Cycamore) -> ('a -> 'b Cycamore))`, it returns a function `('a -> 'b Cycamore)` that is a fixed point of f. Additionally, all calls to the parameter of f within the body of f are memoized on the domain `'a`, thereby allowing cycamore knots to be tied. The name "`memofix`" is intended to convey the intuition that its semantics is similar to `memo o fix`, where `fix` is the usual fix point operator and `memo` is the usual notion of taking a function and returning a memoized version of it.

EXAMPLE 3.6. The following function constructs a regular binary cycamore whose nodes have the values of the arguments to a version of the Fibonacci function. In this version, the base cases are replaced by subtraction modulo `n`. Since all arguments to `g` are in the range $[0..n]$, the resulting cycamore is regular.

```
fun fibModTree n =
      memofix (makeMemKey Int.compare)
              (fn g => (fn v => make (v, [g ((v-1) mod n),
                                           g ((v-2) mod n)])))
              n
```
□

The previous example is rather contrived, since it could easily be expressed via `unfold`. However, in cases where the immediate dependencies of a source value are not apparent, `memofix` can be signficantly easier to use than `unfold`.

The `memofix` function is similar in spirit to the `memo` function of the Glasgow Haskell Compiler (GHC) and the (implicit) memoization function in Hughes's work on lazy memo functions [9]. These have type `('a -> 'b) -> ('a -> 'b)`. The `memofix` function differs from these in two ways: (1) because it has the form `(('a -> 'b) -> ('a -> 'b)) -> ('a -> 'b)`, it does not rely on the recursive binding construct of the language in which it is embedded; and (2) because its purpose is to support knot tying in cycamores, the output type `'b` is constrained to be a cycamore type. Point (1) effectively circumvents SML's restriction on recursive bindings. For instance, here is an attempt to express `fibModTree` using a GHC-like `memo` function in SML:

```
(* Illegal SML example *)
fun fibModTree n =
      let val rec g =
            memo (fn v => make (v, [g ((v-1) mod n),
                                     g ((v-2) mod n)]))
       in g n
      end
```

The above declaration is not legal SML because the expression `memo` ... is not a manifest abstraction.

The `fold` function implements the "standard" notion of folding over (possibly infinite) trees presented in section 2.3. Because SML is an eager language, the first argument to fold (a combining function) is necessarily strict, and `fold` will diverge on any cycamore that represents an infinite tree. However, `fold` can return a value for finite trees.

EXAMPLE 3.7. The following function counts the nodes in a finite cycamore:

```
fun countNodes t =
      fold (fn _ => fn ns => 1+(List.foldr op+ 0 ns)) t
```

For example, `countNodes(t0)` returns 4, where `t0` is the tree defined in example 3.1. But `countNodes(t1)` diverges since `t1` stands for an infinite tree. □

The `cycfold` function implements the alternative notion of folding over regular trees developed in Section 2.4. It takes four arguments: (1) a *bottom* element of the target type `'a` that is used to prime the fixed-point process; (2) a *comparator* that determines if one target type element is greater than or equal to another; (3) a *combining function* that at each node combines the label of the node (of type `'b`) with the results of accumulating over the list of children to produce an element of target type `'a`; and (4) a *source cycamore* over which the accumulation takes place. The result returned by `cycfold` is the final value of the fixed point process accumulated at the root node of the source cycamore. An alternative design would be to return a table specifying the final values at all nodes in the cycamore.

EXAMPLE 3.8. The following `labels` function is the translation into SML of the function given in example 2.12 (see also example 2.10). It takes (1) a label comparison function of type `('a * 'a) -> order` and (2) a cycamore of type `'a Cycamore`, and returns a set of all the labels in the cycamore.

```
fun labels compare =
  let val emptySet = Set.empty compare
   in cycfold emptySet
             (fn (s1,s2) => Set.isSubset(s2,s1))
             (fn lab =>
                fn sets => Set.add(List.foldr Set.union
                                              emptySet
                                              sets,
                                   lab))
  end
```

Assume that `Set.empty compare` creates a set that uses `compare` to test element equality, and that `Set.isSubset`, `Set.add`, and `Set.union` are, respectively, the subset testing, insertion and union operators on sets. For example, `labels String.compare t1` returns the set {`"a"`, `"b"`}. □

3.2 Example: Cyclic Lists

As an extended example involving cycamores, we consider examples involving infinite lists similar to those in [9]. Figure 2 gives operations for *cyclists* — values of type `'a CycList` — which are potentially cyclic lists with elements of type `'a`. Cyclists are constructed via `cons` and `cnil` and deconstructed via the head (`hd`) and tail (`tl`) operations. The `match` construct serves as a poor man's pattern matcher for cyclists. The `toCyclicCycList` function uses `fix` to create a infinitely repeating cyclist of the given elements.

Given a memoization key k, a function f and a seed value x, `infList` k f x constructs a conceptually infinite list whose ith element (0-based) is $f^i(x)$. If the set $S = \{f^i(x) \mid i \geq 0\}$ is finite, then the result of `infList` is a regular tree and will be represented as a finite cycamore. For example, `upto` n returns an infinite repeating cyclist of the integers $0, 1, \ldots n$. Note that `infList` will diverge (or, in practice, run out of memory) if S is infinite.

The `CLmap` function maps a given function over a cyclist. It uses the special memoization key `cycMemKey` exported by the `Cycamore` module that allows cycamore nodes themselves to be memoized. The existence of `cycMemKey` means that cycamores created via `unfold` and `memofix` can

```
datatype 'a CycListLabel = Nil | Cons of 'a

type 'a CycList = 'a CycListLabel Cycamore

fun cnil () = make (Nil, [])

fun cons a b = make (Cons a, [b])

fun match cycl nullCase nodeCase =
    case view cycl of
      (Nil,[]) => nullCase()
    | (Cons(hd),[tl]) => nodeCase(hd,tl)

fun hd cycl = match cycl
                (fn () => raise Fail "head of empty CycList")
                (fn (hd,_) => hd)

fun tl cycl = match cycl
                (fn () => raise Fail "tail of empty CycList")
                (fn (_,tl) => tl)

fun toCyclicCycList xs =
    cycfix (fn c => let fun to [] = c
                          | to (y::ys) = cons y (to ys)
                     in to xs
                    end)

fun infList MemKey f = unfold MemKey (fn x => (Cons x, [f x]))

val infInts = infList (makeMemKey Int.compare)

fun upto n = infInts (fn y => (y + 1) mod n) 0

fun CLmap f = unfold (cycMemKey)
                     (fn c => (Cons (f (hd c)), [tl(c)]))

fun CLzip (xs,ys) =
    unfold (pairMemKeys(cycMemKey,cycMemKey))
           (fn (c1,c2) => (Cons (hd(c1),hd(c2)),
                           [(tl(c1),tl(c2))]))
           (xs,ys)
```

Figure 2: Cyclists implemented as SML cycamores.

themselve be used as the source domain for creating new cycamores via these functions. This closure property is extremely important in practice.

The CLzip function zips together two infinite cyclists. Given lists L_1 of length m and L_2 of length n, CLZip L_1 L_2 returns a list whose length is the least common multiple of m and n. The CLzip function uses the function pairMemKeys to construct a memoization key for a pair from the memoization keys for its components. This abstract manipulation of memoization keys means that the details of comparisons on cycamore nodes effectively remain hidden. That is, there is no direct way for a client of the Cycamore module to compare two cycamore nodes for "pointer equality". This helps to prevent small changes in the order of creation of cycamores from changing the meaning of the program and prevents clients from determining for a given cycamore its particular finite representation of the regular tree it denotes. Of course, these notions only makes sense in the context of a purely functional subset of SML; using unfold, cycfold, memofix, etc., with functions that perform assignments or raise exceptions can potentially be used to explore the internal representation of a cycamore.

4. IMPLEMENTING CYCAMORES IN SML

4.1 A Simple Implementation

We first consider a straightforward implementation of cycamores in SML, and then discuss alternative strategies.

Here is a simple representation of the Cycamore type:

```
datatype 'a CycTree = CycNode of ('a * ('a Cycamore list))
  withtype 'a Cycamore = (int * ('a CycTree option)) ref
```

In this representation, each cycamore is a mutable reference cell that contains a pair of a unique identifer (UID) and an optional tree node that pairs a label with children. The reference cell enables tying knots in cycamores. SML's reference cells can be compared via pointer equality, but cannot be arranged in a total order or used as the argument to a hash function; using them directly as keys in a memoization table would be inefficient. This problem is solved via node UIDs, which support the memoization and comparison of nodes. The node UIDs are implemented in terms of a global counter, which is bumped each time make is invoked:

```
val uidRef = ref 0

fun nextUid() = (uidRef := (! uidRef) + 1; !uidRef)

fun uidOf(ref(uid,_)) = uid

fun make(lbl,kids) = ref (nextUid(), SOME(CycNode(lbl, kids)))
```

As part of the knot-tying process, a reference representing a cycamore node is sometimes initialized to a pair of a UID and NONE. Examine the contents of such a node leads to a so-called *black hole* error common in Haskell implementations.

```
fun view cycref =
  let val (_, nodeOpt) = !cycref
   in case nodeOpt of
        (SOME (CycNode (valu,kids))) => (valu,kids)
      | NONE => raise Fail "Cyacamore black hole!"
  end
```

The workhorse for tying knots in cycamores is memofix:

```
fun ('a,'b) memofix (memKey : ('a MemKey))
                    (f : ('a -> 'b Cycamore)
                           -> ('a -> 'b Cycamore)) =
  let val mtab = MemoTable.new memKey
       fun cyc (src : 'a) =
         case MemoTable.find (src, mtab) of
           (SOME result) => result
         | NONE =>
             let val trg = ref (0, NONE)
                       val _ = MemoTable.bind(src, trg, mtab)
                 val _ = trg := (! (f cyc src))
                 (* can yield black hole if f returns trg! *)
              in trg
             end
  in cyc
  end
```

The implementation of memofix assumes the existence of a MemoTable module with stateful operations for creating a new memo table (new), inserting a key/value binding into a memo table (bind), and looking up the value associated with a key (find). Cycamore knots are tied by first binding the source value src to trg, a fresh reference cell containing the black hole token (0, NONE), and later setting this cell to the result of calling f. Note in this implementation that the memoization table mtab is shared across different calls to the returned cyc function. An alternative strategy would be to allocate a new table for each call to cyc. The above strategy enhances opportunities for sharing, but in so doing might increase space complexity.

Both fix and unfold can be implemented in terms of memofix, as show below:

```
fun 'a fix (f : 'a Cycamore -> 'a Cycamore) =
  memofix (makeMemKey (fn ((),()) => EQUAL))
          (fn g => fn x => f (g x))
          ()

fun ('a,'b) unfold (MemKey : ('a MemKey))
  (f : 'a -> (('b Cycamore list -> 'b Cycamore) * 'a list)) =
  let fun f' cyc src = let val (cnstr,deps) = f src
                        in cnstr (map cyc deps)
                       end
   in memofix MemKey f'
  end
```

The fold function is a straightforward recursive accumu-

```
fun ('a,'b) cycfold (bottom : 'a)
                    (geq : ('a * 'a) -> bool)
                    (f : 'b -> ('a list) -> 'a)
                    (cyc : 'b Cycamore) =
  let
    fun eq(x,y) = geq(x,y) andalso geq(y,x)
    fun get m x =
      case Tbl.find (m, x) of
        SOME y => y
      | NONE => raise (Fail "cycfold invariant failed!")
    val emptyTbl = Tbl.empty cycMemKey
    fun insertAll (cyc, tbl) =
      case Tbl.find (tbl, cyc) of
        SOME _ => tbl
      | NONE => let val (_, kids) = view cyc
                    val tbl' = Tbl.insert (tbl, cyc, bottom)
                 in List.foldl insertAll tbl' kids
                end
    val initialTbl = insertAll (cyc, emptyTbl)
    fun updateTbl tbl =
      let val flag = ref false
          fun updateNode (node, prev) =
            let val (lbl, kids) = view node
                val curr = f lbl (List.map (get tbl) kids)
                val changed = not (eq (prev, curr))
                val _ = (flag := (!flag orelse changed))
             in if not (geq (curr, prev)) then
                  raise (Fail "cheating in cycfold")
                else curr
            end
       in (Tbl.mapi updateNode tbl, !flag)
      end
    fun fixedPoint tbl =
      let val (tbl', changed) = updateTbl tbl
      in if changed then fixedPoint tbl' else tbl
      end
    val finalTbl = fixedPoint initialTbl
  in
    get finalTbl cyc
  end
```

Figure 3: SML implementation of `cycfold`.

lation on a rose tree:

```
fun ('a,'b) fold (f : ('b -> ('a list) -> 'a))
                 (cyc : ('b Cycamore)) =
  let val (label,kids) = view cyc
   in f label (map (fold f) kids)
  end
```

The `cycfold` function (see figure 3) is a simple (and inefficient) implementation of fixed point iteration. For each time step t of the fixed point iteration process, it builds a table mapping each cycamore node to the result accumulated at that node during iteration t. The process starts with a table in which all nodes map to the bottom value and stops when no change to a table is detected during an iteration. This process is similar to the fixed point iteration in data flow analysis [17], except that the node functions take multiple arguments rather than merging the information from multiple inputs. As in dataflow analysis, the iteration is guaranteed to terminate if the target domain is a finite lattice and the combination function is monotonic. If either of these conditions does not hold, `cycfold` may diverge.

The comparison function argument `geq` of `cycfold` is used for two purposes: (1) to determine when the fixed point iteration should stop (when used within `eq`); and (2) to check that the combination function is monotone w.r.t. the partial order of the result. Purpose (2) is not strictly necessary, but is practically helpful in debugging for catching a combining function that "cheats" by producing a new result lower than or incomparable with the previous one. If only purpose (1) is desired, an equality predicate is sufficient.[6] Indeed, this is the approach taken by Gibbons in the Haskell implementation of his efold function [6]. Modulo differences in implementation language, table representation, and means of testing termination, the implementation of efold is effectively the same as that of `cycfold`.

[6]If purpose (2) is also desired, it is more efficient (but also more cumbersome) to supply a separate equality predicate rather than simulating it with two calls to the comparator.

Memoization keys are just comparison functions:

```
type 'a MemKey = ('a * 'a) -> order
fun makeMemKey cmp = cmp
fun cycMemKey (c1,c2) = Int.compare(uidOf(c1),uidOf(c2))
fun pairMemKeys (cmp1, cmp2) ((x1,y1),(x2,y2)) =
      case cmp1(x1,x2) of
          EQUAL => cmp2(y1,y2)
        | other => other
```

Exporting `makeMemKey` but not the implementation of the `MemKey` type makes `MemKey` an abstract data type. Exporting `cycMemKey` with (abstract) type `('a Cycamore) MemKey` permits it to be used as an argument to `unfold` and `memofix`, but prohibits it from being used to directly compare cycamore nodes. The UID information encapsulated in this manner does not "leak out" to the rest of the program. The `pairMemKeys` function is just one example of a library of memoization key combinators that should be exported. Without such combinators, the `cycMemKey` black box would be rather inflexible. For example, `pairMemKeys` is essential for implementing `CLzip` in Section 3.2.

4.2 Alternative Implementations

We have experimented with several implementation strategies other than the simple one presented above. One strategy saves both space and time by using reference cells only at the points where knots need to be tied. The data type declaration for this implementation disinguishes such reference (indirect nodes) from direct nodes:

```
datatype 'a Cycamore =
  (* direct node *)
  CycNode of (int                    (* UID *)
            * 'a                     (* node value *)
            * ('a Cycamore list))    (* children *)

  (* indirect node *)
  | Indirect of 'a Cycamore option ref
```

It is an invariant of the implementation that the reference in an indirect node contains a direct node.

We have also implemented cycamores on top of a system for cyclic hash consing. In cyclic hash consing, each regular tree is guaranteed to be represented by a unique finite representation using the minimal number of nodes. We use an implementation of cyclic hash consing developed by Considine and Wells that is an improvement upon the one proposed in [15]. An intriguing feature of this implementation is that it is not necessary to hide node equality — the uniqueness of cycamore nodes representing regular trees implies that node equality is the same as tree equality. Node equality checking, which is implemented as a fast pointer-equality check in such a system, can and should be exposed in the cycamore interface for this implementation.

The implementation of `cycfold` is independent of the representation details for cycamore nodes; it only depends on `view` and the ability to index a table with cycamore nodes (which is permitted inside of but not outside of the cycamore implementation). We have experimented with two implementations of `cycfold` that are more efficient than the one presented above in terms of reducing the number of invocations of the combining function during the fixed point iteration. One strategy builds a specialized updating function on tables in an initial walk over the cycamore graph, and repeatedly executes this function starting with an initial table until no change is detected. The other strategy schedules updates of the combining function at individual nodes based on heuristics involving changes to children values since the last update. Our experiments with these strategies are still preliminary, and it is too early to report how they compare with the naive strategy and with each other.

5. CYCAMORES IN HASKELL

5.1 Implementing Cycamores in Haskell

As evidence that cycamores are a reasonable abstraction for cyclic data, we now discuss implementing cycamores in Haskell. Our implementation uses Haskell's laziness to tie knots in cycamores. The tricky part of this exercise is threading the state of a global UID counter and memoization tables through a purely functional computation.

We begin with a simple data type declaration:

```
data Cycamore a = CycNode Integer      -- UID
                          a            -- value
                          [Cycamore a] -- subtrees

instance Eq (Cycamore a) where
  (CycNode i1 v1 l1) == (CycNode i2 v2 l2) = (i1 == i2)

instance Ord (Cycamore a) where
  compare (CycNode i1 _ _) (CycNode i2 _ _) = compare i1 i2
```

As in SML, Haskell does not support user-visible pointer equality on heap allocated values, so cycamore nodes must maintain UIDs for memoization and comparison. Using Haskell's `Eq` and `Ord` type classes obviates the need to pass memoization keys explicitly to `unfold` and `memofix`. While this Haskell idiom is convenient, it unfortunately allows UID information to "leak out", making it possible for small changes in the order of creation of cycamores to affect the results. We leave plugging this leak as future work.

Haskell implementations of the core cycamore functions are presented in figure 4. While `view` is trivial, `make` is anything but. Each call to `make` needs to somehow update a global UID counter that is threaded through the computation. To do this, and also to handle the threading of memoization tables, we introduce the `Cycle` monad (figure 5). The `Cycle` monad, which is implemented in terms of the state transformer monad `ST` [13], uses a state variable (of type `STRef s Integer`) to maintain the value of the UID counter. As with the state transformer monad's `runST`, the similar `runCycle` function requires a rank-2 polymorphic type in order for type checking to guarantee the encapsulation of state.

The `make` function (see figure 4) extracts the UID counter from the monad and updates it as part of creating a new node. Note that the type signature for `make` in Haskell is necessarily different than that in SML because it must expose the threading of state in the type. Because the type `Cycle s (Cycamore a)` will appear many times below, we introduce the following abbreviation:

```
type Cycamore' s a = Cycle s (Cycamore a)
```

The `cycfix` function computes a fixed point over a cycamore but *not* over the monadic state. It is implemented

```
view :: Cycamore a -> (a, [Cycamore a])
view (CycNode _ val kids) = (val,kids)

make :: (a, [Cycamore a]) -> Cycle s (Cycamore a)
make (val, kids) =
  C(\uidRef -> do uid <- readSTRef uidRef
                  writeSTRef uidRef (uid + 1)
                  return (CycNode uid val kids))

cycfix :: (Cycamore a -> (Cycamore' s a)) -> (Cycamore' s a)
cycfix f = C( uidRef -> fixST (\x -> (unC (f x)) uidRef))

unfold :: Ord a => (a -> (b, [a])) -> a -> (Cycamore' s b)
unfold gen seed =
  do mtab <- memTabNew
     cyc mtab seed
     where cyc mt src =
             do probe <- memTabFind src mt
                case probe of
                  Just result -> return result
                  Nothing ->
                    let (label, deps) = gen src
                     in cycfix (\trg ->
                             do memTabBind src trg mt
                                deps' <- mapM (cyc mt) deps
                                trg <- make(label, deps')
                                return trg)

memofix :: Ord a => ((a -> (Cycamore' s b))
                     -> (a -> (Cycamore' s b)))
                    -> (a -> (Cycamore' s b))
memofix f start =
  do mtab <- memTabNew
     cyc mtab start
     where cyc mt src =
             do probe <- memTabFind src mt
                case probe of
                  Just result -> return result
                  Nothing ->
                    cycfix (\trg ->
                         do memTabBind src trg mt
                            -- next line can yield black hole
                            -- if f returns (lifted) trg
                            trg <- (f (cyc mt) src)
                            return trg)

fold :: (b -> [a] -> a)     -- combining function
           -> (Cycamore b)  -- source cycamore
           -> a             -- result
fold f cyc = f label (map (fold f) kids)
  where (label,kids) = view cyc

cycfold :: (POrd a)  =>
           a                  -- bottom
           -> (b -> [a] -> a)  -- combining function
           -> (Cycamore b)     -- source cycamore
           -> a                -- result
cycfold bot f cyc =
  get finalTbl cyc where
    get m x =
      case Tbl.search x m of
        Just v -> v
        Nothing -> error ("cycfold invariant failed!")
    finalTbl = fixedPoint updateTbl initialTbl
    initialTbl = insertAll cyc Tbl.empty
    insertAll cyc tbl =
      case Tbl.search cyc tbl of
        Just _ -> tbl
        Nothing ->
          foldr insertAll (Tbl.insert cyc bot tbl) kids
            where (_,kids) = view cyc
    fixedPoint f x = fp x
      where fp z = if z == z' then z else fp z'
              where z' = f z
    updateTbl m =
      Tbl.mapi (\cyc ->
                 \val ->
                    let val' = f label (map (get m) kids)
                        (label,kids) = view cyc
                    in case pordCompare val' val of
                         (Just GT) -> val'
                         (Just EQ) -> val'
                         _ -> error "cheating in cycfold"
                    )
               m
```

Figure 4: Cycamore operations in Haskell.

```
newtype Cycle s a = C { unC :: STRef s Integer -> ST s a }

instance Monad (Cycle s) where
    return = returnCycle
    (>>=) = thenCycle

returnCycle :: a -> Cycle s a
returnCycle x = C(\uidRef -> return x)

thenCycle :: Cycle s a -> (a -> Cycle s b) -> Cycle s b
thenCycle (C c) f = C(\uidRef -> do x <- c uidRef
                                    ((unC (f x)) uidRef))

runCycle :: (forall s. Cycle s a) -> a
runCycle c = runST (do uidRef <- newSTRef 0
                       ((unC c) uidRef))
```

Figure 5: The `Cycle` monad.

```
type MemTab s a b = STRef s (Tbl.Map a (Cycamore b))

memTabNew :: Cycle s (MemTab s a b)
memTabNew = C(\uidRef -> newSTRef (Tbl.empty))

memTabFind :: Ord a => a                          -- key
                   -> (MemTab s a b)              -- table
                   -> Cycle s (Maybe (Cycamore b)) -- result
memTabFind key mt =
  C(\uidRef -> do mtMap <- readSTRef mt
                  return (Tbl.search key mtMap))

memTabBind :: Ord a => a                -- key
                    -> (Cycamore b)     -- value
                    -> (MemTab s a b)   -- table
                    -> Cycle s ()       -- result
memTabBind key val mt =
  C(\uidRef -> do mtMap <- readSTRef mt
                  writeSTRef mt (Tbl.insert key val mtMap))
```

Figure 6: Memoization table interface in Haskell.

in terms of `fixST`, the fixed point function for the `ST` monad.

The implementation of `unfold` and `memofix` require memoization tables. Assuming the existence of a `Tbl` module implementing finite tables, these can be implemented as state variables in the `Cycle` monad (see figure 6). In order to tie the cyclic knot, `unfold` and `memofix` use a combination of `cycfix` and laziness. Such idioms for computing recursive values in the context of a monad are explored in [4].

Unlike the SML implementation, the Haskell implementation does not share memo tables across different invocations of `unfold f` for a given `f`. Instead, it creates a fresh memo table for each such invocation. The definition can be modified to have greater sharing, but the signature would have to change. In particular, the `(a -> (Cycamore' s b))` would need to become `(Cycle s (a -> (Cycamore' s b)))`. Similar comments hold for `memofix`.

Because the implementations of `fold` and `cycfold` are (mostly) orthogonal to the issue of threading state, they are implemented much as in SML. The Haskell implementation of `cycfold` does not take an explicit comparator argument, but instead takes an implicit one (`pordCompare`) via the following `POrd` type class, which models partial orders:

```
class (Eq a) => POrd a  where
    pordCompare :: a -> a -> Maybe Ordering
```

The `pordCompare` function returns `Just ord`, where `ord` is the ordering (`EQ`, `LT`, or `GT`) between values, or `Nothing` if the values are incomparable in the partial order.

EXAMPLE 5.1. Since Haskell is lazy, `fold` can return nontrival values for cycamores that denote infinite trees. E.g.:

```
one = Cyc.runCycle (Cyc.fix (\c -> Cyc.make("a",[c])))

factGen = (\c -> \[f] -> \ n -> if n == 0 then 1
                                else n * (f (n - 1)))

fact = Cyc.fold factGen one
```

Here, `one` is a one-node cyacamore denoting an infinite linear chain and `factGen` is a factorial generating function. The result of `Cyc.fold factGen one` is the factorial function. □

The given implementation of `fold` calculates `f` at a node every time that node is encountered in a walk over the cycamore, which is inefficient if the cycamore denotes an infinite regular tree. An alternative is to memoize the results of `f` in a table; Haskell's laziness is a big help for this purpose. This is the approach taken by Gibbons in his implementation of ifold [6]. Gibbons notes that his ifold is equivalent to the composition of a fold function on rose trees (foldtree) and a function that unwinds graphs into trees (untie).

5.2 Example: Deterministic Finite Automata

We illustrate the use of cycamores in Haskell in the context of implementing some deterministic finite automata (DFA) operations. We stress that laziness is not essential for this example; it can just as well be implemented straightforwardly using the SML implementation of cycamores.

Suppose that a DFA is represented as a list of states:

```
type DFA = [State]

data State = S(Int,           -- state number
               Bool,          -- is this an accepting state?
               [(Char,Int)]) -- transitions out of state;

instance Eq State where (S(i,_,_)) == (S(i',_,_)) = i == i'

instance Ord State where
   compare (S(i,_,_)) (S(i',_,_)) = compare i i'
```

Each state has an identifying number, a boolean indicating whether it is an accepting state, and a list of character/state number pairs indicating transitions out of the state. If a character does not appear in the list, it is assumed that a transition is taken to a distinguished non-accepting "dump state". For example, here is a sample DFA:

```
dfa1 = [S(0, True, [('a',1)]),
        S(1, True, [('a',2), ('b',0), ('c', 1)]),
        S(2, False, [('b',2), ('c',0)])]
```

Given a DFA, a state s and a character c, the following transition function returns the state that results from taking a transition from s via c (asuming that the transition is not to the dump state):

```
trans :: DFA -> State -> Char -> State
trans dfa (S(q,b,ts)) c = s
  where Just (_,q) = find (\(c',_) -> c' == c) ts
        Just s = find (\(S(q',_,_)) -> q == q') dfa
```

A DFA can also be represented as a cycamore where each node label is of type `StateLabel`:

```
data StateLabel = SL(Int,   -- state number
                     Bool,  -- is this an accepting state?
                     [Char] -- transition labels
                     )      -- (paired with children)
  deriving (Show, Eq)

type DFACyc = Cyc.Cycamore StateLabel
```

The following function converts between the list and cycamore representations of DFAs. If `dfa` is a finite list of states, `unfold` will construct a finite cycamore encoding the possibly cyclic transition graph of the automaton.

```
dfaToCyc :: DFA -> Cyc.Cycle s DFACyc
dfaToCyc dfa =
  Cyc.unfold (\(state @ (S(q,b,ts))) ->
              let chars = map fst ts
               in (SL(q,b,chars), map (trans dfa state) chars))
             (head dfa)
```

The following `strings` function returns the set of all strings whose length is less than or equal to the given number `n` that are accepted by automaton represented by the given cycamore. Here, `cycfold` is used to collect the strings accepted at each node via a fixed point iteration that starts with the empty set.

```
strings dfaCyc n =
  Cyc.cycfold Set.empty
          (\(SL(q,accept,chars)) ->
           (\sets ->
            (Set.filter (\str -> length(str) <= n)
              (foldr Set.union Set.empty
                (map (\(c, set) ->
                       let set' = Set.map (c :) set
                        in if accept
                           then Set.insert "" set'
                           else set')
                     (zip chars sets))))))
          dfaCyc
```

For example, evaluating the Haskell expression

```
strings (runCycle (dfaToCyc dfa1)) 4
```

yields the following set:

```
{"", "a", "aabc", "aac", "aaca", "ab", "aba", "abab", "abac",
 "ac", "acac", "acb", "acba", "acc", "accb", "accc"}
```

6. CONCLUSION

We have presented a theory of unfolding and folding for regular trees, and have introduced abstractions for simplifying the creation and manipulation of finite data structures that represent infinite regular trees. We have also demonstrated that these abstractions can be implemented in both eager and lazy languages. In each case, particular details of the language add complexity to the implementation. In the case of SML, the need to explicitly pass memoization keys is cumbersome, and the restriction on recursive bindings can be suffocating. In Haskell, threading the state of the global UID counter and memoization tables through the computation is awkward.

Cycamores would be easier to use if they were integrated more smoothly into particular languages. We have already noted that the lack of pattern matching on cycamores makes them unwieldy. Having to encode information like list elements in node labels is awkward. More problematic is the lack of typing. As currently designed, there is one space of untyped cycamore nodes, and all checks on labels and number of children are dynamic. We hope to address these problems in future work.

Acknowledgments

We thank Jeffrey Considine for his work on the implementation of cyclic hash-consing, and Allyn Dimock for helping to debug an early version of `memofix`. We also acknowledge the helpful support from our colleagues in the Church Project.

7. REFERENCES

[1] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs (2nd ed.).* MIT Press, 1996.

[2] R. Bird and O. de Moor. *Algebra of Programming.* Prentice Hall, 1997.

[3] R. M. Burstall and J. Darlington. A transformational system for developing recursive programs. *J. ACM*, 21(1):44–67, Jan. 1977.

[4] L. Erkok and J. Launchbury. Recursive monadic bindings. In *Proc. 2000 Int'l Conf. Functional Programming*, pp. 24–35. ACM Press, 2000.

[5] L. Fegaras and T. Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, Programs from outer space). In *Conf. Rec. POPL '96: 23rd ACM Symp. Princ. of Prog. Langs.*, pp. 284–294. ACM, 1996.

[6] J. Gibbons. Graph folds. Working Paper 804 CBB-8, IFIP WG2.1#55, Cochabamba, Bolivia. http://web.comlab.ox.ac.uk/oucl/work/jeremy.gibbons /wg21/meeting55/graphs.lhs, Jan. 2001.

[7] J. Gibbons and G. Jones. The under-appreciated unfold. In ICFP '98 [12], pp. 273–279.

[8] B. T. Howard. Inductive, coinductive, and pointed types. In *Proc. 1996 Int'l Conf. Functional Programming*, pp. 102–109. ACM Press, 1996.

[9] R. Hughes. Lazy memo-functions. In *IFIP Int'l Conf. Funct. Program. Comput. Arch.*, vol. 201 of *LNCS*, Nancy, France, 1985. Springer-Verlag.

[10] G. Hutton. Fold and unfold for program semantics. In ICFP '98 [12], pp. 280–288.

[11] G. Hutton. A tutorial on the universality and expressiveness of fold. *J. Funct. Programming*, 9(4):355–372, July 1999.

[12] *Proc. 1998 Int'l Conf. Functional Programming.* ACM Press, 1998.

[13] J. Launchbury and S. L. Peyton Jones. Lazy functional state threads. In *Proc. ACM SIGPLAN '94 Conf. Prog. Lang. Design & Impl.*, pp. 24–35, 1994.

[14] G. Malcom. Data structures and program transformation. *Sci. Comput. Programming*, 14(2–3):255–279, Oct. 1990.

[15] L. Mauborgne. Improving the representation of infinite trees to deal with sets of trees. In *Programming Languages & Systems, 9th European Symp. Programming*, vol. 1782 of *LNCS*, pp. 275–289. Springer-Verlag, 2000.

[16] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *FPCA '91, Conf. Funct. Program. Lang. Comput. Arch.*, vol. 523 of *LNCS*, pp. 124–144, Cambridge, MA. U.S.A., 1991. Springer-Verlag.

[17] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer, 1999.

[18] L. C. Paulson. *ML for the Working Programmer (2nd ed.).* Cambridge University Press, 1996.

[19] D. Schmidt. *Denotational Semantics.* Allyn and Bacon, 1986.

[20] P. Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Conf. Rec. 14th Ann. ACM Symp. Princ. of Prog. Langs.*, pp. 307–313, 1987.

[21] D. Wile. Graph catamorphisms. Abstract of talk given at IFIP Working Group 2.1 55th meeting, Cochabamba, Bolivia. http://web.comlab.ox.ac.uk /oucl/work/jeremy.gibbons/wg21/meeting55/minutes.html, Jan. 2001.