Cycle Therapy

A Prescription for Fold and Unfold on Regular Trees

Franklyn Turbak

Wellesley College

J. B. Wells

Heriot-Watt University
Cyclic Structures Are Ubiquitous

```
fn 0 => 1
| n => n * (fact (n - 1))
```

```
env
code
```

```
S
R
```
Cycles Are Tricky To Manipulate

Consider Haskell’s $\text{alts} = 0:1:\text{alts}$

- Naïve generation $\Rightarrow$ unbounded structures:
  
  ```haskell
  let inf x y = x:(inf y x) in inf 2 3
  ```

  ![Diagram of unbounded structures]

- Naïve accumulation $\Rightarrow$ divergence:
  
  ```haskell
  foldr (+) 0 alts
  foldr Set.insert Set.empty alts
  ```

- Dependency on language features: laziness, side effects, node equality, recursive binding constructs, etc.
Road Map

- Viewing cyclic structures as infinite regular trees.
- Adapting the tree-generating `unfold` function to generate cyclic structures for infinite regular trees.
- Adapting the tree-accumulating `fold` function to return non-trivial results for strict combining functions and infinite regular trees.
- Cycamores: an abstraction for manipulating regular trees that we have implemented in ML and Haskell.
A tree is *regular* if it has a finite number of distinct subtrees.
Cyclic Representatives

Finite cyclic graphs denote infinite regular trees. The same tree may be represented by many finite graphs.

<table>
<thead>
<tr>
<th>Infinite Regular Tree</th>
<th>Some Finite Cyclic Representatives</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Infinite Regular Tree Diagram" /></td>
<td><img src="image2" alt="Some Finite Cyclic Representatives Diagram" /></td>
</tr>
</tbody>
</table>
Goals

Develop high-level abstractions for creating and manipulating regular trees that:

- efficiently represent regular trees using cyclic graphs;
- do not expose the finite representative denoting an infinite regular tree;
- are relatively insensitive to the features of the programming language in which they are embedded.
Road Map

- Viewing cyclic structures as infinite regular trees.
- Adapting the tree-generating `unfold` function to generate cyclic structures for infinite regular trees.
- Adapting the tree-accumulating `fold` function to return non-trivial results for strict combining functions and infinite regular trees.
- Cycamores: an abstraction for manipulating regular trees that we have implemented in ML and Haskell.
Tree Generation via Unfold

The unfold operator generates a tree from a generating function and a seed.

\[
\text{unfold} : (S \rightarrow (L \times (S^\omega)))) \rightarrow S \rightarrow \text{Tree}(L)
\]

- generating function \(\psi\)
- seed
- trees over \(L\)
### Unfold Example 1: Regular Tree

<table>
<thead>
<tr>
<th>Generating Function $\psi$</th>
<th>Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
### Unfold Example 1: Regular Tree

#### Generating Function $\psi$

<table>
<thead>
<tr>
<th>$n$</th>
<th>$n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

#### Tree

- Root node $0$
- Node 0 connected to 0
- Node 0 connected to 1
- Node 1
Unfold Example 1: Regular Tree

Generating Function $\psi$

Tree

<table>
<thead>
<tr>
<th>Generating Function $\psi$</th>
<th>Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Unfold Example 1: Regular Tree

<table>
<thead>
<tr>
<th>Generating Function $\psi$</th>
<th>Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
### Unfold Example 1: Regular Tree

<table>
<thead>
<tr>
<th>Generating Function $\psi$</th>
<th>Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td><img src="tree_diagram.png" alt="Tree Diagram" /></td>
</tr>
<tr>
<td>$\n$</td>
<td><img src="tree_diagram.png" alt="Tree Diagram" /></td>
</tr>
<tr>
<td>$0$  $1$</td>
<td><img src="tree_diagram.png" alt="Tree Diagram" /></td>
</tr>
</tbody>
</table>

$$\text{deps}(0, \psi) = \{0, 1\}$$
Unfold Example 2: Non-regular Tree

<table>
<thead>
<tr>
<th>Generating Function $\psi$</th>
<th>Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>$0$</td>
</tr>
<tr>
<td>$2n$</td>
<td></td>
</tr>
<tr>
<td>$2n+1$</td>
<td></td>
</tr>
</tbody>
</table>
Unfold Example 2: Non-regular Tree

Generating Function $\psi$

<table>
<thead>
<tr>
<th>$n$</th>
<th>$2n$</th>
<th>$2n+1$</th>
</tr>
</thead>
</table>

Tree

0

1
Unfold Example 2: Non-regular Tree

<table>
<thead>
<tr>
<th>Generating Function $\psi$</th>
<th>Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>0</td>
</tr>
<tr>
<td>2n</td>
<td>0</td>
</tr>
<tr>
<td>2n+1</td>
<td>1</td>
</tr>
<tr>
<td>0  1</td>
<td>2</td>
</tr>
<tr>
<td>0  1</td>
<td>3</td>
</tr>
</tbody>
</table>
Unfold Example 2: Non-regular Tree

<table>
<thead>
<tr>
<th>Generating Function $\psi$</th>
<th>Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td></td>
</tr>
<tr>
<td>$2n$</td>
<td></td>
</tr>
<tr>
<td>$2n+1$</td>
<td></td>
</tr>
<tr>
<td>$0$</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>
Unfold Example 2: Non-regular Tree

<table>
<thead>
<tr>
<th>Generating Function $\psi$</th>
<th>Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td></td>
</tr>
<tr>
<td>$2n$</td>
<td></td>
</tr>
<tr>
<td>$2n+1$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$$\text{deps}(0, \psi) = \{0, 1, 2, 3, 4, 5, 6, 7, \ldots\}$$
Unfold Lemma

If $\text{deps}(x, \psi)$ is finite, then $\text{unfold}(\psi)(x)$ is a regular tree.

- Converse of this lemma does not hold.
- Basis for implementation of $\text{unfold}$ that “ties cyclic knots” for (some) regular trees via memoization on seeds (a la Hughes’s *Lazy Memo Functions*, FPCA’85).
generating fcn.: \[ \text{fun P n} = (n, [(n+1) \mod 2]) \]

initial seed: \[ 0 \]

\[ \text{unfold P 0} \]
Unfold Implementation: Standard ML

generating fcn.: \texttt{fun} P n = (n, [(n+1) \mod 2])

initial seed : 0

\texttt{unfold} P 0

Memo Table
Unfold Implementation: Standard ML

generating fcn.: \( \text{fun} \ P \ n = (n, [(n+1) \mod 2]) \)

initial seed : 0

\[
\begin{array}{c}
\text{Memo Table} \\
\text{unfold } P \ 0 \\
\text{unfold } P \ 1 \\
\text{NONE} \\
\text{CycNode( 0, [ ])}
\end{array}
\]
Unfold Implementation: Standard ML

generating fcn.: \[
\text{fun } P \ n = (n, [(n+1) \mod 2])
\]
initial seed : 0

CycNode( 0, [] )

Memo Table

unfold P 0

unfold P 1

NONE

NONE
Unfold Implementation: Standard ML

generating fcn.: \[ \text{fun } P \ n = (n, [(n+1) \mod\ 2]) \]

initial seed : 0

Memo Table

CycNode( 0, [ ])

CycNode( 1, [ ])

unfold P 0

unfold P 1

Unfold Implementation: Standard ML

generating fcn.:  \textbf{fun} \ P \ n = (n, [(n+1) \mod 2])
initial seed: \ 0

\begin{center}
\begin{tabular}{|c|c|}
\hline
0 & NONE \\
\hline
1 & NONE \\
\hline
\end{tabular}
\end{center}

\begin{center}
unfold P 0
\end{center}

\begin{center}
unfold P 1
\end{center}

CycNode( 0, [ ])

CycNode( 1, [ ])

Memo Table
Unfold Implementation: Standard ML

generating fcn.: \( \text{fun } P \ n = (n, [(n+1) \mod 2]) \)

initial seed : 0
Unfold Implementation: Standard ML

generating fcn.: \[
\text{fun } P \ n = (n, [(n+1) \ mod \ 2])
\]

initial seed: 0

<table>
<thead>
<tr>
<th>0</th>
<th>NONE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SOME(CycNode(1, []))</td>
</tr>
</tbody>
</table>

Memo Table
Unfold Implementation: Standard ML

generating fcn.: \( \textbf{fun} \ P \ n = (n, [(n+1) \mod 2]) \)

initial seed: 0

Memo Table

\[
\begin{array}{c|c}
0 & \text{Unfold } P \ 0 \\
1 & \text{SOME(CycNode(0, [ ]))} \\
\end{array}
\]

\[
\begin{array}{c|c}
0 & \text{SOME(CycNode(1, [ ]))} \\
\end{array}
\]
Unfold Implementation: Standard ML

generating fcn.: \( \text{fun} \ P \ n = (n, [(n+1) \mod 2]) \)
initial seed: 0

Memo Table

\[ \begin{array}{c|c|c}
0 & \rightarrow & \text{SOME(CycNode(0, [ ]))} \\
1 & \rightarrow & \text{SOME(CycNode(1, [ ]))} \\
\end{array} \]
generating fcn.: \[ \text{fun } \ P \ n = (n, [(n+1) \mod 2]) \]
initial seed: 0

Memo Table

<table>
<thead>
<tr>
<th>UID</th>
</tr>
</thead>
<tbody>
<tr>
<td>231</td>
</tr>
</tbody>
</table>

UID Counter

0

(231, SOME(CycNode(0, [])))

1

(230, SOME(CycNode(1, [])))
Unfold Implementation: Discussion

- Can use fewer reference cells in SML implementation.
- Cyclic hash-consing yields minimal graphs (Mauborgne, ESOP 2000; Considine & Wells, unpublished).

Haskell implementation:

- Uses laziness to tie cyclic knots.
- Uses a Cycle monad to thread UID counter and memoization tables through computation.
- Tricky to tie cyclic knots in presence of monad; use techniques of Erkok and Launchbury (ICFP ’00).

In practice, a memofix function is more flexible than unfold (see paper).
Road Map

- Viewing cyclic structures as infinite regular trees.
- Adapting the tree-generating `unfold` function to generate cyclic structures for infinite regular trees.
- Adapting the tree-accumulating `fold` function to return non-trivial results for strict combining functions and infinite regular trees.
- Cycamores: an abstraction for manipulating regular trees that we have implemented in ML and Haskell.
Tree Accumulation via Fold

The fold operator accumulates a result from a tree using a combining function.

\[
\text{fold} : \left( \left( L \times (C_{\text{res}}^\omega) \right) \xrightarrow{\text{cont}} C_{\text{res}} \right) \rightarrow \left( \text{Tree}(L) \xrightarrow{\text{cont}} C_{\text{res}} \right)
\]

- **Label**: \( L \)
- **Results for Children**: \( R_1, \ldots, R_k \)
- **Combining Function**: \( \phi \)
- **Result**: \( R \)
- **Tree**: \( \text{Tree}(L) \)
- **Tree Valuation**: \( \theta \)
- **Accumulating Function**: \( \phi \)-catamorphism
Folding over a Finite Tree

### Accumulating Function $\phi$

| $n_0$ | $n_1$ | ... | $n_k$ |

### Tree

```
    6
   /|
  2 5
 /|
4
```

$n_0 + n_1 + \ldots + n_k$
## Folding over a Finite Tree

### Accumulating Function $\phi$

\[
n_0 + n_1 + \ldots + n_k
\]

### Tree

- Node 6
- Node 2
- Node 5
- Node 4

The diagram illustrates the accumulation of values $n_0, n_1, \ldots, n_k$ through a tree structure, where each node represents a sum operation.
Folding over a Finite Tree

Accumulating Function $\phi$

$n_0 + n_1 + \ldots + n_k$

Tree

Folding over a Finite Tree

Accumulating Function $\phi$

$n_0 + n_1 + \ldots + n_k$

Tree

$n_0$

$n_1 \ldots n_k$
Folding over a Finite Tree

<table>
<thead>
<tr>
<th>Accumulating Function $\phi$</th>
<th>Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n_0 + n_1 + \ldots + n_k$</td>
<td><img src="" alt="Diagram" /></td>
</tr>
</tbody>
</table>

Folding over an Infinite Regular Tree

Expect \( \theta \) to be \( \phi \)-consistent: for each subtree \( t \) of a given tree,
\[
\theta(t) = \phi(\text{label}(t), \text{map}(\theta)(\text{children}(t)))
\]
Folding over an Infinite Regular Tree
Folding over an Infinite Regular Tree

Cycle Therapy, NEPLS, Oct. 5, 2001 – p.18/30
Folding over an Infinite Regular Tree

This fold may be desirable, but it is not the computed one.
Folding over an Infinite Regular Tree

This fold is not computed either.
Fold: Formalism

Let the result domain be $C_{res}$ (a pointed cpo).

$$\text{fold} : \underbrace{( (L \times (C_{res}^\omega)) \xrightarrow{\text{cont}} C_{res})}_{\text{accumulating function } \phi} \rightarrow \underbrace{\text{Tree}(L) \xrightarrow{\text{cont}} C_{res}}_{\text{tree valuation } \theta}$$

$(\phi\text{-catamorphism})$

$\text{fold(}\phi)\text{ is the least fixed point of:}$

$$\text{recalc}(\phi) = \lambda\theta . \lambda t . \phi(\text{label}(t), \text{map}(\theta)(\text{children}(t)))$$

For a strict $\phi$ and any infinite tree $t$, $(\text{fix(}\text{recalc}(\phi)))(t) = \bot$. 
Cycfold: Goals

Given a strict combining function $\phi$, want $\text{cycfold}(\phi)$ that:
- Coincides with $\text{fold}(\phi)$ on finite trees;
- Can return a non-trivial result for regular trees;
- Diverges on non-regular trees.
Cycfold: The Idea

Use a result domain $C_{\text{res}}$ that is a \textit{lifted} pointed cpo (i.e., doubly pointed) and require the combining function $\phi$ to be strict and monotone.

For a given tree $t$, calculate $\text{cycfold}(\phi)(t)$ as follows:

- If $t$ not regular, return $\bot_{\text{undef}}$.
- Otherwise:
  - Let tree valuation $\theta_0$ map all subtrees of $t$ to $\bot_{\text{user}}$.
  - Iteratively calculate $\theta_{i+1}$ from $\theta_i$ using $\phi$.
  - If $\theta_{k+1} = \theta_k$ then return $\theta_k(t)$ else return $\bot_{\text{undef}}$. 

Cycle Therapy, NEPLS, Oct. 5, 2001 – p.21/30
Cycfold Example 1: Node Labels
Cycfold Example 1: Node Labels
Cycfold Example 1: Node Labels

![Node Labels Diagram]

- Node labels: 0, 1, and U
- Edges: 230, 231
- Cycle structure

...
Node labels can encode other aspects of cyclic data.
Cycfold Example 2: DFA Strings

map ('a' :)

map ('b' :)

(0, False, ['a','b'])

map ('c' :)

map ('d' :)

{}
Cycfold Example 2: DFA Strings

(0, False, ['a','b'])

map ('a' :)

map ('b' :)

(1, True, ['d','c'])

map ('c' :)

map ('d' :)

map ('e' :)

Cycfold Example 2: DFA Strings

(0, False, [‘a’, ‘b’])

map (‘a’ :)

map (‘b’ :)

{b}

(1, True, [‘d’, ‘c’])

map (‘d’ :)

map (‘c’ :)

{“”, c}

{“”, c}

{“”}

{“”}

Cycfold Example 2: DFA Strings

\[\begin{align*}
(0, \text{False}, [\text{'a', 'b'}]) & \rightarrow \text{map ('a' :)} \\
(1, \text{True}, [\text{'d', 'c'}]) & \rightarrow \text{map ('d' :)} \\
\{\text{ab, b, bc}\} & \rightarrow \text{map ('b' :)} \\
\{\text{',', c, cc, db}\} & \rightarrow \text{map ('c' :)} \\
\end{align*}\]
Cycfold Example 2: DFA Strings

\[
\begin{align*}
(0, \text{False}, [\text{'}a\text{',} \text{'}b\text{']}) & \quad \text{U} \quad \{\text{aab, ab, abc, b, bc, bcc, bdb}\} \\
(1, \text{True}, [\text{'}d\text{',} \text{'}c\text{']}) & \quad \text{U} \\
\{\text{'}\text{'}\text{'}\text{'}\}, \text{c, cc, ccc, cdb, dab, db, dbc}\} & \quad \text{U} \\
\end{align*}
\]

\[
\begin{align*}
\text{map ('}a\text{':)} & \\
\text{map ('}b\text{':)} & \\
\text{map ('}c\text{':)} & \\
\text{map ('}d\text{':)} & \\
\end{align*}
\]
Cycfold: Related Work

- Iterative fixed points common in compiler data flow.
- Graph folds (Gibbons, unpublished):
  - $\text{ifold} = \text{foldtree} \circ \text{untie}$, analogous to $\text{fold}$.
  - $\text{efold}$ analogous to $\text{cycfold}$.
- Catamorphisms over datatypes with embedded functions (Fegaras & Sheard, POPL'96):
  - Express cycles via embedded functions. E.g.,
    ```
    \text{val alts} = \text{Rec}(\text{fn } x \Rightarrow \text{Cons}(0, (\text{Cons} 1 x)))
    ```
  - Can express catamorphisms over such cycles (e.g., $\text{map}$), but these can expose the structure of the representative.
Road Map

- Viewing cyclic structures as infinite regular trees.
- Adapting the tree-generating `unfold` function to generate cyclic structures for infinite regular trees.
- Adapting the tree-accumulating `fold` function to return non-trivial results for strict combining functions and infinite regular trees.
- Cycamores: an abstraction for manipulating regular trees that we have implemented in ML and Haskell.
Cycamore\( (L) \) is the type of potentially cyclic graphs, with a hidden UID for each node, parameterized over label type.

Examples:

- Key operations: make, view, unfold, fold, cycfold.
- Other operations: cycfix, memofix (see paper).
- Implementations in Standard ML and Haskell.
# Cycamore Signatures 1

<table>
<thead>
<tr>
<th>Standard ML</th>
<th>Haskell</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>val make</strong></td>
<td>make ::</td>
</tr>
<tr>
<td><code>(a * a Cycamore list)</code></td>
<td>(a, [Cycamore a])</td>
</tr>
<tr>
<td><code>-&gt; a Cycamore</code></td>
<td><code>-&gt; Cycle s (Cycamore a)</code></td>
</tr>
<tr>
<td><strong>val view</strong></td>
<td>view ::</td>
</tr>
<tr>
<td><code>a Cycamore</code></td>
<td><code>Cycamore a</code></td>
</tr>
<tr>
<td><code>-&gt; (a * a Cycamore list)</code></td>
<td><code>-&gt; (a, [Cycamore a])</code></td>
</tr>
<tr>
<td><strong>val unfold</strong></td>
<td>unfold ::</td>
</tr>
<tr>
<td><code>a MemKey</code></td>
<td>Ord a =&gt;</td>
</tr>
<tr>
<td><code>-&gt; (a -&gt; (b * a list))</code></td>
<td>(a -&gt; (b, [a]))</td>
</tr>
<tr>
<td><code>-&gt; a</code></td>
<td><code>-&gt; a</code></td>
</tr>
<tr>
<td><code>-&gt; b Cycamore</code></td>
<td><code>-&gt; Cycle s (Cycamore b)</code></td>
</tr>
</tbody>
</table>
## Cycamore Signatures 2

<table>
<thead>
<tr>
<th>Standard ML</th>
<th>Haskell</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>val fold</strong>:</td>
<td><strong>fold ::</strong></td>
</tr>
<tr>
<td>(`b -&gt; (’a list) -&gt; ’a)</td>
<td>(b -&gt; [a] -&gt; a)</td>
</tr>
<tr>
<td>-&gt; (`b Cycamore)</td>
<td>-&gt; (Cycamore b)</td>
</tr>
<tr>
<td>-&gt; ’a</td>
<td><strong>a</strong></td>
</tr>
<tr>
<td><strong>val cycfold</strong>:</td>
<td><strong>cycfold ::</strong></td>
</tr>
<tr>
<td>’a</td>
<td>(POrd a) =&gt;</td>
</tr>
<tr>
<td>-&gt; ((’a * ’a) -&gt; bool)</td>
<td>a</td>
</tr>
<tr>
<td>-&gt; (`b -&gt; (’a list) -&gt; ’a)</td>
<td>-&gt; (b -&gt; [a] -&gt; a)</td>
</tr>
<tr>
<td>-&gt; (`b Cycamore)</td>
<td>-&gt; (Cycamore b)</td>
</tr>
<tr>
<td>-&gt; ’a</td>
<td>-&gt; a</td>
</tr>
</tbody>
</table>

- **combining fcn.**
- **cycamore**
- **result**
- **partial order class**
- **user bottom**
- **geq**
- **combining fcn.**
- **cycamore**
- **result**
Future Work

Theory:
- Non-strict combining functions with cycfold.
- Can cycfold return a cynamore?
- Version of fold based on greatest fixed points.

Practice:
- Avoiding single-threaded UID generation.
- Memoization strategies.
- cycfold implementation heuristics.
- Cyclic hash-consing experimentation.
- Extending ML/Haskell with general cyclic data types.