Cycle Therapy

A Prescription for Fold and Unfold on Regular Trees

Franklyn Turbak  J. B. Wells
Wellesley College  Heriot-Watt University
Cyclic Structures Are Ubiquitous

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

![Diagram showing cyclic structures and a function definition for the factorial.](image)
Digression: Strictness

Let \( \bot \) (pronounced “bottom”) stand for a computation which diverges (e.g., loops infinitely) or signals an error.

A mathematical function is *strict* in a parameter if the function returns \( \bot \) whenever that parameter is \( \bot \).

Examples:

- The \( + \) operator is strict in both arguments.
- The function \( f(x, y) = x \) is strict in the \( x \) parameter but non-strict in the \( y \) parameter.
**Digression: Eagerness vs. Laziness**

- An *eager* language models all programming language functions as mathematical functions that are strict in all parameter positions. E.g., the previous \( f \) would be treated as if it were written:

  \[
  f(x, y) = (\text{if } y == \bot \text{ then } \bot \text{ else } x)
  \]

  Most programming languages are eager. E.g.: Java, C, C++, Pascal, Fortran, Scheme, ML, ...

- A *lazy* language models programming language functions with their “natural” strictness. In particular, all data constructors are non-strict in all arguments. E.g.:

  \[
  f(3, (\text{loop})) = 3 \\
  \text{length}(\text{loop}):(\text{loop}):[])) = 2
  \]

  Haskell is an example of a lazy language.
Cycles Are Tricky To Manipulate

Consider Haskell’s \( alts = 0:1:alts \)

- Naïve generation ⇒ unbounded structures:
  - \( \text{let } \text{inf } x \ y = x:(\text{inf } y \ x) \ \text{in } \text{inf } 2 \ 3 \)

- map (\( \lambda \ x \to x + 2 \)) \( alts \)

- Naïve accumulation ⇒ divergence:
  - \( \text{foldr } (+) \ 0 \ alts \)
  - \( \text{foldr } \text{Set.insert } \text{Set.empty} \ alts \)

- 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.

### Infinite Regular Trees

- Level 1: 2 → 3 → 2 → 3 → 2
- Level 2: 0
- Level 3: 0, 1, 0, 1, 0, 1, 0, 1

### Infinite Non-regular Trees

- Level 1: 1 → 2 → 3 → 4 → 5
- Level 2: 0, 1
- Level 3: 0, 1, 1
- Level 4: 0, 1, 1, 1, 1
- Level 5: 0, 1, 1, 1, 1
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 Image" /></td>
<td><img src="image2" alt="Some Finite Cyclic Representatives Image" /></td>
</tr>
</tbody>
</table>

Cycle Therapy, Wellesley CS Dept., Nov 7, 2001 – p.8/33
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.
The unfold operator generates a tree from a generating function and a seed.

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

- Generating function \( \psi \)
- Seed \( S \)
- Trees over \( L \)
- \( \psi \)-anamorphism

Diagram: 
- Seed \( S \) 
- Generating Function 
- Label \( L \) 
- Seeds for Children \( S_1 \ldots S_k \)
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>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></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><img src="image" alt="Node n" /></td>
<td><img src="image" alt="Tree Diagram" /></td>
</tr>
</tbody>
</table>

Diagram shows a regular tree generated by the function $\psi$. The tree structure is depicted with nodes labeled 0 and 1, illustrating the recursive nature of the tree generation process.
Unfold Example 1: 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></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

Cycle Therapy, Wellesley CS Dept., Nov 7, 2001 – p.12/33
**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>$n$</td>
<td>$0$</td>
</tr>
<tr>
<td></td>
<td>$1$</td>
</tr>
<tr>
<td>$0$</td>
<td>$0$</td>
</tr>
<tr>
<td></td>
<td>$1$</td>
</tr>
<tr>
<td>$0$</td>
<td>$0$</td>
</tr>
<tr>
<td></td>
<td>$1$</td>
</tr>
<tr>
<td>$0$</td>
<td>$0$</td>
</tr>
<tr>
<td></td>
<td>$1$</td>
</tr>
<tr>
<td>$0$</td>
<td>$1$</td>
</tr>
<tr>
<td></td>
<td>$1$</td>
</tr>
<tr>
<td>$1$</td>
<td>$0$</td>
</tr>
<tr>
<td></td>
<td>$1$</td>
</tr>
<tr>
<td>$1$</td>
<td>$0$</td>
</tr>
<tr>
<td></td>
<td>$1$</td>
</tr>
</tbody>
</table>

Cycle Therapy, Wellesley CS Dept., Nov 7, 2001 – p.12/33
### 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.png" alt="Tree Diagram" /></td>
</tr>
<tr>
<td>$\psi$</td>
<td></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

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

Tree

- $0$
- $1$
- $2$
- $3$
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>$2n, 2n+1$</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>7</td>
</tr>
</tbody>
</table>

$\psi : \mathbb{N} \rightarrow \mathbb{N}_0$
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 )</td>
<td>( 2 )</td>
</tr>
<tr>
<td>( 1 )</td>
<td>( 3 )</td>
</tr>
<tr>
<td>( 2 )</td>
<td>( 4 )</td>
</tr>
<tr>
<td>( 3 )</td>
<td>( 5 )</td>
</tr>
<tr>
<td>( 4 )</td>
<td>( 6 )</td>
</tr>
<tr>
<td>( 5 )</td>
<td>( 7 )</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).
Unfold Implementation: Standard ML

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

initial seed : 0

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

initial seed : 0

\textbf{unfold} P 0

Memo Table

\textbf{NONE}
generating fcn.: \texttt{fun} \ P \ n = (n, [(n+1) \ mod \ 2])

initial seed : 0
Unfold Implementation: Standard ML

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

\begin{figure}
\begin{center}
\begin{tikzpicture}
\node[draw] (n0) at (0,0) {0};
\node[draw] (n1) at (0,-2) {1};
\node[draw] (root) at (1,0) {NONE};
\node[draw] (cyc) at (1,-2) {NONE};
\node (node0) at (1,-1) {CycNode( 0, [ ])};
\node (node1) at (-1,-1) {CycNode( 1, [ ])};
\draw[->] (n0) -- (root);
\draw[->] (n1) -- (root);
\draw[->] (root) -- (cyc);
\draw[->] (cyc) -- (node0);
\draw[->] (node0) -- (node1);
\end{tikzpicture}
\end{center}
\end{figure}

Memo Table
Unfold Implementation: Standard ML

generating fcn.:  \textbf{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

Memo Table

0

1

unfold P 0

NONE

CycNode(0, [])

unfold P 1

NONE

CycNode(1, [])
Unfold Implementation: Standard ML

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

Memo Table

\[ \begin{array}{c|c}
0 & \text{NONE} \\
1 & \text{SOME}(	ext{CycNode}(1, [ ])) \\
\end{array} \]
Unfold Implementation: Standard ML

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

\[
\text{unfold } P \ 0 \\
0 \quad \rightarrow \quad \text{NONE} \quad \rightarrow \quad \text{SOME(CycNode( 0, [ ]))}
\]

\[
1 \quad \rightarrow \quad \text{SOME(CycNode( 1, [ ]))}
\]
Unfold Implementation: Standard ML

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

Memo Table

unfold P 0

SOME(CycNode( 0, [ ]))

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

initial seed: \( 0 \)

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

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

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

Memo Table

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

(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.

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

- **Label**: \( L \)
- **Results for Children**: \( R_1, \ldots, R_k \)
- **Combining Function**: \( \phi \)
- **Result**: \( R \)

Cycle Therapy, Wellesley CS Dept., Nov 7, 2001 – p.18/33
Folding over a Finite Tree

Accumulating Function $\phi$

$\sum_{i=0}^{k} n_i$
Folding over a Finite Tree

Accumulating Function $\phi$

$n_0 + n_1 + \ldots + n_k$

$n_0$

$n_1 \ldots n_k$

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="diagram.png" alt="Tree Diagram" /></td>
</tr>
</tbody>
</table>

Cycle Therapy, Wellesley CS Dept., Nov 7, 2001 – p.19/33
## Folding over a Finite Tree

### Accumulating Function $\phi$

<table>
<thead>
<tr>
<th>$n_0 + n_1 + \ldots + n_k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n_0$</td>
</tr>
<tr>
<td>$n_1 \ldots n_k$</td>
</tr>
</tbody>
</table>

### Tree

```
  6 +
 /|
/ |
2 +/
/|
5 +/
/|
4 +/
/|
9 +/
```

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

Accumulating Function $\phi$

$\begin{align*}
n_0 + n_1 + \ldots + n_k
\end{align*}$

Tree

$\begin{align*}
6 + 17 &= 22 \\
2 + 5 &= 7 \\
4 + 9 &= 13
\end{align*}$
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
Folding over an Infinite Regular Tree

This fold may be desirable, but it is not the computed one.
This fold is not computed either.
A value $x$ is a fixed point of a function $f$ if $f(x) = x$.

What are the fixed points of the following:

$$f_i :: \text{Int} \rightarrow \text{Int}$$

$$f_1(x) = \frac{x}{2} + 3$$

$$f_2(x) = x^2$$

$$f_3(x) = x$$

$$f_4(x) = x - 1$$

Can also have fixed points over functions manipulating data structures and other functions:

$$g :: [\text{Int}] \rightarrow [\text{Int}]$$

$$g(x) = 0 : 1 : x$$

$$h :: (\text{Int} \rightarrow \text{Int}) \rightarrow (\text{Int} \rightarrow \text{Int})$$

$$h(k) = \backslash n \rightarrow \text{if } n == 0 \text{ then } 1 \text{ else } n \times (k(n-1))$$
Digression: Least Fixed Points

Under certain conditions, functions over data structures and functions have a so-called least fixed point. In particular, the function must be a continuous function between two pointed complete partial orders.

Intuitively, a pointed complete partial order is a lattice rooted at \( \perp \) where elements are arranged by information content and every chain has a limit.

Intuitively, the least fixed point of a function \( f \) is found by starting at \( \perp \) and applying \( f \) until a limit is reached.

For strict \( f \), the least fixed point will always be \( \perp \).
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 $C_{res}$ that is a *lifted* pointed cpo (i.e., doubly pointed) and require the combining function $\phi$ to be strict and monotone.

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

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

}\newbottom
Cycfold Example 1: Node Labels

![Diagram of node labels with nodes 0, 1, and U connected by arrows and sets {0, 1, 2} and {0, 1, 2} depicted.]
Cycfold Example 1: Node Labels

\[
\begin{array}{c}
\text{0} \\
\text{1} \\
\{1\} \\
\{0\}
\end{array}
\]

\[
\begin{array}{c}
\text{U} \\
\text{U}
\end{array}
\]

\[
\ldots \\
\{0, 1, 2\} \\
\{0, 1\} \\
\{0\} \\
\{\}
\]\n
\[
\begin{array}{c}
\text{undef}
\end{array}
\]
Cycfold Example 1: Node Labels
Cycfold Example 1: Node Labels
Cycfold Example 2: DFA Strings

Node labels can encode other aspects of cyclic data.

![Diagram showing a cycle with node labels and transitions labeled with specific tuples.](image-url)
Cycfold Example 2: DFA Strings

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

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

map ('a' :)

map ('b' :)

map ('c' :)

map ('d' :)

{}
Cycfold Example 2: DFA Strings

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

(map 'a' :)

(map 'b' :)

(map 'c' :)

(map 'd' :)

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

(map 'c' :)

(map 'd' :)

{}
Cycfold Example 2: DFA Strings

U

map (‘a’ :)

map (‘b’ :)

(0, False, [‘a’, ‘b’])

map (‘c’ :)

{b}

{“”, c}

(1, True, [‘d’, ‘c’])

{“”}

map (‘d’ :)

map (‘c’ :)

U

Cycle Therapy, Wellesley CS Dept., Nov 7, 2001 – p.27/33
Cycfold Example 2: DFA Strings

map ('a' :)
map ('b' :)
map ('c' :)
map ('d' :)

{ab, b, bc}
{"", c, cc, db}
{""", d, dd, dc}
{"""

(0, False, ['a','b'])
(1, True, ['d','c'])

Cycle Therapy, Wellesley CS Dept., Nov 7, 2001 – p.27/33
Cycfold Example 2: DFA Strings

- **States:**
  - State 0: False, \(['a', 'b']\)
  - State 1: True, \(['d', 'c']\)
- **Initial State:** State 0
- **Accepting States:** State 1
- **Transitions:**
  - **map ('a' :)
    - From State 0 to State 0
    - From State 0 to State 1
  - **map ('b' :)
    - From State 0 to State 0
    - From State 1 to State 0
  - **map ('c' :)
    - From State 1 to State 1
    - From State 1 to State 0
  - **map ('d' :)
    - From State 0 to State 0
    - From State 0 to State 1
- **Strings Accepted:**
  - State 0: \{'aab, ab, abc, b, bc, bcc, bdb\}
  - State 1: \{'', c, cc, ccc, cdb, dab, db, dbc\}
  - State 0: \{'', c, cc, ccc, cdb, dab, db, dbc\}

Cycle Therapy, Wellesley CS Dept., Nov 7, 2001 – p.27/33
Cycfold: Related Work

- Iterative fixed points common in compiler data flow.
- Graph folds (Gibbons, unpublished):
  - `ifold = foldtree ◦ untie`, analogous to `fold`.
  - `efold` analogous to `cycfold`.
- Catamorphisms over datatypes with embedded functions (Fegaras & Sheard, POPL’96):
  - Express cycles via embedded functions. E.g.,
    ```
    val alts = Rec(fn x => Cons(0, (Cons 1 x)))
    ```
  - Can express catamorphisms over such cycles (e.g., `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.
Cycamores

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><strong>make ::</strong></td>
</tr>
<tr>
<td>('a * 'a Cycamore list)</td>
<td>(a, [Cycamore a])</td>
</tr>
<tr>
<td>-&gt; 'a Cycamore</td>
<td>-&gt; Cycle s (Cycamore a)</td>
</tr>
<tr>
<td><strong>val view:</strong></td>
<td><strong>view ::</strong></td>
</tr>
<tr>
<td>'a Cycamore</td>
<td>Cycamore a</td>
</tr>
<tr>
<td>-&gt; ('a * 'a Cycamore list)</td>
<td>-&gt; (a, [Cycamore a])</td>
</tr>
<tr>
<td><strong>val unfold:</strong></td>
<td><strong>unfold ::</strong></td>
</tr>
<tr>
<td>'a MemKey</td>
<td>Ord a =&gt;</td>
</tr>
<tr>
<td>-&gt; ('a -&gt; ('b * 'a list))</td>
<td>(a -&gt; (b, [a]))</td>
</tr>
<tr>
<td>-&gt; 'a</td>
<td>-&gt; a</td>
</tr>
<tr>
<td>-&gt; 'b Cycamore</td>
<td>-&gt; Cycle s (Cycamore b)</td>
</tr>
<tr>
<td>Standard ML</td>
<td>Haskell</td>
</tr>
<tr>
<td>-------------</td>
<td>---------</td>
</tr>
<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>a</td>
</tr>
<tr>
<td><strong>val cycfold :</strong></td>
<td></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 \texttt{cycfold}.
- Can \texttt{cycfold} return a cycamore?
- Version of \texttt{fold} based on greatest fixed points.

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