# A Functional Graph Library

Based on
*Inductive Graphs and Functional Graph Algorithms*
by Martin Erwig

Presentation by
Christian Doczkal
March 22nd, 2006

# Structure

- Motivation
- Inductive graph definition
- Implementation
  - Binary search trees
  - Version-tree implementation
- Algorithms I (DFS)
- Conclusions
- Algorithms II

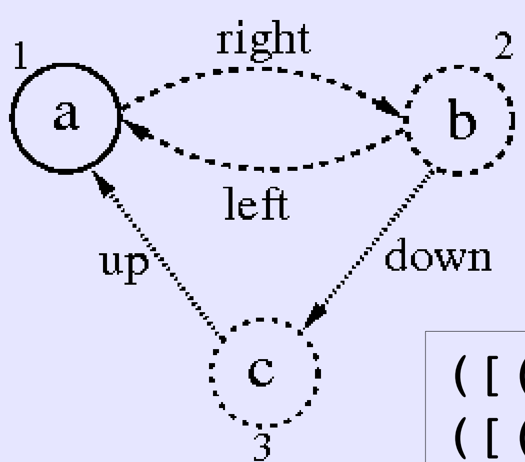
# Motivation

- Goals
  - Find inductive model for graphs
  - Provide efficient graph implementations that meet imperative time bounds
  - Make functional languages suitable for teaching graph algorithms
  - Increase overall acceptance of functional languages
- Benefits
  - Inductive programming style gives clarity and elegance
  - Inductive proofs over graph algorithms possible

# Inductive graph definition

```
type Node           = Int
type Adj b          = [(b,Node)]
type Context a b    = (Adj b, Node, a, Adj b)
```

```
data Graph a b = Empty | Context a b & Graph a b
```




```
([(“ down”,2)],3,'c',[(“  up”,1)]) &
([(“right”,1)],2,'b',[(“left”),1]) &
           ([],1,'a',[])             & Empty
```

# Inductive graph definition

- Fact 1 (Completeness):
Each labeled multi-graph can be represented by a Graph term

- Fact 2 (Choice of Representation)
For each graph *g* and each node *v* contained in g there exist *p,l,s* and *g'* such that
*(p,v,l,s) & g'* denotes *g*.

# Implementation

- Requirements
  - Construction
    - Empty Graph (*Empty*)
    - Add context (*&*)
  - Decomposition
    - Test for Empty Graph (*Empty-match*)
    - Extract arbitrary context (*&-match*)
    - Extract specific context (*&$^v$-match*)
- Definitions for time bounds G = (V,E):
  - $$n := |V| \quad m := |E| \quad c_v := |suc\, v| + |pred\, v|$$
    $$c := max\{v \in V / c_v\}$$

# Binary search trees

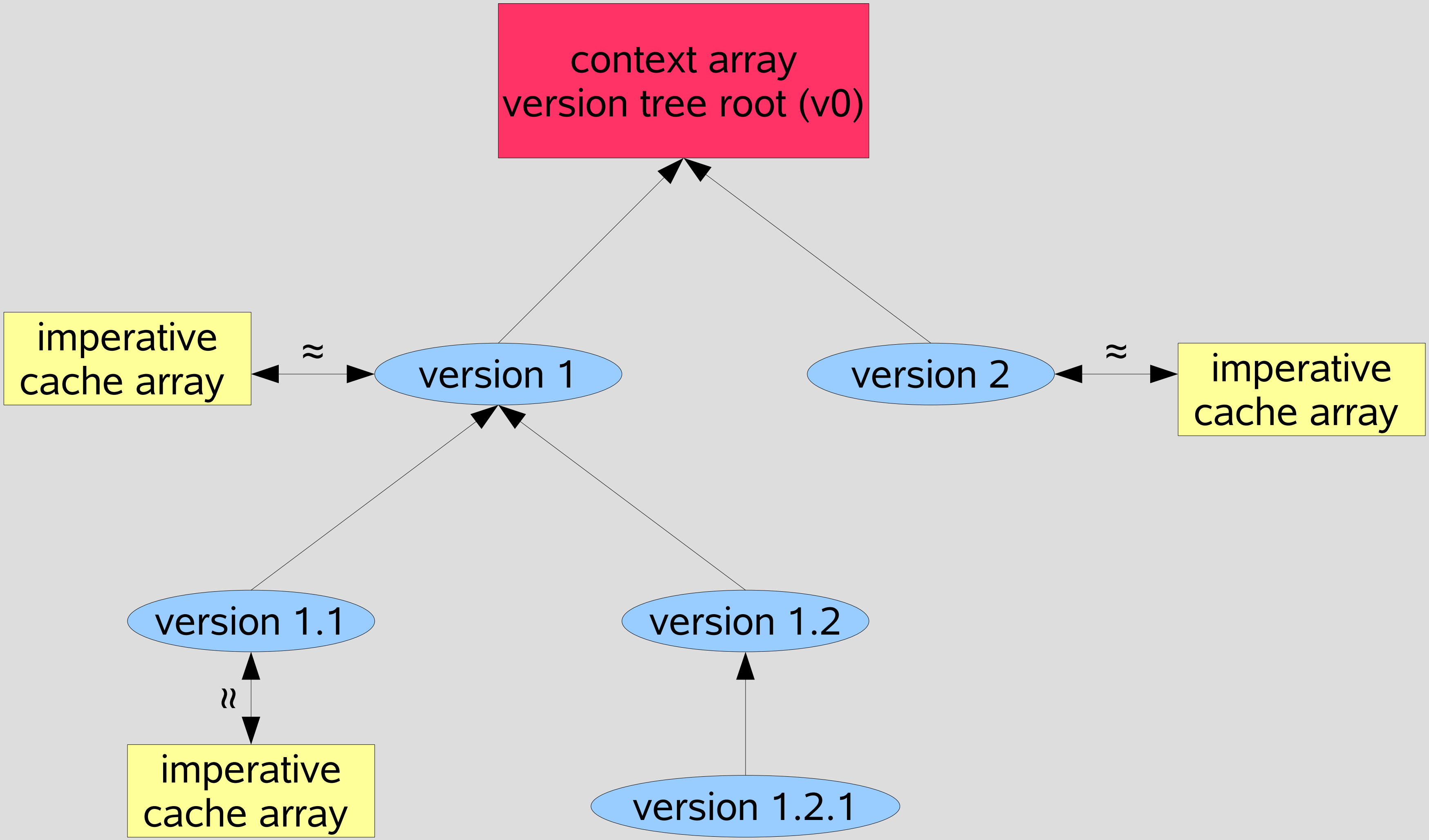
- Graph is represented as pair (t,m)
  - t = binary search tree of (node,(predecessor,label,successor))
  - m = highest node occurring in t
  - Predecessors/successors stored as binary search trees
- Time bounds
  - Node insertion:
  - Node deletion:
  - &/&$^v$ -match:

$$\left.\begin{array}{l} \\ \\ \\ \end{array}\right\} O(c_v \log c \log n) \subset O(n \log^2 n)$$

# Array version tree

- Implementation for functional arrays
- Implementation
  - Inward directed tree of (index, value) pairs
  - Original Array is the root of the tree
  - New versions inserted as children of the version they are derived from ( $O(1)$ )
  - Every version is a pointer to some node in the tree
  - Lookup follows tree structure terminating at root
  - ( $O(u)$ where u is the number of updates to the array)

# Version-tree representation



# Version-tree optimizations

- Avoiding Node Deletion
  - positive integer stamps for nodes and edges
  - node deletion ≈ negate integer for that node
  - adjacency ignores non matching stamps
  - insertion ≈ negate again and increment stamp
- &-match, Empty-match and insertion
  - $k := |V|$ so Empty-match ≈ $k = 0$
  - *elem* array stores partition of deleted and inserted nodes
  - *index* array stores position of nodes in elem array
  - &-match ≈ $\&^{elem[1]}$-match

# ADT – version-tree time bounds

- Test for Empty Graph (*Empty-match*)
- Extract arbitrary context (*&-match*)
- Extract specific context (*&$^v$-match)*

} $O(1)$

- *Add context (&)* $O(c_v \log c)$

- Multi threaded usage adds a factor *u* corresponding to number of previous updates

# Algorithms I (DFS)

- Depth first search

```
dfs :: [Node] -> Graph a b -> [Node]
dfs []       g          = []
dfs vs       Empty      = []
dfs (v:vs)  (c &v g)    = v : dfs (suc c ++ vs) g
dfs (v:vs)  g           = dfs vs g
```

- Breadth first search:

```
bfs (v:vs)  (c &v g) = v : dfs (vs ++ suc c) g
```

(or queue implementation for efficiency)

# Conclusions

- Goals met?
  - Code shows both clarity and elegance
  - Same time complexity as imperative implementations
- Problems
  - Double representation of edges and cache arrays cause a lot of memory overhead.
  - time complexity met only on single threaded graph usage

# Algorithms II

DF Spanning Forest:

```
data Tree a = Br a [Tree a]
postorder (Br v ts) = concatMap postorder ts ++ v

df :: [Node] -> Graph a b -> ([Tree Node], Graph a b)
df []        g         = ([],g)
df (v:vs)   (c &v g)  = (Br v f:f',g2)
                         where (f,g1)  = df (suc c) g
                               (f',g2) = df   vs     g1
df (v:vs)    g         = df vs g

dff :: [Node] -> Graph a b -> [Tree Node]
dff vs g = fst (df vs g)
```

# Algorithms II

Strongly connected groups:

```
topsort :: Graph a b -> [Node]
topsort g = reverse.concatMap postorder.(dff (nodes g) g)

scc :: Graph a b -> [Tree Node]
scc g = dff (topsort g) (grev g)
```

# Algorithms II (Dijkstra)

```
type Lnode a = (Node, a)
type Lpath a = [Lnode a]
type LRTree a = [Lpath a]
```

```
instance Eq a => Eq (Lpath a) where
((_,x):_) == ((_,y):_) = x == y

instance Ord a => Ord (Lpath a) where
((_,x):_) < ((_,y):_) = x < y
```

```
getPath Node -> LRTree a -> Path
getPath = reverse . map fst . first (\((w,_):_) -> w == v)

sssp :: Real b => Node -> Node -> Graph a b -> Path
sssp s t = getPath t . dijkstra (unitHeap [(s,0)])
```

# Algorithms II (Dijkstra)

Dijkstra SSSP:

```
expand ::   Real b =>
            b -> LPath b -> Context a b -> [Heap(LPath b)]
expand d p (_,_,_,s) = map(\(l,v) -> unitHeap((v,l+d):p)) s

dijkstra :: Real b =>
            Heap(LPath b) -> Graph a b -> LRTree b

dijkstra h g
   | isEmptyHeap h || isEmpty g = []

dijkstra (p@((v,d):_) << h)  (c &v g) =
         p:dijkstra (mergeAll (h:expand d p c)) g

dijkstra (_ << h)            g        = dijkstra h g
```