

Implementing Xappings

(also Sets, Maps, Tuples, Tables, &c)

Preston Briggs
Google

Xappings?

Long ago, Danny Hillis and Guy Steele described a Lisp for the Connection Machine.

Mostly, it was Common Lisp extended with a new data structure, the *xapping*.

It combined a compact notation for expressing parallelism along with ways to express locality.

I'm curious to learn about it, both as a way of programming and as an implementation challenge.

Xappings

A xapping is an unordered set of ordered pairs.

Each ordered pair has an index and a value.

Indexes and values may be any Lisp object, including xappings.

The pairs in a given xapping must have distinct indexes.

Here's some notation

```
{yertle→turtle horton→elephant}
```

Xets

Sometimes, the index and value of a pair are the same. Such a pair may be abbreviated as just the value. So...

```
{ball→red blue grass→green black}
```

is the same as

```
{ball→red blue→blue grass→green black→black}
```

If all the elements in a xapping may be abbreviated in this fashion, we call it a *xet*.

Xectors

If the domain of a mapping consists of consecutive integers starting with zero, then the mapping is known as a *xector*.

A xector may be abbreviated by writing its values in order surrounded by brackets. So...

$$\{1 \rightarrow \text{on} \quad 2 \rightarrow \text{pop} \quad 0 \rightarrow \text{hop}\}$$

may be written as

$$[\text{hop on pop}]$$

Infinite mappings

There are three sorts of *infinite mapping*.

A *constant mapping* has the same value for every index.

$$\{\rightarrow v\}$$

The *universal mapping* is the set of all Lisp objects.

$$\{\rightarrow\}$$

A *lazy mapping* uses a unary function to compute a value given the index.

$$\{. \text{sqrt}\}$$

Exceptions

Infinite mappings may have a finite number of explicit exceptions.

$$\{1 \rightarrow 5 \quad 10 \rightarrow 3 \quad 100 \rightarrow 3.14 \quad \rightarrow 0\}$$

We may also specify that certain indexes are undefined.
For example

$$\{1 \rightarrow 20 \quad 2 \rightarrow \quad \rightarrow 3\}$$

defines an infinite mapping where most values are mapped to 3, 1 is mapped to 20, and 2 is not mapped at all.

What's it all good for?

I like to write compilers, but most people at Google have other ideas. More generally, they're good for expressing all sorts of *symbolic* data-parallel computation.

We can add pairs to a xapping, or given the index, get the associated value, all rather like a hash table.

*But those are small time;
operations over complete xappings are big time.*

For example, we can get the domain or range of a xapping, or find the union or intersection of several xappings.

We can also define our own operations over complete xappings.

Applying a xapping

Lisp's function call mechanism is extended to allow xappings of functions to be called as functions, provided that the arguments are also xappings.

$$(\{\rightarrow+\} [1\ 2\ 3] [4\ 5\ 6\ 7]) \Rightarrow [5\ 7\ 9]$$

The domain of the result is the intersection of the domains of all the xappings.

This is how we express parallelism.

More closely

$$(\{\rightarrow+\} [1\ 2\ 3] [4\ 5\ 6\ 7]) \Rightarrow [5\ 7\ 9]$$

might be viewed as

$$([+ + +] [1\ 2\ 3] [4\ 5\ 6]) \Rightarrow [5\ 7\ 9]$$

or even

$$\begin{array}{l} [(+ 1 4) \\ (+ 2 5) \\ (+ 3 6)] \end{array} = \begin{array}{l} [5 \\ 7 \\ 9] \end{array}$$

A bit more syntax

Rather than writing $\{\rightarrow v\}$, we can write αv . So...

$$(\alpha+ [1\ 2\ 3] [4\ 5\ 6\ 7]) \Rightarrow [5\ 7\ 9]$$

We might think of α as "apply to all" since it's forcing the application of $+$ to all the elements of the arguments.

But consider

$$(\alpha+ \alpha1\ \alpha2) \Rightarrow \{\rightarrow 3\}$$

and

$$\alpha(+\ 1\ 2) \Rightarrow \{\rightarrow 3\}$$

So α distributes over function calls.

Still more

Suppose we have something a bit more interesting, like

$$(\alpha + (\alpha * c \alpha^{9/5}) \alpha^{32})$$

converting a mapping of measurements from Centigrade to Fahrenheit.

We've accumulated a lot of alphas, which seems awkward. We could "factor them out" if they were applied to every subform, but the mapping c doesn't have one.

They introduce \bullet as a kind of inverse to α , where $\alpha \bullet x = x$, so

$$\alpha(+ (* \bullet c 9/5) 32)$$

An analogy

Lispers will note the similarity with backquote and comma. That is,

$$\alpha(+ 1 \bullet x)$$

is syntactically analogous to

$$\backslash(+ 1 ,x)$$

(non-Lispers will want to study up!)

Reductions

The α syntax is used, in effect, to replicate or broadcast data. Another syntax, using the β character, is used to express the gathering up of data to produce a single result; that is, *reduction*. For example,

$$(\beta+ [1 2 3 4]) \Rightarrow 10$$

and

$$(\beta* [1 2 3 4]) \Rightarrow 24$$

Another opportunity for parallelism.

Examples

```
(define histogram  
  (λ (x)  
    (β+ x α1)))
```

```
(histogram '[a b a f e f f]) ⇒ {a→2 b→1 e→1 f→3}
```

```
(define histogram
  ( $\lambda$  (x)
    ( $\beta+$  x  $\alpha$ 1)))
```

```
( $\beta+$  '[a b a f e f f]  $\alpha$ 1)
```

```
( $\beta+$  '[a b a f e f f]
      [1 1 1 1 1 1 1])
```

```
{a $\rightarrow$ (1 + 1)
 b $\rightarrow$ 1
 e $\rightarrow$ 1
 f $\rightarrow$ (1 + 1 + 1)}
```

If the dense matrix

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

is represented as a vector of vectors, we can perform a reduction over rows or columns, like this:

$$(\alpha\beta+ \begin{bmatrix} [1 & 2 & 3] \\ [4 & 5 & 6] \\ [7 & 8 & 9] \end{bmatrix}) \Rightarrow [6 \ 15 \ 24]$$

$$(\beta\alpha+ \begin{bmatrix} [1 & 2 & 3] \\ [4 & 5 & 6] \\ [7 & 8 & 9] \end{bmatrix}) \Rightarrow [12 \ 15 \ 18]$$

$(\beta\alpha+ \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix})$

$(\beta\{-\rightarrow+\} \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix})$

$(\beta[+ + +] \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix})$

$([+ + +] \begin{bmatrix} 1 & 2 & 3 \\ ([+ + +] \begin{bmatrix} 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}) \end{bmatrix})$

The same code works for sparse matrices, like so

$$\begin{aligned} &(\alpha\beta+ \{0\rightarrow\{4\rightarrow93 \rightarrow0\} \\ &\quad 1\rightarrow\{1\rightarrow67 \ 3\rightarrow89 \rightarrow0\} \\ &\quad 3\rightarrow\{1\rightarrow23 \ 4\rightarrow35 \rightarrow0\} \\ &\quad 4\rightarrow\{0\rightarrow14 \ 4\rightarrow56 \rightarrow0\} \\ &\quad \rightarrow\{\rightarrow0\}\}) \\ \Rightarrow &\{0\rightarrow93 \ 1\rightarrow156 \ 3\rightarrow58 \ 4\rightarrow60 \rightarrow0\} \end{aligned}$$

It also extends nicely to matrices with more dimensions, e.g.,

$$\alpha\alpha\beta+ \quad \alpha\beta\alpha+ \quad \beta\alpha\alpha+$$

Here's how to find the next generation in Conway's Life

```
(define step
  (λ (g)
    (let* ((sum (αα+ (fill (offset g -1 -1))
                          (fill (offset g -1 0))
                          (fill (offset g -1 1))
                          (fill (offset g 0 -1))
                          (fill (offset g 0 1))
                          (fill (offset g 1 -1))
                          (fill (offset g 1 0))
                          (fill (offset g 1 1))))))
      (old (in (αin sum g) g)))
    (merge αα(if (= 3 ••sum) 1)
           αα(if (= 2 ••old) 1))))))
```

What's cool here?

Compact notation

Data parallelism for symbolic programs (versus just numeric)

No threads, no synchronization

Furthermore,

Parallelism associated with α has locality

Parallelism associated with β doesn't

Implementation

Long ago, I read about SETL and pondered how the sets, maps, multimaps, etc. might be implemented.

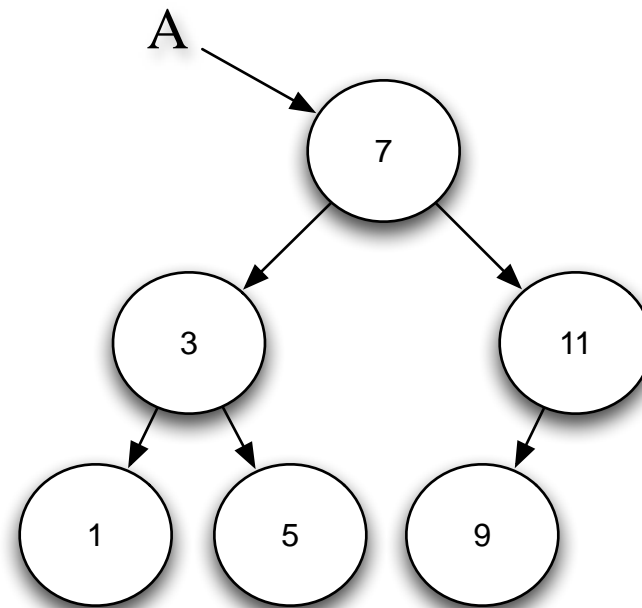
Then Myers came to Rice and talked about his '84 POPL paper

Efficient Applicative Data Types

and showed us a data structure he called AVL DAGs.

AVL DAGs

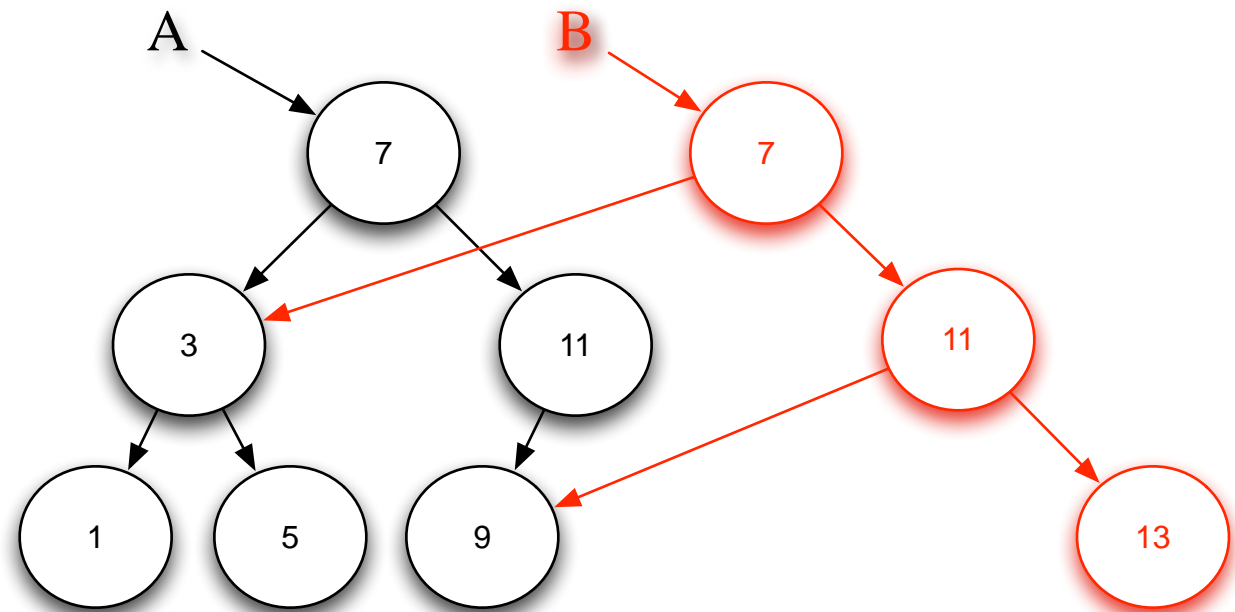
Suppose we've got a set $A = \{1, 3, 5, 7, 9, 11\}$ represented by a binary tree, like this



and we want to compute $B \leftarrow A \cup \{13\}$

AVL DAGs

Suppose we've got a set $A = \{1, 3, 5, 7, 9, 11\}$ represented by a binary tree, like this

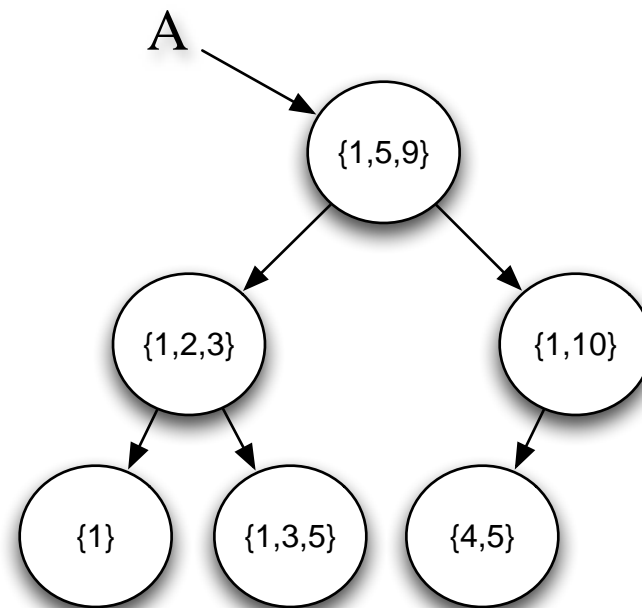


and we want to compute $B \leftarrow A \cup \{13\}$

Ouch!

I tried to implement a some sets, maps, etc. using C++, and tripped badly over the need for nested sets, maps, etc.

Suppose A is a set of sets



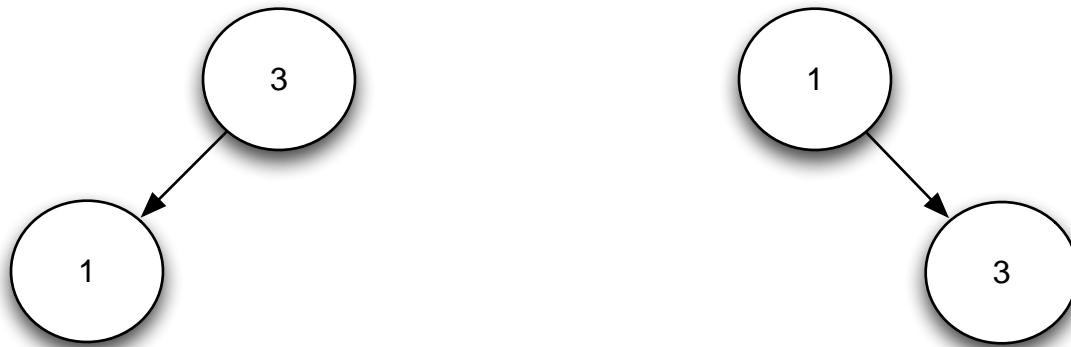
and we want to compute $B \leftarrow A \cup \{2, 3, 5, 7, 11\}$

Hash consing

By using a form of hash consing whenever I allocate a tree node, I'd be able to compare two trees for equality in constant time.

But that doesn't work here...

Imagine we want to represent the set $\{1, 3\}$ using a balanced binary tree. There are two balanced representations:

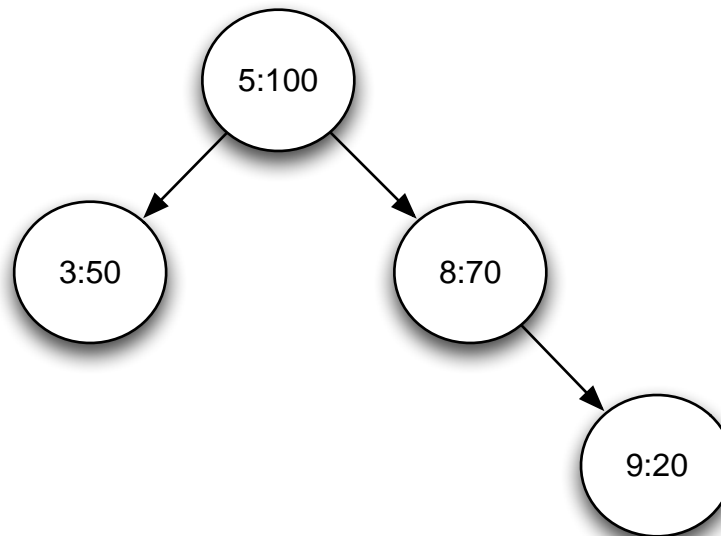


I need some sort of *unique representation*.

Treaps

In their '89 FOCS paper, Aragon and Seidel described randomized search trees, or *treaps*.

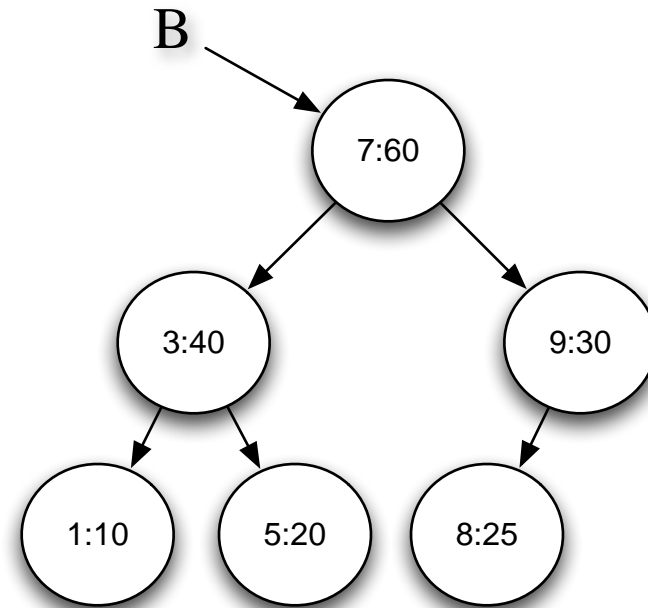
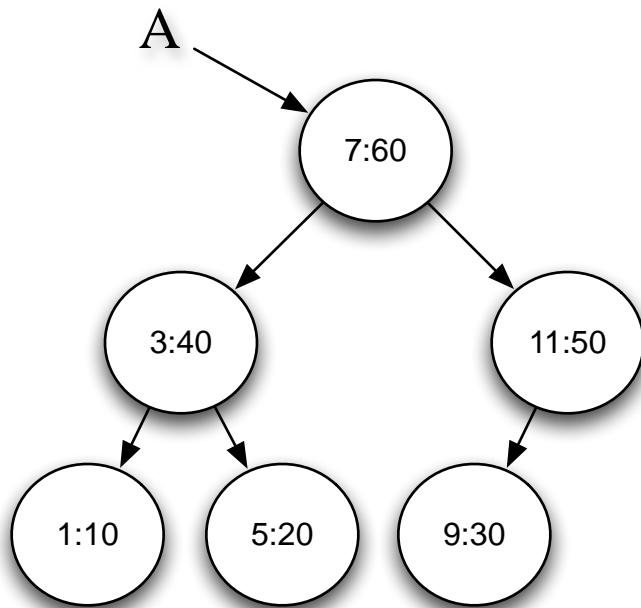
Treaps are probabilistically balanced binary trees, organized by key value and priority.



In my implementation, the priority is just a hash of the key. This yields a unique representation.

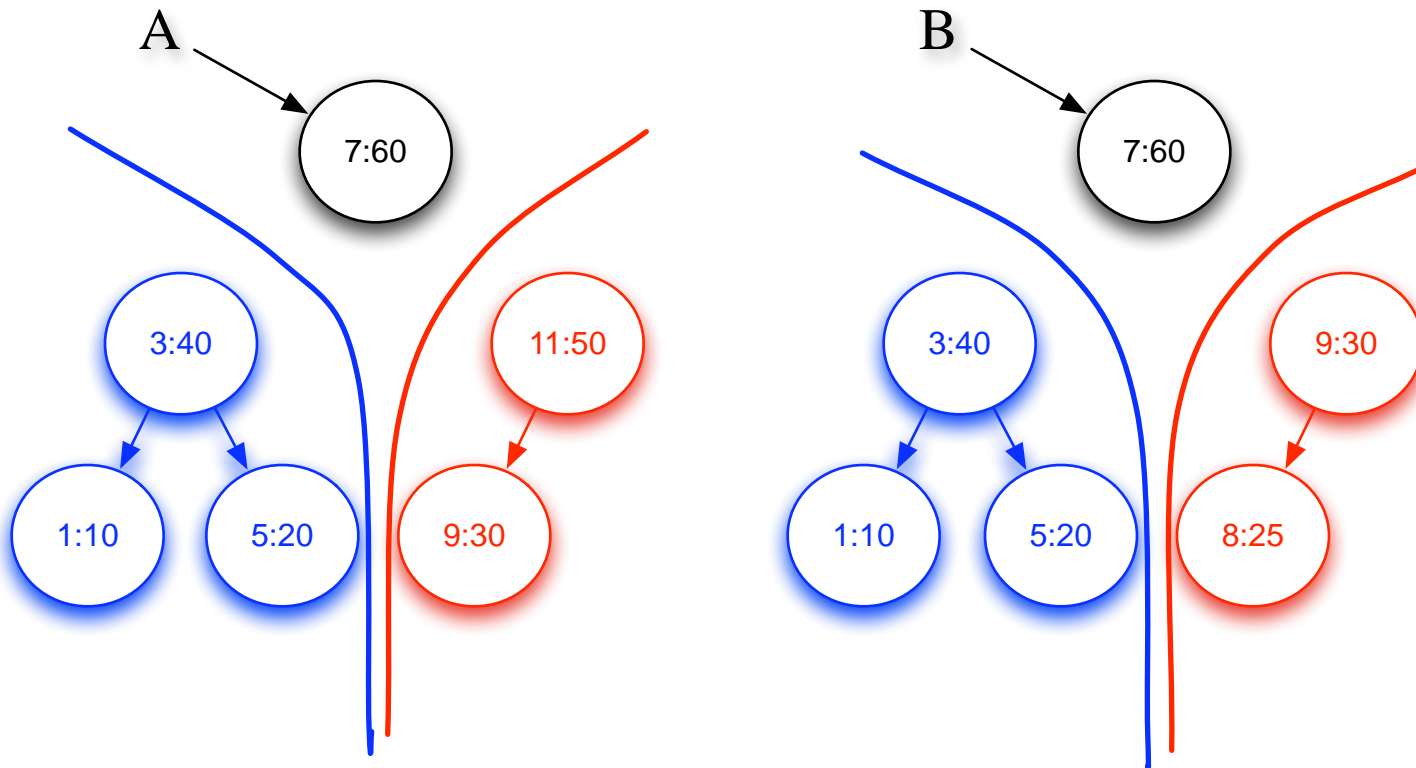
Parallelism

Suppose we want to find the union of two sets, A and B, represented as treaps.



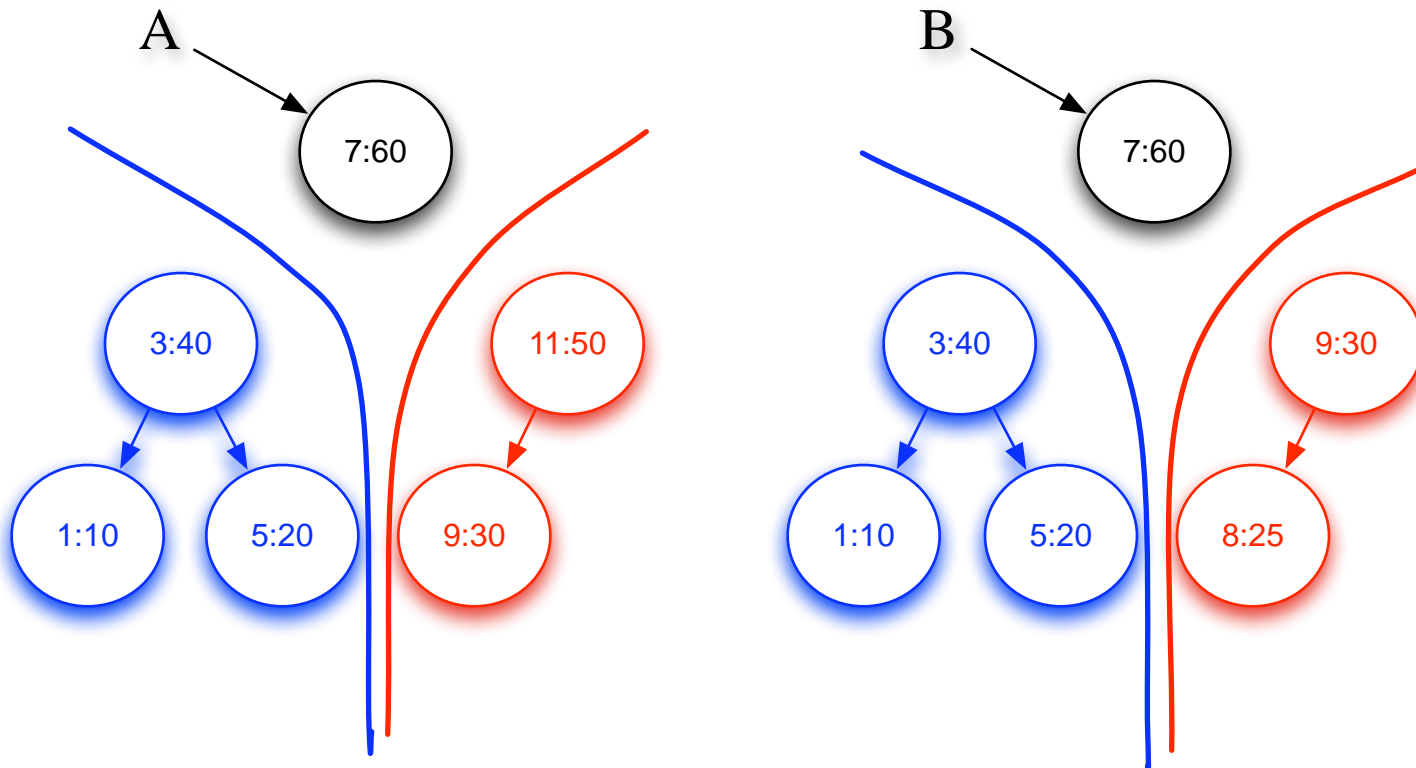
Parallelism

Suppose we want to find the union of two sets, A and B, represented as treaps.



Parallelism

Suppose we want to find the union of two sets, A and B, represented as treaps.



Hash consing helps a lot. Memoization and reference counting are also practical.

Easy optimizations

Suppose we want to compute $A \cup (B \cap C)$.

Dumb way:

```
union2(A, intersect2(B, C))
```

Smart way:

```
customUnionIntersect(A, B, C)
```

where the code for `customUnionIntersect` is generated by the compiler, on demand.

Need to explore the limits of this idea. Certainly it works well for expressions constructed using α .

Harder optimizations

Sometimes large mappings must be explicitly constructed,

e.g., as the result of a β

What's the best choice of data structure?

Experience from SETL effort should apply.

Across an expression tree, a tool like BURG may be helpful.

Summary

We like applicative data structures for the clarity they bring to ordinary programming. For parallel programming, the benefits are multiplied.

My plans for future work have been centered on optimizations; but the more interesting problems are probably language design issues.

- side effects
- explicit synchronization
- equality