Speed and Simplicity for Incremental Sequence Computation

How do we make Incremental Computation easier to use?

How will it compete with the speed of native code?

PROGRESS

Incremental Code

memo table

```
mtable = new Memo;
fn max(l: List) -> Num {
 fn memo(l) -> Num {
  let ma = match mtable.get(a){
   Some(a) => a,
   None => {
    let ma = max(l);
    mtable.put(a,ma);
    ma
   }
  }
  let (a,b) = match split(l) {
   None => l.pop(),
   Some((a,b)) => {
    bin_max(memo(a),memo(b))
   }
  }
 }
}
```

Explicit memo table

Table handling

Explicit subsequence

library

```
fn max(ml: MemoList) -> Num {
 let l = read(ml);
 let (a,b) = match split(l) {
  None => l.pop(),
  Some((a,b)) => {
   bin_max(
    memo(max(write(a))),
    memo(max(write(b)))
   )
  }
 }
}
```

read/write data for dependency tracking

Explicit subsequence

automatic change propagation

collections

```
fn max(l: Giraz) -> Num {
 namespace("calc_max",||{
  l.fold_lr(|a,b|{
   bin_max(a,b)
  })
 })
}
```

action labels

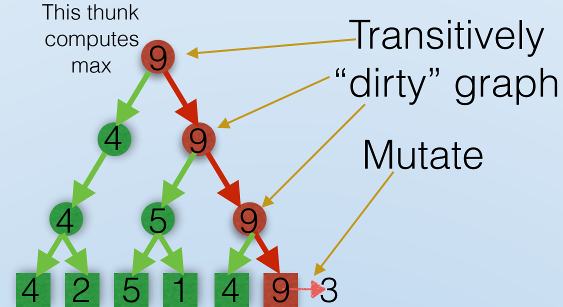
basic code

automatic change propagation

Prior incremental code, like the **memo-table and library versions** to the left, required users to think directly about the incremental strategies used. This becomes limiting when trying to divide up work into subproblems. The user must consider rearranging code that spans multiple recursive calls.

Incremental computation works best with **large data sets** or long-running computations. Incremental libraries are well-suited for long-running computations, but not for simple computations on large datasets. The collections strategy makes up for this deficiency.

Dependency Graphs

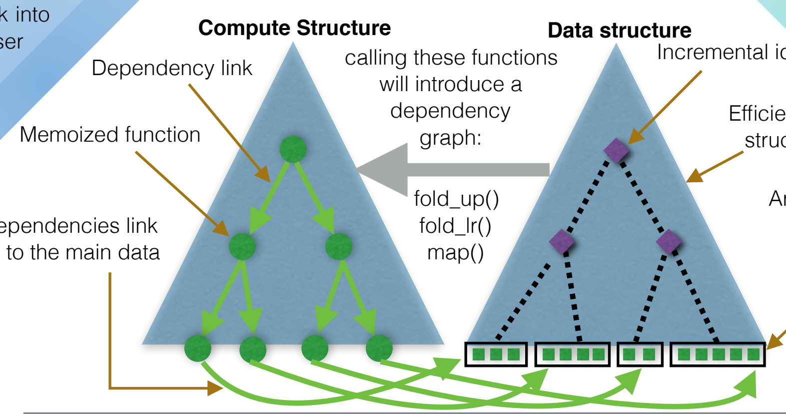


Dependency graphs are important for general purpose incremental computation. Nodes in the graph store code. When changing an input node, all dependents are marked as needing re-computation. In this way, dependency graphs can abstract some of the incremental code management away from the user.

Introducing:

Giraz

A collection for incremental sequences



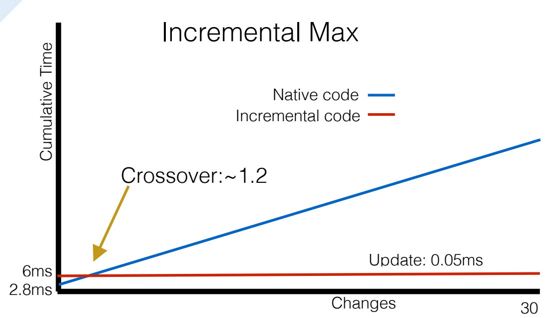
The Giraz is based on the RAZ data structure. It has an **edit mode and a compute mode**. When in edit mode, the user can change data as if they had a cursor between two linked lists. In compute mode, the user calls one of the provided methods with some non-incremental code they wrote. All the incremental work is done by the Giraz.

The Giraz **stores data in arrays** internally, which may or may not be exposed to the user, depending on the incremental method call. The array boundaries are defined when the data is added to the collection. The user calls **"archive" functions** instead of inserting a data element. These calls insert markers used internally to structure the whole collection.

To test the Giraz we compare with native code. We run a similar computation over the Giraz and a single array. We insert a new item in a random location and re-run the computation. The Giraz completes the re-computation much faster than the array!

max	find the largest number in the sequence	58x Speedup
quickhull	find the subset of points that surround the others	4x Speedup
calc	parse numbers and symbols and perform the calculation	24x Speedup
to_string	change all the numbers to text	450x Speedup
reverse	change the order of the elements	22x Speedup

There were 10^6 elements and most tests updated in under 1ms.



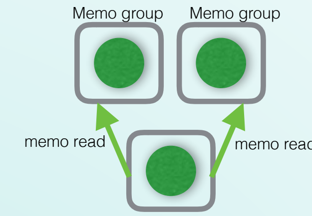
Systems language

Incremental computation requires a lot of memory management. References and garbage collectors in some languages can get in the way. The best choice is a systems language like rust that does not have a garbage collector.

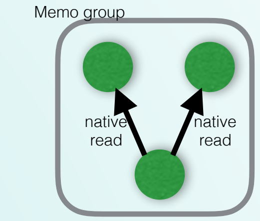
"Rust is a systems programming language that runs blazingly fast, prevents segfaults, and guarantees thread safety."

Garbage collectors are usually great for languages that use references and linked data goes out of scope during execution. With incremental computation, large dependency graphs are stored and their scope doesn't necessarily match the program structure. The garbage collector ends up traversing these graphs, spending a lot of time searching for stale data that doesn't exist.

Overhead Reduction

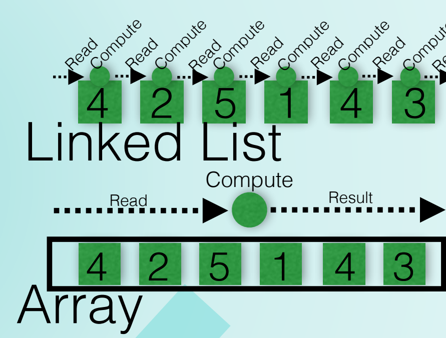


Many incremental computation libraries use one memoized thunk per user function. This requires a lot of overhead.



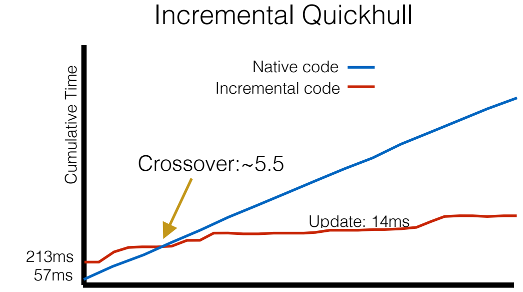
Grouping up functions and running them together when the thunk requires recompilation can save a lot of time looking up parameters.

Cache Coherency



Arrays require far fewer memory accesses than linked lists. Using arrays prevents pauses in computation. In incremental computation, reads have to check whether the data has changed or if it's in the memo table, requiring more time.

The Adapton Incremental Computation Engine uses "names" as memo table keys. They allow more **complex usage of memorization primitives**. Making good use of names in complex situations is a difficult research problem, but we can encode some working strategies into our collections abstractions. As more research comes out, we can improve the library without asking users to change their code.



by Kyle Headley

More info at kyleheadley.github.io

CUPLV

Programming Languages and Verification

University of Colorado Boulder