Incremental Interactive Computation

Matthew Hammer
hammer@cs.umd.edu
Batch Computation vs Interactive Computation

**Batch Model**
- No time, no dialogue

**Interactive Model**
- Dialogue in time
Interaction is a Dialogue

Interacting User → Reactive Computation

- Time
- Creates input
- Demands Output
- Mutates input
- Demands Output

Tuesday, October 7, 2014
Elements of Interactive Dialogue

• User and system interact across time
• User mutates / changes input structure
• User demands / observes output structure
• The system maintains a correspondence between input and output structures
• I/O correspondence is computational
Incremental Interactive Computation
Incremental Interactive Computation

Claim: Interesting interactive systems consist of incremental computations
**Example: Spreadsheets**

<table>
<thead>
<tr>
<th>Input Structure</th>
<th>Cell Formulae</th>
<th>Formula evaluation</th>
<th>Cell values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Incremental Computation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output Structure</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Spreadsheet Example:****

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>x^2</td>
<td>xy</td>
</tr>
<tr>
<td>2</td>
<td>1.5</td>
<td>5.1</td>
<td>2.25</td>
</tr>
<tr>
<td>3</td>
<td>2.3</td>
<td>2.4</td>
<td>5.29</td>
</tr>
<tr>
<td>4</td>
<td>3.1</td>
<td>0.6</td>
<td>9.61</td>
</tr>
<tr>
<td>5</td>
<td>3.9</td>
<td>-2.7</td>
<td>15.21</td>
</tr>
<tr>
<td>6</td>
<td>6.2</td>
<td>-6</td>
<td>38.44</td>
</tr>
<tr>
<td>7</td>
<td>17</td>
<td>-0.6</td>
<td>70.8</td>
</tr>
<tr>
<td>10</td>
<td>=((D7-A7<em>B7)/(C7-A7</em>A7))</td>
<td>a</td>
<td>b</td>
</tr>
</tbody>
</table>

**Cell Formulae Example:**

- A7 = 17
- B7 = -0.6
- C7 = 70.8
- D7 = -32.7
Example: Word processing

- **Input Structure**
- **Incremental Computation**
- **Document Content**
- **Spell-check each word**
- **Output Structure**
- **Highlight misspellings**

IC systems work by recording traces of computations and then reusing portions of those traces as inputs change. Unfortunately, prior IC approaches have two major limitations in trace reuse. For example, consider using IC to implement a spreadsheet, so that visible formulae are minimally...
<table>
<thead>
<tr>
<th>Input</th>
<th>Text file</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computation</td>
<td>Parse tokens</td>
</tr>
<tr>
<td>Output</td>
<td>AST</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Input</th>
<th>AST</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computation</td>
<td>Lexical scoping</td>
</tr>
<tr>
<td>Output</td>
<td>Def/use edges</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Input</th>
<th>AST + Def/use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computation</td>
<td>Type inference</td>
</tr>
<tr>
<td>Output</td>
<td>Type errors</td>
</tr>
</tbody>
</table>
Computing Incrementally

1. Input changes are gradual
   Full re-computation is often **redundant**

2. Output observation is limited
   Full re-computation is often **overly eager**
Computing Incrementally

1. Input changes are gradual

2. Output observation is limited

Example: Spreadsheet
Cells change slowly

Example: Word processing
Document and dictionary both change slowly

Example: Programming
Program changes slowly
Computing Incrementally

1. Input changes are gradual

2. Output observation is limited

Example:
Spreadsheet
One worksheet is active, others hidden

Example:
Word processing
Viewport shows one or two pages

Example:
Programming
Viewport shows one file, module or function
Adapton

Programming Abstractions for Incremental Interaction
Adapton Programming Abstractions

- **Mutable references:**
  - Hold changing input structure

- **Lazy thunks:**
  - Demand-driven computations
  - Output structure
Mutable references

Incremental Computation (thunks)

Demand-driven Outputs
Mutable references

Incremental Computation (thunks)

Demand-driven Outputs
Mutable references

Incremental Computation (thunks)

Demand-driven Outputs

Switch Demand:
Mutable references

Incremental Computation (thunks)

Demand-driven Outputs

Switch Demand:
Mutable references

Incremental Computation (thunks)

Demand-driven Outputs

Sharing

1

2

3

4

max

2

+ 5

min

3

max

7

+ 7

max

7

* 15

min

3

Tuesday, October 7, 2014
Input Change:

Mutable references

Incremental Computation (thunks)

Demand-driven Outputs

1 → max 2 → + 7 → max 7 → 15
2 → + 5 → 15
3 → min 3
4 → min 3

Tuesday, October 7, 2014
Mutable references

Incremental Computation (thunks)

Demand-driven Outputs

Some previous results are affected
Mutable references

Incremental Computation (thunks)

Demand-driven Outputs

Demand new output:
Mutable references

Incremental Computation (thunks)

Demand-driven Outputs

Demand new output:
Lazy Structures

• **Spreadsheet example:**
  • Each thunk returns a **single number**

• **Lazy Lists:**
  • Each thunk returns `Nil` or `Cons`
  • `Cons` holds head value and a thunk tail
  • Laziness can be applied to **trees, graphs,** and essentially any other data structure

• **Inputs:** Special thunks are **mutable**
Background: Lazy Lists

type 'a thunk = unit -> 'a

let force : 'a thunk -> 'a
  = fun t -> t ()

let rec from_list l =
  match l with
  | []   -> `Nil
  | h::t -> `Cons(h,fun() -> from_list t)
type 'a lzlist = [ `Nil | `Cons of 'a * ('a lzlist thunk) ]

let rec merge l1 l2 = function
| l1, `Nil -> l1
| `Nil, l2 -> l2
| `Cons(h1,t1), `Cons(h2,t2) ->
  if h1 <= h2 then
  `Cons(h1, fun() -> merge (force t1) l2)
  else
  `Cons(h2, fun() -> merge l1 (force t2))
Mergesort Example

Input: [3, 5, 8, 2, 1, 7]

Singletons: [[3], [5], [8], [2], [1], [7]]

Merge #1: [[3, 5], [2, 8], [1, 7]]

Merge #2: [[3, 5], [1, 2, 7, 8]]

Merge #3: [[3, 5], [1, 2, 7, 8]]

Merge #4: [[1, 2, 3, 5, 7, 8]]

Flatten: [1, 2, 3, 5, 7, 8]
Course Project
Project: Interactive Program Analysis

- Assume: Interactive “Structure Editor”
- Programmer manipulates AST directly
- Learn: Adapton IC framework
- Build: Incremental Program Analysis
- Example: Use/def information
- Example: Type Inference
- Example: Control-flow analysis
Background: Structure Editors

- Philosophical claims:
  - Programs consist of **rich structure**
  - Rich interaction exposes this structure

- Example prototypes:
  - Haskell -- [http://www.youtube.com/watch?v=v2ypDcUM06U](http://www.youtube.com/watch?v=v2ypDcUM06U)
  - Citris -- [http://www.youtube.com/watch?v=47UcOspbZ2k](http://www.youtube.com/watch?v=47UcOspbZ2k)
  - TouchDevelop -- [http://www.youtube.com/watch?v=a6GRg2glKpc](http://www.youtube.com/watch?v=a6GRg2glKpc)