Towards practical incremental recomputation for scientists

Philip J. Guo and Dawson Engler
Workshop on the Theory and Practice of Provenance
Feb 22, 2010
Talk outline

1. **Motivation**: ad-hoc data analysis scripts

2. **Technique**: fully automatic memoization

3. **Benefits**: faster iteration with simple code
Types of programs

Size & complexity

All programs written

Research prototypes

Data munging and analysis scripts
Problem

Scientific data processing and analysis scripts often execute for several minutes to hours, which slows down the scientist’s iteration and debugging cycle.
Manually coping

Write initial single-file Python prototype

Re-write to break computation into multiple stages (files) and selectively comment-out code to test

Write code to cache intermediate results to disk and manually manage dependencies

Iteration / re-execution time

Code size and complexity
Let interpreter cache and manage intermediate results

Code amount and complexity

Iteration / re-execution time

Write initial single-file Python prototype

Automated solution
Ideal workflow

1. Write simple first version of script
2. Execute and wait for 1 hour to get results
3. Interpret results and notice a bug
4. Edit script slightly to fix that bug
5. Re-execute and wait for a few seconds
6. Enhance script with new functions
7. Re-execute and wait for a few minutes
Technique

Fully automatic and persistent memoization for a general-purpose imperative language
Traditional memoization

def \textbf{Fib}(n):
    \textbf{if} n \leq 2:
        \textbf{return} 1
    \textbf{else}:
        \textbf{return} \textbf{Fib}(n-1) + \textbf{Fib}(n-2)
Traditional memoization

MemoTable = {}

def Fib(n):
    if n <= 2:
        return 1
    else:
        if n in MemoTable:
            return MemoTable[n]
        else:
            MemoTable[n] = Fib(n-1) + Fib(n-2)
            return MemoTable[n]

<table>
<thead>
<tr>
<th>Input (n)</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>13</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Auto-memoizing real programs

1. Code changes
2. External dependencies
3. Side-effects
def \texttt{stageC}(\texttt{datLst}): 
    res = ...  \# run for 10 minutes munging \texttt{datLst} 
    return res

Auto-memoizing real programs: 
Detecting code changes

<table>
<thead>
<tr>
<th>Input (datLst)</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>{1,2,3,4}</td>
<td>10</td>
</tr>
<tr>
<td>{5,6,7,8}</td>
<td>20</td>
</tr>
<tr>
<td>{9,10,11,12}</td>
<td>30</td>
</tr>
</tbody>
</table>
def stageC(datLst):
    res = ... # run for 10 minutes munging datLst
    return res

Auto-memoizing real programs:
Detecting code changes

<table>
<thead>
<tr>
<th>Input (datLst)</th>
<th>Code deps.</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1,2,3,4]</td>
<td>stageC -&gt; C₁</td>
<td>10</td>
</tr>
<tr>
<td>[5,6,7,8]</td>
<td>stageC -&gt; C₁</td>
<td>20</td>
</tr>
<tr>
<td>[9,10,11,12]</td>
<td>stageC -&gt; C₁</td>
<td>30</td>
</tr>
</tbody>
</table>
Auto-memoizing real programs: Detecting code changes

def stageC(datLst):
    res = ...  # run for 10 minutes munging datLst
    return (res * -1)

<table>
<thead>
<tr>
<th>Input (datLst)</th>
<th>Code deps.</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1,2,3,4]</td>
<td>stageC -&gt; C₁</td>
<td>10</td>
</tr>
<tr>
<td>[5,6,7,8]</td>
<td>stageC -&gt; C₁</td>
<td>20</td>
</tr>
<tr>
<td>[9,10,11,12]</td>
<td>stageC -&gt; C₁</td>
<td>30</td>
</tr>
</tbody>
</table>
Auto-memoizing real programs: Detecting code changes

```python
def stageC(datLst):
    res = ...  # run for 10 minutes munging datLst
    return (res * -1)
```

<table>
<thead>
<tr>
<th>Input (datLst)</th>
<th>Code deps.</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1,2,3,4]</td>
<td>stageC -&gt; C₁</td>
<td>10</td>
</tr>
<tr>
<td>[5,6,7,8]</td>
<td>stageC -&gt; C₁</td>
<td>20</td>
</tr>
<tr>
<td>[9,10,11,12]</td>
<td>stageC -&gt; C₁</td>
<td>30</td>
</tr>
<tr>
<td>[1,2,3,4]</td>
<td>stageC -&gt; C₂</td>
<td>-10</td>
</tr>
</tbody>
</table>
Auto-memoizing real programs: Detecting file reads

def stageB(queryStr):
    db = sql_open_db("test.db")
    q = db.query(queryStr)
    res = ... # run for 10 minutes processing q
    return res

<table>
<thead>
<tr>
<th>Input (queryStr)</th>
<th>Code deps.</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT * FROM tbl1</td>
<td>stageB -&gt; B₁</td>
<td>1</td>
</tr>
<tr>
<td>SELECT * FROM tbl2</td>
<td>stageB -&gt; B₁</td>
<td>2</td>
</tr>
</tbody>
</table>
Auto-memoizing real programs:
Detecting file reads

```python
def stageB(queryStr):
    db = sql_open_db("test.db")
    q = db.query(queryStr)
    res = ... # run for 10 minutes processing q
    return res
```

<table>
<thead>
<tr>
<th>Input (queryStr)</th>
<th>Code deps.</th>
<th>File deps.</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT * FROM tbl1</td>
<td>stageB -&gt; B₁</td>
<td>test.db -&gt; DB₁</td>
<td>1</td>
</tr>
<tr>
<td>SELECT * FROM tbl2</td>
<td>stageB -&gt; B₁</td>
<td>test.db -&gt; DB₁</td>
<td>2</td>
</tr>
</tbody>
</table>
Auto-memoizing real programs: Detecting global variable reads

MULTIPLIER = 5

def stageB(queryStr):
    db = sql_open_db("test.db")
    q = db.query(queryStr)
    res = ... # run for 10 minutes processing q
    return (res * MULTIPLIER)

<table>
<thead>
<tr>
<th>Input (queryStr)</th>
<th>Code deps.</th>
<th>File deps.</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT * FROM tbl1</td>
<td>stageB -&gt; B₂</td>
<td>test.db -&gt; DB₁</td>
<td>5</td>
</tr>
</tbody>
</table>
Auto-memoizing real programs: Detecting global variable reads

MULTIPLIER = 5

def stageB(queryStr):
    db = sql_open_db("test.db")
    q = db.query(queryStr)
    res = ... # run for 10 minutes processing q
    return (res * MULTIPLIER)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT * FROM tbl1</td>
<td>stageB -&gt; B₂</td>
<td>test.db -&gt; DB₁</td>
<td>MULTIPLIER -&gt; 5</td>
<td>5</td>
</tr>
</tbody>
</table>
Auto-memoizing real programs:
Detecting global variable reads

MULTIPLIER = 10
def stageB(queryStr):
    db = sql_open_db("test.db")
    q = db.query(queryStr)
    res = ... # run for 10 minutes processing q
    return (res * MULTIPLIER)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT * FROM tbl1</td>
<td>stageB -&gt; B₂</td>
<td>test.db -&gt; DB₁</td>
<td>MULTIPLIER -&gt; 5</td>
<td>5</td>
</tr>
<tr>
<td>SELECT * FROM tbl1</td>
<td>stageB -&gt; B₂</td>
<td>test.db -&gt; DB₁</td>
<td>MULTIPLIER -&gt; 10</td>
<td>10</td>
</tr>
</tbody>
</table>
Auto-memoizing real programs: Detecting transitive dependencies

```python
def stageA(filename):
    lst = []
    for line in open(filename):
        lst.append(stageB(line))
    transformedLst = stageC(lst)
    return sum(transformedLst)
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>queries.txt</td>
<td>stageA -&gt; A₁</td>
<td>queries.txt -&gt; Q₁</td>
<td></td>
<td>50</td>
</tr>
</tbody>
</table>
def stageA(filename):
    lst = []
    for line in open(filename):
        lst.append(stageB(line))
    transformedLst = stageC(lst)
    return sum(transformedLst)

def stageB(queryStr):
    db = sql_open_db("test.db")
    q = db.query(queryStr)
    res = ... # run for 10 minutes processing q
    return (res * MULTIPLIER)

def stageC(datLst):
    res = ... # run for 10 minutes munging datLst
    return res

queries.txt

MULTIPLIER = 5

test.db

Auto-memoizing real programs:
Detecting transitive dependencies
Auto-memoizing real programs: Detecting transitive dependencies

```python
def stageA(filename):
    lst = []
    for line in open(filename):
        lst.append(stageB(line))
    transformedLst = stageC(lst)
    return sum(transformedLst)
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>queries.txt</td>
<td>stagesA -&gt; A₁</td>
<td>queries.txt -&gt; Q₁</td>
<td>MULTIPLIER -&gt; 5</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>stagesB -&gt; B₁</td>
<td>test.db -&gt; DB₁</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>stagesC -&gt; C₁</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
“Before memoizing a given routine, the programmer needs to verify that there is no internal dependency on side effects. This is not always simple; despite attempts to encourage a functional programming style, programmers will occasionally discover that some routine their function depended upon had some deeply buried dependence on a global variable or the slot value of a CLOS [Common Lisp Object System] Instance.” [Hall and Mayfield, 1993]
Auto-memoizing real programs: Detecting impurity

• All functions start out pure
• Mark all functions on stack as impure when:
  – Mutating a non-local value
  – Writing to a file
  – Calling a non-deterministic function
• Data analysis functions mostly pure
def stageA(filename):
    lst = []
    for line in open(filename):
        lst.append(stageB(line))
    transformedLst = stageC(lst)
    return sum(transformedLst)

def stageB(queryStr):
    db = sql_open_db("test.db")
    q = db.query(queryStr)
    res = ... # run for 10 minutes processing q
    return (res * MULTIPLIER)

def stageC(datLst):
    res = ... # run for 10 minutes munging datLst
    return res

MULTIPLIER = 5
Benefits

1. Less code and bugs
2. Faster iteration cycle
3. Real-time collaboration
Talk review

1. **Motivation**: ad-hoc data analysis scripts

2. **Technique**: fully automatic memoization

3. **Benefits**: faster iteration with simple code
Ongoing and future work

• Provenance browsing
• Database-aware caching
• Network-aware caching
• Lightweight programmer annotations
• Finer-grained tracking within functions
Implementation

- **Python** as target language
- **Plug-and-play** with no code changes
- **Low** run-time overhead
- Compatible with 3\textsuperscript{rd}-party **libraries**