Lazy computing

Thibaut Balabonski
GALLIUM

November 20, 2012
Solving problems with computers

Problem

Program

Computer
Solving problems with computers

Problem

Program

Computer

$(1 + 2) \times (3 + 4)$
Solving problems with computers

Problem

Program

Computer

\[(1 + 2) \times (3 + 4)\]

<table>
<thead>
<tr>
<th>Const 3</th>
<th>Offsetint 4</th>
<th>Push</th>
</tr>
</thead>
<tbody>
<tr>
<td>Const 1</td>
<td>Offsetint 2</td>
<td>Mulint</td>
</tr>
<tr>
<td>Return 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Solving problems with computers

Problem

Program

Computer

(1 + 2) \times (3 + 4)

<table>
<thead>
<tr>
<th>Const 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Offsetint 4</td>
</tr>
<tr>
<td>Push</td>
</tr>
<tr>
<td>Const 1</td>
</tr>
<tr>
<td>Offsetint 2</td>
</tr>
<tr>
<td>Mulint</td>
</tr>
<tr>
<td>Return 1</td>
</tr>
</tbody>
</table>

Computer

Memory

7
Solving problems with computers

Problem

Program

Computer

(1 + 2) \times (3 + 4)

| Const 3 |
| Offsetint 4 |
| Push |
| Const 1 |
| Offsetint 2 |
| Mulint |
| Return 1 |

Computer

Memory

7
Solving problems with computers

Problem: $(1 + 2) \times (3 + 4)$

Program:
- Const 3
- Offsetint 4
- Push
- Const 1
- Offsetint 2
- Mulint
- Return 1

Computer:
- 1
- Memory:
  - 7
Solving problems with computers

<table>
<thead>
<tr>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1 + 2) × (3 + 4)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>Const 3</td>
</tr>
<tr>
<td>Offsetint 4</td>
</tr>
<tr>
<td>Push</td>
</tr>
<tr>
<td>Const 1</td>
</tr>
<tr>
<td>Offsetint 2</td>
</tr>
<tr>
<td>Mulint</td>
</tr>
<tr>
<td>Return 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Computer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>7</td>
</tr>
</tbody>
</table>
Solving problems with computers

Problem

Program

Computer

\[(1 + 2) \times (3 + 4)\]

<table>
<thead>
<tr>
<th>Const 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Offsetint 4</td>
</tr>
<tr>
<td>Push</td>
</tr>
<tr>
<td>Const 1</td>
</tr>
<tr>
<td>Offsetint 2</td>
</tr>
<tr>
<td>Mulint</td>
</tr>
<tr>
<td>Return 1</td>
</tr>
</tbody>
</table>

Computer

Memory

21
Solving problems with computers

\[(1 + 2) \times (3 + 4)\]

<table>
<thead>
<tr>
<th>Const 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Offsetint 4</td>
</tr>
<tr>
<td>Push</td>
</tr>
<tr>
<td>Const 1</td>
</tr>
<tr>
<td>Offsetint 2</td>
</tr>
<tr>
<td>Mulint</td>
</tr>
<tr>
<td>Return 1</td>
</tr>
</tbody>
</table>

Computer

Memory
Solving problems with computers

Gallium-related issues:
- Programming languages and compilation
- Emphasis on safety
Solving problems with computers

Gallium-related issues:
- Programming languages and compilation
- Emphasis on safety

In this talk:
- Emphasis on **efficiency**
Chosing a path

\[ (1 + 2) \times (3 + 4) \]

\[ 3 \times (3 + 4) \]

\[ 3 \times 7 \]

\[ 21 \]

\[ (1 + 2) \times 7 \]
Freedom implies responsibility

- Outermost prevents unneeded computations.
- Innermost prevents duplication of subprograms.
Freedom implies responsibility

- Outermost prevents unneeded computations.
- Innermost prevents duplication of subprograms.

\[
\text{sq}(e) = e \times e
\]

\[
\text{sq}(1 + 2) = (1 + 2) \times (1 + 2) = 3 \times (1 + 2) = 3 \times 3 = 9
\]

\[
\text{sq}(3) = 3 \times 3 = 9
\]
Freedom implies responsibility

- Outermost prevents unneeded computations.
- Innermost prevents duplication of subprograms.

\[ \text{sq}(0 \times (1 + 2)) \]
Evaluation strategies

Goal

Minimize the number of rewriting steps.

Question

In which order should we perform the steps?
Richer programming languages

- Functions
- Data structures

\[
\text{map}(f, \text{[]}) = \text{[]}
\]
\[
\text{map}(f, x : xs) = f(x) : \text{map}(f, xs)
\]

> \text{map}(\text{sq}, 1 : 2 : 3 : \text{[]})
1 : 4 : 9 : \text{[]}
Richer programming languages

- Functions
- Data structures

map(f, [ ]  ) = [ ]
map(f, x : xs) = f(x) : map(f, xs)

> map(sq, 1 : 2 : 3 : [ ])
1 : 4 : 9 : [ ]

Second-order rewriting
Lambda-calculus: computing with functions
Church, 1936

\[ (\lambda x. f) a \]

\[ (\lambda x. xx)((\lambda y. y) a) \]

\[ ((\lambda y. y) a)((\lambda y. y) a) \]
Shortest simple path

Theorem: uncomputability (Barendregt et al. 1976)

Optimal strategies for the \( \lambda \)-calculus cannot be computable.
Wadsworth’s call-by-need (1971)

\[
(\lambda x.xx)((\lambda y.y)a) \rightarrow (\lambda y.y)a
\]

\[
(\lambda y.y)a \rightarrow a((\lambda y.y)a)
\]

\[
((\lambda y.y)a)((\lambda y.y)a) \rightarrow a((\lambda y.y)a)
\]

\[
(\lambda x.xx)((\lambda y.y)a) \rightarrow a((\lambda y.y)a)
\]

\[
(\lambda x.xx)a
\]
Wadsworth’s call-by-need (1971)

((\lambda y \cdot y)a)((\lambda y \cdot y)a)

((\lambda x \cdot xx)(\lambda y \cdot y)a)

(\lambda x \cdot xx)a

aa
Real compilers use weak reduction

Restriction on evaluation: not inside functions.

OCaml (Call-by-Value)

Haskell (Call-by-Need)
New features of weak reduction (my work)

- The optimal strategy is still *uncomputable*.
- Call-by-need is *as good as* the optimal strategy.
New features of weak reduction (my work)

- The optimal strategy is still *uncomputable*.
- Call-by-need is *as good as* the optimal strategy.
Shared needed evaluation

- Use shared evaluation.
- Consider only needed steps.
Shared needed evaluation

- Use shared evaluation.
- Consider only needed steps.

Call-by-Need
Shared needed evaluation

- Use shared evaluation.
- Consider only needed steps.
Conclusion

- Optimal strategies are not computable.
- Sharing adds shortcuts to the reduction space.
- Shared evaluation is as good as an optimal strategy and is computable.

**Question:** what is the cost of sharing?