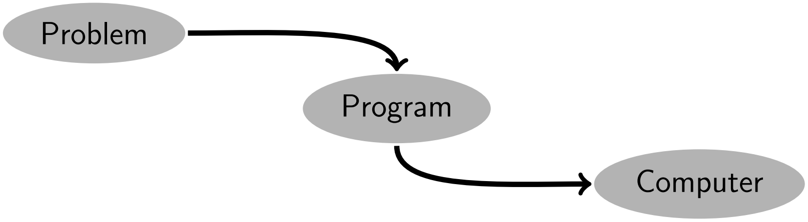
Lazy computing

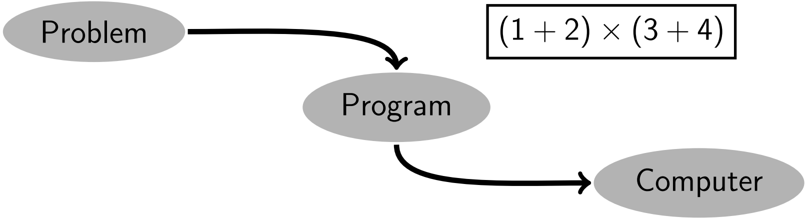
Thibaut Balabonski
GALLIUM

November 20, 2012

Solving problems with computers

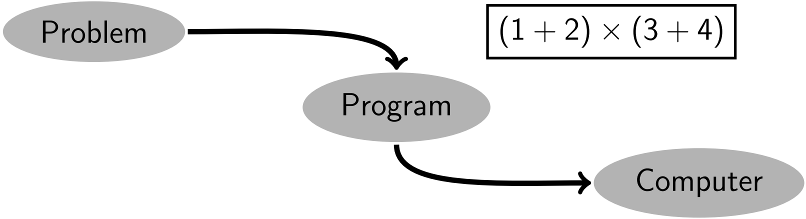


Solving problems with computers



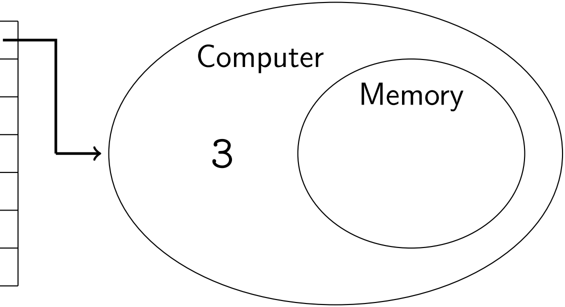
$$\boxed{(1+2)\times(3+4)}$$

Solving problems with computers

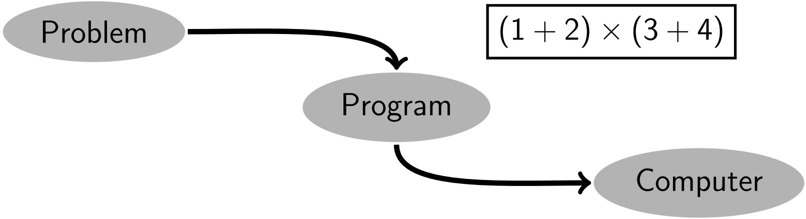


$$\boxed{(1+2)\times(3+4)}$$

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

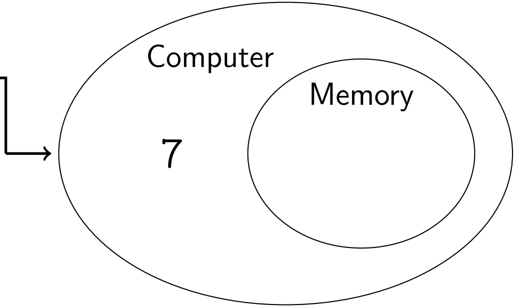


Solving problems with computers

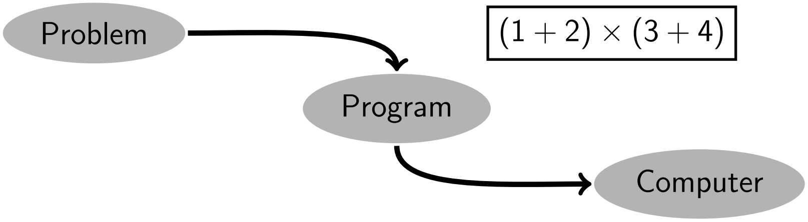


$$\boxed{(1+2)\times(3+4)}$$

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

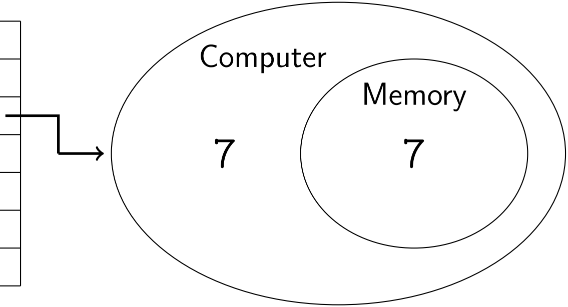


Solving problems with computers

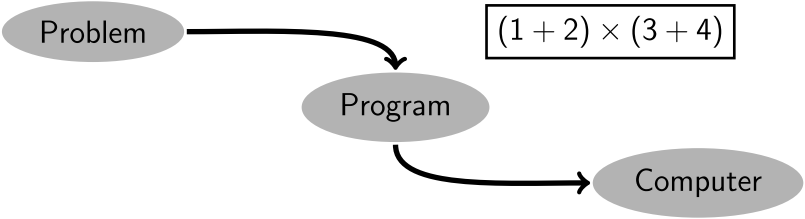


$$\boxed{(1+2) \times (3+4)}$$

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

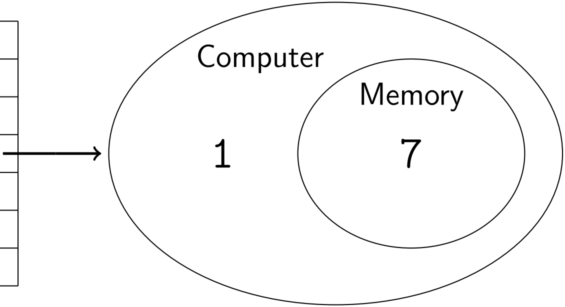


Solving problems with computers

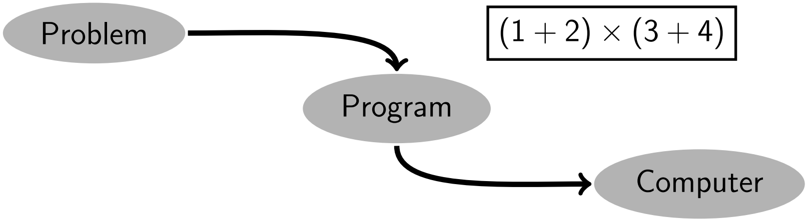


$$\boxed{(1+2)\times(3+4)}$$

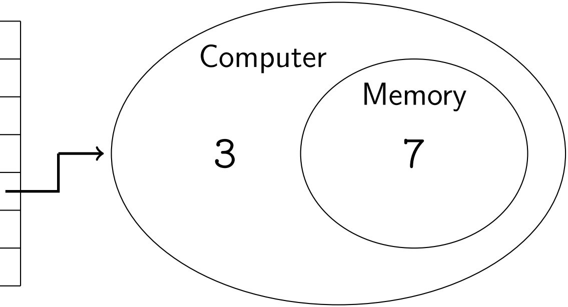
`Const 3`
`Offsetint 4`
`Push`
`Const 1`
`Offsetint 2`
`Mulint`
`Return 1`



Solving problems with computers

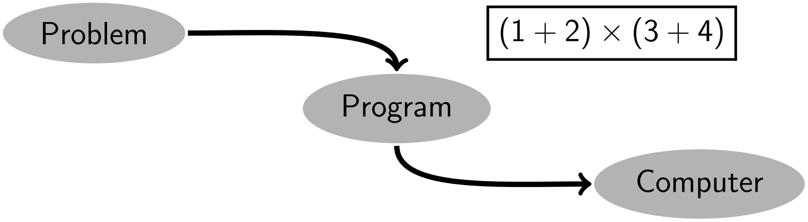



```
Const 3
Offsetint 4
Push
Const 1
Offsetint 2
Mulint
Return 1
```

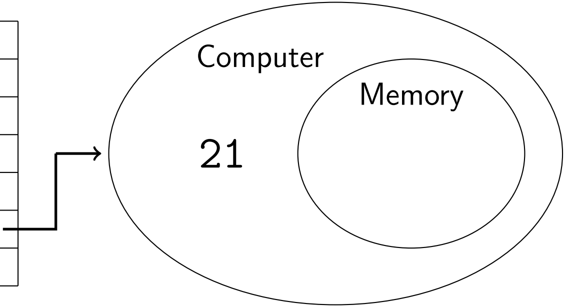


Solving problems with computers

$$\boxed{(1+2)\times(3+4)}$$

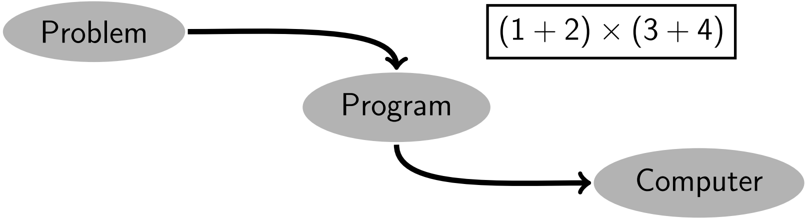


`Const 3`
`Offsetint 4`
`Push`
`Const 1`
`Offsetint 2`
`Mulint`
`Return 1`



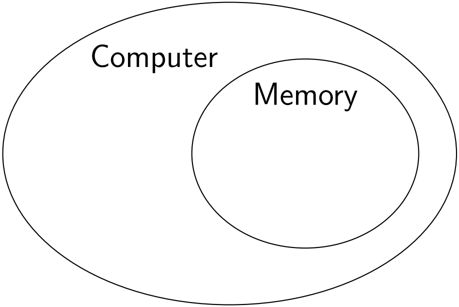
Solving problems with computers

$$\boxed{(1+2)\times(3+4)}$$

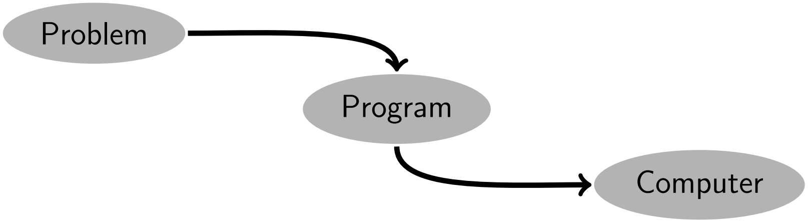


`Const 3`
`Offsetint 4`
`Push`
`Const 1`
`Offsetint 2`
`Mulint`
`Return 1`

21



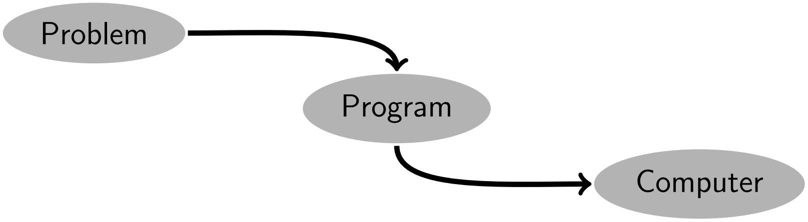
Solving problems with computers



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:

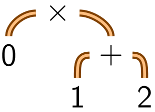
- Emphasis on **efficiency**

Chosing a path

$$
\begin{tikzcd}
& (1+2)\times(3+4) \arrow[dl] \arrow[dr] & \\
3\times(3+4) \arrow[dr] & & (1+2)\times 7 \arrow[dl] \\
& 3\times 7 \arrow[d] & \\
& 21 &
\end{tikzcd}
$$

Freedom implies responsibility

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

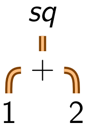


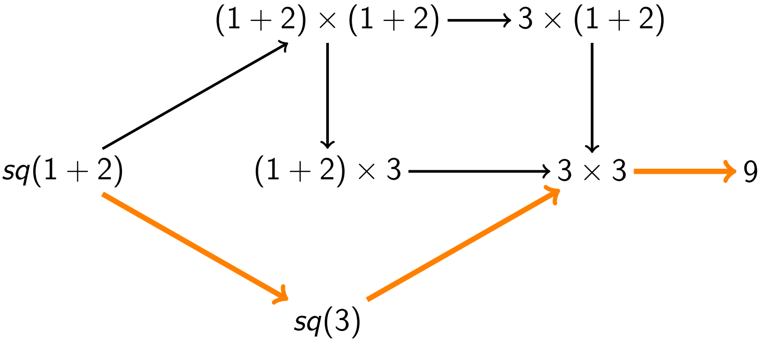
$$\begin{tikzcd}
0 \times (1+2) \arrow[rr] \arrow[dr] & & 0 \\
& 0 \times 3 \arrow[ur] &
\end{tikzcd}$$

Freedom implies responsibility

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

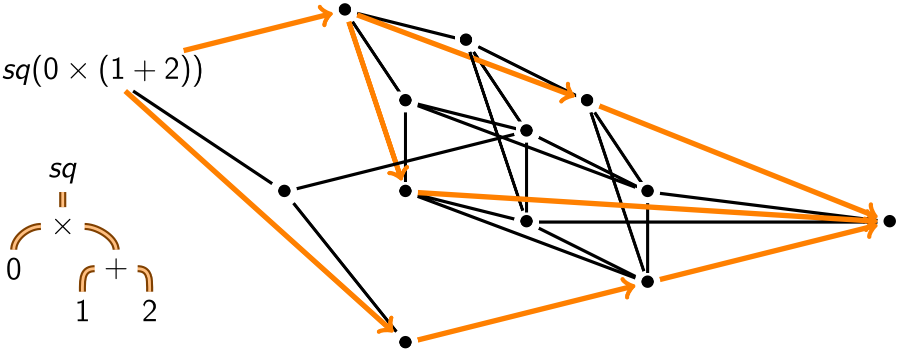
$$sq(e) \;=\; e \times e$$





Freedom implies responsibility

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



Evaluation strategies

Goal

Minimize the number of rewriting steps.

Question

In **which order** should we perform the steps?

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 : []
```

Richer programming languages

- Functions
- Data structures

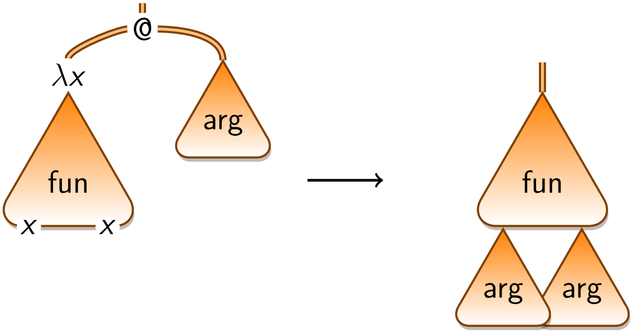
```
map(f, []     )  =  []
map(f, x : xs)  =  f(x) : map(f, xs)

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

Second-order rewriting

Lambda-calculus: computing with functions

Church, 1936



$(\lambda x.f)a$ $\longrightarrow$ $f^{\{x:=a\}}$

$(\lambda x.xx)((\lambda y.y)a)$ $\quad$ $((\lambda y.y)a)((\lambda y.y)a)$

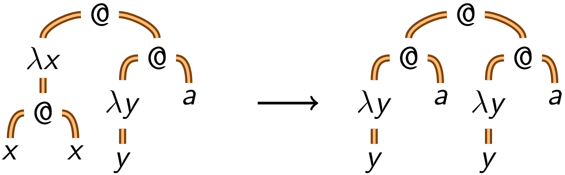
Shortest simple path

$$\begin{tikzcd}
 & ((\lambda y.y)a)((\lambda y.y)a) \arrow[r] \arrow[d] & a((\lambda y.y)a) \arrow[d] \\
(\lambda x.xx)((\lambda y.y)a) \arrow[ur] \arrow[dr] & ((\lambda y.y)a)a \arrow[r] & aa \\
 & (\lambda x.xx)a \arrow[ur] &
\end{tikzcd}$$

Theorem: uncomputability (Barendregt et al. 1976)

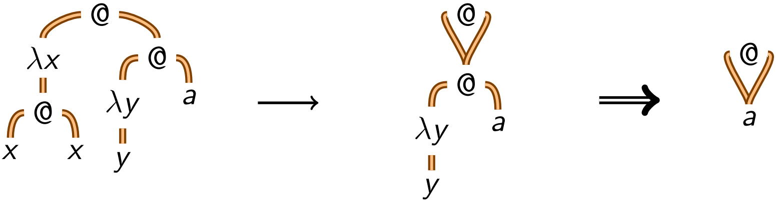
Optimal strategies for the λ-calculus
cannot be computable.

Wadsworth's call-by-need (1971)



$$
\begin{tikzcd}
 & ((\lambda y.y)a)((\lambda y.y)a) \arrow[r] \arrow[d] & a((\lambda y.y)a) \arrow[d] \\
(\lambda x.xx)((\lambda y.y)a) \arrow[ur] \arrow[dr] & ((\lambda y.y)a)a \arrow[r] & aa \\
 & (\lambda x.xx)a \arrow[ur] &
\end{tikzcd}
$$

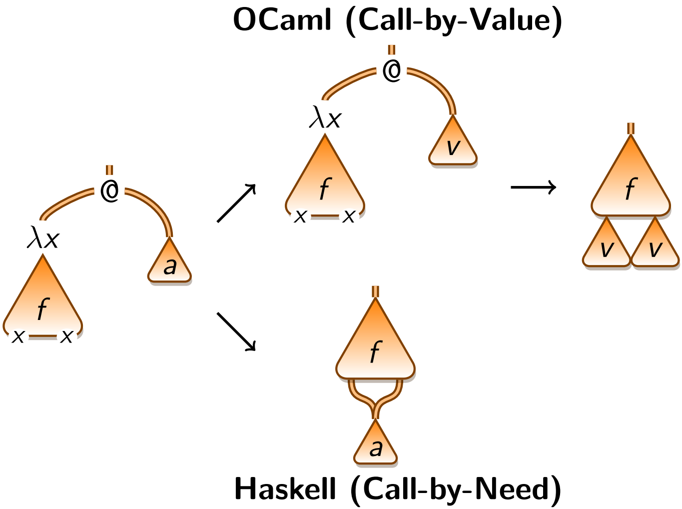
Wadsworth's call-by-need (1971)



$$
\begin{tikzcd}
& ((\lambda y.y)a)((\lambda y.y)a) \arrow[dr, Rightarrow] & \\
(\lambda x.xx)((\lambda y.y)a) \arrow[ur] \arrow[dr] & & aa \\
& (\lambda x.xx)a \arrow[ur] &
\end{tikzcd}
$$

Real compilers use weak reduction

Restriction on evaluation: not inside functions.



New features of weak reduction (my work)

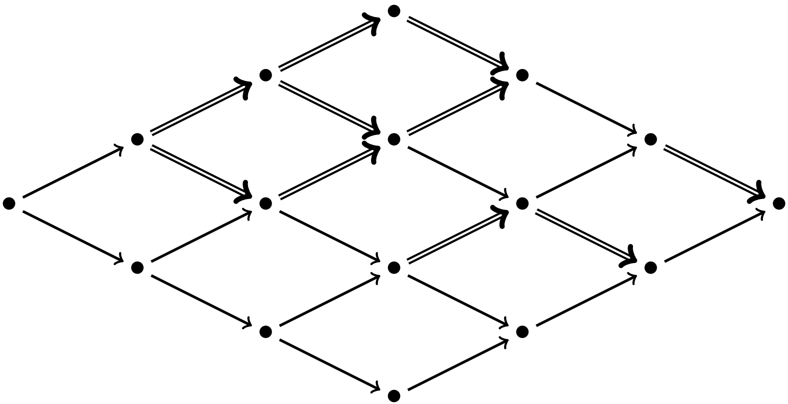
- The optimal strategy is still **uncomputable**.
- Call-by-need is **as good as** the optimal strategy.

New features of weak reduction (my work)

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

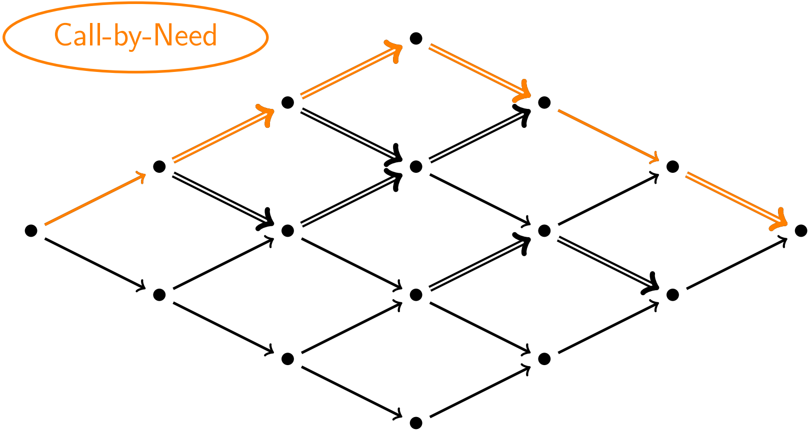
Shared needed evaluation

- Use shared evaluation.
- Consider only needed steps.



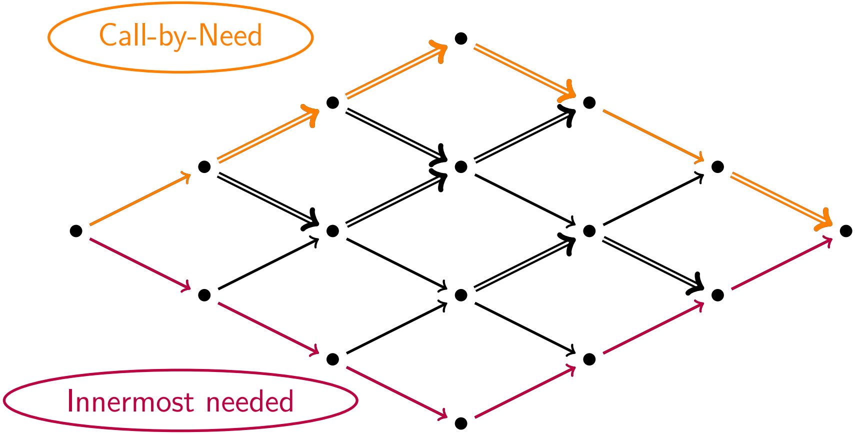
Shared needed evaluation

- Use shared evaluation.
- Consider only needed steps.



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?