Efficient Evaluation of Lazy Programs, or Compilation of Call-by-Need

Junyoung Jang
junyoung.jang@mail.mcgill.ca
McGill University

Declaimer

This presentation is based on

Simon P. Jones and David R. Lester's
Implementing Functional Languages: a Tutorial

Declaimer

This presentation is based on

Simon P. Jones and David R. Lester's
Implementing Functional Languages: a Tutorial

For a "modernized" version of its appendix,
one can see https://github.com/Ailrun/core-lang-haskell

Advantages of Lazy Programs

Advantages of Lazy Programs

- Make it easier to deal with recursion when using combinator libraries

Advantages of Lazy Programs

- Make it easier to deal with recursion when using combinator libraries
- Bring us efficient persistent data structures

Advantages of Lazy Programs

- Make it easier to deal with recursion when using combinator libraries
- Bring us efficient persistent data structures
- Provide a way to encode co-data (e.g. streams)

The Question

Are these advantages real?

The Question

Is it possible to efficiently execute a lazy program?

Difficulties of Efficient Evaluation of Lazy Programs

Difficulties of Efficient Evaluation of Lazy Programs

1. Sharing

Difficulties of Efficient Evaluation of Lazy Programs

1. Sharing
2. (Redirection)

Difficulties of Efficient Evaluation of Lazy Programs

1. Sharing
2. (Redirection)
3. **Instantiation**

Difficulties of Efficient Evaluation of Lazy Programs

1. Sharing
2. (Redirection)
3. **Instantiation**

- What are These Difficulties?

Difficulties of Efficient Evaluation of Lazy Programs

1. Sharing
2. (Redirection)
3. **Instantiation**

- What are These Difficulties?
- How to Solve These Difficulties?

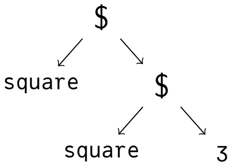
The First Difficulty: Sharing - 1

```
square x = x * x
main = square (square 3)
```

The First Difficulty: Sharing - 1

```
square x = x * x
main = square (square 3)
```

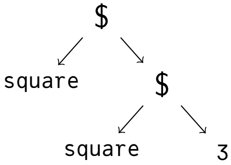
AST for `main`



The First Difficulty: Sharing - 1

```
square x = x * x
main = square (square 3)
```

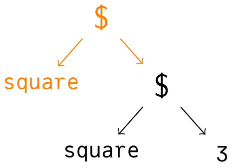
AST for `main`



How can we reduce this tree into `81`?

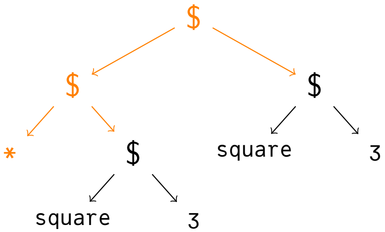
The First Difficulty: Sharing - 2

```
square x = x * x
main = square (square 3)
```



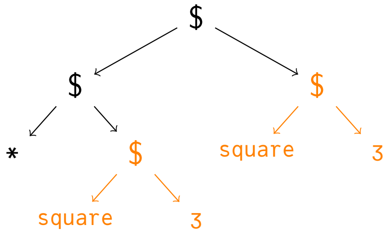
The First Difficulty: Sharing - 2

```
square x = x * x
main = square (square 3)
```



The First Difficulty: Sharing - 2

```
square x = x * x
main = square (square 3)
```

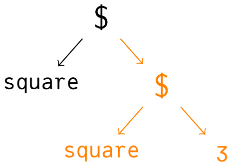


We wastefully repeat the computation!

The First Difficulty: Sharing - 3

Let's share `x` part

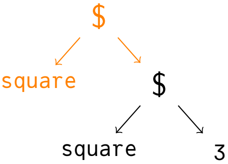
```
square x = x * x
main = square (square 3)
```



The First Difficulty: Sharing - 3

Let's share `x` part

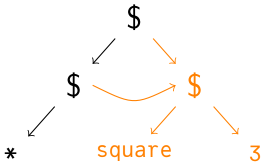
```
square x = x * x
main = square (square 3)
```



The First Difficulty: Sharing - 3

Let's share `x` part

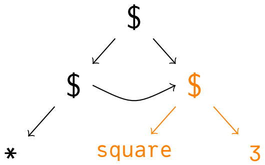
```
square x = x * x
main = square (square 3)
```



The First Difficulty: Sharing - 3

Let's share `x` part

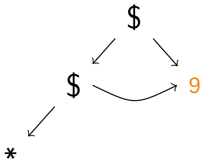
```
square x = x * x
main = square (square 3)
```



The First Difficulty: Sharing - 3

Let's share `x` part

```
square x = x * x
main = square (square 3)
```



The Solution for Sharing

We reduces *a (directed) graph*, *not a tree*.

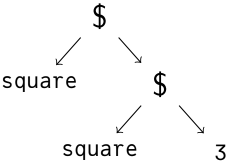
Moreover, we need to *update* a node
(so that multiple refereces share the evaluation cost)

Efficiency Requirement

Each graph reduction step should be as *local and small* as possible
(for further optimizations, machine-compilability, etc.)

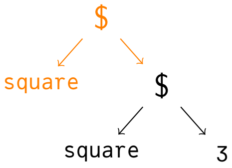
The Second Difficulty: Redirection - 1

```
id x = x
square x = (id x) * x
main = square (square 3)
```



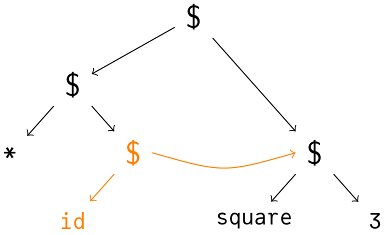
The Second Difficulty: Redirection - 1

```
id x = x
square x = (id x) * x
main = square (square 3)
```



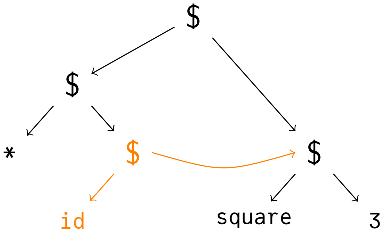
The Second Difficulty: Redirection - 1

```
id x = x
square x = (id x) * x
main = square (square 3)
```



The Second Difficulty: Redirection - 1

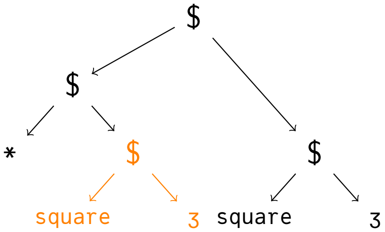
```
id x = x
square x = (id x) * x
main = square (square 3)
```



How should we reduce the application node for `id`?

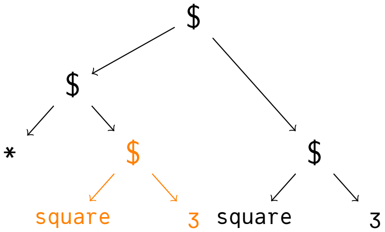
The Second Difficulty: Redirection - 1

```
id x = x
square x = (id x) * x
main = square (square 3)
```



The Second Difficulty: Redirection - 1

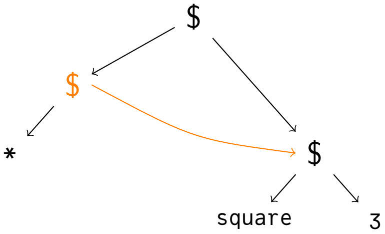
```
id x = x
square x = (id x) * x
main = square (square 3)
```



We lose sharing!

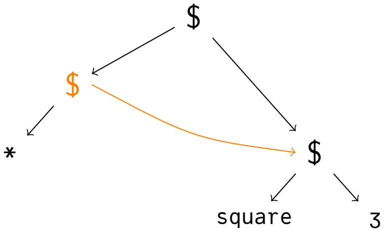
The Second Difficulty: Redirection - 1

```
id x = x
square x = (id x) * x
main = square (square 3)
```



The Second Difficulty: Redirection - 1

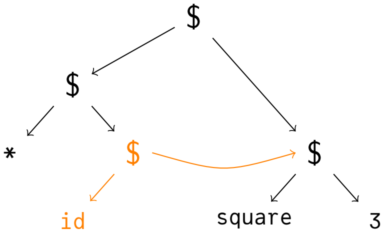
```
id x = x
square x = (id x) * x
main = square (square 3)
```



We need to modify an ancestor (depending on the depth of `id`)

The Second Difficulty: Redirection - 1

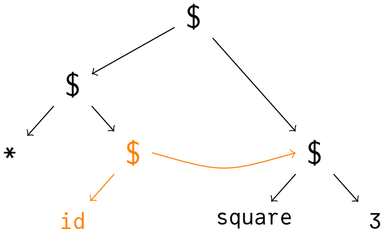
```
id x = x
square x = (id x) * x
main = square (square 3)
```



How should we handle this?

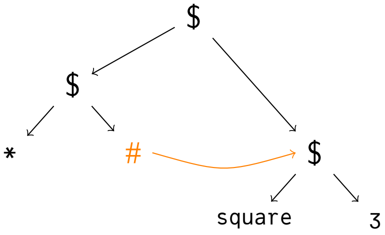
The Second Difficulty: Redirection - 2

```
id x = x
square x = (id x) * x
main = square (square 3)
```



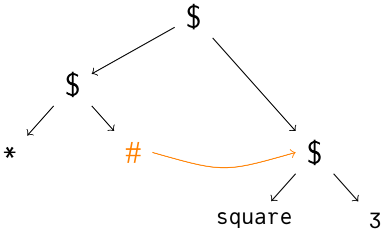
The Second Difficulty: Redirection - 2

```
id x = x
square x = (id x) * x
main = square (square 3)
```



The Second Difficulty: Redirection - 2

```
id x = x
square x = (id x) * x
main = square (square 3)
```



We follow this *redirection* node (#) when we reduce the parent

The Solution for Redirection

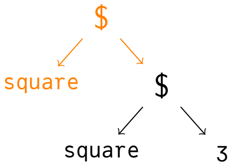
We introduce an “run-time only” node (#) to handle it

Efficiency Requirement - Again

Each graph reduction step should be as *local and small* as possible

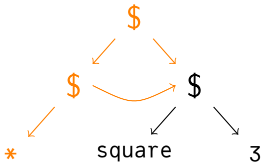
Unfortunate "Change-all" Step

```
square x = x * x
main = square (square 3)
```



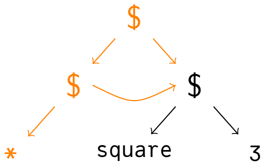
Unfortunate "Change-all" Step

```
square x = x * x
main = square (square 3)
```



Unfortunate "Change-all" Step

```
square x = x * x
main = square (square 3)
```



This changes almost entire structure of graph!

Unfortunate "Change-all" Step - Specifically

When we instantiate a function definition as a part of a graph,
we need to analyze the current graph
and
to construct a new graph

The Main Difficulty: Instantiation

How can we divide this huge step into smaller steps?

The First Solution for Instantiation: G-Machine - 1

Let's start with construction of graph of `main` itself

```
square x = x * x
main = square (square 3)
```

The First Solution for Instantiation: G-Machine - 1

We construct it in a argument-first way, so start at `3`

```
square x = x * x
main = square (square 3)
```

3

The First Solution for Instantiation: G-Machine - 1

and then `square`

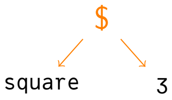
```
square x = x * x
main = square (square 3)
```

square 3

The First Solution for Instantiation: G-Machine - 1

Now we can form an application node

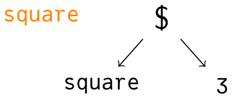
```
square x = x * x
main = square (square 3)
```



The First Solution for Instantiation: G-Machine - 1

Once we add another `square`

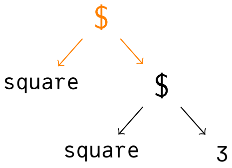
```
square x = x * x
main = square (square 3)
```



The First Solution for Instantiation: G-Machine - 1

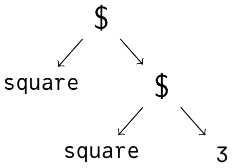
We can finish the main graph by constructing an application node

```
square x = x * x
main = square (square 3)
```



The First Solution for Instantiation: G-Machine - 2

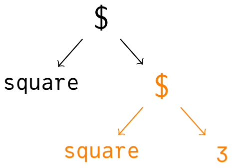
Now let's apply `square`



```
square x = x * x
main = square (square 3)
```

The First Solution for Instantiation: G-Machine - 2

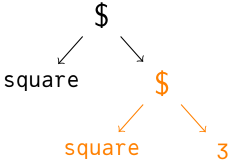
We construct the arugment first



```
square x = x * x
main = square (square 3)
```

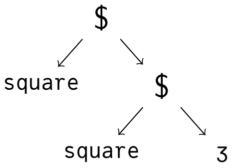
The First Solution for Instantiation: G-Machine - 2

To construct its application node, we need to construct the function node



```
square x = x * x
main = square (square 3)
```

The First Solution for Instantiation: G-Machine - 2

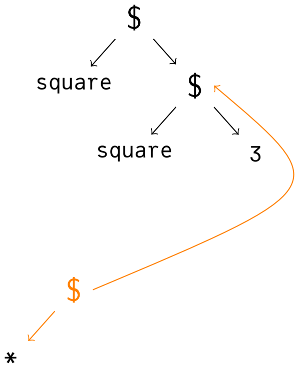


```
square x = x * x
main = square (square 3)
```

*

The First Solution for Instantiation: G-Machine - 2

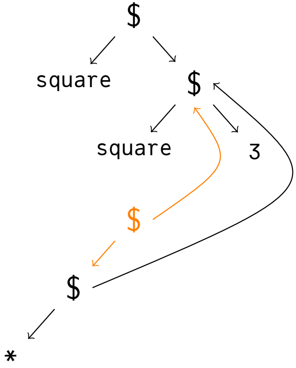
```
square x = x * x
main = square (square 3)
```



The First Solution for Instantiation: G-Machine - 2

and then the top-level application node

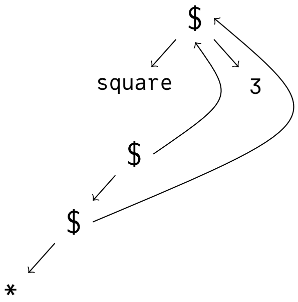
```
square x = x * x
main = square (square 3)
```



The First Solution for Instantiation: G-Machine - 2

Now, clean up the old root

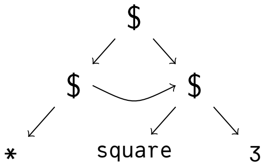
```
square x = x * x
main = square (square 3)
```



The First Solution for Instantiation: G-Machine - 2

After visual rearrange, it is clear that we get the expected graph

```
square x = x * x
main = square (square 3)
```



The First Solution for Instantiation: G-Machine - 3

Can we translate this into recordable code pieces?
Then we can “compile” `main` and `square` into those.

```
square x = x * x
main = square (square 3)
```

The First Solution for Instantiation: G-Machine - 3

Let's repeat the `main` construction first.

```
square x = x * x
main = square (square 3)
```

The First Solution for Instantiation: G-Machine - 3

We construct it in a bottom-up way, so start at `3`

```
PushInt 3
```

```
square x = x * x
main = square (square 3)
```

The First Solution for Instantiation: G-Machine - 3

and then `square`

```
PushInt 3
PushGlobal square
```

```
square x = x * x
main = square (square 3)
```

The First Solution for Instantiation: G-Machine - 3

Now we can form an application node

```
square x = x * x
main = square (square 3)
```

```
PushInt 3
PushGlobal square
MakeApp
```

The First Solution for Instantiation: G-Machine - 3

Once we add another `square`

```
square x = x * x
main = square (square 3)
```

```
PushInt 3
PushGlobal square
MakeApp
PushGlobal square
```

The First Solution for Instantiation: G-Machine - 3

We can finish the main graph by constructing an application node

```
square x = x * x
main = square (square 3)
```

```
PushInt 3
PushGlobal square
MakeApp
PushGlobal square
MakeApp
```

The First Solution for Instantiation: G-Machine - 3

One more step here: we need to continue the graph reduction process

```
square x = x * x
main = square (square 3)
```

```
PushInt 3
PushGlobal square
MakeApp
PushGlobal square
MakeApp
Unwind
```

The First Solution for Instantiation: G-Machine - 4

Now let's compile `square` too

```
square x = x * x
main = square (square 3)
```

The First Solution for Instantiation: G-Machine - 4

We construct the arugment first

```
Push 0
```

```
square x = x * x
main = square (square 3)
```

The First Solution for Instantiation: G-Machine - 4

To construct its application node, we need to construct the function node

```
Push 0
Push 1
```

```
square x = x * x
main = square (square 3)
```

The First Solution for Instantiation: G-Machine - 4

```
square x = x * x
main = square (square 3)
```

```
Push 0

Push 1

PushGlobal *
```

The First Solution for Instantiation: G-Machine - 4

```
square x = x * x
main = square (square 3)
```

```
Push 0
Push 1
PushGlobal *
MakeApp
```

The First Solution for Instantiation: G-Machine - 4

and then the top-level application node

```
square x = x * x
main = square (square 3)
```

```
Push 0
Push 1
PushGlobal *
MakeApp
MakeApp
```

The First Solution for Instantiation: G-Machine - 4

Now, clean up the old root

```
square x = x * x
main = square (square 3)
```

```
Push 0
Push 1
PushGlobal *
MakeApp
MakeApp
Update 2
Pop 2
```

The First Solution for Instantiation: G-Machine - 4

Note that we `Update` the root here to share the work

```
square x = x * x
main = square (square 3)
```

```
Push 0
Push 1
PushGlobal *
MakeApp
MakeApp
Update 2
Pop 2
```

The First Solution for Instantiation: G-Machine - 4

Again, for the further evaluation, we need to add the `Unwind` instruction

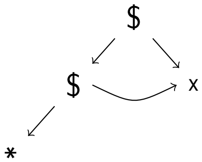
```
square x = x * x
main = square (square 3)
```

```
Push 0
Push 1
PushGlobal *
MakeApp
MakeApp
Update 2
Pop 2
Unwind
```

Problem Solved! Or... Did It? - 1

What does `Unwind` at the end of `square` do?

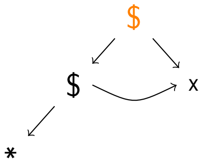
```
square x = x * x
main = square (square 3)
```



Problem Solved! Or... Did It? - 1

It first checks whether the root is a global/primitive value

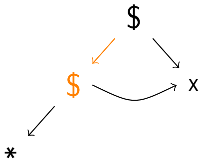
```
square x = x * x
main = square (square 3)
```



Problem Solved! Or... Did It? - 1

Otherwise, it steps into the function side

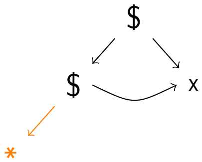
```
square x = x * x
main = square (square 3)
```



Problem Solved! Or... Did It? - 1

Until it reaches a global/primitive value

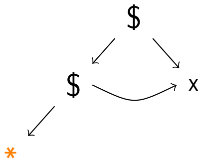
```
square x = x * x
main = square (square 3)
```



Problem Solved! Or... Did It? - 1

And jump to the code for the global

```
square x = x * x
main = square (square 3)
```



Problem Solved! Or... Did It? - 2

`Unwind` has linear time complexity to the length of a “spine” (the curried applications on a function)

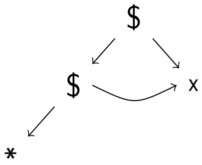
The Second Solution for Instantiation: TIM - 1

To avoid the problem of a spine, we will use “closures” to represent a call

The Second Solution for Instantiation: TIM - 1

To avoid the problem of a spine, we will use “closures” to represent a call

```
square x = x * x
```

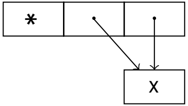


Not this

The Second Solution for Instantiation: TIM - 1

To avoid the problem of a spine, we will use “closures” to represent a call

```
square x = x * x
```



But this

The Second Solution for Instantiation: TIM - 2

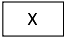
How can we build this?

```
square x = x * x
```

The Second Solution for Instantiation: TIM - 2

First, we take an argument

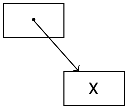
```
square x = x * x
```



The Second Solution for Instantiation: TIM - 2

Then, we put it into stack twice for the `*` call

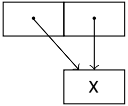
```
square x = x * x
```



The Second Solution for Instantiation: TIM - 2

Then, we put it into stack twice for the $\ast$ call

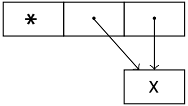
```
square x = x * x
```



The Second Solution for Instantiation: TIM - 2

Now we invoke $*$

`square x = x * x`



The Second Solution for Instantiation: TIM - 3

We can translate this into a code form

```
square x = x * x
```

The Second Solution for Instantiation: TIM - 3

First, we take an argument

Take 1

```
square x = x * x
```

The Second Solution for Instantiation: TIM - 3

Then, we put it into stack twice for the $*$ call

```
square x = x * x
```

```
Take 1
Push (Arg 0)
```

The Second Solution for Instantiation: TIM - 3

Then, we put it into stack twice for the `*` call

```
square x = x * x
```

```
Take 1
Push (Arg 0)
Push (Arg 0)
```

The Second Solution for Instantiation: TIM - 3

Now we invoke $*$

```
square x = x * x
```

```
Take 1
Push (Arg 0)
Push (Arg 0)
Enter (Label *)
```

Now it is really done, right? ...Right?

Note that we do not have anything corresponds to `Update` here.

In fact, only with these 3 instructions, we lose sharing of evaluations!

Now it is really done, right? ...Right?

In fact, `Update` is one of the key issue of TIM.
It is quite costly to have a correct sharing with
TIM due to its machine-level representation details

Now it is really done, right? ...Right?

Can we combine G-machine's simple memory structure (only pointers) and TIM's "closure"-based approach?

Let there be GHC

Let there be GHC

- G. L. Burn, S. L. P. Jones, and J. D. Robson. “The spineless G-machine”. LFP’88

Let there be GHC

- G. L. Burn, S. L. P. Jones, and J. D. Robson. “The spineless G-machine”. LFP’88
- S. L. P. Jones and J. Salkild. “The spineless tagless G-machine”. FPCA’89
- S. L. P. Jones. “Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine”. JFP’92

Let there be GHC

- G. L. Burn, S. L. P. Jones, and J. D. Robson. “The spineless G-machine”. LFP’88
- S. L. P. Jones and J. Salkild. “The spineless tagless G-machine”. FPCA’89
- S. L. P. Jones. “Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine”. JFP’92

- L. Maurer, P. Downen, Z. M. Ariola, and S. L. P. Jones. “Compiling without continuations”. PLDI’17

In summary

- In lazy evaluation, graph reduction is the key to handle sharing
- G-machine solves instantiation inefficiency but is still inefficient in its `Unwind`
- TIM removes some inefficiency of G-machine, but it also adds some for `Update`

- Their combinations can be more efficient...
  For that, see spineless tagless G-machine and "compiling without continuation"