Non-Strict Semantics with Abstract Machines

Andrea Laretto

Università di Pisa
Dipartimento di Informatica

17 febbraio 2021

What is Non-Strictness?

An evaluation strategy where calculations are delayed until needed.
Laziness: if their results are saved and computed only once (sharing)

```
(* Let's try to reimplement a ternary operator in OCaml: *)
let ternary c a b = if c then a else b
(* But does this program terminate? *)
let res = 42
let loop () = loop ()
let result = ternary true res (loop ()) in
print result (* In a lazy language, yes! *)
```

- Fully implemented in pure languages like Lazy ML (1984), Miranda (1985), and Haskell (1990)
- Can be found nowadays in many languages, albeit in different forms: Scheme (`delay` and `force`), OCaml (`lazy` and `Lazy.force`)

Why Laziness?

```haskell
nats :: [Integer]
nats = 0 : map succ nats

fibs :: [Integer]
fibs = 1 : 1 : zipWith (+) fibs (tail fibs)
```

- Advantages:
    - Infinite data structures (streams, sequences)
    - Can be more efficient, avoids unneeded calculations
    - Short-circuiting operations (&&, `?:`) can be implemented as abstractions within the language itself
- Disadvantages:
    - Clashes with mutability and side-effects
    - Non-trivial to implement, poor hardware support
    - Inefficient in many practical cases (overhead)

General Approaches to Laziness

- Closure-based implementations
  - Simple, easy to implement and embed with macros
  - Enables laziness to be explicitly used on-demand
  - The user must explicitly handle delaying and forcing laziness
  - Requires keeping (a part of) activation records on the heap
- Abstract machines
  - Very simple to implement
  - Directly relate to the operational semantics of the language
  - Extensive literature, complexity studied in depth
  - Sometimes inefficient, introduce *execution overhead*
  - Need to be mapped back into real hardware
- Graph reduction
  - Similar to abstract machines, operate on expression graphs
  - Used in Haskell: Spineless Tagless G-machine
  - Easily parallelizable with multiple reductions

What are Closures?

A closure is a pair (f, e) where f is a function, and e is the *environment* storing the *free variables* required to evaluate f.

- A classic example:

```
(* The value of "a" gets captured in the closure: *)
let incrementer a = fun x -> a + x
let incrementer' a x = a + x (* Equivalently, with currying *)
let inc_by_six = incrementer 6
let inc_by_two = incrementer 2
```

- Closures are essential in *lexically scoped languages* that can define and **return** *first-class functions* (upwards funarg problem).

Non-Strictness with Closures: An Example

We can use closures to *delay* evaluation and implement a form of laziness!

```
(* Delayed/wrapped values *)
type 'a delayed = unit -> 'a
(* Unwrap the delayed values by calling the thunk: *)
let ternary c a b = if c () then a () else b ()
let res = 42
let _ = (* Wrap each function argument inside a function   *)
  ternary (fun () -> true) (* Pass the arguments as thunks *)
          (fun () -> res)  (* Value of "res" is captured   *)
          (fun () -> loop ())
```

In order to delay the computation of the arguments, wrap each of them inside a **closure** to create a so-called **"thunk"**. The evaluation thunks will be unwrapped by the callee to perform the actual calculation, *if needed*.

Implementing Closures

- However, a closure might *live longer* than the function that creates it:

```
(* Both "x" and "y" get captured in the final closure: *)
let pair = fun x -> fun y -> fun p -> p x y
let first  p = p (fun x y -> x)
let second p = p (fun x y -> y)
(* Each closure must store and remember its own values: *)
let example1 = pair 3 "hello"
let example2 = pair 7 "world"
```

- The variables captured by the closure might have to be moved from the AR, following the closure around (possibly, from the *stack to the heap*).
- The definition of closure points to a way on how to treat them:
  1. At compile-time, *identify* the variables captured in the closure
  2. At run-time, return a *record* with the entry-point of the function **and** the values of the variables captured (or a reference to access them)

Compiling Closures

- This definition opens up to at least two techniques to compile closures:
  - **Shared closures**: access to variables gets chained through outer environments (traverses the lexical chain in $O(n)$, slow but it saves space), similar to *Access Links*
  - **Flat closures**: the environment keeps a copy/reference of every free variable (fast and requires only $O(1)$ in access, but space expensive)
- Note: allocating closures on the heap *might* require using GC!
- After calling a closure, we can memoize its result and return it when called again ($\rightarrow$ laziness/sharing)
- Peter J. Landin first introduced the term in 1964, later used in his *abstract machine* SECD

A Formal Treatment of Laziness

- If *everything* is lazy, sequencing statements gets complex to manage, the evaluation order of programs is difficult to reconstruct at runtime
- Sequencing techniques to *impose* an evaluation order are required (e.g.: IO monad in Haskell)
- Memoizing/sharing is impossible if re-evaluating the same expression can give a different result!
- In order to formally treat laziness we need a pure calculus free from side-effects, along with a formal setting to treat evaluation orders and non-strictness: λ-**calculus**

λ-calculus

- Foundation of functional programming languages (Haskell, OCaml)
- Core of the FUN language described in the laboratory course
- Extremely simple and easy to define:

$$
\begin{array}{rcll}
<\text{variable}> & = & \text{x, y, z, ...} & \\
<\text{term}> & = & <\text{variable}> & (\textit{variables}) \\
 & | & <\text{term}> <\text{term}> & (\textit{function application}) \\
 & | & \lambda <\text{var}> \rightarrow <\text{term}> & (\textit{anonymous functions})
\end{array}
$$

- Semantics is defined by β-reduction (i.e.: applying functions)

$$(\lambda x \rightarrow b)\ v \Longrightarrow_{\beta} b[x/v]$$

"Substitute each occurrence of x inside b with v"

- This is sufficient for a Turing-complete language free from side-effects
- Establishes a formal setting to describe strict and non-strict evaluation orders

Some Evaluation Strategies in λ-Calculus

$$(\lambda x\ y \to x)\ z\ ((\lambda x \to x\ x)\ k)$$

What are *some* possible ways of evaluating this term?
(i.e.: for each function call, perform β-reduction and substitution)

- Call-by-value: arguments are first fully evaluated (innermost reduction)

$$\begin{aligned} & (\lambda x\ y \to x)\ z\ ((\lambda x \to x\ x) \bullet k) \\ \Longrightarrow_\beta \quad & (\lambda x\ y \to x) \bullet z\ (k\ k) \\ \Longrightarrow_\beta \quad & (\lambda y \to z) \bullet (k\ k) \\ \Longrightarrow_\beta \quad & z \end{aligned}$$

- Call-by-name: arguments are left unevaluated, substituted as-is (outermost)

$$\begin{aligned} & (\lambda x\ y \to x) \bullet z\ ((\lambda x \to x\ x)\ k) \\ \Longrightarrow_\beta \quad & (\lambda y \to z) \bullet ((\lambda x \to x\ x)\ k) \\ \Longrightarrow_\beta \quad & z \end{aligned}$$

(what if we had $k = (\lambda x \to x\ x)$?)

- Call-by-need: functions are evaluated first, *arguments are memoized*

Non-strictness can be implemented with *call-by-name* or *call-by-need*.

Abstract Machines

"An abstract machine is a theoretical step-by-step computer used to define a model of computation."

- Provide an intermediate language stage for compilation
- Explicitly expose evaluation orders and reduction strategies

Sufficiently **abstract** that we do not get tangled up in the very low-level details	Sufficiently **concrete** that we can be sure we are not hiding a lot of complexity in definitions

- A well-studied concept in the early study of functional languages
- Usually defined by the following elements:
  - A minimal instruction set
  - State representation (stack, heap, garbage collection, etc.)
  - An initial state
  - Small-step operational semantics/a state transition relation

Strict Abstract Machines

Most influential strict functional abstract machines:

- **1964, Landin**: SECD machine (Stack, Environment, Control, Dump) for implementing call-by-value
- **1983, Cardelli**: Functional Abstract Machine (FAM), formed the basis of the first native-code implementation of ML
- **1985, Curien et al.**: Categorical Abstract Machine (CAM), derived from Category Theory, used in the CAML implementation of ML
- **1990, Leroy**: Zinc Abstract Machine (ZAM), optimized, strict version of the Krivine machine; foundation for bytecode versions of Leroy's Caml Light and OCaml implementations

Lazy Abstract Machines

Most influential lazy functional abstract machines:

- **1979, Turner**: SK-machine for SASL language, based on SK combinatory logic with two instructions
- **1984, Augustsson et al.**: G-machine for call-by-need evaluation with supercombinators, compiles to sequential code for graph manipulation; became basis of Lazy ML
- **1985, Krivine**: Krivine machine, call-by-name evaluation with three instructions corresponding to the three λ-constructs
- **1986, Fairbairn et al.**: Three Instruction Machine (TIM), evaluation of call-by-name supercombinators
- **1989, Peyton Jones et al.**: Spineless-Tagless G-machine, a refinement of the G-machine, used in the GHC Haskell compiler

The Krivine Machine

- Designed by Jean-Louis Krivine at the beginning of 1980s
- Can be used as a *compilation target* for λ-terms
- Can also be used as an *interpreter* to evaluate λ-terms directly
- Extensible and modular foundation for many other abstract machines
- Implements a *weak head normal form* reduction order (i.e.: call-by-name, but we do not reduce in unapplied λ-abstractions)
- Semantics is defined operationally with a transition relation $\Longrightarrow$:

$$(T, E, S) \Longrightarrow (T', E', S')$$

Operational Semantics of Krivine Machines

- The state of the machine is formalized with these types:

$$
\begin{array}{lcl}
State & = & Term \times Env \times Stack \\
T \in Term & = & \lambda\text{-terms} \\
E \in Env & = & \text{associations from variables to closure pairs } (Term, Env) \\
S \in Stack & = & \text{lists of closure pairs } (Term, Env)
\end{array}
$$

- The evaluation of a term t starts with the state $(t, \{\ \}, [\ ])$, using the empty environment $\{\ \}$ and the empty stack $[\ ]$.

- The transitions of a Krivine machine are defined as follows:

$$
\begin{array}{lllcllr}
(m\ n, & E, & S) & \Longrightarrow & (m, & E, & (n, E) :: S) \\
(\lambda x \to b, & E, & (a, C) :: S) & \Longrightarrow & (b, & E[x \mapsto (a, C)], & S) \\
(x, & E, & S) & \Longrightarrow & (t', & E', & S)
\end{array}
$$

$$\text{where } (t', E') = \text{lookup } E\ x$$

An Intuition for Krivine Machines

- The *stack* corresponds to a list of unevaluated arguments: when we find an application, **push** it in the stack, saving both the argument and the current environment (i.e.: create a closure).

$$(m\ n, E, S) \Longrightarrow (m, E, (n, E) :: S)$$

- The *environment* is a map from variables to closures: when we find an abstraction and the stack is not empty, **pop** the last value from it and associate it to the variable indicated by the abstraction.

$$(\lambda x \rightarrow b, E, (a, C) :: S) \Longrightarrow (b, E[x \mapsto (a, C)], S)$$

- When we encounter variables, we **lookup** the corresponding closure in the environment and start evaluating it: the stack remains unchanged.

$$(x, E, S) \Longrightarrow (t', E', S), \text{ where } (t', E') = \text{lookup } E\ x$$

- Krivine machines implement a **push/enter** evaluation model: arguments are pushed on the stack, and the *callee* retrieves them (*callee* performs β-reduction)

A Concrete Example of Evaluation

$$(\lambda x\ y \to x)\ z\ ((\lambda x \to x\ x)\ k)$$

$$\begin{array}{lll}
& ((\lambda x\ y \to x)\ z\ ((\lambda x \to x\ x)\ k), & \{\ \}, \qquad [\ ]) \\
\Longrightarrow & ((\lambda x\ y \to x)\ z, & \{\ \}, \qquad [\ (((\lambda x \to x\ x)\ k), \{\ \})\ ]) \\
\Longrightarrow & (\lambda x\ y \to x, & \{\ \}, \qquad [\ (z, \{\ \}), (((\lambda x \to x\ x)\ k), \{\ \})\ ]) \\
\Longrightarrow & (\lambda y \to x, & \{\ x \mapsto (z, \{\ \})\ \}, \ [\ (((\lambda x \to x\ x)\ k), \{\ \})\ ]) \\
\Longrightarrow & (x, & \{\ x \mapsto (z, \{\ \}),\ y \mapsto (((\lambda x \to x\ x)\ k), \{\ \})\}, [\ ]) \\
\Longrightarrow & (z, & \{\ \}, \qquad [\ ])
\end{array}$$

The stack is empty and no more transitions can be applied.
The result is the final term z.

A Krivine Machine-Based λ-Interpreter in OCaml

```
type lam
  = Abs of lam
  | App of lam * lam
  | Var of int (* de Bruijn encoding *)

type stack = S of (lam * stack) list

let rec km =
  function
  | (App(a,b), S e, S s     ) -> km (a,  S e,       S ((b, S e)::s))
  | (Abs(t),   S e, S (c::s)) -> km (t,  S (c::e), S s)
  | (Var(n),   S e, S s     ) -> let (t', e') = List.nth e n in
                                 km (t', e',       S s)
  | (t,        S e, S []    ) -> t

let eval t = km (t, S [], S [])
```

Compilation for the Krivine Machine

- We can define a *serialized representation* of λ-terms and treat them as a sequence of executable instructions
- The machine instructions for the Krivine machine are as follows:
  - **Push**(t): save and **push** a closure on the stack for the term t
  - **Grab**(x): extract the first closure with a stack **pop** and pair it with the variable x in the environment
  - **Access**(v): perform a **lookup** "jump" to the closure indicated by the variable v, and restart evaluation
- Define a **compilation function** $[\![\cdot]\!]$: Term $\rightarrow$ [Instr], as follows:

$$
\begin{aligned}
[\![m\ n]\!] &= \mathrm{Push}([\![n]\!])\ ;\ [\![m]\!] \\
[\![\lambda x \rightarrow b]\!] &= \mathrm{Grab(x)}\ ;\ [\![b]\!] \\
[\![x]\!] &= \mathrm{Access}(x)
\end{aligned}
$$

- Inside machine instructions we use lightweight *code pointers* that refer to the beginning of other serialized terms

A Concrete Example of Compilation

$$
\begin{aligned}
& [\![(\lambda x\; y \to x)\; z\; ((\lambda x \to x\; x)\;\; k)]\!] \\
= \;& \mathsf{Push}([\![(\lambda x \to x\; x)\;\; k]\!])\; ;\; [\![(\lambda x\; y \to x)\;\; z]\!] \\
= \;& \mathsf{Push}([\![(\lambda x \to x\; x)\;\; k]\!])\; ;\; \mathsf{Push}([\![z]\!])\; ;\; [\![\lambda x\; y \to x]\!] \\
= \;& \mathsf{Push}([\![(\lambda x \to x\; x)\;\; k]\!])\; ;\; \mathsf{Push}([\![z]\!])\; ;\; \mathsf{Grab}(x)\; ;\; [\![\lambda y \to x]\!] \\
= \;& \mathsf{Push}([\![(\lambda x \to x\; x)\;\; k]\!])\; ;\; \mathsf{Push}([\![z]\!])\; ;\; \mathsf{Grab}(x)\; ;\; \mathsf{Grab}(y)\; ;\; [\![x]\!] \\
= \;& \mathsf{Push}([\![(\lambda x \to x\; x)\;\; k]\!])\; ;\; \mathsf{Push}([\![z]\!])\; ;\; \mathsf{Grab}(x)\; ;\; \mathsf{Grab}(y)\; ;\; \mathsf{Access}(x) \\
= \;& \mathsf{Push}(5)\; ;\; \mathsf{Push}([\![z]\!])\; ;\; \mathsf{Grab}(x)\; ;\; \mathsf{Grab}(y)\; ;\; \mathsf{Access}(x)\; ;\; [\![(\lambda x \to x\; x)\;\; k]\!] \\
= \;& \mathsf{Push}(5)\; ;\; \mathsf{Push}([\![z]\!])\; ;\; \mathsf{Grab}(x)\; ;\; \mathsf{Grab}(y)\; ;\; \mathsf{Access}(x)\; ;\; \mathsf{Push}([\![k]\!])\; ;\; [\![\lambda x \to x\; x]\!] \\
= \;& \mathsf{Push}(5)\; ;\; \mathsf{Push}([\![z]\!])\; ;\; \mathsf{Grab}(x)\; ;\; \mathsf{Grab}(y)\; ;\; \mathsf{Access}(x)\; ;\; \mathsf{Push}([\![k]\!])\; ;\; \mathsf{Grab}(x)\; ;\; [\![x\; x]\!] \\
= \;& \mathsf{Push}(5)\; ;\; \mathsf{Push}([\![z]\!])\; ;\; \mathsf{Grab}(x)\; ;\; \mathsf{Grab}(y)\; ;\; \mathsf{Access}(x)\; ;\; \mathsf{Push}([\![k]\!])\; ;\; \mathsf{Grab}(x)\; ; \\
& \mathsf{Push}([\![x]\!])\; ;\; [\![x]\!] \\
= \;& \mathsf{Push}(5)\; ;\; \mathsf{Push}([\![z]\!])\; ;\; \mathsf{Grab}(x)\; ;\; \mathsf{Grab}(y)\; ;\; \mathsf{Access}(x)\; ;\; \mathsf{Push}([\![k]\!])\; ;\; \mathsf{Grab}(x)\; ; \\
& \mathsf{Push}([\![x]\!])\; ;\; \mathsf{Access}(x) \\
= \;& \mathsf{Push}(5)\; ;\; \mathsf{Push}(9)\; ;\; \mathsf{Grab}(x)\; ;\; \mathsf{Grab}(y)\; ;\; \mathsf{Access}(x)\; ;\; \mathsf{Push}(10)\; ;\; \mathsf{Grab}(x)\; ; \\
& \mathsf{Push}(11)\; ;\; \mathsf{Access}(x)\; ;\; \mathsf{Access}(z)\; ;\; \mathsf{Access}(k)\; ;\; \mathsf{Access}(x)
\end{aligned}
$$

Call-by-need vs. Call-by-value

- As they have been defined, Krivine machines are somewhat inefficient: they might still recalculate the same variable many times
- This is the intrinsic difference between *call-by-name* and *call-by-need*:

```
let square x = x * x in     (* In an hypothetical   . *)
    square (3 + 4)          (* call-by-need language: *)
 => (3 + 4) * (3 + 4)
 => 7 * (3 + 4)
 => 7 * 7
 => 49
```

- In λ-calculus, we have the same setting:

$$
\begin{aligned}
& (\lambda x \to f\,(x\ x))\ (\text{... complex ...}) \\
\Longrightarrow_\beta\ & f\,(\text{... complex ...})\ (\text{... complex ...}) \\
\Longrightarrow_\beta^*\ & f\ \mathit{result}\ (\text{... complex ...}) \\
\Longrightarrow_\beta^*\ & f\ \mathit{result}\ \mathit{result}
\end{aligned}
$$

- Ideally, each argument should be evaluated once, if at all, and then reused later if required again (*sharing/memoization*)

An Extension: Lazy Krivine Machines

- Add a new element to the State: a **heap** to memoize computed values
- The idea is to use heap *Locations* to add an extra level of indirection for terms in the environment
- When a term finishes its evaluation, we can physically *replace* its variable on the heap with the final result
- We can use a *mark* on the stack to indicate when (and where on the heap) we can update the variable with the same value
- The definition of the lazy Krivine machine is as follows:

$$
\begin{array}{lcl}
\mathit{State} & = & \mathit{Term} \times \mathit{Env} \times \mathit{Stack} \times \mathit{Heap} \\
\mathit{Loc} & = & \text{abstract heap locations} \\
T \in \mathit{Term} & = & \lambda\text{-terms} \\
E \in \mathit{Env} & = & \text{associations from variables to heap locations } \mathit{Loc} \\
S \in \mathit{Stack} & = & \text{lists of either locations Mark}(\mathit{Loc}) \text{ or pairs Arg}(\mathit{Term}, \mathit{Env}) \\
H \in \mathit{Heap} & = & \text{associations from locations } \mathit{Loc} \text{ to closures } (\mathit{Term}, \mathit{Env})
\end{array}
$$

Operational Semantics of Lazy Krivine Machines

$$
\begin{array}{rlllll}
 & (\; m \; n, & E, & S, & H &) \\
\Longrightarrow & (\; m, & E, & \mathrm{Arg}(n, E) :: S, & H &) \\[1em]
 & (\; \lambda x \to b, & E, & \mathrm{Arg}(a, C) :: S, & H &) \\
\Longrightarrow & (\; b, & E[x \mapsto \ell], & S, & H[\ell \mapsto (a, C)] &) \\[1em]
 & (\; \lambda x \to b, & E, & \mathrm{Mark}(\ell) :: S, & H &) \\
\Longrightarrow & (\; \lambda x \to b, & E, & S, & H[\ell \mapsto (\lambda x \to b, E)] &) \\[1em]
 & (\; x, & E, & S, & H &) \\
\Longrightarrow & (\; t', & E', & \mathrm{Mark}(\ell) :: S, & H &)
\end{array}
$$

$$
\text{where} \quad \begin{array}{rcl} \ell & = & \mathrm{lookup}\ E\ x \\ (t', E') & = & \mathrm{deref}\ H\ \ell \end{array}
$$

Compilation for Lazy Krivine Machines

- Extend the compilation function $[\![\cdot]\!] : \text{Term} \to [\text{Instr}]$ as follows:

$$
\begin{array}{lcl}
[\![m\ n]\!] & = & \text{Push}([\![n]\!])\ ;\ [\![m]\!] \\
[\![\lambda x \to b]\!] & = & \text{PopMark}\ ;\ \text{Grab(x)}\ ;\ [\![b]\!] \\
[\![x]\!] & = & \text{PushMark}\ ;\ \text{Access}(x)
\end{array}
$$

1. PushMark adds a new mark to the stack using a fresh heap location
2. PopMark updates the heap at the location given on the stack, when the given term is in weak head normal form

- This compilation schema, however, updates the closure everytime that a variable is accessed (*caller-update*)
- Ideally, we would like closures to update themselves with their value on the stack after evaluation only *once*
- However, we have to push the mark many times because we cannot tell if a term has already been evaluated: introduce a flag!

Lazy Krivine Machines with Callee-Update

- Slightly more complicated definitions on the machine state:

$$\begin{array}{lcl} \mathit{State} & = & \mathit{Term} \times \mathit{Env} \times \mathit{Stack} \times \mathit{Heap} \\ T \in \mathit{Term} & = & \lambda\text{-terms} \\ E \in \mathit{Env} & = & \text{associations from variables to heap locations } \mathit{Loc} \\ S \in \mathit{Stack} & = & \text{lists of either Mark}(\mathit{Loc}) \text{ or Arg}(\mathit{Loc}) \text{ locations} \\ H \in \mathit{Heap} & = & \text{associations from locations } \mathit{Loc} \text{ to either} \\ & & \text{Delayed}(\mathit{Term}, \mathit{Env}) \text{ or Computed}(\mathit{Term}, \mathit{Env}) \text{ closures} \end{array}$$

- The heap now keeps track of whether arguments have already been evaluated or not, avoiding useless updates
- The stack does not keep closures, marks are still required to know *when* updates have to be performed (i.e.: arguments not yet evaluated)

Callee-Update Operational Semantics

$$\begin{array}{lllll} & (\; m\ n, & E, & S, & H \;) \\ \Longrightarrow & (\; m, & E, & \mathrm{Arg}(\ell, E) :: S, & H[\ell \mapsto \mathrm{Delayed}(n, E)] \;) \end{array}$$

$$\begin{array}{lllll} & (\; \lambda x \to b, & E, & \mathrm{Arg}(\ell) :: S, & H \;) \\ \Longrightarrow & (\; b, & E[x \mapsto \ell], & S, & H \;) \end{array}$$

$$\begin{array}{lllll} & (\; \lambda x \to b, & E, & \mathrm{Mark}(\ell) :: S, & H \;) \\ \Longrightarrow & (\; \lambda x \to b, & E, & S, & H[\ell \mapsto \mathrm{Computed}(\lambda x \to b, E)] \;) \end{array}$$

$$\frac{\text{if } \mathrm{Delayed}(t', E') = \text{deref } E\ \ell\,,\ \text{ with } \ell = \text{lookup } E\ x}{(\; x,\ E,\ S,\ H \;) \Longrightarrow (\; t',\ E',\ \mathrm{Mark}(\ell) :: S,\ H \;)}$$

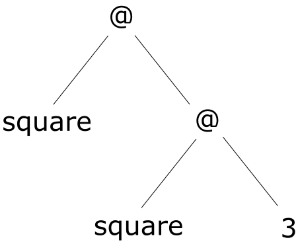
$$\frac{\text{if } \mathrm{Computed}(v, E') = \text{deref } E\ \ell\,,\ \text{ with } \ell = \text{lookup } E\ x}{(\; x,\ E,\ S,\ H \;) \Longrightarrow (\; v,\ E',\ S,\ H \;)}$$

Graph Reduction

- Another approach to laziness: represent the expression tree as a *graph*, moving *pointers* to implement *sharing* of unevaluated expressions
- Can be implemented using abstract machines, the first being the G-machine (Johnsson and Augustsson, 1984)
- Most relevant variation: the Spineless Tagless G-machine (Peyton Jones, 1989), used in GHC Haskell
- The main non-strictness idea: only operate on the *leftmost outermost* possible reducible expression (i.e.: the function closest to the root of the graph)

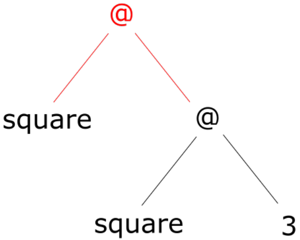
Graph Reduction: An Example

```
let square x = x * x in
    square (square 3)
```



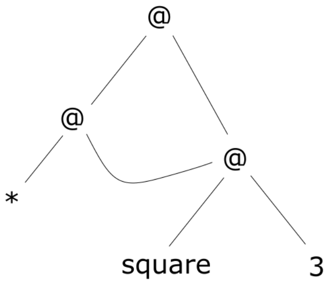
Graph Reduction: An Example

```
let square x = x * x in
    square (square 3)
```



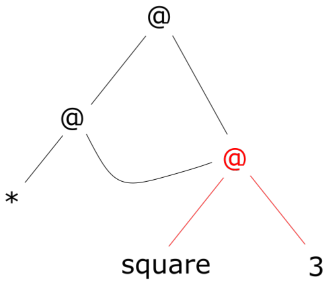
Graph Reduction: An Example

```
let square x = x * x in
    square (square 3)
```



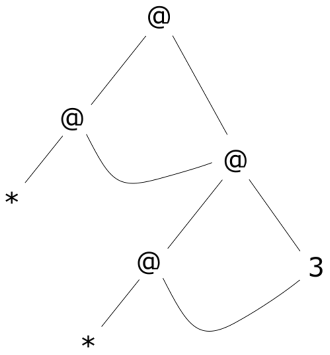
Graph Reduction: An Example

```
let square x = x * x in
    square (square 3)
```



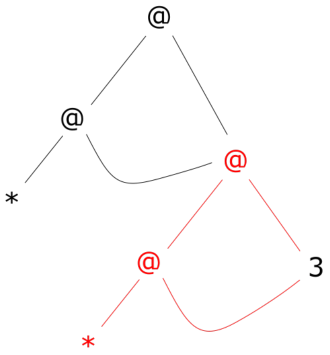
Graph Reduction: An Example

```
let square x = x * x in
    square (square 3)
```



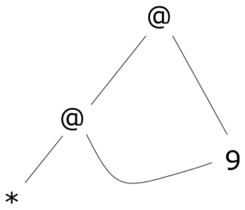
Graph Reduction: An Example

```
let square x = x * x in
    square (square 3)
```



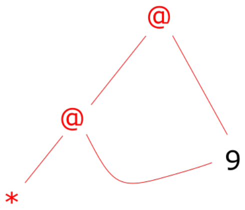
Graph Reduction: An Example

```
let square x = x * x in
    square (square 3)
```



Graph Reduction: An Example

```
let square x = x * x in
    square (square 3)
```



Graph Reduction: An Example

```
let square x = x * x in
    square (square 3)
```

81

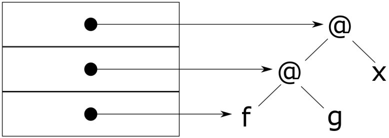
- Graph reduction steps are, in concrete, expressed as instructions for a suitable abstract machine!

Compiling Graph Reduction with Abstract Machines

- A simplified version of the G-machine (Augustsson et al. 1984)
- The idea: use a **stack** to point to the left-branching chain of application nodes (so-called *spine*) in order to find the leftmost outermost reducible expression.
- Instructions operate on the stack, building the expression graph and performing reductions using pointers:
  - **Push**(n): put a copy of the n-th element on the top of the stack
  - **Mkap**: pop two elements and create an application node
  - **Slide**(n): pop $n+1$ elements from the stack, *except for the top-most*
  - **Pushglobal**(t): simply push a built-in operator
  - **Unwind**: end the execution of the current function, find again the outermost reducible expression
- The G-machine uses *pointers* to move the code graph, thus avoiding unneeded evaluation and implement sharing
- (Technical details: the program must be a list of top-level closed definitions, called *supercombinators*; another stack used for Unwind)

Graph Reduction with the G-Machine: An Example

```
let f a b = K (a b) in f g x
```

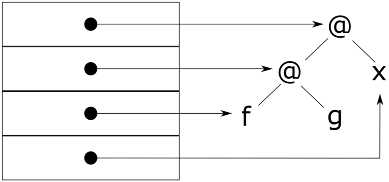


Compiled G-machine code for `f`:

Push 2
Push 2
Mkap
Pushglobal K
Mkap
Slide 3
Unwind

Graph Reduction with the G-Machine: An Example

```
let f a b = K (a b) in f g x
```

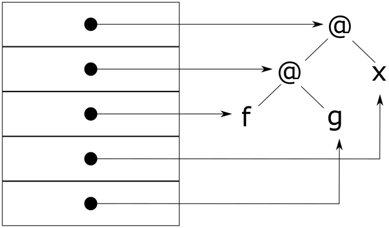


Compiled G-machine code for `f`:

Push 2
Push 2
Mkap
Pushglobal K
Mkap
Slide 3
Unwind

Graph Reduction with the G-Machine: An Example

```
let f a b = K (a b) in f g x
```

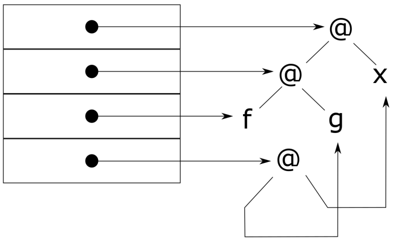


Compiled G-machine code for `f`:

Push 2
Push 2
Mkap
Pushglobal K
Mkap
Slide 3
Unwind

Graph Reduction with the G-Machine: An Example

```
let f a b = K (a b) in f g x
```

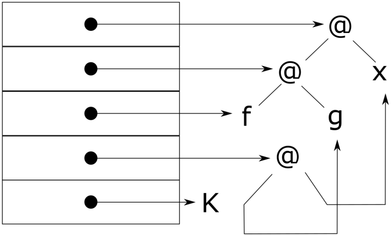


Compiled G-machine code for `f`:

Push 2
Push 2
Mkap
Pushglobal K
Mkap
Slide 3
Unwind

Graph Reduction with the G-Machine: An Example

```
let f a b = K (a b) in f g x
```

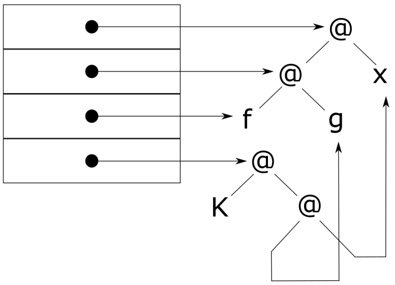


Compiled G-machine code for `f`:

Push 2
Push 2
Mkap
Pushglobal K
Mkap
Slide 3
Unwind

Graph Reduction with the G-Machine: An Example

```
let f a b = K (a b) in f g x
```

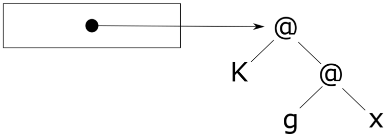


Compiled G-machine code for `f`:

Push 2
Push 2
Mkap
Pushglobal K
Mkap
Slide 3
Unwind

Graph Reduction with the G-Machine: An Example

```
let f a b = K (a b) in f g x
```

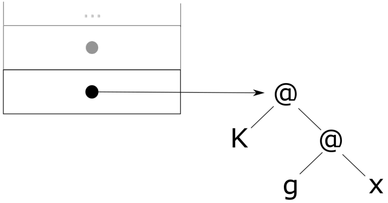


Compiled G-machine code for `f`:

Push 2
Push 2
Mkap
Pushglobal K
Mkap
Slide 3
Unwind

Graph Reduction with the G-Machine: An Example

```
let f a b = K (a b) in f g x
```



Compiled G-machine code for `f`:

Push 2
Push 2
Mkap
Pushglobal K
Mkap
Slide 3
Unwind

Conclusion and Related Topics

- Non-strict and lazy semantics open up to many implementation and analysis possibilities: demand analysis, **strictness analysis** using *abstract interpretation*, etc.
- Many practical and theoretical approaches: how and when to efficiently compile closures, when avoiding laziness can be convenient, parallelization, etc.
- Abstract machines allow for both theoretical and concrete explorations of implementation techniques
- Abstract machines more recently: mechanically deriving and synthesizing abstract machines, proving their correctness and other useful properties

Thank you for your attention!

Bibliography I

Jean-Louis Krivine.
A call-by-name lambda-calculus machine.
High. Order Symb. Comput., 20(3):199–207, 2007.

Stephan Diehl, Pieter H. Hartel, and Peter Sestoft.
Abstract machines for programming language implementation.
Future Gener. Comput. Syst., 16(7):739–751, 2000.

Simon L. Peyton Jones.
Implementing lazy functional languages on stock hardware: The spineless tagless G-machine.
J. Funct. Program., 2(2):127–202, 1992.

Rémi Douence and Pascal Fradet.
The next 700 krivine machines.
High. Order Symb. Comput., 20(3):237–255, 2007.

Bibliography II

Rémi Douence and Pascal Fradet.
A systematic study of functional language implementations.
ACM Trans. Program. Lang. Syst., 20(2):344–387, 1998.

Peter Sestoft.
Deriving a lazy abstract machine.
J. Funct. Program., 7(3):231–264, 1997.

Daniel P. Friedman, Abdulaziz Ghuloum, Jeremy G. Siek, and Onnie Lynn Winebarger.
Improving the lazy krivine machine.
High. Order Symb. Comput., 20(3):271–293, 2007.

Simon L. Peyton Jones.
The Implementation of Functional Programming Languages.
Prentice-Hall, 1987.