Reverse your Abstract Machine!

How can we use λ-calculus to create a baseline for reversible programing at a low level using abstract machines?

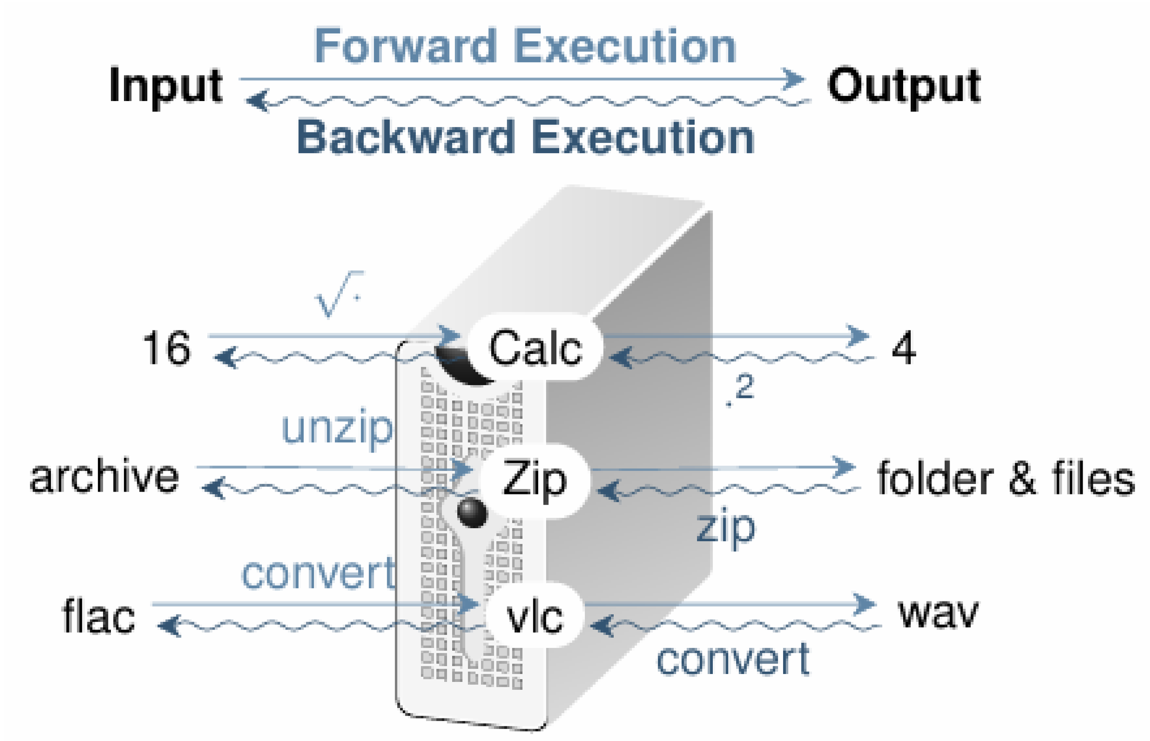
Dr. Clément Aubert, Nate Schwartz

Augusta University

Summer Scholars Program — Center for Undergraduate Research and Scholarship — Innovate · Investigate · Discover

Why Use Reversibility?

- All computation today is nonreversible; given an output, we cannot return the input without an inverse function
- Reversibility is a machine's ability to revert to any previous state regardless of current position
- Such a machine saves time and energy retracing its steps rather than running more code to revert an action



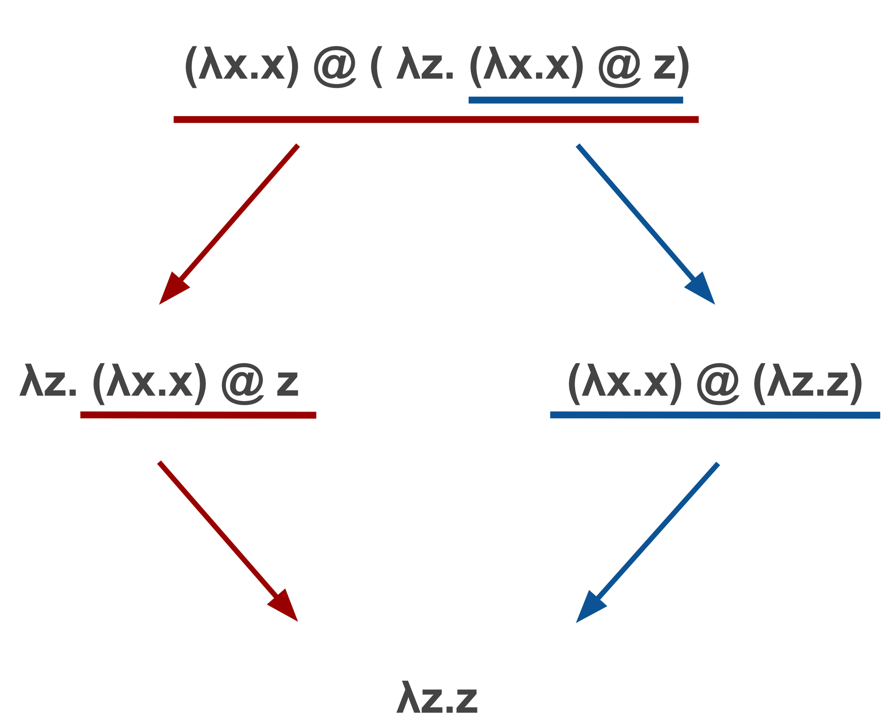
Introduction to λ-Calculus

- λ-Calculus is the lowest-level functional programming language. By understanding it, we can influence the efficiency of higher-level languages
- λ-Calculus is only made up of three terms: variables, applications and abstractions.
- Using β-Reduction we can reduce the expression λx.t @ u into t{u/x}, where all instances of x in t are replaced with u.

t, s, u ≔	terms:
x	variable
λx.t	abstraction
t @ s	application

β-Reduction:

$$(\lambda x.t) @ u \rightarrow_\beta t\{u/x\}$$



- Using β-reduction to reduce a 'redex' is simple, however finding one can be tricky. Some expressions (see above) have multiple redexes and it may not be clear which to evaluate: a strategy is needed
- In order to find redexes to reduce, we can either use a predetermined strategy, or preform all choices at once in a tree diagram

Full Beta Reduction:	Normal Order Strategy:	Call by Name Strategy:
Reduce any redex disered	Reduce outermost redex first	No reductions inside abstractions
(λx.x) @ (λz. (λx.x) @ z)	(λx.x) @ (λz. (λx.x) @ z)	λz. (λx.x) @ z *None!*

Fig. 5 examples of strategies to find redexes

Abstract Machines

- Abstract machines are lists of steps and conditions that, exactly like computers, run programs to reduce terms
- If a machine has a particular way of reducing an expression, it's called deterministic. If it has no method, and takes every possible path, it is non-deterministic
- Expressions are often applied and reduced to a context $\mathbb{E}$. When a step is made, a part of an application/abstraction (or frame) is added to the expression's context as a history of steps done

$$\langle u \rangle_{\mathsf{zs}} \mapsto \langle u \mid \bullet \rangle_{\mathsf{app}}$$

$\langle u @^{\Sigma} s \mid \mathbb{E} \rangle_{\mathsf{app}} \mapsto \langle u \mid @^{\Sigma} s :: \mathbb{E} \rangle_{\mathsf{app}}$	if $\mathsf{app} \notin \mathsf{an}(u)$
$\langle u @^{\Sigma} s \mid \mathbb{E} \rangle_{\mathsf{app}} \mapsto \langle s \mid u @^{\Sigma} :: \mathbb{E} \rangle_{\mathsf{app}}$	if $\mathsf{app} \notin \mathsf{an}(s)$
$\langle u @^{\Sigma} s \mid \mathbb{E} \rangle_{\mathsf{app}} \mapsto \langle u \mid @^{\Sigma} s :: \mathbb{E} \rangle_{\mathsf{lam}}$	if $\mathsf{lam} \notin \mathsf{an}(u)$
$\langle \lambda^{\Sigma} x.u \mid \mathbb{E} \rangle_{\mathsf{app}} \mapsto \langle u \mid \lambda^{\Sigma} x :: \mathbb{E} \rangle_{\mathsf{app}}$	if $\mathsf{app} \notin \mathsf{an}(u)$
$\langle u \mid \mathfrak{F} :: \mathbb{E} \rangle_{\mathsf{app}} \mapsto \langle \mathfrak{F}[u^{\cup \mathsf{app}}] \mid \mathbb{E} \rangle_{\mathsf{app}}$	otherwise
$\langle u \mid \bullet \rangle_{\mathsf{app}} \mapsto \langle u \rangle_{\mathsf{nf}}$	otherwise
$\langle \lambda^{\Sigma} x.u \mid @^{\Sigma'} s :: \mathbb{E} \rangle_{\mathsf{lam}} \mapsto \langle \lvert \mathbb{E}[u\{s/x\}] \rvert \rangle_{\mathsf{zs}}$	
$\langle u \mid \mathfrak{F} :: \mathbb{E} \rangle_{\mathsf{lam}} \mapsto \langle \mathfrak{F}[u^{\cup \mathsf{lam}}] \mid \mathbb{E} \rangle_{\mathsf{app}}$	otherwise

Fig. 6 Our current Non-Deterministic Abstract Machine (NDAM)

- If the machine ever makes a mistake, it backtracks by adding the last used frame back into the expression, restoring its previous state
- To know not to make the same mistake, the machine puts an annotation on a term wrongly reduced

$\langle x@y \rangle_{zs} \mapsto \langle x@y \mid \bullet \rangle_{app}$	**Initial**
$\langle x@y \mid \bullet \rangle_{app} \mapsto \langle x \mid @y :: \bullet \rangle_{app}$	**By 2**
$\langle x \mid @y :: \bullet \rangle_{app} \mapsto \langle x^{\cup\, app}@y \mid \bullet \rangle_{app}$	**By 6**
$\langle x^{\cup\, app}@y \mid \bullet \rangle_{app} \mapsto \langle y \mid x^{\cup\, app}@ :: \bullet \rangle_{app}$	**By 3**
$\langle y \mid x^{\cup\, app}@ :: \bullet \rangle_{app} \mapsto \langle x^{\cup\, app}@y^{\cup\, app} \mid \bullet \rangle_{app}$	**By 6**
$\langle x^{\cup\, app}@y^{\cup\, app} \mid \bullet \rangle_{app} \mapsto \langle x^{\cup\, app} \mid @y^{\cup\, app} :: \bullet \rangle_{lam}$	**By 4**
$\langle x^{\cup\, app} \mid @y^{\cup\, app} :: \bullet \rangle_{lam} \mapsto \langle x^{\cup\, app\, \cup\, lam}@y^{\cup\, app} \mid \bullet \rangle_{app}$	**By 9**
$\langle x^{\cup\, app\, \cup\, lam}@y^{\cup\, app} \mid \bullet \rangle_{app} \mapsto \langle x^{\cup\, app\, \cup\, lam}@y^{\cup\, app} \rangle_{nf}$	**Forced/Final**

Fig. 7 One way the above NDAM could evaluate the λ-expression x@y

Conclusion and Proving Equivalency

- In the paper Non-Deterministic Abstract Machines by Malgorzata Biernacka et al (2022), another abstract machine for λ-calculus was developed
- Their machine was not made to be reversible, but it had features to make it easy to implement
- We are proving our simpler machine has equal structure to theirs, so we can easily add reversibility and create a baseline for reversible programming
- We are developing the scaffolding of reversible programming; these will become the tools for other researchers to construct more advanced reversible languages