May 8th, 11:00 AM - 1:00 PM

# Story of Your Lazy Function’s Life: A Bidirectional Demand Semantics for Mechanized Cost Analysis of Lazy Programs

Laura Israel
*Portland State University*

Nicholas Coltharp
*Portland State University*

# Story of Your Lazy Function's Life: A Bidirectional Demand Semantics for Mechanized Cost Analysis of Lazy Programs

Li-yao Xia, Laura Israel[A], Maite Kramarz[B], Nicholas Coltharp[A], Koen Claessen[C], Stephanie Weirich[D], Yao Li[A]

A: Portland State University, B: McGill University, C: Chalmers University of Technology, D: University of Pennsylvania

## What is Lazy Evaluation?

- Normally, we compute values "strictly."
  - If I write "`x = 1 + 2`," then my computer calculates "`1 + 2`" immediately and records "`x = 3`."
- In lazy evaluation, we get to put off work until later!
  - If I write "`x = 1 + 2`," my computer records the fact that "`x = 1 + 2`." It doesn't actually add "`1 + 2`" until it needs the value of "`x`" for something.

## Why is Lazy Evaluation good?

- Lazy evaluation can **save us time**.
- Lazy evaluation lets us **compose functions more easily**.
- Lazy evaluation lets us use **infinite data** with less risk of an infinite loop.

## Why is Lazy Evaluation hard?

- The order of operations gets confusing.
  - When does the 4 get popped?

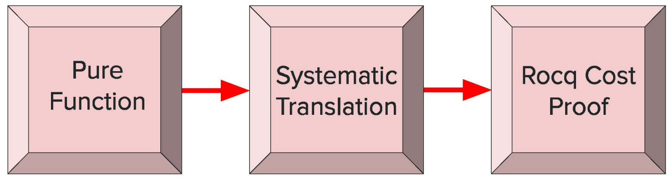
```
queue = [1, 2, 3]
push 4 queue
//Pop 4 elements from the queue
newQueue = pop4 queue
print(newQueue)
```

- How do we know what will be evaluated?
  - When we write "`x = 1 + 2`", we usually think of that as meaning that x will definitely be set to "`1 + 2`". But if we never use x, this won't happen.
  - Does this code run forever?

```
// "[1..]" is an infinite list
xs = [1..]
foo = reverse xs
```

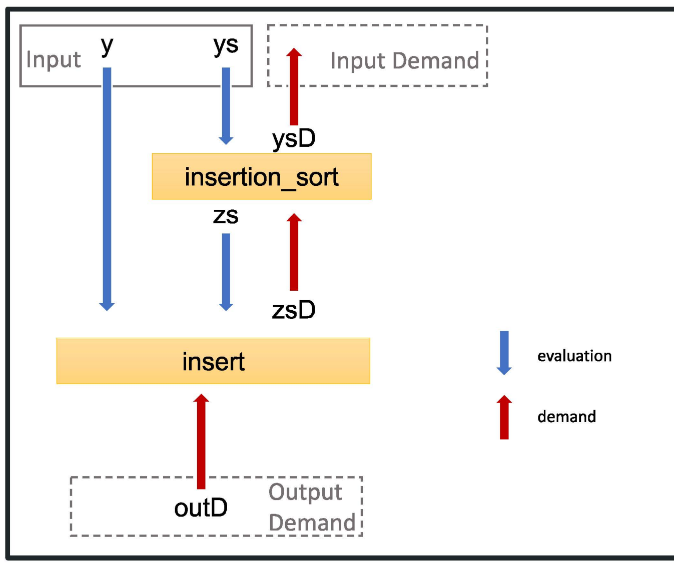
- **Function costs depend on future demand.**

**Read our pre-print!**




### Insertion Sort Cost

```
Theorem take_insertion_sortD_cost (n :
  nat) (xs : list nat) (outD : listA nat)
  : Tick.cost (take_insertion_sortD n xs
  outD) <= (n + 1) * (length xs + 2) + 1.
```




## Our Contributions

- A novel **semantics** for reasoning about the computational cost of laziness
- Used **Coq** to prove **time bounds, with amortization and persistence** for some of Okasaki's data structures.
  - Banker's Queue
  - Implicit Queue (partial)
- Developed the **reverse physicist's method** for reasoning about amortization

## Example: Insertion Sort

- Insertion sort is a common sorting algorithm. It is typically $O(n^2)$ time, but with laziness, it can take $O(n)$ time for certain operations.
- **Insertion sort definition** (Gallina):

```
Fixpoint insertion_sort (xs : list
nat) : list nat :=
  match xs with
  | nil => nil
  | y :: ys =>
    let zs := insertion_sort ys in
      insert y zs
  end.
```

- **Insertion sort's demand semantics version** (Gallina):

```
Fixpoint insertion_sortD (xs: list
nat) (outD : listA nat) :
Tick (T (listA nat)) :=
  tick >> match xs with
  | [] => ret (Thunk NilA)
  | y :: ys => let zs :=
      insertion_sort ys in
        let+ zsD := insertD y zs outD
        in
          let+ ysD := thunkD
          (insertion_sortD ys) zsD in
            ret (Thunk (ConsA (Thunk
             y) ysD))
```