Lambdachine:
A Virtual Machine for Haskell

Thomas Schilling
Erudify

FP Afternoon 2013, Zürich, Switzerland

Introduction

- Haskell is a nice language and some very nice tools (GHCi, ThreadScope, criterion), but could be better.
- Profiling requires recompiling your program and all the libraries it depends on!
- Libraries are distributed in source form and compilation can be quite slow.
- Solution: virtual machine
  - Bonus: Send code over network?
  - Bonus: JIT compiler may even provide better performance

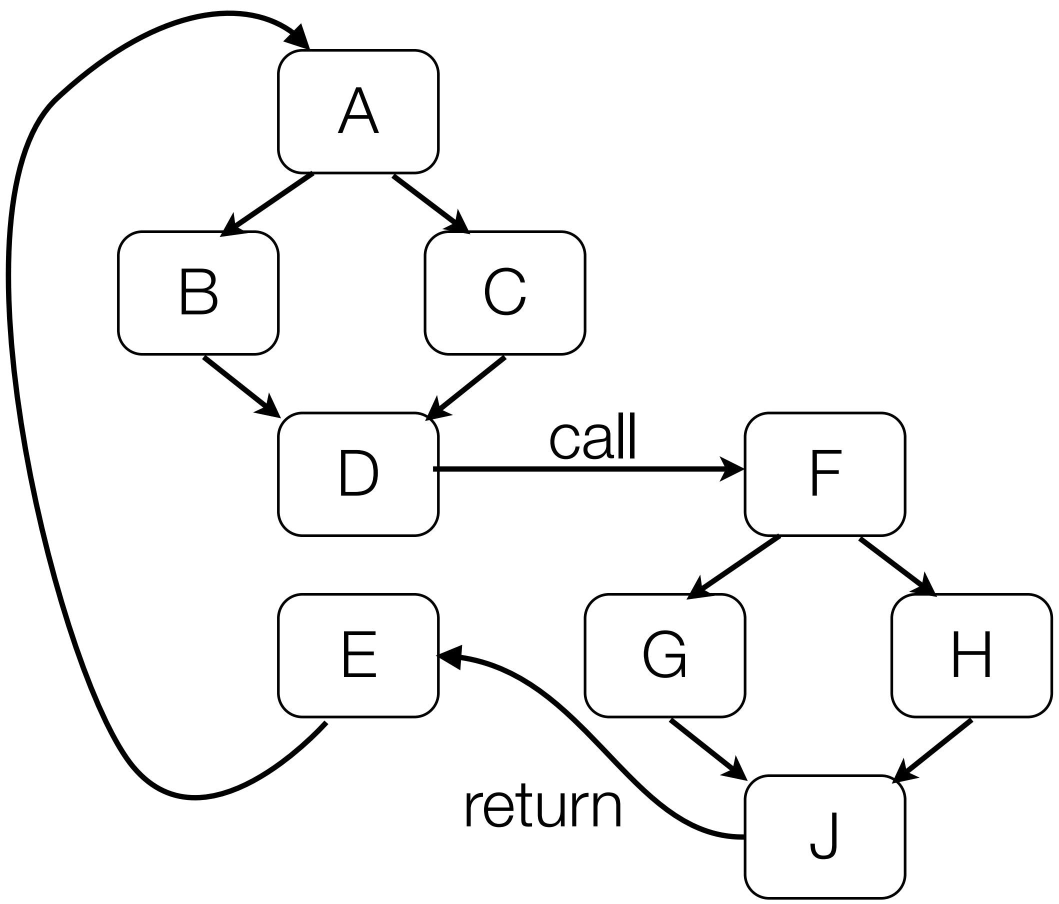
Option 1: Use JVM / CLR

- Pro: 100s to 1000s of person years of implementation effort
- Pro: Battle-tested and available on many platforms (at least in some form)
- Pro: Interface with huge number of existing libraries
- Con: Designed with different usage pattern in mind -- Haskell is very different in a number of aspects. GHC's runtime system takes advantage of Haskell's idiosyncrasies in a number of ways.
- Con: Interfacing with existing language requires mapping of types which is often awkward.

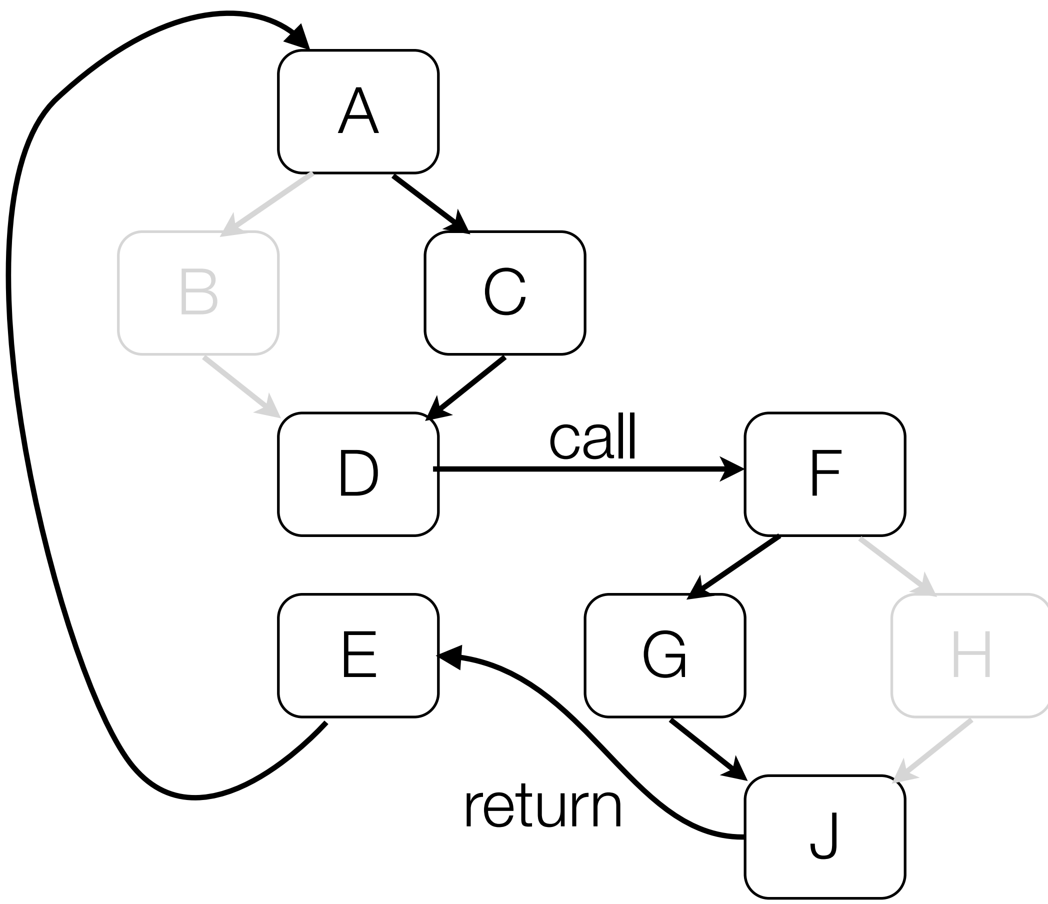
Option 2: Build custom VM

- Lots of effort, though it may be possible to reuse some of GHC's runtime system.
- Features can be tuned for executing Haskell.
- May also be a good platform for other (statically typed) functional languages.
- Use ideas and some code from open source projects: PyPy, V8, Mono, LuaJIT

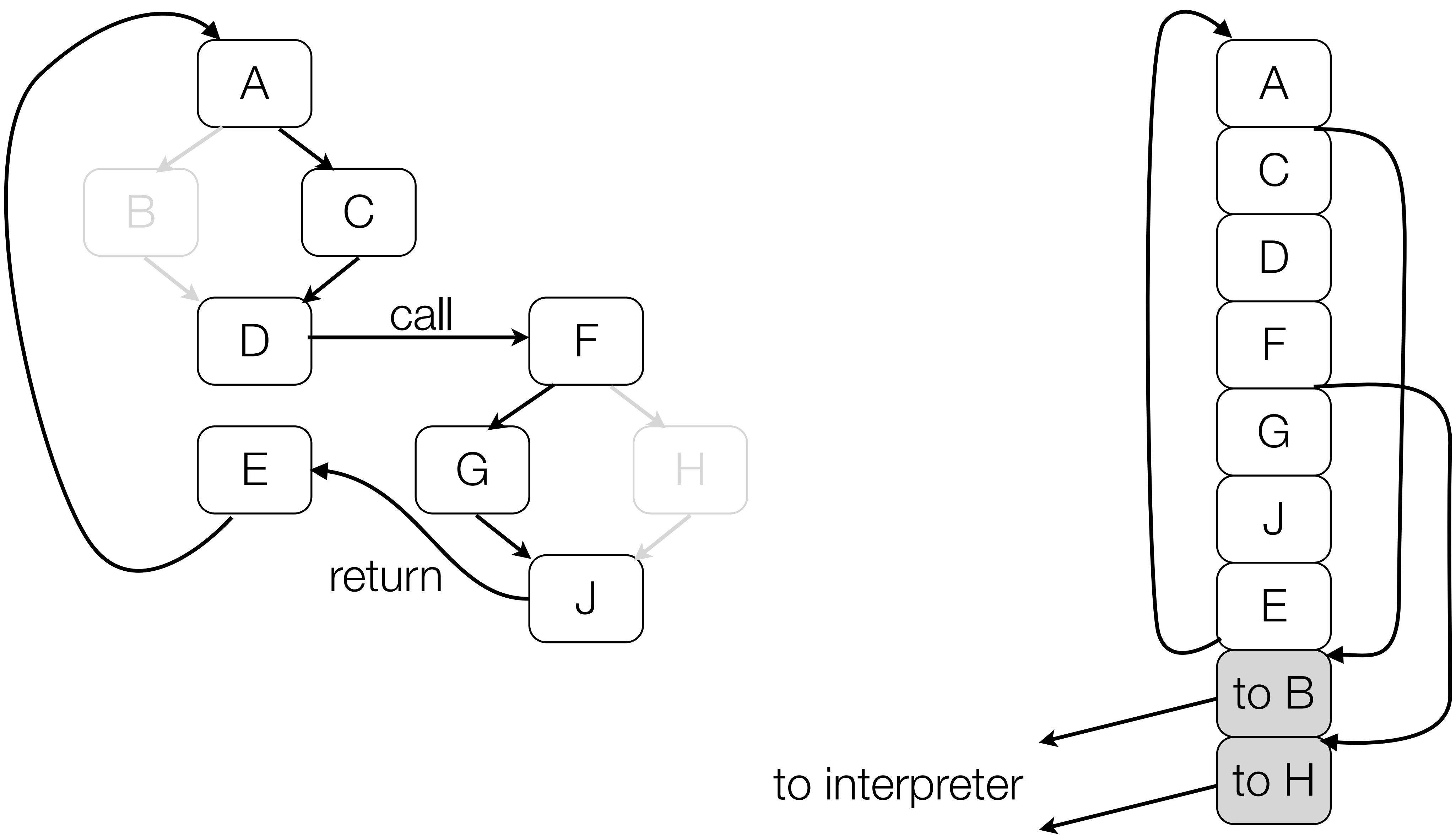
Trace-based Just-in-time compilation



Trace-based Just-in-time compilation



Trace-based Just-in-time compilation



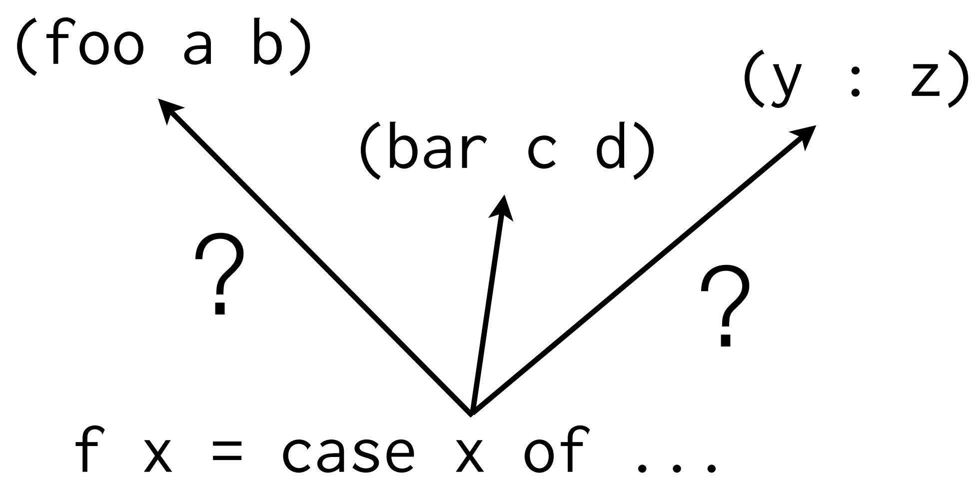
scope of optimisation

Trace-based Just-in-time Compilation

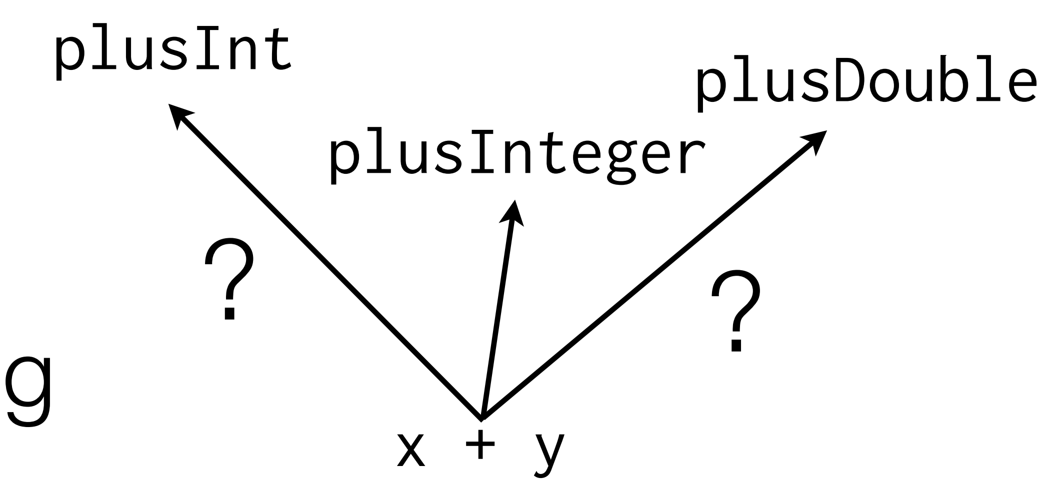
- Tracing JITs have recently found their way into the programming language mainstream: Tracemonkey (Javascript), LuaJIT 2, Android's Dalvik VM, PyPy (Python), others: SPUR (CIL/.net)
- Most of these languages are dynamically typed.
- Great for dynamic languages - very large static control flow graphs (due to runtime type checks).
- A trace-based JIT creates a specialised monomorphic version for each frequently executed path.
- Simple and quick compiler, thus short warm-up time.

Haskell is dynamic at runtime, too!

Thunks



Overloading



Thunks and Specialisation

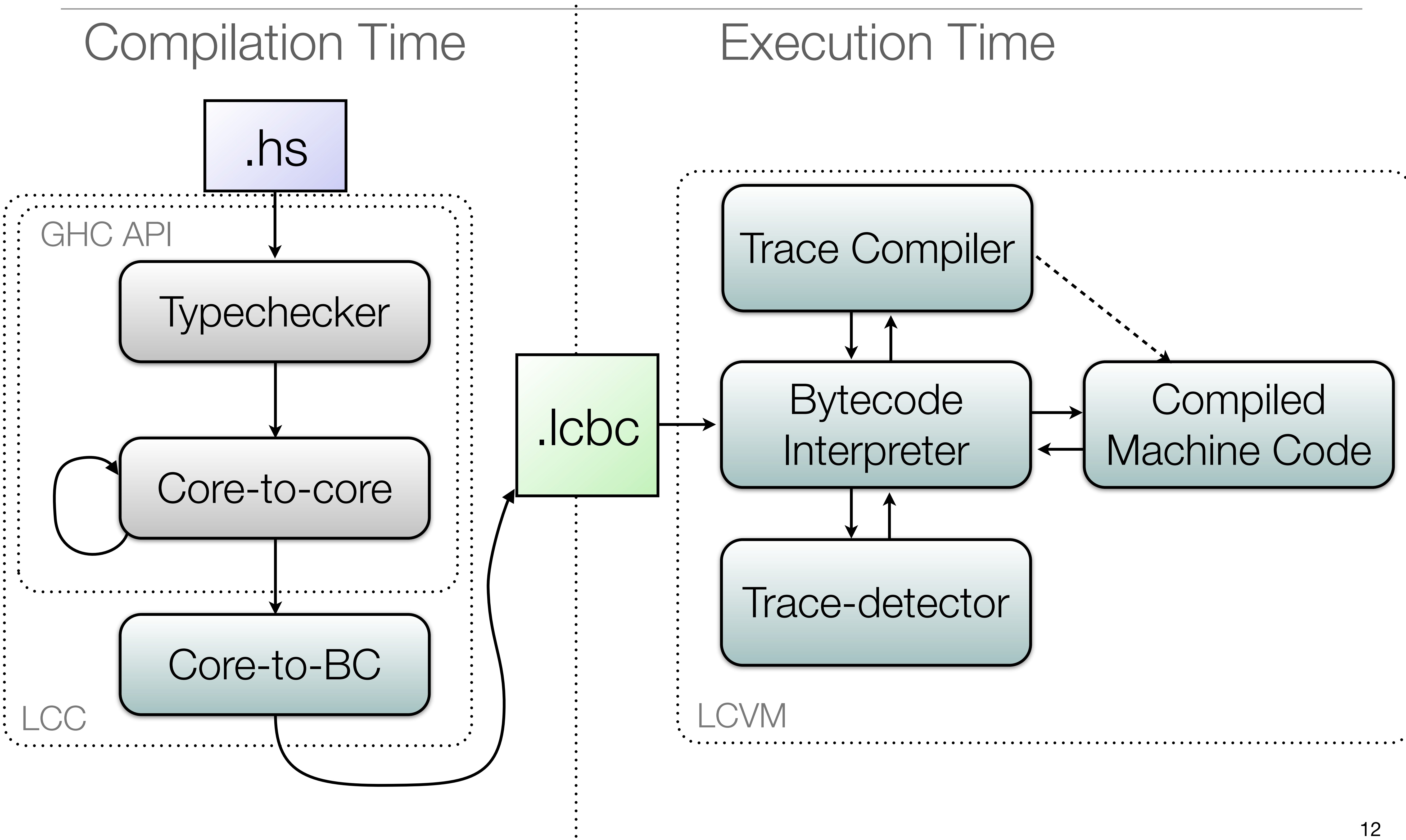
- map :: (a -> b) -> [a] -> [b]
- map (+1) :: [Int] -> [Int]
- map (+1) (generateList ...)
- sum (map (+1) (upto 1 100))

- Supercompilation is not yet ready for production (progress seems stalled?)

Overloading

- add :: Num a => a -> a -> a
- add :: DNum a -> a -> a -> a
- add d x y = (plus d) x y
- addInt :: Int -> Int -> Int
  addInt (I x) (I y) = I (primIntAdd x y)

Lambdachine



An Example

- ```
  upto :: Int -> Int -> [Int]
  upto lo hi =
    if lo > hi then [] else lo : upto (lo + 1) hi
  ```

- ```
  sum :: Int -> [Int] -> Int
  sum !acc l = case l of
    []     -> acc
    (x:xs) -> sum (acc + x) xs
  ```

- ```
  test = print (sum 0 (upto 1 100))
  ```
  ```

- ```
  upto :: Int -> Int -> [Int]
  upto lo hi =
    if lo > hi then [] else lo : upto (lo + 1) hi
  ```

- ```
  sum :: Int -> [Int] -> Int
  sum !acc l = case l of
    []     -> acc
    (x:xs) -> sum (acc + x) xs
  ```

- ```
  sum 55 (upto 11 100)
  case (upto 11 100) of ...
  case (upto 11 100) of ...
  case (if 11 > 100 then ...) of ...
  case (if False then ...) of ...
  case (11 : upto 12 100) of ...
  case (11 : upto 12 100) of ...
  sum (55 + 11) (upto 12 100)
  sum 66 (upto 12 100)
  ```
  ```

```
start:
 Obj *acc = base[0];  // load arg0
 guard (info(acc) == Int);
 Obj *l = base[1];    // load arg1
 guard (info(l) == upto_thunk)
   // Enter thunk
 int lo = l[1];  // load free var0
 int hi = l[2];  // load free var1
 guard (lo <= hi);
 Obj *y = new I#(lo);
 int lo2 = lo + 1;
 Obj *ys = new upto_thunk(lo2, hi);
 Obj *res = new Cons(y, ys);
 update(l, res);
   // Return to sum
 guard (info(res) == Cons);
 Obj *x = res[0];
 Obj *xs = res[1];
 int x_u = x[0];
 int acc_u = acc[0];
 int acc_u2 = acc_u + x_u;
 Obj *acc2 = new I#(acc_u2);
 base[0] = acc2
 base[1] = xs
goto start;
```

- ```
  upto :: Int# -> Int# -> [Int]
  upto lo hi =
    if lo ># hi then [] else
      I# lo : upto (lo +# 1#) hi
  ```

- ```
  sum :: Int -> [Int] -> Int
  sum !acc l = case l of
    []     -> acc
    (x:xs) -> sum (acc + x) xs
  ```

```
start:
 Obj *acc = base[0];  // load arg0
 guard (info(acc) == Int);
 Obj *l = base[1];    // load arg1
 guard (info(l) == upto_thunk)
   // Enter thunk
 int lo = l[1];  // load free var0
 int hi = l[2];  // load free var1
 guard (lo <= hi);
 Obj *y = new I#(lo);
```

A few microseconds later ...

```
   // Return to sum
 guard (info(res) == Cons);
 Obj *x = res[0];
 Obj *xs = res[1];
 int x_u = x[0];
 int acc_u = acc[0];
 int acc_u2 = acc_u + x_u;
 Obj *acc2 = new I#(acc_u2);
 base[0] = acc2
 base[1] = xs
goto start;
```

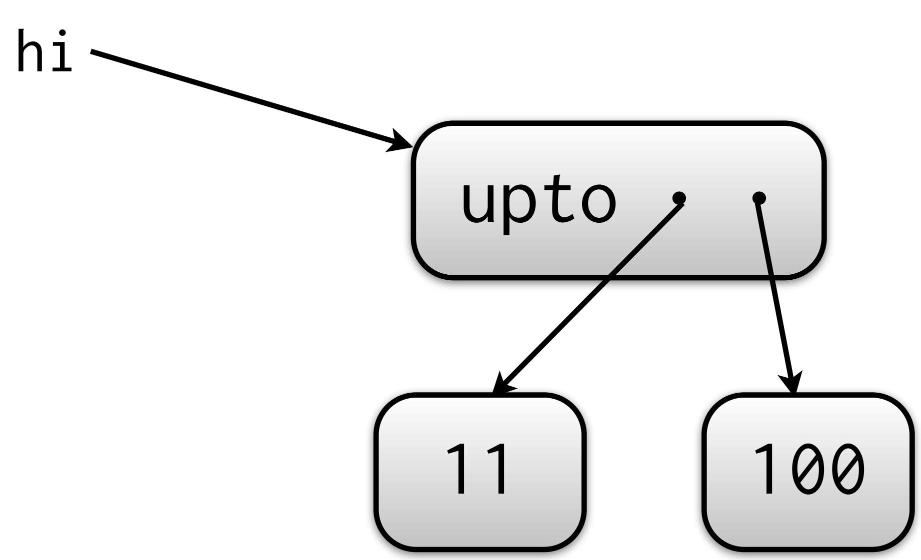
- ```
  upto :: Int# -> Int# -> [Int]
  upto lo hi =
    if lo ># hi then [] else
      I# lo : upto (lo +# 1#) hi
  ```
- ```
  sum :: Int -> [Int] -> Int
  sum !acc l = case l of
  ```

xs

```
start:
 Obj *acc = base[0];
 guard (info(acc) == Int);
 Obj *l = base[1];
 guard (info(l) == upto_thunk)
 int lo = l[1];
 int hi = l[2];
 guard (lo <= hi);
 Obj *y = new I#(lo);
 int lo2 = lo + 1;
 Obj *ys = new upto_thunk(lo2, hi);
 Obj *res = new Cons(y, ys);
 int acc_u = acc[0];
 int acc_u2 = acc_u + lo;
 Obj *acc2 = new I#(acc_u2);
loop:
 guard (lo2 <= hi);
 lo2 = lo2 + 1;
 acc_u2 = acc_u2 + lo2;
 goto loop;
```

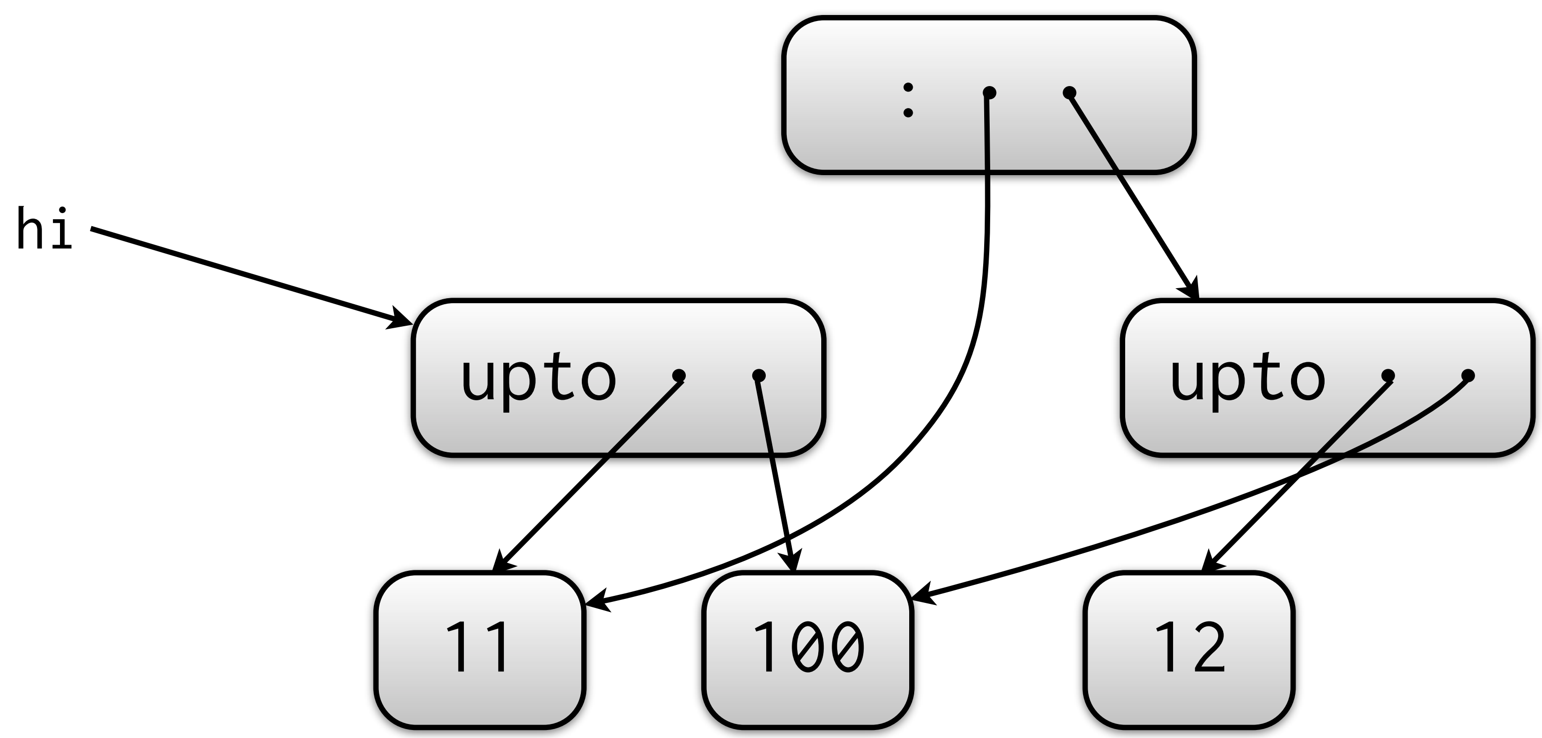
Thunks and Updates

```
sum 55 (upto 11 100)
```



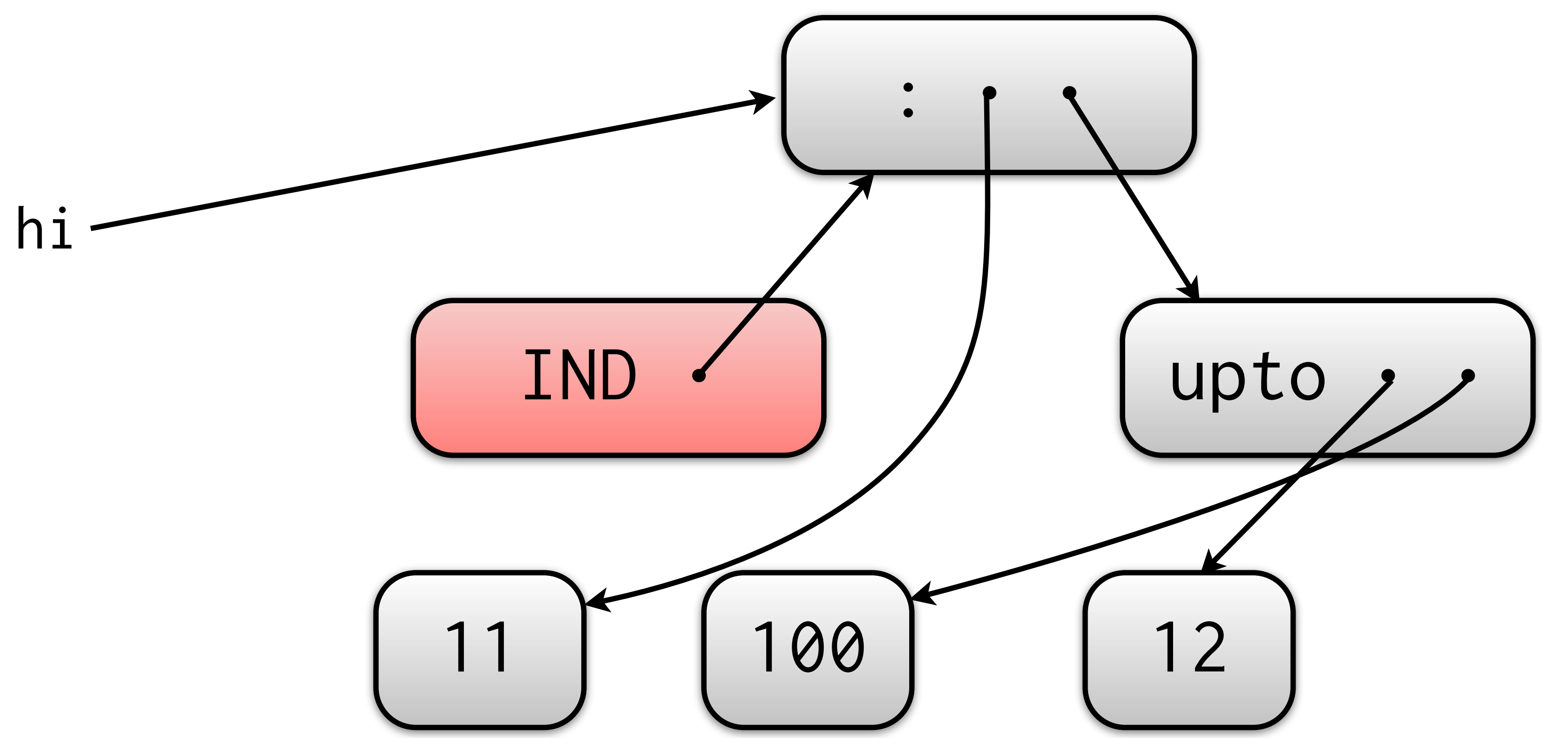
Thunks and Updates

```
sum 55 (11 : upto 12 100)
```



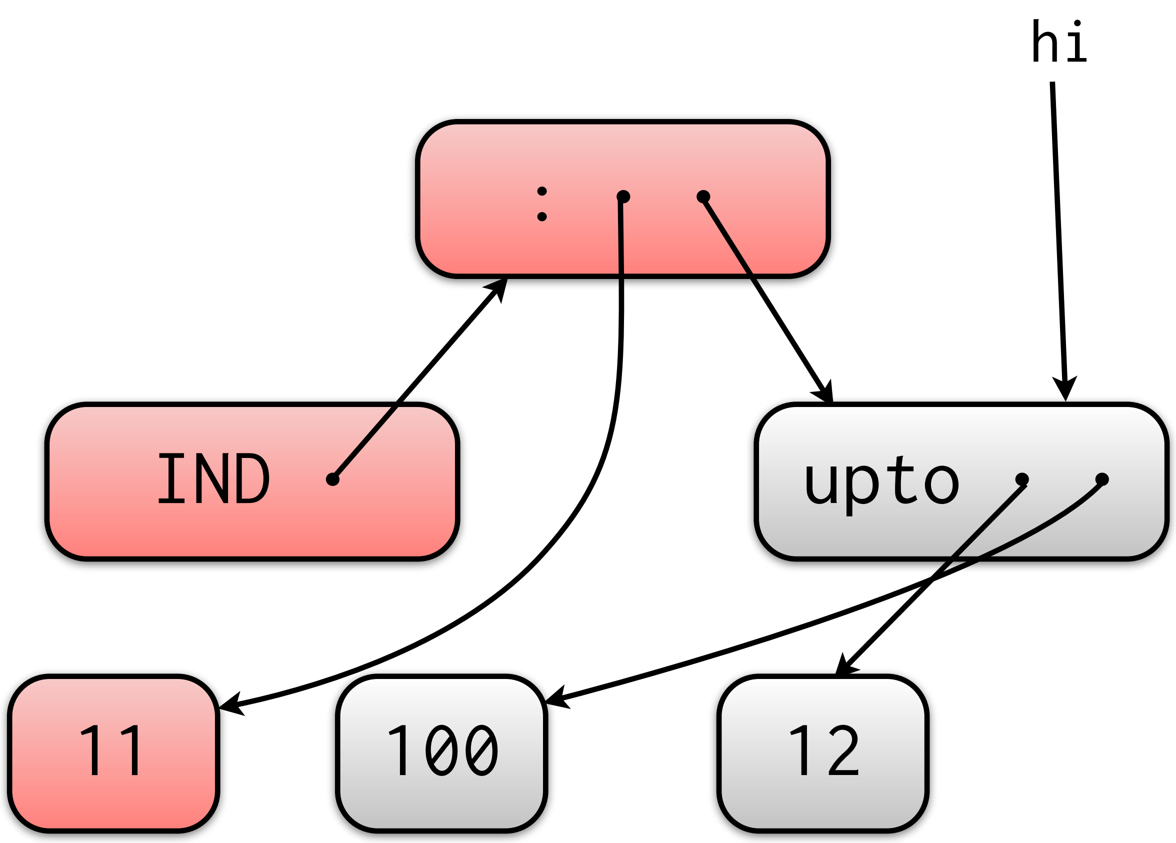
Thunks and Updates

```
sum 55 (11 : upto 12 100)
```



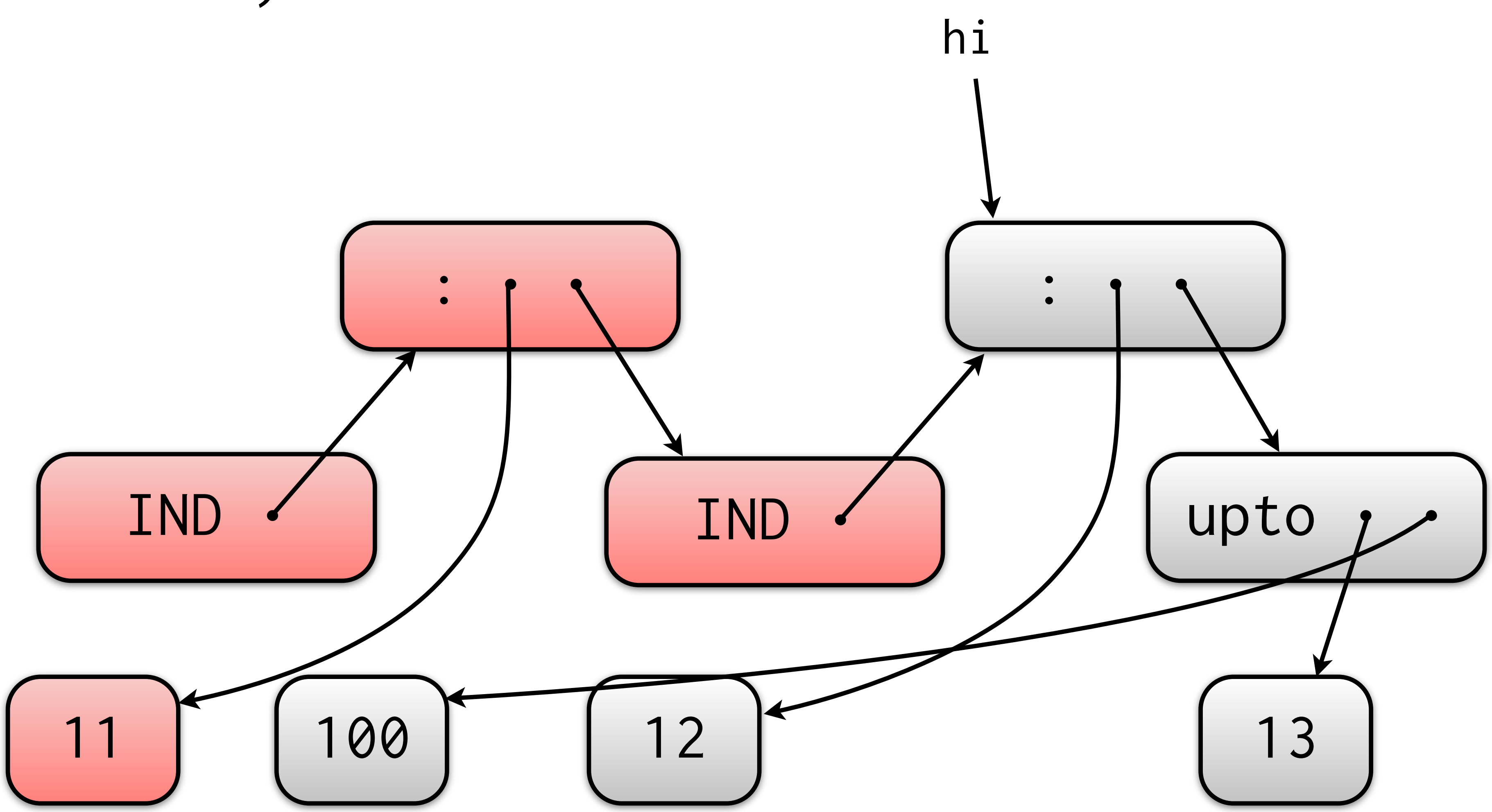
Thunks and Updates

```
sum 66 (upto 12 100)
```



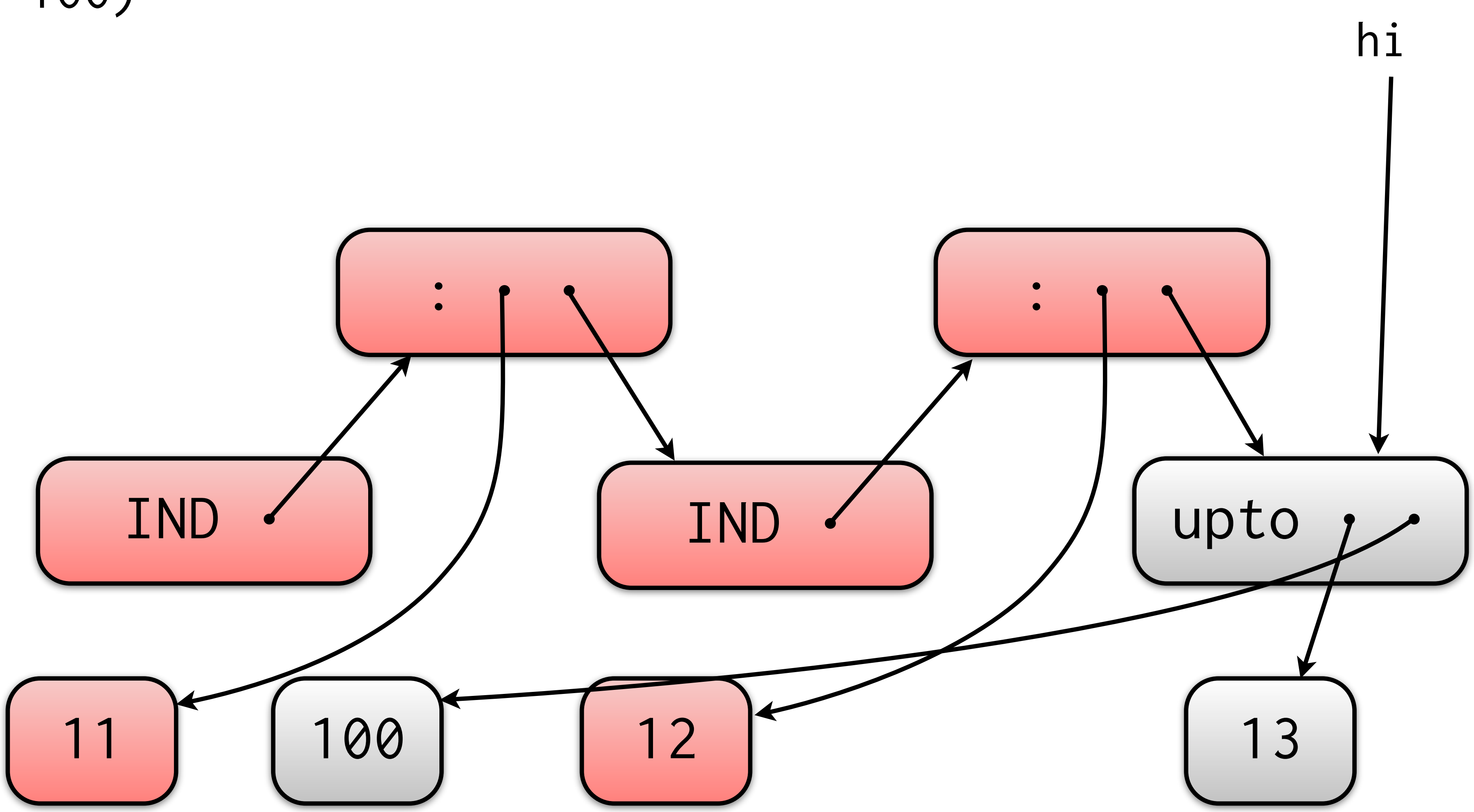
Thunks and Updates

```
sum 66 (12 : upto 13 100)
```



Thunks and Updates

```
sum 78 (upto 13 100)
```



Benchmarks (micro)

Benchmark	GHC -O2	GHC -O2 + JIT
SumFromTo1	2.31s	1.80s
SumFromTo2	4.02s	2.32s
SumSquare	2.45s	2.35s
SumStream	0.24s	1.11s
Tak	1.01s	0.84s

Benchmarks (small)

Benchmark	GHC -O2	GHC -O2 + JIT
WheelSieve2	0.34s	0.59s
Boyer	0.75s	0.74s
Constraints	0.88s	0.89s
Circsim	1.71s	3.49s
Lambda	0.74s	1.06s

Conclusions & Ongoing Work

- Trace selection is tricky. Haskell’s control flow graphs are large messy. Preferring tail-recursive loops could help.
- How can we detect (cheaply) when an update can be omitted? -- Some recent work by SPJ et al
- Only a subset of the Prelude currently supported.
- Single-threaded and very simple garbage collector -- need to integrate GHC runtime system (scheduler, generational GCs, sparks, STM, ...)

Questions?