Lambdachine: A Trace-based Just-in-time Compiler for Haskell

Thomas Schilling
University of Kent

HIW 2012, Copenhagen, Denmark

Problems with Static Compilation

- GHC is a very good static compiler. However, the static approach has a problems, too.

- Multiple compilation “ways” (vanilla, profiled, threaded, HPC, ...). Usually, all dependency must be available in the same mode.

- How to get the compiler to fully optimise my critical loops? “Why didn’t that rule fire? ... Hm, it didn’t inline that ... Why not?” - “Oops. Now the compiled code is 3x larger and takes 3x longer to compile. Well, fair enough, I guess.” - “Why does it construct that dictionary in the inner loop?” - etc.

- No portable bytecode or serializable closures. (GHCi bytecode might be portable, but it’s slow.)

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)

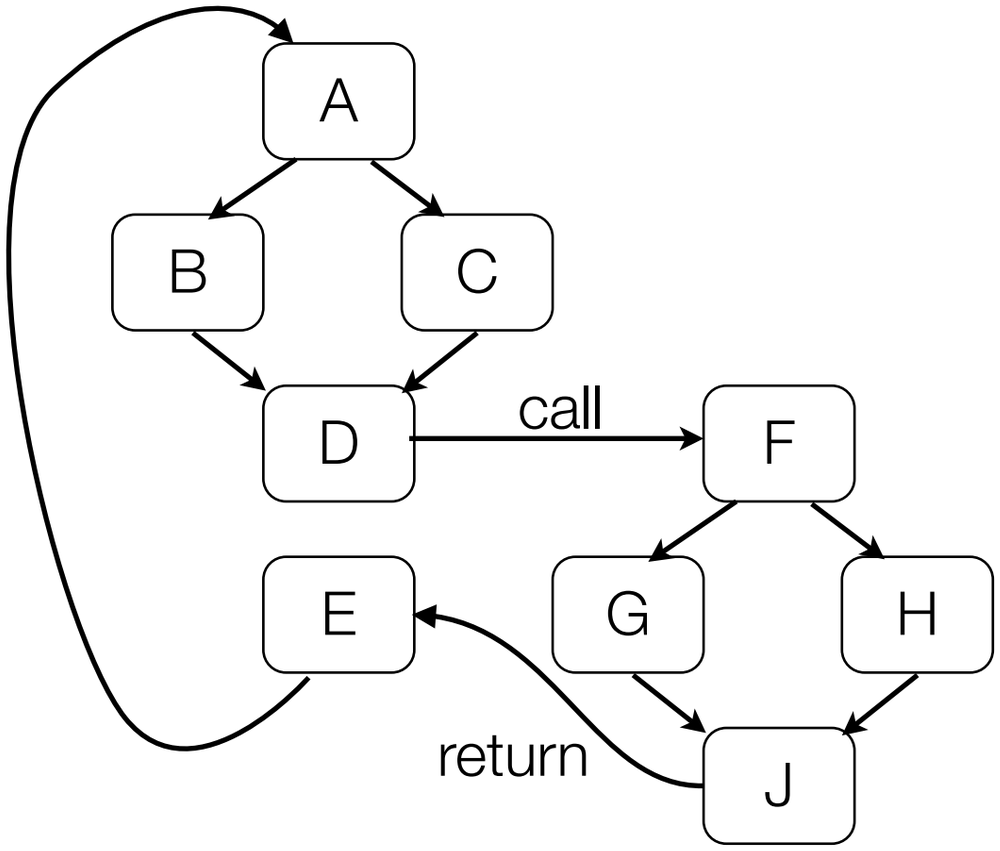
- Most of these languages are dynamically typed.

- Great for dynamic languages - very large static control flow graphs (due to runtime type checks).

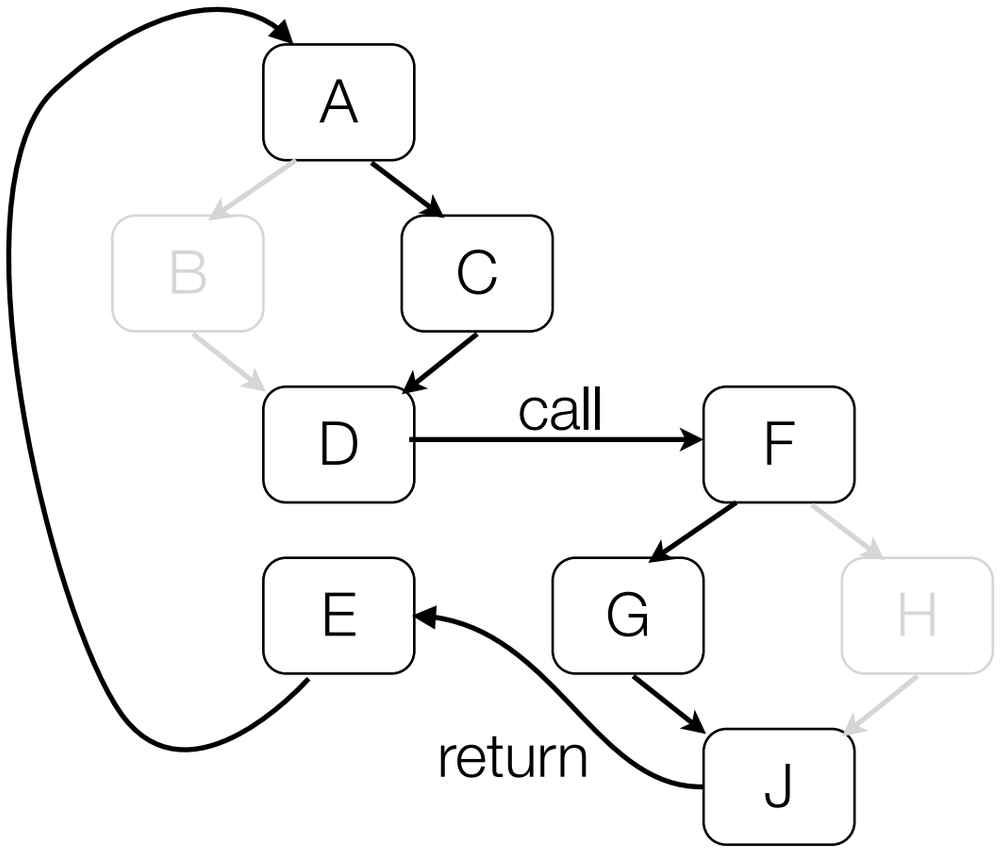
- A trace-based JIT creates a specialised monomorphic version for each frequently executed path.

- Simple and quick compiler, thus short warm-up time.

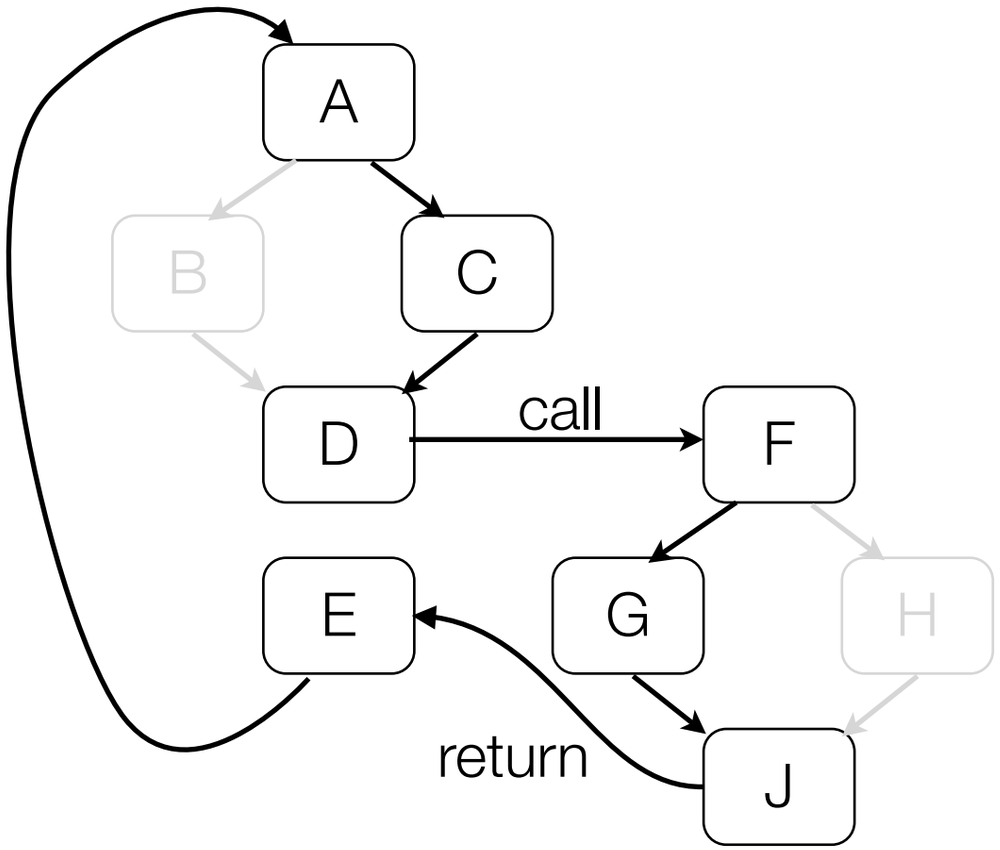
Trace-based Just-in-time compilation

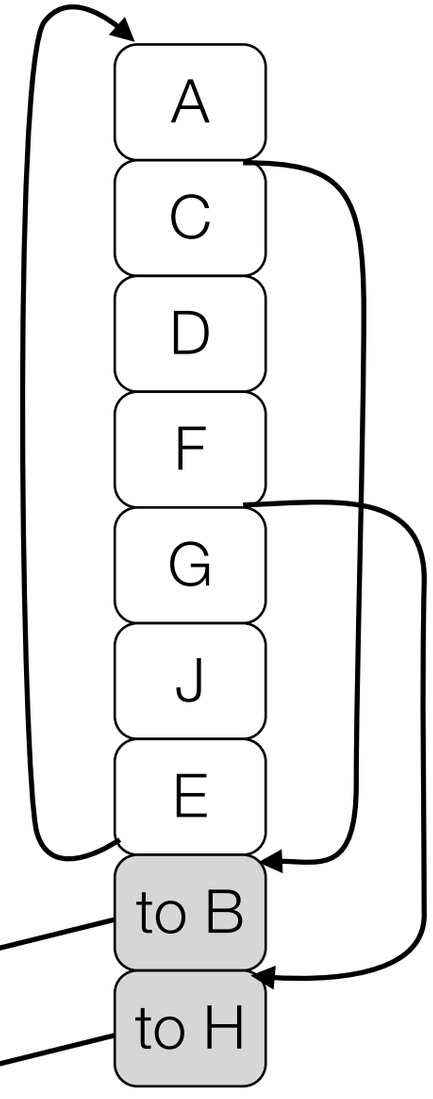


Trace-based Just-in-time compilation



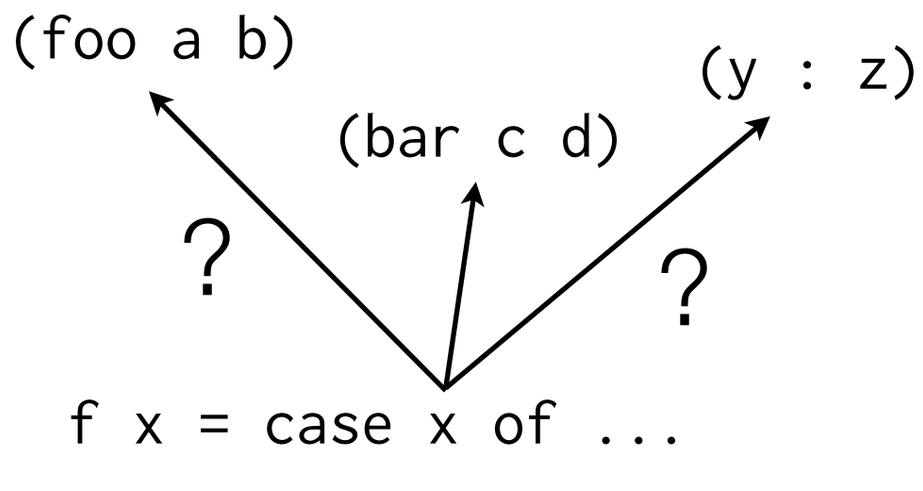
Trace-based Just-in-time compilation



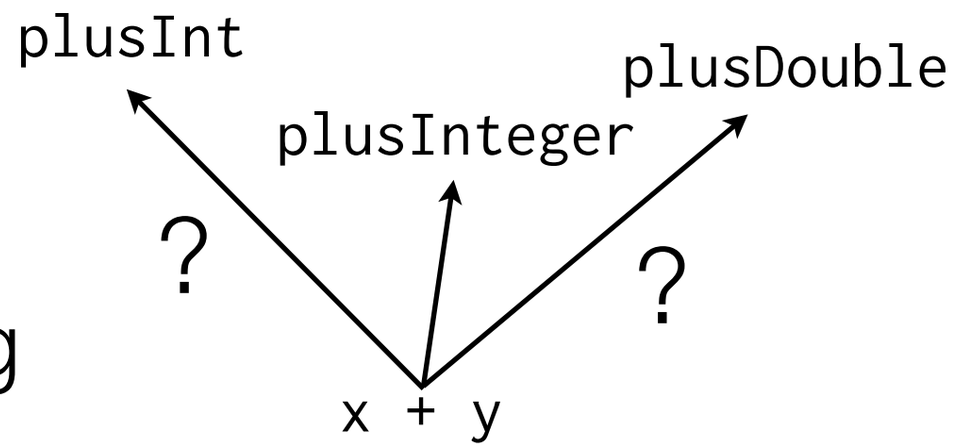


scope of optimisation

Haskell is dynamic at runtime, too!

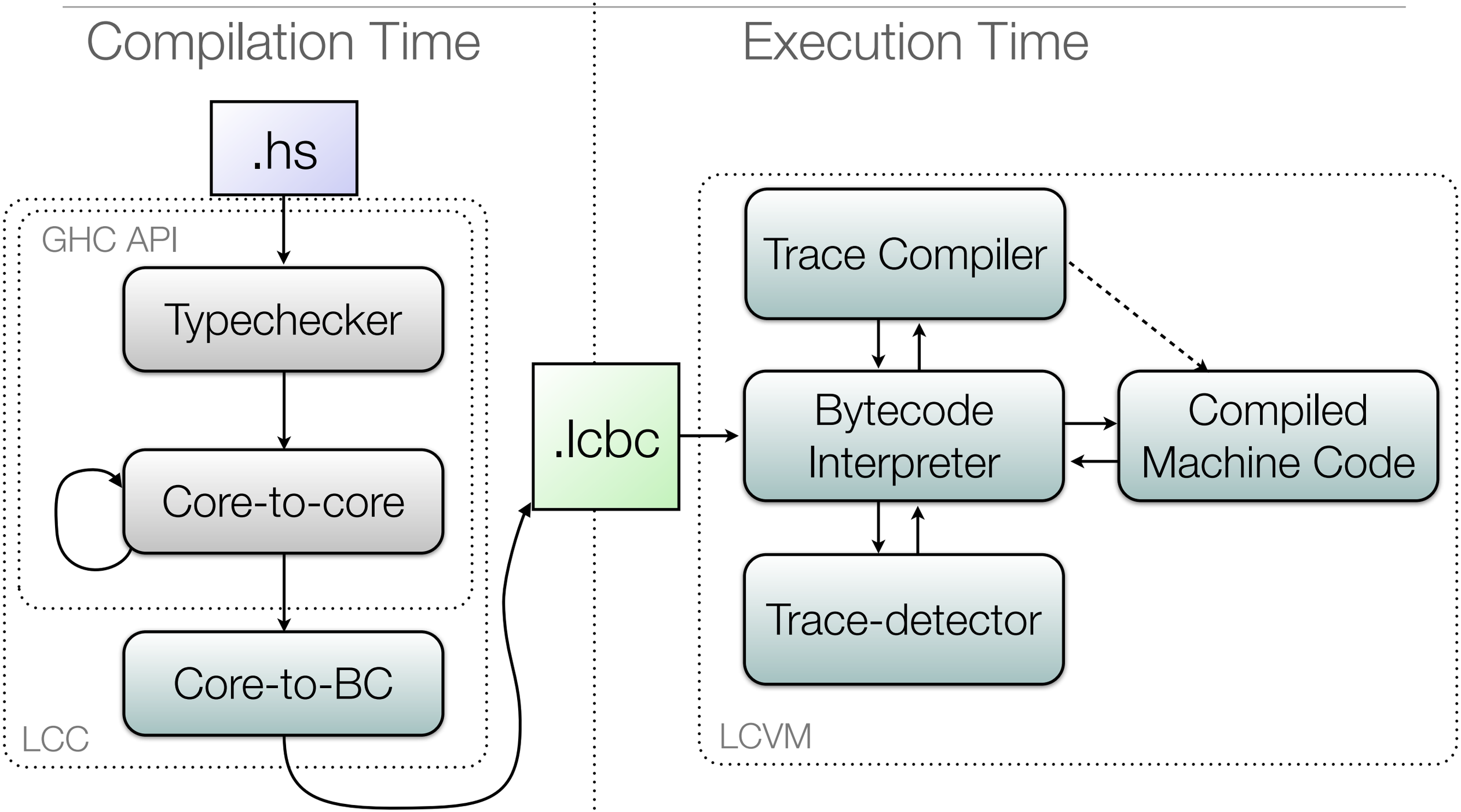


Thunks



Overloading

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);
 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;
```

We just allocated res, so we know its constructor.

```
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 = y;
   // Obj *xs = ys;
   // int x_u = x[0] = lo;
  int acc_u = acc[0];
  int acc_u2 = acc_u + lo;
  Obj *acc2 = new I#(acc_u2);
  base[0] = acc2
  base[1] = ys
goto start;
```

```
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 = y;
   // Obj *xs = ys;
   // int x_u = x[0] = lo;
 int acc_u = acc[0];
 int acc_u2 = acc_u + lo;
 Obj *acc2 = new I#(acc_u2);
 base[0] = acc2
 base[1] = ys
goto start;
```

If there's no other pointer to "l" we can omit the update.

```
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 = y;
   // Obj *xs = ys;
   // int x_u = x[0] = lo;
  int acc_u = acc[0];
  int acc_u2 = acc_u + lo;
  Obj *acc2 = new I#(acc_u2);
  base[0] = acc2
  base[1] = ys
goto start;
```

There are a lot of redundancies across the loop boundary.

“Peel” the loop once.

```
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 = y;
   // Obj *xs = ys;
   // int x_u = x[0] = lo;
 int acc_u = acc[0];
 int acc_u2 = acc_u + lo;
 Obj *acc2 = new I#(acc_u2);
 base[0] = acc2
 base[1] = ys
loop:
```

```
loop:  // start of peeled loop
 Obj *acc' = base[0];
 guard (info(acc') == Int);
 Obj *l' = base[1];
 guard (info(l') == upto_thunk)
   // Enter 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');
   //
   // Return to sum
   //
   //
   //
   //
 int acc_u' = acc'[0];
 int acc_u2' = acc_u' + lo';
 Obj *acc2' = new I#(acc_u2');
 base[0] = acc2'
 base[1] = ys'
 goto loop
```

```
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 = y;
   // Obj *xs = ys;
   // int x_u = x[0] = lo;
 int acc_u = acc[0];
 int acc_u2 = acc_u + lo;
 Obj *acc2 = new I#(acc_u2);
 base[0] = acc2
 base[1] = ys
loop:
```

```
loop:  // start of peeled loop
 Obj *acc' = acc2;
 guard (info(acc') == Int);
 Obj *l' = ys;
 guard (info(l') == upto_thunk)
   // Enter 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');
   //
   // Return to sum
   //
   //
   //
   //
 int acc_u' = acc'[0];
 int acc_u2' = acc_u' + lo';
 Obj *acc2' = new I#(acc_u2');
 base[0] = acc2'
 base[1] = ys'
 goto loop
```

```
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 = y;
   // Obj *xs = ys;
   // int x_u = x[0] = lo;
 int acc_u = acc[0];
 int acc_u2 = acc_u + lo;
 Obj *acc2 = new I#(acc_u2);
 base[0] = acc2
 base[1] = ys
loop:
```

```
loop:  // start of peeled loop
 Obj *acc' = acc2;
 guard (info(acc2) == Int);
 Obj *l' = ys;
 guard (info(ys) == upto_thunk)
   // Enter thunk
 int lo' = ys[1];
 int hi' = ys[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');
   //
   // Return to sum
   //
   //
   //
   //
 int acc_u' = acc2[0];
 int acc_u2' = acc_u' + lo';
 Obj *acc2' = new I#(acc_u2');
 acc2 = acc2‘
 ys = ys'
 goto loop
```

```
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 = y;
   // Obj *xs = ys;
   // int x_u = x[0] = lo;
 int acc_u = acc[0];
 int acc_u2 = acc_u + lo;
 Obj *acc2 = new I#(acc_u2);
 base[0] = acc2
 base[1] = ys
loop:
```

```
loop:  // start of peeled loop
 Obj *acc' = acc2;
 guard (info(acc2) == Int);
 Obj *l' = ys;
 guard (info(ys) == upto_thunk)
   // Enter thunk
 int lo' = lo2;
 int hi' = hi;
 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');
   //
   // Return to sum
   //
   //
   //
   //
 int acc_u' = acc_u2;
 int acc_u2' = acc_u' + lo';
 Obj *acc2' = new I#(acc_u2');
 acc2 = acc2‘
 ys = ys‘
 lo2 = lo2'
 goto loop
```

```
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 = y;
   // Obj *xs = ys;
   // int x_u = x[0] = lo;
 int acc_u = acc[0];
 int acc_u2 = acc_u + lo;
 Obj *acc2 = new I#(acc_u2);
 base[0] = acc2
 base[1] = ys
loop:
```

```
loop:  // start of peeled loop
 Obj *acc' = acc2;  // was: base[0]
 guard (info(acc2) == Int);
 Obj *l' = ys;      // was: base[1]
 guard (info(ys) == upto_thunk)
   // Enter thunk
 int lo' = lo2;
 int hi' = hi;
 guard (lo2 <= hi);
 Obj *y' = new I#(lo2);
 int lo2' = lo2 + 1;
 Obj *ys' = new upto_thunk(lo2', hi);
 Obj *res' = new Cons(y', ys');
   //
   // Return to sum
   //
   //
   //
   //
 int acc_u' = acc_u2;
 int acc_u2' = acc_u2 + lo2;
 Obj *acc2' = new I#(acc_u2');
 acc2 = acc2‘; acc_u2 = acc_u2';
 ys = ys‘
 lo2 = lo2'
 goto loop
```

```
start:
 Obj
 gua
 Obj
 gua
   //
 int
 int
 gua
 Obj
 int
 Obj
 Obj
   //
   //
   //
   //
   //
   //
 int
 int
 Obj
 base
 base
loop:
```

The results of heap allocations are only needed when exiting the trace.

Thus: Only do the allocation when a guard fails.

```
loop:  // start of peeled loop
 Obj *acc' = acc2;  // was: base[0]
 guard (info(acc2) == Int);
 Obj *l' = ys;      // was: base[1]
 guard (info(ys) == upto_thunk)
   // Enter thunk
 int lo' = lo2;
 int hi' = hi;
 guard (lo2 <= hi);
 Obj *y' = new I#(lo2);
 int lo2' = lo2 + 1;
 Obj *ys' = new upto_thunk(lo2', hi);
 Obj *res' = new Cons(y', ys');
   //
   // Return to sum
   //
   //
   //
   //
 int acc_u' = acc_u2;
 int acc_u2' = acc_u2 + lo2;
 Obj *acc2' = new I#(acc_u2');
 acc2 = acc2'; acc_u2 = acc_u2'
 ys = ys'
 lo2 = lo2'
 goto loop
```

```
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 = y;
   // Obj *xs = ys;
   // int x_u = x[0] = lo;
 int acc_u = acc[0];
 int acc_u2 = acc_u + lo;
 Obj *acc2 = new I#(acc_u2);
 base[0] = acc2
 base[1] = ys
loop:
```

```
loop:  // start of peeled loop
 Obj *acc' = acc2;  // was: base[0]
 guard (info(acc2) == Int);
 Obj *l' = ys;      // was: base[1]
 guard (info(ys) == upto_thunk)
   // Enter thunk
 int lo' = lo2;
 int hi' = hi;
 guard (lo2 <= hi);
   Obj *y' = new I#(lo2);
 int lo2' = lo2 + 1;
   Obj *ys' = new upto_thunk(lo2', hi);
   Obj *res' = new Cons(y', ys');
   //
   // Return to sum
   //
   //
   //
   //
 int acc_u' = acc_u2;
int acc_u2' = acc_u2 + lo2;
   Obj *acc2' = new I#(acc_u2');
   acc2 = acc2'; acc_u2 = acc_u2'
   ys = ys'
lo2 = lo2'
goto loop
```

```
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 = y;
   // Obj *xs = ys;
   // int x_u = x[0] = lo;
 int acc_u = acc[0];
 int acc_u2 = acc_u + lo;
 Obj *acc2 = new I#(acc_u2);
 base[0] = acc2
 base[1] = ys
loop:
```

```
loop:  // start of peeled loop
 guard (lo2 <= hi);
 int lo2' = lo2 + 1;
 int acc_u2' = acc_u2 + lo2;
 acc_u2 = acc_u2';
 lo2 = lo2';
 goto loop;
```

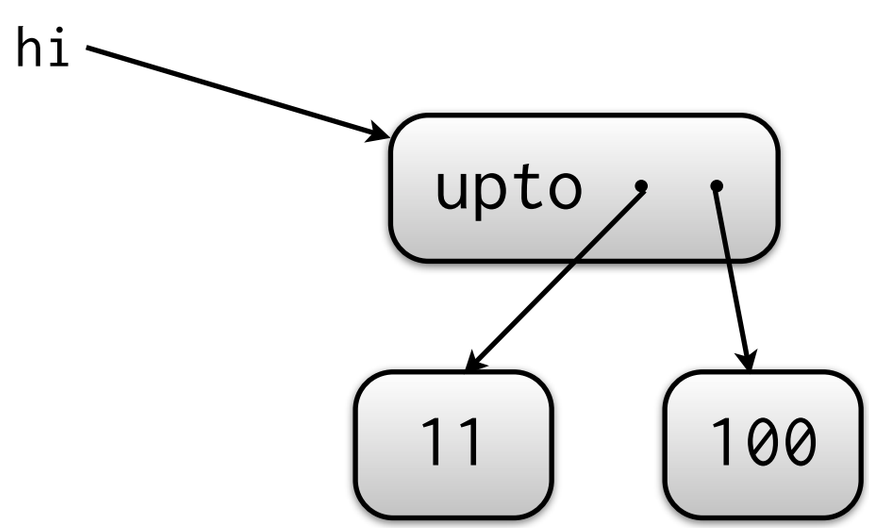
```
start:
 Obj *acc = base[0];  // load arg0
 guard (info(acc) == Int);
 Obj *l = base[1];    // load arg1
 guard (info(l) == upto_thunk)
   // Enter thunk
 int lo = l[1];  // load free var0
 int hi = l[2];  // load free var1
 guard (lo <= hi);
 Obj *y = new I#(lo);
 int lo2 = lo + 1;
 Obj *ys = new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

```
loop:  // start of peeled loop
 guard (lo2 <= hi);
 lo2 = lo2 + 1;
 acc_u2 = acc_u2 + lo2;
 goto loop;
```

```
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;
```

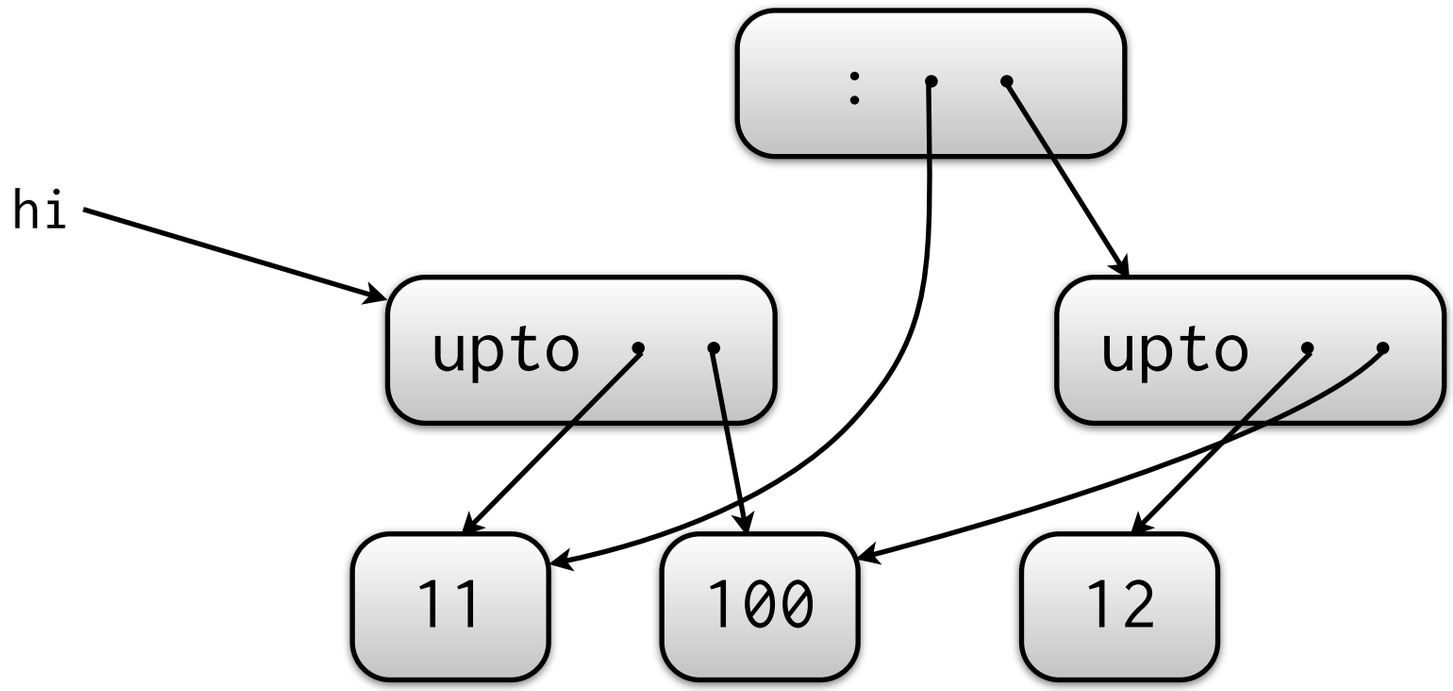
Thunks and Updates

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



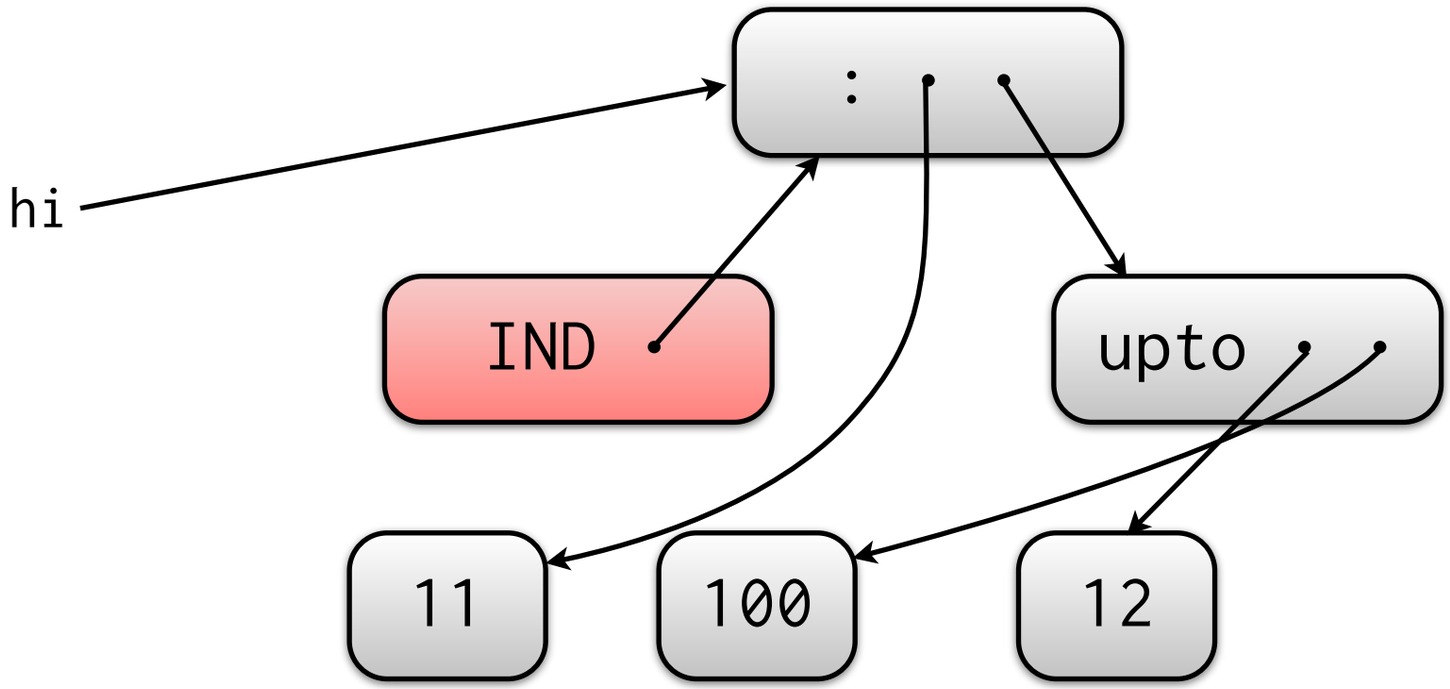
Thunks and Updates

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



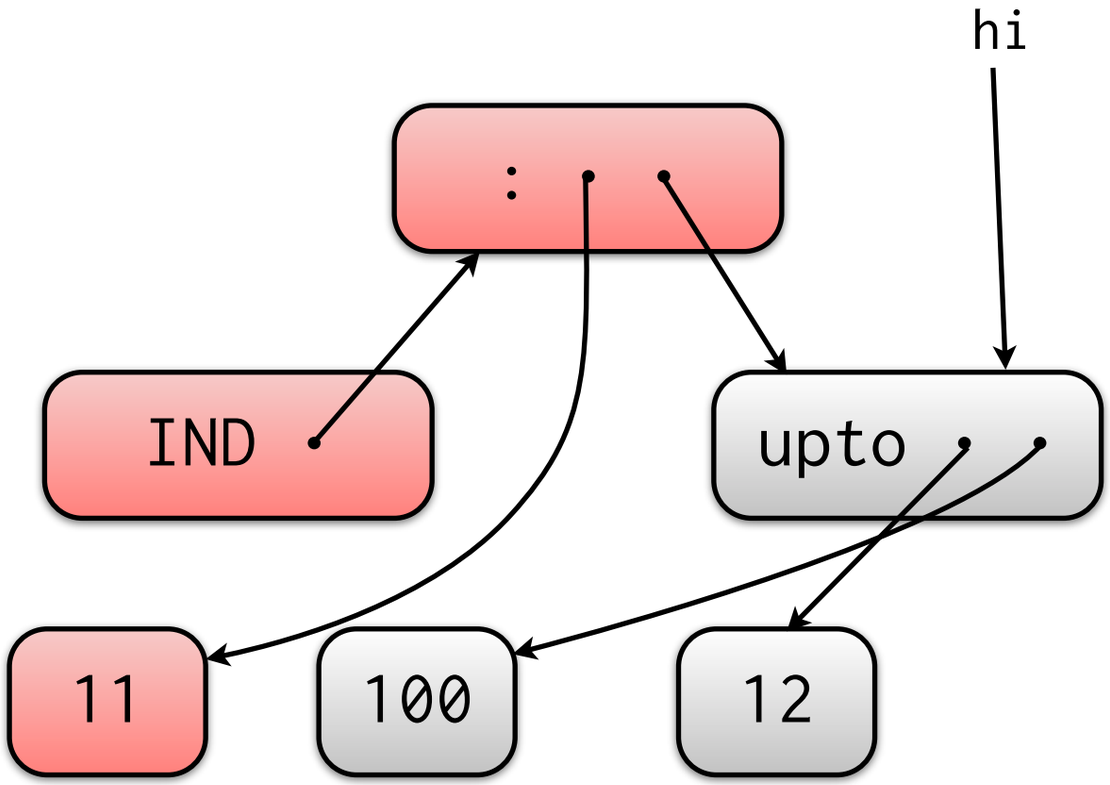
Thunks and Updates

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



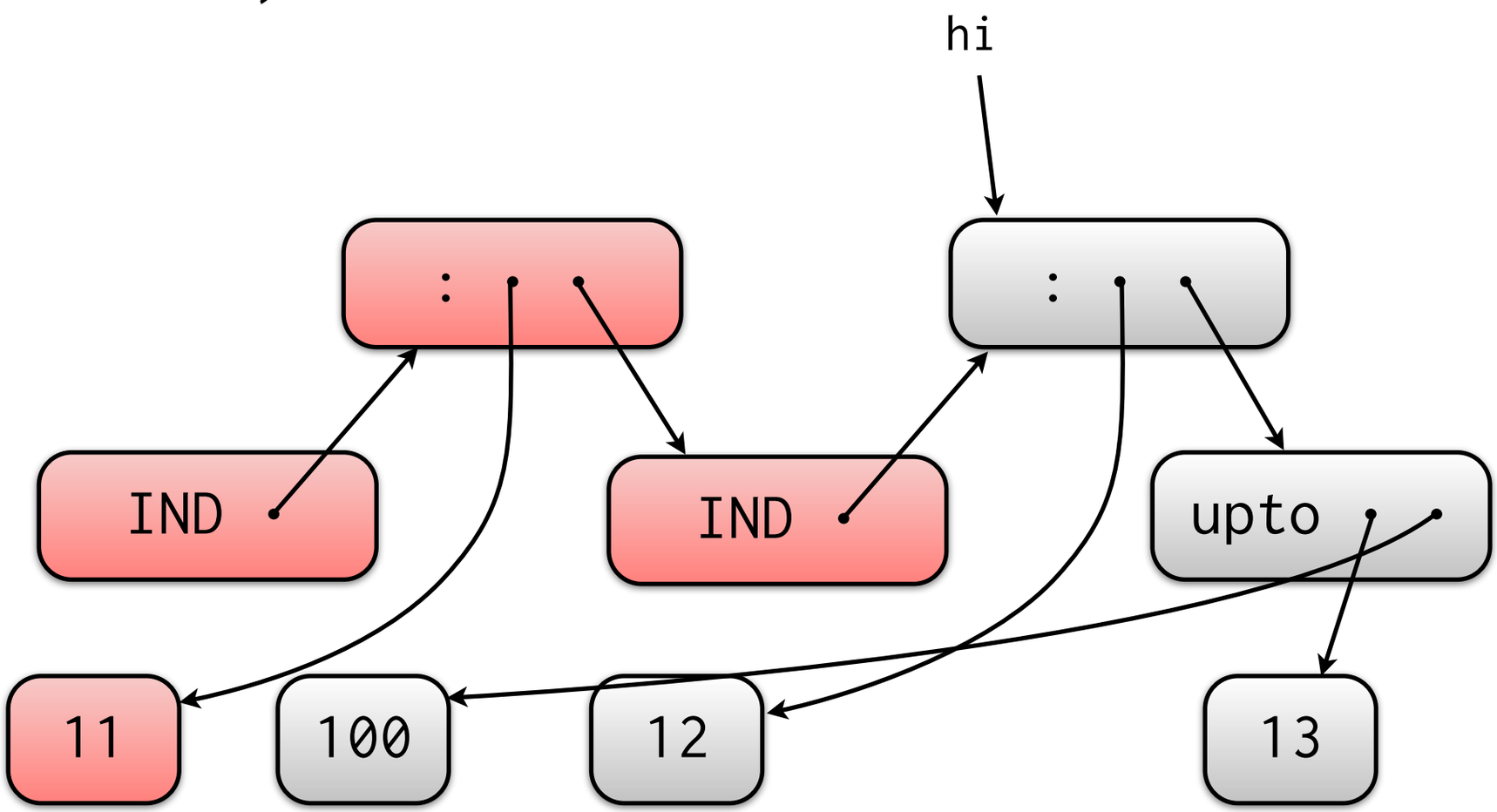
Thunks and Updates

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



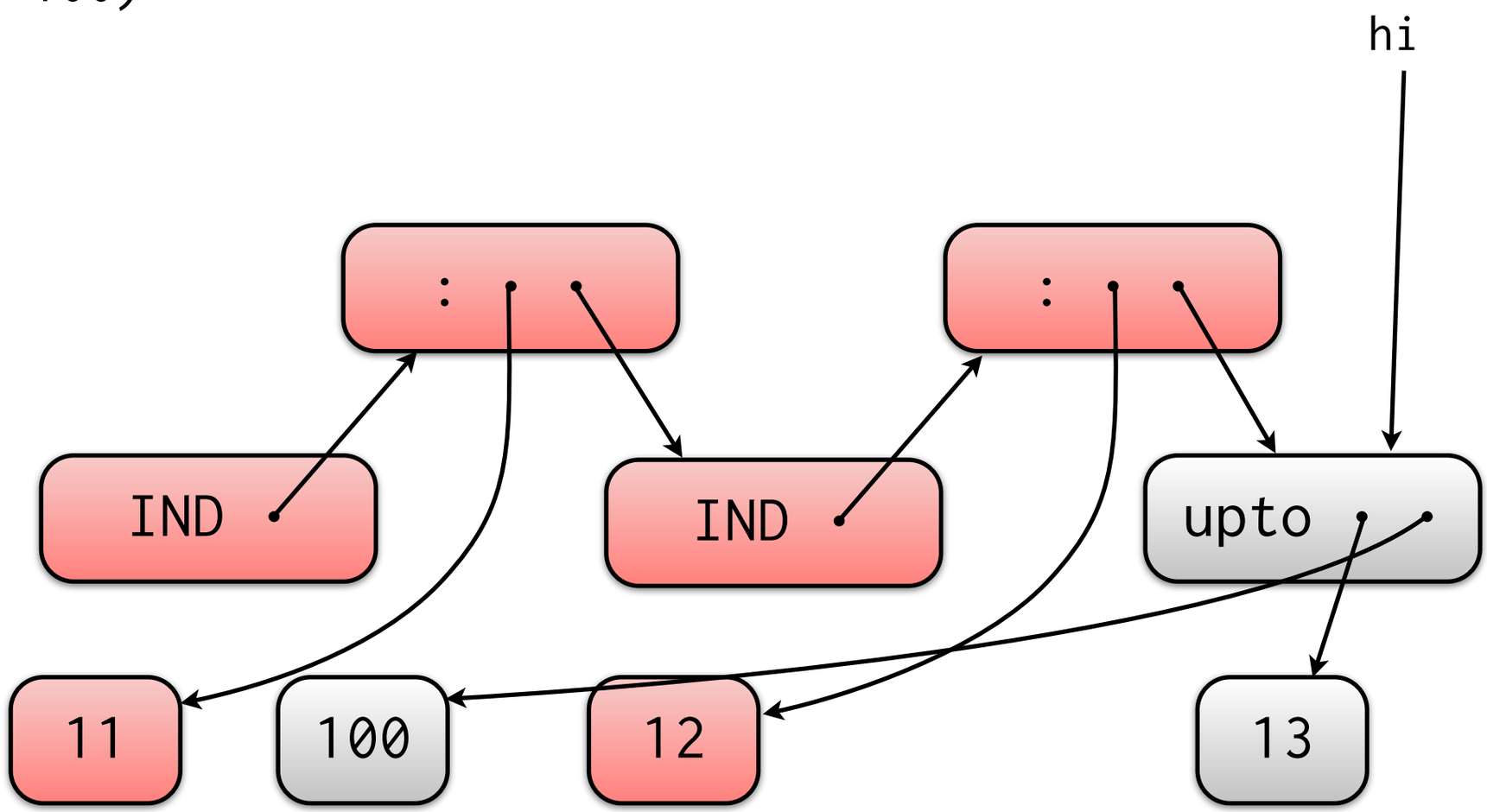
Thunks and Updates

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



Thunks and Updates

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



Demo

Demo (Backup)

- ```
  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))
  ```
  ```

Demo (Backup)

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

GHC 7.0.3 -O2: 1.14s MUT (1.18s total) 6.5GB/s

```
    []     -> acc
```

Lambdachine: 1.23s MUT (1.24s total) 6.5GB/s

Demo (Backup)

- ```
  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
  ```

- ```
  map' :: (a -> b) -> [a] -> [b]
  map' _ [] = []
  map' f (x:xs) = let !y = f x in y : map' f xs
  ```

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

Demo (Backup)

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

GHC 7.0.3 -O2: 2.38s MUT (2.46s total) 6.2GB/s

```
  []     -> acc
```

Lambdachine: 2.50s MUT (2.54s total) 6.1GB/s

```
map' _ [] = []
map' f (x:xs) = let !y = f x in y : map' f xs
```

```
test = print (sum 0 (map (plusInt 1) (upto 1 100)))
```

Conclusions & Ongoing Work

- ...
- Trace selection is tricky. Haskell's control flow graphs are large messy.
- How *do* we detect when an update can be omitted?
- Only a small subset of the Prelude supported.

Questions?