HARVARD
John A. Paulson
School of Engineering
and Applied Sciences

CS153: Compilers
Lecture 13:
Functional Programming Optimization

Stephen Chong

https://www.seas.harvard.edu/courses/cs153

Announcements

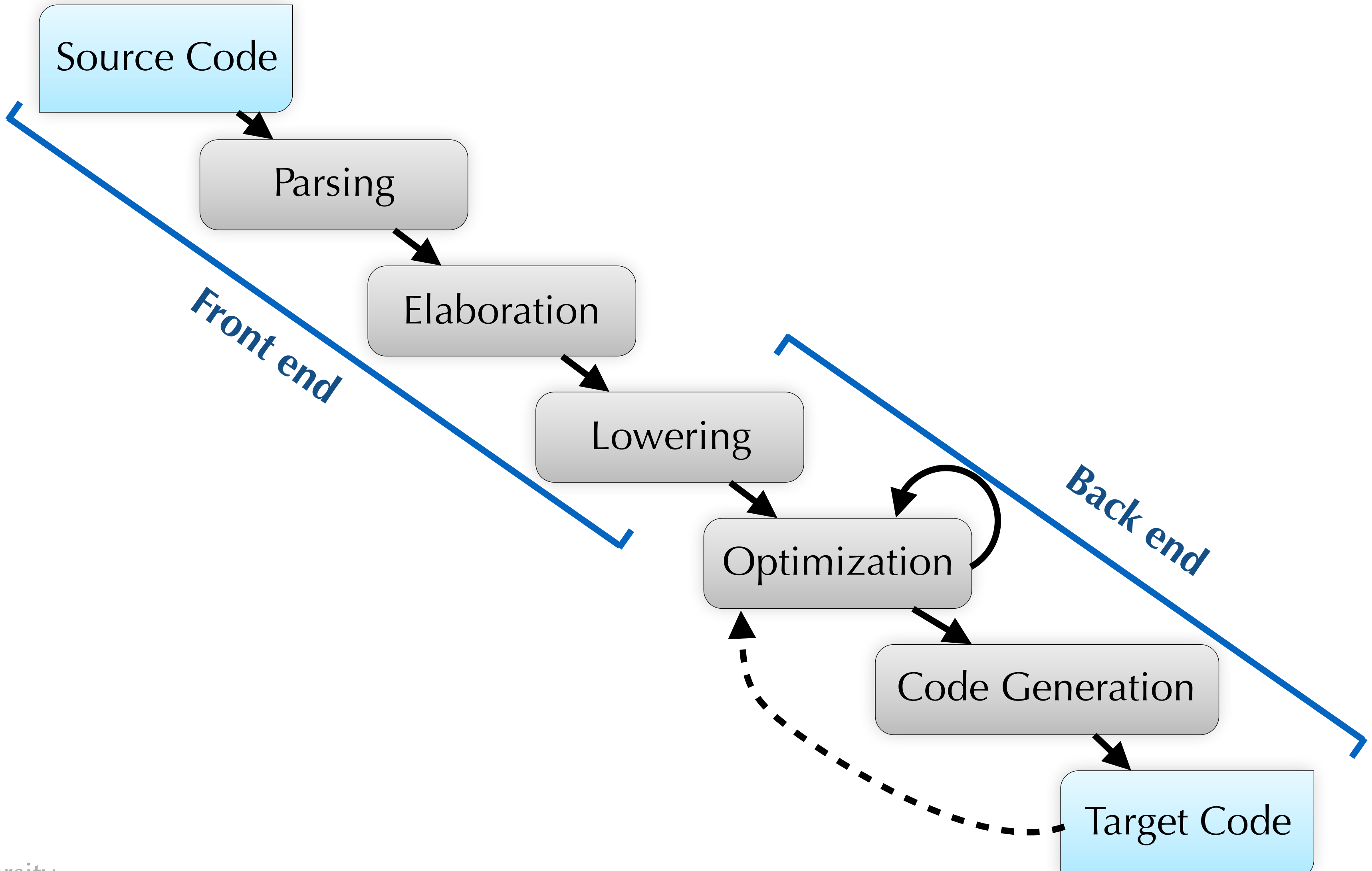
- Project 4 out
  - Due Thursday Oct 25 (9 days)
- Project 5 out
  - Due Tuesday Nov 13 (28 days)
- Project 6 will be released next week

Today

- Functional programming optimization
  - Decurryfication
  - Inlining
  - Tail call elimination
  - Lazy evaluation

Optimization

- Start of a series of lectures on optimization and analysis
- Today: Opportunities for optimizing functional programs!
  - Some at source level, some at code generation level...



Decurryfication

- Turn sequence of functions into tuples
- E.g., convert

```
let add = fun x -> fun y -> x + y
```

  to

```
let add = fun (x, y) -> x + y
```

- When is this applicable? Not applicable?
  - Can't use when nested function is used by itself
- What are the potential benefits?
  - Remove overhead of closure for the nested function
  - Tuple of arguments can be handled efficiently in registers

Inlining

- Consider the function `f(a₁,...,aₙ) = e`
- We can inline the function where it is used
- If `E` is a context, we can rewrite

  $E[f(v_1,...,v_n)]$

  to

  $E[\ e[a_1 \mapsto v_1,...,a_n \mapsto v_n]]$

  (where $e[a_1 \mapsto v_1,...,a_n \mapsto v_n]$ is expression e with var a_i replaced with value v_i)
- E.g., `g(x,y) = 1+x+y+y`
  Can rewrite `4+g(12,3)*2` to `4+(1+12+3+3)*2`

Inlining

- Consider the function `f(a1,...,an) = e`, i.e. $f(a_1,\ldots,a_n) = e$
- We can inline the function where it is used
- If `E` is a context, we can rewrite

  $E[f(v_1,\ldots,v_n)]$

  to

  $E[\ e[a_1 \mapsto v_1,\ldots,a_n \mapsto v_n]]$
- What is the benefit?
  - Avoids overhead of function call (stack frame allocation, saving registers, etc.)
  - Specializes function body to actual argument. Enables additional optimizations!
- When is it applicable? Not applicable?
  - Is applicable to recursive functions, but just not well... (more soon)
  - What if arguments are expressions?

Inlining 2

- What if arguments of `f` are non-trivial?
- If E is a context, we can rewrite

  $\texttt{E[f(e}_1\texttt{,...,e}_n\texttt{)]}$

  to

  $\texttt{E[let x}_1\texttt{=e}_1\texttt{ and ... and x}_n\texttt{=e}_n$
  $\texttt{in e[a}_1\mapsto\texttt{x}_1\texttt{,...,a}_n\mapsto\texttt{x}_n\texttt{]]}$

  where $\texttt{x}_1,\ldots,\texttt{x}_n$ are fresh variables
- Note: given `double(y) = y + y` inlining in `double(g())` produces `let x = g() in x + x` does **not** produce `g() + g()`!
  - Why is the distinction important?

Inlining recursive functions

- Consider recursive function, e.g.,

```
f(x,y) = if x < 1 then y
                else x * f(x-1,y)
```

- If we inline it, we essentially just unroll one call:
  - `f(z,8) + 7`
    becomes
    `(if z < 0 then 8 else z*f(z-1,8)) + 7`
  - Can't keep on inlining definition of `f`; will never stop!
- But can still get some benefits of inlining by slight rewriting of recursive function...

Rewriting Recursive Functions for Inlining

- Rewrite function to use a loop pre-header

```
function f(a₁,...,aₙ) = e
```

becomes

```
function f(a₁,...,aₙ) =
    let function f′(a₁,...,aₙ) = e[f↦f']
    in f′(a₁,...,aₙ)
```

- E.g., `function f(x,y) = if x < 1 then y else x * f(x-1,y)`

```
function f(x,y) =
   let function f′(x,y) = if x < 1 then y
                          else x * f′(x-1,y)
   in f′(x,y)
```

Rewriting Recursive Functions for Inlining

```
function f(x,y) =
    let function f'(x,y) = if x < 1 then y
                           else x * f'(x-1,y)
    in f'(x,y)
```

- Remove **loop-invariant arguments**
  - e.g., `y` is invariant in calls to `f'`

```
function f(x,y) =
    let function f'(x) = if x < 1 then y
                         else x * f'(x-1)
    in f'(x)
```

Rewriting Recursive Functions for Inlining

```
function f(x,y) =
    let function f'(x) = if x < 1 then y
                                else x * f'(x-1)
    in f'(x)
```

- Now inlining recursive function is more useful!
- E.g., `6+f(4,5)` becomes

```
6 + (let function f'(x) =
               if x < 1 then 5
               else x * f'(x-1)
        in f'(4))
```

When to Inline

- Inlining functions can explode the size of the code!
  - Why?
- So when to inline a function?
- Some heuristics:
  - Expand only function call sites that are called frequently
    - Determine frequency by execution profiler or by approximating statically (e.g., loop depth)
  - Expand only functions with small bodies
    - Copied body won't be much larger than code to invoke function
  - Expand functions that are called only once
    - Dead function elimination will remove the now unused function

Tail Call Elimination

- Consider the two recursive functions

```
let rec add(m,n) = if (m = 0) n else 1 + add(m-1,n)

let rec add(m,n) = if (m=0) n else add(m-1,n+1)
```

- First function: after recursive call to `add`, still have computation to do (add 1)
- Second function: after recursive call, nothing to do but return to caller

Tail Call Elimination

```
let rec add(m,n) = if (m=0) n else add(m-1,n+1)
```

- Can reuse stack frame!
  - Don't need to allocate new stack frame for recursive call
- Values of arguments (`n`, `m`) can remain in registers
- The function call becomes a single jump
  - No memory access required
- Combined with inlining, a recursive function can become as cheap as a while loop
- Even for non-recursive functions: if last statement is function call (tail call), can still reuse stack frame

Leaf Functions

- **Leaf functions** don't call other functions
  - In call tree, these are leaf nodes
- If leaf function needs only caller-save registers, don't need a stack frame at all!
  - Significant savings!

Lazy Evaluation

- In **lazy languages** (e.g., Haskell), computation is delayed until needed
- E.g.,

```
let f x y = if x < 0 then 0 else y
f -42 (fact 10000)
```

  - `fact 10000` will never be computed, since `-42 < 0`, argument `y` is never needed
- Lazy evaluation can save unnecessary computation
- But:
  - If computation has side-effects (modifying memory, failing to terminate, etc.) program behavior may be difficult to predict
  - Delayed computations that are never used may end up using a lot of memory

Summary

- We saw a collection of techniques for optimizing functional programs
  - Decurryfication
  - Inlining
  - Tail call elimination
  - Lazy evaluation
- More next week...