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
• 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...

```
Source Code
  Parsing
  Elaboration
  Lowering
  Optimization
  Code Generation
  Target Code
```

Front end

Back end
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 $f(a_1, \ldots, a_n) = e$
• 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
  $$E[e[a_1 \mapsto v_1, \ldots, a_n \mapsto v_n]]$$
  (where $e[a_1 \mapsto v_1, \ldots, a_n \mapsto v_n]$ is expression $e$ with var $a_i$ replaced with value $v_i$)
• E.g., $g(x, y) = 1 + x + y + y$
  Can rewrite $4 + g(12, 3) \times 2$ to $4 + (1 + 12 + 3 + 3) \times 2$
Inlining

• Consider the function $f(a_1, \ldots, a_n) = e$
• 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?
• What if arguments of $f$ are non-trivial?

• If $E$ is a context, we can rewrite

$$E[f(e_1, \ldots, e_n)]$$

to

$$E[\text{let } x_1 = e_1 \text{ and } \ldots \text{ and } x_n = e_n \text{ in } e[a_1 \mapsto x_1, \ldots, a_n \mapsto x_n]]$$

where $x_1, \ldots, x_n$ are fresh variables

• Note: given $\text{double}(y) = y + y$ inlining in $\text{double}(g())$ produces $\text{let } x = g() \text{ in } x + x$

does not produce $g() + g()$!

• Why is the distinction important?
Inlining recursive functions

• Consider recursive function, e.g.,
  \( f(x, y) = \begin{cases} 
  y & \text{if } x < 1 \\
  x \times f(x-1, y) & \text{else} 
  \end{cases} \)

• If we inline it, we essentially just unroll one call:
  \( f(z, 8) + 7 \)
  becomes
  \( (\text{if } z < 0 \text{ then } 8 \text{ else } z \times f(z-1, 8)) + 7 \)

• 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 \( f(a_1, \ldots, a_n) = e \)

  becomes

  function \( f(a_1, \ldots, a_n) = \)

  let function \( f'(a_1, \ldots, a_n) = e[f \mapsto f'] \)

  in \( f'(a_1, \ldots, a_n) \)

• E.g., function \( f(x,y) = \) if \( x < 1 \) then \( y \) else \( x \times f(x-1,y) \)

  function \( f(x,y) = \)

  let function \( f'(x,y) = \) if \( x < 1 \) then \( y \)

  else \( x \times f'(x-1,y) \)

  in \( f'(x,y) \)
Rewriting Recursive Functions for Inlining

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

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

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 \texttt{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.,
  \[
  \text{let } f \ x \ y = \text{if } x < 0 \\text{ then } 0 \\text{ else } y \\
  f\ -42\ (\text{fact } 10000)
  \]

  • \text{fact } 10000 \text{ will never be computed, since } -42 < 0, \text{ argument } y \text{ 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...