HARVARD
John A. Paulson
School of Engineering
and Applied Sciences

CS153: Compilers
Lecture 15: Local Optimization

Stephen Chong

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

Announcements

- Project 4 out
  - Due Thursday Oct 25 (2 days)
- Project 5 out
  - Due Tuesday Nov 13 (21 days)
- Project 6 will be released today
  - Due Tuesday Nov 20 (28 days)

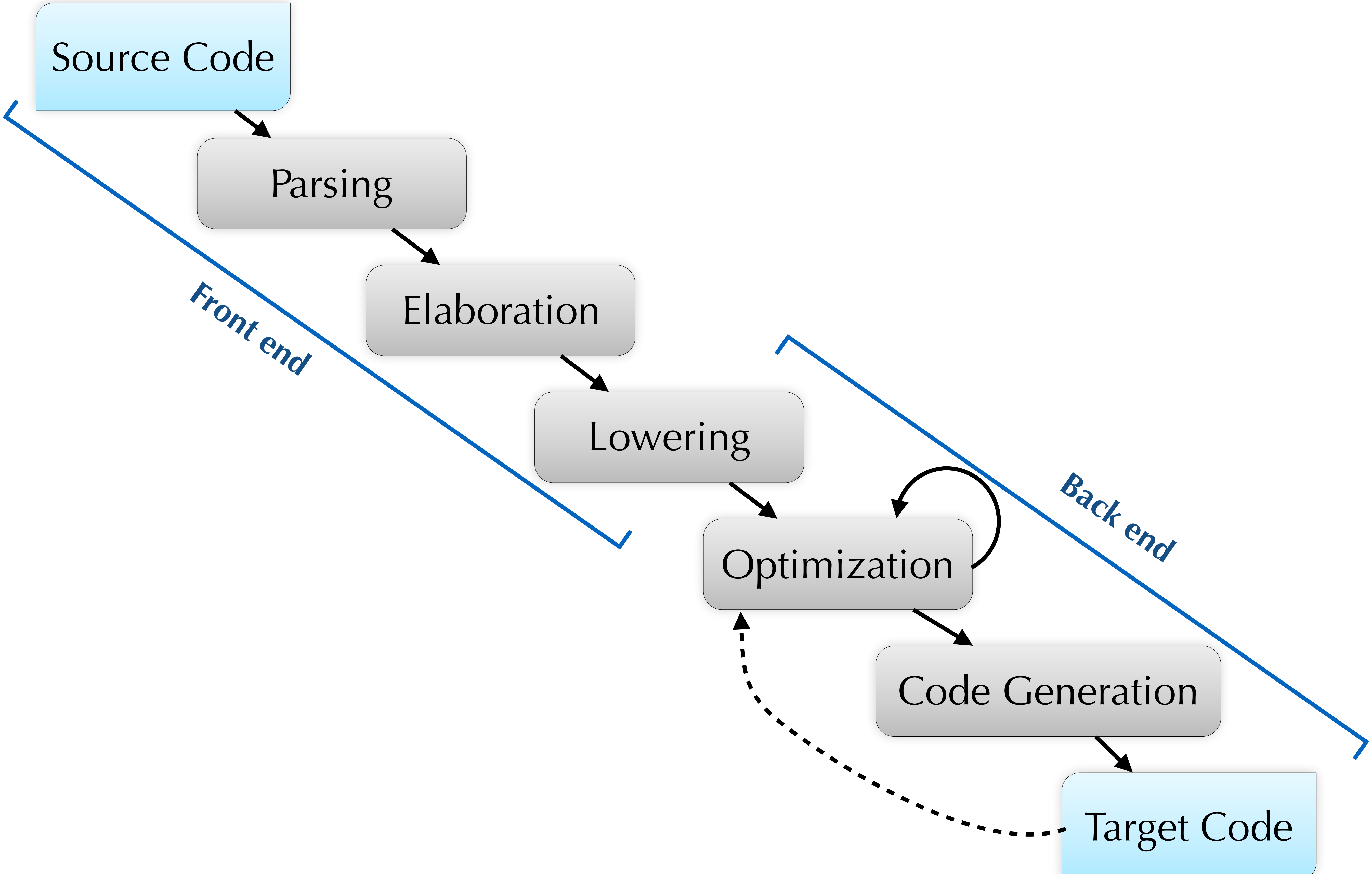
Today

- Tour of many optimizations
  - Algebraic simplification
  - Constant folding
  - Strength reduction
  - Constant propagation
  - Copy propagation
  - Dead-code elimination
  - Common sub-expression elimination
  - Loop fusion, deforestation
  - Flattening/unboxings

Optimization

- Want to rewrite code so that it's:
  - faster, smaller, consumes less power, etc.
  - while retaining the "observable behavior"
    - usually: input/output behavior
  - often need analysis to determine that a given optimization preserves behavior.
  - often need profile information to determine that a given optimization is actually an improvement.
- Often have two flavors of optimization:
  - high-level: e.g., at the AST-level (e.g., inlining)
  - low-level: e.g., right before instruction selection (e.g., register allocation)

Basic Architecture



Tour of Optimizations

- Local optimizations
  - Algebraic simplification
  - Constant folding
  - Strength reduction
  - Constant propagation
  - Copy propagation
  - Dead-code elimination
  - Common sub-expression elimination
  - Loop fusion, deforestation
- Additional optimizations
  - Inlining
  - Flattening/unboxings
  - Uncurrying
  - ...

Algebraic Simplification

- Use algebraic arithmetic identities
- E.g.,
  - `e+0` becomes `e`
  - `e*1` becomes `e`
  - `e*0` becomes `0`
  - etc.

Strength Reduction

- Replace "powerful"/expensive operations with cheaper ones
- E.g.,
  - `x*2` becomes `x+x`
  - `x div 8` becomes `x>>3`
    - On many machines bit shifting is faster than multiplication and division
  - `x*15` becomes `let t = x<<4 in t-x`

Constant Folding

- aka delta reductions
- Operations on constants can be done at compile time!
- E.g.,
  - `3+4` becomes `7`
  - `if true then s else t` becomes `s`

Copy and Constant Propagation

- If variable `x` is defined as a constant or another variable, can replace `x` with its definition
- E.g., constant propagation
  - `let x = 3 in x + x` becomes `3 + 3`
  - `let foo = 4 in foo + bar` becomes `4 + bar`
- E.g., copy propagation
  - `let x = y in x + x` becomes `y + y`

Dead Code Elimination

- Dead code = code that doesn't contribute to the program's result
- E.g.,
  - `let x = e`$_1$ `in e`$_2$ becomes `e`$_2$
    (if `x` doesn't appear in `e`$_2$)

Common Sub-Expression Elimination

- Don't need to recompute the same thing multiple times!
- Identify and remove common subexpressions
  - e.g., `f(x+y,8+x+y)` becomes

```
let t = x+y in f(t, 8+t)
```

Deforestation

- Think about the execution of `map g (map f l)`
- The first `map` produces a new list that is consumed by the second `map`
  - memory allocation, pressure on the memory bandwidth, garbage collection
- What if we could do `map (compose f g) l` instead?
- In general, functional programming produces lots of intermediate terms (trees)
- **Deforestation** is the removal of these intermediate trees
  - aka **fusion**

Unboxing

- For uniformity, we often represent all data as pointers
  - Allows functions like `map: 'a list -> ('a -> 'b) -> 'b list` to work on all data types, including `int`s, records, etc.
- Data represented by a pointer is called **boxed**
- Data represented directly in registers is **unboxed**
- **Unboxing** changes representation from pointer to value
  - In Java this is the difference between, e.g., `Integer` and `int`
- What is the benefit?
  - More efficient access to data! Can store in register rather than memory
- When is it applicable? Not applicable?
  - So long as value doesn't **escape** (i.e., need to be passed in memory to other function, caller, ...)

Unboxing Example

```
function foo(x) =
  let y = (x, 13) in
  let z = (y, 14) in
  (bar y) + #2 z
```

- Function constructs 2 pairs, `y` and `z`
- `y` escapes (as argument to function `bar`)
- `z` does not escape
- Could unbox `z` to the following (enabling further optimizations)

```
function foo(x) =
  let y = (x, 13) in
  let z1 = y in
  let z2 = 14 in
  (bar y) + z2
```

Monomorphization

- Polymorphic code works for many types
  - E.g., `map: 'a list -> ('a -> 'b) -> 'b list` works for all types `'a` and `'b`
- But code could be more efficient if it were specialized for a specific type and then optimized
- When is it applicable?
  - When we have polymorphic code
  - E.g., C++ templates
- What are the benefits?
  - In presence of dynamic dispatch, may be able to turn some into static dispatch
  - May enable optimizations like unboxing
- What are the drawbacks?
  - Potential for code bloat!

Local Optimizations

- Most of the optimizations we've just seen are local optimizations
  - They can be applied just looking locally at computation
  - No need to understand control flow
- Applying one local optimization may enable more optimizations!
- Can just keep applying local optimizations until we can't apply any more...

Optimization Example

```
let a = x ** 2 in
let b = 3 in
let c = x in
let d = c * c in
let e = b * 2 in
let f = a + d in
e * f
```

Copy and constant propagation

```
let a = x ** 2 in
let d = x * x in
let e = 3 * 2 in
let f = a + d in
e * f
```

Constant folding

```
let a = x ** 2 in
let d = x * x in
let e = 6 in
let f = a + d in
e * f
```

Strength reduction

```
let a = x * x in
let d = x * x in
let e = 6 in
let f = a + d in
e * f
```

Common sub-expression elimination

```
let a = x * x in
let d = a in
let e = 6 in
let f = a + d in
e * f
```

Copy and constant propagation

```
let a = x * x in
let f = a + a in
6 * f
```

When to Perform Local Optimization?

- Can be done at intermediate language representation and at assembly level
- Local optimizations at assembly level called **peephole optimizations**
    - Examine some small set of instructions and replace with different set
    - Often very machine specific

When is it Safe to Rewrite?

- When can we safely replace `e1` with `e2`?
- 1. when `e1 == e2` from an input/output point of view
- AND
- 2. when `e1` ≤ `e2` from our improvement metrics (e.g., performance, space, power)
  - "Optimization" is a misnomer; not producing optimal program. Improving program...

I/O Equivalence

- Consider let-reduction:
- `(let x = e1 in e2)` =?= `(e2[x↦e1])`
  where `e2[x↦e1]` is `e2` with `e1` substituted for `x`

- When does this equation hold?

Non-Examples

- `let x = print "hello“;2 in x+x`
- `let x = print "hello" in 3`
- `let x = raise Foo in 3`
- `let x = ref 3 in x := !x + 1; !x`
- `let x = print “hello” in print “world”;`
- `let x = foo() in x + x`

For ML

- `(let x = e1 in e2)` =?= `(e2[x↦e1])`
- Holds for sure when `e1` has no observable effects.

- Observable effects include:
  - diverging
  - input/output
  - allocating or reading/writing refs & arrays
  - raising an exception

Side-Effect Free by Construction

- Define a syntax for expressions that are guaranteed to be side-effect free
- So we can guarantee that (**`let x = v in e`**) == (**`e[x↦v]`**) when **`v`** is drawn from the subset of expressions:

```
v ::= i              (* constants *)
    | x              (* variables *)
    | v op v         (* binops of vals *)
    | (v,…,v)        (* tuples of vals *)
    | #i v           (* select of a val *)
    | D v            (* constructors *)
    | fun x -> e     (* functions *)
    | let x = v in v
```

- What expressions are missing from here?

Another Problem

- Variable names!
- Consider the following program

```
let x = foo() in
let y = x+x in
let x = bar() in
  y * y
```

- Let's replace `y` with `x + x`...

```
let x = foo() in
let x = bar() in
(x+x) * (x+x)
```

- Uh oh...

Variable Capture

- When substituting a value **`v`** for a variable **`x`**, we must make sure that none of the free variables in **`v`** are accidentally captured.
- A simple solution is to just rename all the variables so they are unique (throughout the program) before doing any reductions.
- Must be sure to preserve uniqueness.

Avoiding Caputre

- Returning to previous example

```
let x = foo() in
let y = x+x in
let x = bar() in
  y * y
```

- Rename variables to be unique

```
let x = foo() in
let y = x+x in
let z = bar() in
  y * y
```

- Now replacing `y` with `x + x` avoids variable capture

```
let x = foo() in
let y = x+x in
let z = bar() in
  (x+x) * (x+x)
```

Monadic Form

- We will put programs into **monadic form**
  - A syntactic form that lets us easily distinguish side-effecting expressions from pure expressions
  - Enable simpler implementations of optimizations
  - Take CS152 to find out why it's called monadic form!
- Next lecture...