IA010: Principles of Programming Languages

Optimisation

Achim Blumensath
blumens@fi.muni.cz

Faculty of Informatics, Masaryk University, Brno

Optimisation

- generally a good idea
- **slow,** currently the largest contribution to a compiler's runtime (together with type checking)
- **trade-off:** speed vs. code size
- makes debugging harder (stepping through code)
- reduces predictability (hard to predict what code is produced, optimisations can be very fragile)
- required for abstraction-heavy programming styles

Optimisation

- generally a good idea
- **slow,** currently the largest contribution to a compiler's runtime (together with type checking)
- **trade-off:** speed vs. code size
- makes debugging harder (stepping through code)
- reduces predictability (hard to predict what code is produced, optimisations can be very fragile)
- required for abstraction-heavy programming styles

Low-Level Optimisation: preserves the source code and tries to improve the translation to assembler

High-Level Optimisation: transforms the source code to make it more efficient

Optimisation: Inlining

Inlining: insert the function body at the function call

Optimisation: Inlining

Inlining: insert the function body at the function call

- avoids the overhead of a function call
- enables further optimisations
- increases code size
- hard to predict whether a function call will be inlined

Optimisation: Constant Folding and Propagation

Constant Propagation: replace variables with known values by constants

Constant Folding: evaluate operations with constant arguments

```
let x = 1;
let y = 2;
let z = x + y;
f(z)
```

```
let x = 1;
let y = 2;
let z = 1 + 2;
f(z)
```

```
let x = 1;
let y = 2;
let z = 3;
f(z)
```

```
let x = 1;
let y = 2;
let z = 3;
f(3)
```

Optimisation: Common Subexpression Elimination

Common Subexpression Elimination: compute common expressions only once

```
f(x + 1, x + 1)
```

```
let z = x + 1;
f(z, z)
```

Optimisation: Function Specialisation

Function Specialisation: generate special instances of functions with known arguments

```
let f(x, y) { ... x ... y ... }

f(u, 1)
```

```
let f(x, y) { ... x ... y ...}
let f1(x)   { ... x ... 1 ...}
f1(u)
```

Optimisation: Dead Code Elimination

Dead Code Elimination: remove unreachable code

```
let x = 1;
if x == 2 then
  f()
else
  g(x)
end
```

```
let x = 1;
g(x)
```

Optimisation: Dead Store Elimination

Dead Store Elimination: remove assignments to variables that are not used anymore

```
let x = 1;
f(x);
x := 2;
g();
```

```
let x = 1;
f(x);
g();
```

Optimisation: Code Motion

Moving Code out of Branches or Loops

```
if x > 0 then
  f(x);
  g(x);
else
  h(x);
  g(x);
end;
```

```
if x > 0 then
  f(x);
else
  h(x);
end;
g(x);
```

Moving Code into Branches

```
if x > 0 then
  f(x);
else
  g(x);
end;
if x > 0 then
  h(x);
else
  k(x);
end;
```

```
if x > 0 then
  f(x);
  if x > 0 then h(x) else k(x) end;
else
  g(x);
  if x > 0 then h(x) else k(x) end;
end
```

Optimisation: Loop Unrolling

Loop Unrolling: duplicate the body of loops

```
for i = 0 to n-1 {
  a[i] = f(i);
}
```

```
for i = 0 to n/4 - 1 {
  a[4*i]   = f(4*i);
  a[4*i+1] = f(4*i+1);
  a[4*i+2] = f(4*i+2);
  a[4*i+3] = f(4*i+3);
}
```

Dataflow Analysis

General Idea:

- compute information about each identifier
- this information is ordered (less knowledge < more knowledge)
- compute a least fixed-point by iteration:
  - start with the empty information
  - go over the whole program and add any information we can deduce
  - repeat until nothing can be added anymore

This gets more complicated, if one wants to support first-class functions ($\Rightarrow$ k-CFA algorithm).

Alias Analysis

Problem: For many optimisations we need to know which memory locations can be accessed by other pointers.

This is particularly important when deciding which variables can be kept in registers.

Solution: Use dataflow analysis to determine which values get their address taken.

```
let x = 1;
f(x);         // x = 1
let p = &x;
g(p);
h(x);         // x can have any value here.
```

Register Allocation

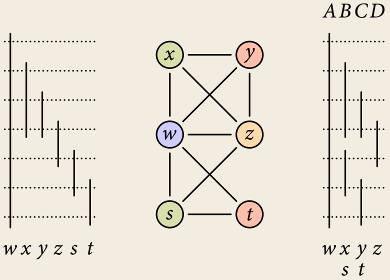
Idea: minimise the number of local variables on the stack by keeping n of them in registers.

Algorithm: reduction to the graph colouring problem
With each variable associate the interval where it is used.

vertices: variables
edges: intersecting intervals

Any n-colouring of this graph gives a valid register assignment.

```
let w = f(u);
let x = w+1;
let y = x*w;
let z = g(x,y);
let s = y+z;
let t = s-1;
let v = h(w,t);
```



Register Allocation

Idea: minimise the number of local variables on the stack by keeping n of them in registers.

Algorithm: reduction to the graph colouring problem
With each variable associate the interval where it is used.

vertices:	variables
edges:	intersecting intervals

Any n-colouring of this graph gives a valid register assignment.

Complications

- If no assignment exists, we have to split intervals.
- Assembler instructions may require arguments in specific registers.