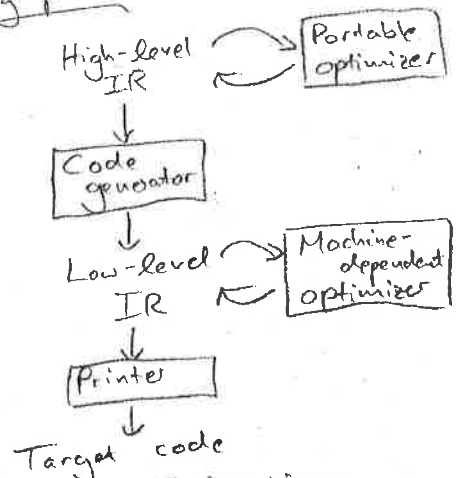
Optimization

Lecture topics

I Intro to optimizations
II Constant folding
III Data flow analysis
IV Obstacles to optimization

I Intro to optimizations

Big picture



Typical optimizations

Name	Before	After
Constant folding	h=60*60	h=3600
Copy propagation	t=h d=24*t	t=h d=24*h
Dead code elimination	t=h d=24*h return d	d=24*h return d
Common subexpression elimination	h=60*60 d=24*60*60	h=60*60 d=24*h

Algebraic simplification	while i<=n-1{
	...
	}
		while i<n{
		...
		}

Strength reduction	x=y*2 z=y/4	x=y+y z=y>>2

Loop invariant code motion	while i<=n-1{
	...
	}
		m=n-1;
		while i<=m{
		...
		}

Origins of optimization opportunities

- avoidable redundancy in source code (e.g. 24*60*60 more readable)
- inherent redundancy in source code (e.g. array bounds checks in Java)
- simpler compiler pass (e.g. spurious temporaries /copies)
synergies between optimizations

II Constant folding

NC= not constant
?=undefined

	i	n	r	t
`foo = fun(i:int)->int {`	NC	?	?	?
`n= 3*5;`		15		
`r=0;`			0	
`while i!=0 {`	NC	15	~~0~~NC	~~?~~NC
`t=7`				7
`i:=i/2;`	NC			
`if i!=0 {`				
`t:=t+n;`				22
`}`				NC
`r:=r+t;`			NC	
`}`	NC	15	NC	NC
`-> r;`				
`}`				

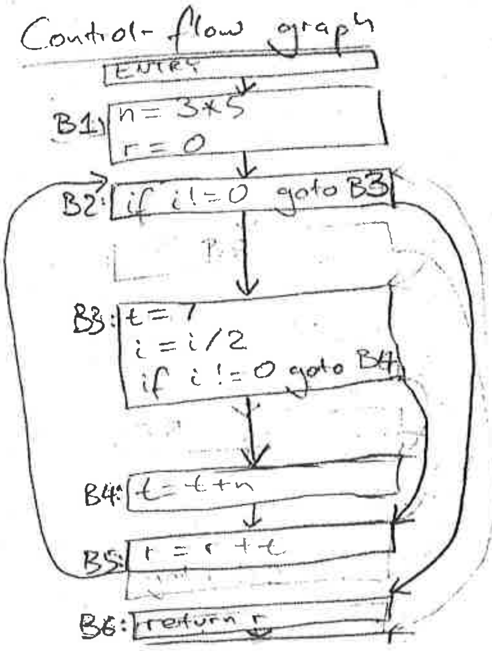
After constant propagation

```
foo = fun(i:int)->int {
    n= 15; ~ dead code
    r= 0;
    while i!=0 {
        t=7
        i:=i/2;
        if i!=0 {
            t:=22;
        }
        r:=r+t;
    }
    -> r;
}
```

Basic block

- sequence of straight-line instructions
- no internal jumps or jump targets
- vertex in control-flow graph

Control-flow graph



III Data-flow analysis

Forward analysis algorithm

```
for all blocks B, initialize B.in / B.out
worklist.put(ENTRY)
while worklist not empty {
        B = worklist.pop()
        B.in = merge P.out from predecessors
        apply transfer functions for instructions
              in B to compute B.out
        if B.out changed {
             add successors of B to worklist
        }
}
```

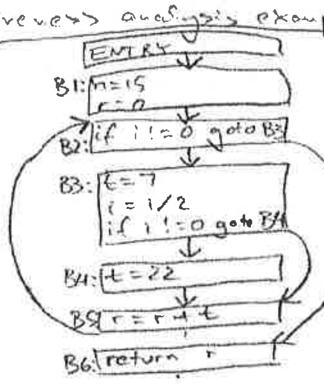
Constant folding specification

Direction:	Forwards
Info:	For each variable, NC/?/constant
Initialization:	Parameters NC, rest ?
Transfer function:	Case x = y + z: if both y/z constant: x.info = y.info + z.info else if one ?: x.info = the other else x.info = NC
Merge:	If x.info same on all predecessors x.info = that value else x.info = NC

Liveness analysis specification

Direction:	Backwards
Info:	For each variable, dead / live
Initialization:	Return value live, rest dead
Transfer function:	case x = y + z x.info = dead y.info = live z.info = live
Merge	if x live on at least one successor x.info = live else x.info = dead

Liveness analysis example



		i	n	r	t
B1	in	L	D	D	D
	out	L	D	L	D
B2	in	L	D	L	D
	out	~~D~~ L	D	L	D
B3	in	L	D	L	D
	out	L	D	L	L
B4	in	L	D	L	D
	out	L	D	L	D
B5	in	L	D	L	L
	out	L	D	L	D
B6	in	D	D	L	D
	out				

Uses of data-flow analysis

- optimization (e.g. dead-code elimination)
- register allocation (e.g. use liveness for interference graph in graph coloring)
- bug-finding (e.g. uninitialized variable)

IV Obstacles to optimization

Calls

Before

```
while i<pow(2,n) {
   ...
}
```

After

```
p=pow(2,n);
while i<p {
   ...
}
```

Only legal if ...

- n unchanged
- pow side-effect free
- pow deterministic

Solutions

- inlining
- interprocedural analysis

Pointers

Before	After
x = p.f	x = p.f
q.f = 5	q.f = 5
return p.f	return x

Only legal if ...

- p and q point to different records

Solutions:

- may-alias analysis
- scalar replacement