Chapter 10

Code generation

Course "Compiler Construction"

Martin Steffen

Spring 2018

Section

Targets

Chapter 10 “Code generation”
Course “Compiler Construction”
Martin Steffen
Spring 2018

Chapter 10

Learning Targets of Chapter “Code generation”.

1. 2AC
2. cost model
3. register allocation
4. control-flow graph
5. local liveness analysis (data flow analysis)
6. “global” liveness analysis

Chapter 10

Outline of Chapter “Code generation”.

Targets

Intro

2AC and costs of instructions

Basic blocks and control-flow graphs

Code generation algo

Global analysis

Section

Intro

Chapter 10 “Code generation”
Course “Compiler Construction”
Martin Steffen
Spring 2018

Code generation

- note: *code generation* so far: AST^{+} to intermediate code
  - three address code (3AC)
  - P-code
- ⇒ *intermediate code generation*
- i.e., we are still not there ...
- material here: based on the (old) *dragon book* [**?**] (but principles still ok)
- there is also a new edition [**?**]

Targets
Targets & Outline
Intro
2AC and costs of instructions
Basic blocks and control-flow graphs
Code generation algo
Global analysis

Intro: code generation

- goal: translate intermediate code (= 3AI-code) to machine language
- machine language/assembler:
  - even *more* restricted
  - here: 2 address code
- limited number of *registers*
- different *address modes* with different *costs* (registers vs. main memory)

Goals

- efficient code
- small code size also desirable
- but first of all: correct code

Targets

Targets & Outline

Intro

2AC and costs of instructions

Basic blocks and control-flow graphs

Code generation algo

Global analysis

Code "optimization"

- often conflicting goals
- code generation: *prime* arena for achieving *efficiency*
- optimal code: undecidable anyhow (and: don't forget there's trade-offs).
- even for many more clearly defined subproblems: *untractable*

"optimization"

interpreted as: *heuristics* to achieve "good code" (without hope for *optimal* code)

- due to importance of optimization at code generation
  - time to bring out the "heavy artillery"
  - so far: all techniques (parsing, lexing, even type checking) are computationally "easy"
  - at code generation/optmization: perhaps *invest* in agressive, computationally complex and rather advanced techniques
  - many different techniques used

Section

2AC and costs of instructions

Chapter 10 “Code generation”
Course “Compiler Construction”
Martin Steffen
Spring 2018

2-address machine code used here

- “typical” op-codes, but not a instruction set of a *concrete* machine
- two address instructions
- Note: cf. 3-address-code interpmediate representation vs. 2-address machine code
  - machine code is not lower-level/closer to HW because it has one argument less than 3AC
  - it’s just one illustrative choice
  - the new dragon book: uses 3-address-machine code (being more modern)
- 2 address machine code: closer to *CISC* architectures,
- *RISC* architectures rather use 3AC.
- translation task from IR to 3AC or 2AC: comparable challenge

2-address instructions format

Format

OP source dest

- note: *order* of arguments here
- restriction on *source* and *target*
  - register or memory cell
  - source: can additionally be a constant

Side remark: 3A machine code

Possible format

```
OP source1 source2 dest
```

- but then: what's the *difference* to 3A *intermediate* code?
- apart from a more restricted instruction set:
- restriction on the operands, for example:
  - only *one* of the arguments allowed to be a memory access
  - *no fancy addressing* modes (indirect, indexed ... see later) for memory cells, only for registers
- not "too much" memory-register traffic back and forth per machine instruction
- example:

```
&x = &y + *z
```

may be 3A-intermediate code, but not 3A-machine code

Cost model

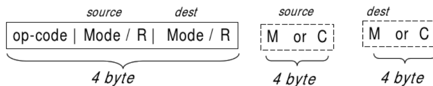
- "optimization": need some well-defined "measure" of the "quality" of the produced code
- interested here in *execution* time
- not all instructions take the same time
- estimation of execution
- factor outside our control/not part of the cost model: effect of *caching*

cost factors:

- *size* of instruction
  - it's here not about code size, but
  - instructions need to be *loaded*
  - longer instructions ⇒ perhaps longer load
- address modes (as *additional costs*: see later)
  - registers vs. main memory vs. constants
  - direct vs. indirect, or indexed access

Targets
Targets & Outline
Intro
2AC and costs of instructions
Basic blocks and control-flow graphs
Code generation algo
Global analysis

Instruction modes and additional costs



Mode	Form	Address	Added cost	
absolute	M	M	1	
register	R	R	0	
indexed	c(R)	$c + cont(R)$	1	
indirect register	*R	$cont(R)$	0	
indirect indexed	*c(R)	$cont(c + cont(R))$	1	
literal	$\#M$	the *value* M	1	only for source

- indirect: useful for elements in "records" with known off-set
- indexed: useful for slots in arrays

Examples `a := b + c`

Using registers

```
MOV b, R0   // R0 = b
ADD c, R0   // R0 = c + R0
MOV R0, a   // a = R0

cost = 6
```

Memory-memory ops

```
MOV b, a     // a = b
ADD c, a     // a = c + a

cost = 6
```

Data already in registers

```
MOV *R1, *R0 // *R0 = *R1
ADD *R2, *R1
// *R1 = *R2 + *R1

cost = 2
```

Assume `R0`, `R1`, and `R2` contain *addresses* for `a`, `b`, and `c`

Storing back to memory

```
ADD R2, R1   // R1 = R2 + R1
MOV R1, a    // a = R1

cost = 3
```

Assume `R1` and `R2` contain *values* for `b`, and `c`

Targets
Targets & Outline
Intro
2AC and costs of instructions
Basic blocks and control-flow graphs
Code generation algo
Global analysis

Section

Basic blocks and control-flow graphs

Chapter 10 “Code generation”
Course “Compiler Construction”
Martin Steffen
Spring 2018

Basic blocks

- machine code level equivalent of straight-line code
- (a largest possible) sequence of instructions without
  - jump out, or
  - jump in
- elementary unit of code analysis/optimization[1]
- amenable to analysis techniques like
  - static simulation/symbolic evaluation
  - abstract interpretation
- basic unit of code generation

[1] Those techniques can also be used across basic blocks, but then they become considerably more costly/challenging.

Targets
Targets & Outline
Intro
2AC and costs of instructions
Basic blocks and control-flow graphs
Code generation algo
Global analysis

Control-flow graphs

CFG

basically: *graph* with

- nodes = basic blocks
- edges = (potential) jumps (and “fall-throughs”)

- here (as often): CFG on 3AIC (linear intermediate code)
- also possible CFG on low-level code,
- or also:
  - CFG extracted from AST[2]
  - here: the opposite: synthesizing a CFG from the linear code
- explicit data structure (as another intermediate representation) or implicit only.

[2] See also the exam 2016.

Targets
Targets & Outline
Intro
2AC and costs of instructions
Basic blocks and control-flow graphs
Code generation algo
Global analysis

From 3AC to CFG: "partitioning algo"

- remember: 3AIC contains *labels* and (conditional) jumps

⇒ algo rather straightforward

- the only complication: some labels can be ignored
- we ignore procedure/method calls here
- concept: "leader" representing the nodes/basic blocks

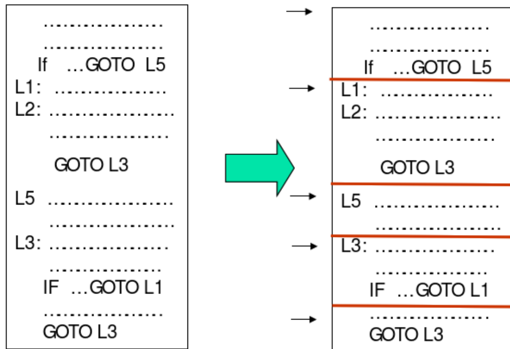
Leader

- first line is a leader
- $\mathbf{GOTO}\, i$: line labelled i is a leader
- instruction *after* a $\mathbf{GOTO}$ is a leader

Basic block

instruction sequence from (and including) one leader to (but excluding) the next leader or to the end of code

Partitioning algo

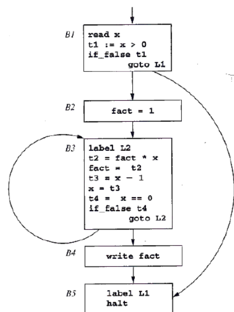


- note: no line jumps to L_2

3AIC for faculty (from before)

```
read x
t1 = x > 0
if_false t1 goto L1
fact = 1
label L2
t2 = fact * x
fact = t2
t3 = x - 1
x = t3
t4 = x == 0
if_false t4 goto L2
write fact
label L1
halt
```

Faculty: CFG



- goto/conditional goto: never *inside* block
- not every block
  - ends in a goto
  - starts with a label
- ignored here: function/method calls, i.e., focus on
- *intra-procedural* cfg

Levels of analysis

- here: *three* levels where to apply code analysis / optimizations
  1. local: per basic block (block-level)
  2. global: per function body/intra-procedural CFG
  3. inter-procedural: really global, whole-program analysis
- the "more global", the more *costly* the analysis and, especially the optimization (if done at all)

Loops in CFGs

- *loop optimization*: "loops" are thankful places for optimizations
- important for analysis to *detect* loops (in the cfg)
- importance of *loop discovery*: not too important any longer in modern languages.

Loops in a CFG vs. graph cycles

- concept of loops in CFGs not identical with cycles in a graph
- all loops are graph cycles but not vice versa

- intuitively: loops are cycles originating from source-level looping constructs ("while")
- goto's may lead to non-loop cycles in the CFG
- importants of loops: loops are "well-behaved" when considering certain optimizations/code transformations (goto's can destroy that...)

Loops in CFGs: definition

- remember: strongly connected components

Definition (Loop)

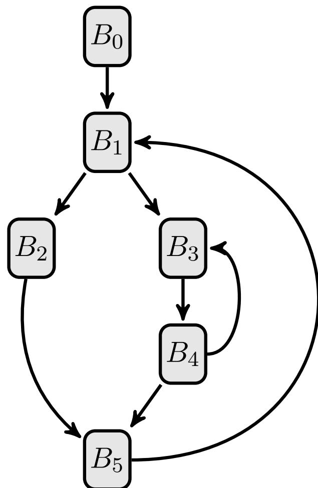
A loop L in a CFG is a collection of nodes s.t.:

- strongly connected component (with edges completely in L
- 1 (unique) *entry* node of L, i.e. no node in L has an incoming edge[3] from outside the loop *except* the *entry*

- often additional assumption/condition: "root" node of a CFG (there's only one) is *not* itself an entry of a loop

[3] alternatively: general reachability

Loop



- Loops:
  - $\{B_3, B_4\}$
  - $\{B_4, B_3, B_1, B_5, B_2\}$
- Non-loop:
  - $\{B_1, B_2, B_5\}$
- unique entry marked red

Targets
Targets & Outline
Intro
2AC and costs of instructions
Basic blocks and control-flow graphs
Code generation algo
Global analysis

Loops as fertile ground for optimizations

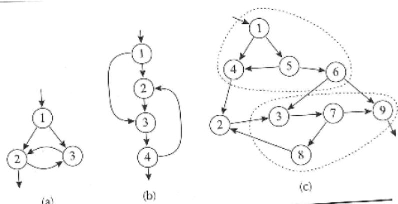
```
while (i < n) {i++; A[i] = 3*k }
```

- possible optimizations
  - move `3*k` "out" of the loop
  - put frequently used variables into *registers* while in the loop (like i)
- when moving out computation from the loop:
- put it "right in front of the loop"

⇒ add extra node/basic block in front of the *entry* of the loop[4]

[4] That's one of the motivations for unique entry.

Loop non-examples



Targets

Targets & Outline

Intro

2AC and costs of instructions

Basic blocks and control-flow graphs

Code generation algo

Global analysis

Data flow analysis in general

- general *analysis technique* working on CFGs
- many concrete forms of analyses
- such analyses: basis for (many) *optimizations*
- *data*: info stored in memory/temporaries/registers etc.
- *control*:
  - movement of the instruction pointer
  - abstractly represented by the CFG
    - inside elementary blocks: increment of the IS
    - edges of the CFG: (conditional) jumps
    - jumps together with RTE and calling convention

Data flowing from (a) to (b)

Given the control flow (normally as CFG): is it *possible* or is it *guaranteed* (“may” vs. “must” analysis) that some “data” originating at one control-flow point (a) reaches control flow point (b).

Data flow as abstraction

- data flow analysis DFA: fundamental and important *static* analysis technique
- it's impossible to decide statically if data from (a) *actually* "flows to" (b)

$\Rightarrow$ approximative (= abstraction)

- therefore: work on the CFG: if there is two options/outgoing edges: *consider both*
- Data-flow answers therefore approximatively
  - if it's *possible* that the data flows from (a) to (b)
  - it's *neccessary* or unavoidable that data flows from (a) to (b)
- for *basic blocks*: exact answers possible

Data flow analysis: Liveness

- prototypical / important data flow analysis
- especially important for register allocation

Basic question

When (at which control-flow point) can I be *sure* that I don't need a specific variable (temporary, register) any more?

- optimization: if sure that not needed in the future: register can be used otherwise

Definition (Live)

A "variable" is live at a given control-flow point if there *exists* an execution starting from there (given the level of abstraction), where the variable is *used* in the future.

Definitions and uses of variables

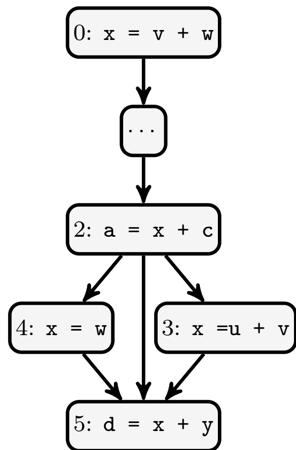
- talking about "variables": also temporary variables are meant.
- basic notions underlying most data-flow analyses (including liveness analysis)
- here: def's and uses of *variables* (or temporaries etc.)
- all data, including intermediate results) has to be stored somewhere, in variables, temporaries, etc.

Def's and uses

- a "definition" of x = assignment to x (store to x)
- a "use" of x: read content of x (load x)

- variables can occur more than once, so
- a definition/use refers to *instances* or *occurrences* of variables ("use of x in line l " or "use of x in block b ")
- same for liveness: "x is live here, but not there"

Defs, uses, and liveness



- x is "defined" (= assigned to) in 0, 3, and 4
- x is live "in" (= at the end of) block 2, as it *may* be *used* in 5
- a *non-live* variable at some point: "dead", which means: the corresponding memory can be reclaimed
- *note*: here, liveness across block-boundaries = "global" (but blocks contain only one instruction here)

Def-use or use-def analysis

- use-def: given a “use”: determine all possible “definitions”
- def-use: given a “def”: determine all possible “uses”
- for straight-line-code/inside one basic block
  - deterministic: each line has has exactly one place where a given variable has been assigned to last (or else not assigned to in the block). Equivalently for uses.
- for whole CFG:
  - approximative (“may be used in the future”)
  - more advanced techiques (caused by presence of loops/cycles)
- def-use analysis:
  - closely connected to liveness analysis (basically the same)
  - *prototypical* data-flow question (same for use-def analysis), related to many data-flow analyses (but not all)

Calculation of def/uses (or liveness . . .)

- three levels of complication
  1. inside basic block
  2. branching (but no loops)
  3. Loops
  4. [even more complex: inter-procedural analysis]

For SLC/inside basic block

- deterministic result
- simple "one-pass" treatment enough
- similar to "static simulation"
- [Remember also AG's]

For whole CFG

- iterative algo needed
- dealing with non-determinism: over-approximation
- "closure" algorithms, similar to the way e.g., dealing with *first* and *follow* sets
- = fix-point algorithms

Inside one block: optimizing use of temporaries

- simple setting: *intra*-block analysis & optimization, only
- temporaries:
  - symbolic representations to hold intermediate results
  - generated on request, assuming unbounded numbers
  - intentions: use registers
- limited about of register available (platform dependent)

Assumption about temps

- temp's *don't transfer* data across blocks ($\neq$ program var's)

$\Rightarrow$ temp's *dead* at the beginning and at the end of a block

- but: variables have to be *assumed* live at the end of a block (block-local analysis, only)

Intra-block liveness

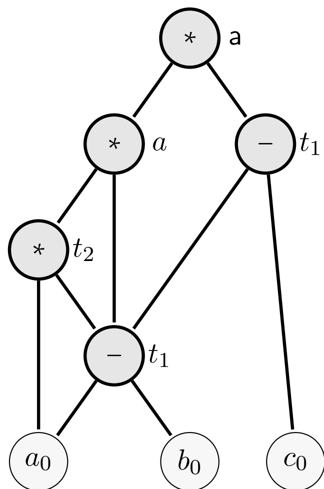
```
t1 := a - b
t2 := t1 * a
a  := t1 * t2
t1 := t1 - c
a  := t1 * a
```

- neither temp's nor vars in the example are "single assignment",
- but first occurrence of a temp in a block: a definition (but for temps it would often be the case, anyhow)
- let's call *operand:* variables or temp's
- *next use* of an operand:
- uses of operands: on the rhs's, definitions on the lhs's
- not good enough to say "t_1 is live in line 4" (why?)

Note: the 3AIC may allow also literal constants as operator arguments, they don't play a role right now.

Targets
Targets & Outline
Intro
2AC and costs of instructions
Basic blocks and control-flow graphs
Code generation algo
Global analysis

DAG of the block

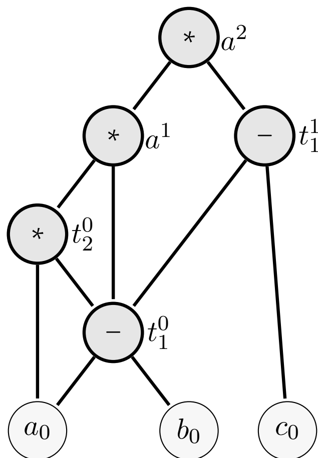


- no linear order (as in code), only *partial order*
- *the* next use: meaningless
- but: *all "next" uses* visible (if any) as "edges upwards"
- node = occurrences of a variable
- e.g.: the "lower node" for "defining"*assigning* to t_1 has *three* uses
- different "versions" (instances) of t_1

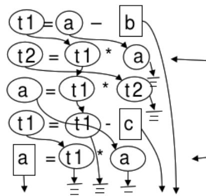
DAG / SA

SA = "single assignment"

- indexing different "versions" of right-hand sides
- often: temporaries generated as single-assignment already
- cf. also *constraints* + remember AGs



Intra-block liveness: idea of algo



- liveness-status of an operand: *different* from lhs vs. rhs in a given instruction
- informal definition: an operand is live at some occurrence, if it's used some place in the future

Definition (consider statement $x_1 := x_2 \ op \ x_3$)

- A variable x is live at the *beginning* of $x_1 := x_2 \ op \ x_3$, if
  1. if x is x_2 or x_3, or
  2. if x live at its *end*, if x and x_1 are different variables
- A variable x is live at the *end* of an instruction,
  - if it's live at *beginning of the next* instruction
  - if no next instruction
    - temp's are dead
    - user-level variables are (assumed) live

Liveness

Previous "inductive" definition

expresses liveness status of variables *before* a statement dependent on the liveness status of variables *after* a statement (and the variables used in the statement)

- *core* of a straightforward iterative algo
- simple backward scan[5]
- the algo we sketch:
  - not just boolean info (live = yes/no), instead:
  - operand live?
    - yes, and with next use inside is block (and indicate instruction where)
    - yes, but with no use inside this block
    - not live
  - even more info: not just that but indicate, where's the next use

[5] Remember: intra-block/SLC. In the presence of loops/analysing a complete CFG, a simple 1-pass does not suffice. More advanced techniques ("multiple-scans" = fixpoint calculations) are needed then.

Algo: dead or alive (binary info only)

```
// ----- initialise T   ------------------------------
  for all entries: T[i,x] := D
  except: for all variables a // but not temps
              T[n,a] := L,
//------- backward pass ------------------------------
for instruction i =  n−1 down to 0
   let current instruction at i+1: x := y op z;
      T[i.x] := D // note order; x can ``equal'' y or z
      T[i.y] := L
      T[i.z] := L
end
```

- Data structure T: table, mapping for each line/instruction i and variable: boolean status of "live"/"dead"
- represents liveness status per variable *at the end (i.e. rhs)* of that line
- basic block: n instructions, from 1 until n, where "line 0" represents the sentry imaginary line "before" the first line (no instruction in line 0)
- *backward scan* through instructions/lines from n to 0

Algo′: dead or else: alive with next use

- More refined information
- not just binary "dead-or-alive" but next-use info

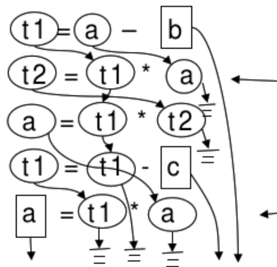
⇒ three kinds of information

1. Dead: D
2. Live:
   - with *local* line number of *next use*: $L(n)$
   - *potential* use of outside local basic block $L(\bot)$

- otherwise: basically the same algo

```
// ----- initialise T  ----------------------------
  for all entries: T[i,x] := D
  except: for all variables a // but not temps
              T[n,a] := L(⊥),
//------- backward pass ----------------------------
for instruction i = n−1 down to 0
   let current instruction at i+1: x := y op z;
      T[i,x] := D // note order; x can ``equal'' y or z
      T[i,y] := L(i+1)
      T[i,z] := L(i+1)
end
```

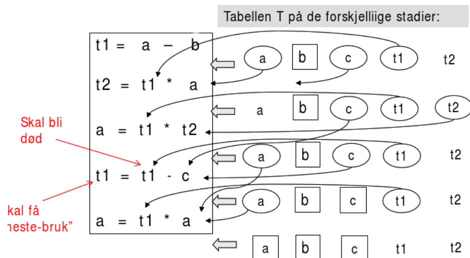
Run of the algo′



line	a	b	c	t_1
[0]	$L(1)$	$L(1)$	$L(4)$	$L(2)$
1	$L(2)$	$L(\bot)$	$L(4)$	$L(2)$
2	D	$L(\bot)$	$L(4)$	$L(3)$
3	$L(5)$	$L(\bot)$	$L(4)$	$L(4)$
4	$L(5)$	$L(\bot)$	$L(\bot)$	$L(5)$
5	$L(\bot)$	$L(\bot)$	$L(\bot)$	D

```
t1  :=  a  -  b
t2  :=  t1  *  a
a   :=  t1  *  t2
t1  :=  t1  -  c
a   :=  t1  *  a
```

Liveness algo remarks



- here: T data structure traces (L/D) status per variable × “line”
- in the *remarks* in the notat:
  - alternatively: store liveness-status *per variable* only
  - works as well for one-pass analyses (but only without loops)
- this version here: corresponds better to *global* analysis: 1 line can be seen as one small basic block

Section

Code generation algo

Chapter 10 “Code generation”
Course “Compiler Construction”
Martin Steffen
Spring 2018

Simple code generation algo

- simple algo: *intra-block* code generation
- core problem: register use
- register allocation & assignment [6]
- hold calculated values in registers longest possible
- intra-block only ⇒ at exit:
  - all *variables* stored back to main memory
  - all temps assumed "lost"
- remember: assumptions in the intra-block liveness analysis

[6] Some distinguish register *allocation*: "should the data be held in register (and how long)" vs. register *assignment*: "which of available register to use for that"

Limitations of the code generation

- local intra block:
  - no analysis across blocks
  - no procedure calls, etc.
- no complex data structures
  - arrays
  - pointers
  - ...

some limitations on how the algo itself works for one block

- for read-only variables: never put in registers, even if variable is *repeatedly* read
  - algo works only with the temps/variables given and does not come up with new ones
  - for instance: DAGs could help
- no *semantics* considered
  - like *commutativity*: $a + b$ equals $b + a$

Targets
Targets & Outline
Intro
2AC and costs of instructions
Basic blocks and control-flow graphs
Code generation algo
Global analysis

Purpose and “signature” of the *getreg* function

- one *core* of the code generation algo
- simple code-generation here ⇒ simple *getreg*

$getreg$ function

available: *liveness/next-use* info

Input: TAIC-instruction $x := y \ \mathbf{op}\ z$

Output: return *location* where x is to be stored

- location: register (if possible) or memory location

INF5110 – Compiler Construction

Targets

Targets & Outline

Intro

2AC and costs of instructions

Basic blocks and control-flow graphs

Code generation algo

Global analysis

Coge generation invariant

it should go without saying ... :

Basic safety invariant

At each point, “live” variables (with or without next use in the current block) must exist in at least one location

- another invariant: the location returned by getreg: the one where the rhs of a 3AIC assignment ends up

Register and address descriptors

- code generation/$getreg$: keep track of
  1. register contents
  2. addresses for names

Register descriptor

- tracking current "content" of reg's (if any)
- consulted when new reg needed
- as said: at block entry, assume all regs unused

Address descriptor

- tracking location(s) where current value of name can be found
- possible locations: register, stack location, main memory
- >1 location possible (but not due to overapproximation, exact tracking)

Targets
Targets & Outline
Intro
2AC and costs of instructions
Basic blocks and control-flow graphs
Code generation algo
Global analysis

Code generation algo for $x := y \text{ op } z$

1. determine location (preferably register) for result

```
l = getreg( ``x := y op z'')
```

2. make sure, that the value of y is in l :
   - consult address descriptor for $y \Rightarrow$ current locations l_y for y
   - choose the best location l_y from those (preferably register)
   - if value of y *not* in l, generate

```
MOV l_y , l
```

3. generate

```
OP l_z , l // l_z: a current location of z (prefer reg's)
```

   - update address descriptor $[x \mapsto_{\cup} l]$
   - if l is a reg: update reg descriptor $l \mapsto x$

4. exploit liveness/next use info: update register descriptors

Skeleton code generation algo for $x := y$ op z

```
l = getreg(``x:= y op z'') // target location for x
if l ∉ T_a(y) then let l_y ∈ T_a(y)) in emit ("MOV l_y, l");
let l_z ∈ T_a(z) in emit ("OP l_z,l");
```

- "skeleton"
  - *non-deterministic*: we ignored how to choose l_z and l_y
  - we ignore *book-keeping* in the *name* and *address* descriptor tables (⇒ step 4 also missing)
  - details of $getreg$ hidden.

Targets
Targets & Outline
Intro
2AC and costs of instructions
Basic blocks and control-flow graphs
Code generation algo
Global analysis

Non-deterministic code generation algo for $x := y$ op z

```
l = getreg(``x:= y op z'') // generate target location for x
if l ∉ T_a(y)
then let l_y ∈ T_a(y)) // pick a location for y
       in  emit (MOV l_y , l)
else skip;
let l_z ∈ T_a(z)) in emit (``OP l_z , l'');
T_a := T_a[x ↦_∪ l];
if   l is a register
then T_r := T_r[l ↦ x]
```

Exploit liveness/next use info: recycling registers

Targets
Targets & Outline
Intro
2AC and costs of instructions
Basic blocks and control-flow graphs
Code generation algo
Global analysis

- register descriptors: don't update themselves during code generation
- once set (e.g. as $R_0 \mapsto t$), the info stays, unless reset
- thus in step 4 for $z := x \ \mathbf{op}\ y$:

Code generation algo for $x := y$ op z

```
l = getreg("i: x := y op z")   // i for instructions line number/label
if l ∉ T_a(y)
then let l_y = best (T_a(y))
       in emit ("MOV l_y, l")
else skip;
let l_z = best (T_a(z))
in emit ("OP l_z,l");
T_a := T_a\(_ ↦ l);
T_a := T_a[x ↦ l];
T_r := T_r[l ↦ x];

if  ¬T_live[i,y] and T_a(y) = r then T_r := T_r\(r ↦ y)
if  ¬T_live[i,z] and T_a(z) = r then T_r := T_r\(r ↦ z)
```

To exploit liveness info by recycling reg's

if y and/or z are currently

- *not live* and are
- in *registers*,

⇒ “wipe” the info from the corresponding register descriptors

getreg algo: $x := y$ op z

- goal: return a location for x
- basically: check possibilities of register uses,
- starting with the "cheapest" option

Do the following steps, in that order

1. **in place:** if x is in a register already (and if that's fine otherwise), then return the register
2. **new register:** if there's an unsused register: return that
3. **purge filled register:** choose more or less cleverly a filled register and save its content, if needed, and return that register
4. **use main memory:** if all else fails

getreg algo: $x := y$ op z in more details

1. if
   - y in register R
   - R holds *no alternative names*
   - y is *not live* and has no next use after the 3AIC instruction
   - $\Rightarrow$ return R
2. else: if there is an empty register R': return R'
3. else: if
   - x has a next use [or operator requires a register] $\Rightarrow$
     - find an occupied register R
     - store R into M if needed (MOV R, M))
     - don't forget to update M 's address descriptor, if needed
     - return R
4. else: x not used in the block *or* no suituable occupied register can be found
   - return x as location L

- choice of purged register: *heuristics*
- remember (for step 3): registers may contain value for > 1 variable $\Rightarrow$ *multiple* MOV's

Sample TAIC

```
d := (a-b) + (a-c) + (a-c)
```

```
t := a - b
u := a - c
v := t + u
d := v + u
```

line	a	b	c	d	t	u	v
[0]	$L(1)$	$L(1)$	$L(2)$	D	D	D	D
1	$L(2)$	$L(\perp)$	$L(2)$	D	$L(3)$	D	D
2	$L(\perp)$	$L(\perp)$	$L(\perp)$	D	$L(3)$	$L(3)$	D
3	$L(\perp)$	$L(\perp)$	$L(\perp)$	D	D	$L(4)$	$L(4)$
4	$L(\perp)$	$L(\perp)$	$L(\perp)$	$L(\perp)$	D	D	D

Code sequence

	3AIC	2AC	reg. descr.		addr. descriptor						
			R_0	R_1	a	b	c	d	t	u	v
[0]			⊥	⊥	a	b	c	d	t	u	v
1	t := a − b	**MOV** a, R0	$[a]$		$[R_0]$						
		SUB b, R0	t		~~R_0~~				R_0		
2	u := a − c	**MOV** a, R1	·	$[a]$	$[R_0]$						
		SUB c, R1		u	~~R_0~~					R_1	
3	v := t + u	**ADD** R1, R0	v	·					~~R_0~~		R_0
4	d := v + u	**ADD** R1, R0	d					R_0			~~R_0~~
		MOV R0, d									
			R_i: unused		all var's in "home position"						

- address descr's: "home position" not explictely needed.
- e.g. variable a always to be found "at a ", as indicated in line "0".
- in the table: only *changes* (from top to bottom) indicated
- after line 3:
  - t dead
  - t resides in R_0 (and nothing else in R_0)
  - → reuse R_0
- Remark: info in [brackets]: "ephemeral"

Targets

Targets & Outline

Intro

2AC and costs of instructions

Basic blocks and control-flow graphs

Code generation algo

Global analysis

Section

Global analysis

Chapter 10 “Code generation”
Course “Compiler Construction”
Martin Steffen
Spring 2018

From "local" to "global" data flow analysis

- data stored in variables, and "flows from definitions to uses"
- liveness analysis
  - one *prototypical* (and important) data flow analysis
  - so far: *intra-block* = straight-line code
- related to
  - *def-use* analysis: given a "definition" of a variable at some place, where it is (potentially) used
  - *use-def*: (the inverse question, "reaching definitions"
- other similar questions:
  - has a value of an expression been calculated before ("available expressions")
  - will an expression be used in all possible branches ("very busy expressions")

Targets

Targets & Outline

Intro

2AC and costs of instructions

Basic blocks and control-flow graphs

Code generation algo

Global analysis

Global data flow analysis

- block-local
  - block-local analysis (here liveness): *exact* information possible
  - block-local liveness: *1 backward scan*
  - important use of liveness: *register allocation*, temporaries typically don't survive blocks anyway
- global: working on complete CFG

2 complications

- branching: *non-determinism*, unclear which branch is taken
- loops in the program (loops/cycles in the graph): simple *one pass* through the graph does not cut it any longer

- *exact* answers no longer possible (undecidable)

⇒ work with safe approximations

- this is: general characteristic of DFA

Generalizing block-local liveness analysis

- *assumptions* for block-local analysis
  - all program variables (assumed) *live* at the end of each basic block
  - all temps are assumed *dead* there.
- now: we do better, info across blocks

at the end of each block:

which variables may be used in subsequent block(s).

- now: re-use of temporaries (and thus corresponding registers) across blocks possible
- remember local liveness algo: determined liveness status per var/temp *at the end* of each "line/instruction"

Targets
Targets & Outline
Intro
2AC and costs of instructions
Basic blocks and control-flow graphs
Code generation algo
Global analysis

Connecting blocks in the CFG: *inLive* and *outLive*

- CFG:
  - pretty conventional graph (nodes and edges, often designated start and end node)
  - *nodes* = basic blocks = contain straight-line code (here 3AIC)
  - being conventional graphs:
    - conventional representations possible
    - E.g. nodes with lists/sets/collections of immediate *successor nodes* plus immediate *predecessor nodes*
- remember: local liveness status
  - can be different *before* and *after* one single instruction
  - liveness status *before* expressed as dependent on status *after*

  ⇒ backward scan
- Now per block: *inLive* and *outLive*

inLive and *outLive*

- tracing / approximating set of live variables[7] at the *beginning* and *end* per basic block
- *inLive* of a block: depends on
  - *outLive* of that block and
  - the SLC inside that block
- *outLive* of a block: depends on *inLive* of the *successor* blocks

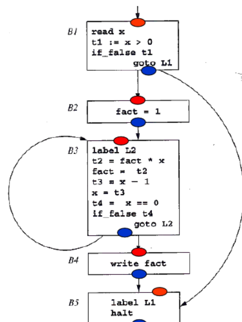
Approximation: To err on the safe side

Judging a variable (statically) live: always *safe*. Judging wrongly a variable *dead* (which actually will be used): unsafe

- goal: smallest (but safe) possible sets for *outLive* (and *inLive*)

[7] To stress "approximation": *inLive* and *outLive* contain sets of *statically* live variables. If those are dynamically live or not is undecidable.

Example: Faculty CFG



- $inLive$ and $outLive$
- picture shows arrows as *successor nodes*
- needed *predecessor nodes* (reverse arrows)

node/block	predecessors
B_1	$\emptyset$
B_2	$\{B_1\}$
B_3	$\{B_2, B_3\}$
B_4	$\{B_3\}$
B_5	$\{B_1, B_4\}$

Block local info for global liveness/data flow analysis

- 1 CFG per procedure/function/method
- as for SLC: algo works backwards
- for each block: underlying block-local liveness analysis

3-valued block local status per variable

result of block-local live variable analysis

1. *locally live* on entry: variable used (before overwritten or not)
2. *locally dead* on entry: variable overwritten (before used or not)
3. status not locally determined: variable neither assigned to nor read locally

- for efficiency: *precompute* this info, before starting the global iteration ⇒ avoid *recomputation* for blocks in loops

Targets
Targets & Outline
Intro
2AC and costs of instructions
Basic blocks and control-flow graphs
Code generation algo
Global analysis

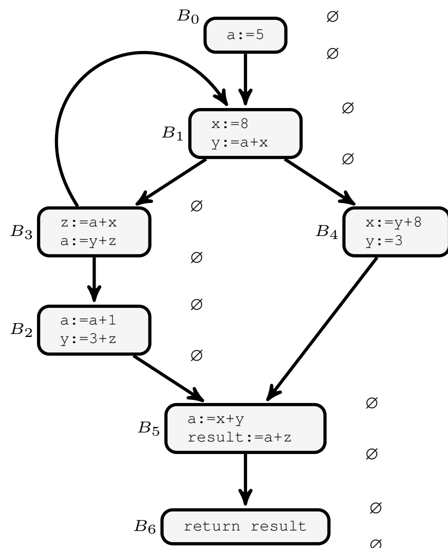
Global DFA as iterative "completion algorithm"

- different names for the general approach
  - *closure* algorithm, *saturation* algo
  - *fixpoint* iteration
- basically: a big loop with
  - iterating a step approaching an intended solution by making current approximation of the solution *larger*
  - until the solution stabilizes
- similar (for example): calculation of first- and follow-sets
- often: realized as *worklist algo*
  - named after central data-structure containing the "work-still-to-be-done"
  - here possible: worklist containing nodes untreated wrt. liveness analysis (or DFA in general)

Example

```
    a := 5
L1: x := 8
    y := a + x
    if_true x=0 goto L4
    z := a + x          // B3
    a := y + z
    if_false a=0  goto L1
    a := a + 1          // B2
    y := 3 + x
L5  a := x + y
    result := a + z
    return result       // B6
L4: a := y + 8
    y := 3
    goto L5
```

CFG: initialization



- $inLive$ and $outLive$: *initialized* to ∅ everywere
- note: start with (most) *unsafe* estimation
- extra (return) node
- but: analysis here *local per procedure*, only

Iterative algo

General schema

Initialization start with the "minimal" estimation (∅ everywhere)

Loop pick one node & update (= enlarge) liveness estimation in connection with that node

Until finish upon stabilization. no further enlargement

- order of treatment of nodes: in princple arbitrary[8]
- in tendency: following edges backwards
- comparison: for linear graphs (like inside a block):
  - no repeat-until-stabilize loop needed
  - 1 simple backward scan enough

[8]There may be more efficient and less efficient orders of treatment.

Liveness: run

B_0
```
a:=5
```
∅
∅

B_1
```
x:=8
y:=a+x
```
∅
∅

B_3
```
z:=a+x
a:=y+z
```
∅
∅

B_4
```
x:=y+8
y:=3
```
∅
∅

B_2
```
a:=a+1
y:=3+z
```
∅
∅

B_5
```
a:=x+y
result:=a+z
```
∅
∅

B_6
```
return result
```
∅
∅

Targets

Targets & Outline

Intro

2AC and costs of instructions

Basic blocks and control-flow graphs

Code generation algo

Global analysis

Liveness: run

B_0 a:=5 — $\emptyset$ / $\emptyset$

B_1 x:=8 y:=a+x — $\emptyset$ / $\emptyset$

B_3 z:=a+x a:=y+z — $\emptyset$ / $\emptyset$

B_4 x:=y+8 y:=3 — $\emptyset$ / $\emptyset$

B_2 a:=a+1 y:=3+z — $\emptyset$ / $\emptyset$

B_5 a:=x+y result:=a+z — $\emptyset$ / $\emptyset$

B_6 return result — $\{r\}$ / $\emptyset$

Targets
Targets & Outline
Intro
2AC and costs of instructions
Basic blocks and control-flow graphs
Code generation algo
Global analysis

Liveness: run

B_0 `a:=5` — $\emptyset$ / $\emptyset$

B_1 `x:=8` `y:=a+x` — $\emptyset$ / $\emptyset$

B_3 `z:=a+x` `a:=y+z` — $\emptyset$ / $\emptyset$

B_4 `x:=y+8` `y:=3` — $\emptyset$ / $\emptyset$

B_2 `a:=a+1` `y:=3+z` — $\emptyset$ / $\emptyset$

B_5 `a:=x+y` `result:=a+z` — $\emptyset$ / $\{r\}$

B_6 `return result` — $\{r\}$ / $\emptyset$

Liveness: run

B_0 `a:=5` — $\varnothing$ / $\varnothing$

B_1 `x:=8` `y:=a+x` — $\varnothing$ / $\varnothing$

B_3 `z:=a+x` `a:=y+z` — $\varnothing$ / $\varnothing$

B_4 `x:=y+8` `y:=3` — $\varnothing$ / $\varnothing$

B_2 `a:=a+1` `y:=3+z` — $\varnothing$ / $\varnothing$

B_5 `a:=x+y` `result:=a+z` — $\{x, y.z\}$ / $\{r\}$

B_6 `return result` — $\{r\}$ / $\varnothing$

Liveness: run

B_0: `a:=5` — $\emptyset$ / $\emptyset$

B_1: `x:=8`, `y:=a+x` — $\emptyset$ / $\emptyset$

B_3: `z:=a+x`, `a:=y+z` — $\emptyset$ / $\emptyset$

B_4: `x:=y+8`, `y:=3` — $\emptyset$ / $\{x, y, z\}$

B_2: `a:=a+1`, `y:=3+z` — $\emptyset$ / $\{x, y, z\}$

B_5: `a:=x+y`, `result:=a+z` — $\{x, y.z\}$ / $\{r\}$

B_6: `return result` — $\{r\}$ / $\emptyset$

Liveness: run

B_0: `a:=5` — $\varnothing$ / $\varnothing$

B_1: `x:=8`, `y:=a+x` — $\varnothing$ / $\varnothing$

B_3: `z:=a+x`, `a:=y+z` — $\varnothing$ / $\varnothing$

B_4: `x:=y+8`, `y:=3` — $\{x, y, z\}$ / $\{x, y, z\}$

B_2: `a:=a+1`, `y:=3+z` — $\varnothing$ / $\{x, y, z\}$

B_5: `a:=x+y`, `result:=a+z` — $\{x, y.z\}$ / $\{r\}$

B_6: `return result` — $\{r\}$ / $\varnothing$

Liveness: run

B_0
```
a:=5
```
$\emptyset$

$\emptyset$

B_1
```
x:=8
y:=a+x
```
$\emptyset$

$\{x, y, z\}$

B_3
```
z:=a+x
a:=y+z
```
$\emptyset$

$\emptyset$

B_4
```
x:=y+8
y:=3
```
$\{x, y, z\}$

$\{x, y, z\}$

B_2
```
a:=a+1
y:=3+z
```
$\emptyset$

$\{x, y, z\}$

B_5
```
a:=x+y
result:=a+z
```
$\{x, y.z\}$

$\{r\}$

B_6
```
return result
```
$\{r\}$

$\emptyset$

INF5110 – Compiler Construction

Targets

Targets & Outline

Intro

2AC and costs of instructions

Basic blocks and control-flow graphs

Code generation algo

Global analysis

Liveness: run

B_0
```
a:=5
```
$\emptyset$ / $\emptyset$

B_1
```
x:=8
y:=a+x
```
$\emptyset$ / $\{x, y, z\}$

B_3
```
z:=a+x
a:=y+z
```
$\emptyset$ / $\emptyset$

B_4
```
x:=y+8
y:=3
```
$\{x, y, z\}$ / $\{x, y, z\}$

B_2
```
a:=a+1
y:=3+z
```
$\{a, x, z\}$ / $\{x, y, z\}$

B_5
```
a:=x+y
result:=a+z
```
$\{x, y.z\}$ / $\{r\}$

B_6
```
return result
```
$\{r\}$ / $\emptyset$

Liveness: run

B_0 `a:=5` — $\varnothing$ / $\varnothing$

B_1 `x:=8` `y:=a+x` — $\varnothing$ / $\{x, y, z\}$

B_3 `z:=a+x` `a:=y+z` — $\varnothing$ / $\{a, z, x\}$

B_4 `x:=y+8` `y:=3` — $\{x, y, z\}$ / $\{x, y, z\}$

B_2 `a:=a+1` `y:=3+z` — $\{a, x, z\}$ / $\{x, y, z\}$

B_5 `a:=x+y` `result:=a+z` — $\{x, y.z\}$ / $\{r\}$

B_6 `return result` — $\{r\}$ / $\varnothing$

Liveness: run

B_0 `a:=5` — $\emptyset$ / $\emptyset$

B_1 `x:=8` `y:=a+x` — $\emptyset$ / $\{x, y, z\}$

B_3 `z:=a+x` `a:=y+z` — $\{a, x, y\}$ / $\{a, z, x\}$

B_4 `x:=y+8` `y:=3` — $\{x, y, z\}$ / $\{x, y, z\}$

B_2 `a:=a+1` `y:=3+z` — $\{a, x, z\}$ / $\{x, y, z\}$

B_5 `a:=x+y` `result:=a+z` — $\{x, y.z\}$ / $\{r\}$

B_6 `return result` — $\{r\}$ / $\emptyset$

INF5110 – Compiler Construction

Targets

Targets & Outline

Intro

2AC and costs of instructions

Basic blocks and control-flow graphs

Code generation algo

Global analysis

Liveness: run

B_0
```
a:=5
```
$\emptyset$

$\emptyset$

B_1
```
x:=8
y:=a+x
```
$\emptyset$

$\{a, x, y, z\}$

B_3
```
z:=a+x
a:=y+z
```
$\{a, x, y\}$

$\{a, z, x\}$

B_4
```
x:=y+8
y:=3
```
$\{x, y, z\}$

$\{x, y, z\}$

B_2
```
a:=a+1
y:=3+z
```
$\{a, x, z\}$

$\{x, y, z\}$

B_5
```
a:=x+y
result:=a+z
```
$\{x, y.z\}$

$\{r\}$

B_6
```
return result
```
$\{r\}$

$\emptyset$

Targets
Targets & Outline
Intro
2AC and costs of instructions
Basic blocks and control-flow graphs
Code generation algo
Global analysis

Liveness: run

B_0
```
a:=5
```
$\emptyset$

$\emptyset$

B_1
```
x:=8
y:=a+x
```
$\{a, z\}$

$\{a, x, y, z\}$

B_3
```
z:=a+x
a:=y+z
```
$\{a, x, y\}$

$\{a, z, x\}$

B_4
```
x:=y+8
y:=3
```
$\{x, y, z\}$

$\{x, y, z\}$

B_2
```
a:=a+1
y:=3+z
```
$\{a, x, z\}$

$\{x, y, z\}$

B_5
```
a:=x+y
result:=a+z
```
$\{x, y.z\}$

$\{r\}$

B_6
```
return result
```
$\{r\}$

$\emptyset$

Liveness: run

B_0 `a:=5` — $\varnothing$ / $\{a, z\}$

B_1 `x:=8` `y:=a+x` — $\{a, z\}$ / $\{a, x, y, z\}$

B_3 `z:=a+x` `a:=y+z` — $\{a, x, y\}$ / $\{a, z, x\}$

B_4 `x:=y+8` `y:=3` — $\{x, y, z\}$ / $\{x, y, z\}$

B_2 `a:=a+1` `y:=3+z` — $\{a, x, z\}$ / $\{x, y, z\}$

B_5 `a:=x+y` `result:=a+z` — $\{x, y.z\}$ / $\{r\}$

B_6 `return result` — $\{r\}$ / $\varnothing$

Liveness: run

B_0

```
a:=5
```

$\{z\}$ $\{a, z\}$

B_1

```
x:=8
y:=a+x
```

$\{a, z\}$ $\{a, x, y, z\}$

B_3

```
z:=a+x
a:=y+z
```

$\{a, x, y\}$ $\{a, z, x\}$

B_4

```
x:=y+8
y:=3
```

$\{x, y, z\}$ $\{x, y, z\}$

B_2

```
a:=a+1
y:=3+z
```

$\{a, x, z\}$ $\{x, y, z\}$

B_5

```
a:=x+y
result:=a+z
```

$\{x, y.z\}$ $\{r\}$

B_6

```
return result
```

$\{r\}$ $\varnothing$

INF5110 – Compiler Construction

Targets

Targets & Outline

Intro

2AC and costs of instructions

Basic blocks and control-flow graphs

Code generation algo

Global analysis

Liveness example: remarks

- the shown traversal strategy is (cleverly) backwards
- example resp. example run simplistic:
- the *loop* (and the choice of "evaluation" order):

"harmless loop"

after having updated the $outLive$ info for B_1 following the edge from B_3 to B_1 *backwards* (propagating flow from B_1 back to B_3) does not increase the current solution for B_3

- no need (in this particular order) for continuing the iterative search for stabilization
- in other examples: loop iteration cannot be avoided
- note also: end result (after stabilization) independent from evaluation order! (only some strategies may stabilize faster...)

Targets

Targets & Outline

Intro

2AC and costs of instructions

Basic blocks and control-flow graphs

Code generation algo

Global analysis

Another, more interesting, example

B_0
```
x:=5
y:=a-1
```
∅ ∅

B_1
```
x:=y+8
y:=a+x
```
∅ ∅

B_3
```
a:=y+z
```
∅ ∅

B_4
```
x:=y+x
y:=3
```
∅ ∅

B_2
```
y:=3+z
```
∅ ∅

B_5
```
a:=x+y
result:=a+1
```
∅ ∅

B_6
```
return result
```
∅ ∅

INF5110 – Compiler Construction

Targets

Targets & Outline

Intro

2AC and costs of instructions

Basic blocks and control-flow graphs

Code generation algo

Global analysis

Another, more interesting, example

B_0
```
x:=5
y:=a-1
```
$\varnothing$ $\varnothing$

B_1
```
x:=y+8
y:=a+x
```
$\varnothing$ $\varnothing$

B_3
```
a:=y+z
```
$\varnothing$ $\varnothing$

B_4
```
x:=y+x
y:=3
```
$\varnothing$ $\varnothing$

B_2
```
y:=3+z
```
$\varnothing$ $\varnothing$

B_5
```
a:=x+y
result:=a+1
```
$\varnothing$ $\varnothing$

B_6
```
return result
```
$\{r\}$ $\varnothing$

INF5110 – Compiler Construction

Targets

Targets & Outline

Intro

2AC and costs of instructions

Basic blocks and control-flow graphs

Code generation algo

Global analysis

Another, more interesting, example

B_0

```
x:=5
y:=a-1
```

∅

∅

B_1

```
x:=y+8
y:=a+x
```

∅

∅

B_3

```
a:=y+z
```

∅

∅

B_4

```
x:=y+x
y:=3
```

∅

∅

B_2

```
y:=3+z
```

∅

∅

B_5

```
a:=x+y
result:=a+1
```

∅

$\{r\}$

B_6

```
return result
```

$\{r\}$

∅

Targets

Targets & Outline

Intro

2AC and costs of instructions

Basic blocks and control-flow graphs

Code generation algo

Global analysis

Another, more interesting, example

```
B0: x:=5
    y:=a-1
B1: x:=y+8
    y:=a+x
B3: a:=y+z
B2: y:=3+z
B4: x:=y+x
    y:=3
B5: a:=x+y
    result:=a+1
B6: return result
```

Annotations: B_0: $\emptyset$, $\emptyset$; B_1: $\emptyset$, $\emptyset$; B_3: $\emptyset$, $\emptyset$; B_2: $\emptyset$, $\emptyset$; B_4: $\emptyset$, $\emptyset$; B_5: $\{x, y\}$, $\{r\}$; B_6: $\{r\}$, $\emptyset$

INF5110 – Compiler Construction

Targets

Targets & Outline

Intro

2AC and costs of instructions

Basic blocks and control-flow graphs

Code generation algo

Global analysis

Another, more interesting, example

B_0: `x:=5`, `y:=a-1` — before: $\varnothing$, after: $\varnothing$

B_1: `x:=y+8`, `y:=a+x` — before: $\varnothing$, after: $\varnothing$

B_3: `a:=y+z` — before: $\varnothing$, after: $\varnothing$

B_4: `x:=y+x`, `y:=3` — before: $\varnothing$, after: $\{x, y\}$

B_2: `y:=3+z` — before: $\varnothing$, after: $\{x, y\}$

B_5: `a:=x+y`, `result:=a+1` — before: $\{x, y\}$, after: $\{r\}$

B_6: `return result` — before: $\{r\}$, after: $\varnothing$

Another, more interesting, example

B_0
```
x:=5
y:=a-1
```
$\emptyset$ / $\emptyset$

B_1
```
x:=y+8
y:=a+x
```
$\emptyset$ / $\emptyset$

B_3
```
a:=y+z
```
$\emptyset$ / $\emptyset$

B_4
```
x:=y+x
y:=3
```
$\{x, y\}$ / $\{x, y\}$

B_2
```
y:=3+z
```
$\emptyset$ / $\{x, y\}$

B_5
```
a:=x+y
result:=a+1
```
$\{x, y\}$ / $\{r\}$

B_6
```
return result
```
$\{r\}$ / $\emptyset$

INF5110 – Compiler Construction

Targets

Targets & Outline

Intro

2AC and costs of instructions

Basic blocks and control-flow graphs

Code generation algo

Global analysis

Another, more interesting, example

B_0 — $\varnothing$
```
x:=5
y:=a-1
```
$\varnothing$

B_1 — $\varnothing$
```
x:=y+8
y:=a+x
```
$\{x, y\}$

B_3 — $\varnothing$
```
a:=y+z
```
$\varnothing$

B_4 — $\{x, y\}$
```
x:=y+x
y:=3
```
$\{x, y\}$

B_2 — $\varnothing$
```
y:=3+z
```
$\{x, y\}$

B_5 — $\{x, y\}$
```
a:=x+y
result:=a+1
```
$\{r\}$

B_6 — $\{r\}$
```
return result
```
$\varnothing$

Targets
Targets & Outline
Intro
2AC and costs of instructions
Basic blocks and control-flow graphs
Code generation algo
Global analysis

Another, more interesting, example

B_0: $\emptyset$
```
x:=5
y:=a-1
```
$\emptyset$

B_1: $\{y, a\}$
```
x:=y+8
y:=a+x
```
$\{x, y\}$

B_3: $\emptyset$
```
a:=y+z
```
$\emptyset$

B_4: $\{x, y\}$
```
x:=y+x
y:=3
```
$\{x, y\}$

B_2: $\emptyset$
```
y:=3+z
```
$\{x, y\}$

B_5: $\{x, y\}$
```
a:=x+y
result:=a+1
```
$\{r\}$

B_6: $\{r\}$
```
return result
```
$\emptyset$

Targets
Targets & Outline
Intro
2AC and costs of instructions
Basic blocks and control-flow graphs
Code generation algo
Global analysis

Another, more interesting, example

B_0 — $\emptyset$

```
x:=5
y:=a-1
```

$\{y, a\}$

B_1 — $\{y, a\}$

```
x:=y+8
y:=a+x
```

$\{x, y\}$

B_3 — $\emptyset$

```
a:=y+z
```

$\emptyset$

B_4 — $\{x, y\}$

```
x:=y+x
y:=3
```

$\{x, y\}$

B_2 — $\emptyset$

```
y:=3+z
```

$\{x, y, a\}$

B_5 — $\{x, y\}$

```
a:=x+y
result:=a+1
```

$\{r\}$

B_6 — $\{r\}$

```
return result
```

$\emptyset$

Targets
Targets & Outline
Intro
2AC and costs of instructions
Basic blocks and control-flow graphs
Code generation algo
Global analysis

Another, more interesting, example

B_0 $\{a\}$

```
x:=5
y:=a-1
```

$\{y, a\}$

$\{y, a\}$

B_1

```
x:=y+8
y:=a+x
```

$\{x, y\}$

$\varnothing$

B_3

```
a:=y+z
```

$\varnothing$

$\{x, y\}$

B_4

```
x:=y+x
y:=3
```

$\{x, y\}$

$\varnothing$

B_2

```
y:=3+z
```

$\{x, y, a\}$

$\{x, y\}$

B_5

```
a:=x+y
result:=a+1
```

$\{r\}$

$\{r\}$

B_6

```
return result
```

$\varnothing$

Targets
Targets & Outline
Intro
2AC and costs of instructions
Basic blocks and control-flow graphs
Code generation algo
Global analysis

Another, more interesting, example

B_0: `x:=5`, `y:=a-1` — in: $\{a\}$, out: $\{y, a\}$

B_1: `x:=y+8`, `y:=a+x` — in: $\{y, a\}$, out: $\{a, x, y\}$

B_3: `a:=y+z` — in: $\varnothing$, out: $\varnothing$

B_4: `x:=y+x`, `y:=3` — in: $\{x, y\}$, out: $\{x, y\}$

B_2: `y:=3+z` — in: $\{x, z, a\}$, out: $\{x, y, a\}$

B_5: `a:=x+y`, `result:=a+1` — in: $\{x, y\}$, out: $\{r\}$

B_6: `return result` — in: $\{r\}$, out: $\varnothing$

Targets
Targets & Outline
Intro
2AC and costs of instructions
Basic blocks and control-flow graphs
Code generation algo
Global analysis

Another, more interesting, example

B_0: $\{a\}$
```
x:=5
y:=a-1
```
$\{y, a\}$

B_1: $\{y, a\}$
```
x:=y+8
y:=a+x
```
$\{a, x, y\}$

$\emptyset$

B_3:
```
a:=y+z
```
$\{x, z, a\}$

B_4: $\{x, y\}$
```
x:=y+x
y:=3
```
$\{x, y\}$

B_2: $\{x, z, a\}$
```
y:=3+z
```
$\{x, y, a\}$

B_5: $\{x, y\}$
```
a:=x+y
result:=a+1
```
$\{r\}$

B_6: $\{r\}$
```
return result
```
$\emptyset$

Targets
Targets & Outline
Intro
2AC and costs of instructions
Basic blocks and control-flow graphs
Code generation algo
Global analysis

Another, more interesting, example

B_0

```
x:=5
y:=a-1
```

$\{a\}$ / $\{y, a\}$

B_1

```
x:=y+8
y:=a+x
```

$\{y, a\}$ / $\{a, x, y\}$

B_3

```
a:=y+z
```

$\{x, y, z\}$ / $\{x, z, a\}$

B_4

```
x:=y+x
y:=3
```

$\{x, y\}$ / $\{x, y\}$

B_2

```
y:=3+z
```

$\{x, z, a\}$ / $\{x, y, a\}$

B_5

```
a:=x+y
result:=a+1
```

$\{x, y\}$ / $\{r\}$

B_6

```
return result
```

$\{r\}$ / $\varnothing$

Another, more interesting, example

INF5110 – Compiler Construction

Targets

Targets & Outline

Intro

2AC and costs of instructions

Basic blocks and control-flow graphs

Code generation algo

Global analysis

Another, more interesting, example

B_0 $\{a\}$

```
x:=5
y:=a-1
```

$\{y, a\}$

B_1 $\{y, z, a\}$

```
x:=y+8
y:=a+x
```

$\{a, x, y, z\}$

B_3 $\{x, y, z\}$

```
a:=y+z
```

$\{x, z, a\}$

B_4 $\{x, y\}$

```
x:=y+x
y:=3
```

$\{x, y\}$

B_2 $\{x, z, a\}$

```
y:=3+z
```

$\{x, y, a\}$

B_5 $\{x, y\}$

```
a:=x+y
result:=a+1
```

$\{r\}$

B_6 $\{r\}$

```
return result
```

$\varnothing$

Targets

Targets & Outline

Intro

2AC and costs of instructions

Basic blocks and control-flow graphs

Code generation algo

Global analysis

Another, more interesting, example

B_0 $\{a\}$

```
x:=5
y:=a-1
```

$\{y, z, a\}$

$\{y, z, a\}$

B_1

```
x:=y+8
y:=a+x
```

$\{a, x, y, z\}$

$\{x, y, z\}$

B_3

```
a:=y+z
```

$\{x, z, a\}$

$\{x, y\}$

B_4

```
x:=y+x
y:=3
```

$\{x, y\}$

$\{x, z, a\}$

B_2

```
y:=3+z
```

$\{x, y, z, a\}$

$\{x, y\}$

B_5

```
a:=x+y
result:=a+1
```

$\{r\}$

$\{r\}$

B_6

```
return result
```

$\varnothing$

Targets
Targets & Outline
Intro
2AC and costs of instructions
Basic blocks and control-flow graphs
Code generation algo
Global analysis

Another, more interesting, example

B_0 $\{a, z\}$

```
x:=5
y:=a-1
```

$\{y, z, a\}$

$\{y, z, a\}$

B_1

```
x:=y+8
y:=a+x
```

$\{a, x, y, z\}$

B_3

```
a:=y+z
```

$\{x, y, z\}$

$\{x, z, a\}$

B_4

```
x:=y+x
y:=3
```

$\{x, y\}$

$\{x, y\}$

B_2

```
y:=3+z
```

$\{x, z, a\}$

$\{x, y, z, a\}$

B_5

```
a:=x+y
result:=a+1
```

$\{x, y\}$

$\{r\}$

B_6

```
return result
```

$\{r\}$

$\varnothing$

Example remarks

- loop: this time leads to updating estimation more than once
- evaluation order not chose ideally

Precomputing the block-local "liveness effects"

- *precomputation* of the relevant info: efficiency
- traditionally: represented as *kill* and *generate* information
- here (for liveness)
  1. kill: variable instances, which are overwritten
  2. generate: variables used in the block (before overwritten)
  3. rests: all other variables won't change their status

Constraint per basic block (transfer function)

$$inLive = outLive \setminus kill(B) \cup generate(B)$$

- note:
  - order of kill and generate in above's equation
  - a variable killed in a block may be "revived" in a block
- simplest (one line) example: `x := x +1`

Example once again: kill and gen

B_0 k: $\{x, y\}$, g: $\{a\}$ — $\emptyset$, $\emptyset$

B_1 k: $\{x, y\}$, g: $\{a, y\}$ — $\emptyset$, $\emptyset$

B_3 k: $\{a\}$, g: $\{y, z\}$ — $\emptyset$, $\emptyset$

B_4 k: $\{x, y\}$, g: $\{x, y\}$ — $\emptyset$, $\emptyset$

B_2 k: $\{y\}$, g: $\{z\}$ — $\emptyset$, $\emptyset$

B_5 k: $\{r, a\}$, g: $\{x, y\}$ — $\emptyset$, $\emptyset$

B_6 k: $\{\}$, g: $\{r\}$ — $\emptyset$, $\emptyset$

Targets
Targets & Outline
Intro
2AC and costs of instructions
Basic blocks and control-flow graphs
Code generation algo
Global analysis

Example once again: kill and gen

B_0 k: $\{x, y\}$, g: $\{a\}$ — $\varnothing$, $\varnothing$

B_1 k: $\{x, y\}$, g: $\{a, y\}$ — $\varnothing$, $\varnothing$

B_3 k: $\{a\}$, g: $\{y, z\}$ — $\varnothing$, $\varnothing$

B_4 k: $\{x, y\}$, g: $\{x, y\}$ — $\varnothing$, $\varnothing$

B_2 k: $\{y\}$, g: $\{z\}$ — $\varnothing$, $\varnothing$

B_5 k: $\{r, a\}$, g: $\{x, y\}$ — $\varnothing$, $\varnothing$

B_6 k: $\{\}$, g: $\{r\}$ — $\{r\}$, $\varnothing$

Example once again: kill and gen

B_0 k: $\{x, y\}$, g: $\{a\}$ — $\varnothing$ / $\varnothing$

B_1 k: $\{x, y\}$, g: $\{a, y\}$ — $\varnothing$ / $\varnothing$

B_3 k: $\{a\}$, g: $\{y, z\}$ — $\varnothing$ / $\varnothing$

B_4 k: $\{x, y\}$, g: $\{x, y\}$ — $\varnothing$ / $\varnothing$

B_2 k: $\{y\}$, g: $\{z\}$ — $\varnothing$ / $\varnothing$

B_5 k: $\{r, a\}$, g: $\{x, y\}$ — $\varnothing$ / $\{r\}$

B_6 k: $\{\}$, g: $\{r\}$ — $\{r\}$ / $\varnothing$

Targets
Targets & Outline
Intro
2AC and costs of instructions
Basic blocks and control-flow graphs
Code generation algo
Global analysis

Example once again: kill and gen

B_0 k: $\{x, y\}$, g: $\{a\}$ — $\emptyset$, $\emptyset$

B_1 k: $\{x, y\}$, g: $\{a, y\}$ — $\emptyset$, $\emptyset$

B_3 k: $\{a\}$, g: $\{y, z\}$ — $\emptyset$, $\emptyset$

B_4 k: $\{x, y\}$, g: $\{x, y\}$ — $\emptyset$, $\emptyset$

B_2 k: $\{y\}$, g: $\{z\}$ — $\emptyset$, $\emptyset$

B_5 k: $\{r, a\}$, g: $\{x, y\}$ — $\{x, y\}$, $\{r\}$

B_6 k: $\{\}$, g: $\{r\}$ — $\{r\}$, $\emptyset$

Example once again: kill and gen

B_0 k: $\{x, y\}$, g: $\{a\}$ — $\varnothing$ / $\varnothing$

B_1 k: $\{x, y\}$, g: $\{a, y\}$ — $\varnothing$ / $\varnothing$

B_3 k: $\{a\}$, g: $\{y, z\}$ — $\varnothing$ / $\varnothing$

B_4 k: $\{x, y\}$, g: $\{x, y\}$ — $\varnothing$ / $\{x, y\}$

B_2 k: $\{y\}$, g: $\{z\}$ — $\varnothing$ / $\{x, y\}$

B_5 k: $\{r, a\}$, g: $\{x, y\}$ — $\{x, y\}$ / $\{r\}$

B_6 k: $\{\}$, g: $\{r\}$ — $\{r\}$ / $\varnothing$

Example once again: kill and gen

B_0 k: $\{x, y\}$, g: $\{a\}$ — $\varnothing$ / $\varnothing$

B_1 k: $\{x, y\}$, g: $\{a, y\}$ — $\varnothing$ / $\varnothing$

B_3 k: $\{a\}$, g: $\{y, z\}$ — $\varnothing$ / $\varnothing$

B_4 k: $\{x, y\}$, g: $\{x, y\}$ — $\{x, y\}$ / $\{x, y\}$

B_2 k: $\{y\}$, g: $\{z\}$ — $\varnothing$ / $\{x, y\}$

B_5 k: $\{r, a\}$, g: $\{x, y\}$ — $\{x, y\}$ / $\{r\}$

B_6 k: $\{\}$, g: $\{r\}$ — $\{r\}$ / $\varnothing$

Targets

Targets & Outline

Intro

2AC and costs of instructions

Basic blocks and control-flow graphs

Code generation algo

Global analysis

Example once again: kill and gen

B_0 k: $\{x, y\}$, g: $\{a\}$ — $\varnothing$ / $\varnothing$

B_1 k: $\{x, y\}$, g: $\{a, y\}$ — $\varnothing$ / $\{x, y\}$

B_3 k: $\{a\}$, g: $\{y, z\}$ — $\varnothing$ / $\varnothing$

B_4 k: $\{x, y\}$, g: $\{x, y\}$ — $\{x, y\}$ / $\{x, y\}$

B_2 k: $\{y\}$, g: $\{z\}$ — $\varnothing$ / $\{x, y\}$

B_5 k: $\{r, a\}$, g: $\{x, y\}$ — $\{x, y\}$ / $\{r\}$

B_6 k: $\{\}$, g: $\{r\}$ — $\{r\}$ / $\varnothing$

Example once again: kill and gen

Example once again: kill and gen

B_0 k: $\{x, y\}$, g: $\{a\}$ — $\varnothing$ / $\{y, a\}$

B_1 k: $\{x, y\}$, g: $\{a, y\}$ — $\{y, a\}$ / $\{x, y\}$

B_3 k: $\{a\}$, g: $\{y, z\}$ — $\varnothing$ / $\varnothing$

B_4 k: $\{x, y\}$, g: $\{x, y\}$ — $\{x, y\}$ / $\{x, y\}$

B_2 k: $\{y\}$, g: $\{z\}$ — $\varnothing$ / $\{x, y, a\}$

B_5 k: $\{r, a\}$, g: $\{x, y\}$ — $\{x, y\}$ / $\{r\}$

B_6 k: $\{\}$, g: $\{r\}$ — $\{r\}$ / $\varnothing$

Example once again: kill and gen

B_0 k: $\{x, y\}$, g: $\{a\}$ — $\{a\}$, $\{y, a\}$

B_1 k: $\{x, y\}$, g: $\{a, y\}$ — $\{y, a\}$, $\{x, y\}$

B_3 k: $\{a\}$, g: $\{y, z\}$ — $\varnothing$, $\varnothing$

B_4 k: $\{x, y\}$, g: $\{x, y\}$ — $\{x, y\}$, $\{x, y\}$

B_2 k: $\{y\}$, g: $\{z\}$ — $\varnothing$, $\{x, y, a\}$

B_5 k: $\{r, a\}$, g: $\{x, y\}$ — $\{x, y\}$, $\{r\}$

B_6 k: $\{\}$, g: $\{r\}$ — $\{r\}$, $\varnothing$

INF5110 – Compiler Construction

Targets

Targets & Outline

Intro

2AC and costs of instructions

Basic blocks and control-flow graphs

Code generation algo

Global analysis

Example once again: kill and gen

B_0 k: $\{x, y\}$, g: $\{a\}$ — $\{a\}$, $\{y, a\}$

B_1 k: $\{x, y\}$, g: $\{a, y\}$ — $\{y, a\}$, $\{a, x, y\}$

B_3 k: $\{a\}$, g: $\{y, z\}$ — $\emptyset$, $\emptyset$

B_4 k: $\{x, y\}$, g: $\{x, y\}$ — $\{x, y\}$, $\{x, y\}$

B_2 k: $\{y\}$, g: $\{z\}$ — $\{x, z, a\}$, $\{x, y, a\}$

B_5 k: $\{r, a\}$, g: $\{x, y\}$ — $\{x, y\}$, $\{r\}$

B_6 k: $\{\}$, g: $\{r\}$ — $\{r\}$, $\emptyset$

Targets

Targets & Outline

Intro

2AC and costs of instructions

Basic blocks and control-flow graphs

Code generation algo

Global analysis

Example once again: kill and gen

B_0 k: $\{x, y\}$, g: $\{a\}$ — $\{a\}$ / $\{y, a\}$

B_1 k: $\{x, y\}$, g: $\{a, y\}$ — $\{y, a\}$ / $\{a, x, y\}$

B_3 k: $\{a\}$, g: $\{y, z\}$ — $\varnothing$ / $\{x, z, a\}$

B_4 k: $\{x, y\}$, g: $\{x, y\}$ — $\{x, y\}$ / $\{x, y\}$

B_2 k: $\{y\}$, g: $\{z\}$ — $\{x, z, a\}$ / $\{x, y, a\}$

B_5 k: $\{r, a\}$, g: $\{x, y\}$ — $\{x, y\}$ / $\{r\}$

B_6 k: $\{\}$, g: $\{r\}$ — $\{r\}$ / $\varnothing$

Example once again: kill and gen

B_0 k: $\{x, y\}$, g: $\{a\}$ — $\{a\}$ / $\{y, a\}$

B_1 k: $\{x, y\}$, g: $\{a, y\}$ — $\{y, a\}$ / $\{a, x, y\}$

B_3 k: $\{a\}$, g: $\{y, z\}$ — $\{x, y, z\}$ / $\{x, z, a\}$

B_4 k: $\{x, y\}$, g: $\{x, y\}$ — $\{x, y\}$ / $\{x, y\}$

B_2 k: $\{y\}$, g: $\{z\}$ — $\{x, z, a\}$ / $\{x, y, a\}$

B_5 k: $\{r, a\}$, g: $\{x, y\}$ — $\{x, y\}$ / $\{r\}$

B_6 k: $\{\}$, g: $\{r\}$ — $\{r\}$ / $\varnothing$

Targets
Targets & Outline
Intro
2AC and costs of instructions
Basic blocks and control-flow graphs
Code generation algo
Global analysis

Example once again: kill and gen

B_0 k: $\{x, y\}$, g: $\{a\}$ — $\{a\}$ / $\{y, a\}$

B_1 k: $\{x, y\}$, g: $\{a, y\}$ — $\{y, a\}$ / $\{a, x, y, z\}$

B_3 k: $\{a\}$, g: $\{y, z\}$ — $\{x, y, z\}$ / $\{x, z, a\}$

B_4 k: $\{x, y\}$, g: $\{x, y\}$ — $\{x, y\}$ / $\{x, y\}$

B_2 k: $\{y\}$, g: $\{z\}$ — $\{x, z, a\}$ / $\{x, y, a\}$

B_5 k: $\{r, a\}$, g: $\{x, y\}$ — $\{x, y\}$ / $\{r\}$

B_6 k: $\{\}$, g: $\{r\}$ — $\{r\}$ / $\varnothing$

INF5110 – Compiler Construction

Targets

Targets & Outline

Intro

2AC and costs of instructions

Basic blocks and control-flow graphs

Code generation algo

Global analysis

Example once again: kill and gen

B_0 k: $\{x, y\}$, g: $\{a\}$ — $\{a\}$, $\{y, a\}$

B_1 k: $\{x, y\}$, g: $\{a, y\}$ — $\{y, z, a\}$, $\{a, x, y, z\}$

B_3 k: $\{a\}$, g: $\{y, z\}$ — $\{x, y, z\}$, $\{x, z, a\}$

B_4 k: $\{x, y\}$, g: $\{x, y\}$ — $\{x, y\}$, $\{x, y\}$

B_2 k: $\{y\}$, g: $\{z\}$ — $\{x, z, a\}$, $\{x, y, a\}$

B_5 k: $\{r, a\}$, g: $\{x, y\}$ — $\{x, y\}$, $\{r\}$

B_6 k: $\{\}$, g: $\{r\}$ — $\{r\}$, $\varnothing$

Targets

Targets & Outline

Intro

2AC and costs of instructions

Basic blocks and control-flow graphs

Code generation algo

Global analysis

Example once again: kill and gen

B_0 k: $\{x, y\}$, g: $\{a\}$ — $\{a\}$, $\{y, z, a\}$

B_1 k: $\{x, y\}$, g: $\{a, y\}$ — $\{y, z, a\}$, $\{a, x, y, z\}$

B_3 k: $\{a\}$, g: $\{y, z\}$ — $\{x, y, z\}$, $\{x, z, a\}$

B_4 k: $\{x, y\}$, g: $\{x, y\}$ — $\{x, y\}$, $\{x, y\}$

B_2 k: $\{y\}$, g: $\{z\}$ — $\{x, z, a\}$, $\{x, y, z, a\}$

B_5 k: $\{r, a\}$, g: $\{x, y\}$ — $\{x, y\}$, $\{r\}$

B_6 k: $\{\}$, g: $\{r\}$ — $\{r\}$, $\emptyset$

Example once again: kill and gen

B_0: k: $\{x, y\}$, g: $\{a\}$ — $\{a, z\}$, $\{y, z, a\}$

B_1: k: $\{x, y\}$, g: $\{a, y\}$ — $\{y, z, a\}$, $\{a, x, y, z\}$

B_3: k: $\{a\}$, g: $\{y, z\}$ — $\{x, y, z\}$, $\{x, z, a\}$

B_4: k: $\{x, y\}$, g: $\{x, y\}$ — $\{x, y\}$, $\{x, y\}$

B_2: k: $\{y\}$, g: $\{z\}$ — $\{x, z, a\}$, $\{x, y, z, a\}$

B_5: k: $\{r, a\}$, g: $\{x, y\}$ — $\{x, y\}$, $\{r\}$

B_6: k: $\{\}$, g: $\{r\}$ — $\{r\}$, $\emptyset$

References I

Targets

Targets & Outline

Intro

2AC and costs of instructions

Basic blocks and control-flow graphs

Code generation algo

Global analysis

*Bibliography

[1] Louden, K. (1997). *Compiler Construction, Principles and Practice*. PWS Publishing.