



Course Script

INF 5110: Compiler construction

INF5110, spring 2018

Martin Steffen

Contents

I	1
10 Code generation	2
10.1 Intro	2
10.2 2AC and costs of instructions	4
10.3 Basic blocks and control-flow graphs	7
10.4 Code generation algo	21
10.5 Ignore for now	27
10.6 Global analysis	27

Chapter

Code generation

What
is it
about?

Learning Targets of this Chapter

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

Contents

10.1 Intro	2
10.2 2AC and costs of instructions	4
10.3 Basic blocks and control-flow graphs	7
10.4 Code generation algo	21
10.5 Ignore for now	27
10.6 Global analysis	27

10.1 Intro

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* [2] (but principles still ok)
- there is also a new edition [1]

This section is based on slides from Stein Kroghdahl, 2015. In this section we work with 2AC as machine code (as from the older, classical “dragon book”). An alternative would be 3AC also on code level (not just intermediate code); details would change, but the principles would be comparable. Note: the message of the chapter is *not*: in the last translation and code generation step, one has to find a way to translate 3-address code two 2-address code. If one assumed machine code in a 3-address format, the principles would be similar. The core of the code generation is the (here rather simple) treatment of registers. In other words, the code generation presented here is rather straightforward (in the sense that it’s done without much optimizations).

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

When not said otherwise: efficiency refers in the following to efficiency of the generated code. Fastness of compilation may be important, as well (and same for the size of the compiler itself, as opposed to the size of the generated code). Obviously, there are trade-offs to be made.

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/optimization: perhaps *invest* in aggressive, computationally complex and rather advanced techniques
 - **many** different techniques used

The above statement that everything so far was computationally simple is perhaps an oversimplification. For example, type inference, aka type reconstruction, is computationally heavy, at least in the worst case. There are indeed technically advanced type systems around. Nonetheless, it’s often a valuable goal not to spend too much time in type checking and furthermore, as far as later optimization is concerned one could give the user the option how much time he is willing to invest and consequently, how aggressive the optimization is done.

The word “untractable” on the slides refers to computational complexity; untractable are those for which there is no *efficient* algorithm to solve them. Tractable refers conventionally to *polynomial type* efficiency. Note that it does not say how “bad” the polynomial is, so being tractable in that sense still might mean not useful. For non-tractable problems, it’s guaranteed that they don’t scale.

10.2 2AC and costs of instructions

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 intermediate 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

Also the book Louden [3] uses 2AC. In the 2A machine code there for instance on page 12 or the introductory slides, the order of the arguments is the opposite!

```
ADD a b // b := a + b
SUB a b // b := b - a
MUL a b // b := b + a
GOTO i // unconditional jump
```

- further opcodes for conditional jumps, procedure calls

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:

$$\&x = \&y + *z$$

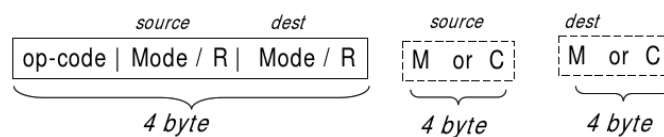
may be 3A-intermediate code, but not 3A-machine code

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 \Rightarrow perhaps longer load
- address modes (as *additional costs*: see later)
 - registers vs. main memory vs. constants
 - direct vs. indirect, or indexed access

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 <i>value</i> M	1	only for source

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

Examples $a := b + c$

Two variants

1. Using registers

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

cost = 6
```

2. Memory-memory ops

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

cost = 6
```

Use of registers

1. 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

2. 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

10.3 Basic blocks and control-flow graphs

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/optimization¹
- amenable to analysis techniques like
 - static simulation/symbolic evaluation
 - abstract interpretation
- basic unit of code generation

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 AST²
 - here: the opposite: synthesizing a CFG from the linear code
- explicit data structure (as another intermediate representation) or implicit only.

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
- **GOTO** *i*: line labelled *i* is a leader
- instruction *after* a **GOTO** is a leader

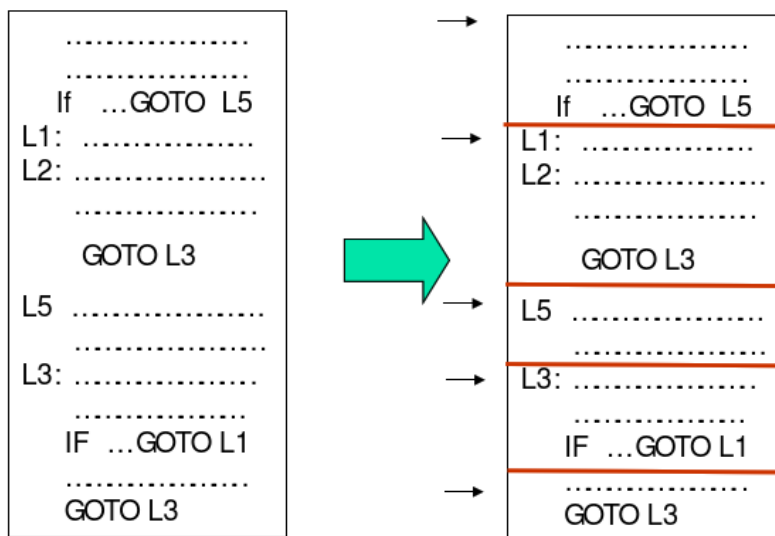
¹Those techniques can also be used across basic blocks, but then they become considerably more costly/challenging.

²See also the exam 2016.

Basic block

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

Partitioning algo

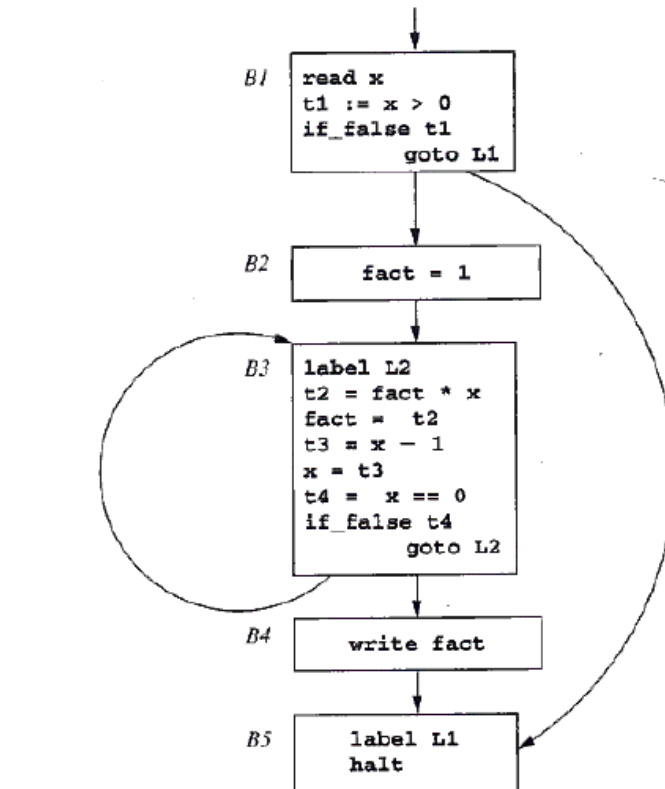


- note: no line jumps to L_2

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**CFG picture****Remarks**

- 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

Intra-procedural refers to “inside” one procedure. The opposite is *inter-procedural*. Intra-procedural analyses and inter-procedural optimizations are quite harder than *intra-procedural*. In this lecture, we don’t cover inter-procedural considerations. Except that call sequences and parameter passing has to do of course with relating different procedures and in that case deal with inter-procedural aspects. But that was in connection with the run-time environments, not what to do about in connection with analysis, register allocation, or optimization. So, in this lecture resp. this chapter, “local” refers to inside one basic block, “global” refers to across many blocks (but inside one procedure). Later, we have a short

look at “global” liveness analysis. As mentioned, we don’t cover analyses across procedures, in the terminology used here, they would be even “more global” than what we call “global”.

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
- importance of loops: loops are “well-behaved” when considering certain optimizations/code transformations (goto’s can destroy that...)

Cycles in a graph are well-known. The definition of loops here, while closely related, is *not* identical with that. So, loop-detection is not the same as cycle-detection (otherwise there’d be no much point discussing it, since cycle detection in graphs is well known, for instance covered in standard algorithms and data structures courses like INF2220).

Loops are considered for specific graphs, namely CFGs. They are those kinds of cycles which come from high-level looping constructs (while, for, repeat-until).

Loops in CFGs: definition

- remember: strongly connected components

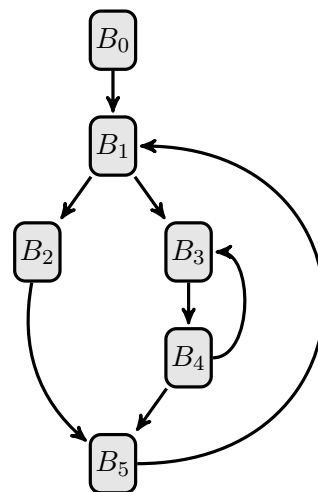
Loop

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 edge³ 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

Loop

CFG



- 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**

Loops as fertile ground for optimizations

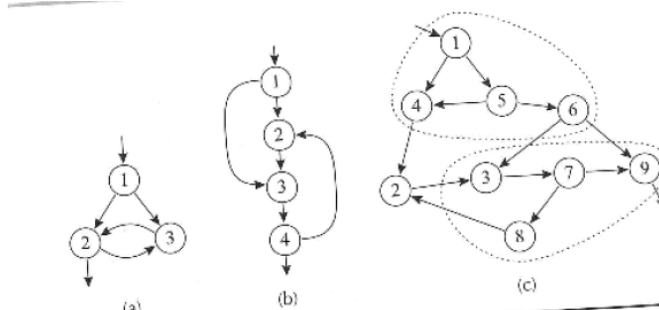
```
while ( i < n ) { i++; A[i] = 3*k }
```

- 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”

³alternatively: general reachability

⇒ add extra node/basic block in front of the *entry* of the loop⁴

Loop non-examples



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)
- ⇒ 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)

⁴That’s one of the motivations for unique entry.

- it's *necessary* or unavoidable that data flows from (a) to (b)
- for *basic blocks*: **exact** answers possible

Treatment of basic blocs

Basic blocks are “maximal” sequences of straight-line code. We encountered a treatment of straight-line code also in the chapter about *intermediate* code generatation. The technique there was called *static similation* (or symbolic execution). *Static simulation* was done for basic blocks only and for the purpose of *translation*. The translation of course needs to be exact, non-approximative. Symbolic evaluation also exist (also for other purposes) in more general forms, especially also working approximatively on abstractions.

In summary, the general message is: for SLC and basic blocks, *exact* analyses are possible, it's for the global analysis, when one (necessarily) resorts to overapproximation and abstraction.

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

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.

Static liveness

The notion of liveness given in the slides correspond to static liveness (the notion that static liveness analysis deals with). A variable in a given concrete execution of a program is *dynamically live* if in the future, it is still needed (or, for non-deterministic programs: if there exists a future, where it's still used. Dynamic liveness is undecidable, obviously.

Definitions and uses of variables

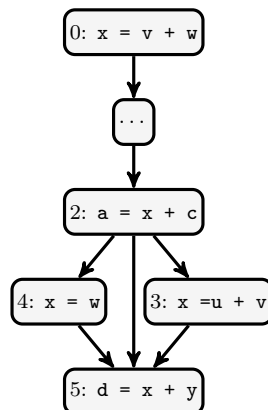
- talking about “variables”: also temporary variables are meant.
- basic notions underlying most data-flow analyses (including liveness analysis)
- here: def’s and uses of *variables* (or temporaries etc.)
- all data, including intermediate results) has to be stored somewhere, in variables, temporaries,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

CFG



- 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 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 techniques (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)

Side-remark: SSA

Side remark: *Static single-assignment* (SSA) format:

- at most one assignment per variable.
- “definition” (place of assignment) for each variable thus *clear* from its name

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 amount 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

Code

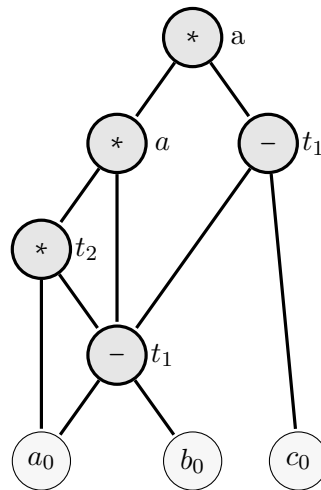
```
t1 := a - b
t2 := t1 * a
a  := t1 * t2
t1 := t1 - c
a  := t1 * a
```

- 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.

DAG of the block

DAG



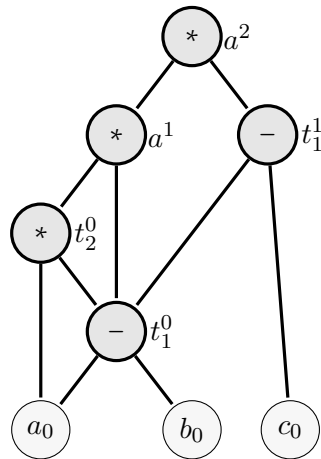
Text

- 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

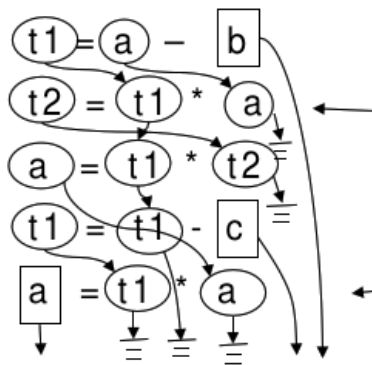
SA = “single assignment”

- 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

Picture



- 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

consider statement $x_1 := x_2 \text{ op } x_3$

- A variable x is live at the *beginning* of $x_1 := x_2 \text{ op } x_3$, 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** scan⁵
- 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**

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 $L(\perp)$

⁵Remember: 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.

- otherwise: basically the same algo

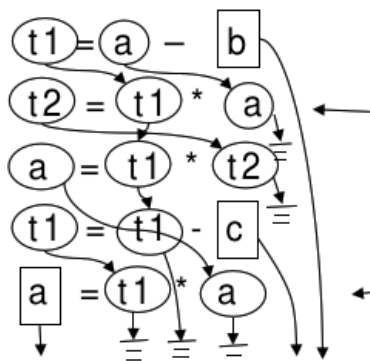
```
// ----- initialise T -----
for all entries: T[i,x] := D
except: for all variables a // but not temps
    T[n,a] := L(1),
//----- backward pass -----
for instruction i = n-1 down to 0
    let current instruction at i+1: x := y op z;
    T[i,x] := D // note order; x can 'equal' y or z
    T[i,y] :=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

Run of the algo'

Run/result of the algo

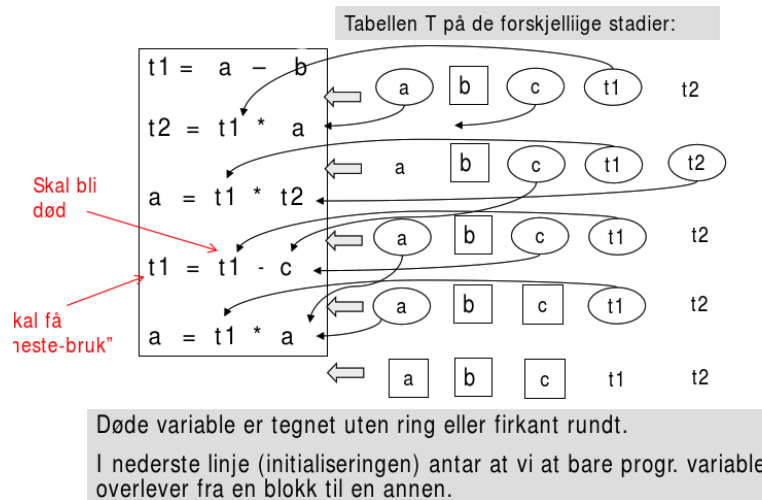
line	a	b	c	t ₁	t ₂
[0]	L(1)	L(1)	L(4)	L(2)	D
1	L(2)	L(1)	L(4)	L(2)	D
2	D	L(1)	L(4)	L(3)	L(3)
3	L(5)	L(1)	L(4)	L(4)	D
4	L(5)	L(1)	L(1)	L(5)	D
5	L(1)	L(1)	L(1)	D	D

Picture



```
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 \times “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

10.4 Code generation algo

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 \Rightarrow at exit:
 - all *variables* stored back to main memory
 - all temps assumed “lost”
- remember: assumptions in the intra-block liveness analysis

Limitations of the code generation

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

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

- 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 $b + a$

The limitation that read-only variables are not put into registers is not a “design-goal”, it’s a not so smart side-effect on the way the algo works. The algo is a quite straightforward way of making use of registers which works block-local. Due to its simplicity, the treatment of read-only variables leaves room for improvement. The code generation makes use of liveness information, if available. In case one has invested in some global liveness analysis (as opposed to a local one), the code generation could profit from that by getting more efficient. But its *correctness* does not rely on that. Even without liveness information, it is correct, by assuming conservatively or defensively, that all variables are always live (which is the worst-case assumption).

Purpose and “signature” of the *getreg* function

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

getreg function

available: *liveness/next-use* info

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

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

- **location:** register (if possible) or memory location

Code 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)

By saying that the register descriptor is needed to track the content of a register, it’s not meant the actual *value* (which will only be known at run-time). It’s rather keeping track of the following information: the content of the register correspond to the (current content of the following) variable(s). Note: there might be situations where a register corresponds to more than one variable.

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 $x := y \text{ op } z$

```
l = getreg(`x:= y op z') // target location for x
if l ∉ Ta(y) then let ly ∈ Ta(y) in emit ("MOV ly, l");
let lz ∈ Ta(z) in emit ("OP lz, 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 (\Rightarrow step 4 also missing)
 - details of *getreg* hidden.

Non-deterministic code generation algo for $x := y \text{ op } z$

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

Exploit liveness/next use info: recycling registers

- 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 $z := x \text{ op } y$:

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

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

if ¬Tlive[i, y] and Ta(y) = r then Tr := Tr(r ↦ y)
if ¬Tlive[i, z] and Ta(z) = r then Tr := Tr(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

- side remark: for address descriptor
 - no such “wipe” needed, because it won't make a difference (y and/or z are not-live anyhow)
 - their address descriptor won't be consulted further in the block

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

```

l = getreg(" x := y op z ")
if l ∉ Ta(y)
then let ly = best (Ta(y))
      in emit ("MOV ly, l ")
else skip;
let lz = best (Ta(z))
in emit ("OP lz, l ");
Ta := Ta(_ ↦ l);
Ta := Ta[x ↦ l];
Tr := Tr[l ↦ x]

```

getreg algo: $x := y \text{ op } z$

- 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 unused 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: $x := y \text{ op } z$ 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 suitable 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	$L(2)$	$L(1)$	$L(2)$	D	$L(3)$	D	D
2	$L(1)$	$L(1)$	$L(1)$	D	$L(3)$	$L(3)$	D
3	$L(1)$	$L(1)$	$L(1)$	D	D	$L(4)$	$L(4)$
4	$L(1)$	$L(1)$	$L(1)$	$L(1)$	D	D	D

Code sequence

Code sequence

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

- address descr's: "home position" not explicitly 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"

10.5 Ignore for now

10.6 Global analysis

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")

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”

We said that “now” a re-use of temporaries is possible. That is in contrast to the block *local* analysis we did earlier, before the code generation. Since we had a local analysis only, we had to work with assumptions concerning the variables and temporaries at the end of each block, and the assumptions were “worst-case”, to be on the safe side. Assuming variables live, even if actually they are not, is safe, the opposite may be unsafe. For temporaries, we assumed “deadness”. So the code generator therefore, under this assumption, must not reuse temporaries across blocks.

One might also make a parallel to the “local” liveness algorithm from before. The problem to be solved for liveness is to determine the status for each variable *at the end of each block*. In the local case, the question was analogous, but for the “end of each line”. For sake of making a parallel one could consider each line as individual block. Actually, the global analysis would give *identical* results also there. The fact that one “lumps together”

maximal sequences of straight-line code into the so-called *basic blocks* and thereby distinguishing between local and global levels is a matter of efficiency, not a principle, theoretical distinction. Remember that basic blocks can be treated in one single path, whereas the whole control-flow graph cannot: do to the possibility of loops or cycles there, one will have to treat “members” of such a loop potentially more than one (later we will see the corresponding algorithm). So, before addressing the global level with its loops, its a good idea to “pre-calculate” the data-flow situation per block, where such treatment requies one pass for each individual block to get an *exact* solution. That avoid potential line-by-line *recomputation* in case a basic block needs to be treated multiple times.

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*

Loops vs. cycles

As a side remark. Earlier we remarked that loops are closely related to cycles in a graph, but not 100% the same. Some forms of analyses resp. algos assume that the only cycles in the graph are *loops*. However, the techniques presented here work generally, i.e., the worklist algorithm in the form presented here works just fine also in the presence of general cycles. If one had no cycles, no loops. special strategies or variations of the worklist algo could exploit that to achieve better efficiency. We don't pursue that issue here. In that connection it might also be mentioned: if one had a program *without* loops, the best strategy would be *backwards*. If one had straight-line code (no loops and no branching), the algo corresponds directly to “local” liveness, explained earlier.

inLive and *outLive*

- tracing / approximating set of live variables⁷ at the *beginning* and *end* per basic block
- *inLive* of a block: depends on

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

- *outLive* of that block and
- the SLC inside that block
- *outLive* of a block: depends on *inLive* of the *successor* blocks

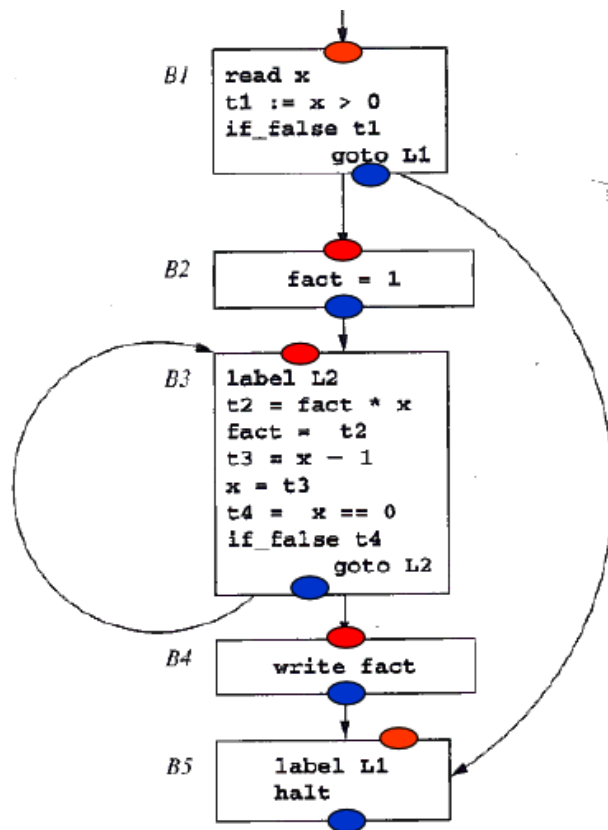
Approximation: To err on the safe side

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*)

Example: Faculty CFG

CFG picture



Explanation

- *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 \Rightarrow avoid *recomputation* for blocks in loops

Precomputation

We mentioned that, for efficiency, it's good to *precompute* the local data flow per local block. In the smallish examples we look at in the lecture or exercises etc.: we don't pre-compute, we often do it simply on-the-fly by "looking at" the blocks' of SLC.

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)

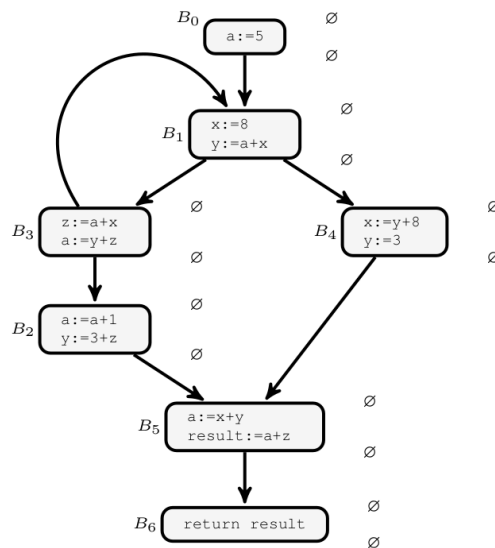
Example

```

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

Picture



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

Iterative algo

General schema

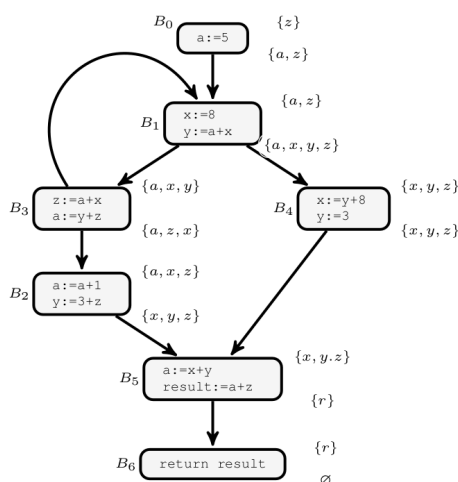
Initialization start with the “minimal” estimation (\emptyset everywhere)

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 principle arbitrary⁸
- in tendency: following edges **backwards**
- comparison: for linear graphs (like inside a block):
 - no repeat-until-stabilize loop needed
 - 1 simple backward scan enough

Liveness: run



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”

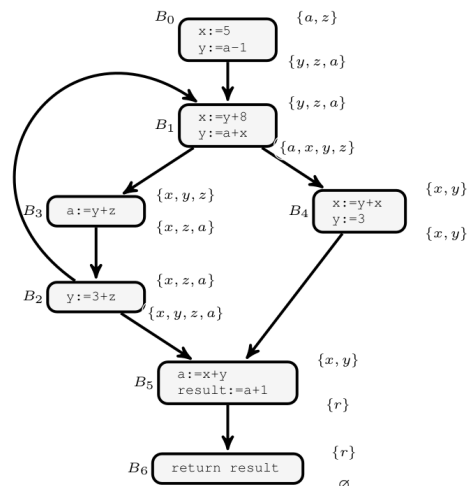
after having updated the *outLive* info for B_1 following the edge from B_3 to B_1 *backwards* (propagating flow from B_1 back to B_3) **does not increase the current solution for B_3**

- 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...)

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

In the script, the figure shows the end-result of the global liveness analysis. In the slides, there is a “slide-show” which shows step-by-step how the liveness-information propagates (= “flows”) through the graph. These step-by-step overlays, also for other examples, are not reproduced in the script.

Another, more interesting, example



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)

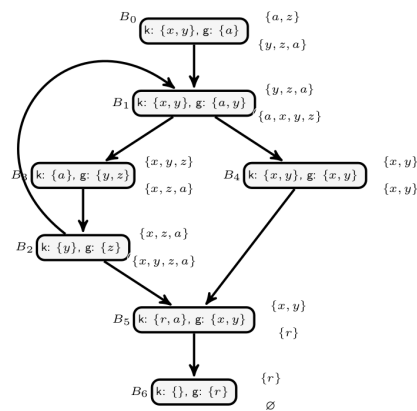
$$inLive = outLive \setminus kill(B) \cup generate(B)$$

- 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`

Order of kill and generate

As just remarked, one should keep in mind the order of kill and generate in the definition of transfer functions. In principle, one could also arrange the opposite order (interpreting kill and generate slightly differently). One can also define the so-called transfer function directly, without splitting into kill and generate (but for many (but not all) such a separation in kill and generate functionality is possible and convenient to do). Indeed using transfer functions (and kill and generate) works for many other data flow analyses as well, not just liveness analysis. Therefore, understanding liveness analysis basically amounts to having understood data flow analysis.

Example once again: kill and gen



Bibliography

- [1] Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2007). *Compilers: Principles, Techniques and Tools*. Pearson, Addison-Wesley, second edition.
- [2] Aho, A. V., Sethi, R., and Ullman, J. D. (1986). *Compilers: Principles, Techniques, and Tools*. Addison-Wesley.
- [3] Loudon, K. (1997). *Compiler Construction, Principles and Practice*. PWS Publishing.

Index

basic block, 6

code generation, 1

control-flow graph, 6

cost model, 4

leader, 6