Course Script

INF 5110: Compiler construction

INF5110, spring 2021

Martin Steffen

Contents

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

Overview

This chapter does the last step, the "real" code generation. Much of the material is based on the (old) *dragon book* [2]. The book is a classic in compiler construction. The principles on which the code generation are discussed are still fine. Technically, the code generation is done for two-adddress machine code, i.e., the code generation will go from 3AIC to 2AC, i.e., to an architecture with 2A instruction set, instructions with a 2-address format. For *intermediate* code, the two-address format (which we did not cover), is typically not used. If one does not use a "stack-oriented" virtual machine architecture, 3AIC is more convenient, especially when it comes to analysis (on the intermediate code level).

For *hardware* architectures, 2AC and 3AC have different strengths and weaknesses, it's also a question of the technological state-of-the-art. There are both RISC and CISC-style designs based on 2AC as well as 3AC. Also whether the processor uses 32-bit or 64-bit instructions plays a role: 32-bit instructions may simply be too small to accomodate for 3 addresses. These questions, how to design an instruction set that fits to the current state or generation of chip or processor technology for some specific application domain belongs to the field of *computer architecture*. We assume a instruction set as given, and

base the code generation on a 2AC instruction set, following Aho et al. [2]. There is also a new edition of the dragon book [1], where the corresponding chapter has been "ported" to cover code generation for 3AC in the new version, vs. the 2AC generation of the older book. The principles don't change much. One core problem is register allocation, and the general issues discussed in that chapter would not change, if one would do it for a 2A instruction set.

Register allocation

Of course, details would change. The register allocation we will do will be on the one hand actually pretty simple. Simple in the sense that the code generator does not make a huge effort of optimization. One focus will be on code generation of "straight-line intermediate code", i.e. code *inside* one node of a control-flow graph. Those code-blocks are also known as *basic blocks*. Anyway, the register allocation method walks through one basic block, keeping track on which variable and which temporary currently contains which value, resp. keeping track for values, in which variables and/or register they reside. This book-keeping is done via so-called *register descriptors* and *address descriptors*. As said, the allocation is conceptually simple: focusing on not-very agressive allocation inside one basic block. The presentation also ignores the more complex addressing modes we discussed in the previous chapter. Still, the details will look, well, already detailed and thus complicated. Those details would, obviously change, if we would use a 3AC instruction set, but the notions of address and register descriptors would remain. Also the way, the code is generated, walking through the instructions of the basic block, could remain. The way it's done is "analogous" on a very high level to what had been called *static simulation* in the previous chapter. "Mentally" the code generator goes line by line through the 3AIC, and keeps track of where is what (using address and register descriptors). That information useful to make use of register, i.e., generating instructions that, when executed, reuse registers, etc.

That also includes making "decisions" which registers to reuse. We don't go much into that one (like asking: if a register is "full", contains a variable, is it profitable to swap out the value?). By swapping, I mean, saving back the value to main memory, and loading another value to the register. If the new value is more "popular" in the future, being needed more often etc, and the old value maybe less, then it is a good idea to swap them out, in case all registers are filled already. If there is still registers free, the simple strategy will not bother to store anything back (inside one basic block), it would simply load variables to registers as long as there is still space for it.

Optimization (and "super-optimization"), local and global aspects

Focusing on straightline code, we are dealing with a finite problem (similar to the setting when translating p-code to 3AIC in the previous chapter), so there is no issue with non-termination and undecidability. One could try therefore to make an "absolutely optimal" translation of the 3AIC. The chapter will discuss some measures how to estimate the quality of the code in the form of a simple *cost model* in Section 10.2. One could use that cost model or other, more refined ones, to define what optimal means, and then produce optimal code for that. Optimizations that are ambitious in that way are sometimes called

"super-optimization" [5] and compiler phases that do that are super-optimizers. Super-optmization may not only target register usage or cost-models like the one used here, it's a general (but slighty weird) terminology for transforming code into one which genuinely and demonstrably optimal (according to a given criterion). In general, that's of course fundamentally impossible, but for straight-line code it can be done.

The code generation here does *not* do that. Actually, super-optimization is not often attempted outside this lecture, as well. One reason should be clear: it's costly. For long pieces of staight-line code (i.e., big basic blocks) it may take too much time. There is also the effect of reducing marginal utility. A relatively modest and simple "optimization" may lead to initially drastic improvement, compared to not doing anything at all. However, to get the last 10% of speed-up or improvement pushes up the effort disproportionally.

A related reason is: super-optimization can be achieved at all only for parts of the code (like straightline code and basic blocks). One can push the boundaries there, as long as it remains a finite problem, for instance, allowing branching (but leaving out loops). that will make the problem a more complicated and targets larger chunk of code, which drives up the effort, as well. As a side remark: Symbolic execution is an established terminology and technique addressing also conditionals, but typically not loops. It also can be seen as some form of "static simulation" but

Generally, even if we are target larger chunks of code or are more aggressive in the goals of optimization, there are boundaries of what can be done. If we stick to our setting, where we currently generate code *per basic block*, super-optimization may be costly but doable. But it would be only locally optimal, per *one block*. Especially when having a code, where local blocks are small, that would have the positive effect that locally super-optimized code may be done without too much effort. But what good would that do, if the non-local quality is bad? Focusing all optimization effort onto the local blocks and ignoring the global situation may be a an unbalanced use of resources. It may be better to do a decent (but not super-optimal) local optimization that, with a low-effort approach achieves already drastic improvments, and *also* invests in a simple global analysis and optimization (perhaps approximative), to also reap there low-effort but good initial gains.

That's also the route the lecture takes: now we are doing a simple register allocation, without much optimization or strategy to find the best register usage (and we discuss also one global aspect of program, across the boundaries of one elementary block. That global aspect will be *live variable analysis*, that will come later in Section 10.7, because first let's discuss local live variable analysis which is used for the local code generation. We can remark already here, that live variable analysis can be done locally or globally; the generation just uses live variable information for its task, whether that information is local or global. So the code generation is, in that way, *independent* form whether one invests on local or on global live variable analysis. It's just produces better code, i.e., makes better use of registers, when being based on better information (like using live variable information coming from a global live variable analysis). Indeed, the code generation would produce semantically *correct* code, without *any* live variable analysis! In that way, the analysis and the code generation are separate problems (but not independent, as the register allocation in the code generation makes use of the information from live variable analysis).

Concerning the "degree of locality" of the code generation. The algorithm works super-locally, insofar that it generates 2AC and makes decisions on registers line by line: every line of 3AIC is translated onto 2 (or sometimes one) line of 2AC. There is no attempt afterwards to go through the 2AC again, getting some more global perspective and them optimize it further, for instance rearranging the lines, or obtaining a register usage better that the one that had been arranged for by the line-by-line code generation. The code generation steps through the 3AC line-by-line but is not completely local, it does some book-keeping about registers used, i.e., allocated in the past. And, not to forget, the code generator has access to liveness information, which is information about the *future* use of registers. In the previous chapter, the *macro expansion* was really line-by-line local, where 3AIC was translated to 1AIC (i.e., p-code): each 3AIC line was expanded into some lines of p-case in a completely "context-free" manner, focusing on each individual, line independent from in which context the line is used. That simplistic expansion ignored the *past*, i.e., what happened before, and it ignored the future, i.e., what will happen afterwars. The code generation here takes care for both aspects, in a simple manner, . What has happened in the past is kept tracked by the register and address descriptors. Aspects of the future are taken care of by the liveness analysis. Depending on whether one does as block-local liveness analysis or a global analysis just changed how "far into the future" the analysis looks. As far as the past in concerned: that one is (in our presentation) just block-local. The book-keeping with the register and address descriptor starts fresh with each block, there is here *no* memory of what potentially had happened in some earlier block.

Live variable analysis

Now, what is live variable analysis anyway, if we mention it here already, and what role does it play here? Actually, being alive means a simple thing for a variable: it means the variable "will" be used in the future. One could dually also say, a variable is dead, if that is not the case. Only that one normally talks about variables being live, not so much about their death. "Death analysis" would not sound appealing.... At any rate, it's important information, especially for register allocation: if it so happens that the value of a variable is stored in a register and if one figures additionally out, that the variable is dead (i.e., not used in the future), the register may be used otherwise. What that involves, we elaborate on further below, in first approximation we can think that the register is simply "free" and can just be used when needed otherwise.

Now, the "definition" for a variable of being live is a bit *unprecise*, and we wrote that the variable "will be used in the future" using quotation marks. What's the problem? The problem is that the future may be unknown, and in general it's impossible to know the exact future. There can be different reasons for that. One is, depending how which language one targets for the analysis, fundamental principles like undecidablity may prevent the the future behavior from exactly be known. There can be actually another reason, namely if one analyzes not a global program but only a fragment (maybe one basic block, one loop body, one procedure body). That means, the program fragment being analyzed is *"open"* insofar its behavior may depend on data coming from outside. In particular, the program fragment's behavior depends on that outside data or "input", when conditionals or conditional jumps are used. Even if the possible input is finite, maybe just a single bit, i.e., a

single input of "boolean type", that may influence the behavior. One behavior where, at a given point a variable *will* be used, and another behavior, where that variable will *not* be used. In one future behavior, the variable is live, in the other future, it is dead. Without knowing whether the input is true of false, one cannot say that the variable "will" be used or not, it simply depends. This obstacle is a different one than the principle undecidability of general programs, which applies to *closed* programs already. For finite possible inputs (and without loops) the problem is still finite: an analysis can just "statically simulate" all runs one by one for each input, and for each individual behavior it is exactly known at each point, whether a variable will be used or not, assuming that the program is deterministic. But overall, without the input known, the program behavior is unknown.

Coming back to the "definition" of liveness. The long discussion hopefully clarified, that in a general setting, when analyzing a (piece of a) program it cannot be about whether a variable *will* be used. The question is whether the variable *may* be used. We want to use the liveness information in particular to see if one can consider a register as free again. If there *exists a possible future* where the variable *may* be used, then the code generator cannot risk reusing the register. That means, the notion of (static) liveness is a question of a condition that "may-in-the-future" apply. There are other interesting conditions of that sort. Some would be characterized by "must" instead of "may". And some may refer to the past, not the future. That would lead to the area of *data-flow analysis* (or more ambitiously, abstract interpretation). We won't go deep there, we stick to live-variable analysis (for the purpose of code generation). That will be done in Section 10.7. However, if one understands live variable analysis, especially the *global* live variable analysis covered later, one has understood core principles of many other flavors of data flow analysis (may or must, forward or backward).

Talking about conditions applying to the "past", perhaps we should defuse a possible misconception. Liveness of a variable refers to the future, and we said, there are reasons why one cannot know the future. Everyone knows, it's hard do predictions, in particular those concerning the future. So one may come to believe that analyzing the past would not face the same problems. When running a (closed) program that may be true: we cannot know the future, but we can record the past ("logging"), so the past is known. But here we are still inside the compiler, doing *static* analysis and we may deal with *open* program fragments. For concretness sake, let's use some particular question for illustration: "undefined variables" (or nil-pointer analysis). That refers to some condition in the past, at some point, a variable is not initialized, perhaps containing a nil-pointer, and the reason is that there was not point in the past run where the variable were initialized. Statically, a compiler warning about "uninitalized variables" typically means, the variable is potentially uninitalized ("may"), namely there may *exist* a run, where there is no initialization of a variable. Or dually, a variable is properly initialized at some point, when *for all* pasts that lead to that point the variable has been initialized. But for open programs (and/or working with abstractions), there may statically be more than one possible past and we cannot be sure which one will concretely be taken. Maybe indeed all or some of them will be taken at run time, when the code fragment being under scrutiny is executed more than once. That is the case when the analyzed code is part of a loop, or correspond to a function body called variously with different arguments. In summary, the discinction between "may" and "must" applies also to statically analysizing properties concerning the past.

Reusing and "freeing" a register

We said that the liveness status of a variable is very important for register usage. That's understable: a variable being dead does not need to occupy precious register space, and the register can be "freed". We promised in the previous paragaph that we would elaborate on that a bit, as it involves some fine points that we will see in the algo later, which may not be immediately obvious from reading the code.

First of all, as far as the hardware platform is concerned, there is no such thing as a full, non-free or empty or free register. A register is just some fast and small piece of specific memory in hardware in some physical state, which corresponds to a bit pattern or binary representation. The latter one is a simplification or abstraction, insofar the registers may be in some "intermediate, instable" state in (very short) periods of time between "ticks" of the hardware clock. So, the binary illusion is an abstraction maintained typically with the help of a clock, and compilers rely on that: registers contain bit strings or words consisting of bits. But it's not the case that `0000` "means" empty, for course. But when is a register empty then? As said, as far as the hardware is concered, the hardware executes for instance the 2AC we are now about to generate, fullness and emptyness of registers simply does not exists. It only consists conceptually inside the compiler and code generator, which has to keep track of the status and "picturing" registers as full and empty. If the code generator wants to reuse a register (in that it generates a command that loads the relevant piece of data into the chosen register) the generator prefers to use an "empty" one, for instance one that so far has not been used at all. Initially, it will rate all registers as empty (though certainly some bit pattern is contained in them in electric form, so to say). Now in case a register contains the value for a variable, but the variable is known to be dead, doesn't that qualify for the register being free? So isn't it as easy as the following?

> A register is free if it contains dead data (or "no data" insfoar as the register has not been used before)?

Sure enough, that's indeed also why liveness analysis is so crucial for register allocation. However, one has to keep in mind another aspect. Just because the value of a register is connected to a variable that is dead does not mean one can "forget" about it and, by reusing the register, overwrite it. So, why not, isn't that the definition of being dead? In a way, yes. But there are two aspects of why that's not enough. One is, that the content of a variable is kept in *two copies*, one in main memory and one in the register. And it may well be the case that the one in main memory "is out of sync". After all, the code generator loaded the variable to register to faster manipulate the "variable", therefore it is a good sign, so to say, that it's out of sync. Keeping main memory and registers always 100% in sync is meaningless; then we would be better off without registers at all. Still, if the variable is really dead, what does this inconsistency matter? That's the second point we need to consider: the concrete code generator later will make a "local" life analysis, only. So it can only knows what's going whether *in the current block* the variable is life or dead (respectively, all variables are "assumed" to be live at the end of a block. That's different from temporaries, that are assumed to be dead. That means, "one" has to store the value back to main memory. Actually, "one" needs to store that value back, if "one" suspects the values disagrees, if there is an *inconsistency* between them. Who is the "one" that needs to store them value back? Of course that's the code generator, that has to generate, in case of need, a corresponding store command, and it has to consult the register and

address descriptors to make the right decision. After “synchronizing” the register with the main memory, the register can be considered as “free”.

Local liveness analysis here

That was a slightly panoramic view about topics we will touch upon in this chapter, but really only slightly panoramic, since register allocation in general is a complex and much addressed problem. But the chapter will be more focused and concrete: code generation from 3AIC to 2AC, making use of liveness analysis which is mainly done *locally, per basic block.* We so far discussed live variable analysis and problems broader than we actually need for what is called *local* analysis here (local in the sense per basic block local). For basic blocks, which is straight-line code, there is neither looping (via jumps) nor is there branching (which would lead to "don't-know" non-determinism in the way described). That's the reason why techniques similar to what has been earlier called “static simulation” will be used. The live variable analyzer steps through the code line by line, and that may be called simulation (the terms simulation or static simulation are, however, not too widely used in that context).

There are two aspects worth noting in that context. One is, when talking about “simulation” it's not that the analysis procedure does *exactly* what the program will do. Since we are doing local analysis of only a *fragement* of a program, a basic block, we don't know the concrete values, that's not easily done (one could do it symbolically, though). But we don't need to pre-calculate the outcome, as we are not interested in what the program exactly does, we are interested in one particular aspect of the program, namely the question of the liveness-status of variables. In other words, we can get away with by working with an *abstraction* of the actual program behavior. In the setting here, for local liveness, even given the fact that the basic block is “open”, that allows *exact* analysis, in particular we know exactly whether the variable *is* live or *is not*. So the “may” aspect we discussed above is irrelevant, locally. The fact that we don't use the exact values of the variables (coming potentially from “outside” the basic block under consideration) does not influence the question of liveness, it's independent from concrete values. If we would have conditionals, that would change, because values would influence the control-flow. So, in that way it's not a “static simulation” of actual behavior, it's more simulation stepping through the progam but working with an abstract representation of the involved data. As said, the concrete values can be abstracted away, in this case without loosing precision.

There is a second aspect we would like to mention in connection with calling the analysis some form of “static simulation”. Actually, the live analysis that comes before the code generation, steps through the program in a *backward* manner. In that sense, the term “simulation” may be dubious (actually, the term static simulation is not widely used anyway). But actually, in a more general setting of general data flow analysis, there are many useful backward analyses (live variable analysis being one prominent example) as well as many useful forward analyses (undefined variable analysis would be one).

Therefore, in our setting of code generation: the code generation will “step” though the 3AIC in a *forward* manner, generating 2AIC, keeping track of book-keeping information known as register descriptors and address destriptors. In that process, the code generation makes use of information whether a variable *is* locally live or *is not* locally live (or on

whether a variable *may* be globally live or not when having global liveness info at hand). That means, prior to the code generation, there is a liveness analysis phase, which works *backwardly*.

Exactness of local liveness analysis (some finer points) To avoid saying something incorrect, let's qualify the claim from above that stipulated: for straight-line 3AIC, *exact* liveness calculation is possible (and that what we will do). That's pretty close to the truth...

However, we look at the code generation ignoring *complicating factors*, like more complex addressing modes, and "pointers". We stated above: liveness status of a variable does not depend on the actual value in the variable, and that's the reason why exact calculation can be done. Unfortunately, in the presence of pointers, aliasing enter the picture, and the actual content of the pointer variable plays a role. Similar complications for other more complex addressing modes. We don't cover those complications though. We focus on the most basic 3AIC instructions, but when dealing with a more advanced addressing modes (as done in realistic settings), the exact future liveness status would be known, not even for straight-line code. [2] covers also that, but it's left-out from the slides and the pensum.

There is another fine point. The assumption that in straight-line code, we know what each line is executed *exactly once* is actually not true! In case our instruction set would contain operations like division, there may be division-by-zero exceptions raised by the (floating point) hardware. Similarly, there may be overflows or underflows by other hardware-triggered errors. Whether or not such an exception occurs *depends* on the concete data. So, it's not strictly true that we know whether a variable *is* live or *is* not. It may be, that an exception derails the control flow, and, from the point of the exception, the code execution in that block stops (something else may continue to happen, but at least not in this block). One may say: well, if such a low-level error occurs, probably trashing the program, who cares if the live variable analysis was not predicting the exact future 100%?

That's a standpoint, but a better one is: the analysis actually did not do anything incorrect. The liveness analysis is a "may" analysis, and that even applies to straight-line code. The analysis says a variable in that block may be used in the future, but in the unlikely event of some intervening catastrophe, it actually may not be used. And that's fine: considering a variable live, when in fact it turns out not to be the case is to *err on the safe side*. Inacceptable would would be the opposite case: an exception would trick the code generator to rate variables as dead, when, in an exception, they are not. But fortunately that's not the case, so all is fine.

Code generation

- note: *code generation* so far: AST^+ to **intermediate code**
  - three address intermediate code (3AIC)
  - P-code
- $\Rightarrow$ *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]

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 for intermediate code); details would change, but the principles could 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, code generation would face similar problems. The core of the code generation is the (here rather simple) treatment of *registers*. The code generation and register allocation presented here is rather straightforward; it may look "detailed" and "complicated", but it's not very complex in that the optimization puts very much computational effort into the code generation. One optimization done is is based on liveness analysis. An occurrence of a variable is "dead", if the variable will not be read in the future (unless it's first overwritten). The opposite concept is that the occurrence of a variable is live. It should be obvious that this is essential for making good decisions for register allocation. The general problem there is: we have typically less registers than variables and temps. So the compiler must make a selection: which data should be in a register and which not not?

A scheme like "the first variables in, say, alphabetical order, should be in registers as long as there is space, the others not" is not worth being called optimization... First-come-first-serve like "if I need a variable, I load it to a registers, if there is still some free, otherwise not" is not much better. Basically, what is missing is taking into account information when a variable is no longer used (when no longer live), thereby figuring out, at which point a register can be considered *free again*. Note that we are not talking about run-time, we are talking about code generation, i.e., compile time. The code generator must generate instructions that loads variables to registers it has figured out to be free (again). The code generator therefore needs to keep track over the free and occupied registers; more precisely, it needs to keep track of which variable is contained in which register, resp. which register contains which variable. Actually, in the code generation later, it can even happen that one register contains the values of *more* than one variable (in case two variables at some point are know to contain the same value). Based on such a book-keeping the code generation must also make decisions like the following: if a value needs to be read from main memory and is intended to be in a register but all of them are full, which register should be "purged". As far as the last question is concerned, the lecture will not drill deep. We will concentrate on liveness analysis and we will do that in two stages: a block-local one and a global one in a later section. the local one concentrates on one basic block, i.e., one block of straight-line code. That makes the code generation kind of like what had been called "static simulation" before. In particular, the liveness information is *precise* (inside the block): the code generator knows at each point which variables are live (i.e., will be used in the rest of the block) and which not (but remember the remarks at the beginning of the chapter, spelling out in which way that this may not be a 100% true statement). When going to a *global* liveness analysis, that precision is no longer doable, and one goes for an approximative approach. The treatment there is *typical* for data flow analysis. There are *many* data flow analyses, for different purposes, but we only have a look at liveness analysis with the purpose of optimizing register allocation.

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 (or quality) of the *generated* code. Fastness of compilation, or with a limited memory footprint may be important, as well (likewise may the code size of the compiler itself be an issue, as opposed to the size of the generated code). Obviously, there are trade-offs to be made.

But note: even if we compile *for* a memory-restricted platform, it does not mean that we have to compile *on* that platform and therefore need a "small" compiler. One can, of course, do cross-compilation.

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 sometimes 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 on the slides that everything so far was computationally simple is perhaps an over-simplificcation. For example, type inference, aka type reconstruction, is typically computationally heavy, at least in the worst case and in languages not too simple. There are indeed technically advanced type systems around (including undecidable ones,

like the one for C++...). 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 agressive the optimization is done. For our coverage of type systems in the lecture and the oblig: that one is rather simple and elementary, and poses no problems wrt. efficiency.

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 time* efficiency. Note that it does not say how "bad" the polynomial is, so being tractable in that sense still might not mean practically useful. For non-tractable problems, it's often guaranteed that they don't scale.

10.2 2AC and costs of instructions

Here we look at the instruction set of the 2AC. Well, actually only a small subset of it. In particular, we look at it from the perspective of a "cost model". Later, we want to at least get a feeling that the code we are generating is "good" but then we need a feeling what the "cost" is of the generated code, i.e., the cost of instructions.

When talking about 2AC, it's actually not a concrete instruction set of a concrete platform. Concrete chips have complicated inststruction sets, so it's more that we focus on a (very small) subset of what could be an instruction set of a 2A platform. Now, isn't that another "intermediate code"? We will see that the code now (independent from the fact that its 2AC) is more low-level than before. In that way, it could be a real instruction set of some hardware. The intermediate code from before could not. One could tell the same story we are doing here, translating from 3AIC to 2AC also by doing a translation from 3AIC to 3AC. Still that would pose equivalent problems (register allocation, cost model, etc.), but the presentation here happens to make use of a 2AC.

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**
- translation task from IR to 3AC or 2AC: comparable challenge

2-address instructions format

Format

OP source dest

- note: *order* of arguments here (esp. for minus)
- restrictions on *source* and *target*
  - register or memory cell
  - source: can additionally be a constant

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

- further opcodes for conditional jumps, procedure calls ....

Also the book Louden [4] 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.

Side remarks: 3A machine code

Possible format

```
OP source1 source2 dest
```

- but: 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

As we said, the code generation could analogously be done for 3AC instead of 2AC. But what's the difference then between 3AIC and 3AC, would the translation not be trivial? Not quite, there is a gap between intermediate code and code using the instruction set. The most important difference is the use of registers. Related to that, 3AC instructions typically impose restrictions on the operands of the instructions. In the purest form, one may allow instructions only of the form `r1 := r2 + r3` (here addition as an example), where all arguments, sources and target, must all be in registers. That would result in a pure load-store architecture: before doing any operation at all, the code generator must issue appropriate load-commands, and the result needs to be stored back explicitly. That obviously leads at least to longer machine code, measured in number of instructions (but perhaps the instructions themselvelse may be represented shorter). Analogous restrictions may concern the indirect addressing modes. Instruction sets with a load-store design are often used in RISC architectures.

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
- factors 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

The cost model (like the one here) is intended to model relevant aspects of the code, that influence the efficiency, in a proper and useful manner. The goal is not a 100% realistic representation of the timings of the processor. It will be based on assigning rule-of-thumb numerical costs to different instructions. Actually, it's very simple. The main observation is: accessing a register is "very much" faster than accessing main memory. But the model does not use realistic figures (maybe by consulting the specs of the machine or doing measurements). Indeed, "main memory" access may not have a uniform access cost (in terms of access time). There are factors outside the control of the code generation, which have to do with the memory hierarchy. The code is generated as if there are only two levels: registers and main memory. But, of course, that's not realistic: there is caching (actually a whole hierarchy of caches may be used). Furthermore, data may even be stored in the background memory, being swapped in and out under the control of an operating system. Being not under the control of the code generator, those are stochastic influences. The compiler is not completely helpless facing caches and other memory hierarchy effects. Based on assumptions how caching and paging typically works, the code generator could try to generate code that has good characterisics concerning "locality" of data. Locality means that in general it's a good idea to store data items "than belong together" in close vicinity, and not sprinkle them randomly across the address space (whatever "belonging together" means). That's because the designer of the code generator knows that this suites chaching or swapping algorithms, that perhaps swap out cache lines, banks of adjacent addresses, whole memory pages etc. As far as caches is concerned, that's simply a rational hardware design. But one can also turn the argument around: hardware designers know, that it's "natural" that data structures coming from a high-level data structure of a structured programming language (and which contain conceptually data "that belongs together) will be generated in a way being "localized". Even if the compiler writer has never thought of efficiency and memory hierarchies, it's simply natural to place different fields of a record side by side. Also for more complex, dynamic data structures, such principles are often observed: the nodes of a tree are all placed into the same area

and not randomly. More tricky maybe the the presence of a garbage collector, that could mess that up, if done mindlessness. But also the garbage collector can maken an effort to preserve locality. So, in a way, it all hangs together: well-designed memory placement will be rewarded by standard ways managing the memory hierarchy, and well-designed memory management will run standard memory layout by compilers faster. It's almost a situation of co-evolution.

But all that is more a topic for how the compiler arranges memory (beyond the general principles we discussed in connection with memory layout and the run-time environments). Here we are looking more focused on the code generation and trying to attribute costs *on individual instruction* (so questions of locality cannot be considered, as they are about the global arrangement, neither are questions of caching, etc., as one individual instruction and the instruction set is not aware of caching, let alone the influence of the operating system. So, how can we express the very rough observation "registers are very much fast than memory accesses"? That's easy, register access costs "nothing", it will have a zero costs. Main memory accesses will have cost of 1. Mathematically it means, memory access is infinitely most costly than registers, but as said, it's a model that may be use to generate efficient code, not as a realistic prediction of actual running time in the physical world. Also, the cost of 0 vs. the cost of 1, it's about time *additional* to the load and execution time to the operation. So doing `ADD r1 r2`, and addition involving 2 registers, is not infinitely fast, it costs 1 (say, one load cycle), only the register accesses don't add to the costs, their access and carrying out the addition are done within one single load cycle. Even if we had realistic figures from somewhere (via profiling and measuring average execution times under typical conditions or similar), the use would be limited: as stressed a few times, genuine and absolute optimal performance is (and cannot be) the goal (super-optimization aside). The goal is getting good or excellent performance with a decent amount of effort. We are content to use the cost model as a rough guideline (for the code generator) on decisions like

> when translating *one* line of 3AIC, shall I use a register right now or rather not?

We will see that this is the way the code generator will work. One might not even call it "optimization", at least not in the sense the first some code is generated which afterwards is improved (optimized). The code generator takes the cost model into account on-the-fly, while spitting out the code. Actually, it does not even consults the cost model (by invoking a function, comparing different alternatives for the next lines, and then choosing the best). It simply compiles line after line, and the decisions are plausible, and one convince oneself of the plausibility by looking at the cost model. Actually, one can convince oneself of the plausibility even without looking at the cost model, just knowing that registers should be preferred when possible. But actually that's one of two important pieces of common knowledge the cost model captures.

What's the second piece then? The other piece is that executing one command costs also something. So, each "line" costs 1. In that sense, the 0-costs of register access is realistic, insofar registers access is typically done in one processor cycle, i.e., in the same time slice than the loading and executing the instruction as a whole. So, in that sense, register accesses really don't cost anything additional. Other accesses incur additional costs, and since we don't aim at absolute realism, all the non-register accesses costs 1.

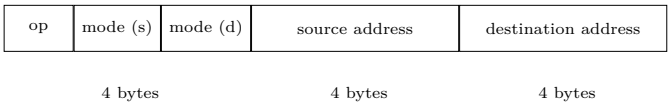


Figure 10.1: Instruction format

Instruction modes and additional costs

mode	abbr.	address	ddded 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

Table 10.1: Cost model

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

The instruction format is shown in Figure 10.1. In the most general case, for our 2AC and for instructions with 2 arguments, an instruction is split into 3 parts each 4 bytes or 4 octets long. 4 bytes are also called one *word* in that architecture. The first word represents the op-code including information how to interpret the following two words, namely the source and the destination address; as mentioned, the destination address is also the address of a source, in a binary operation. The content of the source and destination arguments can be interpreted differently, that called their *mode* in the corresponding op-code. The mode of the two instruction arguments can be specified independantly and the various modes are summarized in Table 10.1 (in the right-hand column).

We see that there are no real restictions when and when not memory access are allowed and when registers. Earlier we mentioned something like "load-store" architectures, where binary operations may only work on registers, or other restrictions. That is not the case here.

The format here corresponds to a 32-bit architecture, which is a popular format (actually, it's pretty old, there had been 32-bit machines early on, likewise also 64-bit (not microprocessors at that time). There are 16-bit microprocessors (in the past), and there are 64-bit processors as well. Of course, having 4 bytes for the op-code does not mean all codes are actually used for actual instructions (that would be way too many). But we have to keep in mind (or at least in the back of our mind, as that's no longer the concern of a compiler writer): the instructions need to be handled by the given hardware with a given size of the "bus", there is no longer the freedom and flexibility of software. In particular, it's not "byte code" (more like 4-bytes code...) And actually, it's nice to think of a binary code as to represent "addition" or "jump", but the 0 and 1's in the code actually are connected to hardware, the slots in the 32-bit word are "wired up" connecting them to logical gates that open and close and trigger other bits/electrons to flow from

here to there that ultimatly result in another bit pattern that can be interpreted as that an addition has happened (on our level of abstraction). So the actual bit-codes for the logical machine instructions are are "sparcely" distributed, and some bit-pattern are not simply unused ("undefined") but would open and close the "logic gates" of the chip in a weird, meaningless manner. As said, all that is not the concern of a compiler writer, who can see an `add`-code as addition, but it's interesting that the story does not end there, there are complex layers of abstraction below that and also, that we are leaving the world of "anything goes" of software: the compiler writer can design any form of intermediate representations in intermediate codes and translate between them etc. But below that, things get more restricted by the physics and the laws of nature.

We can also compare that to byte codes, for instance the one for the mandatory assignment. Byte codes are called like that because the first part of an instruction, the opcode, is byte-size. Unlike the instruction set here, where the opcode part 4 bytes. But otherwise, the byte code from the mandatory assignments resembles the instruction set here to some extent. The byte code is, on the one hand, a p-code-style instruction set that works with a stack; that's of course not the case here. Hovever, the byte code operations sometimes also expect (additional) arguments which are not on the stack but kept in follow-up data. The size of the data there it is not 4 bytes, but the size of a *short* (a short or short integer is typically 2 bytes, also for the mandatory assignment). Another difference is, of course, the byte code is intermediate code, the 2AC represents actual instructions. That's also the reason why the instructions here take 4 bytes to represent the opcodes, as explained above.

Examples `a := b + c`

The following examples are not breathtakingly interesting. They show different possible translations and their costs. The first pair of examples shows two equivalent ways of translating the given assignment, one operating directly on the main memory, one partly loading the arguments to a register and then using that. Both versions have the same cost, in our cost model (despite the fact that the first program executes 3 commands and the second only 2).

The other two examples calculate the same command, but under a different assumption, namely: the arguments are already loaded in some registers. That drives down the costs. But that should be pretty clear, that's why one has registers, after all.

We also see that it to profit from the use of registers, the code generator needs to know which variables are stored in the registers already. That will be done by so-called address descriptors and register destriptors.

Also, especially the second example shows, that sometime the generated code is a bit strange: Since we have only 2AC, one argument is source, the other one is source *and* destination. That means, 2AC like addition "destroy" one argument. That means, in general we need to temporarily copy that argument somewhere else, otherwise it get lost. In the second example, since `a` is updated anyway, the first step uses `a` for that temporary copy of `b`. That's a general pattern of this form of code.

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
```

Addresses 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`

10.3 Basic blocks and control-flow graphs

We have mentioned (in the introductory overview of this chapter and elsewhere) the concepts of basic blocks and control-flow graphs already. Before we continue we introduce those concepts more robustly. The notion of control flow graph is in this lecture used at the level of IC (maybe 3AIC), resp. also machine code. The notion of CFG makes also sense on highler levels of abstractions, i.e., one can do a control-flow graph also for abstract syntax . In our setting, there would be not much difference between to control-flow graphs from intermediate code and machine code. Both representations make use of jumps and conditional jumps and labels (resp. addresses), and that determines the edges of the graph.

In this section, we work with 3AIC, generated from some AST probably with higher-level control-flow constructs like two-armed conditionals and loops. Now we "reconstruct" a more high-level representation of the code by figuring out the CFG (at that level). It is not uncommon to do a CFG first, and *use* the CFG assisting in the (intermediate) code generation.

Anyway, the general concept of CFG works analogously at all levels, same for basic blocks, at least when working with a standard procedural language. The notion of control-flow graph is less applicable to languages with higher-order functions or more specific-purpose languages like constraint-solving notations or logic languages.

Basic blocks

- machine code level equivalent of straight-line code
- (a largest possible) sequence of instructions without
  - jump out
  - 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

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.

When saying, a CFG is "basically" a graph, we mean that, apart from some fundamentals which makes them graphs, details may vary. In particular, it may well be the case in a compiler, that cfg's are some accessible intermediate representation, i.e., a specific concrete data structure, with concrete choices for representation. For example, we present here control-flow graphs as *directed* graphs: nodes are connected to other nodes via edges (depicted as arrows), which represent potential successors in terms of the control flow of the program. Concretely, the data structure may additionally (for reasons of efficiency)

[1] Those techniques can also be used across basic blocks, but then they become more costly and challenging.
[2] See also the exam 2016.

also represent arrows from successor nodes to predecessor nodes, similar to the way, that linked lists may be implemented in a doubly-linked fashion. Such a representation would be useful when dealing with data flow analyses that work "backwards". As a matter of fact: the one data flow analysis we cover in this lecture (live variable analysis) is of that "backward" kind. Other bells and whistles may be part of the concrete representation, like dedicated start and end nodes. For the purpose of the lecture, when don't go into much concrete details, for us, cfg's are: nodes (corresponding to basic blocks) and edges. This general setting is the most conventional view of cfg's.

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

Basic block

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

The CFG is determined by something that is called here "partitioning algorithm". That's a big name for something rather simple. We have learned in the context of *minimization* of DFAs the so-called partitioning refinement approach, which is a clever thing. The partitioning here is really not fancy at all, it hardly deserves being called an algorithm. The task is to find in the linear IC largest stretches of straight-line code, which will be the nodes of the CFG. Those blockes are demarkated by *labels* and *gotos* (and of course the overall beginning and end of the code.) There is only one small addition to that: an unused label, i.e., a label not being the target of some jump, does not demarkate the border between to blocks, obviously. An unused label might as well be not there, anyway.

Partitioning algo

- note: no line jumps to L_2

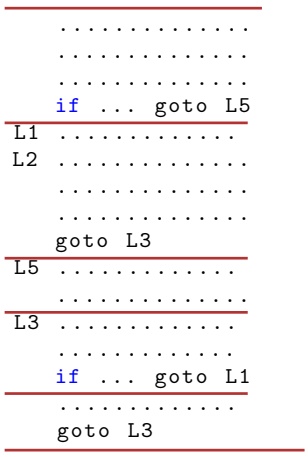


Figure 10.2: Partitioning (illustration)

The partitioning is illustrated in Listing 10.2. The read lines show the demarcations between the code of the basic blocks. The lines at the same time correspond to what we called *leaders*: the leaders are the lines *following* the the red lines and they indicate the first line of a basic block.

Threre is one exception, that's the red line at the end of the program. That one, obviously, does not correspond to a leader or the beginning of some bacic block. It demarcates, however, the end of the last basic block.

Note also that the line labelled L_2 is *not* a leader. The reason is that in the sketched program, this label is not used as jump target, unconditional or otherwise.

It may be worth thinking about what would happen if we considered L_2 a header nonetheless. In that case, the basic blocks would no longer be the *largest* sequences of straight-line code not jumped into. In this example, we would end up with 6 basic blocks instead of 5.

That should, however, not affect the correctness of the generated code. As mentioned, basic blocks are the elementary level of optimizations and code generation. Cutting the basic blocks smaller than necessary will lead to smaller stretches of code targeted by local analysis. An example would be the local liveness analysis covered later. If one uses liveness analysis only on the local level, i.e., only inside basic blocks, then the smaller than necessary basic blocks would lead to a less precise analysis. Liveness analysis (like others) can be precise within basic block, but typically resorts to approximation more globally, like doing analysis for a whole control-flow graph. In Section 10.7, we will look into that kind of global analysis. But when doing only a local one, the analysis ignores what happens outside the current basic block, and thus, to play it safe and assumes variables at the end of a basic block potentially used later. It assumes a variable at the end of a block to be *live*, though a global analysis may reveal that it is not. This safe overapproximation is typical for many forms of analyses, in particular data flow analyses, but also type checking. As a consequence, unnessessarily small blocks of straight-line code lead loss of precision, an overapproximation still safe safe but needlessly approximative.

Indirectly therefore, also register allocation is affected by a too finegrained block structure. As long as the lifeness approximation is correct or safe, the register allocator will lead to correct code as well, though presumably slower code compared to more presise (but correct) liveness information.

This describes what happens to liveness analysis and register allocation if the straight-line code blocks are needlessly small, if one assumes local analysis only. The situation for other kinds of analyses would be similar.

What would happen using small straight-line blocks if one employed a global analysis? In this case, one typically would *not* loose precision. The global analysis anyway looks at the whole control-flow graph. Unlike for local liveness analysis, to stick with this form of analysis, a global analysis will not of course *assume* that a variable at the end of a basic block is live, just to be safe. It will investigate to figure out if the variable may be used in the future or if it's sure it's dead. That's what the global analysis is good for, after all.

Does it mean, if one is doing a global analysis anyway, the size of the blocks of straight-line code does not matter? In a way yes. As said, one will not loose precision by being to finegrained. In an extremal case, one could use every instruction line as one elementary block. Why would one still work with the largest possible stretches of straigh-line code, i.e., with basic blocks of the form introduced?

The reason is mostly that the global analysis can be done if not more precisely, but more efficiently. Global analysis typically involves the analysis of loops or cycles, something that, by definition, is not needed for straight-line code. The analysis of cycles in the control-flow graph entails that one does analysis steps *repeatedly* for nodes participating in a cycle. If one has a large basic block as part of a cycle, one can anayse relevant information (for instance concerning the liveness status of variables) one in summarized form. For instance, let's assume the first usage of a a variable, say x in a given basic block is that it's assigned to like in a line of the form $x := e$, where e does not refer to x. That means, x actually is dead at the beginning of said basic block. A local analysis of the block will find that out, and one can use the information in summarizing corresponding information for the basic block for all variables. Resp. one could do that summary information for all basic block, which form the nodes of the control flow graph. What good would that do? The local analysis, needed for the summary information steps through the lines of the basic block. As we will see (resp., one single pass through the lines is enought). Actually, it should be even intuitively clear, that one pass should be enough to see locally, for each variable, if it's used or not. Anyway, the basic block, as said, may be part of a cycle in the graph, and this need repeatedly be treated. But with the summary information precomputed by local analyses, one at least need to step through the individual ones over and over again, to (re-)discover the liveness status of the involved variables, like rediscovering that x is dead at the entrance of the basic; that information is remembered in the symmary. In this way, working with basic block may not increase precision of the analysis and thereby increase the quality of the produced code, but the analysis itself may become faster.

Let's have a look at some more concrete example. Listing 10.1 shows 3AIC for the faculty function from the previous chapter, and Figure 10.3 shows the corresponding control-flow graph. The code contains 5 basic block and thus the illustration of the control-flow graph 5 nodes. The first line in each node is the corresponding header. Unlike in the schematic example from Figure 10.2, all labels in the code are jump targets. Typically,

the (intermediate) code generator would not generate labels not being used as jump-target, though they are not "harmful"; the partitioning algo does not treat them as leaders and the label instructions from the 3AC are *pseudo-intructions*, i.e., the don't correspond ultimately lead actual machine-code instructions.

3AIC for faculty (from previous chapter)

```
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
```

Listing 10.1: Faculty (3AIC)

Faculty: CFG

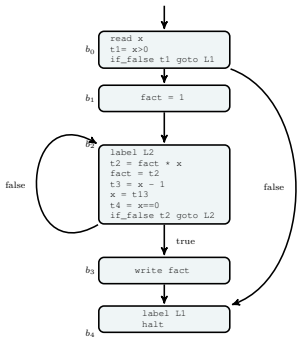


Figure 10.3: Control-flow graph (faculty)

- 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. For example, Figure 10.3 contains the control-flow graph of the faculty procedured, resp. the graph for a main procedure of a program that realizes the faculty function. Note that the program does not do procedure

calls; the faculty is calculated using a while-look in the source language and not the recursive faculty solution, one often finds.

The opposite of intra-procedural analysis is *inter-procedural*. Inter-procedural analyses and the corresponding 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 now 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 dont' cover analyses across procedures, in the terminology used here, they would be even "more global". Actually, in the more general literature, global program analysis would typically refer to analysis spanning more than one procedure. Indeed, one should avoid talking about local analysis without further qualifications; it's better to speak of block-local analysis, procedure-local, method-local, or thread-local, to make clear which level of locality is addressed. We are doing block-local analysis resp. procedure-local analysis (the latter we will also 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)

These three levels are the conventional hierarchy of analyses. But there might be further distinctions, for example. in concurrent programs, one can distinguish between a thread-local (and inter-procedural) analysis and a global analysis, that takes multiple threads into account. Techniques to analyse programs in the presence of concurrency can become much more challenging and also, concurrency analysis requires other techniques that the notion of control-flow graphs resp. further techniques on top of that. Control-flow graph are an inherently sequential concept, capturing the structure of sequential control-constructs like conditionals and loops on the source-language levels. We don't look into analysis in the presence of parallism or concurrency.

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/IN2010.

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

Loop

Loop L in a CFG: set of nodes, including **header node** $h \in L$:

1. any node in L: a path in L to h
2. a path in L from h to any node in L
3. no edge in the graph goes into h from outside L

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

The definition is taken basically from [3], and corresponds also to the one from Aho et al. [1], which calles the loop header *loop entry* instead. Sometimes one also finds the definition, that the header or entry of a loop is *reachable* from the CFG's initial node exactly via one edge (in the last step) from outside the loop. The definition given here does not mention reachability here, but there is no real difference, in particular, since all nodes in a control-flow graph are typically reachable from the initial nodes. That in particular holds for a control-flow graph coming from a program written in source code (and potentially compiled to intermediate code or machine code, depending at which level one generates the control-flow graph). More precisely, if the source language allows goto's and conditional jumps to labels, *then* the CFG can contain unreachable nodes. The same applies when considering intermediate or machine code written by hand, not coming from conventional source code.

The first two points from above make the nodes of a loop *strongly connected*: every node in a loop can reach each other. From the algorithm and data structure lecture INF2220/IN2010, one may have encountered the notion of *strongly connected component* and the corresponding algorithm. A strongly connected component adds "maximality" to the requirement of being strongly connected; i.e., a strongly connected component is a maximal set of strongly connected nodes. This maximality is not required for loops.

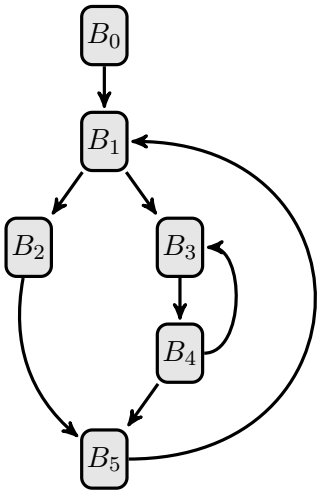


Figure 10.4: Loop example

Loops are strongly connected sets of nodes with a unique header node

For instance in the graph from Figure 10.4, the inner loop $\{B_3, B_4\}$ is strongly connected (and a loop), but not strongly connected, since one could enlarge it to $\{B_1, B_2, \ldots, B_5\}$ and still stay strongly connected. The notion of stronly connnectd components corresponds to outermost loops (including all nested loops within).

Loop

The definition is best understood in a small example.

CFG

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

The *additional assumption* mentioned on the slide about the special role of the root node of a control-flow graph is reminiscent, for example, of the condition we assumed for the start-symbol of context-free grammars in the LR(0)-DFA construction: the start symbol must not be mentioned on the right-hand side of any production (and if so, one simply added another start symbol S'). The reasons for the assumption here are similar: assuming that the root node is not itself part of a loop is not a fundamental thing, it just avoids (in some degenerate cases) a special case treatment. The assumption about the form of the control-flow graph is sometime called “isolated entry”. A corresponding restriction for the “end” of a control-flow graph is “isolated exit”.

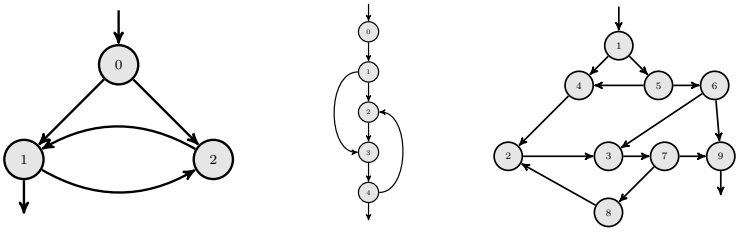


Figure 10.5: Non-loops

Loop non-examples

Figure 10.5

We don't need to explore loops further, actually for the way we do global analysis later (in the form of global liveness analysis) that will work for non-loop cycles ("unstructured" programs) as well as for loop-only graphs, at least in the version we present it. If one knows that there are loops-only, one could improve the analysis (and others). Not in making the result of the analysis more precise, but making the analysis algorithm more efficient. That could be done by exploiting the structure of the graph better, for instance exploiting that loops are nested, for instance targeting inner-loops first. In the examples here, such "tricks" would not work. They violate that each loop is supposed to have a well-defined, unique entrance node. Since we don't exploit the presence of loops, we don't dig deeper here. It should be noted that the definition of loops (with unique entry points) is classical in CFG and program analysis, one may find material where the notion of "loop" is used more loosely (ignoring the traditional definition) where loop and cycle is basically used interchangably.

One is interested in loops not necessarily as a concept in itself, but in the larger context of optimization. We called loops a fertile ground of optimizations, which is of course also true for general cycles: both involve (potential) repetition of code snippets, and shaving off execution time there, that's a good idea. Often, the optimization is about moving things outside of the loop, typically "in front" of the loop. That's when a unique entrance of a loop comes in handy (sometimes called a loop-header). The non-loop examples don't have a single loop-header.

In the more or less distant past, loop detection (and cycle detection) would be a task a compiler would engage it. Now, that most programs are written following structured progamming, there a no non-loop cycles. Additially, when compiled from source code, the program structure contains all the information where the loops are, so there not need to make an analyis (for instance for the intermediate code) to (re-)discover them at the lower level. However, the partitioning algorithm we discussed is a bit in that spirit. The control flow structure is (re-)discovered from (intermediate) code, in the form of the control flow graph.

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[3]

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 instruction pointer
    - 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).

The characterization of data flow may sound plausible: some data is "created" at some point of origin and then "flows" through the graph. In case of branching, one does not know if the data "flows left" or "flows right", so one approximates by taking both cases into account. The "origin" of data seems also clear, for instance, an assignment "creates" or defines some piece of data (as l-value), and one may ask if that piece of data is (potentially or necessarily) used someplace else (as r-value), without knowing resp. being interesting in its exact value that is being used. This is sometimes also called def-use analysis. Later we will discuss definitions and uses. Another illustration of that picture may be the following question: assuming one has an data-base program with user interaction. The user can interact by inputting data via some (web)-interface or similar. That information is then processed and forwarded to some SQL-data base. Now, the inputs are points of origin, and one may ask if this data may reach the SQL database without being "sanitized" first (i.e., checked for compliance and whether the user did inject into the input some escapes and SQL-commands).

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

Anyway, this picture of (user) data originating somewhere in a CFG and then flowing through it is plausible and not wrong per se, but is too narrow in some way. It sounds as if every data flow analysis traces (in an abstract, approximative manner) the flow of pieces of data through the graph.

Not all data flow analyses are like that. Actually, the live variable analysis will be an example for that. So more generally, it's more like that "information of interest" is traced through the graph. For liveness analysis, the piece of information being traced is future usage. Since the information of interests may not be an abstract version of real data, it may also not necessarily be traced in a *forward* manner. For liveness analysis, one is interested in whether a variable may be used in the future. So the information of interest is the locations of usage. That are the points of origin of that information one is interested in. And from those points on, the information is traced *backwards* through the graph. So, this is an example of a *backward analysis* (there are others). Of course, when the program runs, real data *always* "flows" forwardly, as the program runs forwardly: first data orignates and later is may be consumed. But for some analysis (like liveness analysis), one changes perspective: instead of asking: where will information originating here (potentially or necessarily) flows to in the future, one asks:

> where did information or data arriving here orignate (potentially or necessarily) from in the past.

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 are 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

Treatment of basic blocks

Basic blocks are "maximal" sequences of straight-line code. We encountered a treatment of straight-line code also in the chapter about *intermediate* code generation. The technique there was called *static simulation* (or simple 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 on conditionals.

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. We cover liveness analysis in basic blocks in Section 10.4, and global liveness analysis in Section 10.7.

10.4 Liveness analysis (general)

Introductory remarks

Liveness analysis is a classical data flow analysis. The introductory remarks of this chapter already introduced what liveness of a variable means, and why that is important for register allocation. This section covers block-local liveness analysis; analysis whole control-flow graphs will be introduced in Section 10.7. There are many different data flow analyses, and liveness anaysis is only one typical example of that form of semantics analysis (but a very important one). It's typical insofar that the ideas and technique for liveness analysis apply analogously to other data flow analyses. Of course, details what kind in information is traced by data-flow analysis techniques are different for each analysis -here we are tracing liveness information- but the underlying principles remain. By the underlying principles, we mean mostly the way the *global* analaysis is approached in Section 10.7 and the way the approximation is done in an iterative manner. That approach is characteristic for a large rage of analyses.

But in this section here, we first tackle basic blocks. The question is to figure out at each point in a given block, whether a variable is live or not. Live at least as far as the current block is concerned. Focusing locally one single block means, the analysis does not have information about what will happen after that block. As a consequence, the analysis *assumes* that variables are live at the *end* of a basic block. This assumption is done in the spirit of safe approximation. If, seen globally, a variable would actually not be live (statically or dynamically), the register allocation would at least make no error. On the other hand, if a variable would be judged dead in contrast to the real situation, that could lead to wrong code in that the content of the variable may get lost, even if it still needed in the future.

In the live variable analysis, variables and temporaries (i.e., temporary variables) are treated analogously, with one except, an that the mentioned assumption at the of a block: proper variables are assumed live, as explained. For temporaries, the liveness analysis exploits knowledge about how temporaries have been generated in the intermediate code generated. For each intermediate results, for instance for compound expressions, a new temporary variable is created to hold that intermediate value *temporarily*. The way that works also implies that temporary variables are never (re-)used across the boundaries of a basic block. So that means, at the end of a basic block, temporaries are assumed to be dead, and that is more than an assumption; it reflects reality, at least as long as the intermediate code is generated the way described.

In the following, when we say "variable" we mean proper variables as well as temporary variables.

The question whether a variable is live or not refers to "points" in the program. So it's not about "is variable x live or dead?", it's about "is variable x live here". Obviously, at some points in the program, x may be used in the future, and, at other points, it may no longer be used. If held in a register, when a variable's status turns from live to dead, the register allocator may decide to re-use the register, which may involve to save the register back to main memory. But that will the register allocator's task in Section 10.8, here we are just figuring out the liveness information the allocator can make use of.

The "points" in the program here refers to “lines” in the straight-line code. Actually, it’s not actually that variable live in a given line, that is too imprecise. It’s actually question, whether a variable is live right before a given line, or right after it. One has to make that distinction, since obviously, the liveness status of a given variable can change at a given line. For example, for a statement `x := 4`, variable `x` is definitely dead before that statement, but may well be live afterwards.

Of course, the liveness status right after a line number n is identical to the live status right in front of line $n + 1$. This distinction between “right-in-front-of” and “right-afterwards” can also be applied to whole basic blocks. One can figure out, what is the liveness status of a variable right in from of a basic block, which means right in front of its first line, and right-afterwards. One cannot do that for a single basic block in isolation. For the same reason, one cannot for instance, figure out for a single line inside a basic block in isolation, say for `x := 4` from above, whether `x` is live or not afterwards inside. That’s also the reason why for local blocks and local liveness analysis only, proper variables are *assumed* live at the end; when doing only local analysis, one simply does not know what is the case. So when lifting the “right-in-front-of” and “right-afterwards” considerations to the level to whole basic blocks, that will be done for the global level, analysing a complete control-flow graph in Section 10.7. The corresponding information will be called $"in\backslash_{live}"$ and $"out\backslash_{live}"$.

Coming back to the local analysis of this section: it should be intuitively clear that it’s quite straightforward to do the liveness analysis, i.e. to determine the liveness status for each each variables and for each point after resp. before each line in block. A variable is live at a given point, if it is used later, but without being overwritten in the means. The situation where a proper variable is neither used nor overwritten for the rest of the block is not much different; same the assumption that temporaries are assumed dead at the end.

So to check if a variable, say `x` is live at a given point, one may be tempted to proceed forwardly and check for the following lines if `x` will be used in the future inside this block without being overwritten first (then it’s live), or the first thing that happens in the future is being overwritten (then it’s dead), or nothing happens the future inside this block, neither reading from it nor overwriting it, in which case the variable is assumed live resp. dead, depening on whether we are speaking about a proper variable or a temporary variable.

One can do that for all points in the straight-line code and for all variables, and that considerations shows that the determining the lifeness status inside a basic block is decidable. However looking for future uses of variables in the sketched way, checking each point in the program *independently* is absurdly inefficient.

The reason why it’s inefficient is that an independent checking partly obtains the same information over and over again. As part of the problem, one needs to determine the liveness status for a variable at a point say at beginning of line n (or for all variables at that point, it does not matter for the argument). For instance in Figure 10.6, assume we want to determine the liveness status for x for all lines. For instance, if one wants to determine it at the end of line 2, one can search forward. Assuming that the lines of the form `.....` have nothing to do with x, then the first use of x is discover in line 7, at which point it becomes clear that x is not live at the end of line 2.

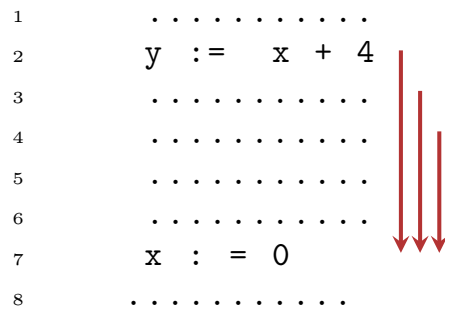


Figure 10.6: Local liveness

To do local liveness analysis means we need to do similar considerations for all points in the basic block. For instance for the end of 3 and line 4 etc, as illustrated by the other two arrows in the figure. Of course, figuring out the liveness information for the end of line 3 and 4 simply repeats the search done for the end of line 2 already. So, the different lines are better not treated as independent problems; one better reuses the information obtained for one line when doing another line. And the best way to do that is to proceed *backwardly*

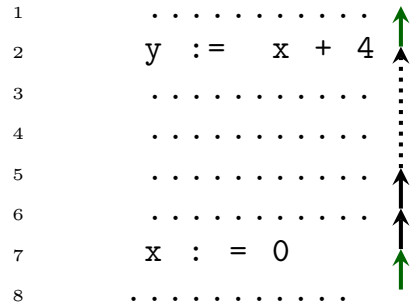


Figure 10.7: Local liveness: backward analysis

That is illustrated in Figure 10.7, again for variable x. Of course, the analysis that steps through the program in the sketched backward manner will treat all variables at the same time, not doing the same backward scan over an over for each variable. Anyway, proceeding backwards means, the analysis starts at the very *end* of the basic block. Assuming that x is a proper variable, it means, the x is *assumed* live at the end of the block. That's indicated by the green arrow. Stepping thought the code backwards, it remembers right in front of the assignment in line 7, resp. at the end of line 6, that variable x is dead now, indicated by the black arrows. Continuing the backward scan, this information is propagated though the lines with decreasing number, since nothing happens wrt. to variable x, until at the beginning of line 2 resp. end of line 1. At this point, the variable is *live* again, indicated the green arrow. The being live information is then propaged further on backwards, in the example till the beginning of the program. Actually, the liveness algo later will not just propagate the binary liveness information live vs. dead (here green vs. black), but it also indicates for live variable, the location of the next use.

That's an information that could be exploited by the code generator resp. register allocator. When it has to make the decision which of two live variables to keep in a register, preference could go to the one whose next use is nearer in the future. The actual code generator we look at in Section 10.8, does not actually make use of that information, resp. we don't go so deep into the details of the decision making process of the code generator to see in which way that next-use information of life variables can be used.

Figure 10.7 sketches how the "data" is propagated through the lines of the basic block, resp. information of interest *about* the data. The real data, integers in the example, is handled in the programs via assigning it to variables ("defs") an reading the variable later ("uses"). In this way, the data "flows" forward in an execution. After all, an execution does so in a forward manner. Here, the information of interested is not the data itself, but information about when corresponding variables are assigned to resp. read. This (information about the) data flows backwards. For straight-line code as in this section, that leads to a single pass through the code. In that sense the information "flows" through the code exactly ones, here backwardly (which corresponds to the fact, that the lines of a piece of straight-line code *in isolation* execute exactly onces, as well, though in a forward manner, of course.)

Going beyond straight-line code, there will be edges of the control flow graph to be considered. In case of multiple edges connected to a node, the information of the analysis will flow equally "both ways" (or more in case of more than two edges). In the more general setting, the basic blocks are then part of a control flow graph, which typically contains cycles. Thus, a single-pass of analysis is no longer sufficient, and the "data flow " *circulates* through the graph. That will be covered in Section 10.7.

This treatment, single pass for straight-line code resp. circulating data flow in a whole control-flow graph is characteristic for data-flow analysis. What is *not* characteristic for all of them is that the analysis data flows *backwards* through the straight-line code as in 10.7, resp. backwards through the graph as discussed later. Liveness is information about the *future*, i.e. whether there will be (or might be) a place where a variable is used. As explained, instead of seaching forward as illustrated in Figure 10.6, one arranges for a backward propagation of the relevant information. In other situations, one is interested in information about the past instead. For instance, analyzing wheher all variables have been properly initialized previously. This reverses the picture, and the corresponding analysis works by *forward* data flow.

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 not needed for sure 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 corresponds to static liveness (the notion that static liveness analysis deals with). That is hidden in the condition "given the level of abstraction" for example, using the given control-flow graph. A variable in a given *concrete* actual 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. We are concerned here with static liveness.

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
- u is **live** "in" (= at the end of) block 2, as it *may* be *used* in 3
- 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)

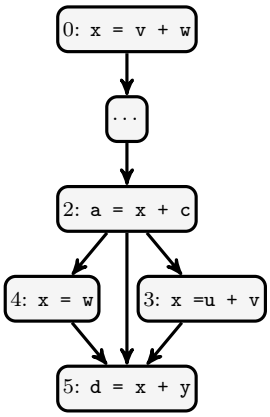


Figure 10.8: Defs and uses in a CFG

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)

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

We don't go into SSA, but we shortly mention it here, as it's a very inportant intermediate representation, which is related to the issues we are discussing here (data flow analysis, def-use and use-def). As we hinted at: there are many data-flow analyses (not just liveness), many of them quite similar concerning the underlying principles. Transforming code into SSA is an effort, i.e., involves some data-flow techniques itself. However, once in SSA format, many data-flow analysis become more efficient. Which means, investing one time in SSA may pay off multiple times, if one does more than just liveness analysis.

As a final remark: temporaries in our 3AIC within one elementary block follows the "single-assignment" principle. Each one is assigned to not more than once. The user variables, though can be assigned to more than once. For straight-line code, i.e., local per elementary block, having also the other variables follow the single-assignment scheme would be very easy. Instead of assigning to the same variable `a` multiple times, one simply renames the variables into `a1`, `a2`, `a3` etc. each time the original `a` is updated (and keeping track of the new names). So, for SLC, SSA is not a big deal. It becomes more interesting and tricky to figure out how to deal with branching and loops, but, as said, we don't go there.

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

We encountered a closure or saturation algorithm in other contexts, for instance when calculating the first and follow sets (potentially using a worklist algo). Also the calculation of the ϵ-closure is an example, and there are others.

10.5 Local liveness: dead or alive

After having discussed different aspects concerning data-flow analysis, including aspects of global analysis which comes later. Let's turn back to the more focused task of local

liveness analysis. We start first with a plain version, which does liveness analysis the way explained, but nothing more. The algo calculates, for each point in the basic block and for each variable, calculates whether the *binary* information whether the variable at a given point is variables is live or not, dead or alive, so to say.

That's straightforward and should be reasonably clear already from the informal discussion so far, in particular in connection with Figure 10.7, motivating the backward-scan idea. Later, we extend that version and introduce the concept of *dependence graph* in Section 10.6.

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
  - intention: use **registers**
- limited about of register available (platform dependent)

Assumption about temps (here)

- 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

We use the 3AIC from Listing 10.2 to illustrate the backward scan more conretely. The example will be reused also when extending later in Section 10.6 the current liveness algo which works with binary "dead-or-alive" information.

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

Listing 10.2: 3AIC code example

- 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
- 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 for liveness analysis.

In intermediate code generated the way disucssed in the previous chapter: temporaries are *always* generated new for each intermediate result, so t_1 in the example is "unrealistic" for generated intermediate code. But also that is not important for liveness analysis. It works for variables and temporaries alike, re-assigned or not.

Single step per line: transfer function

- 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 $x_1 := x_2 \ op \ x_3$

- 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

The definition explains for each line, how the liveness status in front of the live depends on the lifeness status at the end of the line. It does so for lines of the given form $x_1 := x_2 \ op \ x_3$; for other forms of lines, like $x_1 := op \ x_2$, the definition needs to be adapted in the obivous manner. Of course, statemens line `jump 3` need *not* to be considered for a block-local analysis. Jumps transfer control between different basic blocks.

Back to the 3 address assignment statement: as said, the definition explains the dependence of the flow information before the statement on the status of the information at the end of the statement. This dependence is a *function* from the exit of the statement to the entry of it. This functional dependency is called the *transfer function* (of that line, resp. the statement in a given line). Note that the liveness information of the *entry* point of a line is expressed as function of the corresponding information at the *exit*, not the other way around. This is of course characteristic for *backwards* analyses like liveness analysis.

```
// ------ 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,o] := T[i+1,o] (for all other vars o)
      T[i,x] := D // note order; x can ``equal'' y or z
      T[i,y] := L
      T[i,z] := L
end
```

Listing 10.3: Local liveness (dead or alive)

- 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

The table is a two dimensional, there is one slot per variable and per line. Each line can change the liveness information for one or more variables (that what conteptually the transfer function is doing) so the liveness information at the end of each line is different from that in front of a line. The entries in the table or two-dimensional array represent the information **at the end** of the corresponding line. That's of course the same as at the beginning of the next line. It's assumed that the line numbers go from 1 to n (not from 0 to n). The loop steps down to determine the effect of all lines numbered n to 1: note that what is called "current instruction" in the loop refers to the line with $i + 1$ in the code. Even of there is no instruction with line number 0, the corresponding entry representing the “end of line 0” respresents the liveness information at the beginning of the first line, i.e., at the beginning of the whole block.

Earlier we mentioned in passing the notion of *transfer functions*, without going into details. The code of Listing 10.3, stepping backwards through the lines does not *explicitly* make use of a separately defined transfer function. Implicitly, the transfer function is executed in the body of the loop, updating the entries of the table.

The result of the run for the code from Listing 10.3 is given in Table 10.2.

line	a	b	c	t_1	t_2
[0]	L	L	L	D	D
1	L	L	L	L	D
2	D	L	L	L	L
3	L	L	L	L	D
4	L	L	L	L	D
5	L	L	L	D	D

Table 10.2: Liveness analysis example: result of the analysis

The analysis operaters with binary information and thus the table contains binary information (dead or alive). Later in Section 10.6 we will extend that information and (mildly) extend the algorithm. Revisiting the same example means that we get a (mildly) extended version of this table. The extension is the following: for live variables, one does not report the fact that the variable is live, but also point to the line where it is used next.

10.6 Local liveness^{++}: Dependence graph

In this section we revisit the dead-or-alive algorithm from Listing 10.3. The previous version was binary, determining for every point in the straight-line code the liveness status for each variable. It needs only a minor extension to obtain better information. Instead of determining whether a variable is dead or alive inside the current block (resp. *assumed* live in case of a proper variable at the end of the block), one can determine for live variables, where resp. when they are used in the future. This is done below in Listing 10.4, a mild extension indeed of the one from Listing 10.3.

So, the extended algo keeps track of where the next use each variables will be. That's done here by tracking the *line number* of the next use. That's usually precise enough. One might also track where a variable is used *inside* a line, as the first argument of a operation or the second argument (or both). Anyway, we track the line only. I.e., the analysis works not with the binary information L and D, but for liveness, the information is $L(n)$, where n is the line number where the variable or temporary in question is used next. That next usage refers to a line *inside* the basic block. As explained, proper variables can also be *assumed* live at the end of a block (unlike temporaries, which are rated as dead). Of course, in such a situation, the analysis can't determine the line of the next use. We are currently doing a block-local analysis, so we have no information about subsequent blocks; that's why we just *assume* variables to be potentially live. Besides that, on a more global level, it makes no real sense of talking about *the* next use of a variable. Due to branching, there may be multiple next uses. If one wanted a next-use information, that would be a set of next-use points in the general case.

Anyway, here the situation where are variable is assumed live is captured by the notation $L(\bot)$.

```
// ------ 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,o] := T[i+1,o] (for all other vars o)
      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
```

Listing 10.4: Local liveness (with next use information)

The result of applying the algo on the 3AI code of Listing 10.2 is shown in Table 10.3. Since the algorithm is a straightforward generalization of the previous binary version, the new table is a straightforward generalization of the previous Table 10.2 (on the same example).

So, we see the next-use extension is really straighforward. We mentioned already in the discussions at the beginning of Section 10.4, in which way register allocation can profit from the extra next-use information and we could leave it at that. However, the next-use information that one can calculate by doing liveness analysis is closely related to another

line	a	b	c	t_1	t_2
[0]	$L(1)$	$L(1)$	$L(4)$	D	D
1	$L(2)$	$L(\bot)$	$L(4)$	$L(2)$	D
2	D	$L(\bot)$	$L(4)$	$L(3)$	$L(3)$
3	$L(5)$	$L(\bot)$	$L(4)$	$L(4)$	D
4	$L(5)$	$L(\bot)$	$L(\bot)$	$L(5)$	D
5	$L(\bot)$	$L(\bot)$	$L(\bot)$	D	D

Table 10.3: Liveness analysis example: result of the analysis

important concept resp. another intermediate representation. So we take the opportunity to discuss that shortly here as well.

The intermediate representation is known as **dependence graph**. We will (shortly) discuss it for basic blocks since we are currently doing *local* liveness. One can also generalize it to whole control-flow graph analogous to the fact that one can do liveness on a whole control-flow graph. But let's stick to the local level for now.

Let's assume we have a 3A code, like the 3AIC from example from Listing 10.2. 3A code instead of intermediate code would work similarly; likewise one could do a dependence analysis for forms of 2-address codes.

At any rate: a typical line in the 3-address code consists of a left-hand side and a right-hand side, as in instructions $x_1 := x_2 \ op \ x_3$. In such a line, x_1 on the left-hand side is "defined" and x_2 and x_3 are "used". The next-use form of liveness analysis figures out where inside a block for each point and for each variable, the next use will occur (resp. assumed somewhere outside the block).

One can generalize that even more, and track *all* usages in the future (inside the block, reps. somewhere outside the block). That's not a big extension. The register allocator may perhaps also profit from this more detailed information about the use of variables. But something else is more important.

If we track all future usages of a variable at different points in the basic block, we in particular have information in a line of like $x_1 := x_2 \ op \ x_3$ about the next usages of the variable *defined* in that line, i.e. x_1.

This information thus connects the definition of x_1 with all its future uses. Normally, one contents oneself to connect definition and usages per line (as we did with the next-use information). There is anyway only *one* variable on the left-hand side of each line, and whether this variable "definition" is later used as first operand or second operand in some line is not really important (though one could take that into account as well, if one feels the need).

Keeping it on the per-line level, it means one connects a line like $x_1 := x_2 \ op \ x_3$ with all lines later that *use* x_1 (without that x_1 is being overwritten in the meantime, of course). The later lines (resp. the uses in that later lines) then are said to *depend* on said line (resp. depend on the *definition* of x_1 on the right-hand side of that line).

So, that's a **def-use** situation, and an analysis that figures it out is a **def-use analysis**. We mentioned earlier, that def-use analysis is closely related to liveness analysis, and here we see more clearly how.

If one tracks the dependencies of the described kind (i.e., def-use connections) that results in a graph, the mentioned *dependence graph*. This is another well-known intermediate representation (different from control-flow graphs, ASTs, 3AIC, etc. and maybe used alongside those other representations).

Inside a basic block, the dependence graph is *acyclic*. In other words, the corresponding graph is a directed acyclic graph (DAG) with the lines as nodes and the (direct) dependencies as edges. Typically from the "defining" line to the "using" lines, where one says the using lines (resp. the right-hand side variables therein) *depend on* the defining one (resp. depending on the left-hand variable defined in that line).

Instead of viewing the lines and dependencies as DAG, one can equivalently view the lines as *partially ordered*. That is so, since each DAG corresponds to a partial order, and vice versa.

Why is that def-use information, i.e. the dependency graph relevant? It expresses ordering constraints, making clear which lines of the code (here 3AIC) needs to be executed *before* others, since the latter depend on the former. This dependency is only a partial order on the lines of a basic block, not a total or linear order. In other words, some lines are independent: there is no dependence directly or indirectly in either direction. Being independent means, the order or execution is irrelevant. In other words, the line-wise linearization in the 3AIC (or later 3AC or 2AC etc.) is a particular linear arrangement, more strict than actually necessary given the partial order of dependencies.

A *dependency analysis* could reveal the looser partial order and thus reveal whether 2 statements need necessarly be executed in the order as listed in the code, the latter one depending or the earlier one, or whether the compiler could *reorder* then. That is known as **out-of-order** execution. Figuring out a good order of execution, in case of independent instructions is known as **instruction scheduling**. Actually, also the processor can have facilities of out-of-order execution of machine code instruction, but that's outside the control of the compiler. However, knowing the rules and conditions under which a processor does out-of-order or overlapping execution could be exploited by a code generator "scheduling" the instructions in a way that suits well to the platform's corresponding capabilities. For that, the compiler needs to figure out dependencies of the data (and registers, etc.).

This explanation analyzes the linear IR of, say, 3AC or 3AIC to determine which orders in the sequential arrangment of lines of instructions are real and which are spurious. With that help, one can rearrange the linear code, if deemed profitable. An alternative view is **not** to take the linear instruction sequence as primary intermediate representation, for instance inside a basic block. One would use DAG-based representation instead, i.e., the dependence graph and linearize the DAG in a subsequent stage. These alternatives are comparable to the situation with control-flow graphs. One can use them as intermedate representation to a lower level intermediate representation. Or, analyze a lower-level representation like the machine code or intermediate code to "reconstruct" from the linear

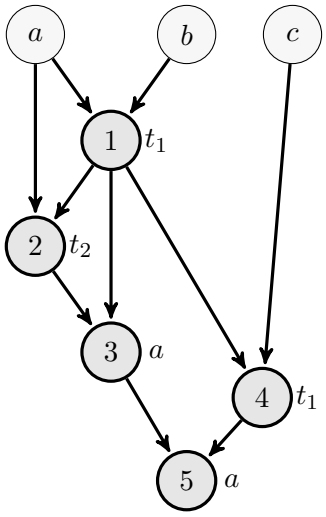


Figure 10.9: DAG for the 3AIC code block

representation the control-flow graph (as done by the simple partitioning algorithm in Section 10.3).

Revisiting the example

So much about motivating dependency graphs and what they represent. To get a clearer pictures let's look an example, and let's look a Listing 10.2 again. The result of the next-use analysis from Table 10.3 leads to a dependence graph of Figure 10.9. We see that the temporary defined in line 1 has three uses, namely in the lines 2, 3, and 4. In the linear code arrangement, the next use of the definition t_1 in line 1 is by the subsequent line 2. The DAG also makes clear that those lines 2, 3, and 4 are independent and could be executed in any order (or in parallel). These three edges correspond to the fact that in Table 10.3, the definition or assignment to t_1 in line one is marked as $L(2)$, $L(3)$, $L(4)$ in lines 1, 2, and 3.

In line 4, t_1 is "re-defined", i.e., assigned to again. Therefore, the entry $L(5)$ in the table does not refer to the first assignment to t_1 in line one, but the assignment in line 4. Remember, in Table 10.2, the stored information corresponds to the next-use or liveness information at the *end* of the corresponding code line.

ASTs and DAGs

Let's have a last look at the DAG, and explore the connection with ASTs. In principle, this is not new information, we have introduced both concepts, but perhaps it worthwile to spell it out more explicitly, looking at a few more examples. Let start with the 3AIC from $x := (x + 2 * z) - (a + b)$ in Listing 10.5.

```
t1 :=  2 * z
t2 :=  x + t1
t3 :=  a + b
```

```
x := t2 - t3
```

Listing 10.5: 3AIC for $x := (x + 2 * z) - (a + b)$

The corresponding dependence graph is shown in Figure 10.10. As inner nodes of the DAG, we use the line numbers from 1 to 4. The inner nodes in the picture are also labelled with the variable or temporary being *"defined"* in the node. The first three lines calculate the side-effect free expression on the right-hand side of source code assignment, and the "numbers" of the temporaries $t_1, \ldots, t_3$ correspond to the line number; that's the way the intermediate codegenerator works (if we assume "numbered" temporaries starting from 1). The DAG shows also a node correspond to the constant 2. Normally one would not bother to include the node and the corresponding edge into a DAG. The outcome or value of t_1, which is "defined" by the right-hand side $2 \times z$ does not depend on 2 insofar it is a constant anyway, it depends on z though.

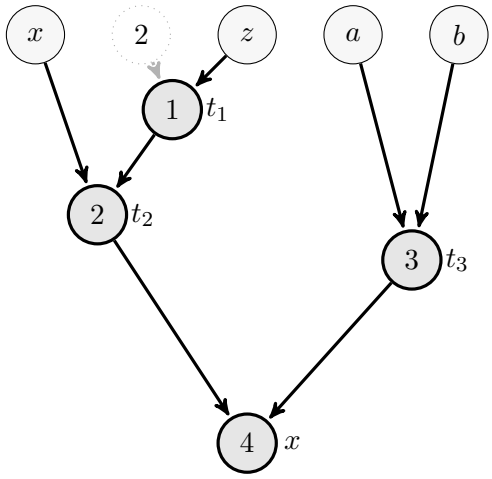


Figure 10.10: DAG of the code example

It should not come as a complete surprise that the dependence graph from Figure 10.10 is basically nothing else the abstract syntax tree of the right-hand side $(x + 2 * z) - (a + b)$ upside down; Normally, ASTs are written, like trees often are, with the root node on top. For the DAG, I choose to write it the other way around, so that the "def"-nodes come before the "use"-node, i.e., higher-up, as in the code and the dependency edges go down. An AST is shown in Figure 10.11.

So, determining the DAG from a piece of 3AIC reconstructs in some way the AST. At least conceptually and in this example. Of course, the AST as concrete data structure is most probably represented differently from the DAG, and the two structures, if a compiler uses them both, serve different purposes. That the AST and the DAG are basically the same in this example is also caused by the fact that the code example is very simple: just an assignment to a variable with a pure, side-effect free expression on the right-hand side. As seen in the corresponding chapter, the intermediate code generator simply traverses the abstract syntax tree, generates a fresh temporary for each intermediate result, i.e. or each inner node of the AST. So we obtain three temporaries t_1, t_2, t_3, which are *defined* i.e., assigned-to, in the corresponding lines of the 3AIC in Listing 10.5. These

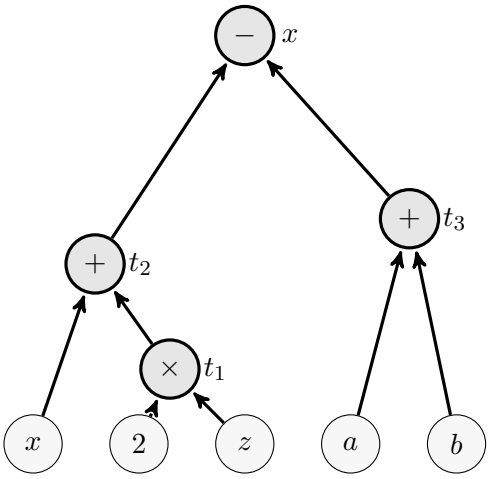


Figure 10.11: The AST of the assignment

lines or temporaries are the source nodes of the DAG, the soucre node of an edge is always the "def", the target node(s) are the "use(s)". In the simple example, each defined temporary has exactly one use, which correspond to the parent node. That makes the DAG in the example a tree (the syntax tree of the expression on the right-hand side of the assignment).

The connection between AST and dependency graph is less close in more general situations, like the DAG for a basic block and even less for a complete procedure; actually in the presence of loops in the code or cycles in the CFG, also the def-use dependencies may no longer be *acyclic*, i.e., the dependence graph is no longer a DAG.

In the simple situation of the DAG from Figure 10.10, the tree structure illustrates a fact which we knew all along: in a pure, side-effect free expression, it does not matter in which subexpressions are executed. The nodes or the different subtrees are indepdendent in the DAG, i.e., unconnected by dependence edges.

In general, there could be additional reasons why the DAG does not resemble the AST, not even in a simple situation like that. Here, for expressions and basic blocks, the connection between AST and code, here 3AIC, is very direct. For expressions, the connection is so direct and close, that we can effectively revert the translation, so to say, *decompile* the code from Listing 10.5 back to the AST. The connection may not be so direct. The 3AIC may have undergone some optimization. Same for the AST from the source code. If we look not at 3AIC, but at machine code (for which one could likewise do a dependency analysis), the distance is even larger, end additional optimization may have done.

The compiler resp. compiler related tools could even make effort to make decompilation harder resp. harder to understand the result of a decompilation. Often, that is done, however, at the source-code level. In that case, it's known as source-code obsfuscation or code hardening or encryption.

To conclude, let's have a look at another, slightly more complex example $(x := x + 3) + 4$. The corresponding code is given Listing 10.6; We have seen a quite similar example and

the code already in the chapter about intermediate code generation. It's an "expression" containing side effects.

```
t1 = x + 3
x  = t1
t2 = t1 + x
```

Listing 10.6: 3AIC for $(x := x + 3) + x$

The corresponding dependence graph is shown in Figure 10.12. This time, it's not a tree, but still a DAG. Note also, the order of evaluation in the expression now *does* matter, unlike before. The generated code assumes that the arguments of a binary operator like + are to be evaluated from left to right. The generated 3AIC does exactly that, the code of the left operand comes first. In the top-level addition, the x on the right-hand side uses the incremented value of x. That's also visible in the dependence graph in the edge from 2 to 3. The lines of the code cannot be reordered of course. Corresponding, changing the source code to $x + (x := x + 3)$ will change the meaning of the program.

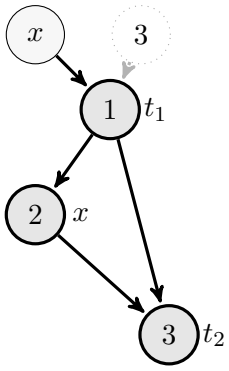


Figure 10.12: DAG of an expression with side effects

Connection to SSA

Starting from intra-block liveness analysis, we took the opportunity to introduce next-use information resp. introduced def-use analysis and the notion of dependency graphs, here DAGs. We take another opportunity and discuss to some extent another important concept used in intermediate representations, and that's the notion of *single assignment* formats.

We can stratch only the surface of it. In particular, since we focus on basic blocks. If generalizing to whole control flow graphs, additional complications would enter the picture, which we don't cover here.

Nonetheless, the conceptual core of single-assignment format can be understood here. Actually, focusing on on straigh-line code it's almost trivial, and is also connected to the def-use analysis and thus live variable analysis. So here is a good place to introduce some ideas behind the single assignment format.

The section header mentioned *static* single assignment or SSA. For straightline code, there is no difference btween static single assignment and (general) single assignment. Statically, a variable is "single-assignment" in a piece of code, if there is at most one assginment to it mentioned in the code. In the terminology here, there is statically at most one definition of a variable. We can exclude the degenerated case of variables that are defined but not used. Those are useless, they are never live at all. If we exclude that, the requirement for static single assignment is that all variables are defined exactly once resp. assigned to exactly one. Of course there are in general more than one use per definition.

In the presence of loops and procedures, the fact that there is exactly one place in the code where a variable is defined, that does not guarantee that, at run-time, that variables is assigned-to only once. A program that assigns every variable only once at run-time is single-assignment, which is (much) more restrictive than to be in static single assignment or SSA format. For blocks of straight-line code, like we are discussing here, there is no difference in SSA and single-assignment. Being defined textually once in a basic block means, it's assigned-to exactly once per execution of that block (if no "exceptions" derail the execution and prevent the assignment).

Actually, the same could be said about acyclic control-flow graphs; those could originate from a program using conditionals, but not loops. See for instance the one from Figure 10.8. Also for code of that shape, assigning every variable once implies single assignment. But let's stick to basic blocks.

Looking at Listing 10.2, that one is **not** in single-assignment format. Both variable a an temporary t_1 are assigned to twice. As mentioned earlier, the code from that listing cannot be the result of the intermediate code generator we discussed, at least not directly. Maybe indirectly via some optimization or other, or manually give, but it does not matter: it's anyway a good idea that liveness analysis or dependency analysis works not just for some particular way of generating (intermediate) code.

But, the code generator would indeed *not* reuse t_1 in the second assignment but would use t_3 instead. Of course, the use in the last line of what is now t_1 would then refer consistely to t_3 when using the definition of t_3 in line 5. In other words, the intermediate code generator already generates temporaries that follow the single-assignment pattern. If a global counter is used for the temporaries, it's even in single-assignment format globally; in particular and additionally, temporaries don't transfer data between blocks.

So far so good, but what about proper variables, not temporaries. In the code example also the proper variable a is assigned to more than once. Such a "re-definition" would come from situations, where a source code level, a variable is assigned to more than once in one piece of straight-like code, and the code generator dutifully generates code that does the anologous thing at (intermediate) code level.

But is straightforward to obtain intermediate code that avoids that. Instead of reusing a, one simply uses different variable as shown in Listing 10.7. Typically that's done by first generating code without adhering the the single assignment format, which is then translated into the format afterward. That is done by consistently renaming or "re-indexing" variables like a in the example. In the code, we assume that a_0, b_0 and c_0 refer to the versions resp. the values of the three variables coming from *at the beginning*. It's like the *input* values to the basic block.

```
t1 := a0 - b0
t2 := t1 * a0
a1 := t1 * t2
t3 := t1 - c0
a2 := t3 * a1
```

Listing 10.7: 3AIC code example (single assignment)

For basic blocks, one can easily achieve that in one pass, directly doing single-assignment. The reason why it's mostly done in a 2-stage manner is, that, while for straight-line code, (static) single assignment is trivial, the generalization to branching code (including code with loops) requires additional insight and tricks. Only then one would even speak of "the compiler uses SSA as intermediate format".

Terminology aside, the appropriately renamed code leads to the dependence graph of Figure 10.13.

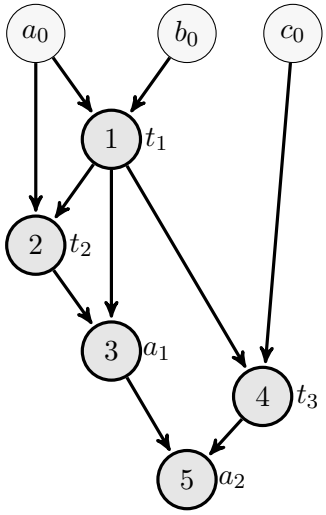


Figure 10.13: DAG for the 3AIC code block

One could get the impression, that's all fine and good, and actually pretty straightforward, in particular for straigh-line code. So, what's the big deal with SSA as intermediate format?

In a way, it's another angle or elaboration of the liveness-analysis, next-use analysis, and dependency or def-use analysis. After transforming a block into single assignment format, the uses of a variable being defined at some point are directly visible from the code: it's all the mentionings of the particular variable after the point of definition. Analogously, live span of a variables is directly visible from the code: from the point of being defined until the last mentioning, at least on a general level. Inside a basic block, as we *assume* variables to be live at the end, the local live span of each variables starts at the point of their definition and lasts till the end of the block. That should be plausible: if variables are never overwritten because of single-assignment, they never become dead (if we assume live at the end). Basically, local liveness becomes pretty trivial: for variables "defined" in a local block, they are dead before the (unique) defining line, and live afterwards. For

variables defined (or assumed defined) outside the block, like the a_0, b_0, and c_0 in the last single-assignment example, they are live throughout the block.

Also in a more global setting of control-flow graphs, the connection of defs and uses of variables is clear from the names of the variables and temporaries: all other mentionings of a variable defined in one line must be uses. So, the variables carry the def-use information and most of the (global) liveness information in their name.

Of course, that's not for free. As discussed coming up with that format actually *requires* to do basically something like liveness analysis, not in it's binary form we started out with, but more refined, but still analogous, and additionally following a proper renaming scheme. For fairness sake, for the global level, there are additional complications, beyond straightforwardly generalizing the binary liveness information and some easy renaming, but we don't go there in this lecture.

To have SSA as a format that makes liveness analysis quite simple does not in itself breathtakingly useful. After all, one could do liveness analysis straightforwardly without doing that intermediate format (which, as said, gives additional challenges to overcome not discussed here focusing on the local level). The importance and popularity of SSA comes from the following:: the format does *not only* help for live variable analysis. As hinted at, there are many data flow analyses one might want to do, which serve different purposes as liveness analysis, but using similar techniques, and many of them similarly profit from that format. So the effort to transform the code into SSA may pay off multiple time, in case the compiler employs multiple analyses, not just liveness analysis, on the given level of abstraction.

We don't look at other analyses, we focus on liveness later when talking about global analysis.

10.7 Global analysis

We have discussed general ideas behind liveness analysis earlier and covered local liveness in Section 10.5. Additionally, in Section 10.6, we discussed closely related concepts, like next-use analysis, def-use analysis and the concept of dependency graphs. We did so mostly focusing on the local level, i.e., on basic blocks, only hinting at that some things change or get more involved when doing analysis for a whole control-flow graph.

Here we fill in a few more details on this, but without also covering explicitly extensions like dependency analysis again.

There are basically 2 complications to deal with, namely *branching* of the control flow and *cycles*. Both originate from control-flow constructs in the source code, like conditionals and loops.

Branching is conceptually the simpler problem, though if one has loops in the source language one invariably also has to face branching. Indeed, on the 3A(I)C level, there are no loops, there are just jumps and conditional jumps. Conditional jumps obviosly leads to branching, the true-case and the fall-through alternative, so the block a conditional jump at the end has two successor blocks or successor nodes in the control-flow graph-

But also jumps typically may involve branching: the block jumped to, which is stared with the corresponding jump label, may be entered also otherwise, for instance with by a fall-through or being the jump target from different sites. though in a “backward” manner: the node starting with a jump label may have more then one predecessor node in the control-flow graph.

Let’s start with branching and let’s dicuss it for liveness of variables. Conceptually, with branching, one does no longer tries to find out if a variable, at a point *is* live or is *not live*, at least not exactly. We mentioned earlier that *approximation* is characteristic and crucial for all kinds of semantic analysis, and in particular for data-flow analyses, like live variable analysis.

Dynamically, i.e., at run-time, at one given point in a program execution, the liveness information is still binary: a variable will be used in the future, or it will be not. At lest it’s binary when we are dealing with deterministic programs and if we ignore external influences, like that the operating system crashes etc. (but that can be seen as a form of non-determinism as well, some external and unforseeable factors outside of the control of running program). Whether or not a variable (or address) will be used in the future at a given point in the executing is the question of *dynamic liveness*. That’s of course undecidable in general and that’s not what data-flow analysis aims at.

It gives an approximative answer, using the control-flow graph as level of abstraction. As far as branching is concered, data-flow does not try to find out which branch is taken. Not knowing which branch is actually taken. the approximation explore both, combining the information. Indeed, a running program will often explores both branches, though at each given point in time, the program either turns left or else turns right. That’s in partcular the case for loops (unless they run forever or they are never entered): for some number of iteration, the body is entered, and when the termination criterion finally is satified, the exit-edge is followed.

For static liveness, we have to keep in mind what we need the information for. The intention is to support register allocation. In particular the code generator will consider a register containing a dead variable’s value as “free” and reusable for other values. That means, if we mistake a live variable for being dead, that easily can lead to erroneous code; the compiler is incorrect. The opposite mistake, rating a actually dead variable to be live may lead to a missed oportunity of reusing a corresponding register, but that’s not an error. The code may just been slower than it could have been without making that misjudgment.

For live variable analysis it means, when in doubt count a variable as live. We did the same at the local level, when judging variables live at the end of a block (since locally one does not know what actually is the case). Same principle here: when facing a situation in a graph with two alternatives, one where the variables is used in the future, further down the graph, and another where it’s not used, the variable needs to be rated as live.

So, the approximation for liveness is about whether the variable **may** be used in the future, not that it’s guaranteed that it will be used (**must**). There are different data flow analyses that work with a must-kind of approximation instead of may-type. It’s the difference between over-approximation and under-approximation. Which one is the right choice depends on what the compiler does with this information, the intended usage. For

live variable analysis used for register allocation, it has to be over-approximative (may be live). We don't look at other data flow analyses for other purposes, so we don't cover in the lecture must analyses, though if one knows how to do a may analysis like liveness, it's straightforward to do a must analysis.

One may even say, indirectly we do a must-analysis. If we would prefer to call what we are /dead-variable analysis instead of a live-variable analysis, and looking for situations where variables are dead, then, what we need are situations when it's guaranteed that the variable is dead. In that sense, it's a matter of perspective anyway.

Now, approximation of relevant information, like "may the variable be used in the future" is the way to deal with branching. As described earlier, for straight-line code, the way the liveness algorithm propages this information is *backwards*. That will also be done for the global liveness analysis, i.e., the information flows in the reverse direction of the edges of control-flow graph: from suceesor nodes to predecessors.

Now, what about **cycles** in a graph? That is a different problem and makes the problem harder. That refers the complexity of the problem, but also the required theoretical background. We don't go into the latter here, we just hint at why it's harder and what to do about it, without explainging why it actually works, resp. under which circumstances it works. For us it's enough to know that for live variable analysis the sketched approach does indeed work. Same as we did for many other algorithms in this lecture.

As explained, the local liveness analysis "walks" through the code in a single pass, namely in a backward manner. The same can be done if one had a branching structure without loop, like for instance in the CFG from Figure 10.8. The only thing that is more complex compared to local analysis is that one has to treat the liveness information approximative. In the graph of that figure, that would concern the treatment of node 2, where the three "flows" coming from below merge.

In the presences of cycles, one cannot expect to propagate the information one time along each edge and through each node and be done. Information propagadge through the graph, say in tendency "upwards" in a picture, will in a loop also be propagaged back down again to a place already explored.

In that way the information, for instance about liveness status of variables, *circulates* through the graph, sometimes "going" through a cycle multiple times. That directly makes that task computationally more complex than the single-pass approach that suffices to deal with acyclic structures. Also termination of the data-flow analysis may be of concern, though actually for live variable analysis in our setting (and similar data flow analysis) termination is guaranteed. It's only not 100% immeadiate as for acyclic structures, where the analysis stops after having treated evey line or node exactly once.

The characteristics of the core data flow algorithm (for liveness and others) is captured by fixing 3 aspects:

> Firstly, with wich (liveness) information should the data flow start. Secondly, how to repeatedly propagate the (liveness) information during the analysis. "Repetion" there will be a loop whose body "lets the data flow". Finally, when to terminate, i.e., exit the loop.

These three aspects shape the general skeleton of the data flow algorithm. There will be an initialization, a loop, and the loop has an exit condition. Actually, the simpler liveness analysis for basic blocks from Listing 10.3 is of that shape.

The difference to the global setting is that the termination condition is more complex. The straight line code version simply stops after having treated the first line in its backward pass through the code. The second difference is that the traversal for the graph is not so rigid, like backwards. In the presence of loops, when following the edges, there is no single possible plausible strategy, and one generally has to treat the nodes of the cycle multiple times anyway.

So far the general picture of how the algorithm is shaped. We won't give pseudo-code for it, we mainly explain by way of examples how it works. But still we fill in some details of the skeleton, sketching what actually is done during initialization, what step(s) are iterated, and when actually to stop.

We discuss it without makeing a difference between temporaries and variables. The algo treats them analogously. That's different than for the local anlysis only, which assumed temporaries dead at the end of the block. We don't need to assume that here, like we don't need to assume that variables are live at the end of a block. With the control-flow graph at hand, the global analysis (approximatively) figures out which variables are statically live and which not, and for the temporaries it will figure out that indeed they are all dead (for temporaries generated the way described). No need to "assume" anything. At the very end of the whole program, all variables are dead for sure, proper variables and temporaries alike.

initialisation: minimal information all variables are assumed dead.

increase repeatedly the body of the loop treats one element of the graph, like one node or one edge and updates the current liveness information. The update works in a **monotone, increasing** manner: a variable previously still considered dead is flipped to be rated live, but never in the other direction.

termination by stabilization when no more information can be added, so no more live variables at some places are detected, the algorithm stops.

We can picture it as follow: the algorithms starts with *no knowledge* about liveness status, and considers for a start all variables dead at all places. The same was done for the liveness analysis for basic blocks from Listing 10.3 (except that for the last line, the proper variables were assumed live). We can see that starting point as absolute *minimal* information and discovering more variables at more places live during the analysis can be considered to *increase* the information. It's not so much that there is a larger set of live variables detected at some point. It's not so much that the sets (of live variables) grow larger (though they do). It's more that the amount of confirmed information about the liveness status at different places increases. If, at some place, the current status in the algo of a variable switches from dead to live means that the algo has explored and confirmed that there is a potential path to a future use of the variables. Once established, further exploration may find further statically live variables at other places, but the liveness information added right now never has to be revered back, once establed by new information discover later.

In that sense, the liveness information steadily grows and never shrinks, i.e., monotonously increases during the iteration of the algo. This is crucial and characteristic for data flow analysis.

It also makes clear when to stop, namely when the information cannot be increased any more. The algorithm reaches stabilization or it saturates or a "closure".

Indeed, one finds this "monotonously adding information until stabilization" idea not just for data flow analyses. For instance, the first- and follow-set calculations worked similarly. Also there, one basically explored a graph, namely the way the non-terminals of the grammars hang together. Since context-free grammars use recursive definitions, the corresponding "graph" contains cycles (we never drew the grammars as graphs, we noted them in BNF...) So, for instance, the set of terminals (or ϵ) confirmed to be in the first-set of various non-terminals increases until no more such information is added, at which point the first-set saturation procedure terminates.

Basic blocks and live-in and live-out

Before presenting the liveness mainly by way of examples, we explain an practical aspect, namely the treatment of basic blocks. We mentioned that in passing earlier. For efficiency, it's best to analyse basic blocks, the nodes or the control flow graph first and summarize the corresponding information. That will allow the global analysis to treat them effectively as if they were single lines.

This summarizing local analysis is not identical to the local liveness information we did, but it basically works the same (like steping backwards through the lines). Here, it's not about whether a variable is (assumed or factually) live of dead in the different lines of the basic block. For some variables one cannot determine that locally (for some one can). It's about *changes* to the liveness status of all the variables. It's much like looking to one line as in the local analysis. Consider one single line of the form $x_1 := x_2 \ op \ x_3$ as they were treated in the local analysis and let's assume all three variables are all different).

For that line, it's clear that at the beginning of the line, x_1 is dead and x_2 and x_3 are live. Considering that as a change of information from the situation after the line to that before that line, it's as following. The situation for x_2 and x_3 will be set to live, and x_1 to dead, independent from how it there status is (currently) afterward. For all other variables, the information from after the line is left unchanged.

Basically, one can do the same for basic blocks, only more than just 3 variables may change their status. We will not show code how do calculate that, it's easy enough.

At any rate, relevant for the global analysis is not what happens line by line in the basic blocks, relevant is only the situation right in front of each basic block, resp. right afterwards. This is also called *inLive* (in front) and *outLive* (afterward). What the global analysis needs to know how the *inLive* and *outLive* information hang together. Doing a backward analysis, in particular how, for a given block, the *inLive* information is calculated for a given *outLive* information, not the other way around.

And therein lies the improvement: having precomputed the block-local effect on the flow of lifeness information per block, the global analysis can just use that, and avoid stepping

through the individual lines of the bloks over and over. When occuring as part of a cycles in a graph, blocks will have to be evaluated more than once in general, and it's to be expected that the precomputation will make the analysis more efficient. Blocks not occurring in a loop would not profit from a pre-computation.

Anyway, pre-computing the effect or not will not influence the result of the analysis, only perhaps the running time.

In the examples later, we don't precompute anything explicitly, the figures illustrated the *inLive* and *outLive* information and whether this is the result of being smart and having precomputed some bits or whether one does it over and over again, is not visible, and the outcome, as said, is the same anyway.

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 converning 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 determined 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 neeeds 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 variables[4] 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

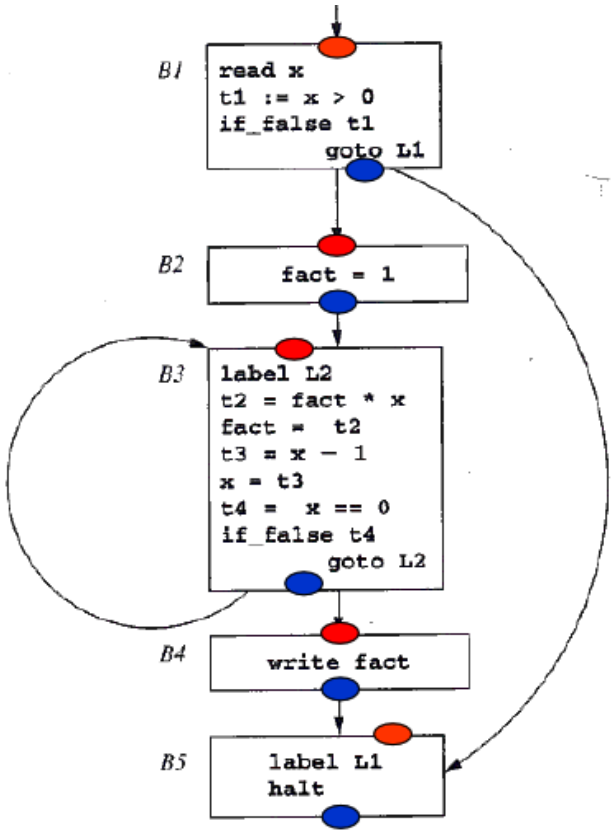
Judging a variable (statically) live: always *safe*. Judging wrongly a variable *dead* (which actually will be used): **unsafe**

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

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

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 ⇒ 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)

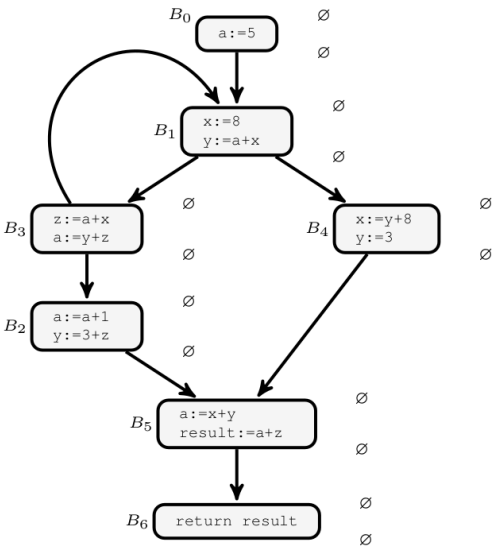
Example

```
    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 $\emptyset$ 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 ($\emptyset$ everywhere)

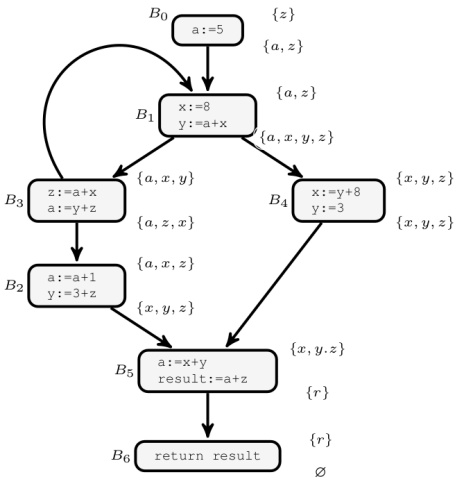
Loop pick one node & update (= enlarge) liveness estimation in connection with that node

Until finish upon stabilization (= no further enlargement)

- order of treatment of nodes: in princple arbitrary[5]
- in tendency: following edges **backwards**
- comparison: for linear graphs (like inside a block):
  - no repeat-until-stabilize loop needed
  - 1 simple backward scan enough

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

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):

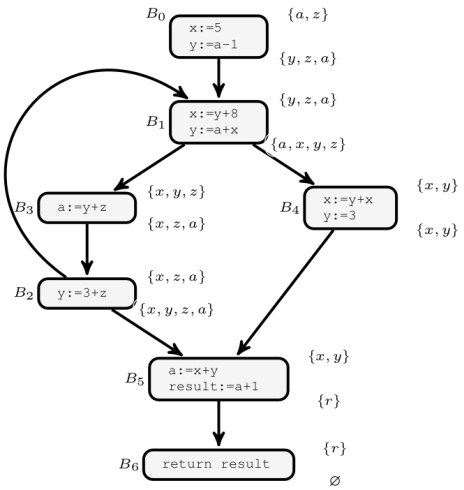
"harmless loop"

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. . .)

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 chosen 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)

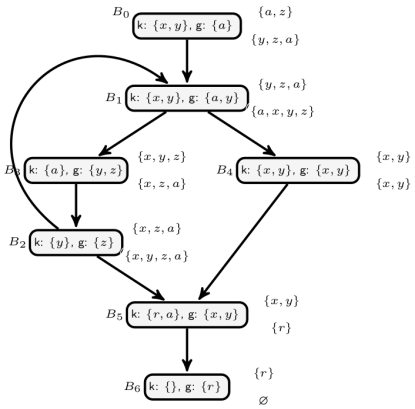
$$inLive = outLive \backslash kill(B) \cup generate(B)$$

- 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 oder of kill and generate in the definition of transfer functions. In principle, one could also arrange the opposite order (interpreting kill and generatate 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



10.8 Code generation algo

Finally, we cover the code generation proper. We focus on generating code for straight-line code and register allocation. Indeed, translating jumps and conditional jumps from 3AIC is not very complicated. The intermediate code is already a linear code from, and one can expect that the (conditional) jumps have a more or less direct correspondence in machine code. Ultimately, one will have to get rid of the labels labels. The serve in the intermediate code as symbolic addresses, and need to be replaced by real addresses, in a first stage say, *relocatable* addresses. That involves calculating with the actual byte sizes of the commands for the instruction set of the target platform. For instance like the size shown in Figure 10.1 earlier.

Properly accounting with the concrete instruction sizes to obtain concrete (relocatable) addresses has nothing to do with register usage and is an indepedent problem. As said, we focus on register allocation for basic blocks, making use of liveness information. This focus does not mean that the code generator uses only local liveness information. Also liveness analysis is an independent problem, and the code generator will be correct as long as the liveness information is correct, i.e., a safe over-approximation of the actual future use of variables.

The code generation will proceed line by line through the 3AIC, and it will *forwardly*. The generator will make decisions concerning register usage *on the fly*, i.e., while generating the code. To do so, it obviously needs to keep track of which registers are currently in used and for which variables (resp. which registers will be in use at the different points in the code once the program will be run, of course.) To do that kind of book-keeping, the code generator will maintain specific data structures. They are called *register descriptors* and *address descriptors*. This way the code generator has an overview which variable is stored in which register(s) (if any) and at which address it resides in main memory. That's the address descriptors. The register descriptor information records at each point for each register, which "variable(s)" it contains if any. More precisely, to which value of which variable(s) the current register content corresponds, if any.

In principle, the address descriptor information would be enough actually: if one has that information for all variables, the code generator can figure out the register usage by searching through the corresponding table, in some form of reverse look-up. That is of course inefficiecent, and the code generate is better of keeping track of the relevant information in two tables.

Aside: Graph coloring register allocation

As said, the code generator generates 2AC instructions on-the-fly including making decisions on which registers to use. That's not the only way one can do register allocation. An well-known and widely used approach is known as register allocation by *graph coloring*. We don't cover that, but since it's a standard approach, it's well worth mentioning. The idea is actually simple and elegant: first get an overview over the *live-spans* of variables. Live spans are particular simple for code in SSA format, insofar a particular variable becomes live at some point, stays live for some while, and then becomes dead. For non-SSA usage, a variable may switch from dead to live and back multiple times.

Be it as it may, knowing the live spans is important insofar: two variables with an *overlapping* live time cannot occupy the same register, they need to be in different ones. If course one could try to keep a variable in one register for some time, kick it out from the register for a short while and store it back to main memory, to make room for another one for a short while, perhaps there is only as short period of overlap, and then, later load it back again into the same register (or a different one). But there is only so much sophistication one can do, and juggling values back and forth between registers and memory is costly at any rate.

So a clear and useful arrangement is the following: if it's decided to place a particular variable in a register, the association is fixed. The variable is put into the designated register at the start of its live span, it's keep there it there during the live span, and will be saved back to main memory at the end of the live span, if the value in the register has changed in the meantime. That is a particularly clear strategy for SSA-style code.

In the sketched strategy, having an overlap in live times has the mentioned consequence: the register allocator has to select two different registers for the two variables. Variables with such an overlap are said to be **in conflict**. One can represent then the conflict situation via an undirected *graph*. Variables are the nodes of the graph, and conflicts are the edges. The number the nodes in the graph corresponds to the number of variables we

need to take care of. The register allocation task is to color the nodes with registers in such a way that two neighboring nodes don't reside in the same register, i.e. have different colors. Typically, the number of nodes in the graph exceeds the number of available registers, otherwise the problem would be trivial.

This is a particilar graph coloring problem, the registers correspond to "colors" (there are different graph coloring problems). Solving it is a problem of high computational *complexity*, i.e., finding an answer to the question: Can the given graph be colored with the given colors (and if so, how).

So graph coloring register allocation does not attempt to solve the exact graph-coloring; that would cost too much time. Instead, the allocator would do a decent effort, choosing colors or registers for the variables avoiding conflicts as long as possible. But when no non-conflicting choice is possible for the next node, not attempt is made to try again with a different coloring scheme to see if that turns out to be more successful. Instead, one simple resorts to the main memory ("spilling") for the node that cannot be colored right now, and then the allocator proceeds with coloring the rest.

Details of the register allocation may become involved for practical languages and platforms, starting already with the fact that some platforms put restrictions on what registers can or have to be used for what, and other complications and fine-tunings. However, the basic idea is elegant and straightforward and hopefully understandable from the high-level description. Graph coloring register allocation is widely used. The original proposal is described in a software patent (by IBM), one quite early software patents. Not everyone agreed and agrees in which way or to which extent ideas like that can be patented. In this particular case, the graph coloring idea directly employs a recursive strategy described over 100 years ago, tackling in a heuristic manner a particular graph coloring problem.

Simple code generation algo

- simple algo: *intra-block* code generation
- core problem: **register use**
- register allocation & assignment
- 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

Some make a distinction between register *allocation*: "should the data be held in register (and how long)" vs. register *assignment*: "which of the available registers 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 $b + a$

The limitation that read-only variables are not put into registers is not a "design-goal": it's a not so smart side-effect of the way the algorithm 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 discussed so far), the code generation could profit from that by getting more efficient. But its *correctness* does not rely on that. Even without liveness information at all, it is correct, by assuming conservatively or defensively, that all variables are always live (which is the worst-case assumption).

We decompose the code generation into two parts, discussed separately: the code generation itself and, afterwards `getreg`, as auxiliary procedure where to store the result. One may even say, there is a third ingredient to the code generation, namely the liveness information, which is however, calculated separately in advance (and we have discussed that part already). The code generation, though, goes through the straight-line 3AIC line-by-line and in a *forward* manner, calling repeatedly `getreg` as helper function to determine which register or memory address to use. We start by mentioning the general purpose of the `getreg` function, but postpone the realization for afterwards.

As far as the code generation may is concerned: finally there's no way around the fact that we need to translate 3-address lines of code to 2-address instructions. Since the two-address instructions have one source and the second source is, at the same time, also the destination of the instruction, one operand is "lost". So, in many cases, the code generation need to save one of its 3 arguments in a first step somewhere, to avoid that one operand is really overwritten. We have gotten a taste of that in the simple examples earlier used to illustrate the cost model. The "saving place" for the otherwise lost argument is, at the same time the place where the end result is supposed to be and it's the place determined by `getreg`.

Of course, there are situations, when the operand does not need to be moved to the "saving place". One is, obviously, when it's already there. The register and address descriptors help in determining a situation like that.

We explain the code generation algo in different levels of details, first without updating the book-keeping, afterwards keeping the books in sync, and finally, also keeping *liveness*

information into account. Still, even the most detailed version hide some details, for instance, if there is more than one location to choose from, which one is actually taken. The same will be the case for the `getreg` function later: some choice-points are left unresolved. It's not a big deal, it's not a question of correctness, it's more a question of how efficient the code (on average) is going to be.

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 $x := y$ **op** z

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

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

In the 3AIC lines, x, y, and z can also stand for temporaries. Resp. there's no difference anyhow, so it does not matter. Temporaries and variables are different, concerning their treatment for (local) liveness, but that information is available via the liveness information. For locations (in the 2AC level), we sometimes use l representing registers or memory addresses.

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 result 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, we don't mean to track 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 in that sense.

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

We start with a "textual" version first, followed by one using a little more programming/-math notation. One can see the general form of the generated code. One 3AIC line is translated into 2 lines of 2AC or, if lucky, in 1 line of 2AC

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 ($\Rightarrow$ step 4 also missing)
- details of *getreg* hidden.

The `let` $l_y \in \ldots$ notation is meant as pseudo-code notation for non-deterministic choice for, in this case, location `l_y` from some set of possible candidates. Note the invariant we mentioned: it's guaranteed, that y is stored *somewhere* (at least when still live), so it's guaranteed that there is at least one l_y to pick.

Also note (again), the order of the argument in 2AC. We save `y` at some location, in the slide called l. That one is mentioned as second argument in the 2AC. But the second argument, which at the same time is also the destination location may better be thought of as *first* input. For addition, it may not matter much, but for example `SUB b a` corresponds to `a - b` (with the result stored in `a`). Because of that and thhe way, the translation works also makes clear that we save `y` and not `z`.

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 $z := x$ **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];
if   l is a register
then 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)
```

Updating and exploit liveness info by recycling reg's

if y and/or z are currently

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

$\Rightarrow$ "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 wont' be consulted further in the block

In the pseudo-code we make use of some math-like notation. We write T_a and T_r for the 2 tables. They may be implemented as arrays or look-up structures. For updating we use notations like $T_a[x \mapsto l]$. This is meant to say: after the update, x is stored in l, the old information overwritten. Variables can be stored in different locations, but updating x in such an assignment *invalidates* all other locations, they become *out-of-date* or *stale*. The only place where x resides in l. By $T_a \backslash (\_ \mapsto l)$ we mean, we *remove* bindings, namely all that mention l.

Since there are situations, where one location can contain (the content of) more than variable, one may also have to suppert operations like $T_r[l \mapsto_\cup x]$, meaning that old information (here for l) is not overritten, but another "binding" is added: after the update, location l contains *also* (the value) of x, without forgetting the old values. This is not needed in the translation of our 3AIC instruction, but would occur when translating $x := y$ for instance, i.e., copying values.

We could also check whether x is live and do the corresponding wiping for x as well. In which case, the whole assignment is meaningless, and (as a consequence, also the liveness status of y and z could change in turn...).

As an invariant, a variable never resides in more than one register.

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 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	$L(2)$	$L(\perp)$	$L(2)$	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]			$\perp$	$\perp$	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 explicitely needed.
- e.g. variable a to be found "at a " (if not stale), 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)

  $\rightarrow$ **reuse** R_0
- Remark: info in [brackets]: "ephemeral"

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] Appel, A. W. (1998). *Modern Compiler Implementation in ML*. Cambridge University Press.

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

[5] Massalin, H. (1987). Superoptimizer — a look at the smallest program. In *Proceedings of the Second Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*, pages 122–126.

Index