Course Script

Static analysis and all that

IN5440 / autumn 2020

Martin Steffen

Contents

Chapter 2
Data flow analysis

What is it about?

Learning Targets of this Chapter

- various DFAs
- monotone frameworks
- operational semantics
- foundations
- special topics (SSA, context-sensitive analysis ...)

Contents

2.1 Introduction

In this part, we cover classical *data flow analysis*, first in a few special, specific analyses. Among other ones, we do one more time reaching definitions analysis. Besides that, we also cover some other well-known ones. Those analyses are based on common principles. Those principles then lead to the notion of *monotone framework*. All of this is done for

the simple while-language from the general introduction. We also have a look at some extensions. One is the treatment of *procedures*. Those will be *first-order* procedures, not higher-order procedures. Nonetheless, they are already complicating the data flow problem (and its computational complexity), leading to what is known as *context-sensitive* analysis. Another extension deals with dynamically allocated memory on *heaps* (if we have time). Analyses that deal with that particular language feature are known as *alias analysis*, *pointer analysis*, and *shape analysis*. Also we might cover SSA this time.

2.2 Intraprocedural analysis

As a start, we basically have a closer look at what we already discussed in the introductory warm-up: the very basics of data flow analysis, without complications like procedures, pointers, etc. The form of analysis is called *intraprocedural* analysis, i.e., an analysis focusing on the body of *one* procedure (or method, etc.). As already discussed, data flow analysis is done on top of so-called *control-flow graphs*. The control flow of each procedure can be abstractly represented by one such graph. Later, we will see how to "connect" different control-flow graphs to cover procedure calls and returns and how to extend the analyses for that.

Compared to the introduction, we dig a bit deeper: we show how a CFG can be computed for the while language (not that it's complicated), and we list different kinds of analyses, not just reaching definitions. Looking at those will show that they share common traits, and that prepares for what is known as *monotone frameworks*, a classic general framework for all kinds of data flow analyses.

2.2.1 Determining the control flow graph

This section shows how turn an abstract syntrax tree into a CFG. The starting point is the (labelled) abstract syntax from the introduction.

While language and control flow graph

- starting point: while language from the intro
- *labelled* syntax (unique labels)
- labels = *nodes* of the cfg
- initial and final labels
- edges of a cfg: given by function *flow*

Determining the edges of the control-flow graph Given an program in labelled (and abstract) syntax, the *control-flow graph* is easily calculated. The nodes we have already (in the form of the labels), the edges are given by a function *flow*. This function needs, as auxiliary functions, the functions *init* and *final*

The latter 2 functions are of the following type:

$$\mathit{init} : \mathbf{Stmt} \to \mathbf{Lab} \qquad \mathit{final} : \mathbf{Stmt} \to 2^{\mathbf{Lab}} \tag{2.1}$$

Their definition is straightforward, by induction on the labelled syntax:

	init	final
$[x := a]^l$	l	$\{l\}$
$[\mathsf{skip}]^l$	l	$\{l\}$
$S_1; S_2$	$\mathit{init}(S_1)$	$\mathit{final}(S_2)$
$\mathtt{if}\,[b]^l\,\mathtt{then}\,S_1\,\mathtt{else}\,S_2$	l	$\mathit{final}(S_1) \cup \mathit{final}(S_2)$
$\mathtt{while}\,[b]^l\,\mathtt{do}\,S$	l	$\{l\}$

(2.2)

The label $\mathit{init}(S)$ is the entry node to the graph of S. The language is simple and initial nodes are *unique*, but "exits" are not. Note that the concept of unique entry is not the same as that of "isolated" entry (mentioned already in the introduction). Isolated would mean: the entry is not the *target* of any edge. That's not the case, for instance of the program consists of outer while loop. In general, however, it may be preferable to have an isolated entry, as well, and one can arrange easily for that, adding one extra sentinel node at the beginning. For isolated exits, one can do the same at the end.

Using those, determining the edges, by a function

$$\mathit{flow} : \mathbf{Stmt} \to 2^{\mathbf{Lab}\times\mathbf{Lab}}$$

works as follows:

$$\begin{aligned}
\mathit{flow}([x := a]^l) &= \emptyset \\
\mathit{flow}([\mathsf{skip}]^l) &= \emptyset \\
\mathit{flow}(S_1; S_2) &= \mathit{flow}(S_1) \cup \mathit{flow}(S_2) \\
&\quad \cup \{(l, \mathit{init}(S_2)) \mid l \in \mathit{final}(S_1)\} \\
\mathit{flow}(\mathtt{if}\,[b]^l\,\mathtt{then}\,S_1\,\mathtt{else}\,S_2) &= \mathit{flow}(S_1) \cup \mathit{flow}(S_2) \\
&\quad \cup \{(l, \mathit{init}(S_1)), (l, \mathit{init}(S_2))\} \\
\mathit{flow}(\mathtt{while}\,[b]^l\,\mathtt{do}\,S) &= \mathit{flow}(S_1) \cup \{l, \mathit{init}(S)\} \\
&\quad \cup \{(l', l) \mid l' \in \mathit{final}(S)\}
\end{aligned} \tag{2.3}$$

Two further helpful functions In the following, we make use of two further (very easy) functions with the following types

$$\mathit{labels} : \mathbf{Stmt} \to 2^{\mathbf{Lab}} \quad \text{and} \quad \mathit{blocks} : \mathbf{Stmt} \to 2^{\mathbf{Stmt}}$$

They are defined straightforwardly as follows:

$$\begin{aligned} \mathit{blocks}([x := a]^l) &= [x := a]^l \\ \mathit{blocks}([\mathsf{skip}]^l) &= [\mathsf{skip}]^l \\ \mathit{blocks}(S_1; S_2) &= \mathit{blocks}(S_1) \cup \mathit{blocks}(S_2) \\ \mathit{blocks}(\mathtt{if}\,[b]^l\,\mathtt{then}\,S_1\,\mathtt{else}\,S_2) &= \{[b]^l\} \cup \mathit{blocks}(S_1) \cup \mathit{blocks}(S_2) \\ \mathit{blocks}(\mathtt{while}\,[b]^l\,\mathtt{do}\,S) &= \{[b]^l\} \cup \mathit{blocks}(S) \end{aligned} \tag{2.4}$$

$$\mathit{labels}(S) = \{l \mid [B]^l \in \mathit{blocks}(S)\} \tag{2.5}$$

All the definitions and concepts are really straightforward and should be intuitively clear almost without giving a definition at all. One point with those definitions, though is the following: the given definitions are all "constructive". They are given by structural induction over the labelled syntax. That means, they directly describe recursive procedures on the syntax trees. It's a leitmotif of the lecture: we are dealing with static analysis, which is a phase of a compiler, which means, all definitions and concepts need to be realized in the form of algorithms and data structures: there must be a concrete control-flow graph data structure and there must be a function that determines it.

Flow and reverse flow

$$\mathit{labels}(S) = \mathit{init}(S) \cup \{l \mid (l, l') \in \mathit{flow}(S)\} \cup \{l' \mid (l, l') \in \mathit{flow}(S)\}$$

- data flow analysis can be *forward* (like RD) or backward
- *flow*: for **forward** analyses
- for **backward** analyses: *reverse* flow flow^R, simply invert the edges

Program of interest

- S_*: program being analysed, top-level statement
- analogously $\mathbf{Lab}_*$, $\mathbf{Var}_*$, $\mathbf{Blocks}_*$
- *trivial* expression: a single variable or constant
- $\mathbf{AExp}_*$: non-trivial arithmetic sub-expr. of S_*, analogous for $\mathbf{AExp}(a)$ and $\mathbf{AExp}(b)$.
- useful restrictions
  - *isolated entries*: $(l, \mathit{init}(S_*)) \notin \mathit{flow}(S_*)$
  - *isolated exits* $\forall l_1 \in \mathit{final}(S_*).\ (l_1, l_2) \notin \mathit{flow}(S_*)$
  - *label consistency*

$$[B_1]^l, [B_2]^l \in \mathit{blocks}(S) \quad \text{then} \quad B_1 = B_2$$

"l labels *the* block B"

- even better: *unique* labelling

Often one makes restrictions or assumption over the form of the CFG. A typical one is the notions of entry and exit nodes being *isolated*. These are not really restrictions: a program consiting of a while-loop at the top-level would not enjoy the property of isolated entries, but then adding a skip statement in front of it would, and the parser or the some other part of the compiler could simply add such a statement in the AST, or an extra node in the CFG. Those isolated entries and exits are not crucial, just that the subsequent analysis might get a tiny bit clearer (avoiding taking care of some "special corner cases"). One often finds even more restrictions, namely *unique* entry and exits. As said, in our language, there is anyway unique entries, insofar it's not a additional restriction.

Concerning *label consistency*: That's kind of a weird assumption. Basically: if one labels an abstract syntax tree (respectively builds a control-flow graph), no one would even think of lumping together different, unrelated statements into the same node (the issue of basic blocks aside, we are not talking about that here). The natural way to go for it is: each individual block gets a new, fresh label. That would lead to *unique* labels, no label is reused, which is stronger (and much more natural) than the assumption here.

So why bother about label consistency? Well, actually we don't bother much. It's more like for the "theoretical" account later. Later, we will look at the *operational semantics*. We describe how the program evolves and that evolution changes the "abstract syntax"which represent the *run-time* evolution of the program; describing the behavior by referring to the abstract syntax and how it evolves under execution is a common way to define a language's semantics. It's seldom that a progamming language is actually *implemented* that way, it would correspond to a fully interpreted language, which, as said, is not very common.

At any rate: we show a simple operational semantics working on the abstract syntax of the while language. That representation will actually lead to duplication of parts of the program. More specifically, a loop body will be duplicate, when "unrolling" the loop. At that point, we have representation with a label occuring twice. In order to establish correctness for analyses is always a preservation argument: A core for the argument, that in an arbitrary number of steps, the program does not run into a situation not covered by the static analysis is: "doing one step preserves some relevant property". Doing this inductive argumentation means, we encounter at run-time CFGs with non-unique labelling, even if the original program, at compile time, was labelled uniquely, of course. To cover that, one uses the laxer condition of being consistently labelled (which is preserved by steps of the semantics) Actually, the semantics would preserve a better property, it seems to me. Not only is the labelling "consistent" in the sense defined here. But also the edges and neighbors of a node remains comparable. But the book does not point that out.

2.2.2 Available expressions

This is the first of a few classical data flow analyses we cover (like reaching definitions, as well). The analysis can be used for a so-called *common subexpression elimination*. CSE is a program *transformation* or *optimization* which makes use of the available expression analysis. The idea is easy: if one finds out that an expression is computed twice, it may pay off to store it the first time it's computed, and in the second occurence, look it up again.

Of course, that not just a syntactical problem, i.e., it's not enough to find syntactical occurrences of the same expression. In an imperative language and for expressions containing variables, the content of variables mentioned in such expression may or may not have changed for different occurences of the same expression, and that has to be figured out via a specific *data flow analysis*, namely "available expressions" analysis.

Avoid recomputation: Available expressions

$$[x := a + b]^0; [y := a * b]^1; \quad \texttt{while} \quad [y > a + b]^2 \\ \texttt{do} \quad ([a := a + 1]^3; [x := a + b]^4)$$

- usage: avoid *re-computation* of expression

Goal For each program point: which expressions **must** have already been computed (and not later modified), on all paths to the program point.

One important aspect in the (informal) goal of the analysis is the use of the word "must". That's different from what the situation for reaching definitions. There, it's about whether a "definition" *may* reach a point in questions. It's also worthwhile to reflect about "approximation". As always, exact information is not possible, what's why we content ourselves with "must" information (or "may" in the dual case). In the case here (and related to it): if we have some safe set of available expressions, then a *smaller* set it safe, too. Again, for the may-setting for reaching definitions, *enlarging* sets was safe. The situation here is therefore *dual*.

What obviously is also different is the nature or type of the *information of interest*. Here, it's sets of expressions, in the reaching definitions it is sets containing pairs of locations and variables.

Available expressions: general

- given as flow *equations* (not $\subseteq$-constraints, but not really crucial, as we know already)
- uniform representation of *effect of basic blocks* (= *intra-block flow*)

2 ingredients of intra-block flow

- *kill:* flow information "eliminated" passing through the basic blocks
- *generate:* flow information "generated new" passing through the basic blocks

- later analyses: presented similarly
- different analyses $\Rightarrow$ different kind of flow information + different kill- and generate-functions

In the introduction, the reaching definition analysis was done without explicitly mentioning the notions of "kill" and "generate", but they where there *implicitly* anyway (for the intra-block equations). Now, we formulate basically all analyses using kill and generate functions, so see the commonalities and differences. Not really all data flow analyses (in the monotone framework setting) can actually be phrased that way, but very many, including many important ones.

Available expressions: types

- interested in *sets of expressions*: $2^{\mathbf{AExp}_*}$
- generation and killing:

$$kill_{\mathsf{AE}}, gen_{\mathsf{AE}} : \mathbf{Blocks}_* \to 2^{\mathbf{AExp}_*}$$

- analysis: pair of functions

$$\mathsf{AE}_{entry}, \mathsf{AE}_{exit} : \mathbf{Lab}_* \to 2^{\mathbf{AExp}_*}$$

$\mathbf{AExp}_*$ can be taken as *all* arithmetic expressions occuring in the program, including all their subsexpressions. To be hyper-precise, one may refine it in that *trivial* (sub-)expressions don't count. Trivial expressions are constants and single variables. Those trivial expressions are uninteresting from the perspective of available expressions and therefore are left out. They are likewise left out for the very busy expression analysis which will be discussed soon.

Intra-block flow specification: Kill and generate

$$\begin{array}{rcl}
kill_{\mathsf{AE}}([x := a]^l) & = & \{a' \in \mathbf{AExp}_* \mid x \in fv(a')\} \\
kill_{\mathsf{AE}}([\mathsf{skip}]^l) & = & \emptyset \\
kill_{\mathsf{AE}}([b]^l) & = & \emptyset \\
\\
gen_{\mathsf{AE}}([x := a]^l) & = & \{a' \in \mathbf{AExp}(a) \mid x \notin fv(a')\} \\
gen_{\mathsf{AE}}([\mathsf{skip}]^l) & = & \emptyset \\
gen_{\mathsf{AE}}([b]^l) & = & \mathbf{AExp}(b)
\end{array}$$

The interesting case is, of course, the one for assignments (for generation, also the boolean equations are similar). An assignment kills all expressions, which contain the variable assigned to, and generates all (non-trivial) sub-expressions of the expression on the right-hand side of the assignment.

For *generation,* we have, however, to be *careful*: those sub-expressions of a which contain the variable x are of course not generated (because they are no longer "valid" after the assignment); note (on the next slide): the flow in a block is forward, and the flow at the exits depends on the in-flow *in the following order*:

1. *first kill*, and
2. *then generate*.

Because of this order, we cannot generate sub-expressions which contain x. The data flow analysis, at least those which are formulated with the help of kill and generate function, use them in that order. One might as well use killing and generating in the *opposite* order, but obviously, in that case, the exact definition of the kill and generate functions needs to take that into account and would have to be adapted to reflect that.

Flow equations: $\mathsf{AE}^{=}$

split into

nodes: **intra**-block equations, using *kill* and *generate*

edges: **inter**-block equations, using *flow*

Flow equations for AE

$$\mathsf{AE}_{entry}(l) = \begin{cases} \emptyset & l = init(S_*) \\ \bigcap\{\mathsf{AE}_{exit}(l') \mid (l', l) \in flow(S_*)\} & \text{otherwise} \end{cases}$$

$$\mathsf{AE}_{exit}(l) = \mathsf{AE}_{entry}(l) \setminus kill_{\mathsf{AE}}(B^l) \cup gen_{\mathsf{AE}}(B^l)$$

where $B^l \in blocks(S_*)$

- note the *"order"* of kill and generate

Apart from the fact that before we did not make use of some *explicit* kill and generate functions, the flow equations here are pretty similar to the ones for reaching definitions. One conceptual difference is the replacement of $\bigcup$ (may) for reaching definitions by $\bigcap$ (must).

Note that the definition of the flow equations assume *isolated entries,* which can be seen at the equation for $\mathsf{AE}_{entry}(l)$, in the case where l is the initial label (otherwise it would be a bit more complex). Note also: for AE_{entry}, we must make the case distinction of initial nodes (no incoming edges) and others, otherwise: the empty intersection would be something like the "full set" of expressions.

As subtle and perhaps not too relevant remark in that condition: that the empty intersection corresponds to the "full set" is by definition (of ultimately dealing with lattices). That sounds strange, but it's ok due to the following observation: the *initial* node is the *only* node in the control flow graph which —being isolated— has no incoming edge. The one and only node without incoming edge is of course $init(S_*)$ (if we assume isolated entries). Having an isolated entry is *not* guaranteed by the syntax, which means, we have to additionally assume it resp. ensure it otehrwise.

As mentioned: be aware of the *order* of kill and generate in the equation for the exit: first, the killed ones are removed, then the generated ones are added. Because of that order, one must make sure, that no expressions are generated that contain the assigned variable.

Available expressions

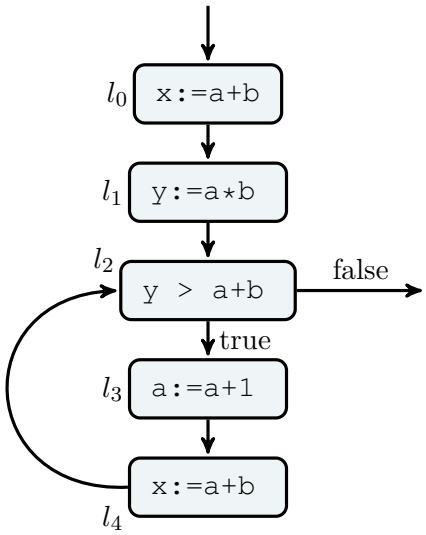
- *forward* analysis (as RD)
- interest in *largest* solution (unlike RD)

⇒ **must** analysis (as opposed to *may*)

- expression is available: if *no path kills it*
- remember: informal description of AE: expression available on *all paths* (i.e., not killed on any)

Example AE

$$[x := a + b]^0; [y := a * b]^1; \quad \texttt{while} \quad [y > a + b]^2 \\ \texttt{do} \quad ([a := a + 1]^3; [x := a + b]^4)$$



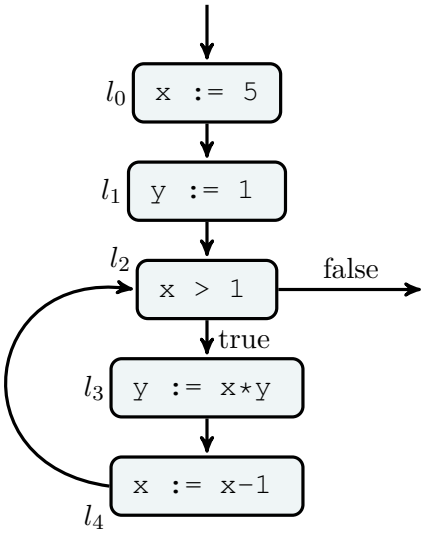
Worthwhile is (for instance) the entry of node / block l_2. At that point, expression $a + b$ is available. That's despite the fact that a is changed inside the body of the loop.

2.2.3 Reaching definitions

Reaching definitions

- remember the intro
- here: the *same* analysis, but based on the new definitions: kill, generate, flow ...

$$[x := 5]^0; [y := 1]^1; \texttt{while}[x > 1]^2 \, \texttt{do}([y := x * y]^3; [x := x - 1]^4)$$



Reaching definitions: types

- interest in *sets of tuples of var's and program points i.e., labels*:

$$2^{\mathbf{Var}_* \times \mathbf{Lab}_*^?} \quad \text{where} \quad \mathbf{Lab}_*^? = \mathbf{Lab}_* + \{?\}$$

- generation and killing:

$$kill_{\mathsf{RD}}, gen_{\mathsf{RD}} : \mathbf{Blocks}_* \to 2^{\mathbf{Var}_* \times \mathbf{Lab}_*^?}$$

- analysis: pair of mappings

$$\mathsf{RD}_{entry}, \mathsf{RD}_{exit} : \mathbf{Lab}_* \to 2^{\mathbf{Var}_* \times \mathbf{Lab}_*^?}$$

The information is the same as in the introduction (except here, we are explicit that it should be not just sets of variables, but that only the sets of variables of the program are of interest, which here is denoted as $\mathbf{Var}_*$. Similarly for $\mathbf{Lab}_*$). But that's just a bit more precise (perhaps overly so).

As far as the mappings or functions RD_{entry} and RD_{exit} are concerned: In a practical implementation, one might use *arrays* for that. If the implementation identifies nodes by "numbers", one can have an integer-indexed standard array, which typically is a fast way of prepresenting that information.

Reaching defs: kill and generate

$$\begin{array}{rcl} kill_{\mathsf{RD}}([x := a]^l) & = & \{(x, ?)\} \cup \\ & & \bigcup\{(x, l') \mid B^{l'} \text{ is assgm. to } x \text{ in } S_*\} \\ kill_{\mathsf{RD}}([\mathsf{skip}]^l) & = & \emptyset \\ kill_{\mathsf{RD}}([b]^l) & = & \emptyset \end{array}$$

$$\begin{array}{rcl} gen_{\mathsf{RD}}([x := a]^l) & = & \{(x, l)\} \\ gen_{\mathsf{RD}}([\mathsf{skip}]^l) & = & \emptyset \\ gen_{\mathsf{RD}}([b]^l) & = & \emptyset \end{array}$$

Similar to the AE analysis: the interesting case is, of course, the one for assignments. The generation and killing is indeed also quite similar to before. It is the assignment *to* x which affects the flow, of course. Now, it eliminates all pairs of similar assignments, in the AE-analysis, it invalidates all expressions, which mention x. For the generation, the AE has been a bit more complex than the analysis here: here, just the current pair of label and the variable is added (actually, for unique labelling, even the label alone would suffice). For AE, the relevant generated information is not drawn from x in an assignment $x := a$, but from a (its non-trivial sub-expressions).

Flow equations: $\mathsf{RD}^=$

split into

- *intra*-block equations, using *kill* and *generate*
- *inter*-block equations, using *flow*

Flow equations for RD

$$\mathsf{RD}_{entry}(l) = \begin{cases} \{(x, ?) \mid x \in fv(S_*)\} & l = init(S_*) \\ \bigcup\{\mathsf{RD}_{exit}(l') \mid (l', l) \in flow(S_*)\} & \text{otherwise} \end{cases}$$

$$\mathsf{RD}_{exit}(l) = \mathsf{RD}_{entry}(l) \setminus kill_{\mathsf{RD}}(B^l) \cup gen_{\mathsf{RD}}(B^l)$$

where $B^l \in blocks(S_*)$

- same order of kill/generate

2.2.4 Very busy expressions

This is a another example of a classical data flow analysis. As for AE, one is interested in *expressions* (not assignments). This time it's about if an expression is "needed" in the future. Compared to AE, the perspective has changed. It's not about if an expression that has been evaluated *in the past* is still available as some given point. It the opposite: will an expression be of use *in the future*.

This change of perspective also means that VB is an example of a *backward* analysis. The natural way of analysing very busy expressions is: at the place where an expression is actually used, immediately in front of that place it's definitely very busy. And then from there, let the information flow *backwards*: in the previous location, it's also very busy (unless relevant variables are change which "destroy" the the "busy-ness"), then the continue the argument.

Being very busy also means an expression is used *on all future paths*, which makes it a *must* analysis.

One can make use of very busy information as follows: if an expression is very busy, it may pay off to calculate it already now, i.e., it can be used for a program transformation, that moves the calculation of expression in an "eager" fashion as early as posssible. Transformations like this are known as expression "hoisting".

This is may lead to *shorter code* of an expression, which is being calculated in two branches of a conditional, for example, can be move earlier *outside* the branching construct. Note that while that may be reduce the *code size* but not really the run-time for executing the code.

Transformations like the one mentioned are often done (also) on low level code (like machine-code or low-level intermediate reprentations which are already close to machine code, but still machine-independent). Executing one command ("one line of machine code") costs clock-cycle(s) already, since the command itself needs to be loaded to the processor; on top of that comes costs for loading the operands. So, shortening *straight-line code* may well improve the execution time. However, hoisting an expression out from both branches of a conditional and position it in front of the branch shorten the size of the code without making it faster.

Very busy expressions

$$\begin{array}{ll} \texttt{if} & [a > b]^1 \\ \texttt{then} & [x := b - a]^2; [y := a - b]^3 \\ \texttt{else} & [a := b - a]^4; [x := a - b]^5 \end{array}$$

Definition 2.2.1 (Very busy expression)**.** An expression is *very busy* at the exit of a label, if for "all" paths from that label, the expression is used before any of its variables are "redefined" (= overwritten).

- usage: expression "hoisting"

Goal For each program point, which expressions are very busy at the *exit* of that point.

Note that I wrote "all" paths in quotation marks. It's a subtle point that we look at later in an example. In princple one may think in the definition to refer to all paths, it's only that for *infinitely long* paths, one has to be careful. In a way it's counter-intutive (as we perhaps see in the example) and it's a more theoretical consequence of the way the fixpoints are defined. Practically one may ignore it perhaps, infinite paths are not too

relevant in practice, But infinite loops do exits, even if sometimes they are indicating a problem (the program "hangs").

Note that the definition and the goal are formulated in a subtle way. It's about information *at the exits* of the basic blocks, not the entry. In principle, and as far as the equations or constraints are concerned, the formulation will mention VB_{entry} and VB_{entry} (see later) in the same way as the equations for reaching definitions, for example, mentioned RD_{entry} and RD_{exit}. So it seems to be, one calculates exits *and* entries.

Nothing wrong with that, but looking carefully to the pseudo-code formulation of the algorithms later, a refinement of the rather sketchy random iteration of the introduction, we will see that what is given back is *indeed* only the very busy information at the *exits* of the basic blocks. The reason why it's the exits (and not the entries) is because the very busy expression analysis works *backwards*, for forward analyses it's the corresponding information at the *entries* of the blocks.

Why is that? Basically (in the case of the backward analysis), having the solution at the exits allows to reconstruct immediately the solution values at the entries per block (via the kill and generate function attached to the block, something which will be called the *transfer function* of the block). The pseudo-code will indeed work with "arrays" VB_{exit} and VB_{entry}, it's only that what the algo will give back VB_{exit}, only. One can, however, implement basically the same algorithm leaving out VB_{entry}, storing *only* VB_{exit} throughout the run.

Anyway, the reason why the goal is formulated like that is as (for backward analysis) the exit information is the crucial one, if one has that, the entry information follows by applying the transfer function (a combination of kill and generate) to the exit communcation, so there is not need to store it seperately in RD_{entry} if one wants to do without a second array during the run.

Very busy expressions: types

- interested in: *sets of expressions*: $2^{\mathbf{AExp}_*}$
- generation and killing:

$$kill_{\mathsf{VB}}, gen_{\mathsf{VB}} : \mathbf{Blocks}_* \to 2^{\mathbf{AExp}_*}$$

- analysis: pair of mappings

$$\mathsf{VB}_{entry}, \mathsf{VB}_{exit} : \mathbf{Lab}_* \to 2^{\mathbf{AExp}_*}$$

Very busy expr.: kill and generate

core of the intra-block flow specification

$$
\begin{array}{rcl}
kill_{\mathsf{VB}}([x := a]^l) & = & \{a' \in \mathbf{AExp}_* \mid x \in fv(a')\} \\
kill_{\mathsf{VB}}([\mathsf{skip}]^l) & = & \emptyset \\
kill_{\mathsf{VB}}([b]^l) & = & \emptyset
\end{array}
$$

$$
\begin{array}{rcl}
gen_{\mathsf{VB}}([x := a]^l) & = & \mathbf{AExp}(a) \\
gen_{\mathsf{VB}}([\mathsf{skip}]^l) & = & \emptyset \\
gen_{\mathsf{VB}}([b]^l) & = & \mathbf{AExp}(b)
\end{array}
$$

A comparison with the kill and generate functions for AE might be interesting. First of all, in both cases, the functions have the *same types*, i.e., operate on the same domains. Of course, one difference is, that now the flow is backwards. For the blocks without side effects, this does not matter, i.e., the generate function is identical in both cases (the kill-function as well, of course). For the assignment, there are obviously differences. Let's first look at the kill-case. Literally, the two definitions *coincide,* but they have a different intuition (backward vs. forward). *Here* for VB we ask, because we are thinking backwards, which expressions are very busy at the *entry* of that block [misleading]. Of course, also the killing works backwards: whatever was very busy at the exit of the block, all expressions that contain x are modified and thus are *not* very busy at the entry of the block (one could say, as there is no branching withing one block, they are not even busy at all), and thus the kill-function removes those. The reasoning for the AE case is similar, only working forward.

For the generation function, as we are working backwards, the assignment generates a as very busy at the entry of the block. Unlike for AE, the free occurrence of x does not play a role. That's because the order of the applications is still do *first* kill and *then* generate, also when working backwards.

Flow equations.: $\mathsf{VB}^=$

split into

- intra-block equations, using kill/generate
- inter-block equations, using *flow*

however: everything works backwards now

Flow equations: VB

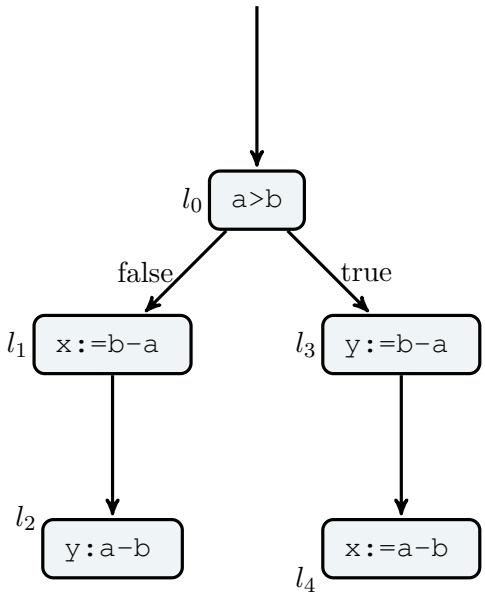
$$
\begin{array}{rcl}
\mathsf{VB}_{exit}(l) & = & \begin{cases} \emptyset & l \in final(S_*) \\ \bigcap\{\mathsf{VB}_{entry}(l') \mid (l', l) \in flow^R(S_*)\} & \text{otherwise} \end{cases} \\
\mathsf{VB}_{entry}(l) & = & \mathsf{VB}_{exit}(l) \setminus kill_{\mathsf{VB}}(B^l) \cup gen_{\mathsf{VB}}(B^l)
\end{array}
$$

where $B^l \in blocks(S_*)$

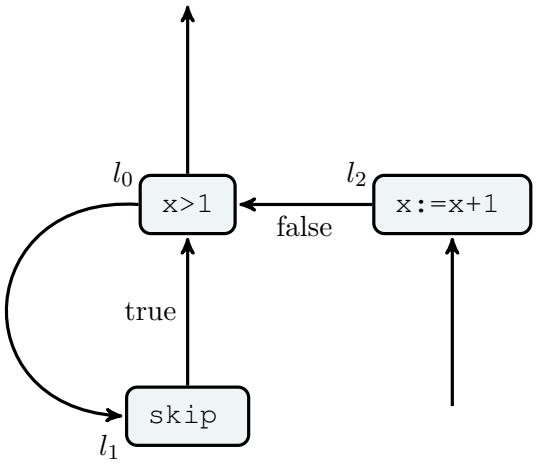
Note: Doing a *backward* analysis, the roles of entries and exits are now reversed. The kill and generate functions now calculate the *entry* as function of the exit point. Analogously,

the inter-block flow equations (of the graph) calculate the exit of a block as function of the entries of others.

Example



Example: infinite loop



Since the very busy expression analysis works *backwards*, the illustration show the *reversed* control flow graph.

Besides that: The looping example is quite instructive. It illustrates a *subtle(!)* point which might not immediately clear from the informal formulation of what "very busy" means. The example is a bit artificial, and the only expression occuring at all is $x + 1$ in node l_2. Now the question is:

Is expression $x + 1$ very busy at the beginning of the program or not?

Assuming that $x > 1$, there is obviously an infinite loop and the assignment of l_2 will never be executed. Consequently, the expression will not be needed anyhow. That's of course naive in that standard data flow analysis does not try to figure out if a left-branch or a right branch is taken; in the case of the example, whether the loop body is ignored or not.

On the other hand: it seems that the analysis could make the assumption that there *is* actually a path on which the $x + 1$ is *never, ever* evaluated. That seems to indicate that one should intuitively consider the expression $x + 1$ *not very busy*. If we don't know how often the loop body is executed (if at all), and since we cannot exclude that the body is taken infinitely (as in this case), it seems plausible to say, there's a chance that $x + 1$ may not executed and therefor count it as not very busy.

Plausible as that argument is: **it's wrong and $x + 1$ is indeed very busy!** Informally, the reason being that in a way, "infinite paths don't count" (like the one cycling infinitely many times through the skip-body). Formally, the fact comes from fact that we are interested in *the largest safe solution* and the way the *largest* fixpoint it defined (and then they way that the fixpoint iteration, like the chaotic iteration calculates is).

Later, the same example will be used for live variable analysis. Like the one here, it's a *backward* analysis. Different from very busy expressions, it's a *may* analysis (and consequently it's about the smallest possible safe solution). Being a may analysis will make use of $\cup$. Anyway, the variable x will be counted as *live* at the beginning of the program, as there is a possibility that x is used (in l_2) and that possiblity does not involve making an argument about infinite paths. Unlike the situation of the very busy expressions, this seems intuitively plausible.

2.2.5 Live variable analysis

This analysis focuses on *variables* again (not on expressions). If we use "dead" for being not live, a variable is dead intuitively if its value is definitely ("must") not used in the future. This is important information, in that the memory bound to the variable can be "deallocated".

That is in particular done at lower levels of the compiler. There, the compiler attempts to generate code which make "optimal" use of avaible registers (except that real optimality is out of reach, so it's more like the compiler typically makes a decent effort in making good use of registers, at least on average). A register currently containing a dead variable can be recycled (to be very precise: the register can be recycled if it contains *only* dead variables as in some cases, a register can hold the content of more than one variable ...). So a variable is *live* if there is a *potential* use of it in the *future* ("may").

Referring to the *future* use of variables entails that the question for liveness of variables leads to a *backward* analysis, similar to the situation of very busy expressions, which was also backwards.

For the participants of the compiler construction lecture (INF5110). That lecture covered live variable analysis, as well, namely in a *local* variant for elementary blocks of *straight-line code*. Additionally, a "global" live analysis was sketched, which correspond to the one here.

When can var's be "recycled": Live variable analysis

$$[x := 2]^0; [y := 4]^1; [x := 1]^2;$$
$$(\texttt{if}[y > x]^3 \texttt{ then } [z := y]^4 \texttt{ else } [z := y * y]^5); [x := z]^6$$

Live variable A variable is **live** (at the exit of a label) if there *exists* a path from the mentioned exit to the *use* of that variable which does not assign to the variable (i.e., redefines its value)

Goal therefore for each program point: which variables may be live at the exit of that point.

- usage: *register allocation*

Live variables are about: when is a variable still "needed". If not needed, one can *free* the space. In some sense and very generally, the question resembles reaching definition, at least, in that it's again about "variables" not expressions. In both cases we like to connect the assignment (also called *definition* of a variable to its *use*). The perspective here is different, though. For RD, the question is: given an assignment, what locations can it reach. For LV it's the opposite: given a location, which assignments can have reached me. This switch in perspective is the difference between *forward* and *backward* analysis.

Unlike in the informal definition of very busy expressions, here the word is *may*. With the may-word, the intuition is, that making the solution *larger* is ok, therefore we are interested in the smallest solution. This is consistent with the fact of making use of live variable analysis for recycling variables. If we estimate too many variables as live, we cannot reuse their memory, which is safe, we only may lose efficiency. Making the opposite approximation, marking an actually live variable erronously as non-live, that can cause errors and is therefore unsafe.

Note again the formulation of the goal: "backward" corresponds to "we are interested at the exit".

Live variables: types

- interested in sets of variables $2^{\mathbf{Var}_*}$
- generation and killing:

$$kill_{\mathsf{LV}}, gen_{\mathsf{LV}} : \mathbf{Blocks}_* \to 2^{\mathbf{Var}_*}$$

- analysis: pair of functions

$$\mathsf{LV}_{entry}, \mathsf{LV}_{exit} : \mathbf{Lab}_* \to 2^{\mathbf{Var}_*}$$

Live variables: kill and generate

$$\begin{array}{rcl} kill_{\mathsf{AE}}([x := a]^l) & = & \{x\} \\ kill_{\mathsf{LV}}([\mathsf{skip}]^l) & = & \emptyset \\ kill_{\mathsf{LV}}([b]^l) & = & \emptyset \end{array}$$

$$\begin{array}{rcl} gen_{\mathsf{LV}}([x := a]^l) & = & fv(a) \\ gen_{\mathsf{LV}}([\mathsf{skip}]^l) & = & \emptyset \\ gen_{\mathsf{LV}}([b]^l) & = & fv(b) \end{array}$$

We need to remember that the calculation is *backwards*. As for kill: in the only interesting case of assignment, the question is: given the live variables at the end, which ones are live at the entry. Certainly, x is no longer live, as it is not used (forward) before overwritten.

That also explains the generation: all free variables in a, resp. in b are live at the beginning of a block that mentions the resp. expression. In particular, the x does not play a role in the generation function for assignments, as we are working backwards.

Flow equations $\mathsf{LV}^=$

split into

- *intra*-block equations, using kill/generate
- inter-block equations, using flow

however: everything works backwards now

Flow equations LV

$$\mathsf{LV}_{exit}(l) = \begin{cases} \emptyset & l \in final(S_*) \\ \bigcup\{\mathsf{LV}_{entry}(l') \mid (l', l) \in flow^R(S_*)\} & \text{otherwise} \end{cases}$$

$$\mathsf{LV}_{entry}(l) = \mathsf{LV}_{exit}(l) \setminus kill_{\mathsf{LV}}(B^l) \cup gen_{\mathsf{LV}}(B^l)$$

where $B^l \in blocks(S_*)$

The example, why one is this time interested in the smallest solution is the same program as for the VB: a simple recursive equation (induced by a trivial while-loop). This time the loop contains a $\cup$. We can make the solution as large as possible (but not as small as possible, the empty set is not a solution). However, the smallest set is the most informative one. That can be guessed from the words "may be live" already. Also the intended use of freeing/re-using "non-live" variables makes clear that it's "larger is less precise".

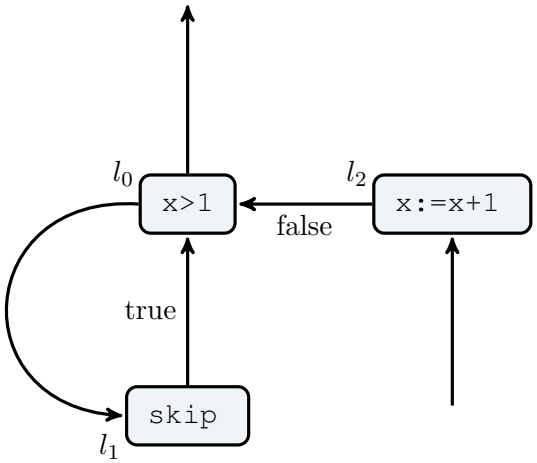
Example

$$(\texttt{while}\ [x > 1]^{l_0}\ \texttt{do}\ [\mathsf{skip}]^{l_1}); [x := x + 1]^{l_2}$$

As one can see in the flow equations, especially the case dealing with the final nodes, variables are considered *dead* at the end. One may also have the intuition, the variables (or some) are returned to somewhere, in which case they are still needed "after" the final node (for being returned) and hence they are marked *live*. In the latter intuition, it would probably be clearer to have an explicit return statement (after which the variables are *really dead*).

For the participants of the compiler construction course (INF5110). The course presented a *local* live variable analysis, which concentrated on *straight line code*. The code was so-called three-address code and had two types of variables: normal ones and so-called "temporaries" (temporary variables). The standard variables were assumed *live* at the end of the straight-line code. The reason being that the SLC is code contained *inside one basic block*. Since the end if the block is not necessarily the end of the program, the analysis had to assume conservatively that chances are, that variables may by used by any potentially following block, and thus they variables were assumed live. Temporaries, on the other hand, were treated as *dead* at the end, which was justfied by the fact that the code *never* used temporaries from a previous block. That, of course, depended on the knowledge how this code was actually generated. The general point is that of course the formulation of the live variable analysis (or others) must go hand in hand with what is actually going on, i.e., the semantics of the language and the assumptions about how the program is used ("will the content of the variables be returned to a caller after the program code or not", "might there by a block after the code being analyzed or not, and if so, will it make use of temporaries resp. variables or not").

Looping example



2.3 Theoretical properties and semantics

2.3.1 Semantics

So far we have formulated a number of analyses (using flow equations or constraints). We also stressed the importance that the analyses are *safe* (or correct or sound), meaning that the information given back from the analysis says something "true" about the program, more precisely about the program's behavior. So far, that is an empty claim as we have not fixed what the behavior actually is. Doing so may look superfluous, in particular as the while language we are currently dealing with is so simple that its semantics seems pretty "obvious" for most. That, however, may no longer be the case when dealing with more advanced or novel features or non-standard syntax etc. Being clear about the semantics also pays off when implementing a language, after all, the ultimately running program is expected to implement exactly the specified semantics, down to the actual machine code on a particular platform. Leaving the semantics up-to the implementation or the platform is not considered a very dignified engineering approach ("the semantics of a program is what happens if you run it, you'll see.").

In this section we will precisely define the semantics of the while-language. The semantics is defined on a rather high-level, on the level of the abstract syntax, and the task of the compiler would be to preserve exactly the semantics through all its phases. The task of the static analyses is to *soundly approximate* this semantics. If optimizations and transformations are done, for example based on some static analyses, it's the task of the optmization to *preserve* the semantics, as well. So the semantics is the *yardstick* which all further actions of the compiler are measured against.

The *semantics* of a programming language can be specified in different styles or flavors. We make use of *operational semantics*, a style of semantics which describe *steps* a program does. In particular, we make use of *strucutural operational semantics* (SOS), which refers to the fact that the steps are described making inductive use of the *structure* of the program (i.e., its abstract syntax).

That's arguably a straightforward way for fixing the semantics. It basically descibes the semantics as steps tranforming an abstract syntax tree step by step and can be seen as an formal description of an *interpreter*.

There is, however, also not *one* unique way how such an operational semantics is defined, also there different flavors and styles exists. Here, we are doing what is know a *small-steps* semantics. Perhaps later, for the more complex functional languages, the lecture may cover some variations.

Relating programs with analyses

- analyses
  - intended as (static) *abstraction* or overapprox. of real program behavior
  - so far: *without real connection* to programs
- soundness of the analysis: **safe** analysis
- but: *behavior* or *semantics* of programs not yet defined

- here: "easiest" semantics: *operational*
- more precisely: *small-step SOS* (structural operational semantics)

States, configs, and transitions

fixing some data types

- *state* $\sigma : \mathbf{State} = \mathbf{Var} \to \mathbf{Z}$
- *configuration:* pair of *statement* $\times$ *state* or (terminal) just a *state*

Transitions

$$\langle S, \sigma \rangle \to \acute{\sigma} \quad \text{or} \quad \langle S, \sigma \rangle \to \langle \acute{S}, \acute{\sigma} \rangle$$

Semantics of expressions

$$[\,\_\,]^{\mathcal{A}}_{\_} : \mathbf{AExp} \to (\mathbf{State} \to \mathbb{Z})$$
$$[\,\_\,]^{\mathcal{B}}_{\_} : \mathbf{BExp} \to (\mathbf{State} \to \mathbf{B})$$

simplifying assumption: no errors

$$\begin{array}{rcl} [x]^{\mathcal{A}}_{\sigma} & = & \sigma(x) \\ [n]^{\mathcal{A}}_{\sigma} & = & \mathcal{N}(n) \\ [a_1 \,\mathsf{op}_a\, a_2]^{\mathcal{A}}_{\sigma} & = & [a_1]^{\mathcal{A}}_{\sigma} \;\mathbf{op}_a\; [a_2]^{\mathcal{A}}_{\sigma} \end{array}$$

$$\begin{array}{rcl} [\mathsf{not}\ b]^{\mathcal{B}}_{\sigma} & = & \neg [b]^{\mathcal{B}}_{\sigma} \\ [b_1 \,\mathsf{op}_b\, b_2]^{\mathcal{B}}_{\sigma} & = & [b_1]^{\mathcal{B}}_{\sigma} \;\mathbf{op}_b\; [b_2]^{\mathcal{B}}_{\sigma} \\ [a_1 \,\mathsf{op}_r\, a_2]^{\mathcal{B}}_{\sigma} & = & [a_1]^{\mathcal{A}}_{\sigma} \;\mathbf{op}_r\; [a_2]^{\mathcal{A}}_{\sigma} \end{array}$$

clearly:

$$\forall x \in fv(a).\ \sigma_1(x) = \sigma_2(x) \text{ then } [a]^{\mathcal{A}}_{\sigma_1} = [a]^{\mathcal{A}}_{\sigma_2}$$

In the introductory remarks, we mentioned that we will do some specific form of semantics, namely an *operational semantics*. That's not 100% true. For dealing with the control-flow structure of the while language, we will indeed formulate operational rules to describe the transitions. We must, however, also give meaning to expressions a and b. To do that operationally would be possible, but perhaps an overkill. More straightforward is an inductive definition in the way given. That style corresponds more to a *denotational semantics*. It should be noted that expressions in the while language are *side-effect free*. So things like `x := 5 * y++` which can be found in for example C-like languages, where the right-hand side of the assignment is at the same time an expression as well as having a side effect, are not welcome here. Without such side effects, the denotation-style semantics for expressions is just the easiest way of specifying their meaning. That way we can focus on the part that is more interesting for us, the steps or transitions of the operational semantics.

It would be possible to *also* specify the meaning of expressions in an operational, step-wise manner. In this way there would me more transitions explicitly mentioned in the semantics (maybe distinguising the transitions between configurations and "micro-transitions" evaluating the expressions).

For participants of the compiler construction course (INF5110): an operational semantics for the expressions showed up in some way in the lecture, when translating expression into *three address code*. Since there is no recursion and deeply nested expressions in three-address code (which is used in the definition of $[a]^{\mathcal{A}}_{\sigma}$), the expression has to be "expanded" into sequences of non-nested expression together with temporary variables ("temporaries") to hold intermediate results of subexpressions. That in a way corresponds to an explicit, step-by-step execution of a compound expression. It's not the same as an operational semantics, as it does not specify "transitions", but its "code generation". On the other hand: it's not far away as each single three-address-code instruction would correspond to one transition.

SOS

$$\langle [x := a]^l, \sigma\rangle \to \sigma[x \mapsto [a]^{\mathcal{A}}_{\sigma}] \quad \textsc{Ass} \qquad \langle [\mathsf{skip}]^l, \sigma\rangle \to \sigma \quad \textsc{skip}$$

$$\frac{\langle S_1, \sigma\rangle \to \langle \acute{S}_1, \acute{\sigma}\rangle}{\langle S_1; S_2, \sigma\rangle \to \langle \acute{S}_1; S_2, \acute{\sigma}\rangle}\ \textsc{Seq}_1 \qquad \frac{\langle S_1, \sigma\rangle \to \acute{\sigma}}{\langle S_1; S_2, \sigma\rangle \to \langle S_2, \acute{\sigma}\rangle}\ \textsc{Seq}_2$$

$$\frac{[b]^{\mathcal{B}}_{\sigma} = \top}{\langle \texttt{if } [b]^l \texttt{ then } S_1 \texttt{ else } S_2, \sigma\rangle \to \langle S_1, \sigma\rangle}\ \textsc{If}_1$$

$$\frac{[b]^{\mathcal{B}}_{\sigma} = \top}{\langle \texttt{while } [b]^l \texttt{ do } S, \sigma\rangle \to \langle S; \texttt{while} [b]^l \texttt{ do } S, \sigma\rangle}\ \textsc{While}_1$$

$$\frac{[b]^{\mathcal{B}}_{\sigma} = \bot}{\langle \texttt{while } [b]^l \texttt{ do } S, \sigma\rangle \to \sigma}\ \textsc{While}_2$$

Each construct is convered by one or more rules (the second rule for conditionals is left out). Even if there are constructs where there is more than one rule, the alternatives are mutually excclusive. For instance, in case of the conditional, either the rule for the true-case applies or the rule for the false-case (not shown). This "mutual-exclusiveness" means that the semantics is *deterministic*. Given a configuration, there is at most one sucessor state. That it to be expected from a conventional sequential language. Not only are the conditions are mutual exclusive, the rules and conditions are also "exhaustive": for each configuration, exactly one rule actually applies. That means, each configuration has *exactly* one successor, i.e., not only is the behavior deterministic, a program also does not get "stuck". The only situation, when there is no successor is when the final state has been reached, i.e., when the configuration consists only of a state (and no code). That's the way (in this formalization) end-states look like.

As a side remark about "getting stuck". It may sound strange, why should a program "get stuck" anyway? Real programs don't tend to get stuck, why should it's formal semantics do that? Indeed, there is a point to it, one could make the argument, when a processor executes machine code, then either there [... continue the argument ...]

Derivation sequences

- derivation sequence: "completed" execution:
  - finite sequence: $\langle S_1, \sigma_1\rangle, \ldots, \langle S_n, \sigma_n\rangle, \sigma_{n+1}$
  - infinite sequence: $\langle S_1, \sigma_1\rangle, \ldots, \langle S_i, \sigma_i\rangle, \ldots$
- note: labels do *not* influence the semantics
- CFG for the "rest" of the program only gets "smaller" when running:

Lemma 2.3.1 (Steps and CFG)**.**

1. *$\langle S, \sigma\rangle \to \sigma'$, then $final(S) = \{init(S)\}$*
2. *Assume $\langle S, \sigma\rangle \to \langle \acute{S}, \acute{\sigma}\rangle$, then*
   a) $final(S) \supseteq \{final(\acute{S})\}$
   b) $flow(S) \supseteq \{flow(\acute{S})\}$
   c) $blocks(S) \supseteq blocks(\acute{S})$; if S is label consistent, then so is $\acute{S}$

We don't do the proof here. The properties should actually pretty obvious (if we defined the semantics etc all properly, which we did). Steps decrease the set of labels, for instance. The set of labels may actually stay the same, that will be the case for unrolling a while loop.

Even if we don't do the actual proof, we can remark on how that proof is done. The proofs need to be based on the semantics, and it will involve a big case distinction (in case of skip, such and such, in case of assigmment, such and such etc.). Furthermore, some of the rules involve an inductive case: the step is defined on some construct, say $S_1; S_2$ involves mentioning a step for S_1 in the premise. S_1 is a substructure of $S_1; S_2$. Therefore one can do induction on the structure of the abstract syntax (i.e., the abstract syntax tree). That form of induction is called also proof by structural induction. One could alternatively do a proof by indiction on the (structure of the) derivation of the step $\langle S, \sigma\rangle \to \langle \acute{S}, \acute{\sigma}\rangle$. Note that the operational semantics is given by derivation rules, and the justification of a step is therefore a tree, a derivation tree, and one can do induction of that structure. Both approaches would basically lead to the same proof.

Note: induction over the structure of the syntax tree for the case of the while loop does not work (in the true-case): the configuration after unrolling the loop is not structurally smaller than before the step.

Correctness of live analysis

- LV as example
- given as *constraint system* (not as equational system)

LV constraint system

$$\mathsf{LV}_{exit}(l) \supseteq \begin{cases} \emptyset & l \in final(S_*) \\ \bigcup\{\mathsf{LV}_{entry}(l') \mid (l', l) \in flow^R(S_*)\} & \text{otherwise} \end{cases}$$

$$\mathsf{LV}_{entry}(l) \supseteq \mathsf{LV}_{exit}(l) \setminus kill_{\mathsf{LV}}(B^l) \cup gen_{\mathsf{LV}}(B^l)$$

$$live_{entry}, live_{exit} : \mathbf{Lab}_* \to 2^{\mathbf{Var}_*}$$

"*live* solves constraint system $\mathsf{LV}^{\subseteq}(S)$"

$$live \models \mathsf{LV}^{\subseteq}(S)$$

(analogously for equations $\mathsf{LV}^{=}(S)$)

Equational vs. constraint analysis

We mentioned it a couple of times. For data flow analyses, the formulation based on equational constraints and the one based on $\subseteq$ constraints are "essentially" the same. Essentially insofar that they give the same result as far as the smallest solution is concerned. The lemma makes the explicit. Note, the lemma is about a *may* analysis, namely live variable analysis. For must-analyses, a corresponding result would not mention the least solution, but the largest.

Lemma 2.3.2 (Equational and inclusion constraints)**.**

1. *If* $live \models \mathsf{LV}^{=}$*, then* $live \models \mathsf{LV}^{\subseteq}$
2. *The least solutions of* $live \models \mathsf{LV}^{=}$ *and* $live \models \mathsf{LV}^{\subseteq}$ *coincide.*

2.3.2 Intermezzo: Lattices

This section provides some foundations on which data flow analysis rests, in particular, one which the instances of the monotone framework rest. The mathematical structures are known as *lattices* and we will have some basic facts and definitions covered about those. Actually lattices and similar concepts are more widely "used" than in data flow analyses. "Used" in the sense that they provide conceptual foundations also for other fields, besides data flow analysis.

Anyway, the concept of lattice is about *orders* or orderings. An ordered structure is about comparing things, like in being larger resp. smaller, or similar. For us, the possible results of an analysis are indeed ordered, namely as being more or less precise, and we are aiming at precision (under the side condition of being safe or correct): from all safe approximations we are interested in the most precise one (which can be the smallest solution or the largest solution, depending on whether we are dealing with a may or a must analysis).

So, order relations allow to *compare* elements (i.e., order them) and they satisfy some properties; there are also different kinds of order relations (total order, partial order, and

more), and a lattice is a specific form of a partial order with some further requirements on top. Actually, there are different kinds of lattices, as well, distinguished by the extra conditions they satisfy. There is a whole zoo of order relations and lattice relations; the field is called order theory or lattice theory; we will stick to one straightforward one called *complete lattice*.

Let's start with order relations, sticking to the one we will work with, the notion of *partial* order.

Definition 2.3.3 (Partial order)**.** A *partial order* $(A, \sqsubseteq)$ is a set with a reflexive, transitive, and antisymmetric binary relation $\sqsubseteq$.

Reflexivity means $a \sqsubseteq a$ for all elements. Transitivity means that if $a \sqsubseteq b$ and $b \sqsubseteq c$, then $a \sqsubseteq c$. Finally, antisymmetry for a binary relation means that $a \sqsubseteq b$ and $b \sqsubseteq a$ implies $a = b$. Why "partial" order? Unlike in so-called *total* orders, there are uncomparable elements in a partial. A total order would require that for all pairs of elements a and b, we have $a \sqsubseteq b$ or $b \sqsubseteq a$ (or both in which case $a = b$, that's to antisymmetry). Good examples of total order are number domains, like $(\mathbb{N}, \leq)$ or similar. Total orders are sometimes also called *linear orders*, since one can visualize them by arranging the elements one a linear straight line.

One can define partial orders (and total orders) also slightly differently, as so-called *strict* partial order, where the relation is not reflexive, and the property corresponding to antisymmtetry needs to be reformulated. In that case, one would write $\sqsubset$ instead of $\sqsubseteq$ (where $\sqsubseteq \triangleq \sqsubset \cup =$), but otherwise it's the same concept. On the following, if comparing things, we will sometimes in the English text say things like "a is larger than b" etc. We meant "larger" to represent $\sqsubseteq$, but we not always to say more precisely "larger-or-equal".

The relation $\sqsubseteq$ allows to say when an element is smaller (or equal) than another, comparing two elements. Obviously, with $\sqsubseteq$ given, one can also say when one element is larger or than two elements. As boring and obvious as that is, there is a name for it, namely being an upper bound. So: c is an *upper bound* of a and b, if $c \sqsupseteq a$ and $c \sqsupseteq b$. Obviously, we assume that the reader understands that $\sqsupseteq$ is "the same" relation than $\sqsubseteq$, just written the other way around. Anyway, the definition of upper bound or, dually, of lower bound of two elements is completely unspectactular, and it does not get more spectacular, if we "generalize" it to speaking of the upper bound of 3, 4, etc., elements. Like c is a lower bound of a_1, a_2 and a_3, if $c \sqsubseteq a_1$, $c \sqsubseteq a_2$, and $c \sqsubseteq a_3$. Actually, we might as well use the concepts on arbitrary sets of elements from A, speaking of a upper resp. lower bound of a set A' where $A' \subseteq A$.

Let's have a closer look on the "nature" of upper and lower bounds. There it becomes slightly more interesting, especially for partial orders. For total orders, like $(\mathbb{N}, \leq)$, the situation is straightforward. Given two numbers n_1 and n_2, they always have upper and lower bounds. Let's assuming wlog ("without loss of generality") that $n_1 \leq n_2$. Then all numbers n_1' with $n_1' \leq n_1$ are lower bounds of n_1 and n_2. Dually, all numbers n_2' with $n_2' \geq n_2$ are upper bounds of n_1 and n_2. So, both upper and lower bounds exists and they are not unique.

But actually, that's not quite true, or at least we have to be careful what we mean by "upper and lower bounds exits" (let's say for $(\mathbb{N}, \leq)$. Certainly, if we take two numbers n_1

and n_2 as above, those bounds exits; the was nothing wrong with the above argumentation. Same for the upper bound of 3, 4, etc. numbers. But what about the following: is there an upper bound of *all even numbers*, i.e., the set $\{0, 2, 4, 6, \ldots\} \subseteq \mathbb{N}$? Well, there is *no* upper bound, at least not inside $\mathbb{N}$. The point here is that there is a qualitative step from asking for the existance of bounds for *finite* sets to the existence of bounds for *infinite* sets.

For the particular setting of the total order $(\mathbb{N}, \leq)$, upper bounds for infinite sets are not guaranteed, for the lower bound, there would always be at least one lower bound, namely 0. For $(\mathbb{Z}, \leq)$, also lower bounds for arbitrary sets are not guaranteed, obviously.

Let's keep on staying with total orders for a while, for concreteness sake just $(\mathbb{N}, \leq)$. We know that, at least when sticking to finite sets, upper and lower bounds do exist, though there are typically more than one lower resp. upper bound. At least for upper bounds, it's guaranteed that there are infinitely many upper bounds for any finite set of elements. Let's focus on upper bounds in the discussion, since for lower bounds it's the same, just the other way around. Now, as said, there are typically many upper bounds, and of course, the $\leq$-relation orders also the bounds themselves. In particular, that we are still in a total order, for any two different upper bounds, one is larger and one is smaller. Actually, one can easily establish: given a set of upper bounds for two numbers n_1 and n_2, there is all *smallest* upper bound. One conventionally would write $\max(n_1, n_2)$ for that, and it's either n_1 or n_2, depending on which is larger (or "both", if $n_1 = n_2$). One could likewise establish that this applies not just to upper bounds of two resp. finitely many elements, it would apply to arbitrary sets of elements. $(\mathbb{N}, \leq)$, however, is not suitable to illustrate this particular point, simply because there is no infinite set of elements that actually has an upper bound. If we switch two $(\mathbb{Z}, \leq)$ instead, there are now infitite sets that *do* have upper bounds. If a given set $A' \subseteq \mathbb{Z}$ has upper bounds, then it's guaranteed that the is a smallest or least upper bound. When swhich to lower bounds instead of upper bounds, one would speak of *greatest lower bounds*, actually, *the* greatest lower bound, because again, it's unique, if it exists. For numbers, one uses min and max to refer to those. The uniquens of least upper bounds is ultimately a consequence of *antisymmetry* of $\leq$. For linear orders, this is very straightforward to see ("assume that there are *two* least upper bounds ...").

For data flow analysis, we are *not* working with linear or total orders, but with partial orders. What changes then? Now, there elements may be *uncomparable*. Additionally, and as a consequence, there may be pairs of (uncomparable) elements, for which no upper bound exists (same for lower bounds). The partially ordered sets are intended as the domain for the "search space". When having two uncomparable safe approximations, then that's still ok, if there is an upper bound; one can take that a approximation which is looser than the two uncomparable ones but still safe. Even better is, if not only there is an upper bound, but there is a *least* upper bound. Like in the total order setting, one can still establish that least upper bounds are unique (if they exists). The argument is a tad less immediate than with total orders, but still very easy and a consequence again of antisymmetry.

Since least upper bounds and greatest lower bounds are so importantant for us, and since partial orders themselves don't guarantee their existance, we just postulate their existance and focus on partial orders with lubs and glbs. Those are called *complete lattices*.

As mentioned, lubs and glbs are *unique*. That's why we can say, *the* greatest lower bounds, for instance, as opposed to just *a* lower bound. Being unique, we can also introduce a notation or symbol for them: we write $a_1 \sqcup a_2$ for the least upper bound of a_1 and a_2, and $a_1 \sqcap a_2$ for their greatest lower bound.

Actually, we are not quite done for *complete* lattices. We stated that complete lattices "have" glbs and lubs, but, as illustrated for total orders using $\mathbb{N}$ or $\mathbb{Z}$, one has to be careful treating *inifinite* sets. So, the fact that binary operators $\sqcup$ and $\sqcap$ exist that denote the best bounds does not mean that one can get those for *arbitrary* subsets, including inifinite ones. For a lattice to be complete, also those have to exists. We write $\bigsqcup L'$ and $\bigsqcap L'$ for the least upper bounds resp. greatest lower bound of an arbitrary subset $L' \subseteq L$ (where L is the set of elements that form the complete lattice).

A (probably obvious) remark on notation. We write $\sqsubseteq$, $\sqsupseteq$, $\sqcup$, $\sqcap$ when talking about lattices in general. The symbols $\subseteq$ $\supseteq$, $\cup$, $\cap$ are used when dealing with *set*, i.e., i.e., denoting the subset resp. superset relations, set union and set intersection. Sets of sets, together with those operations and relations indeed form a lattice, and in the data flow analyses, we have seen examples for that.

Definition 2.3.4 (Lattice)**.** A *lattice* $(L, \sqsubseteq)$ is a partial order for which binary least upper bounds and binary greatest lower bounds exists, i.e., for all a_1 and a_2, there exists $a_1 \sqcup a_2$ and $a_2 \sqcap a_2$. It's a *complete lattice*, if least upper bounds and greatest lower bounds exits, i.e. $\bigsqcup L'$ and $\bigsqcap L'$ for all $L' \subseteq L$.

Why does one need such specific assumptions, insisting on working with lattices? We are trying to find solutions to a constraint problem (data flow equations or $\sqsubseteq$-constraints). We know already, from discussions earlier, that those constraints have more than one solution (there is more than one safe approximation), that solutions are ordered wrt. their precision, and that solving data-flow constraints corresponds to finding a *fixpoint* (resp. a pre-fixpoint or post-fixpoint). Trying to solve a constraint system is, very generally, a *search problem*: find a solution if it exists (or maybe find all). In particular, since here solutions are ordered, find a good solution. The fact that one is after a good solution, for a given measure, makes the search problem to an *optimization* problem.

How fast one can solve such a problem of course depends on the size of the search space. That's clear and not too insightful. When talking about the (computational) complexity one is anyway interested in the computational effort *depending on* the problem size. Later, however, we have a look on what influences the problem size, basically the size of the program but also the level of abstraction used in the data flow problem; that allows to being more precise wrt. information being tracked (for the same problem setting, like focusing on reaching definitions, for instance) , at the price of enlarging the search space. More information will lead to a better, i.e., more precise analysis. Of course, at one given level of abstraction and precision, still there is exactly one best solution. It's only that there may be even better ones, adding more information being tracked (we will see an example of that effect when we look at *interprocedural* data-flow analysis).

Anyway, a more interesting factor in how fast one can solve optimization problems is the "stucture" of the search space, how it is "shaped". Depending on the properties of the given problem, innumerous specific search strategies exits. There are whole research fields

concerned with optimizations in various settings. For us, the "search spaces", being lattices are particularly well-behaved! This is captured by the lattice-requirement together with the fact that the function for which we look for fixpoints, is *monotone* (resp. *continuous* which is a strengthening of being monotone). Now, this setting —continuous functions over complete lattices— is *exactly* guaranteeing existance and uniqueness of best solutions. Since it's uniques, we can call it *the* best solution, not just "a" best solution. By saying "exactly guaranteing", I mean the following: if one looks at the proof of establishing the result, one sees that every single condition is actually needed to carry out the proof. Not more and not less than monotone resp. continuous functions over some form of lattice is needed to obtain the desired result, and thus no make the mononone frameworks work. So, that's what one has to do to drum up with a new data flow analysis: phrase the problem domain for the information of interest in the form of a lattice, formulate (in)-equations in a way that corresponds to a continuous function, and that's it.

Finally, some remarks about *monotone* resp. *continuous* functions. We will see details later, but such much already here. Both conditions on a function are closely related and every continuous function is monotone. The notion of continuity is *not* 100% identical to the concept one may know from *analysis* (as mathematical field), when dealing with functions over the reals $\mathbb{R}$ (where here we are dealing with monotone functions over some lattice). There are connections and it's not a coincidence that in both situations one speaks of continuity, but it's not very important for us. Anyway, the difference for us between monotone and continuous function is: Monotonicity guarantees that a unique best fixpoint *exists* mathematically. Continuity is a constructive thing: not only does such a best solution exits, there is an algorithmic way to *compute it* (or, to be more precise, approximate it arbitrarily). So there exists a fixpoint algorithm sometimes called fixpoint iteration. Being monotone is kind of like a "finite" property of a function, whereas continuity states how the function behave "in the limit"; monotonicity states, when taking *two* elements, one larger than the other, and when feeding them to a function, the two resulting outcomes are ordered the same way. Continuity requires a similar thing, when feeding an *infinitely long series* of increasing values into a function, not just 2 elements.

As computer scientists, we are after "algorithms", so that means, continuity is what we want. However, in many cases actually, we are in an even better situation. The problem domain, the lattice, is often *finite*. Like for the examples we have seen: in a given program, there are only finitely many labels, finitely many variables, finitely many expressions, etc. Thus, there are only finitely many sets of such things, so everything is finite. In that case, the distinction between monotonicity and continuity *disappears*. In finite settings, both notions coincide, as does the distinction between lattices and *complete* lattices. That means also: not only can we approximate the desired fixpoint arbitrarily close, we can actually *reach* it *exactly* in a finite amount of time by the mentioned fixpoint iteration scheme. And that's what a data flow analyser does.

Intermezzo: orders, lattices etc.

as a reminder:

- partial order $(L, \sqsubseteq)$
- *upper bound* l of $Y \subseteq L$:

- *least* upper bound (lub): $\bigsqcup Y$ (or *join*)
- dually: lower bounds and greatest lower bounds: $\bigsqcap Y$ (or *meet*)
- **complete lattice** $L = (L, \sqsubseteq) = (L, \sqsubseteq, \bigsqcap, \bigsqcup, \bot, \top)$: a partially ordered set where meets and joins exist for *all subsets*, furthermore $\top = \bigsqcap \emptyset$ and $\bot = \bigsqcup \emptyset$.

There are also other forms of lattices, for instance, if one only needs joins, but not meets, one can get away with a semi-lattice, and there are many more variations. For the lecture, we generally simply assume *complete lattices* and thus, the montone framework is happy. In particular, if we are dealing with *finite* lattices, which is an important case, we don't need to consider *infinite* sets, and "standard" lattices with binary meets and joins (and least and largest elements) are complete already.

Fixpoints

given: complete lattice L and monotone $f : L \to L$.

- **fixpoint:** $f(a) = a$

$$Fix(a) = \{a \mid f(a) = a\}$$

- f *reductive* at a, a is a **pre-fixpoint** of f: $f(a) \sqsubseteq a$:

$$Red(f) = \{a \mid f(a) \sqsubseteq a\}$$

- f *extensive* at a, a is a **post-fixpoint** of f: $f(a) \sqsupseteq a$:

$$Ext(f) = \{a \mid f(a) \sqsupseteq a\}$$

Define "lfp" / "gfp"

$$lfp(f) \triangleq \bigsqcap Fix(f) \quad \text{and} \quad gfp(f) \triangleq \bigsqcup Fix(f) \tag{2.6}$$

We define the concept of fixpoint for monotone functions over a lattice. Strictly speaking, one does not need those assumptions for *defining* what a fixpoint is, one just need a function $f : A \to A$ without more assumptions on A. In general, though, without additional assumptions, like the ones here on A and f, fixpoints may or may not exists.

Equation (2.6) just gives the *names* to the two elements of the lattice defined by the corresponding right-hand sides. We know that those elements do exist, thanks to the fact that L is a complete lattice and $lfp(f)$ and $gfp(f)$ as defined are unique elements of the lattice. The chosen names suggest that the two thusly defined elements are the least fixpoint, resp. the greatest fixpoint of the monotone function f. But that requires a separate *argument*. Finally, if we take it really serious, an argument should be found that allows to speaking of *the* least fixpoint. If there is more than one least fixpoint, one should avoid talking about "the least fixpoint" (same for the largest fixpoint). The argument for uniquess of least fixpoints (or for greatest fixpoint) is very simple though, similar to arguing for the uniqueness of "the least upper bound" etc.

If one would carry out the argument, i.e., the proof, that all fits together in the sense that the $lfp(f)$ and $gfp(f)$ defined above *are actually* the least fixpoint and the largest fixpoint, and if one would carefully keep track of what is actually needed to make the proof go through step by step, then one would see that *every single condition* for being a complete lattice is needed (plus the fact that f is monotone). If one removes one condition, the argument fails! Conversely that means the following: We are interested in uniquely "best approximations" (least or greatest fixpoints depending in whether it's a may or a must analysis),

> and, having a monotone f, a complete lattices **is exactly what guarantees that those fixpoints exists**. Exactly that, nothing less and nothing more, If your framework has monotone functions and is based on a complete lattice, it works. If not, it does not work, very simple.

That explains the importance of lattices and monotone functions. Also, I would guess that historically, the *need* to assure existance of fixpoints has led Tarski (the mathematician whose concepts we are currently covering) exactly to the definition of lattice, not the other way around ("oh, someone defined some lattice, let's see what I can find out about them, perhaps I could define some $lfp(f)$ like above and see if I could prove something iteresting about it, perhaps it's a fixpoint?". But as said, that is speculation. One may also remark: there are other "fixpoint theories" around. The one here is based on lattices and partial orders, since that is what corresponds to approximations: they are ordered by their precision. These other, non-lattice based kind of conditions one could use to ensure existance of fixpoints don't play a role for our purposes

Having stressed the importance of complete lattices, for fairness sake it should be said that there's also a place for analyses which fail to meet those conditions. In that case, one might not have a (unique) best solution. Perhaps even worse (and related to that), one might need *combinatorial* techniques (like backtracking), i.e., checking all possible solutions to find an acceptable one. If that happens, the *cost* of the analysis may explode. To avoid that one may give up to look for a "best solution" and settle for a "good enough" one and heuristics that hopefully find an acceptable one efficiently, or even throw the towel and give up "soundness". Anyway and fortunately, plenty of important analyses fit well into the monotone framework with its lattices, its unique best solution and —perhaps best of all– its efficient solving techniques. Therefore this lecture will cover only those here. Those are called classical *data flow analyses*.

Tarski's theorem

Core Perhaps core insight of the whole lattice/fixpoint business: not only does the $\bigsqcap$ of all pre-fixpoints uniquely exist (that's what the lattice is for), but —and that's the trick— *it's a pre-fixpoint* **itself** (ultimately due to montonicity of f).

Theorem 2.3.5. *L: complete lattice, $f : L \to L$ monotone.*

$$\begin{array}{lcll} lfp(f) & \triangleq & \bigsqcap Red(f) & \in \quad Fix(f) \\ gfp(f) & \triangleq & \bigsqcup Ext(f) & \in \quad Fix(f) \end{array} \tag{2.7}$$

- Note: *lfp* (despite the name) is *defined* as glb of all pre-fixpoints

- The theorem (more or less directly) implies *lfp* is the *least* fixpoint

Fixpoint iteration

- often: iterate, approximate least fixed point from below $(f^n(\bot))_n$:

$$\bot \sqsubseteq f(\bot) \sqsubseteq f^2(\bot) \sqsubseteq \dots$$

- not assured that we "reach" the fixpoint ("within" ω)

$$\bot \sqsubseteq f^n(\bot) \sqsubseteq \bigsqcup_n f^n(\bot) \quad \sqsubseteq \quad \begin{array}{l} lfp(f) \\ gfp(f) \quad \sqsubseteq \bigsqcap_n f^n(\top) \sqsubseteq f^n(\top) \sqsubseteq (\top) \end{array}$$

- additional requirement: **continuity** on f for all ascending chains $(l_n)_n$

$$f(\bigsqcup_n (l_n)) = \bigsqcup (f(l_n))$$

- *ascending chain condition* ("stabilization"): $f^n(\bot) = f^{n+1}(\bot)$, i.e., $lfp(f) = f^n(\bot)$
- *descending* chain condition: dually

Basic preservation results

Lemma 2.3.6 ("Smaller" graph → less constraints)**.** *Assume* $live \models \mathsf{LV}^{\subseteq}(S_1)$*. If* $flow(S_1) \supseteq flow(S_2)$ *and* $blocks(S_1) \supseteq blocks(S_2)$*, then* $live \models \mathsf{LV}^{\subseteq}(S_2)$*.*

Corollary 2.3.7 ("subject reduction")**.** *If* $live \models \mathsf{LV}^{\subseteq}(S)$ *and* $\langle S, \sigma\rangle \to \langle \acute{S}, \acute{\sigma}\rangle$*, then* $live \models \mathsf{LV}^{\subseteq}(\acute{S})$

Lemma 2.3.8 (Inter-block flow)**.** *Assume* $live \models \mathsf{LV}^{\subseteq}(S)$*. If* $l \to_{flow} l'$*, then* $live_{exit}(l) \supseteq live_{entry}(l')$*.*

The three mentioned results are actually pretty straightforward, resp. express properties of the live variable analysis which should be (after some reflection) pretty obvious. Analogous results would hold for other data flow analyses. Lemma 2.3.6 compares the analyses results for two programs S_1 and S_2, where S_2 has a "smaller" control-flow graph (less edges and/or less blocks). Since the control flow graph directly corresponds to sets of constraints, removing parts of the graph means removing constraints. That means, *more* solutions are possible, which is expressed by the lemma ($live \models \mathsf{LV}^{\subseteq}(S)$ means that *live* (an assignment of liveness information to all variables of the constraint system) satisfies the constraint system of the program S.

It's probably obvious: the variables of the type system are (of course) not the program variables of the live variable analysis. The constraint variables are the (entry and exit points of the) nodes of the graph (which in turn correspond to the labels in the labelled abstract syntax).

The Corollary 2.3.7 is a direct consequence of that. In general, that's what the term "corollary" means: an immediate interesting follow-up of a preceding lemma or theorem etc.

However, the result is *not* without subtelty. It has to do with the step $\langle S, \sigma\rangle \to \langle \acute{S}, \acute{\sigma}\rangle$, resp, what this step does to the (labelled) program S. The interesting case for that is step covered by one of the rules dealing with the while-loop, namely WHILE$_1$. It's interesting insofar as that it *duplicates* the body of the loop. That leads to a program what is **no longer uniquely labelled** (even if S had been)! It's however still *label consistent*.

The last lemma is a direct consequence of the construction (backward may analysis), where the interblock constraints connect the exits of a pre-block (here l) with the entries of the post-block (here l') These lemmas as such are not interesting in themselves.

Correctness relation

- basic intuitition: **only live variables influence the program**
- proof by *induction*

$\Rightarrow$

Correctness relation on states: Given V = set of variables:

$$\sigma_1 \sim_V \sigma_2 \text{ iff } \forall x \in V. \sigma_1(x) = \sigma_2(x) \tag{2.8}$$

$$\begin{array}{ccccccccc}
\langle S, \sigma_1\rangle & \longrightarrow & \langle S', \sigma_1'\rangle & \longrightarrow & \ldots & \longrightarrow & \langle S'', \sigma_1''\rangle & \longrightarrow & \sigma_1''' \\
\big|_{\sim_V} & & \big|_{\sim_{V'}} & & & & \big|_{\sim_{V''}} & & \big|_{\sim_{X(l)}} \\
\langle S, \sigma_2\rangle & \longrightarrow & \langle S', \sigma_2'\rangle & \longrightarrow & \ldots & \longrightarrow & \langle S'', \sigma_2''\rangle & \longrightarrow & \sigma_2'''
\end{array}$$

Notation: $N(l) = live_{entry}(l)$, $X(l) = live_{exit}(l)$

In the definition of $\sim_V$ above, V is an arbitrary set of "variables". The intention (in the overall argument) will be, that the V's are those variable that are live (resp. variables that the analysis has marked as live). Of course, the set of variables being determined as live *changes* during execution.

In the figure above, the "control-part" of the component, i.e., the code S, S' etc., are identical step by step for both versions. Both program execute the very same steps.

As a side remark: the while language is *deterministic*, meaning a program code S and a state σ *determines* the successor configuration (if we are not yet at the final configuration). Note also: the intra-block (and backward) definition of liveness *directly* gives that for an assignment $x := a$, the free variables in a are live right in front of the assignment. Likewise, variables in a boolean condition b are live right in front of a conditinal or loop, to which b belongs. Those variables therefore are contained in the V-set directly before a step for the two variants of the system. Consequently, both system do *exactly* the same next step. And then the next step is the same again, and then the next .... I.e., by *induction* both systems behave the same, which is exactly what we want to establish ("dead variables don't matter").

Correctness (1)

Lemma 2.3.9 (Preservation inter-block flow)**.** *Assume live* $\models \mathsf{LV}^{\subseteq}$*. If* $\sigma_1 \sim_{X(l)} \sigma_2$ *and* $l \rightarrow_{flow} l'$*, then* $\sigma_1 \sim_{N(l')} \sigma_2$*.*

Correctness

Theorem 2.3.10 (Correctness)**.** *Assume live* $\models \mathsf{LV}^{\subseteq}(S)$*.*

- *If* $\langle S, \sigma_1\rangle \rightarrow \langle \acute{S}, \acute{\sigma}_1\rangle$ *and* $\sigma_1 \sim_{N(init(S))} \sigma_2$*, then there exists* $\acute{\sigma}_2$ *s.t.* $\langle S, \sigma_2\rangle \rightarrow \langle \acute{S}, \acute{\sigma}_2\rangle$ *and* $\acute{\sigma}_1 \sim_{N(init(\acute{S}))} \acute{\sigma}_2$*.*
- *If* $\langle S, \sigma_1\rangle \rightarrow \acute{\sigma}_1$ *and* $\sigma_1 \sim_{N(init(S))} \sigma_2$*, then there exists* $\acute{\sigma}_2$ *s.t.* $\langle S, \sigma_2\rangle \rightarrow \acute{\sigma}_2$ *and* $\acute{\sigma}_1 \sim_{X(init(S))} \acute{\sigma}_2$*.*

$$\begin{array}{ccc} \langle S,\sigma_1\rangle & \longrightarrow & \langle \acute{S},\acute{\sigma}_1\rangle \\ \big| \sim_{N(init(S))} & & \vdots \sim_{N(init(\acute{S}))} \\ \langle S,\sigma_2\rangle & \cdots\!\rightarrow & \langle \acute{S},\acute{\sigma}_2\rangle \end{array} \qquad \begin{array}{ccc} \langle S,\sigma_1\rangle & \longrightarrow & \acute{\sigma}_1 \\ \big| \sim_{N(init(S))} & & \vdots \sim_{X(init(S))} \\ \langle S,\sigma_2\rangle & \cdots\!\rightarrow & \acute{\sigma}_2 \end{array}$$

The picture are drawn in a "specific" manner to capture the formulation of the theorem. In particular see the use of "solid" arrows and lines vs. "dotted" ones. That a diagrammatic way to indicate the "for all such ..." (solid) and "... there exists some ..." (dotted). This notation is rather standard, and allows to express such properties in a short diagrammatic but still precise manner.

Correctness (many steps)

Assume $\mathit{live} \models \mathsf{LV}^{\subseteq}(S)$

- If $\langle S, \sigma_1\rangle \rightarrow^* \langle \acute{S}, \acute{\sigma}_1\rangle$ and $\sigma_1 \sim_{N(init(S))} \sigma_2$, then there exists $\acute{\sigma}_2$ s.t. $\langle S, \sigma_2\rangle \rightarrow^* \langle \acute{S}, \acute{\sigma}_2\rangle$ and $\acute{\sigma}_1 \sim_{N(init(\acute{S}))} \acute{\sigma}_2$.
- If $\langle S, \sigma_1\rangle \rightarrow^* \acute{\sigma}_1$ and $\sigma_1 \sim_{N(init(S))} \sigma_2$, then there exists $\acute{\sigma}_2$ s.t. $\langle S, \sigma_2\rangle \rightarrow^* \acute{\sigma}_2$ and $\acute{\sigma}_1 \sim_{X(l)} \acute{\sigma}_2$ for some $l \in \mathit{final}(S)$.

2.4 Monotone frameworks

We have seen 4 different classical analyses, which all shared some similarities. In this section, those analyses will be systematically put into a larger context, which is known as *monotone framework*. As unifying principle, this was first formulated by Kildall [4] and constitutes in a way the common orthodox and completely standardized understanding of what classical data flow analysis is.

Besides that, it captures many known analyses, it's also a "recipe" for designing other data flow analyses, starting from the program given in the form of a control flow graph. Indeed, the 4 analyses we have seen are only (important) representatives of the 4 classes of analyses that can be formulated as monotone framework. The analysis can be forward or backward, and it can be "may" or "must". That's about it, and that gives 4 different classes.

Besides that, the monotone framework concept lays down exactly what needs to be assumed about the structure of the information that is given back from the analysis. All four analyses somehow dealt with *sets*, like "sets of variables such that this and that" or "sets of expressions such that this or that". Dealing with sets reflected the fact that, being static, the analysis does not *exactly* knows what the program does and has to approximate. Dealing with sets of pieces of flow information also allows to enlarge or shrink the information via taking the subset or the superset. Which direction is safe in a given analysis depends on whether on whether it's a "may" or a "must" analysis. At any rate, sets and the subset relations is a special case of the notion of the more general notion of **lattice**, which is exactly the notion needed to make the monotone framework work.

Even if the monotone framework is based on the general notion of lattice for good reason, the special case of sets und subsets is an important one. Not only is it conceptually simple, it also allows efficient implementations. Finite sets over a given domain may be implemented as *bit vectors* and union and intersection, two crucial operations for "may" resp. "must" analyses can efficiently be implement via logical bitwise "or" resp. "and" on bitvectors.

Monotone framework: general pattern

$$\begin{array}{lcll} \mathit{Analysis}_\circ(l) & = & \begin{cases} \iota & \text{if } l \in E \\ \bigsqcup\{\mathit{Analysis}_\bullet(l') \mid (l', l) \in F\} & \text{otherwise} \end{cases} \\ \mathit{Analysis}_\bullet(l) & = & f_l(\mathit{Analysis}_\circ(l)) \end{array} \tag{2.9}$$

- $\bigsqcup$: either $\bigcup$ or $\bigcap$
- F: either $\mathit{flow}(S_*)$ or $\mathit{flow}^R(S_*)$.
- E: either $\{\mathit{init}(S_*)\}$ or $\mathit{final}(S_*)$
- ι: either the initial or final information
- f_l: **transfer function** for $[B]^l \in \mathit{blocks}(S_*)$.

The definition is "generic" as it leaves open the alternatives "may" vs. "must" as well as "forward" vs. "backard". Another ingredient is the "flow information of interest" and the special case of what the information at initial node resp. the final node is supposed to be. As we have discussed especially in connection with live variable analysis, this has to be decided an a case-by-case consideration, depending on specific conditions of the intended analysis and the language. One final ingredient is the *transfer function*. We have encountered that concept implicitly for the *intra-block* data flow. It's only that we did not explicitly call it transfer function; instead the concept was formulated making use of *kill* and *generate* function. It turns out that many transfer functions can be formulated, as we did, via kill and transfer functions (as many analysis domains are sets of information of interest), but not all. The general monotone framework simply requires a function that transform flow information on one end of a basic block to flow information at the other

end. For a forward analysis, the flow information at the exit of a basic block is expressed as a function on the entry of the block, and for backward information, it's the other way around.

Monotone frameworks

direction of flow:

- **forward** analysis:
  - $F = \mathit{flow}(S_*)$
  - $\mathit{Analysis}_\circ$ for entry and $\mathit{Analysis}_\bullet$ for exits
  - assumption: isolated entries
- **backward** analysis: dually
  - $F = \mathit{flow}^R(S_*)$
  - $\mathit{Analysis}_\circ$ for exit and $\mathit{Analysis}_\bullet$ for entry
  - assumption: isolated exits

sort of solution

- **may** analysis
  - properties for *some* path
  - *smallest* solution
- **must** analysis
  - properties of /all paths
  - *greatest* solution

Into which of the four categories a concrete analysis falls need to be thought through on a case-by-basis of course. However, it may not be a clean cut as it seems, resp. it may also be a matter of perspective. For example, live variable analysis. That one is a *may* (and a backward) analysis. We can switch perspective from concentrating on *live* variables to "dead" variables (those wich are not live), still with the same purpose of recylcing memory of variables with are not live = dead. If the data flow analysis streams sets of deads variables throught its equations instead of live variables, the analysis will be a *must* analysis instead. After all, a variable is dead if it's not used in the future *on all paths* (which is the dual of being live, which refers to usage *on some path*). Consequently, one would be interested in the largest safe solution of dead variables.

In a way, both are the "same" analysis, or rather, *dual* to each other but ultimately equivalent. It's only that convetionally, it's referred to as "live variable analysis" and not as the more morbid dual one.

This switching to the dual perspective is easily possible if we are dealing with *finite* domains in the analysis, as we often do. Like in live variable analysis, there are only finitely many sets of variables.

Basic definitions: property space

- *property space* L, often *complete lattice*
- *combination* operator: $\bigsqcup : 2^L \to L$, $\sqcup$: binary case
- $\bot = \bigsqcup \emptyset$
- often: ascending chain condition (stabilization)

The *property space* (here called L) captures the "information of interest" (sets of variables, sets of non-trivial expressions ...). Technically, it needs to be some form of *lattice* (see the corresponding section). In good approximation, the lattices and its laws resemble closely the situation with sets of information (with $\cup$, $\cap$, $\subseteq$, $\supseteq$, $\emptyset$ ...). Indeed, the power set of some set is an important special case of a lattice.

Transfer functions

$$f_l : L \to L$$

with $l \in \mathbf{Lab}_*$

- associated with the *blocks*
- requirement: *monotone*
- $\mathcal{F}$: monotone functions over L:
  - containing all *transfer functions*
  - containing *identity*
  - *closed under composition*

The transfer functions, as defined above, are attached to the elementary blocks (which here contain one single statement or expression, but in general may contain staight-line code). In other accounts, the control flow graphs and/or the transfer functions may be differently represented, without changing anything relevant. For instance, here, the *nodes* of the CFG contain assigmments and expression (i.e., relevant pieces of abstract *syntax*). Also the transfer functions are attached to the nodes. One can see it like the transfer function is the *semantics* of the corresponding piece of syntax. Not the *"real"* semantics, but on the chosen *abstraction level* of the analysis, i.e., on the level of the *property space* L.

Some authors prefer to attach the pieces of syntax and/or the transfer functions to *edges* of a control-flow graph (which therefore is of a slighty different format than the one we operate with). But it's only a different reprentation of the same principles.

Summary

- complete lattice L, ascending chain condition
- $\mathcal{F}$ monotone functions, closed as stated
- **distributive** framework

$$f(l_1 \sqcup l_2) = f(l_1) \sqcup f(l_2)$$

Instead of the above condition, one might require

$$f(l_1 \sqcup l_2) \sqsubseteq f(l_1) \sqcup f(l_2) \ .$$

This weaker condition is enough as the other way around $f(l_1 \sqcup l_2) \sqsupseteq f(l_1) \sqcup f(l_2)$ follows by monotonicity of f and the fact that $\sqcup$ is the *least* upper bound.

The 4 classical examples

- for a label consistent program S_*, all are *instances* of a monotone, distributive, framework:
- conditions:
  - lattice of properties: immediate (subset/superset)
  - ascending chain condition: *finite* set of syntactic entities
  - *closure* conditions on $\mathcal{F}$
    - monotone
    - closure under identity and composition
  - *distributivity*: assured by using the kill- and generate-formulation

Overview over the 4 examples

	avail. epxr.	reach. def's	very busy expr.	live var's
L	$2^{\mathbf{AExp}_*}$	$2^{\mathbf{Var}_* \times \mathbf{Lab}_*^?}$	$2^{\mathbf{AExp}_*}$	$2^{\mathbf{Var}_*}$
$\sqsubseteq$	$\supseteq$	$\subseteq$	$\supseteq$	$\subseteq$
$\bigsqcup$	$\bigcap$	$\bigcup$	$\bigcap$	$\bigcup$
$\bot$	$\mathbf{AExp}_*$	$\emptyset$	$\mathbf{AExp}_*$	$\emptyset$
ι	$\emptyset$	$\{(x, ?) \mid x \in fv(S_*)\}$	$\emptyset$	$\emptyset$
E	$\{init(S_*)\}$	$\{init(S_*)\}$	$final(S_*)$	$final(S_*)$
F	$flow(S_*)$	$flow(S_*)$	$flow^R(S_*)$	$flow^R(S_*)$
$\mathcal{F}$	$\{f : L \to L \mid \exists l_k, l_g.\ f(l) = (l \setminus l_k) \cup l_g\}$			
f_l	$f_l(l) = (l \setminus kill([B]^l) \cup gen([B]^l))$ where $[B]^l \in blocks(S_*)$			

2.5 Equation solving

We know how to interpret the data flow problem as a constraint system, either in the form of equational constraints or inclusion constraints (typically subset constraints). We are aware that the constraints are formulated over a domain which forms a complete lattice, and that a solution to such a data flow constraint system corresponds to finding a fixpoint (resp. a pre-fixpoint or a post-fixpoint). In general terms, remembering in particular Kleene's result about fixpoint iteration, we also know how to calculate such fixpoint iteratively. The section just fleshes that out a bit. He even has seen a very schematic formulation of an corresponding iterative approximation algo in the introcuction, called *chaotic iteration*.

The algorithm works randomly, i.e., it follows a ramdom strategy. One could say, it follows no strategy at all. One could also say, doing random moves, it does the most general strategy. Since one can show that even this arbirary behavior still is guaranteed to find the fixpoint, all other more specific strategies or heuristics will find it as well. So while chaotic interation is not really practical, it assures: feel free to come up with smart stragegies, depth-first, priority queues, some other heuristic. Whatever heuristics you do, it will still be *correct!*. So, any heuristics is a matter of optmization, not of correctness. That's a very comforting setting.

In this section, we don't do many different heuristics, we show just one organization of how to steer the approximation algorithm. That general form of algorithm is known as *worklist algorithm*. The name comes from a central data structure, that steers the algorithm through its seach space, which in this case means, which particular constraint to tackle next. Since constraints care connected with edges or nodes in the graph, this can also be seen of a traversal strategy though the control flow graph. Chaotic iteration corresponds to randomly pick edges or nodes from the graph. Note that chaotic means really chaotic, not *following* edges randomly, i.e, following some random path but it means picking edges (or constraints) that needs treatment randomly, "jumping" arbitrarily through the graph.

Worklist data structure

Calling the data structure a "worklist" is also a misnomer in a way. It's a collection of elements, but, despite the name, it needs not be a *list*. It could be a stack, a queue, or some other form (like a priority list). It's a collection data structure, containing the "work" still to be done. That work, in our case, corresponds to constraints still not satified or solve, or edges to traverse. Depending on whether the collection follows a FIFO stategy (queue) or LIFO strategy (stack) or another more complex one influences the traversal (breadth-first, depth-first, or something more complex).

Comparison to the chaotic interation

One may compare the worklist approach to the chaotic interation. One connection seems clear: the chaotic appoach randomly picks a "piece of work". That could be seen as a situation, where the worklist is behaves randomly (like a *set*). What's a piece-of-work anyway? In the chaotic iteration, a piece-of-work is one constraint that needs repair, i.e., some j where

$$\mathsf{RD}_j \neq F_j(\vec{\mathsf{RD}}) \ . \tag{2.10}$$

Now, the chaotic iteration does not make explicit how to determine that. *Searching* for such violations is wasteful, especially since it has to be done over and over again. It's better to keep the work to be done in a dedicated data structure to avoid such repeated checks.

Looking carefully, the worklist do *not* contain exactly all current situations described by equation (2.10). It contains a *approximation* thereof! The worklist contains a *superset* of the exact set of violated constraints, a set of constraints that are suspected to be violated and thus potentially in need to further treatment.

That can be seen later in the worklist algo: for once, the initialization fills up the worklist with *all* constraints, independent of whether they are initially violated or not. Secondly, when removing a piece of work from the worklist, it's first check if, at that point, the corresponding contraint actually needs treatment; it may well be, it's already satified, as the worklist contains an overapproximation of pieces of work.

So, another trick is [continue here the argument]

Equation solving and traversal

The worklist data structure can be used to steer the way the constraints (equations or inequations) are solved. It contains the "work still to be done", i.e., individual constraints that, in the current state of the overall iteration, are not yet satisfied.

Solving the analyses

- given: set of equations (or constraints) over finite sets of variables
- domain of variables: complete lattices + ascending chain condition
- *2 solutions* for the monotone frameworks
  - **MFP**: "maximal fix point"
  - **MOP**: "meet over all paths"

Finally, we come to address how to solve the equations. We have seen two glimpses to the problem. One was at the introduction, the chaotic iteration, the other one was the "theory" related to the fixpoints. We will shortly revisit the chaotic iteration. What was lacking there in the introduction was a more *concrete* (= deterministic) realization.

MFP

- terminology: historically "MFP" stands for *maximal* fix point (not minimal)
- iterative **worklist** algorithm:
  - central data structure: *worklist*
  - list (or container/set) of pairs
- related to *chaotic iteration*

Chaotic iteration

Input: equations for reaching defs
for the given program
Output: least solution: $\vec{\mathsf{RD}} = (\mathsf{RD}_1, \ldots, \mathsf{RD}_{12})$

Initialization:
$\mathsf{RD}_1 := \emptyset; \ldots; \mathsf{RD}_{12} := \emptyset$
Iteration:

while $\mathsf{RD}_j \neq F_j(\mathsf{RD}_1, \ldots, \mathsf{RD}_{12})$ for some j
do
$\mathsf{RD}_j := F_j(\mathsf{RD}_1, \ldots, \mathsf{RD}_{12})$

Worklist algorithms

- *fixpoint* iteration algorithm
- general kind of algorithms, for DFA, CFA, ...
- same for *equational and /constraint* systems
- "specialization" i.e., *determinization* of chaotic iteration

⇒ **worklist**: central data structure, "container" containing "the work still to be done"

- for more details (different traversal strategies): see Chap. 6 from [7]

WL-algo for DFA

- WL-algo for *monotone frameworks*

⇒ input: instance of monotone framework

- two central data structures
  - **worklist**: *flow-edges* yet to be (re-)considered:
    1. *removed* when *effect* of transfer function has been taken care of
    2. *(re-)added*, when point 1 *endangers* satisfaction of (in-)equations
  - **array** to store the "current state" of $Analysis_\circ$
- one central *control structure* (after *initialization*): loop until worklist empty

Remember that the result of the analysis is a mapping from the entry *and* the exit points for each block. An array is of course a good representation of a finite function. Here, during the run, only the *entry* blocks are stored.

Why does the algo operate only with the "entry"-parts of the blocks (assuming a forward analysis)? In the chaotic iteration we clearly see pre- and post-states. There, an RD_i depends via F on *all* RD_j. Of course, in reality that's not the case, and moreover, we should distinguish between entry and exit points. For the exit points, of course, they only depend on the entry-point and nothing else. The worklist algorithm actually considers only the relation from the *post*-condition to the pre-condition, more precisely *one pre-condition*. That means, that only the *inter*-flow is actually checked. So, in some sense, the post-conditions *are* represented, but only *implicitely* in that they are *calculated* on the fly from the given pre-condition, when needed. That can be seen also in step 3.

Code

```
Input:  (L, ℱ, F, E, ι, f)
Output: MFP∘, MFP•
Method: step 1: initialization
                W := nil;
                for all (l, l') ∈ F do W := (l, l') :: W;
                for all l ∈ F or ∈ E do
                   if l ∈ E then Analysis[l] := ι
                            else Analysis[l] := ⊥L;
        step 2: iteration
                while W ≠ nil do
                  (l, l') := ( fst(head(W)), snd(head(W)));
                  W := tail W;
                  if fl(Analysis[l]) ⋢ Analysis[l']
                  then  Analysis[l'] := Analysis[l'] ⊔ fl(Analysis[l]);
                        for all l'' with (l', l'') ∈ F do
                              W := (l', l'') :: W;
         step 3: presenting the result:
                  for all l ∈ F or ∈ E do
                    MFP∘(l) := Analysis[l];
                    MFP•(l) := fl(Analysis[l])
```

ML Code

```
let rec solve (wl1 : edge list) : unit =
  match wl1 with
  |   [] -> ()                                        (* wl done *)
  |   (l,l')::wl' ->
      let ana_pre   : var list = lookx (ana,l) (* extract ``states *)
      and  ana_post : var list = lookx (ana,l')
      in let ana_exitpre : var list = f_trans(ana_pre,l)
      in
      if not (subset (ana_exitpre,ana_post))
      then
        (enter (ana,l',union(ana_post,ana_exitpre));
         let (new_edges : edge list) =
           (let (preds : node list) = Flow.Graph.pred (l')
           in List.map (fun n -> (l',n)) preds)
         in solve (new_edges @ wl')
        )
      else                                  (* Nothing to do here. *)
        (solve (wl'))
in
solve wl_init;
fun (x: node) -> lookx (ana, x)
;;
```

This is slice of the code of an implementation, I once did. Actually, it's available at `https://github.uio.no/msteffen/saprojects`.

MFP: properties

Lemma 2.5.1. *The algo*

- *terminates and*
- *calculates the least solution*

Proof.
- termination: ascending chain condition & loop is enlarging
- least FP:
  - invariant: array always below $Analysis_\circ$
  - at loop exit: array "solves" (in-)equations

□

Time complexity

- estimation of *upper bound* of number basic steps
  - at most b different labels in E
  - at most $e \geq b$ pairs in the flow F
  - height of the lattice: at most h
  - non-loop steps: $O(b + e)$
  - *loop*: at most h times addition to the WL

$\Rightarrow$

$$O(e \cdot h) \tag{2.11}$$

or $\leq O(b^2 h)$

2.6 Interprocedural analysis

2.6.1 Introduction

This section makes the analysis, resp. the language a bit more realistic. So far it was really a minimalistic set-up, the most reduced form of imperative language. Now we add *procedures*. That leads to *interprocedural analysis*. Adding procedures requires first to extend the syntax, of course, and also the operational semantics.

Extending the syntax is not a big issue. For the semantics we had before an SOS-formulation, a structural operational semantics. We will stick to that, but the rules dealing with calls and returns look already a bit more complex. For the semantics, the core problem is as follows: with procedures, we have to accomodate calls and returns and their LIFO discipline: procedure activations are nested and the callee always returns to the latest call. On top of that, procedures have local variables and formal parameters. Actually, we will have only formal parameters as procedure-local variables. There will be no syntax to introduce further local variables. Covering also that and adapting the semantics correspongingly would be only mildly more complex compared with formal parameters as the only form of local variables. At any rate, the semantics then needs to cover things like organizing the corresponding *memory*, storing local variables, and arranging for parameter passing. In compiler terms, that is part of the *runtime enviroment*. Global variables as in the pure while language so far are of course also part of the run-time enviromnemt, it's just a more quite more simple part of it, namely part of the static memory.

The formalization of the run-time environment, i.e. here, the organization of the memory for procedures is more complex (being *dynamic*). So far we had a finite amount of variables per program, statically fixed. Now, since procedures can be called recursively, the amount of memory to store the content of local variables resp. parameters is not longer statically fixed. The language extension supports *first-order* procedures of functions, but not higher-order procedures. As a consequence, the memory for the procedure local variables can be

arranged in a *stack*. There are different ways how one can formalize that. For instance, one could explicitly introduce some stack with push an pop etc. We do a stack (because that's how it's done), but it's not so visible, it's more implicit (using a syntactic bind-construct, see later). Instead of a *state* (written σ) as before, the memory is represented for fine-grained, given in terms of *locations* and *stores*. Basically, that representation makes explicit that variables are connected to addresses (locations) and one can allocate new addressess when needed. That allocation works in a LIFO-manner, i.e., it works like a stack. Though, as said, we don't introduce an explicit stack structure.

Adding procedures

- so far: *very simplified* language:
  - minimalistic imperative language
  - reading and writing to variables plus
  - simple control flow, given as flow graph
- now: *procedures*: **interprocedural** analysis
- complications:
  - calls/return (control flow)
  - parameter passing
  - scopes
  - higher-order functions/procedures
- here: top-level procedures, mutual recursion, *call-by-value*
  - *call-by-result*

Extending the syntax

- $\texttt{begin}\, D_*\ S_*\, \texttt{end}$

$$D ::= \texttt{proc}\, p(\texttt{val}\, x, \texttt{res}\, y)\texttt{is}^{l_n} S \texttt{end}^{l_x} \mid D\ D$$

- procedure names p
- statements

$$S ::= \ldots [\texttt{call}\, p(a, z)]^{l_c}_{l_r}$$

- note: call statement with *2 labels*
- *statically scoped* language, CBV parameter passing (1st parameter), and CBN for second
- mutual recursion possible
- assumption: unique labelling, only declared procedures are called, all procedures have different names.

The language is still rather simple. It features *first-order* procedures. The procedures, addiontally, cannot be nested. And furthermore, the procedures need to be defined "before" the main part of the program is defined (in the form of a "statement". However, recursive procedure definitions are allowed

For those from the compiler course. The language corresponds roughly to the language of the obligs there, as far as procedures are concerned. There are syntactic differences: what is called statement here, the body of the program, had to be put into a procedure called `main` or similar. But those are questions of concrete syntax, here we are not concerned with that. Other difference (for now) are that here we don't support heap-allocated data (like records or objects). But that is an issue *orthogonal* to the question of procedures.

Parameter passing What might look a bit unusual is the parameter passing mechanism. It highlights the fact that parameter passing is typically not a one-way street. It's not just about passing actual parameters from the caller to the callee. It also involves a mechanism the pass results back. Of course, one can evaluate a procedure with the intention of getting its side effect only (with a return type like `void` or similar). That is often done when doing call-by-reference, which we don't do. Side-effects would have to be done one the global variables in the current language. That probably should not be called "passing the result back" though it can be used that way. It should not even be called *evaluating* the procedure. The word "evaluating" conventually means, executing something with the purpose of obtaining a *value* at the end and for a side-effect-only procedure, one cannot speak of a resulting value, at least not officially. Now, the mechanism here has not only "input" parameters, it also has "output" or result parameters. The syntax restricts to just one, but it's easy to generalize, and actually, in the examples, we make use of multiple parameters. The mechanism for passing back the return value is called *by result*. The call site syntax `call` $p(a, z)$ can be read as $z := p(a)$: the result will be stored in z which is a variable in the scope of the caller.

Example: Fibonacci

$$
\begin{array}{lll}
\texttt{begin} & \texttt{proc}\, \mathit{fib}(\texttt{val}\, z, u, \texttt{res}\, v)\texttt{is}^1 & \\
& \quad \texttt{if} & [z < 3]^2 \\
& \quad \texttt{then} & [v := u + 1]^3 \\
& \quad \texttt{else} & [\texttt{call}\, \mathit{fib}(z - 1, u, v)]^4_5; \\
& & [\texttt{call}\, \mathit{fib}(z - 2, v, v)]^6_7 \\
& \texttt{end}^8; & \\
& [\texttt{call}\, \mathit{fib}(x, 0, y)]^9_{10} & \\
\texttt{end} & &
\end{array}
$$

Actually, the example is not meant to show how Fibonacci is best implemented. It's a rather inefficent way to do so. The reason for its inefficiency are the *two* recursive calls. However, for illustrating challenges with interprocedural analysis, we intend to have an example with *two* calls, and that's an easy example of that.

2.6.2 Extending the semantics and the CFGs

Next comes the adaptation of the definition of the flow graph. To do so, we need to adapt and extend the definitions of flow, block, etc. Actually, it's pretty straightforward for most. The new part deals, obviously, with the procedures; also that is straightforward. The basic trick is that we introduce *new* kinds of edges to deal with the procedures and

calling them. The definition/presentation proceeds (in the slides) in *two* steps, first the call sites, afterwards the procedures themselves. Besides the fact that we need "non-convential" edges (namely connecting 4 nodes), it's still straightforward.

Another thing one can learn from the example is parameter passing, in particular the way the return is passed "by reference". However, the focus in the following will not be how to program with this syntax (it is not intented as concrete surface syntax anyway), but how to analyze it.

Block, labels, etc.

$$\begin{array}{rcl} \mathit{init}([\texttt{call}\, p(a,z)]^{l_c}_{l_r}) & = & l_c \\ \mathit{final}([\texttt{call}\, p(a,z)]^{l_c}_{l_r}) & = & \{l_r\} \\ \mathit{blocks}([\texttt{call}\, p(a,z)]^{l_c}_{l_r}) & = & \{[\texttt{call}\, p(a,z)]^{l_c}_{l_r}\} \\ \mathit{labels}([\texttt{call}\, p(a,z)]^{l_c}_{l_r}) & = & \{l_c, l_r\} \\ \mathit{flow}([\texttt{call}\, p(a,z)]^{l_c}_{l_r}) & = & \{(\mathbf{l_c}, \mathbf{l_n}), (\mathbf{l_x}, \mathbf{l_r})\} \end{array}$$

where $\texttt{proc}\, p(\texttt{val}\, x, \texttt{res}\, y)\texttt{is}^{l_n} S \texttt{end}^{l_x}$ is in D_*.

- two *new* kinds of flows (written slightly different(!)): *calling* and *returning*
- *static* dispatch only

For procedure declarations

$$\begin{array}{rcl} \mathit{init}(p) & = & l_n \\ \mathit{final}(p) & = & \{l_x\} \\ \mathit{blocks}(p) & = & \{\texttt{is}^{l_n}, \texttt{end}^{l_x}\} \cup \mathit{blocks}(S) \\ \mathit{labels}(p) & = & \{l_n, l_x\} \cup \mathit{labels}(S) \\ \mathit{flow}(p) & = & \{(l_n, \mathit{init}(S))\} \cup \mathit{flow}(S) \cup \{(l, l_x) \mid l \in \mathit{final}(S)\} \end{array}$$

"Standard" flow of complete program

not yet interprocedural flow (IF)

$$\begin{array}{rcl} \mathit{init}_* & = & \mathit{init}(S_*) \\ \mathit{final}_* & = & \mathit{final}(S_*) \\ \mathit{blocks}_* & = & \bigcup\{\mathit{blocks}(p) \mid \texttt{proc}\, p(\texttt{val}\, x, \texttt{res}\, y)\texttt{is}^{l_n} S \texttt{end}^{l_x} \in D_*\} \\ & & \cup \mathit{blocks}(S_*) \\ \mathit{labels}_* & = & \bigcup\{\mathit{labels}(p) \mid \texttt{proc}\, p(\texttt{val}\, x, \texttt{res}\, y)\texttt{is}^{l_n} S \texttt{end}^{l_x} \in D_*\} \\ & & \cup \mathit{labels}(S_*) \\ \mathit{flow}_* & = & \bigcup\{\mathit{flow}(p) \mid \texttt{proc}\, p(\texttt{val}\, x, \texttt{res}\, y)\texttt{is}^{l_n} S \texttt{end}^{l_x} \in D_*\} \\ & & \cup \mathit{flow}(S_*) \end{array}$$

As before: S_*: notation for complete program "of interest"

New kind of edges: Interprocedural flow (IF)

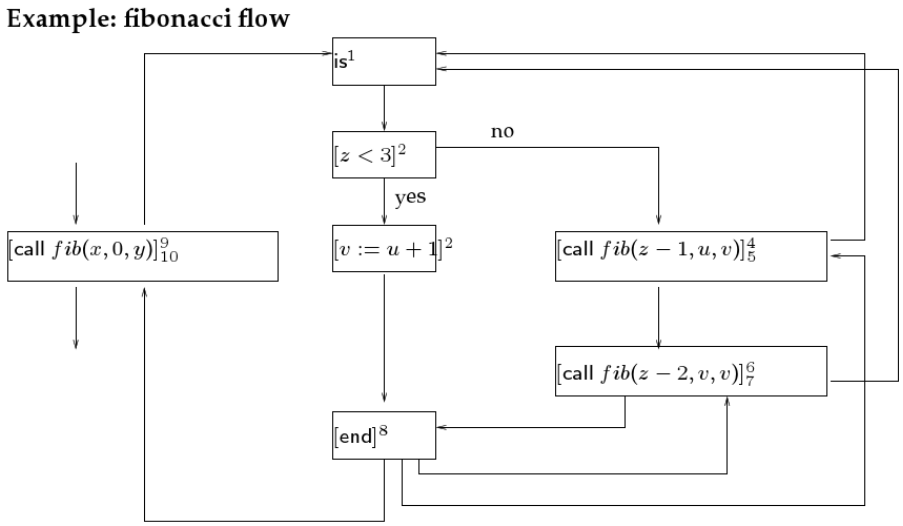
- inter-procedural: from call-site to procedure, and back: $(l_c; l_n)$ and $(l_x; l_r)$.
- more *precise* (= better) capture of flow
- abbreviation: *IF* for *inter-flow*$_*$ or *inter-flow*$_*^R$

IF

$$\textit{inter-flow}_* = \{(l_c, l_n, l_x, l_r) \mid P_* \text{ contains } \begin{array}{l}[\texttt{call}\, p(a,z)]^{l_c}_{l_r} \text{ and} \\ \texttt{proc}(\texttt{val}\, x, \texttt{res}\, y)\texttt{is}^{l_n} S\, \texttt{end}^{l_x}\end{array}\}$$

Now, this is an "edge" between 4 nodes, not 2. This form of generalization is sometimes called a hyper-edge and graphs that allows that are call hyper-graphs. Not that it matters much in our context, how it's called in graph-theoretical terms.

Example: Fibonacci flow



The picture does not show the special interflow edges in some specific manner. In the graphics, they just look like ordinary edges, which correspond to their treatment in the "naive" approach (i.e., non-context-sensitive analysis).

Semantics: stores, locations,...

The operational semantics, as before, will be defined by transitions between *configurations*. Configurations descibed the "situation" a program is currently in. That include the current content of the memory together with the current place where the execution is ("data and control"). That will still be the case here, though we need to do adapations. We start with the data part, i.e., the memory. The big difference is that the memory is no longer *static*, but *dynamic*. We need that insofar that calling a procedure allocates memory for the procedure body. In our setting the allocation is effectively done in a stack based manner.

Assuming that a procedure has a local variable x (for instance a formal parameter), each time we need a new incarnation of that variable, i.e., a new piece of memory where to store that.

One way of doing that is introducing *addresses*, here called locations. Instead of saying that a variable "contains" a value, the picture will be: a variable resides at a particular address in memory, and at that address, its value is store. That is not yet the same as having references, as the user has no way to directly access the addresses. However, such a differentiation between variables and their location in memory is also needed when dealing with heap-allocated data (for instance introducing pointers or reference data like objects etc). Anyway, we are *not* introducing reference data or pointer here, we use the "memory model" to describe procedure calls with a stack-like behavior. In particular, we allow to generate and allocate *new* addresses (namely when we treat procedure calls).

What about *deallocation*? Actually we don't bother. In a stack-based view, there is "push" and "pop" where pushing corresponds to allocation and popping to deallocation. Of course, deallocation in that picture ultimately simply means changing the stack and/or frame pointer, which *marks* the corresponding memory as free but the bytes are still there... They are hopefully just no longer accessible (finding a way to access or manipulate them, thereby destroying the stack abstraction, is a "popular" way of breaching security, usable for security exploits). Anyway, in our fomalization, the addresses allocated via a procedure call are simply no longer accessible (without officially shortening an explicit stack), turning them to "garbage". That results in a stack-discipline.

- not only new *syntax*
- new semantical concept: **local** data!
  - different "incarnations" of a variable $\Rightarrow$ *locations*
  - remember: $\sigma \in \mathbf{State} = \mathbf{Var}_* \to \mathbb{Z}$

Representation of "memory"

$$\begin{array}{lcll} \xi & \in & \mathbf{Loc} & \text{locations} \\ \rho & \in & \mathbf{Env} = \mathbf{Var}_* \to \mathbf{Loc} & \text{environment} \\ \varsigma & \in & \mathbf{Store} = \mathbf{Loc} \to_{\mathit{fin}} \mathbb{Z} & \text{store} \end{array}$$

- $\sigma = \varsigma \circ \rho$: total $\Rightarrow$ $\mathit{ran}(\rho) \subseteq \mathit{dom}(\varsigma)$
- top-level environment: ρ_*: all var's are mapped to **unique locations** (no **aliasing** !!!!)

Explanations The notation $\to_{\mathit{fin}}$ represents *finite* partial functions. For course, also the environment ρ is a finite function, simply because the domain $\mathbf{Var}_*$ is finite. In contrast, the set of potential addresses ξ is infinite, though at any concrete configuration, only a finite amount of memory is allocated.

The conditions mentioned on the slides are well-formedness condition. For instance prevening addresses which have not value stored etc.

SOS steps

Now to the semantics as transitions between *configurations*. We focus on the transitions for the new concepts, i.e., calls and returns. Before we do that, we need to generalize the notion of *configuration*. Part of it is achieved already by introducing locations or addresses and "splitting" the state into two separate mappings (the store and the environment). The environment is responsible to associate addresses to variables. It will also be used to realize the "stack-like" arrangement of accessible address space. There is different ways to do that. The way it's done here is that the envirenment part is "baked into the syntax part" of the configuration. That's one way of achieving the stack-like behavior we need. Currently, however, the syntax is not prepared for that, there's not syntax that can me (mis-)used for that. What we are doing then is simple *add* additional syntax. This syntax is not for the user, i.e., it won't show up in the program before it starts to run. It's ony generate on-the-fly, at run-time with the purpose of capturing some semantics concepts of behavior (in this case calls and returns and the corresponding stack-like run-time environment). Such semantics "semantic trick" are actually quite common. For the simply while language without procedure, we did not see such *run-time syntax*, basically since the language is so trival. In many other situations, one works with such run-syntax. It's a way to formalize *run-time environments*, i.e., for instance in connection with memory treatment, in an abstract way and in a syntax directed way: the spirit of SOS (structural operational semantics) is: the structure of the syntax steer the execution, and if that calls for new syntax, why not.

The new construct (in one of the following slides) is known as bind-construct

- steps *relative* to *environment* ρ

$$\rho \vdash \langle S, \varsigma \rangle \to \langle \acute{S}, \acute{\varsigma} \rangle \tag{2.12}$$

or

$$\rho \vdash \langle S, \varsigma \rangle \to \acute{\varsigma} \tag{2.13}$$

- old rules needs to be adapted
- "global" environment ρ_* (for global vars)

The adaptation of the old rules is straightworward. Nothing basically changes, except that state σ is now split into two components and written in a strange way, where one uses $\vdash$. *Writing* it like that is of course not central, configuration is now a triple, instead of a tuple, at least in the non-terminated form. For the previous statement. In the rules covering old syntax, the environment part never changes, though of course executing an assignment updates the store part (in the analogous way that an assignment in the previous setting updated the state σ). That reflects the fact that executing code inside one current procedure body (more generally, executing code inside one current scope), never changes the association of variables to addresses. That binding of variables to addresses, here represented as evironment ρ, is *fixed* per scope.

More interesting are the rules dealing with calls and return. That's where code is execited in a different scope: body of the call is evaluated in a *different* context or environment than the call. That's where the bind construct comes in (see next).

Before we go to the concrete rules: the transition in equation (2.12) captures that the program with the store executes one step *inside a given environement* ρ. And the environment does not change! How can we arrange that conceptually the body of a procedure is supposedly executed in a *different* environment, one that uses local state space for locally scoped variables? The form of the rules seems not able to capture that since ρ is unchanged. That's a proper observation, and that motivates the addition bind-construct: the bind-contruct allows to add a "local" environment baked into the syntax, that's what the bind-syntax is introduced for.

Call-rule

This and the following slide show the treatment of calls and returns (including the treatment of the bind-construct). Altogether, the behavior is covered by 3 rules. The first one deals with calls, and it introduces the new run-time construct. It can be seen as realizing the parameter passing, actually the parameter passing from caller to callee, not yet passing back the return value (but preparing for it by allocating some memory for it). Calling a procedure p looks up the definition of that procedure in D_* (and we assure it can be found uniquely there). Since the procedure is defined with two formal parameters, we need two places where the parameters can be stored resp. passed (by value resp. by result). Those are the fresh locations ξ_1 and ξ_2 in the rule.

The parameter passing is done in the last line of the premise, assigning values to the new addresses. The first one for ξ_1 represents the call-by-value mechanism for the arguments (here only one for simplicity). The address ξ_2 for the result parameter is already allocated, but of course, the return value is not yet computed. Still, the address needs to have a value already now, and thus a random value v is picked; it is stored simply to avoid "undefined" values. It assures the assumption mentioned earlier that the state σ has to be a *total* function. This random choice can be also seen as reflecting the fact that when reserving some random memory space for the eventual return value, there will be some "old" bit patters at the corresponding address(es) which can be seen as a random value; there is no such thing as a "empty" memory slot. Of course here, in the formalization, it's more like one possible way to specify the behavior.

The conclusion of the rule shows how a call is replaced by a bind-statement. The arrangement for the return of the result involves two variables, the caller-site variable z, and the formal result parameter y at the site of the callee. The last thing that's done after the method body is the assignment $z := y$, copying back the results (in effect, that makes it behave like that z from the caller site looks as if passed by reference, though the caller knows it unter y and copies result back only at the end, in the `then`-part of the bind-construct, not sharing the access during the execution. As we will see in the rules for the bind-construct, its body will be executed in the environment as mentioned in the construct, in the rule $\rho_*[x \mapsto \xi_1][y \mapsto \xi_2]$. The environment ρ_* is the "top-level" environment. It contains the addresses for the global or static variables. The fact that the body is executed in the environment $\rho_*[x \mapsto \xi_1][y \mapsto \xi_2]$ highlights also the restrictions of the language: the procedures cannot be nested. Besides that the language does not bother with procedure local variables (beyond the formal parameters).

Comparison with stack based run-time environments in compilers The discussion here is more for those who followed the compiler lecture. It can be skipped, if wished (it's not pensum). The section here discusses the formalization in comparison with how a compiler arranges its run-time envoronment, expecially stack frames.

In this lecture, there had been some remark and feedback from the audience about what to expect from a semantics. It was mentioned that it should for instance fix the order on which the arguments are passed, and the shape of the data structure where they are stored. In compiler terms, the data structure would be called *stack frame* or *activation record* and the conventions or specification about how to do the stepwise passing of the arguments (like, in which order) is known as *calling conventions*.

The call-rule here deals with that in a more *abstract* form. For example we don't pass arguments in some particular order, like fixing that the function arguments are to be stored "in reverse order" or something. In line with that: the locations, respresenting addresses, are not even *ordered*. Besides that, since the language does not support side-effects in procedure arguments, the order does not really matter (though of course a compiler will arrange the copying in one particular order).

Let's also discuss further aspects of stack frames as can be found concretely in run-time environments of a compiler. One piece of information that belongs to a stack frame is the *return* addresses of the caller. That's not done here. The operational semantics does not make use of program counters, code addresses, jumps. or similar to represent the progression of control during execution. It's a *structural* operational semantics making use of the structure of the program, i.e., its abstract syntax. In case of a situation of a call to p followed by the rest of a program

$$\texttt{call}\, p(a, z); S \ ,$$

the execution proceeds to

$$S_p; S \ .$$

In the post-configuration, S_p is the body of the procedure p, "copied in". The configuration here is simplified for clarity, omitting in particular the treatment of environments ρ. Anyway, the step shows how calls are treated, by copying in the code of the body in front of the rest S. The subsequent S simply represents the point to return to, in that way achiving the same behavior without bothering introducing code addresses.

Besides space for parameter, local data, and return addresses, there are other slots often in standard stack frames or activation records. What is needed depends on the complexity of the language. As mentioned earlier, the language here is rather restricted: first-order procedures, no nesting of procedures, no local variables (besides formal parameters). As far as scoping is concerned, there are basically two levels of variables: global ones and local ones (namely formal parameters). This two-layered scheme would still apply if we allowed local variable declaration in procedures. Doing so would actually not be so hard, especially if we would make a version, we local variables had a lifetime during all the procedure body or for the rest of the procedure body, from their declaration onwards. A bit more complex would be nested scopes (like blocks) inside a procedure, but not radically so.

Disallowing nesting of procedures means: the value for each variable is *either* found in the static memory (i.e., via ρ_*) *or* in the local stack frame, here represented by

$\rho_*[x \mapsto \xi_1][y \mapsto \xi_2]$. In particular, without nested procedures, the value is not found in an "earlier" or older stack frame. In standard run-time environments for *nested* first-order procedures, where there, unlike here, variables exists that are neither global not local but declared in a surrounding procedure, such earlier stack frames can be located by what is called a *static link* or also *access link*, This is basically a pointer in one slot of the stack frame. Our semantics does not need some analogue concept. The semantics here has no need for that. Assuming two different different environments ρ_1 and ρ_2 at some point in the execution, one for the caller one for the callee, ρ_1 and ρ_2 are *unrelated* except of the fact that they share the same bindings for the global variables. I.e., both ρ_1 and ρ_2 extend ρ_* but the callee has no access to bindings in ρ_2 (or other "ealier" stack frames).

Finally, in standard stack frames, there is also the notion of a *dynamic link*, which allows the run-time enviroment to pop-off a stack frame (since stack frames are of difference size). Conceptually, we need to deal with that, but it's done implicitly in the formalization and not with some pointers, in the same way that we conceptually return from the body of procedure call, but the formalization does not need to remember explicit return addresses (as explained above).

$$\frac{\begin{array}{c}\texttt{proc}\, p(\texttt{val}\, x, \texttt{res}\, y)\texttt{is}^{l_n}\ S\ \texttt{end}^{l_x} \in D_* \\ \xi_1, \xi_2 \notin dom(\varsigma) \qquad v \in \mathbb{Z} \\ \acute{\varsigma} = \varsigma[\xi_1 \mapsto [a]^{\mathcal{A}}_{\varsigma \circ \rho}][\xi_2 \mapsto v]\end{array}}{\rho \vdash \langle [\texttt{call}\, p(a,z)]^{l_c}_{l_r}, \varsigma\rangle \to \langle \textsf{bind}\ \rho_*[x \mapsto \xi_1][y \mapsto \xi_2]\ \texttt{in}\, S\, \texttt{then}\, z := y, \acute{\varsigma}\rangle}\ \textsc{Call}$$

Bind-construct

The bind-construct is covered by two rules. The construct is introduces when executing a call-statement and thus the rules for bind describe how to execute the body of a procedure after being called. There are two cases: during execution, and "termination" which in this case means, finishing with the body; in the larger context of being called, that means returning to the caller. When executing the body, it's done as for standard statments (covered by the other rules). Of course, that includes further calls. If that occurs, that leads to another execution of the Call-rule, which in turn leads to execute the second procedure body in another environment. Note that the rule $\textsc{Bind}_1$ is a *recursive* derivation rule ($\textsc{Bind}_2$, as well, and actually as the prior rules covering sequential composition): the step of a bind construct is explained by recursively determining the step of the body. When a code calls a procedure, that in turn calls a further procedure, and that in turn a further one ..., that implicitly (in the derivation tree) leads to a number of environments ρ in a nested fashion in the derivation tree, which is an abstract representation of the stack frames.

$$\frac{\acute{\rho} \vdash \langle S, \varsigma\rangle \to \langle \acute{S}, \acute{\varsigma}\rangle}{\rho \vdash \langle \textsf{bind}\ \acute{\rho}\ \texttt{in}\, S\, \texttt{then}\, z := y, \varsigma\rangle \to \langle \textsf{bind}\ \acute{\rho}\ \texttt{in}\, \acute{S}\, \texttt{then}\, z := y, \acute{\varsigma}\rangle}\ \textsc{Bind}_1$$

$$\frac{\acute{\rho} \vdash \langle S, \varsigma\rangle \to \acute{\varsigma}}{\rho \vdash \langle \textsf{bind}\ \acute{\rho}\ \texttt{in}\, S\, \texttt{then}\, z := y, \varsigma\rangle \to \acute{\varsigma}[\rho(z) \mapsto \acute{\varsigma}(\acute{\rho}(y))]}\ \textsc{Bind}_2$$

- bind-syntax: "runtime syntax"

⇒ formulation of correctness must be adapted, too (Chap. 3)[1]

2.6.3 Naive analysis (non-context-sensitive)

When extending the analysis to cover procedures, we start first with a *naive* formulation, basically treating the control flow transfer between caller and callee as if they were ordinary control flow edges. We had, in the definition of the interprocedural control flow graph, given the edges between a caller and callee (corresponding to calls resp. returns) a special status. The naive approach simply ignores that status and treats them as if they were standard edges, and we know already how to treat those. So one might not even call it interpocedural analysis, as it simple ignores the additional challenges for those. Analyses that take care of the special nature of procedure calls and returns are known as *context-sensitive* analyses. In contrast, the "naive" treatment here is called a non-context-sensitive analysis (not context-free...).

Before addressing "real" interprocedural analyses, we will (after the naive approach) introduce the notion of a *path* through a CFG.

Transfer function: Naive formulation

- first attempt
- assumptions:
  - for each *proc. call:* 2 transfer functions: f_{l_c} (call) and f_{l_r} (return)
  - for each *proc. definition:* 2 transfer functions: f_{l_n} (enter) and f_{l_x} (exit)
- given: *mon. framework* $(L, \mathcal{F}, F, E, \iota, f)$

Naive

- treat IF edges $(l_c; l_n)$ and $(l_x; l_r)$ as **ordinary** flow edges (l_1, l_2)
- *ignore* parameter passing: *transfer* functions for proc. calls and proc definitions are *identity*

Equation system ("naive" version)

$$\begin{aligned} A_\bullet(l) &= f_l(A_\circ(l)) \\ A_\circ(l) &= \bigsqcup\{A_\bullet(l') \mid (l', l) \in F \text{ or } (l'; l) \in F\} \sqcup \iota_E^l \end{aligned}$$

with

$$\iota_E^l = \begin{cases} \iota & \text{if } l \in E \\ \bot & \text{if } l \notin E \end{cases}$$

- analysis: *safe*
- unnecessarily imprecise, too abstract

[1] Not covered in the lecture.

As a side remark: the equational system here is *without* the assumption of isolated entries/exits (unlike what we did before). It makes not a big difference either way.

More important is to reflect on the following: the slide mentions that the analysis is *unnessesarily* imprecise or approximating. That the analysis based on a control-flow abstraction is approximative is clear, data flow analysis is necessarily approximative. The only exceptions to that may be simplistic situations, like straight-line code. In the language without procedures the reason of the imprecision was that the analyses abstract away from the influence of the data on the control flow: in a branching situation, originating from conditions and while-loops, the analysis makes not attempt to figure out which one is taken, so the flow explores boths. Of course, in a program, even for a program with input, it may not be the case, that both actually both branches are taken. When tackling *open* programs (like programs or procedures with input), it may also be the case that no matter what the input, some branches will never be taken, under no circumstances.

A situation like that may be seen as a chance for optimization, removing *dead code*. Or actually, it may be flagged as a suspicions situation: why did the programmer program a condition, whenone branch is never taken. It could be a sign of a bug, perhaps the loop condition or branch condition is slightly off.

One can indeed use data flow analysis similar to the ones we covered to try figure out such situation. But that's not the point here. The point is: if we abstract away from the influence of the data on the control flow (as data flow analysis typically does), then all decision points in the program could go either way, resp. all branches can be taken.

More generally, all *paths* through the cfg and starting from the initial node, are *possible* in that sense. That concrete data for concrete run makes some (actually most) paths impossible is secondary, as data flow analysis is its basic form ignores the influence of data. Besides that, the CFG of the while language is a connected graph, even more, all nodes in the graph are reachable from the initial node. And as a consequence, all paths starting from the initial node are, in principle possible, given that data abstraction.

As a side remark: when adding procedures, some of the latter statements about the cfps are no longer guaranteed, strickly speaking: the graph may no longer be connected and as a consequence, not all nodes may be reachable in the CFG from the initial node. That happens if one defines a procedure, but it's never mentioned in a call-statement in the program. That, though, is an silly anomaly. One could simply assume that this does not occur, and it's easy to check if all procedures are called at least one. Actually, it's slighty more complicated than checking if all procedures are called somewhere. One would have to rule out a situation like the following: two procedures call each other mutually recursively, but the main program does not call either of those. In that and similar situations, the graph is still unconnected and not all nodes are reachable from the initial node. BTW: A graph which shows which procedures call which other procedures is known as *call graph*. It's easy to determine in our setting with static dispatch. But anyway, it's easy enough to check, so let's simply assume that there are no superfluous procedures (resp. that the call-graph is connected).

But back to the remark about *unessessary* imprecision. In the setting with procedures, even if the call-graph is connected, not all paths through the CFG are actually possible (assuming data abstraction). The reason for that is that, no matter the data, the caller

necessarily return to the point of being called. If there are two or more places where a procedure is called, then ignoring that coupling between call-edge and return-edge in a path, as the context-insensitive analysis does, leads to an imprecision *on-top of that caused by abstracting data*. In that sense it "unessessary" or and additional factor of imprecision. Context-sensitive analyses try to tackle or mitigate that additional imprecision.

Path (in)sensivity Before we do the we may further reflect on that imprecision. The data flow analyses, the monotone framework, so far was kind of *memoryless*. In the context of static analysis, a better name would be *path-insensitive*. When analysis a program based on the CFG, and facing a choice point, the analysis makes a choice, and immedate forgets the choice, and proceeds. For instance, if there is a choice based on the condition $x \geq 0$, then, taking the true-branch for further exploration would make clear, that when following that branch for exploration, holds. In contrast when following the other branch, $x < 0$ can be assumed. If course, our data flow analysis don't operate with concrete data like integers. Also they don't remember decisions that are taken, they analyse both branches, but the don't exclude branches based on things that happened in the past nor do they remember anything about a choice that could influence decisions in the future. They don't even have a concept of path. That is visible in the way the analyses are *solved*, but the worklist algorithm or even clearer in the chaotic iteration.

Following an edge corresponding to $x \geq 0$ does not mean one continues exploring that branch. *follows* down that particular *path*. The chaotic iteration may well chose a different edge to explore and repair, "jumping around" in the program at will. It may be profitable, for efficiency of the analysis, to follow a path, but it's not necessary; the data flow problem and its analysis is not based on the notion of path, it's path insensitive.

We used remembering $x \geq 0$ resp. $x < 0$ for illustrating of path-sensitivity, resp. not remembering makes the data flow path insensitive. Now, our analyses don't even operate on concrete data values. With procedures, however, we can remember something else. When a procedure (as in the fibonacci example) is called from different places, then the return node has different successor nodes; all the places it could return to. As explained, when confronted with a choice point, the plain data flow analysis makes the decision on the spot, "memoryless" or path-insensitive, resp. explores all alternatives indisciminately (without being influenced by the past and without remebering anything to influence decisions of the future). To let the flow analysis of the callee's body to return to the place where the procedure had been called, it helps to have remember that place (or "information about that place").

That will be an approach the lecture will pursue: extending

Path sensitivity and context-sensitivity and others In the above discussion, we used as illustation a conditional with $x \geq 0$ as branch condition; in that context we mentioned the terminology of *path sensitivity*. Classical data flow analysis is not path sensitive. It can be made more precise (becoming path-sensitive), if one extended the tracked information in the analysis and have it *remember* information about that choice. Similar remarks apply to our setting here: one can make interprocedural analysis more precise, by remembering information about where a procedure had been called. This information can make the place to return to more precise. As far as terminology is concerned: the first

situation, concerning branching is about path-insensitivity vs. path-sensitivity. The second one concerning procedure calls is called context-insensitivity vs. context-sensitivity of interprocedural analysis.

So even though one distinguishes between path-sensitive extensions and context-sensitive extensions to basic data flow analyses, the underlying theme is the same: the underlying analysis can be made more precise by additional information being "tracked" (i.e., in our setting added to the lattice on which the data flow equations operate). One could view path-sensitive analysis also as "context-sensitive" in the following sense: by remembering information about a particular choice, the corresponding branch following that choice is analysed in or with that "context". In the example, in the "context" $x \geq 0$ in the true-branch, and in the context $x < 0$ in the alternative. That's in the same spirit of context-senstive interprocedural analysis, where call-site information is added and the procedure body is analized with or in this context. But as said, despite those general similarities, one calls improving on the analysis of the branching-situation in that way as path-sensitive, not context-sensitive. And, indeed, the discussion should not be taken as the message, that improving the analysis of conditional branches in a path-senstivie way is *the same* as context-sensitive interprocedural analysis. It's two enhancements in the same spririt: add more more information and you get more precise (but the analysis with take more time).

Now we have a general feeling what needs to be done. Before continuing with technical details, let's reflect on the notion of *paths* in our context and shed more light about some wordings we used in the discussion, namely that the analyses need to be enhanced by addition information, for instance about the call-site, or that the analysis has to "remember" such information, for instance about the call site. We want to clarify that better, because "information about the call-site" is rather wishy-washy. We will cover two different classes of information about the call site, roughly *control-information* and *data-information* (see later).

But first about the notion of *paths*. We want to discuss that, because we just mentioned path-sensitive analyses. Secondly, when covering context-sensitive analyses, the notion of paths will play a role (and we mentioned that already before a bit, see also Section 2.6.4 below. We are dealing with specific graphs, control-flow graphs, which may obay some additional condition, like we assume that they are connected and all nodes are reachable etc. But still they are directed graphs with nodes and edges. In general terms a *path* through some graphs is a way through the graph following the edges one after the other. Technically, it may be defined as a sequences of nodes (each succeeding pair connected by edges), or a sequence of edges ("connected" by nodes) or also a sequence of node-edge-node-edge etc. That depends a bit on whom you ask. Now, in our case, we focus on paths which start in the initial node of the CFG (at least for forward analyses, for backward analyses one follows edges backwardly, start in the final node(s)).

That very conventional notion of paths will be defined in Section 2.6.4. We know that data flow problems in the monotone framework do have unique best solutions. We also know that so far the monote framework does not operate on the notion of paths. An alternative problem is to ask: given a node in a graph and some data flow problem: what it the smallest value in the lattice at that location, considering all the different ways I can reach that location? That different from the standard data flow question, since the data

flow equations don't "follow the edges", i.e., don't explore paths (remember the chaotic iteration). The follow-the-paths solution to a data-flow problem is called **meet-over-all-path (MOP)** or *join-over-all-paths*, depending on whether we do a must or a may analysis). Anyway, asking that question is more complex! In a typical program, with loops, there are **infinitely** many different paths by which nodes can be reached; some nodes of course still can be reached by finitely many paths (nodes at the "beginning" of a program resp. of a control-flow grahp, before encountering any loop).

To exlore infinitely many paths one by one is not a algorithmic option. One can see it more like a definition: what's the smallest solution if one would follow all paths individially (though it would take an infinite amount of time). It's like the ultimate path sensitive analysis (working with actual paths, though still using data abstraction, so there are probably (much) more paths explored than are executed when running the actualy program with concrete data. If analysing a *closed* program (no input), there is actually ony one path, not program without any input are not very common... To be precise, there is one *complete* path, which is finite if the program terminats, plus many prefixes.

Now, what's the relationship between the **MFP** and the **MOP** solution of a data flow problem? The good news is: there are the same (under mild assumptions): both are the same if the analysis is *distributive*. That is the case for all the 4 analyses we have discussed, and it's the case for all analyses with transfer functions given by kill and generate. That is good news indeed, we don't need to worry to loose precision by using a worklist algo which ignores the notion of paths completely.

How about *interprocedural* analysis? As already mentioned, if we ignore the special nature of the call and returns in that the callee should always return to the point where it had been called, we loose precision in that the analysis explores path that in reality would never occur. To remedy this, one can refine the problem of meet-over-*all*-paths to meet-over-all-*valid*-paths (**MVP**). That are less paths, but of course it will typically still be an infinite set of paths. So it's still not an approach one can directly follow in practice, but it shows which in direction to go when extending the naive, context-insensitive approach.

Path sensitivity follows paths in a graph? Let's revisit the notion of path sensitivity after having discussed the notion of path a bit (details will follow). We have not technically introduced path-sensitivity; we illustrated that with a condition like $x \geq 0$. Does path sensitivity means, the analysis operates with paths and "follows paths" in that it explores them? Well, yes and no. It's more like: a path-insensitive analysis, taking a decision like in a conditional has *no influence* in the further analysis, further down the path. If some piece of information is remembered that will have an influnece, then it's path sensitive. Thus, path sensitive means not necessarilty, that the piece of information which is recorded is the whole path (as is done in the *MOP* or *MVP* setting which are more of theoretical intest). Paths may grow unboundedly and there are infinitely many of them overall, so that makes it impractical. Instead, some relevant information is stored. In the above example, the fact that $x \leq 0$

symbolic execution [continue here]

Data-flow analysis is path insensitive and context-insensitive, right? [continue here]

2.6.4 Taking paths into account

It corresponds to the standard notion of a path through a graph. Note that so far that notion did not play a role. The data flow analysis, for instance in the form of a worklist algorithm, is concerned with solving equation, piecewise repairing violations of the equations. Each constraints correspond to an *edge* in the graph (or perhaps a set of edges in the equational approach). Anyway, repairing an constraint corresponds to treating an edge (or a set). But the solving procedure for the constraints is not required to follow the edges one after the other, like following paths through the control flow graph. The concrete semantics and execution will do that, the analysis may choose other traversal strategies. Before we look a bit further in the [continue here]

Paths

Now, finally, the definition of paths, though, as indicated, it basically a standard notion of paths through a graph. Transfer functions are connected to nodes of a CFG for a given program, resp. the edges. Anyway, now that we now that a paths here is a sequence of nodes, following the edges through the graph, we can connect a transfer function to the whole paths. It's simply the composition of the individual transfer functions.

As side remark: remember the definition of monotone frameworks. There, we required a general set $\mathcal{F}$ of transfer functions. It is required that this set of functions is closed under compoition (and includes the identity function).

- remember: "MFP"
- historically: MOP stands for **meet over all paths**
- here: dually mostly *joins*
- 2 "versions" of a path:
  - path to **entry** of a block: blocks traversed from the "extremal block" of the program, but **not** including it
  - path to **exit** of a block

Paths

$$\begin{array}{lcl} path_\circ(l) & = & \{[l_1, \ldots \mathbf{l_{n-1}}] \mid l_i \rightarrow_{flow} l_{i+1} \wedge l_n = l \wedge l_1 \in E\} \\ path_\bullet(l) & = & \{[l_1, \ldots \mathbf{l_n}] \mid l_i \rightarrow_{flow} l_{i+1} \wedge l_n = l \wedge l_1 \in E\} \end{array}$$

- transfer function for paths $\vec{l}$

$$f_{\vec{l}} = f_{l_n} \circ \ldots f_{l_1} \circ id$$

Meet over all paths

- paths:
  - forward: paths from init block to entry of a block
  - backwards: paths from exits of a block to a final block
- two versions for the MOP solution (for given l):
  - up-to but not including l

– up-to including l

MOP

$$MOP_\circ(l) = \bigsqcup\{f_{\vec{l}}(\iota) \mid \vec{l} \in path_\circ(l)\}$$
$$MOP_\bullet(l) = \bigsqcup\{f_{\vec{l}}(\iota) \mid \vec{l} \in path_\bullet(l)\}$$

The definition of meet or join over all paths is straighforward: take all paths, for each take the corresponding transfer function and build the meet (or the joint), depending on what interests you in a particular case. In the definition, there is a small refinment: given a paths, we are explicit about whether we want the transfer function including the last node or not. In the forward case, that means, reaching the exit of the last node, or only until reaching the entry of the last node (thereby ignoring the transfer function of the last node).

Problematic is that there are infinitely many paths, in general. That makes the MOP problem undecidable in general. It is perhaps also intuitively clear that taking paths into account, something that the plain random constraint solving does not do, can only increase the precision. Additionally it can be proven, that MOP and MFP yield the *same* solution, in case the analysis is *distributive* which is often the case. In particular, it's the case for the four classic data-flow analyses we covered.

MOP vs. MFP

- MOP: can be undecidable
  – MFP *approximates* MOP ("$MFP \sqsupseteq MOP$")

Lemma 2.6.1.

$$MFP_\circ \sqsupseteq MOP_\circ \text{ and } MFP_\bullet \sqsupseteq MOP_\bullet \tag{2.14}$$

In case of a distributive framework

$$MFP_\circ = MOP_\circ \text{ and } MFP_\bullet = MOP_\bullet \tag{2.15}$$

If the transfer function is given by kill and generate as shown, the analysis is distributive.

The "all-path" setting was done without taking into account procedures. In particular, in case there are procedures, the calls and returns are not treated specifically. As discussed earlier, with procedures, not all paths make sense. Without procedures, one can say, modulo the fact that we have abstracted the data, all paths make sense.

Next we simply define which paths "make sense" –they will be called *valid*– and then build the meet (or join) only over those. That can only make the analysis more precise, though it does not help with the fact that there are still infinitely many paths to consider, in general.

How to define valid paths? Well, it's about fixing that a return from a procedure returns to the place where the procedure is called. Calls and returns work in a *nested* manner; executing a return always has to return to the last open call.

That kind of nesting can be described well by context-free grammars. The possible finite paths can be seen as some *language*, finite sequences of letters (where the letters are the labels or nodes). Without procedures, the corresponding language is *regular*, with the control-flow graph being considered a "finite-state automaton"). That in language-theoretic terms, languages are typically sequences of letters attached to the edges (of a FSA) and not sequences of nodes is a not-so-relevant detail. Now, valid paths for procedures no longer form a regular language. Instead, they can be captured a *context-free language*.

Note: for the terminology, we are heading towards context-sensitive program analysis. The fact that we are about to characterize the valid paths via a context-free grammar may be confusing, but in one case we talk about the language of valie path, in the other about whether an analysis takes the call-site into account.

Context-free languages are typically given by context-free grammars, which is effectively what we do in a few slide. Though we don't use notations like (E)BNF for that. Actually before we done valid paths, which is what we are interested in, we define as stepping stone something slighty simpler but closely related: *complete* paths.

The concepts hang together as follows: in both cases, returns have to return the the call site. A comple path is one where there are not unanswered calls, i.e., each call is followed by a matching return. A valid paths is a prefix of that concept, i.e., there can be unanswered.

That can be illustrated by some prototypical example of context-free languages, namely those of well-balanced parenthesis or brackes. Let's assume that we have three forms for parentheses (), {} and []. Then the following string is perhaps not "well-balanced": not all opening parentheses are closed later, but it can be extended to a well-balanced word:

$$({[()[]}[$$

In our terminolog, that would be a *valid* word but not *complete* (if we interpret opening parentheses calls and closing parentheses as returns). The analogy actually is pretty accurate. The different "forms" of parantheses are simply the different call sites.

This analogy makes also perhaps plausible that complete paths, corresponding to well-balanced strings are easier to define than valid paths. In case of three parentheses, the standard grammar in BNF looks as follows:

$$S ::= (S) \mid \{S\} \mid [S] \mid S\ S \mid \epsilon\ .$$

The definition of complete and valid paths will be likewise be define by a context-free grammar. The non-terminal (in case of the complete paths) are CP_{l_1,l_2}, representing all complete paths from l_1 to l_2, and the terminals of the grammar are the labels l of a given control-flow graph.

MVP

- take calls and returns (IF) serious
- restrict attention to valid ("possible") paths

⇒ capture the nesting structure

- from MOP to **MVP**: "meet over all **valid** paths"
- *complete* path:
  - appropriate call-nesting
  - all calls are answered

Complete paths

- given $P_* = \texttt{begin}\, D_*\ S_*\, \texttt{end}$
- CP_{l_1,l_2}: complete paths from l_1 to l_2
- generated by the following *productions* (l's are the terminals) (we assume forward analysis here)
- basically a **context-free grammar**

$$\frac{}{CP_{l,l} \longrightarrow l}$$

$$\frac{(l_1, l_2) \in F}{CP_{l_1,l_3} \longrightarrow l_1, CP_{l_2,l_3}}$$

$$\frac{(l_c, l_n, l_x, l_r) \in IF}{CP_{l_c,l} \longrightarrow l_c, CP_{l_n,l_x}, CP_{l_r,l}}$$

The notion of *complete* path is rather straightforward. It informally says that each call is answered by one corresponding return, and also that each return is matched by one corresponding call. It directly corresponds to the prototypical context-free parenthetic languages, except that we have an arbitrary number of different "parentheses" namely the different calls. The calls are not being identified by the name of the function being called, but by the call-sites and the identity of the function being called, more precisely by the two labels at the call site plus the two labels of the entry and the exit of the procedure. That is visible in the third rule.

The definition is given in a rule-like manner. They are not really like derivation *rules*, though. It's more like a family of grammar productions, namely for each label (first rule), for pairs of labels (second rule), resp. quadrupel of labels (last rule). As the premises of the last two rules show, not for all tuples or all quadruples, of course, only those as given by the control flow graph, in particular taking care of the inter-procedural flow in the last rule.

The interpretation is as follows. There is only one complete path from a label to itself, that's the trivial path. The second rule just splits off one label on the left (one could do

also differently). Also in the third rule, there is a split-off of the first step. A *terminating* execution will have a complete path. There are only a finite number of productions.

As a side remark: being a complete path, in some way, is not a *safety* property, whereas being a valid path, is.

Example: Fibonacci

- concrete grammar for fibonacci program:

$$
\begin{array}{rcl}
CP_{9,10} & \longrightarrow & 9, CP_{1,8}, CP_{10,10} \\
CP_{10,10} & \longrightarrow & 10 \\
CP_{1,8} & \longrightarrow & 1, CP_{2,8} \\
CP_{2,8} & \longrightarrow & 2, CP_{3,8} \\
CP_{2,8} & \longrightarrow & 2, CP_{4,8} \\
CP_{3,8} & \longrightarrow & 3, CP_{8,8} \\
CP_{8,8} & \longrightarrow & 8 \\
CP_{4,8} & \longrightarrow & 4, CP_{1,8}, CP_{5,8} \\
CP_{5,8} & \longrightarrow & 5, CP_{6,8} \\
CP_{6,8} & \longrightarrow & 6, CP_{1,8}, CP_{7,8} \\
CP_{7,8} & \longrightarrow & 7, CP_{8,8}
\end{array}
$$

Valid paths (context-free grammar)

Valid path (generated from non-terminal VP_*):

- start at extremal node (E),
- all proc *exits* have matching *entries*

$$
\frac{l_1 \in E \qquad l_2 \in \mathbf{Lab}_*}{VP_* \longrightarrow VP_{l_1,l_2}} \qquad \frac{}{VP_{l,l} \longrightarrow l}
$$

$$
\frac{(l_1, l_2) \in F}{VP_{l_1,l_3} \longrightarrow l_1, VP_{l_2,l_3}}
$$

$$
\frac{(l_c, l_n, l_x, l_r) \in IF}{VP_{l_c,l} \longrightarrow l_c, CP_{l_n,l_x}, VP_{l_r,l}} \qquad \frac{(l_c, l_n, l_x, l_r) \in IF}{VP_{l_c,l} \longrightarrow l_c, VP_{l_n,l}}
$$

The grammar for valid paths is slightly more complex than the one for complete paths. There is an *easy* explanation what a valid path is: a valid path is a *prefix* of a complete path. One could leave it at that. The definition shows, basically, that also that property is a context-free property (namely by giving the corresponding (family of) productions.

MVP

- adapt the definition of paths

$$\begin{array}{rcl} vpath_\circ(l) & = & \{[l_1, \ldots \mathbf{l_{n-1}}] \mid l_n = l \wedge [l_1, \ldots, l_n] \text{ valid}\} \\ vpath_\bullet(l) & = & \{[l_1, \ldots \mathbf{l_n}] \mid l_n = l \wedge [l_1, \ldots, l_n] \text{ valid}\} \end{array}$$

- **MVP** solution:

$$\begin{array}{rcl} MVP_\circ(l) & = & \bigsqcup\{f_{\vec{l}}(\iota) \mid \vec{l} \in vpath_\circ(l)\} \\ MVP_\bullet(l) & = & \bigsqcup\{f_{\vec{l}}(\iota) \mid \vec{l} \in vpath_\bullet(l)\} \end{array}$$

- but still: "meets over paths" is *impractical*

Fixpoint calculations next: how to reconcile the path approach with MFP

2.6.5 Context-sensitive analysis

Next we finnally address the core technical part for interprocedural analysis, which is known as *context-sensitive* analysis. We generally talked about that, namely that is's about adding information with the purpose of making the naive analysis more precise. In particular, to allow to disambiguate situations where a procedure is called more than one (like in the fibonacci example). In the extreme case, the disambiguation could be 100%. That's what we did with the MVP description. Valid paths are defined as those where returns go back exactly to the place where they should. In that way, that disambiguates the situations completely.

However, we MVP was defined by making the meet or join over infinitely many paths in general, and that makes it a non-practical solution. It should not even be seen as *solving* the interprocedural analysis problem, it just specifies the theoretic optimum of precision (under data abstraction), which is practically not obtainable.

Indepedently from that, for interprocedural analysis, making a set of paths first and then building the meet or join over it is also a bad algorithmic design. The chaotic iteration or the work-list algo did not "follow" some paths, they were solving constraints, following strategies as they felt fit (or doing it randomly). What we therefore, is extending that approach, i.e., extending the definition of the data flow problem and thereby also extending or rather reusing the algorithmic approaches for that, like the worklist algorithm. This will be called *embellished* monotone framework, i.e. the old framework with additional information addted (to the lattice). It's a general them of course: more information being tracked leads to a more precise analyses. One could add some information, becoming the *path sensitive*, and then calling that *embellished*.

We focus here on adding information to make the analysis deal with proceducqe, making the analysis *context sensitive*. The additional information is about the call-site. This information is called **context**.

If we added information about the path which leads to a procedure call, one could completely disambiguate the call and return situation and, in this way, obtain the MVP solution as an embellished MF. We will take a look that, though one would not technically need all the paths in the form we have defined it so far. It will be enough to just add the sequence of all open calls, because we have to return to the last still open call. Older calls and internal edges don't need to be remembered. That will be called *call strings*. Still, that call string information, basically an abstract version of the current call stack, is unbounded; adding that yields the same precision as MVP, and that is undecidable, and we will see what to do with that.

But adding path information is *not* the only thing one can do the disambiguation. Adding the full path resp. the call string as the stack of pending calls would lead to full disambiguation. But one can also add different information, namely information about the *data* with which the procedure is called. Perhaps at some places, a particular procedur is called with a non-negative argument, and at other with negative ones. That many not disambiguate completely, but cuts down on the imprecision where to return to and thus improve the precision (and perhaps staying decidable).

To sum up the intro: we will introduce *contexts* as additional info as part of the data-flow problem, and "tracked" by the analysis. The information for the context can be of different nature, like control-flow information like the path or an abstraction thereof, or data, or combinations.

Contexts

- MVP/MOP *undecidable* (but more precise than basic MFP)

⇒ instead of MVP: **"embellish"** MFP

$$\delta \in \Delta \tag{2.16}$$

- δ: **context information**
- for instance: representing/recording of the *path* taken

⇒ "embellishment": adding **contexts**

Embellished monotone framework

$$(\hat{L}, \hat{\mathcal{F}}, F, E, \hat{\iota}, \hat{f})$$

- intra-procedural (no change of embellishment Δ)
- inter-procedural

Embellishment is notationally indicated by placing a hat on top. The following will proceed in two stages. The intra- and the inter-procedural part. The first part is, of course, simpler. One might ask, why we need to consider the first part at all? Well, we change the framework by embellishing it (indicated by the hatted syntax). That will involve a change in the lattice and other concomitant changes. When thinking in terms of an implementation, the type of the analysis changes, like the type of the involved lattice, insofar that a new component is involved, the context. Consequently, also the intraprocedual part needs to be adapted, taking care of the contexts. As far as the intra-procedural part is concert,

that is rather trivial: it basicall consists of "ignoring" the context: as long as one deals with data-flow *within* one function body, the *context* remains **the same**. Nonetheless, the additional context-component has to be mentioned as being unchanged when dealing for the embellished transfer functions and other parts of the monotone framework. But the key message is: the intra-procedural part is *"basically unchanged"* by the embellishement, as one would expect.

Notational remark

In the following we encounter silly problem. It's that we used the letter L and l for different interpretations. On is the lattice L with elements $l \in L$. The other one is the set of labels **Lab**, with typical elements l as well. Actually, in [7], there are two "different kinds" of l's, typeset with two slightly different fonts. For the slides, it's difficult to do the same, as slides are typeset with sans serif fonts, and the difference disappears. Anyway it may be confusing. What we do here instead: we still use l for the labels or nodes (as before) and write l d, d', d_i, etc. for the elements or values of the lattice (d for data).

Actually, this overloading of the letter l does not just occur now, it was already in conflict before. Though, probably, no one noticed, simply it was (hopefully) always clear what is what. Now it becomes a bit more critical: when introducing the contexts, we will encounter situations, where locations from **Loc** are part of the information stored in the lattice L.

In a way, it was like that at least for the reaching definitions: we had labels or nodes, and we had data flow information, which were pairs of variables and labels, so the elements of the lattice L contained labels l from **Lab**. However, it was probably overlooked, as we never talked about the lattice theory at the same time when we talked about reaching definitions, lattices came later. Here, when soon introducing the contexts, the danger of confusion is larger and more obvions, so we change the conventions a bit.

Functional style, using higher-order functions in the following definitions The following material and the slides will be "notationally heavy", involving a number of ingredients. Besides that, complex for a first reading may seem another aspect: the extended analysis will make use of a *functional* way of presenting things. That already starts with the embellished lattice $\hat{L}$, which is of the following form

$$\hat{L} = \Delta \to L \tag{2.17}$$

which contains functions taking a context from Δ to an element of the original, unembellished lattice L. One can convince oneself that, if L is a lattice, then so is $\hat{L}$. But what to think about it? The context reflects (an abstraction of) the situation when inside a procedure body. When not doing a call or a return, i.e., staying within one procedure body, doing intra-procedural analysis, the context is unchanged. Equation (2.17) defines the embellished lattice $\hat{L}$ by giving one lattice element $d \in L$ per context $\delta \in \Delta$. This is a way of *lifting* the lattice to the new setting. Of course we need not just to lift the lattice from L to $\hat{L}$, we need also to lift the intraprocedural transfer functions, which are functions from $L \to L$ to $\hat{L} \to \hat{L}$. If we have a transfer function f, then we write $\hat{f}$ for its

embellished counterpart. We can see this this "hatting" operation, lifting functions f to $\hat{f}$ as a higher-order function as follows:

$$\hat{\_} : (L \to L) \to (\hat{L} \to \hat{L}) \tag{2.18}$$

One could call that operation as lifting, basically mirroring the behavior of f in the new, more complex setting. Lifting is a technical term, which is used in situations like the one just described. One may have seen examples of that. For instance, the map-function over collection data structures, like "mapping" over lists is a well-known example. It's a higher-order function of the form $(A \to B) \to ((ListA) \to (ListB))$.

Actually, we also need to list the lattice relation, i.e., the partial order $\sqsubseteq$ on L to its counterpart on $\hat{L}$. One could write $\hat{\sqsubseteq}$ for that relation. We don't make that definition explicit here, but one can try oneself as an (easy) exercise. Anyway, that form of lifting expresses, that nothing relevant changes, old definitions are just "ported" to a new, more complex structure, without conceptual changes. That covers the intraprocedural aspect of the context-sensitive analysis. Of course, for the *new* aspects, involving calls and returns, we cannot just lift something, we have to come up with something new, but that's for later.

The lifting from the underlying transfer function f to its embellished counterpart $\hat{f}$ is shown in equation (2.22). The transfer function $\hat{f}_l$ (for some location l) is of type $\hat{L} \to \hat{L}$, i.e., a function of type

$$(\Delta \to L) \to (\Delta \to L) \tag{2.19}$$

So, it's a function that takes as argument a function of type $(\Delta \to L)$ and returns a function, namely one with Δ as argument and a return value from L. That's defined in equation (2.22): the "first" functional argument $\hat{d} : \hat{L}$ is applied in the definition to the "second" argument δ and the results is fed into the unlifted transfer function f_l. Equation (2.22) may be written identically as follows, making use of λ-abstraction, and adding the types of the arguments to make it more explicit.

$$\hat{f}_l = \lambda\hat{d}{:}(\Delta \to L).\lambda\delta{:}\Delta.\ f_l(\hat{d}(\delta)) \tag{2.20}$$

Currying and uncurrying Now, $\hat{f}$ is defined in some "higher-order" style, taking a function as argument and applying it (I leave out mentioning the l for now, the location f "belongs to"). There is another aspect in the definition we can discuss here; we make use of that aspect later as well.

The function takes *two* arguments, $\hat{d}$ and δ, only that it does so in a "gradual" way, first $\hat{d}$, resulting in a function that takes the second argument. Indepdent from the concrete types for the transfer function: If we have a function of type $A \to B \to C$, which is understood as \$A $\to$ (B $\to$ C), that works basically as a function that takes two arguments, from A and from B. Instead of saying that the functions two arguments, we may say, the function takes a *pair* of arguments, i.e., it's a function $A \times B \to C$. Another way to see it is that both types

$$A \to B \to C \quad \text{and} \quad A \times B \to C \tag{2.21}$$

essentially the same. Of course the two types are not really identical. A function of the first type is invoked as $(f\ a)b$, a function of the second type as $f(a, b)$, the tuple (a, b) as an element of $A \times B$. The two types are equivalent in the following sense. It's easy to take a function and turn it into a function of the second type, and also in the other direction. Each function of the first type can be "transformed" into one of the second type, but doing the same, and also in the other direction. Not only "transformed" in the one can manually re-write each function of one type equivalently into one of the other form. That might be plausuible. What is more: the exits two functions, higher-order of course, that realize those transformation. Technically, one says the two types are *isomorphic*, and the two transformations are called currying and uncurrying. A function of the functional type in equation (2.21) is said to be in *curried* form, the one operating on pairs is said to be *uncurried*. The terminology is in honor of the logician *Haskell Brooks Curry*, though perhaps historically more correct would be to call the concept schoenfinkelling (after Mosses Schoenfinkel), not currying, but perhaps that sounded too weird...

Anyway, it's not too important, but we touched upon that here, since the concept will pop up here and there in the following, and it helps to understand better how the embellished framework works is written down in the following.

Intra-procedural: basically unchanged

- this part: **"independent"** of Δ
  - property *lattice* $\hat{L} = \Delta \to L$
  - monotone functions $\hat{\mathcal{F}}$
  - transfer functions: **pointwise**

$$\hat{f}_l(\hat{d})(\delta) = f_l(\hat{d}(\delta)) \tag{2.22}$$

- flow equations: "unchanged" for intra-proc. part

$$\begin{aligned} A_\bullet(l) &= \hat{f}_l(A_\circ(l)) \\ A_s \circ (l) &= \bigsqcup\{A_\bullet(l') \mid (l', l) \in F \text{ or } (l'; l) \in F)\} \sqcup \hat{\iota_E^l} \end{aligned} \tag{2.23}$$

- in equation for $A_\bullet$: except for labels l for proc. calls (i.e., not l_c and l_r)

The embellished lattice $\hat{L}$ is defined by the function type $\Delta \to L$, and we have defined the interprocedural transfer functions (lifting them). But let's also have look at what the *solution* of the data-flow problem now is, in the embellished setting. Equivalently, let's have a look at the data structure A (resp. $A_\bullet$ and $A_\circ$), mentioned in equation (2.23) and used in the worklist algo.

The solution of a constraint problem is a mapping from the variables of the constraint system to values of the solution framework. In our setting the labels or nodes of the control flow graph play the role of constraint variable (more precisely, the entry and the exit for each node, but let's irgnore that as a side issue for now, let's use N (nodes) for short here representing $\mathbf{Lab}_*$ (resp. the entry and exits for each node or label).

Now, in the underlying unembellished framework, a solution of a problem is a mapping from the nodes of the cfg to elements of the lattice, i.e., that an unembelished solution is of the type $N \to L$, whereas *now*, the solution is of type $N \to \hat{L}$, i.e.,

$$N \to \Delta \to L \ . \tag{2.24}$$

So a solution gives, for each constraints variable (from N) a function, that yields, for each possible context, a value from the underlying lattice. Referring to the discussion about currying etc., that type is the "same" as

$$(N \times \Delta) \to L \ . \tag{2.25}$$

Seen like that, uncurried version, the context is simply "paired" with the location or node in the control-flow graph and represents relevant information of the call-site where the function was called.

Remarks concerning implementing that

Let's ponder how one could implement that. The unembellished setting is less problematic. Implemting and maintaining a function $N \to L$ is straightforward. The set N is finite, and the function is represented via $A_\circ$ and $A_\bullet$, which one might concretely realize as array.

A practical problem now might be: there can be many possible contexts from Δ, maybe even ininitely many. That poses a "challenge", independently of whether one bases the implementation on a curried or uncurried view (equation (2.24) vs. (2.25)).

In case Δ is finite, one can think of implementing $(N \times \Delta) \to L$ as a *two-dimensional* array. Still, if Δ is huge, that may not be the best choice, wasting memory (but perhaps being faster than alternatives). When Δ is infinite, that's not an option anyway.

What one would need is some solution "on-the-fly". The analysis might explore the control-flow graph, starting at the initial node (for a forward analysis) and starting with only the "initial context", the context for the main-function body. Only if a call happens in the exploration, the relevant context needs to be taking into account, like allocating memory for a new array or whatever. This way one would not need to allocate beforehand memory for contexts from Δ that may never occur during exploration.

Actually, the same "on-the-fly" technique may be used for N (and in the unembellished framework). We said N is finite, so a (updateable) finite mapping from $N \to L$ can nicely be representing as array. Though, if N is huge, one may also treat that on-the-fly. However, that may not bring much for the following reason. One may assume that all nodes are reachable anyway (resp. it's quite easy to find if some nodes are not reachable, as we mentioned earlier), so it's up-front clear what N is, and that the analysis will have to find values for all elements of N (or a known subset, if some functions are not called at all). For the Δ part, it's not a priori clear which contexts will play a role and which not, so there is a larger incentive to treat that part dynamically.

Sign analysis

In the following we illustrate the concepts on some specific analysis, known as sign-analysis. We start by defining that analysis first for the unembellished setting.

The presentation will again be heavy on the notational side, though the analysis is rather mundane. Actually, at it's core, is so straighforward, that most people will know, at some level, how it "works" anyway. Indeed, the definitional core is not even spelled out (see later), we focus on embedding the core idea of sign analysis into the embellished framerwork.

Before we start doing that in the following slides, we still want to comment on the nature of the sign analysis. One the one hand, the analysis is not really interesting at the point here, as we are doing interprocedural analysis. We just need some analysis to start with, and sign-analysis is a problem which is intuitively clear.

On the other hand, we can take the opportinity to highlight a few points. One is that the analysis is slightly different from the ones we have seen before (the 4 instances of the monotine framework). To see that, we need at least informally understand what sign analysis is. Well, the name says it. In the while language, the data to operate on is integers, the values, stored in memory are elements from $\mathbb{Z}$. The sign analysis does not exactly tries to find out what values there are, but it tries to find out an *abstraction* thereof. Namely if the value is positive or negative or 0. We write $+$, $-$, and 0 for that. I hope it's clear, those are meant as abstract values, the abstract value $+$ represents the set of concrete values $\{1, 2, 3, \ldots\}$ etc., $+$ there is not meant as addition. . . .

What makes this analysis different from the ones before? Earlier we covered 4 classical analyses which later led to the concept of monotone frameworks. One thing we stressed is the way that data flow analysis of that kind treats *decisions*, those coming from boolean expressions in conditionals or loops. The analyses did not attempt to figure out which way the actual program would run, i.e., whether the boolean condition would evaluate to true or false. Not only did the analysis not attempt to figure it out, even if one wanted, it could not distinguish the two branches, given the abstract information of the analyses. That was the case for all 4 analysis. As a technical term: the analyses were *path insensitive*.

In order to be path sensitive, the analysis must track information about the *values* or data being handled by the program. Otherwise, in a boolean condition b, it has no chance to figure, at least sometimes, whether the condition is true of false.

The 4 previous analyses were, we stressed that, 4 classical data flow analyses. The data flow constraints expressed conditions in which way abstract information "flows" through the control-flow graph. But, despite being called data flow, the information flowing through the graph was **not** concerned with **data**, data in the sense of information about stored values! That's why the analyses had to be path-insensitive. The information traced by the analysis is information **about** variables for instance, like "this variable is live here, and dead there", but no information what data is stored in the variable, not even approximatively. Similarly for the expressions, like "this or that expression is available here, but not there", but indendent of what the actual value of the expression would be (not even approximatively).

This is *different here*. Traceing information about signs means, there is approximative information about data. It's in some abstracted form, but still. With this information, it could be possible in some situations, to determine between alternatives. If at some point, one has figured out that a particular variable x can be only positive or zero, and that in that situation, the information "flows" through a boolean condition $x \leq 0$, it's clear that the false-branch is not taken.

That's not much (and perhaps one makes the decision that the sign-analysis would not attempt to do so, as being not worth the effort), but at least one could make it *path-sensitive*.

Is what we are doing known as "the sign analysis"? At the core, the sign analysis is intuitively simple, and, since we are interested in interprocedural analysis of procedure, it's even not in the focus. The sign analysis is just used as simple example we can embellish. Still we want to elaborate a bit on sign-analysis, at least we want to avoid the misconception that what we present here is "the" sign-analysis.

As often, when intested in some information as part of some (static) analysis, one can do the analysis at different levels of abstraction, precision (and effort). Let's look at the sign-analysis. At the core, instead of the infinite set $\mathbb{Z}$ of concrete values, the analysis will work on the three-element set $\{-, 0, +\}$ —let's call it **Sign**— of abstract values. The way the analysis "works" with those abstractions may differ, though.

No matter how, unavoidable is that abstraction leads overapproximation and sometimes "losing the overview". Not always one may loose overview, for instance, if one has an expression $x + y$ where it's known that x and y are positive, then it's clear that the result is positive: "plus plus plus gives plus", as everyone knows. If we do a minus on two positive numbers, however, we don't know if the result is positive, negative, or 0, it may be any of those.

Now there are two plausible choices, how to capture that lack of precision. A concrete, precise state σ is an element of $\mathbf{Var}_* \to \mathbb{Z}$. The values are abstracted into the 3-element set **Sign**. So accomodate for the lack of precision, one can let the analysis operates on *sets* of mappings from $\mathbf{Var}_* \to \mathbf{Sign}$. That is done in the analysis covered on the slides.

There is an alternative: one could say, one maps every variable to a /set of values from **Sign**. Both choices, $2^{\mathbf{Var}_* \to \mathbf{Sign}}$ and $\mathbf{Var}_* \to 2^{\mathbf{Sign}}$ form a *lattice*. That's good news, after all we know that data flow analyses basically *need* to be based on complete lattices... The two lattices are shown in equation (2.26) and (2.27).

$$L_{sign} = 2^{\mathbf{Var}_* \to \mathbf{Sign}} \tag{2.26}$$

$$L'_{sign} = \mathbf{Var}_* \to 2^{\mathbf{Sign}} \tag{2.27}$$

In the first case, a variable x known to be non-negative is represented as $\{[x \mapsto +], [x \mapsto 0]\}$. In the second representation as $[x \mapsto \{+, 0\}]$.

That's sound pretty much the "same" (and it is, as long as we have only one single variable). But let's look at a more general example: Assume two variables x and y and a situation in the lattice $L_{\mathbf{Sign}}$

$$\{[x \mapsto -, y \mapsto -], [x \mapsto +, y \mapsto +]\}$$

The "corresponding" information in the second lattice $L'_{\mathbf{Sign}}$ looks as follows:

$$[x \mapsto \{+, -\}, y \mapsto \{+, -\}] \ .$$

That should make obvious, that the two approaches are different (i.e., the two lattices are not isomorphi: $L'_{\mathbf{Sign}} \not\cong L_{\mathbf{Sign}}$), and the first one is strictly more precise (at last if one deals with two variables or more).

Sign analysis (unembellished)

- $\mathbf{Sign} = \{-, 0, +\}$, $L_{sign} = 2^{\mathbf{Var}_* \to \mathbf{Sign}}$
- abstract states $\sigma^{sign} \in L_{sign}$
- for *expressions:* $[\_]^{\mathcal{A}_{sign}}_{\_} : \mathbf{AExp} \to (\mathbf{Var}_* \to \mathbf{Sign}) \to 2^{\mathbf{Sign}}$

Transfer function for $[x := a]^l$

$$f_l^{sign}(Y) = \bigcup \{\phi_l^{sign}(\sigma^{sign}) \mid \sigma^{sign} \in Y\} \tag{2.28}$$

where $Y \subseteq \mathbf{Var}_* \to \mathbf{Sign}$ and

$$\phi_l^{sign}(\sigma^{sign}) = \{\sigma^{sign}[x \mapsto s] \mid s \in [\![a]\!]^{\mathcal{A}_{sign}}_{\sigma^{sign}}\} \tag{2.29}$$

We start with the *unembellished* part, i.e., without even considering contexts. For that basic setting, the lattice we start with is a set of functions; we can think of it as a set of states.

The above definition of the transfer function proceeds in *3 steps*: At the core is the semantic function $[\![\_]\!]$. This function is for *expressions,* and is already non-deterministic. Equation (2.29) reflects the effect of an assignment for one abstract state and equation (2.28) is finally the transfer function. Note, it's another example of lifting: the function ϕ, defined in equation (2.29) per "sign-state", is lifted to work on sets of state in the obvios manner (just on each individual). One also says, ϕ is lifted pointwise.

Why does $[\![\_]\!]^{\mathcal{A}_{sign}}_{\_}$ give back a *set*? Clearly, because of the non-determinism due to abstraction.

The sign-analysis is *not* yet embellished here (embellished = adding context). This means, there is not even a mentioning of Δ here. The real work is done in ϕ: the overall input to that function is Y, which is a set of states, and ϕ_l^{sign} just applies it pointwise, interpreting the expression on the right-hand side of the assignment and updating the state accordingly.

On the next slides, we will *embellish* the analysis. Since we are not yet in the interprocedural part, the embellishment is not very interesting, just a "lifting" to the embellished setting. We know how that work already, abstractly (see equation (2.22)).

As mentioned before, the abstract lattice $\hat{L}$ is of the form $\Delta \to L$. Here the underlying lattice L is of a particular form, namely $2^{\mathbf{Var}_* \to \mathbf{Sign}}$. With this one, we, in a way, make use of uncurrying to represent $\Delta \to 2^{\mathbf{Var}_* \to \mathbf{Sign}}$ slightly diffently (but isomorphically). In equation (2.30), we use the symbol $\simeq$ of that. Since we used a slightly different representation, not a function from contexts to sets of sign-states, but sets of pairs containing contexts and set-states, the transfer function in equation (2.31) below is not in the form of a higher-function as done generally before, in equation (2.22).

Sign analysis: embellished

$$\begin{aligned} \hat{L}_{sign} &= \Delta \to L_{sign} \\ &= \Delta \to 2^{\mathbf{Var}_* \to \mathbf{Sign}} \simeq 2^{\Delta \times (\mathbf{Var}_* \to \mathbf{Sign})} \end{aligned} \tag{2.30}$$

Transfer function for $[x := a]^l$

$$\hat{f}_l^{sign}(Z) = \bigcup \{\{\delta\} \times \phi_l^{sign}(\sigma^{sign}) \mid (\delta, \sigma^{sign}) \in Z\} \tag{2.31}$$

The unembellished one so far was a simple instance of the monotone framework. The transfer function just "joins" all possible outcomes, where it is assumed that we have as function that calculates the set of signs for expression. That was completely standard. Now, it does not get really more complex: equation (2.31) just does *nothing* with the contexts δ, since we are still *within* a single process. In the following we go to the interprocedural fragment and there, things get more complex, since for dealing with calls and returns we have to *connect* the contexts of the caller and the callee. It's a bit like parameter passing.

Inter-procedural

- procedure *definition* $\texttt{proc}(\texttt{val}\, x, \texttt{res}\, y)\ \texttt{is}^{l_n}\ S\ \texttt{end}^{l_x}$:

$$\hat{f}_{l_n}, \hat{f}_{l_x} : (\Delta \to L) \to (\Delta \to L) = id$$

- procedure call: $(l_c, l_n, l_x, l_r) \in IF$
- here: forward analysis
- call: 2 transfer functions/2 sets of equations, i.e., for all $(l_c, l_n, l_x, l_r) \in IF$

2 transfer functions

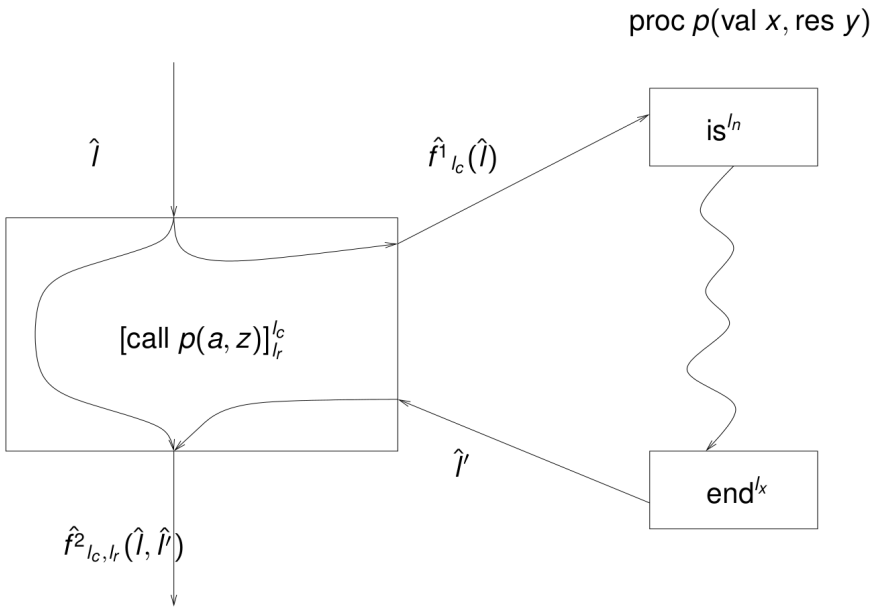
1. for calls: $\hat{f^1}_{l_c} : (\Delta \to L) \to (\Delta \to L)$

$$A_\bullet(l_c) = \hat{f^1}_{l_c}(A_\circ(l_c)) \tag{2.32}$$

1. for returns: $\hat{f^2}_{l_c,l_r} : (\Delta \to L) \times (\Delta \to L) \to (\Delta \to L)$

$$A_\bullet(l_r) = \hat{f^2}_{l_c,l_r}(A_\circ(l_c), A_\circ(l_r))) \tag{2.33}$$

Procedure call

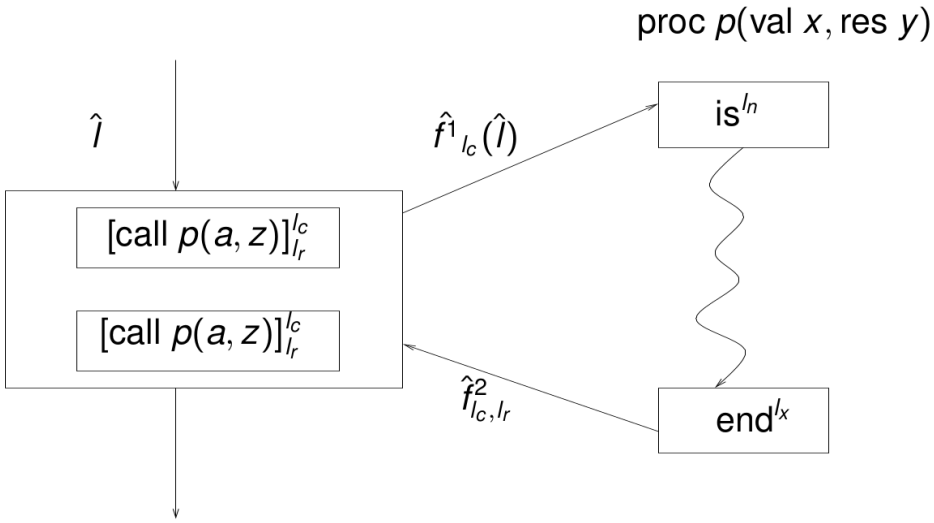


Note: *we* write $\hat{d}$ for elements of the embellished lattice (not $\hat{l}$ as in the pic). Note the unfortunate notational collision in the picture: $\hat{l}$: element of embellished lattice (abstract value), l_c etc: nodes/labels in the control flow graph. The situation may become even more confusing for analyses like RD: there *labels* (which are nodes in the control-flow graph) are part of the values of interest and thus also elements of the lattice.

Next come two different *simplifications* for f^2. However, one way to understand the 2 arguments for the return is that often one wants to **match** the return with the call (via the context).

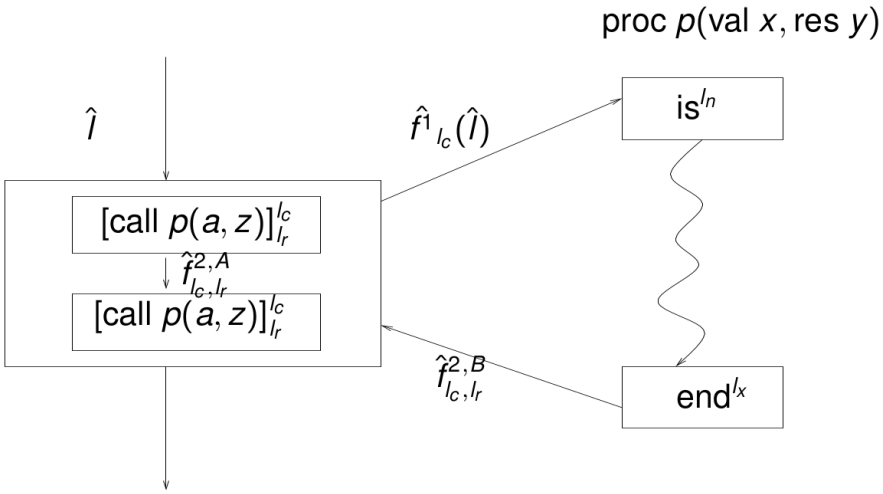
Ignoring the call context

$$\hat{f^2}_{l_c,l_r}(\hat{d}, \hat{d'}) = \hat{f^2}_{l_r}(\hat{d'})$$



Merging call contexts

$$\hat{f}^2_{l_c,l_r}(\hat{d},\hat{d}') = \hat{f}^{2A}_{l_c,l_r}(\hat{d}) \sqcup \hat{f}^{2B}_{l_c,l_r}(\hat{d}')$$



Context sensitivity

- IF-edges: allow to relate returns to **matching calls**
- context **insensitive**: proc-body analysed *combining* flow information from **all** call-sites.
- *contexts*: used to distinguish different call-sites *and* match calls and returns

$\Rightarrow$ context *sensitive* analysis $\Rightarrow$ more precision + more effort

In the following:

2 *specializations*:

1. control ("call strings")
2. data

(combinations are, of course, possible) Combinations of the two approaches are not covered in the lecture. The call-strings corresponds more or less to the previously sketched MVP approach.

Call strings

- context = (some form of) *path*
- call-string = sequence of currently "active" calls
- concentrating on calls: flow-edges (l_c, l_n), where just l_c is recorded

$$\Delta = \mathbf{Lab}^* \qquad \text{call strings}$$

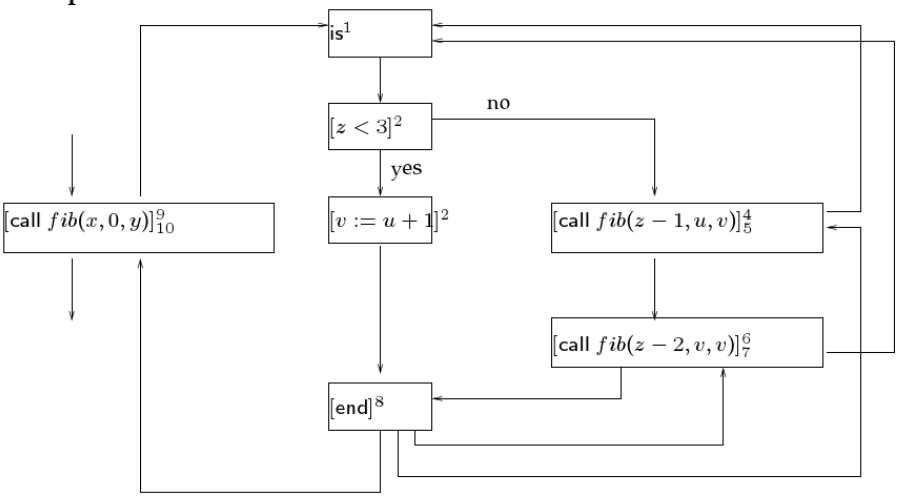
- *extremal* value (from $\hat{L} = \Delta \to L$)

$$\hat{\iota}(\delta) = \begin{cases} \iota & \text{if } \delta = \epsilon \\ \bot & \text{otherwise} \end{cases}$$

The definition of $\hat{\iota} : \hat{L}$ should be clear: at the beginning of the program, there are no calls; hence the call string is empty (represented by ϵ). Note again the higher-order approach. The $\hat{\iota}$ is somehow defined again point-wise.

Fibonacci flow

Example: fibonacci flow



Fibonacci call strings

some call strings:

$$\epsilon, [9], [9,4], [9,6], [9,4,4], [9,4,6], [9,6,4], [9,6,6], \ldots$$

The call strings are *not* the same as the valid paths. Both concepts are related, though. An important difference is the treatment of the returns. In the valid (or complete) path description, the returns are part of the paths, and the paths "never forget", they only grow longer. Here, when dealing with a return, the path does not get longer, it gets *shorter* be removing the the previous call. The call string only tracks the currently open calls. It corresponds to the current depth in the *call-stack*. That is also the way to *match* the contexts of the callee and the caller. The paths as defined before also recorded *internal* nodes, unrelated to calls and returns. That also is ignored in call-strings.

Transfer functions for call strings

- here: forward analysis
- 2 cases: define $\hat{f}^1_{l_c}$ and $\hat{f}^2_{l_c,l_r}$

Transfer functions

- **calls** (basically: check that the path ends with l_c):

$$\begin{array}{rcl} (\hat{f}^1_{l_c}\ \hat{d})\ [\delta; l_c] & = & f^1_{l_c}(\hat{d}(\delta)) \\ (\hat{f}^1_{l_c}\ \hat{d})\ \_ & = & \bot \end{array} \tag{2.34}$$

- **returns** (basically: **match** return with (a same-level) call)

$$\begin{array}{rcl} (\hat{f}^2_{l_c,l_r}\ (\hat{d}, \hat{d}'))\ \delta & = & f^2_{l_c,l_r}(\hat{d}(\delta), \hat{d}'([\delta; l_c])) \\ (\hat{f}^2_{l_c,l_r}\ (\hat{d}, \hat{d}'))\ \_ & = & \bot \end{array} \tag{2.35}$$

- given: underlying $f^1_{l_c}$ and $f^2_{l_c,l_r}$.
- rather "higher-order" way of connecting the flows, using the call-strings as contexts
- *connection* between the arguments (via δ) of f_{l_c,l_r}
- Notation: $[\delta; l_c]$: *concatenation* of calls string
- $\hat{d}'$: at procedure exit.

The definition is quite condensed or "convoluted", since it's pretty higher-order. But what it achieves is exactly what we wanted. One can separate that into two parts: a) if a function is called from two different places, differentiate the two situations (context-sensitivity), and b), when returning from a call, return to the call site. So far what the definition is supposed to capture.

The starting point are the *given* two unembellished transfer functions for the call and for the returns $f^1_{l_c}$ and $f^2_{l_c,l_r}$. Those capture what's really going on, in calls and returns, given some specific analysis. So, the task is on the one hand lift that intended transfer behavior from the unembellished to the embellished setting. Like we did for the intra-procedural part. What is not captured in the underlying $f^1_{l_c}$ and $f^2_{l_c,l_r}$ is two aspects a) and b) just

mentioned: differentiate the call sites and return to the proper place. For that, we make use of the call string context.

Let's first have a look at how it's done for calls in equation (2.34). The "lifting" of f_{l_c} is done analogously the same higher-order way as we have seen in the intraprocedural case (in equation (2.22). But the higher-order application of the underlying function is *not quite identical*. In equation (2.22), the context δ does not play a role at all: $(\hat{f}_l\ \hat{d})\ \delta = f_l\ (\hat{d}\ \delta)$. Now, for equation (2.34), the call-string context δ changes, that's the key thing. The change of δ from the caller to the extended one $[\delta; l_c]$ effectively lifts $f^1_{l_c}$ to the embellished setting connecting analysis-results *only* for matching callers and callees and far as the call-string is concerned. That transfers the data flow information according to the underlying transfer function f_{l_c} from the caller to the callee, and the callee is analyzed with that information and in a context appropriately extended l_c. Remembering that caller-information distinguishes the different call-site instances and allows also allows also to connect the callee back to the site where it had been called in equation (2.35).

In the definitions, _ stands for a "wild-card" argument, like all cases not coverd by the first line. So the two lines of the definitions in (2.34) and in (2.35) are to be read in a first-match manner (as in programming languages).

Note that in the else case is defined as $\bot$, i.e., as the bottom value of the underlying lattice. That means, there is no data flow information transfer in those cases (since $\bot \sqcup d = d$), so there is information transfer *only* between matching pairs of activations of callers and callees. That coupling is exactly what we want.

The same technique is use also for lifing the transfer function $f^2_{l,c}$ (again assumed given) for returns and matching the corresponding activations of callee and caller in equation (2.35).

In the following, we apply the definition in the specific setting of the sign-analysis. The definition of how to cover calls and returns with the corresponding transfer function where "involved", due to the higher-order formulation. In the specific setting of the sign-analysis, we make use of *uncurrying* the representation (as we did before), i.e., we operate on sets of pairs rather than on applying functions (remember equation (2.30)).

Sign analysis (continued)

- so far: "unconcrete", i.e.,
- given some underlying analysis: how to make it context-sensitive
- call-strings as context
- now: apply to some simple case: signs
- remember: $\hat{L} \simeq 2^{\Delta \times (\mathbf{Var}_* \to \mathbf{Sign})}$ (see Eq. (2.30))
- before: standard embellished $\hat{f}^{\mathbf{Sign}}_l$ (with the help of $\phi^{\mathbf{Sign}}_l$)
- now: *inter-procedural*

Sign analysis: aux. functions ϕ

still unembellished

calls: abstract parameter-passing

$$\phi_{l_c}^{sign1}(\sigma^{sign}) \quad = \quad \{\sigma^{sign}[x \mapsto s][y \mapsto s'] \mid s \in [\![a]\!]_{\sigma^{sign}}^{\mathcal{A}_{sign}},\ s' \in \{-, 0, +\}\}$$

returns (analogously)

$$\phi_{l_c,l_r}^{sign2}(\sigma_1^{sign}, \sigma_2^{sign}) \quad = \quad \{\sigma_2^{sign}[x, y, z \mapsto \sigma_1^{sign}(x), \sigma_1^{sign}(y), \sigma_2^{sign}(y)]\}$$

(formal params: x, y, where y is the *result parameter*, actual parameter z)

- non-det "assignment" to y
- remember: operational semantics,

The underlying functions ϕ work straightforwardly. The call-case represents parameter passing. The input parameter (here x)— is treated in the language by call by value. Here, we are working on the abstract level, and instead of passing a concrete value as actual parameter by-value, we pass elements from **Sign**. The values to be passed are determined by $[\![a]\!]_{\sigma^{sign}}^{\mathcal{A}_{sign}}$. Due to abstraction, the evaluation of expression a may be more than one element s from **Sign**. The result-parameter y is treated randomly, picking an arbitrary element from **Sign**. That corresponds the treatment in the operation semantics by rule Call.

Sign analysis: embellished f's for call and returns

calls: abstract parameter-passing + glueing calls-returns

$$\hat{f}_{l_c}^{sign1}(Z) \quad = \quad \bigcup\{\{\delta'\} \times \phi_{l_c}^{sign1}(\sigma^{sign}) \mid (\delta, \sigma^{sign}) \in Z, \delta' = [\delta; l_c]\} \tag{2.36}$$

Returns: analogously

$$\hat{f}_{l_c,l_r}^{sign2}(Z, Z') \quad = \quad \bigcup\{\{\delta\} \times \phi_{l_c,l_r}^{sign2}(\sigma_1^{sign}, \sigma_2^{sign}) \mid \begin{array}{l} (\delta, \sigma_1^{sign}) \in Z \\ (\delta', \sigma_2^{sign}) \in Z' \\ \delta' = [\delta; l_c] \end{array} \} \tag{2.37}$$

(formal params: x, y, actual parameter z)

Here we see the "uncurrying" we discussed before in action, exploiting the special form of the lattice for the sign analysis (equation (2.30)). Well, actually the situation is not so special. Many analyses work with sets of this or that, i.e., with subset-lattices. So in very many cases, one can switch to an uncurried representation., like here. That allows a formulatiion *pairing* contexts δ with the rest of the information, as opposed to feeding the context as argument in some higher-order function, as in the general formulation.

The lifting of the underlying $\phi_{l_c}^{sign1}$ in the case of calls in equation (2.36) extends δ from the caller to $\delta' = [\delta; l_c]$ in the callee. The same connecting callee and caller is done for the returns in equation (2.37)

Call strings of bounded length

Back from the specific sign analysis to the general setting of the embellished monotone framework, in particular, the embellishedment with call strings. What we have achieved is: turning the MVP approach as a instance of a monotone frame by making the lattice more complex and lifting the underlying transfer functions to the more complex setting ("embellished" as it's called here). The embellished lattice, however, has become quite more complex. In many settings, the underlying lattice L is finite (like in the 4 classical examples and also in the case of the sign analysis). The embeløished lattice $\hat{L}$, however, is definitely not finite. That means there's not guarante for *stabilization* for the worklist algo or the chaotic iteration. That perhaps should not come as surprise. It has been mentioned that MVP is undecdable. In general, at least, there may be particular programs where it's not, of course. Still, since adding call strings is a way of encoding MVP, that cannot make a undecidable problem decidable, so there cannot be a guartee of stabilization.

What can we do about? Well, after all that complex formalization, it's pretty simple and pragmatic. It simply say; don't use call strings of unbounded length, just take call string up to a particular length.

That corresponds to the following: calls and returns are matched up-to a predefined depth of the call string. Afterwards, the analysis gives up on that precision, and works like the naive approach. In the extreme case for $k = 0$, i.e., having no call-strings whatsover, we are not suprisingly back in the context-insensitive approach as special case.

- recursion $\Rightarrow$ call-strings of unbounded length

$\Rightarrow$ restrict the length

$$\Delta = \mathbf{Lab}^{\leq k} \qquad \text{for some } k \geq 0$$

- for $k = 0$: context-insensitive ($\Delta = \{\epsilon\}$)

What follows now is an alternative to call strings. It's also an an analysis, that can be seen as analogous to call-strings of bounded length, namely of length $k = 1$. The previous short discussion said, one should simply just consider call-strings up to some boundary and afterwards, cut it off or ignore it. That's plausible, and the next analysis does just that: it will not remember more information than about the last call. That corresponds to a depth of k_1, only what we don't stack up the return addresses (i.e., the l_c's), but information about data or states. But one can see how concretely one can "cut-off" contextual information; it's not hard anyway, instead of a stack ("call string") one uses just "one slot of memory", and each time a new call is done, the "older" information is discarded.

Another form of contexts: assumption sets

As mentioned earlier, there are two different kinds of information one can use for contexts. One can base the contexts on "control" or on "data". They can also be compbined, using information about both data and control (though we don't show an example). What we have seen so for, the call string approach, is based purely on control. It stores an abstract version of the current call-stack, remebering a stack of labels l_c. It's indeed an abstraction of the concrete call stack, ignoring everything except the return addresses (in the form of the labels). With that contextual information, the data flow can analysize the program, returing precisely to the call-site instance (that's what a return address is good for, after all).

But one can use other information besides the return address, one can use the *data value* with which a function is called to at least narrow down where the function was called from. There's no garantee that it locates the call site instance precisely, as the return address, but still in can disambiguate the situation to some extent, thereby adding precision to some extent. In general,

it won't be possible use actual, concrete data values for that. Like in our languages, integers. The data domains for arguments are often infinite, and anyway, the underlying analysis concretely not even track value. Thus, one will generally use *abstractions* as context, not concrete values. And actually, many analyses not even work with abstractions of data. For illustration: our toy language uses integers as data values. However, none of the 4 classical analyses actually worked with abstractions in $\mathbb{Z}$! Only the sign-analysis later did that, abstracting (or partitioning) the infinite set $\mathbb{Z}$ into three classes, the positive, the negative, and the set containing 0.

The earlier 4 analysis worked on other abstraction ("is a variable life here" and similar). But also that information can be used to disambiguate calls to some extent: one call is analyzed from places where x is live, and another instance of a call where it is not.

A small side remark concerning "data" and "control": We made the point that call-strings are contexts with control-information and the assemption set method uses "data" information instead. That's an ok way to see it, but it's perhaps not strictly true in call cases. For example, the information for reaching definitions had been sets pairs of locations and variables. I.e., the lattice containained control information, namely the locations where variables had been defined, i.e., given their last value. So, one should not interpret the situation here too scrict in thinking that there are absolutely no pieces of control information involved. The subsequent slides are rather unspecific of what that information could be, Some set D is assume ("data"), and as lattice one assumes subset lattice:

$$L = 2^D \tag{2.38}$$

In that generality, it covers pretty much. The slides mentions that the lattice represents "assumptions" on states. If we interpret D as containing states, then 2^D contains are sets of states. That can be seen as contraining assumptions or properties of states, namely all those states where the property or assumption holds.

Technically, we had defined (concrete) states as an element from $\mathbf{Var}_* \to \mathbb{Z}$. Though the formulation here is *not* meant as applying only to setting where done does assumptions on particular this definition on states. The idea can be used for lattices of the form as specified in equation (2.38). Which is basically for all analyses we have seen.

There is only one example that may look like an exception: that was when discussing an alternative of the sign-analysis, there the lattice was $\mathbf{Var}_* \to 2^{\mathbf{Sign}}$. That was the more abstract version were the variables were treated uncoupled. However, using "currying/uncurrying" one can make also that conform to the format 2^D.

Conceptually, one can use the idea broadly. Technically the slides show how to lift the transfer function and connect calls and returns making use of a auxiliary function ϕ. We have seen an example of that in the sign-analysis. To define the transfer functions with the help auxiliary functions ϕ was done previously in the sign-analysis (and only there). The analysis made use of "abstract states" $\sigma^{sign} : \mathbf{AState} = \mathbf{Var}_* \to \mathbf{Sign}$. The auxliary function ϕ had been of type $\mathbf{AState} \to 2^{\mathbf{AState}}$, it captured the general case where the (abstract) state is changed. That happens for the sign analysis for assignments and for parameter passing. Working on abstract sign-information, the "result" of the auxiliary function is, in the general case, a *set* of abstract states, i.e., an element of $2^{\mathbf{AState}}$. That captures the "non-determinism" in the effect of the abstraction, in particular for the assignment. Remember in particular equation (2.29), which made that clear for the sign-analysis. This setting, where transfer functions of type $L \to L$, i.e., $2^D \to 2^D$ are defined making use of an auxilary function of type $D \to 2^D$ is characteristic for situations, when D is seen as (abstract) state, and the sign-analysis is an example for that. That means, the other 4 analyses, where the lattices is perhaps not naturally called "state" and where the transfer functions are not defined via some auxiliary construction ϕ, will not be exactly formulated the way we do in the following. That does not imply, one cannot introduce context that contain information "data"

information, like having $\Delta = L$. One can do that, though the embellished transfer functions cannot be written exactly as in the formulas that follow. But one can of course just use the general curried, higher-order formulation from equation (2.20), that works for *all* kinds of contexts. Anyway, here it's in a setting that resembles more the sign-analysis, with D corresponding to the type of abstract states $\mathbf{AState} = \mathbf{Var}_* \to \mathbf{Sign}$.

Assumption sets

- **alternative** to call strings
- not tracking the path, but assumption about the "state"
- assume here: lattice $L = 2^D$

$\Rightarrow$ $\hat{L} = \Delta \to L \simeq 2^{\Delta \times D}$

- dependency on "data" only $\Rightarrow$

$$\Delta = 2^D$$

- $\hat{\iota} = \{(\{\iota\}, \iota)\}$ extremal value

Transfer functions

We show how to lift the underlying transfer functions to the embellished setting and we do that for the cases of calls and returns; the easier lifting the transfer functions when doing intra-procedural analysis are not shown (resp. it's done anyway as before: contexts don't change).

As before in the sign-analysis, we don't lift the actual transfer functions f to their embellished counter-parts $\hat{f}$, but we take the auxiliary function ϕ (used in the definition of the f's) and lift that to $\hat{f}$. For calls, the auxiliary function is of type

$$\phi^1_{l_c} : D \to 2^D$$

and the corresponding embellished transfer function is given in equation (2.39). It's analogous to the definition we did for the sign analysis and with call strings in equation (2.36).

As for the definition from equation (2.39): the type of the transfer function (for calls) is

$$\hat{f}^1_{l_c} : \hat{L} \to \hat{L} = (\Delta \to L) \to (\Delta \to L) \quad \text{with} \quad \hat{L} \simeq 2^{(2^D \times D)} \ .$$

Elements of the lattice are thus sets of pairs, the parts of each tuple consists of the current data or state d in the second position and a set of data or states δ in the first position, representing the *contexts*. As mentioned, the general picture here is: when calling a function, use the current data d to identify the place of the call, where the identification is not 100% precise, so it's about narrowing down possible places where the procedure is called.

But actually, that's not all. It's not just the data d, handed over by the caller that is used as context of the callee to disambiguate different callers. It's additionally the caller-context δ. That's the tuple $(\delta, d) \in Z$ in the definition, where δ came from the caller of the caller.

However, that additional information does not *"stack-up"*. That stacking up was done with the call-string approach, maintaining a call-string during the analysis. Here, what is remembered when analysing a procedure body is the context of the caller (together with d), but not the context of the the earlier callers of the caller. In other words: it corresponds to a call-string *with bound* $k = 1$ (except that the information remembered is not the control-information l_c as for the call-strings, but d from D.

The transfer function for return from equation (2.40) works correspondingly. Connecting the callee back to the caller by requiring the connection between the shape of the caller context δ and the callee context δ'.

- calls

$$\hat{f}^1_{l_c}(Z) = \bigcup\{\{\delta'\} \times \phi^1_{l_c}(d) \mid \begin{array}{l}(\delta, d) \in Z \wedge \\ \delta' = \{d'' \mid (\delta, d'') \in Z\}\end{array}\} \tag{2.39}$$

where $\phi^1_{l_c} : D \to 2^D$

- note: new context δ' for the procedure body
- "caller-callee" connection via the context (= data) δ
- return

$$\hat{f}^2_{l_c,l_r}(Z, Z') = \bigcup\{\{\delta\} \times \phi^2_{l_c,l_r}(d, d') \mid \begin{array}{l}(\delta, d) \in Z \wedge \\ (\delta', d') \in Z' \wedge \\ \delta' = \{d'' \mid (\delta, d'') \in Z\}\end{array}\} \tag{2.40}$$

Small assumption sets

The following cuts down on the contextual information tracked. The only thing tracked is the "data" with which the function is called (abstractly), not by whom (represented by the caller context). That does not mean there is no context, the data is the callee context, but not paired with information about the caller.

As explained, the previous approach can be seen as taking "historical" context-information with a bound $k = 1$ into account. One can simplify that further, storing no such information, i.e., $k = 0$. That's not a context-insensitive approach, as it had been the case with call-strings: there $k = 0$ meant: no information is used whatsoever. Here, we use "data" (or state) information from D, and we simply use that as callee context, but *not* also the context of the caller. I.e., we used the data (here d) itself to distinguish different call instance and connect back the analysis when returning, but *ignore* the context of the data. Basically, we identify all call activations with the same "arguments" from all possible callers, though of course the arguments are actual, concrete data, but the information used in the flow analysis. We ignore how, however, the context of the caller(s), as was done with the previous approach. That approach here known as *small assumption sets* (where as the previous one or similar are consequently *large* assumption set approaches).

- throw away even more information.

$$\Delta = D$$

- instead of $\hat{L} = 2^{(2^D \times D)}$: now only $\hat{L} = D \times D$.
- transfer functions simplified

$$\hat{f}^1_{l_c}(Z) = \bigcup\{\{d\} \times \phi^1_{l_c}(d) \mid (\delta, d) \in Z\}$$

$$\hat{f}^2_{l_c,l_r}(Z, Z') = \bigcup\{\{\delta\} \times \phi^2_{l_c,l_r}(d, d') \mid \begin{array}{l}(\delta, d) \in Z \wedge \\ (d, d') \in Z'\end{array}\}$$

Flow-(in-)sensitivity

The interprocedural analysis became rather involved (and can be costly to do). At the end of that section, we get rather trivial for a change. Before we do that, we talk about "sensitivity": analyses can be characetrized by their sensitivity (or sensitivities, analysis can be sensitive to more than one thing). We have seen examples for that. Right now, we have been doing *context-sensitive* analysis for procedures. Being sensitive to contexts here means, being able to distinguish to some degree diffent call-site instances. Similary, we touched shortly upon *path-sensitivity*: being able resp. remember whether one is analyzing the true- or the false-branch in a conditional.

There is another sensitivity, namely being sensitive for the "flow" or not. By that one means, does the *order* of execution of statements or blocks matter. The control flow graph is a representation of (an abstraction of) the "flow" (= control flow), and respecting the flow means following the edges of the control flow. For a flow-insenstive analysis, not caring about the order of execution, one does not even need a CFG, at least not the edges, as a flow-insentitive analysis does not care.

To avoid a misconception: we did data flow analysis using algorithms like chaotic iteration or some worklist formulations. Especially the chaotic iteration seem not to care about the order in which it tackles edges, it randomly treats them until stabilization. That does not mean the algorithm works flow-insensitive. Flow-sensitivity or flow-insensitivity is about the problem, not about solution strategies. Those algorithms worked on the control-flow graph (resp. constraint systems respresenting the data-flow problem over the control flow graph). The algorithms certainly honored the order of statements as far as the result is concerned, though in the solution process, the traversal may not treat the edges in the order of the flow (though following the edges may be a good strategy, at least better than doing it randomly). And obviously, the data flow problems we looked at had been flow-senstive: for reaching definitions, it's about whether at some location, a variable had been defined (= assigned-to) *before*. For live variables, it's about whether a variable may be used *afterwards*. So, the flow order is crucial.

So we have actually not seen a flow insensitive problem. But it's plausible, that those should be fairly trivial. It might even not be immediate to come up with an *interesting* flow insensitive problem. What can one statically analyze which is not ridiculously easy? One can count the number of occurences of an identifier, but that's so simple that one can do it by grepping the source code, one might not even call that program analysis. It might get a bit more complex if one would take scopes into acount (and anyway, in this lecture we are not analyzing source code, we analyze abstract syntax). But still the analysis would be quite simpler than all we have discussed. Later we will see one that somehow finds out variables, not just all, but all that are modified. Sometimes that's information kept in interfaces (of methods, procedures etc) in the form of a so-called *modify-clause*.

Technically, flow insensitivity does not mean, all "orderings" in the program can be ignores. Technically, it means that the order imposed by sequential composition ("semicolon") is irrelevant, see equation (2.41). But there may be other, less obvious "ordings" in a program. For instance, one may be interested in a so-called *call-graph*: which procedure calls which one, i.e., which proceure may call which one, for instance in the while-language with procedure (in higher-order languages, things get much more involved; we have stumbled upon that already in the introduction). When doing such a call graph, one is interested in that, for instance f calls g, in that sense that the callee g can start executing only *after* that f has started (unless g may call f as well, in which case one might not now). At any rate, the call graph can contain some form of ordering but others not. for instance, if f calls uniformely g_1 an afterwards g_2, then the ordering between g_1 "before" g_2 is not part of the call-graph. The corresponding information is flow-insensitive, in the sense defined. Any it's also not ridiculously trivial, one has to build up some graph, but it's still fairly simple, at least for a while-language with procedures of the form we have seen. Also the analysis IAV later (implicitly) performs such a call-graph analysis.

- "execution order" influences result of the analysis:

$$S_1; S_2 \quad \text{vs.} \quad S_2; S_1 \tag{2.41}$$

- flow in-sensitivity: order is irrelevant
- (quite) less precise (but "cheaper")
- for instance: *kill* is empty
- sometimes useful in combination with inter-proc. analysis

Set of assigned variables

- for procedure p: determine

$$\mathsf{IAV}(p)$$

global variables that may be assigned to (also indirectly) when p is called

- two aux. definitions (straightforwardly defined, obviously flow-insensitive)
  - $\mathsf{AV}(S)$: assigned variables in S
  - $\mathsf{CP}(S)$: called procedures in S

$$\mathsf{IAV}(p) = (\mathsf{AV}(S) \setminus \{x\}) \cup \bigcup \{\mathsf{IAV}(p') \mid p' \in CP(S)\} \tag{2.42}$$

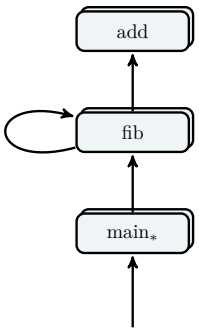
where $\texttt{proc}\, p(\texttt{val}\, x, \texttt{res}\, y)\texttt{is}^{l_n}\, S\, \texttt{end}^{l_x} \in D_*$

- $\mathsf{CP} \Rightarrow$ procedure call graph (which procedure calls which one; see example)

Example

```
begin   proc fib(val z)is
              if    [z < 3]
              then  [call add(a)]
              else  [call fib(z − 1)];
                    [call fib(z − 2)]
        end;
        proc add(val u)is(y := y + 1; u := 0)
        end
        y := 0; [call fib(x)]
end
```

Example



$$\begin{aligned} \mathsf{IAV}(\mathit{fib}) &= (\emptyset \setminus \{z\}) \cup \mathsf{IAV}(\mathit{fib}) \cup \mathsf{IAV}(\mathit{add}) \\ \mathsf{IAV}(\mathit{add}) &= \{y, u\} \setminus \{u\} \end{aligned}$$

$\Rightarrow$ smallest solution

$$\mathsf{IAV}(\mathit{fib}) = \{y\}$$

2.7 Static single assignment

This section is not covered by the book Nielson et al. [7]. It's added this year as it's an interesting and important angle on data flow analysis. The book covers so-called *definition-use* or *use-definition* chains (aka du- and ud-chains), which are left out instead this time. Only so much: du- and du-chains are basically a generalization of reaching definitions, one version working forward (like rd) one backward. And SSA can be seen as a generalization, in turn.

More than one ever wants to know can about SSA can be found in [5].

Origins

Early citations are Rosen et al. [9] Alpern et al. [1], and Cytron et al. [3]. So also historically, it's connected to value numbering.

Intro

- improvement on *def-use chains*, connecting "definitions" of variables to their uses
- *important* intermediate representation
- used in *many* compilers (gcc, go, swift, llvm, ...)
- referential transparency

SSA A program is in SSA form if each variable is a target of *exactly one assignment* in the program *text*.

Referential transparency refers to a in generally welcome property of expressions (in this case variables). It means that a value of an expression (or here a variable) is *independent of where the expression appears*.

The statement should perhaps be qualified in that the value of a variable is always the same (unless there is *no value*, i.e., the variable is undefined). If we ignore the "being undefined" as special status, then a referentially transparent value means indeed the value of the variable is "always the same" (and that means the value is *immutable* and the variable is *single assignment*, and there might be other characterizations as well). It's also related to *functional* or *declarative programming*, If one deals with referentially transparent variables, many things become easier, the concept of variable becomes more *logical*. Variables in imperative programming languages have the flavor of being names to memory addresses, whereas variables in mathematical text books, equations, or in logical formulas do not give that feeling. The reason being that expressions and variables are intuitvely understood as being referentially transparent (not that any math text would ever point that out, it simply does not cross anybody's mind what a variable has to do with a mutable memory cell ...) Also in the lecture, we carefully try to separate assigmnents $x := e$ from equations $x = e$, the latter being understood as referentially transparent (or declarative ...). The opposite of referentially transparent is also known as referentially opaque.

Anyway, one way of understanding the general motivation of SSA is that it's a format that is referentially transparent. One nice property is, that if one has a unique assignment `x:=e;S` which is referentially transparent. which we might also write `x=e;S` or actually `let x = e in S`. The semantics of the construct is, that *first*, e is evaluated to some value, *then* that value is stored in x, and *finally*, the rest S is evaluated. Now, in a single-assignment setting, once x has gotten its value (say v), then that value of x won't change any more. As a consequence, the variable and the value are "synonymous", x "is" v. That captures the most fundamental property what it means to be "equal", the standard mathematical, logical (referentially transparent, declarative ...) meaning of equality:

> *two things being equal can be used interchangably.*

After evaluating e to v, the $\texttt{let}\, x = v\, \texttt{in}\, S$ can be explained as $S[v/x]$, replacing or substituting x by v in S. Obviously, such a substitution explanation does not work for general (non-single) assignments $x := v; S$ (but for single-assignments it holds).

What relevance does that have for static analysis? Basically, static analysis in the form of this lecture is an *automatic logical analysis of (abstract) properties of programs* (like variables being live, etc.). The cleaner the program, the easier the analysis, and if the variables of the program behave like logical variables, that may help in a correct analysis. Note that ultimately, of course, the "logical, single-assignment variables" must be mapped to mutable memory cells of a standard von-Neuman architecture or variations thereof. Nonetheless, during the semantic phase, the analysis may profit from a logically clean intermediate representation.

All that may be a bit philosophical. More down to earth is SSA as intermediate representation: it's a format which has some data flow analysis already *"built in"*. If there is only one assignment to each variable, then when *using* a variable, it's already clear by the identity of the variable "where it comes from", where it was "defined". Because of that, SSA can be seen as a generalization of so-called def-use chains. Since this valuable piece of data flow information is already built into the variables of the SSA, it's a good general starting point for all kinds of more specialized, subsequent analyses. For instance, reaching definitions.

Example in SLC

The core idea for using SSA is actually pretty simple, the following shows an example. If one intends that each "variable" is used only once, one simply has to use enough variables. If the program to start with violates the SSA assumption, like assigning to a variable more than once, one simple "renames" the variable, using a new variable each time an assignment targets the variable. We are using not the while language from before, but some intermediate language formate. It could be called *three address intermediate code*, because the basic data manipulation operation ("assignment") works with three "addresses": two source variables and the target variable. Of course, it's not 100% strict, also something like `x := y` is in three-address format, even if only two variables are involved.

What is *not* in 3AC format are assignment for the while language, which are of the form $x := a$, where a there is an *arithmetic expression*, which may contain arbitrary many variables.

A format like 3AC is some possible form in so-called *intermediate code*. So it's no longer user-level abstract syntax, but closer to a typical machine code level (which may be in 3A-format or 2A-format. There is also something called 1A-format, but that's not used for hardware, only for virtual machines and interpreters, for instance for byte-code (though historically there had been attempts in HW). Turning assignments of the general form of the while language to 3AC is not hard, and the translation resembles what needs to be done for SSA. Simply introducing new variables when one the 3AC cannot handle a compound expression. The additional variables are also called *tempory variables* or just *temporaries*. We illustrte the idea in some later slide.

If one wanted SSA, one could do the translation from user level syntax to SSA in one step. We take as starting point of our discussion a 3AC representation, like that the user syntax has already been simplifed to 3AC. That step, from user syntax to intermedate 3AC, typically would also replace conditionals, while loops etc. to (conditional) jumps, perhaps it would do something about scopes, since the intermediate code is supposed to be closer to the execution format.

Anyway, we don't fix the exact abstract syntax and semantics, and everything would work analogously for the while-language. Also the concept of control-flow graph applies to the intermediate code level, and indeed, often that's the level where a compiler applies the CFG as intermediate format.

At any rate: the "numbering" trick shown on the slides should be obvious. The subscripts here is just a way of systematically make clear, that \$b$_1$, b_2 etc. all have they origin in the variable b. This convention is helpful for reading such pieces of code; how it is implemented is a different issue, but using natural numbers for different versions of a variable is certainly an option.

3AC

```
a := x + y
b := a - 1
a := y + b
b := x * 4
a := a + b
```

3AC in SSA

- x and y: *input* variables, "read only"
- assigned to via initialization, *"before"* the program

```
a1 := x + y
b1 := a1 − 1
a2 := y + b2
b2 := x ∗ 4
a3 := a2 + b2
```

The transformation for straight-line code is *straightforward*. It's so simple that one could ask oneself: what's the big deal? Why is SSA said to be such an important format, if the whole thing seem pretty trivial? Well, as we mentioned a few time: for static analysis, straight-line code is not too complex. Actually, analysing straight-line code allows basically exact analysis, without approximation. There is no question whether a variable may or may not be used on the future: it *is* used in the future or else *it is not* (liveness or dead-ness of a variable is clear). Same for reaching definition etc. Trying to *exacly* exactly analyse straigh-line code and making some "optimal" compilation based on that is sometimes called *super-optimization*. That word is of course, doubly-stupid. Optimization, as we discussed, is mostly *not* finding an optimal program (being undecidable in general). To call the situation for SLC, where it may be optainable "super-optimal", i.e., better than optimal, is pretty silly...

Appel [2] states that it's a form of *value numbering*.

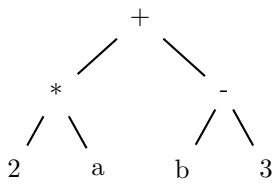
Basis idea (for SLC)

- later more complex
- for straight line code: simple "rename" the variables, like: use **different versions** x_1, x_2, x_3 for x on the left-hand side.

- some easy *data flow analysis* needed to get a fitting "versioning" for left-hand sides

Compare: 3AC (here for expressions)

`2*a+(b-3)`



Three-address code

```
t1 = 2 ∗ a
t2 = b − 3
t3 = t1 + t2
```

Compare to 3AC and temporaries

- code generation of 3AC for *straight-line code*
- three-address code: linear IR, close to machine code
- *restricted* right-hand sides
- *temporaries*: to store intermediate results
- often
  - temporaries = abstract form of register
  - unboundedly many asssumed

  $\Rightarrow$ each one assigned to only once

For the participants of the compiler construction course (INF5110), this comparison with the generation of 3AC should be familiar. But also without having participated, the idea is simple. Complex expressions are not supported by standard 3AC (as it's supposed to be close to machine code). That means they need to be broken into pieces and intermediate results need to be stored somewhere. For that purpose the compiler (at that stage) introduces "special" variables, special at least in the sense, that they don't show up in the source code. Otherwise, they are not too special anyway.

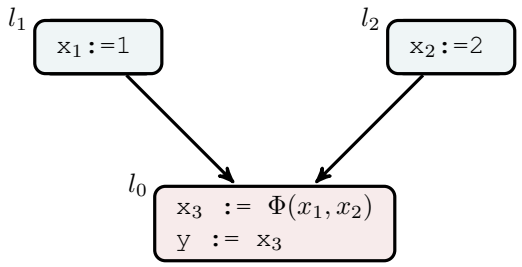
Since the compile can generate a new temporary when needed and since no upper bound on their number is assumed, that leads to a code in *static single assignment* form, at least as far as assignment to temporaries is concerned.

Static single assignment just pushes that idea a bit further and makes sure that also ordinary variable are assigned to once, only (at least "statically").

If we were sticking to straight-line code, without conditional jumps, conditionals, loops etc., everything would be rather simplistic and straightforward. The problem starts with conditionals. That will be treated next, and leads to the introduction of so-called Φ-functions.

Join points and phony functions

- simple illustration: two "definitions" of x



1. Phony functions Φ Assignments using functions like $\Phi(x_1, x_2)$ placed judiciously at (join) nodes to assure SSA format.

SSA in a nutshell

Transformation to SSA

- SSA = Φ + variable (re)naming scheme

Phony functions

- "non-standard" function
- encodes "control flow": value depends on if program "came from the left or from the right" in the last step
- Φ
  - "virtual", for purpose of analysis only, or
  - ultimately "real", i.e., code for Φ's will be generated

2 phase algorithm(s), in this order

1. strategical placement of Φ-functions
2. renaming of variables

- *main challenge*: **placement** of Φ

Brainless SSA form

- place Φ "everywhere"

Maximal SSA recipe

Placement: For all *variables*, at the beginning of each *join block* add

$$x \leftarrow \Phi(x, \ldots, x) \ ,$$

where number of x's is the number of predecessors of the node ≥ 2

Renaming: rename variables consistently (making use of *reaching definition* analysis)

- note: over-generous placement
  - guarantees *single-assignment* format
  - is **sound**

Room for improvements

- phony functions everywhere: sound but *wasteful* and generally undesirable
  - costly extra computations
  - subsequent analyses may suffer **loss of precision**

phony function unnessessary

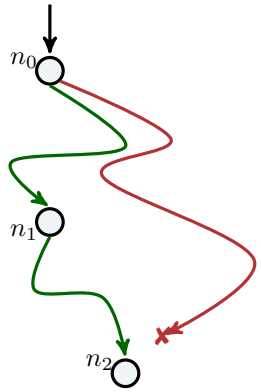
- result not *live*
- no real choice: $x_i \leftarrow \Phi(x_j, x_j)$

- in the following: efficient general algo for Φ-placement based on

dominance

Improvement

Core idea Assume assignment $x := e$ in n_1. If all paths from n_0 to n_2 **must** go through n_1, then n_1's assignment to x does **not** need to be covered by a phony function for x at n_2.



That covers the improvement mention before: if there is *no* actual choice involved: like in SLC, one does not really need a Φ-function. The discussion about dominance does not take care of whether the other optimization can be done, namely: if the target variable is not live, one should not bother assigning to it.

Domination

- CFG: directed graph with 1 entry (and 1 exit)
- "content" of the nodes / basic blocks irrelevant right now

Domination Node n_1 **dominates** n_2, if all paths from entry n_0 to n_2 must pass through n_1.

- $dom(n)$: *dominators* of n ("n being dominated")
- $dom(n)$: determined by a simple data flow analysis (*must* and *forward*)
- also: *strict* dominance

The previous argument pointed at situation, where one does not need a phone function for a variable. That's fine, but it's only half of the story, resp. one should think further: Placing Φ's everywhere *except* where there is exacly one origin (a situation of domination as in the previous illustration) this would be a *naive*. It's better than placing such function everywhere, but one can do much better.

Why is that? Assume a situation where there is *no* dominance. Let's take two additional nodes n_3 and n_4, additional to the previous picture, both containing another assignment to x and both "after" n_2 (which is dominated). We assume that these two nodes are *not* dominated by n_1. Obviously, that implies that n_3 and n_4 are *also* not dominated by n_2. At any rate, n_3 and n_4 are, SSA-wise, in an ambiguous situation wrt. x, therefore they are in need of disambiguation, i.e, some Φ-coverage.

Now, should one cover therefore both nodes by some phony function? That could be. However, in some situations we may do do better! Assume that somehow n_4 comes "after" n_3, in such a way that n_3 dominates n_4.

Dominance frontier

- we know, where **not** to put Φ for a given assignment at n = all nodes being dominated by n
- danger zone: "undominated", but where to put there?
- solution: **as "early" as possible** in the danger zone (thinking backwards)

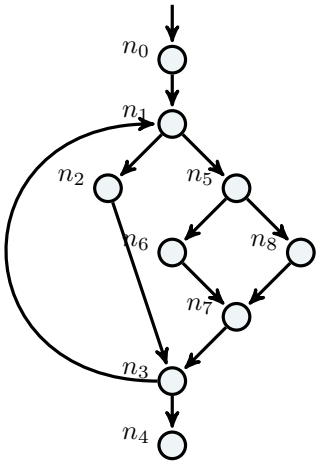
Dominance frontier The *dominance frontier* $df(n)$ of a node is the collection of m s.t.:

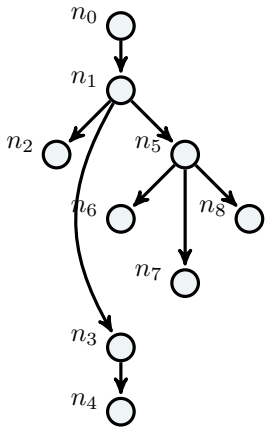
1. n dominates a predecessor of m ($q \rightarrowtriangle m$ and $q \in dom(n)$), and
2. n does *not* strictly dominate m.

- "strict" (non-)dominance condition & loops
- dominance frontier contains *join* nodes only

Dominator trees

- n's dominators: $dom(n)$
- n's **immediate** dominator $idom(n)$ (the dominator closest to n, if exists)





2.7.1 Value numbering

- we know, where **not** to put Φ for a given assignment at n = all nodes being dominated by n
- danger zone: "undominated", but where to put there?
- solution: as early as possible in the danger zone

Dominance frontier

```
for all nodes n in the CFG
    DF(n) := ∅

for all nodes  n
  if    n has multiple predecessors
  then for all predecessors p of n
       runner := p
       while runner ≠ idom(n)
       do    DF(runner) := DF(runner) ∪ {n}
             runner     := idom(runner)
```

The algo calculates, for each node, the dominance frontier $df(n)$. A node dominating another is a node "in the past", but the dominance frontier works forwardly, "to the future", calculating for each node, those other in $\rightarrow^*$ that are being dominated, or at least almost so. So, the nodes of $df(n)$ are *not* being dominated by n, i.e., it's not $n_2 \in df(n_1)$, then n_1 dominates n_2, it's that n_1 does *not* dominate n_2, but almost does, in that its (immediate) predecessor is being dominated. So, the dominator front describes some form of "front" for n but just one step beyond the front into the undominated territory. Remember that being dominated is the "safe" zone in that it does not need Φ-functions, whereas the undominated territory is the "danger zone" where Φ functions are necessary (for join nodes). And the frontier, one step beyond the border, is the place of the earliest convenience, where to put the Φ.

Now to the algorithm. The frontier is calculated for all nodes which have more than one predecessor (Φ-functions are needed for join-nodes only). The nodes are treated one by one.

The algo is simple enough, and needs both the original graph as well as the immediate dominator tree. First one takes all predecessors in the CFG of the chosen node (and there has to be more

than one, as we concentrate on join nodes). From each of those predecessors, we walk up the dominator tree. That walk is "deterministic", as we use the dominator *tree* upwards. Note that we are not only following the tree upwards, it's also like following the original graph backwards, only not step by step, but one dominator-edge may correspond to $\rightarrowtriangle^+$. In case of a loop in the program, the predecessor of n used as a starting point is, of course, both a predecessor as well as a successor of n (that's what makes it a loop). Therefore, following the dominator tree upwards ends up where the `runner` is immediatedly dominated by n (that's the while-loops exit condition). In that loop-situation, `runner` is a successor of n as well as a predecessor to n. Then we add n to all the "runners" (not the other way around). Note also that we climb up the tree until the `runner` equals the immidiate dominator. In case of a loop, that will lead to the situation that the entry-point of a loop dominates itself (which is consistent with the definition, in particular the second part which mentions *strict* dominance. That may be seen in the algo applied to n_1. Note again that applying the inner loop of the algo to a node do not contribute by calculating the dominator front for n, but rather the other way around: adding n to other nodes (represented by `runner`).

Example

	n_0	n_1	n_2	n_3	n_4	n_5	n_6	n_7	n_8
DF	$\emptyset$	$\{n_1\}$	$\{n_3\}$	$\{n_1\}$	$\emptyset$	$\{n_3\}$	$\{n_7\}$	$\{n_3\}$	$\{n_7\}$

Interesting may be the two nodes n_1 (and n_3, in connection with that). Note that n_1 is in the dominance frontier "of itself"? n_1 is the entry node of the loop, so this highlights the way who loops are treated. The basic intuition of dominance frontier is: go forward, and the frontier is all the nodes where one goes from "dominated" to "undominate" territory. That fine, but too simple for the loop-case. If we would apply this intuitition to n_1, the entry of the loop, of course n_1 dominates itself by definition. All paths to n_1 must somehow pass through n_1 (except one could argue about the word "pass *through*". The path may not acually go in an out of the node, they *end* in the node. That's why the second condition in the definition of dominance frontier task about "strict dominance". Here, n_1 dominates a predecessor of n_1 (namely n_3). A successor of n_3 is n_1, looping back. Now, moving from n_3 to its successor n_1 does not switch from dominated to undominated territory, it loops back to dominated one. Still, that does not make that place n_1 safe or unambiguous. The node can be reached inuitively in two different ways, resp. a variable inside n_1 can be influence by n_0 and by n_3, for instance. To cover such looping situations, one needs to talk about *strict* dominance in that part of the definition.

Further improvement

- left out here:
  - how to actually place the Φ
  - how to then rename the variables
- Basically: problem solved: place it at the dominance frontier
- we forgot: adding $x_i \leftarrow \Phi(x_j, x_j)$ is **another assignment** ...
- further improvements possible (liveness: focus on "global" names)

Renaming

```
rename (b)
  for each Φ–function ``x ← Φ(...) in b
    rewrite x as newname(x)
```

```
for each operation x ← y ⊕ z
    rewrite y with subscript top(stack[y])
    rewrite z with subscript top(stack[z])
    rewrite x as newname(x)
for each successor of b in the CFG
    full in Φ–function parameters
for each successor s of b in the dominator tree
  rename s
for each operation ``x ← y ⊕ z'' in b
  and each Φ–function ``x ← Φ(...)''
    pop(stack[x])
```

Bibliography

[1] Alpern, B., Wegman, M. N., and Zadeck, F. K. (1988). Detecting equalities of variables in programs. In [8], pages 1–11.

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

[3] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. (1991). Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490.

[4] Kildall, G. (1973). A unified approach to global program optimization. In *Proceedings of POPL '73*, pages 194–206. ACM.

[5] Lots of authors (2015). SSA book. `http://ssabook.gforge.inria.fr/latest/book.pdf`. The book is available online and under work.

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

[7] Nielson, F., Nielson, H.-R., and Hankin, C. L. (1999). *Principles of Program Analysis*. Springer Verlag.

[8] POPL'88 (1988). *Fifteenth Symposium on Principles of Programming Languages (POPL)*. ACM.

[9] Rosen, B. K., Wegman, M. N., and Zadeck, F. K. (1988). Global value numbers and redundant computations. In [8], pages 12–27.

Index