


# Course Script

## Static analysis and all that

IN5440 / autumn 2024

Martin Steffen

# Contents

**2 Data flow analysis** **1**
2.1 Intraprocedural analysis . . . 1
2.1.1 Determining the control flow graph and further auxiliary constructions 1
2.1.2 Available expressions . . . 4
2.1.3 Reaching definitions . . . 8
2.1.4 Very busy expressions . . . 9
2.1.5 Live variable analysis . . . 13
2.2 Theoretical properties and semantics . . . 15
2.2.1 Semantics . . . 15
2.2.2 Intermezzo: Lattices . . . 20
2.2.3 Correctness of analyses . . . 28
2.2.4 Rework . . . 34
2.2.5 Correctness of the liveness analysis . . . 34
2.3 Monotone frameworks . . . 37
2.4 Equation solving . . . 40
2.4.1 Chaotic iteration (repetition) . . . 40
2.4.2 The idea of a worklist data structure . . . 41
2.4.3 Input, output of the worklist algorithm, and other data structures . 42
2.5 Interprocedural analysis . . . 45
2.5.1 Introduction . . . 45
2.5.2 Extending the syntax . . . 46
2.5.3 Extending the semantics and the CFGs . . . 47
2.5.4 Naive analysis (non-context-sensitive) . . . 55
2.5.5 Taking paths into account . . . 60
2.5.6 Context-sensitive analysis . . . 65
2.6 Static single assignment (not pensum) . . . 83
2.6.1 Introductory remarks . . . 83
2.6.2 A trivial starting point: straight-line code (SLC) . . . 86
2.6.3 Join points and phony functions . . . 89

# Chapter 2
# Data flow analysis

## 2.1 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 **intra**procedural 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.[1] 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.

### 2.1.1 Determining the control flow graph and further auxiliary constructions

This section shows how to turn an abstract syntrax tree into a CFG. The starting point is the (labelled) abstract syntax from the introduction. Actually, we use the **labelled** syntax of the **while language**, where the labels correspond to the **nodes** of the CFG

Given a program in labelled 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:

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

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

[1] For functional languages with higher-order functions, things gets more involved.

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

(2.2)

The label $init(S)$ 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 guaranteed, 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

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

works as follows:

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

We make use of two further (very easy) functions with the following types

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

They are defined straightforwardly as follows:

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

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

All the definitions and concepts are really straightforward and should be intuitively clear almost without giving a definition. 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.

#### 2.1.1.1 Flow and reverse flow

Data flow analyses can be a **forward** analysis (like RD) or a **backward** analysis. Forward analyses work on *flow*, the "standard" control-flow graph. For backward analysis, one uses the edges backwards, we call it the **reverse** flow $flow^R$. When A standard implementation for control-flow graphs uses **pointer** to represent the edged, i.e., each node contains a list of pointer connecting it to its names. To be prepared to all kinds of data-flow analyses, it's usefull to choose a "**doubly-linked**" representation, so where each node points to is successors or "forward-neighbors" as well as its predecessors or "backward-neighbors".

#### 2.1.1.2 Program of interest

For precision, we introduce some notations. We write $S_*$ for the program being analysed, i.e., the top-level statement that we are interested in. Analogously $\mathbf{Lab}_* \subseteq \mathbf{Lab}$ represents the labels from the concrete program under consideration. The set **Lab** of labels in general can be seen as the *data type of labels*. For instance in an implementation one could introduce `type lab = int;` or similar. In contrast to **Lab**, $\mathbf{Lab}_*$ is **finite**, and that's important for the analysis to terminate. Analogously we use $\mathbf{Var}_*$, $\mathbf{Blocks}_*$ etc. for the set of variables, blocks etc. from the concrete program. Finally, we write $\mathbf{AExp}_*$ for the set of **non-trivial arithmetic sub-expressions** of $S_*$, analogous for $\mathbf{AExp}(a)$ and $\mathbf{AExp}(b)$, where *trivial* expressions are single variables or constants.

Often one makes restrictions or assumption over the form of the CFG.

**Definition 2.1.1.** A control-flow graph has an **isolated entry** if $(l, init(S_*)) \notin flow(S_*)$ and **isolated exits** if $\forall l_1 \in final(S_*). \quad (l_1, l_2) \notin flow(S_*)$. The graph satisfies **label-consistency**, if

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

Often one assumes **isolated** entries (and exits). This is not really a restriction: a program consisting 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. And unique labelling is the way one builds the control-flow graphs.

**Remark 2.1.2** (Unique and consistent labelling)**.** *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.* □

### 2.1.2 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. It's kind of like "memoisation" for expressions.

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.

A usage of the available expression analysis is to avoid recomputation.

*Example* 2.1.3 (Available expressions). Consider the following very small "program".

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

There are different occurrences of the compound expression $a + b$. There are two occurrences where the expression is used as right-hand side of an assignment, and in both cases the expression is stored in $x$. A question one could ask is whether the value of $a + b$ is available at node 2, in the condition of the while-loop. If so one could simply look up the value in $x$ (in this case where it's stored in $x$, otherwise one could arrange for an extra variable to store it.)

The answer in the example is "yes", $a + b$ is available at node 2. Note that it does not mean its value is determined. The value of $a + b$ may come from node 0, or from node 4 (and anyway, if to comes from node 4, since it's in a loop, the value may change with each iteration). □

**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 the situation for reaching definitions, which is about whether a "definition" **may** reach a point in question. 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). For must-analyses, if we have some safe information (like here the set of available expressions), then a *smaller* set it safe, too. 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.

#### 2.1.2.1 Available expressions: general

We will present the available expressions as flow **equations**, i.e., as equational constraints and $\subseteq$-constraints. But that is not really crucial, as we know already. We will also present the details and ingredients of the analysis in a specific order, one that we will repeat when afterwards presentin other analyses. This will show the similarities between the different analyses as well as the differences. Since all analyses basically work the same (just on different kinds of information and ingredients), one can understand them all as special cases of a more general data-flow framework, the so-called monotone framework.

As we know already from the introduction, one can distinguish between **intra**-block flow and and **inter**-block flow, i.e., data flow through basic blocks or nodes of the CFG and data flow between the blocks, along the edge.

Unlike in the introduction, we here will charaterize the **intra**-block flow to consist of **two** ingredients. The flow information that is being **generated** by a block, and flow

information that is being eliminated, one says **killed** by the block. The overall effect of a block then is defined as the combination of the generate and the kill contributions.

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.

Now, let's fix the information of interest, i.e. the data *types* for the information that flows through the control-flow graph, resp. that will be in the solution(s) for the constraint systems. We are interested in interested in **sets of expressions** of type $2^{\mathbf{AExp}_*}$. Consequently, the kill and generate function will have the following types

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

and the result of the analysis, i.e., solutions to the constraint system can be seen as a pair of functions (or an array)

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

**Definition 2.1.4** (Intra-block flow specification: Kill and generate)**.**

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

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

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.

Now finally the flow equations for the analysis, split into those for nodes and edges: The **intra**-block equations per node use the *kill* and *generate* defined earlier, thhe **inter**-block equations use *flow* (representing the edges) and $\bigcap$.

**Flow equations for $\mathsf{AE}^=$**

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

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

where $B^l \in blocks(S_*)$ □

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 $\mathsf{AE}_{entry}$, 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 otherwise.

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.

The structure of the cfg (without the content of the nodes) is the same as for the factorial. The kill- and gen- is given at page 40. Interesting is node with $l = 3$. It kills especially $a + 1$. Otherwise it is pretty unspectacular.

*Example* 2.1.5 (Available expressions). Consider the following code snippet, with labelled abstract syntax.

$$[x := a + b]^0; [y := a * b]^1; \quad \texttt{while} \quad [y > a + b]^2 \quad \texttt{do} \quad ([a := a + 1]^3; [x := a + b]^4) \tag{2.6}$$

The corresponding control-flow graph is shown in Figure 2.1 2.1. Worthwhile is (for instance) to look at is 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. □

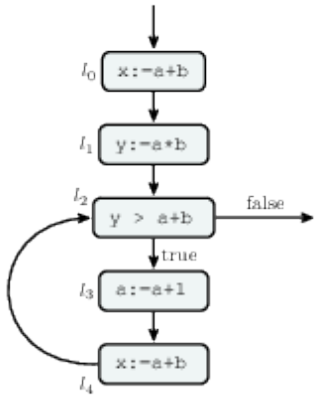



Figure 2.1: Available expressions example: CFG

### 2.1.3 Reaching definitions

This is basically a repetition of the introduction, it's the **same** analysis, only phrased with the help of the new definitions, like the kill, and generate functions. So we also don't recap the motivation of reaching definition analysis, we assume that know here.

*Example* 2.1.6 (Reaching definitions). Let's look at the following piece of code for a simple example.

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

The corresponding CFG is given in Figure 2.2.

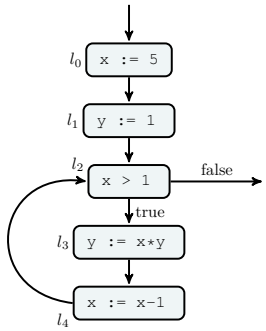



Figure 2.2: Reaching definitions: Simple CFG

The information of interest for reaching definition in are **sets of tuples of var's and labels**, i.e, the information is of **type** $2^{\mathbf{Var}_* \times \mathbf{Lab}_*^?}$, where $\mathbf{Lab}_*^? = \mathbf{Lab}_* + \{?\}$. Consequently, the kill and generate functions have the following types.

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

and the result of the analysis, i.e., solutions to the constraint system can be seen as a pair of functions (or an array)

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

As far as the mappings or functions $\mathsf{RD}_{entry}$ and $\mathsf{RD}_{exit}$ 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.

**Definition 2.1.7** (Intra-block flow specification: Kill and generate)**.**

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

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

As in the AE analysis, the most interesting case is again the one for assignments. 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).

Now the flow equations for the analysis, split as before into those for nodes and edges: The **intra**-block equations per node use the *kill* and *generate* defined earlier, thhe **inter**-block equations use *flow* (representing the edges) and $\bigcup$.

**Flow equations for** $\mathsf{RD}^=$

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

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

where $B^l \in blocks(S_*)$. Note that in the flow-equation, the order of the kill and generate function is the same: "first" remove the information being killed, "afterwards" add the information being generated.

### 2.1.4 Very busy expressions

Here is yet 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 (resp. the value of 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's now 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.

Here is a small example.

*Example* 2.1.8 (Very busy expressions). The example is very unspectacular, there is two branches each of which uses the expressions $a - b$ and $b - a$, so both expressions are very busy at the entry of the conditional.

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

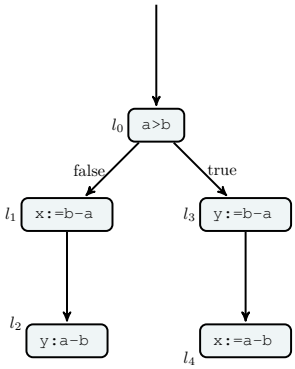



Figure 2.3: Very busy expressions example: CFG

□

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.

**Definition 2.1.9** (Very busy expressions)**.** 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).

**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 principle 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 exists, even if sometimes they are indicating a problem (the program "hangs").

The information of interest for very busy expressions analysis are **sets of expressions**, more precisely, the information is of **type** $2^{\mathbf{AExp}_*}$. Consequently, the kill and generate functions have the following types.

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

and the result of the analysis, i.e., solutions to the constraint system can be seen as a pair of functions (or an array)

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

**Definition 2.1.10** (Intra-block flow specification: Kill and generate)**.**

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

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

A comparison with the kill and generate functions for $\mathsf{AE}$ might be interesting (cf. Definition 2.1.4). 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 obviously are 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.

**Definition 2.1.11. Flow equations:** $\mathsf{VB}^{=}$

$$\begin{aligned} \mathsf{VB}_{exit}(l) &= \begin{cases} \emptyset & l \in final(S_*) \\ \bigcap\{\mathsf{VB}_{entry}(l') \mid (l', l) \in flow^R(S_*)\} & \text{otherwise} \end{cases} \qquad (2.7) \\ \mathsf{VB}_{entry}(l) &= \mathsf{VB}_{exit}(l) \setminus kill_{\mathsf{VB}}(B^l) \cup gen_{\mathsf{VB}}(B^l) \end{aligned}$$

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 2.1.8 was very small and simple, and the next one is not much larger. But it shows some fine-print of the very-busy expression analysis, that has to do with (potentially) infinite loops.

*Example* 2.1.12 (Infinite loop). Let's have a look at the example from Figure 2.4. Since

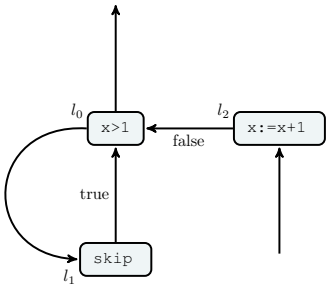



Figure 2.4: Infinite loop (CFG with reversed edges)

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

The looping example is quite instructive or puzzling. 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 possibility does not involve making an argument about infinite paths. Unlike the situation of the very busy expressions, liveles is intuitively plausible. □

### 2.1.5 Live variable analysis

This analysis focuses on *variables* again (not on expressions). A variable at some point (resp. the value it contains) is live, if it **may** be used in the future (without being overwritten) If we use "dead" for being not live, a variable is dead intuitively if its value will definitely ("**must**") not be used in the future. This will 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"). And an important usage of live variable analysis is in **register allocation**, making better use of registers.

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.

> **Liveness**: 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). The **goal** therefore is: determine for each program point: which variables are live at the exit of that point.

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.

The information of interest for live variable analysis are **sets of variables**, more precisely, the information is of **type** $2^{\mathbf{Var}_*}$. Consequently, the kill and generate functions have the following types.

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

and the result of the analysis, i.e., solutions to the constraint system can be seen as a pair of functions (or an array)

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

**Definition 2.1.13** (Intra-block flow specification: Kill and generate)**.**

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

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

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.

**Definition 2.1.14. Flow constraints:** $\mathsf{LV}^{\supseteq}$

$$\begin{aligned} \mathsf{LV}_{exit}(l) &\supseteq \begin{cases} \emptyset & l \in final(S_*) \\ \bigcup\{\mathsf{LV}_{entry}(l') \mid (l', l) \in flow^R(S_*)\} & \text{otherwise} \end{cases} \qquad (2.8) \\ \mathsf{LV}_{entry}(l) &\supseteq \mathsf{LV}_{exit}(l) \setminus kill_{\mathsf{LV}}(B^l) \cup gen_{\mathsf{LV}}(B^l) \end{aligned}$$

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".

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

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*).

*Example* 2.1.15 (Looping example). Let's revisit the CFG from Example 2.1.12 and Figure 2.4. □

## 2.2 Theoretical properties and semantics

### 2.2.1 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 guess work, the implementation, or the platform is not 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** operational semantics. Perhaps later, for the more complex functional languages, the lecture may cover some variations.

#### 2.2.1.1 States, configurations, and transitions

Before we can write down the operational semantics as a set of rewriting rule in Table 2.1, we need to fix a few things. The rules of the semantics will specify for each construct what happens when executing that construct. Generally, executing a command means the executing program "moves forward". When thinking in terms of, say, machine instructions are executed, the instruction pointer points to the instruction being executed next, and executing it changes the instruction pointer or program counter. For most instructions, the instruction pointer afterwards points to the subsequence instruction. That's the case for instructions that don't do (conditional) jumps.

Besides changing the instruction pointer, i.e. affecting the "control", instructions can access and affect the memory ("data"). That includes reading and writing from memory.

Our semantics does not work on the level og machine code, it works with a structured programming language (though admittedly a very small and simplistic one). Still, the operational semantics needs to cover both "control" and "data" aspects of what's going on when a program runs.

The way the semantics is formulated is, that it works directly on the (abstract) syntax and evolving and transforming the syntax itself as a way of "interpreting" it. So instead of having a instruction pointer moving through the code to the indicate where the "control"

currently is, the syntax evolves, and the current state of control alway refers to all of the current syntax, the rest of the program that still needs execution.

What is called a **state** of a program generally refers to the current content of its "memory" (the data part). In our language is stored in **variables** so the state can be seen as a *mapping* or fuction, assigning values to the variables of the given program. The overall "status " of a running progam at a given point in time, consisting of "data" and "control" is called here a **configuration.**

The semantics will use the following notations. **states**, as said, are mappings, and we use $\sigma$ as symbol for states. In math-notation, we write $\sigma : \mathbf{State} = \mathbf{Var} \to \mathbf{Z}$, which just that. The type for states is **State** which is the same as $\mathbf{Var} \to \mathbf{Z}$, and typical elements are written as $\sigma$ (or $\sigma'$, $\sigma_1$ ...)

Operational semantics is about descibing the steps a running program does, and a small steps operational semantics descibes the next indivdual step of a program, one step after the other. It describes the semantics as **transitions**, which in our case are of the form:

$$\langle S, \sigma\rangle \to \langle \acute{S}, \acute{\sigma}\rangle \quad \text{or} \quad \langle S, \sigma\rangle \to \acute{\sigma}\ . \tag{2.9}$$

So transitions go from configuration to configuration, and (in this presentation) there are "special" transitions that go from configuration to a state. That's for the special case of termination, which means the control component has been executed and there is no code $S$ left to be executed.

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

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

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

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

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

Table 2.1: Structural operation semantics: rules for the while language

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.

One ingredient of the operational semantics is still unexplained. That's the treatment of *expressions*. For dealing with the control-flow structure of the while language, we have operational rules to describe transitions between configuration. Expressions are **side-effect free** and have no control-flow aspects.[2] we therefore give the meaning or semantics of expressions "directly", by a

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

The semantics for expressions is really almost the simplest imaginable. That's because we have just side-effect free expressions, we assume tha there no *errors* or *exceptions*. The semantics for arithmetic resp. boolean expressions is given in equation (2.10):

$$\begin{array}{rcl} [x]^{\mathcal{A}}_{\sigma} & = & \sigma(x) \\ [n]^{\mathcal{A}}_{\sigma} & = & \mathcal{N}(n) \\ [a_1\ \mathtt{op}_a\ a_2]^{\mathcal{A}}_{\sigma} & = & [a_1]^{\mathcal{A}}_{\sigma}\ \mathbf{op}_a\ [a_2]^{\mathcal{A}}_{\sigma} \\ \\ [\mathsf{not}\ b]^{\mathcal{B}}_{\sigma} & = & \neg [b]^{\mathcal{B}}_{\sigma} \\ [b_1\ \mathtt{op}_b\ b_2]^{\mathcal{B}}_{\sigma} & = & [b_1]^{\mathcal{B}}_{\sigma}\ \mathbf{op}_b\ [b_2]^{\mathcal{B}}_{\sigma} \\ [a_1\ \mathtt{op}_r\ a_2]^{\mathcal{B}}_{\sigma} & = & [a_1]^{\mathcal{A}}_{\sigma}\ \mathbf{op}_r\ [a_2]^{\mathcal{A}}_{\sigma} \end{array} \tag{2.10}$$

The semantics has the following property, that one should expect, namely that variables that don't occur (freely) in the expression don't play a role:

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

In the introductory remarks, we mentioned that we will do some specific form of semantics, namely an **operational semantics**. That's not 100% true. 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.

[2] For instance, the logical operators like $\wedge$ and $\vee$ are not evaluated using short-circuiting, as is done in many programming languages. That would imply, that for instance for an `or`, if the first argument evaluates to true, the second argument is not evaluated, i.e., jumped over.

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).

**Remark 2.2.1** (Denotational semantics)**.** *The semantics of expressions from Equation 2.10 seems trivial or at least straightforward. Basically it sais that + is interpreted as +, compound expressions are treated recursively, and for variables, the state needs to be consulted. It's not an* operational *way of specifying the semantics and one could make the argument that for a language like, the meaning of arithmethic or logical expressions would need no elaboration at all. Fair enough. If, however, the operational semantics can be seen as a description of an interpreter, and if one wants to implement it, then one also needs to implement what should happen with the expressions. Secondly* □

#### 2.2.1.2 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 2.2.2** (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.

### 2.2.2 Intermezzo: Lattices

This section provides some foundations on which data flow analysis rests, in particular, one on 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 order relations 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 2.2.3** (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 order. 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 anti-symmtetry 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. In the following, if comparing things,

we will sometimes in the English text say things like "$a$ is larger than $b$" etc. We mean "larger" to represent $\sqsubseteq$, but we not always 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, we have to be careful when saying "upper and lower bounds exist" (let's say for $(\mathbb{N}, \leq)$. Certainly, if we take **two** numbers $n_1$ and $n_2$ as above, those bounds exist; there 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 don't exist infinite set of elements from $\mathbb{N}$ that actually has an upper bound.

If we switch to $(\mathbb{Z}, \leq)$ instead, some infitite sets *do* have upper bounds (not all though). If a given set $A' \subseteq \mathbb{Z}$ has upper bounds, then it's guaranteed that there is a smallest or least upper bound. When switching 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 conventionally uses min and max to refer to those. The uniquens of least upper bounds is ultimately a consequence of *antisymmetry* of $\leq$. This is actually 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, 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 *sets*, 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 2.2.4** (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 exist. 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 exists, 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.

Here we are working with a specific form of lattice, called *complete* lattice. It's a very nicely behaved lattice, which makes it useful for the monotone framework. The important condition, which makes the lattice actually *complete* is rather hidden in the above definition. It's that meets and joins exists *for all subsets*. If we drop the "completeness", then one would still require joins or "least upper bounds" $a \sqcup b$ and or "greatest lower bounds" $a \sqcap b$ to exist. The difference here is we speak about *binary* such operations. To be complete, it's also required the meets and joins exists also for *infinite sets*. A priori a lattice is not required to be finite (and many interesting ones are not). For those situations, the existance of *binary* meets and joins implies the existance of *finite* meets and joins (those can always be expressed by a number of binary ones). What is *not* guaranteed it the existance of *arbitrary* meets and joins, including infinite ones. But that's required for a lattice to be complete. 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

Now we talk about **fixpoints** and related concepts. Actually, fixpoints were mentioned already shortly in the introduction, and now we look deeper and in particular connect it to lattices and monotinicity, resp. complete lattices and continuity.

**Definition 2.2.5** (Fixpoint)**.** Assume a function $f : L \to L$. An element $a \in L$ is a **fixpoint** of $f$, if

$$f(a) = a \ .$$

Assume now that $L$ is partially ordered. Then an element $a$ is a **pre-fixpoint**, resp. a **post-fixpoint** of $f$, if

$$f(a) \sqsubseteq a \quad \text{resp.} \quad f(a) \sqsupseteq a \ .$$

Let's $\mathit{Fix}(f)$ denote the set of $f$'s fixpoints:

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

Alternatively we call $f$ *reductive* at $a$, if $a$ is a pre-fixpoint of $f$ and $f$ *extensive* at $a$, if $a$ is a post-fixpoint and $\mathit{Red}(f)$ resp. $\mathit{Ext}(t)$ denotes the sent of pre-fixpoints resp. the set of post-fixpoints of $f$:

$$\mathit{Red}(f) = \{a \mid f(a) \sqsubseteq a\} \quad \text{resp.} \quad \mathit{Ext}(f) = \{a \mid f(a) \sqsupseteq a\}$$

**Definition 2.2.6.** Assume a complete lattice $L$, and a function $f : L \to L$. Then we define $\mathit{lfp}(f)$ and $\mathit{gfp}(f)$ as follows:

$$\mathit{lfp}(f) \triangleq \bigsqcap \mathit{Fix}(f) \quad \text{and} \quad \mathit{gfp}(f) \triangleq \bigsqcup \mathit{Fix}(f) \tag{2.11}$$

Equation (2.11) 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 $\mathit{lfp}(f)$ and $\mathit{gfp}(f)$ 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 function $f$.

But that requires a separate *argument*. The definition just fixed the greatest lower bound resp. least upper bound of all fix-points. But who says that those elements are *themselves* fixpoints? Actually, without assumptions on $f$, they are not necessarily. One additionally needs *monotonicity* of $f$!

If one would carry out the argument, i.e., the proof, that all fits together in the sense that the $\mathit{lfp}(f)$ and $\mathit{gfp}(f)$ 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 lattice **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 in the presented way, 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

The result that connects complete lattices, monotone functions, and the existance of fixpoints is known as Tarski's theorem.

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

That's captured in the following theorem due to Tarski. In the previous wording, it was said that $\bigsqcap Fix(f)$ is a fixpoint in itself. That's correct, but the theorem that follows is stated slightly more general, namely using $Red$ and $Ext$, i.e., pre- and post-fixpoints:

> **Theorem 2.2.7.** *$\mathcal{L}$: complete lattice, $f : \mathcal{L} \to \mathcal{L}$ monotone.*
>
> $$\begin{array}{lcll} lfp(f) & \triangleq & \bigsqcap Red(f) & \in \quad Fix(f) \\ gfp(f) & \triangleq & \bigsqcup Ext(f) & \in \quad Fix(f) \end{array} \tag{2.12}$$

Note: *lfp* (despite the name) is *defined* as the *greatest* (!) lower bound of all pre-fixpoints. the theorem (more or less directly) implies *lfp* is not just a fixpoint, but it's the **least** fixpoint.

### Fixpoint iteration and Kleene

Kleene's fixpoint theorem is imporant, as it guarantees the unique **existance** of fixpoints. That's good to know, but the existance of concepts or solutions is more of interest for mathematician. For computer scientists, a relevant question is: can I compute it? Not everything that can be proven to exist can be computed. And if something is computable, the next question is: what's the complexity to compute it. That's where **Kleene** "improved" on Tarski's existance theorem. We also need to **strengthen** the assumptions a tiny bit. It's not enough that $f$ is monotone, it needs to be **continous** (which implies monotonicity).

Let's first remind ourselves or fix what monotonicy means (something that we need already for Tarski).

**Definition 2.2.8** (Monotonicity)**.** A function $f : L \to L$ (on a partially ordered set) is monotone, if

$$a \sqsubseteq b \quad \text{implies} \quad f(a) \sqsubseteq f(b) \tag{2.13}$$

Before we come to continuity, let's first speculate of how to *compute* a fixpoint, let's say the smallest? Tarski's theorem does not help us much, we cannot first calculate $\mathit{Fix}(f)$ (or $\mathit{Red}(f)$ etc.) and then build the greatest lower bound, since the set may be infinite. And even if not in practical situations, it does not seem an **effective strategy**: inspect all elements in a collection and the find the smallest one (or the smallest lower bound here, which happest to be the smallest in the collection). Better is to just randomly goes through all elements, but to exploit the lattice structure. That is done by *iteration*, here *fixpoint iteration* or *fixpoint approximation*. It goes like this: we start at the lowest element of the lattice (which exists uniquely), apply $f$, then apply it again, and again:

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

Each element in the sequence is larger or equal than the previous (thanks to monotonicity). As a notation, the sequence could be written $(f^n(\bot))_n$ (with $n \in \mathbb{N}$). What one can prove is the following (the equations show the situations both for greatest and smalles fixpoints):

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

Note that we write "$\dots \sqsubseteq \mathit{lfp}(f)$", not "$\dots = \mathit{lfp}(f)$". Which means we are not assured to "reach" the fixpoint ("within" $\omega$). Note also that $\bigsqcup_n f^n(\bot)$ builds the least upper bound over a infinite set, but even that (though it exists) does not guaranteed that we have "reached" the least fix-point. We get closer and closer to it, and even if we do that approximation infinitely long, so we get "arbitrarily close" we don't reach it. To assure that the limit (as one can call it) of all approximation **is indeed** the fixpoint, we need one last condition on the lattice-plus-function setting. That is **continuity**.

Continuity is in a way a "infinite version" of monotonicity (if one considers complete lattices). To see that we can consider first the finite (or binary) situation. Assume we have

a montone function on not just a partial order, but on a lattice. What's the consequences. Under this assumption, we know, that $a \sqsubseteq b$ implies $a \sqcup b = b$. With that awe have for $a \sqsubseteq b$ that $f(a \sqcup b) = f(b) = f(a) \sqcup f(b)$, and the latter equation holds since $f$ is monotone which implies that $f(a) \sqsubseteq f(b)$.

So we simply need to generalize the observation to infinite situations and that gives the definition of continuity. For that we need the concept of ascending chains, which generalizes $a \sqsubseteq b$ to an "infinite situation".

**Definition 2.2.9** (Continuity)**.** Assume a complete lattice. An **ascending chain** is an sequence of elements $l_0, l_1, l_2, \ldots$ where $l_i \sqsubseteq l_{i+1}$. A function $f : L \to L$ is **continuous** if for all *ascending chains* $(l_n)_n$:

$$f(\bigsqcup(l_n)_n) = \bigsqcup(f(l_n))_n \tag{2.14}$$

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

### 2.2.3 Correctness of analyses

This section wants to shed light on how to establish the correctness of analyses. We do that in a general manner, to see the **ideas behind** such arguments, and show also a concrete setting, namely *liveness variable analysis*. Semantic analysis is a static **prediction** obtained from the source code (abstract syntax) on what happens when a program runs. This prediction is necessarily approximative, and we insist on that the analysis is a *safe* approximation (either an over- or an under-approximation).

We are currently doing data flow analyses. For example live variable analysis. That involves constraints (intra- and inter-block constraints) that expesss our understanding of the problem per statement and per edge, for instance the constraint system from equation (2.8) for live variable analysis). For a given program, that results in a constraint system and we expect that **solutions** to that equation system are **safe**. The constraint system provides approximative solutions, which means, that non-solutions, i.e., "results" that violate the equation system, are necesserily unsafe for the given program. But that's not our concern.

As the set of solutions (as well as the overall solution space) is lattice-shaped, there is a unique *best* solution, and we also know that it can be effectivel computed, by fixpoint approximation.

In order to have a firm basis to judge whether a given analysis is correct or not, we took the effort to define a small-step structural operational semantics as a reference points. But there is still the question: How do we convince ourselves, that, given the semantics, all the solutions to the constraint system are safe approximations, i.e., the predict something which is true about the behavior of a program, and of course for *all* programs.

Being safe, as said, is a prediction on the future behavior of the program, and we have to check, whether the prognostication is true, or, to be more precise, whether it's not violated. For instance, if reaching definition claims that a definition of variable $x$ at a

particular label reaches the entry of another label, we don't establish that this is actually the case at run-time (because it might not be), but rather that's never the case that a definition in a running program never reaches a label without the analysis has actually predicted it.

So we need to establish that when running the program (= executing the operational semantics) we don't run into a negative surprise, a situation the analysis had assured us would never occur. That could be called a **run-time analysis error** in analogy to the more familiar term **run-time type error**.

For standard type systems, the situation is analogous to here. To illustrate how to argue for the safety of an analysis, is easier and maybe intuitive for standard type systems. It's more intutive because type system are there to prevent certain kinds of errors, and running into such an error for a well-type program is a violation of the prediction of the type-system.

So for a static type systems to be a safe approximation (or to be safe or correct for short) means if the static type system judges the program as type safe, then running it does not lead to a to a **run-time type error.** What's that? That's a violation of the prognostication and promise the type system gave, the "negative surprise" we mentioned. What's the promise a type system does? It stays, if a program has a type $T$ and it terminates, then the resulting or return value is of that type. For instance a procedure type $\mathsf{string} \to \mathsf{int}$, if provided with an argument of type string[3], then, *if* it terminates, it returns a *value* of type $\mathsf{int}$. It's a promise about the result, and not about "what happens in between".[4]

Or does it? The type system makes a promise that the procedure from the example returns an integer value, i.e. a element from $0, 1, 2, 3, \ldots -1, -2$ if it terminates. But that implies also a promise about absence of errors *while evaluating*, at least implicitly. More precisely absence of run-time type errors. What is that? That's a situation when a *value* is treated "the wrong way" or misused. Our while-language that we use for data flow analysis is too simplistic to demonstrate a run-time type error, no program has typing issues. But if a program in a more complex language supporting also strings would try to execute `x := "abc"`, when the static type considers $x$ of type integer, bad things would happen. In a compiled language, the string `"abc"` may just not fit into the memory space reserved for integers. If the run-time system of the compiler would not care about that but stores `"abc"` anyway at the address meant for `x`, it may very well overwrite other data (ultimately everything is bits at addresses) and that may went unnoticed. Unnoticed in the sense that the program runs smoothly, only some data is corrupted. Not necessarily the one is connected to $x$, but some unrelated one, that the compiler happen to have allocated in the neighborhood of $x$'s place. And then the results may simply be wrong, and it's hard to pinpoint the error and maybe often it's correct, as it depends on the memory placements, something that the user is not aware of. So we have random errors, and ones that may not easy to be reproduce or tested against.

[3] the argument does not need to be a value of type string, i.e., it does not have already been evaluated to a string.

[4] Non-standard type systems may have an effect-parts, that would capture that and also data-flow analyses are not concerned with the outcome, but with things that happen during evaluation.

Such a laissez-fair approach is not considered state of the art. What the run-time system could do instead is: when performing the the assignment `x := "abc"`, it checks the type of the value (a string) and if $x$ is of type integer, it refuses to store the string and raises a **run-time type error** instead. That terminates the program. Note that the check is not done when assigning `x := p(x+546) * 12`, simply because the expression on the right hand side is not yet evaluated. The run-time type check (for assignments) would only be done before the individual step that assigns the value to $x$. Type checking $p(x + 546) * 12$ at run-time would take too much time, and anyway would be more of a static type check, as it is a prediction of the outcome of the expression should the calcuation halt. Doing a run-time type check on `"abc"` is **not a prediction** ("the result if even will be an integer"), it is **"fact-checking"**: "the value behing handled in the next step **is** in fact a string.".

Since run-time type errors terminate programs, a judgement like program $p$ is of type $\mathsf{int}$ not just means that the outcome (when terminating) is in integer, it also says that when calculating it, there won't be **no run-time type errors** when evaluating $p$, but that would derail the program. That's because the type system analyzes the overall program, and checks if all fits together.

That does not mean that $p$ actual terminates or terminates properly. It could be that it diverges, and it could be that there are other kinds of errors during evaluation the type system has no predictive power over, resp. which are not covered by run-time type checks. Maybe a division-by-zero error or stack overflow. Also those sitations terminate the program, but that does not count as proper termination, but an exceptional one, and it's not considered that a value like an integer or similar is returned.

So, back to the question how to establish safety or correctness of the type system? In the type-oriented setting, how to we establish that the type system guarantees that a well-typed program does not run into or through run-time type errors?

The argument is very straightforward. We have given our small-step operational semantics.[5] Now, the previous illustration about discussed type errors partly used arguments refering to actual implementations and what would happen if for instance one would store a string into a place fitting for integers. Small step semantics are on a more higher-level of abstraction, they work more like an interpreter. The are called *structual* operational semantics, as the use transforming actual *syntax*, the structure of a program, to do or specify individual steps.

Let's assume we have a purely functional language (just so that we don't need to have to involve states $\sigma$ in the argument). If we say a program or expression $e$ is well typed with a type $T$, say an integer type, it's written as judgment $\vdash e : \mathsf{int}$ (let's assume the expression is closed and thus we need no context $\Gamma$). That means if we evaluate the $e$ (and you terminate), the result will be a number, i.e., the situation can be described as $e \to^* v \in \{0, 1, 2, \ldots, -1, -2\}$ which means

$$e \to^* v : \mathsf{int} \ .$$

[5] Let's assume that our language is more complex than the while-language that has no relevant type-related properties. Maybe even a functional language. The argument is independent of any concrete language.

As prognosticated, after evaluation (if that terminates), the result is of type integer. One way of seeing this is that the type int of $e$ is **preserved** by evaluation $\rightarrow^*$. Indeed, the type should be preserved under single steps, i.e.

$$e : T \quad \text{and} \quad e \rightarrow e' \quad \text{then} \quad e' : T$$

If one can establish type preservation one step, then one directly has type preservation for $\rightarrow^*$, of course. And actually, if we don't think of $e$ as an concrete program or expression we evaluate, but one case of multiple syntactic construct, like for instance ...

Remember we have a small-step **structural** operational semantics, we transforms the syntax itself ("structual"). That makes is possible to not just type check the program we start with, but conceptually type check all intermediate stages. Given a static type system an interpreter would not do that, checking all the intermediate stages a program can be in. For once it's inefficient, and secondly it not needed: if the type checker has established the well-typedness of the original program, type preservation guarantees that the type is preserved, so no need to double check.

If one has a compiled language, the generated code may have not representations any longer of type information, the run-time system could not even type check the running code, even if it wanted (the type are gone, and the code is no longer abstract systax anyway). But if one has established type preservation on the operational semantics (and the compiler faithfully realizes the semantics, of course ... ), then all is fine.

But isn't there run-time type checking? Yes, in languages without static types systems or dynamically typed languages, there hopefully is run-time type checking. But it's not that before each step, the whole program re-type checked (anyway, the implemenation won't be a direct realization of a structural operational semantics). What is done instead is that checks are done only at critical points, resp. for values. So the run-time type system will to **not** try to **predict** to what type of value a programs evaluates. Instead, before **using a value**, it **checks** its type. That's still an overhead, but not in anyway comparable to re-typechecking expressions on every single step along the way.

We can introduce a notation for that. Let's write $e \overset{\lightning}{\rightarrow}$ of the next step that $e$ is is a type error. That might mean as run-time type error is thrown, or that some data value is used in an improper way (when the problem occurs without run-time type check) which in practice would lead to undesirable results (unspecified behavior, data corruption ... ). The notation $\overset{\lightning}{\rightarrow}$ is introduced just right now for sake of explanation, the rest of the script will not use it.

With that, we have all pieces in place to estrablish establish soundness or correctness of a type system (in case the system is correct ... ). There are 2 parts to the argument, one is that all programs are always well-type. That's an easy induction: the program to start with is well-typed, of course, because that's what the static type system is for: only well-typed programs are run. The induction step is **type preservation:** if a program as type $T$, the it still has the same type after one single small step from the operational semantics. Type preservation is also known under the same **subject reduction**.

The second part of the argument is: a well-type program has no type related error **right now**, i.e., the next step is **not** a $\overset{\lightning}{\rightarrow}$-step, it's a $\rightarrow$-step (or the program is terminated).

1. a) **base case:** (for free). Initially, the program is well-typed.
   b) **induction step:** A well-typed program with type $T$ doing a singe step is still well-typed with $T$
2. A well-typed program has no run-time error **now**.

#### 2.2.3.1 How to make a similar argument for data-flow analyses?

We intend to prove the correctness of the analysis, as an example how one establishes such results. Parts of the argument are specific for the paricular analysis (of course), but some others are not. That the analysis is safe or correct connects the operational semantics with the analysis. That's general for all analyses. That would also be the case for type systems (and type and effect systems). The correctness is a statement that given a program and an analysis for it, the program behaves the way the analysis predicts. What exactly the analysis predicts depends on the analysis of course. In the chosen case, it's a statement about the liveness status of variables, so that's particular.

To establish the correctness of data-flow analyses, the overall argument is similar to the situation for type system we sketched. One difference is, that type systems reject ill-type programs, whereas data-flow analyses just analyses the situation and and prognosticates something about the future behavior (without rejecting programs), for instance, which definitions may reach which points in the programs, or which variables are live where. Still, it's a prognostication that may or may not be correct. One key ingredient will be analogous to the setting for types (it's only not called type preservation): The result of the analysis remains true during execution. A variable that is considered statically dead will not suddenly become statically live after one step, a definition that has been predicted not to reach a certain program point, will not actually predicted to reach it after one step. So the **analysis is preserved** under reduction.

We tackle the preservation part first and we do the argument for live variable analysis. Actually, that part would work for the other analyses analougsly. The second part of the argument, which involves formulating what constitutes a violation of the analysis (corresponding to a run-time type error) is specific for the chosen analysis.

**Preservation of the analysis (for live variable analysis)** Let's use live variable analysis as example, and we assume that it's given as given as *constraint system* (not as equational system, but it does not really matter mostly). Remember, that $\mathsf{LV}$ is a may analysis, and the flow constraints were given in equation (2.8) from Definition 2.8. To solve a constraint system means to find values for all variables over which the constraints are formulated. In our case we are after the liveness status of all variables at each entry and exit for each label or block. That could be represented by to functions $\mathit{live}_{entry}, \mathit{live}_{exit} : \mathbf{Lab}_* \to 2^{\mathbf{Var}_*}$, and let's call the "combination" of both *live*. Some such *live*-assignment are solutions to the constraint systems and some are not. Let write

$$\mathit{live} \models \mathsf{LV}^{\subseteq}(S_*) \tag{2.15}$$

to express that a given *live* **solves** or **satisfies** the constraint system for the given program $S_*$ (analogously for equations $\mathsf{LV}^{=}(S)$).

The preservation of solutions ("subject reduction") are almost obvious. It basically rests on the fact that the less constraints one has, the more solutions exists, and the set of constraints get's never bigger during execution.

**Lemma 2.2.10** ("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 2.2.11** ("Subject reduction")**.** *If* $live \models \mathsf{LV}^{\subseteq}(S)$ *and* $\langle S, \sigma\rangle \to \langle \acute{S}, \acute{\sigma}\rangle$*, then* $live \models \mathsf{LV}^{\subseteq}(\acute{S})$

Analogous results would hold for other data flow analyses. Lemma 2.2.10 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 $S$.

Corollary 2.2.11 is a direct consequence of that. In general, that's what the term "corollary" means: an immediate relevant 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 $\textsc{While}_1$. 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*.

**Correctness guarantees** Unlike the previous part of the correctness argument, the preservation, this one specific for the analysis. What guarantees a specific analysis gives should be reasonably clear. Since the analyses are approximate, there is only a guarantee "in one direction": the current value ofa variable that is analyzed to be dead is *guaranteed* not to read in the future (for all runs starting from the point where the variable was judged to be dead). If a variable at a given point is judged *not* to reach another points, then that's guaranteed (not the other way around). And so on.

That may be clear enough, but that does not help immediately in establishing correctness the way we did when using type systems as illustration. There, one key factor was that a well-typed program does not raise an error **right now**, one way of capturing that is to say that the **next** step is not $\overset{\lightning}{\to}$. One could do that also for the analyses here, for instance for live variable analysis: If one would have a situation where the next steps involves reading from a variable, one could check if the variable is considered statically dead or live at that point. If it's analyzed to be live, all is fine, and the standard semantics step $\to$ would be done. If not live, that constitutes the equivalent of a run-time error.

And it's not always clear how to formulate exactly what it means that the program is safe, more precisely how to formulate it in a "usable" way (we come to what we mean by that later). What correctness means in the speci

## 2.2.4 Rework

### 2.2.4.1 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 2.2.12** (Equational and inclusion constraints)**.**

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

## 2.2.5 Correctness of the liveness analysis

### 2.2.5.1 Correctness and correctness relation

That a variable is (statically) dead is a claim that no matter how the program runs, the variable won't be used in the future, more precisesly it won't be read (unless it's overwritten first). In other words, the current value stored in a statically dead variable will not be read, no matter what. It's of course an approximative claim: a variable considered live may or may not be factually read in a concrete run. There's not even a guarantee that there exists one run where the content of a live variable will be read. Only the opposite is guaranteed: the content of a dead variable will never be used.

That's statement over the about the semantics of a program, about it's future behavior. Formally it's a statement about

$$\langle S, \sigma \rangle \to^*$$

given the result of the liveness analysis for $S$. So the correctness claim is about **arbitrarily many steps** $\to^*$, althogether about infinitely many configurations and states.

Now, how can one prove things? There is basically only one way to do that, and that's **by induction.** That's a general statement, and it applies to many situations, including proving correctness of analysis, given an operational semantics.

Before we look (a bit) at specifics for the argument for liveness analysis, we explain the way such arguments are arranged (beyond just saying it's by induction).

But first to the induction. It's induction over the the number of steps $\to^*$. Induction is a general proof technique. It involves an something one likes to prove, a property. It's sometimes called *induction hypothesis*. Then the proof proceeds by establishing that it holds in the *base case* (or for all base cases) and establishing the *induction case* (or cases). The exact shape depends in the structure for which one proves something. Induction is often first encounterer as induction over the natural numbers, where the base case is 0, i.e., establishing the property for 0, and the induction case involves proving that the property

holds for $n+1$ under the assumption that it holds for $n$. It's also called induction step, and it steps from $n$ to $n+1$, showing *preseveration* of the induction hypothesis when doing that "step". Of course there is also induction over trees (where the base case are the leaves and the induction step involves proceeding from children to parent nodes. Derivation trees could be an application in the context of this lecture.

But back to establishing the soundness of an analysis. It's induction on $\rightarrow^*$. The structure of the arguments is the analogous for all kinds of analysis, including type systems (which we cover later) and type and effect systems, or here for data flow analysis. Even if the general situation is analogous, some aspects are slightly more straightforward and conventional for standard types. Therefore we lay out the argument first for types (and later again, pointing what changes a bit)

So we assume a standard type system, and we have established that a program is well-typed. Now is

basic intuitition: **only live variables influence the program**

The following is the central definition that allows to formulate what we mean that liveness variable analysis is correct. The definition itself is a relation on states, it's an equivalence relation, and actually does not refer to liveness at all. It takes two states and compares them with respect to a given set of variables, here $V$. Two states are $\sim_V$ if all variables in $x$ carry the same values.

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

$$\sigma_1 \sim_V \sigma_2 \text{ iff } \forall x \in V. \sigma_1(x) = \sigma_2(x) \tag{2.16}$$

As said, the definition does not mention liveness, but the intention will be that we make use of that relation using live variables for $V$. I.e. we think about two versions of state that carry the same values for all live variables, but where the dead variables may carry different variables. The live variable analysis correct if (see Figure (2.5)

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

Figure 2.5: Illustration: liveness variables

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.

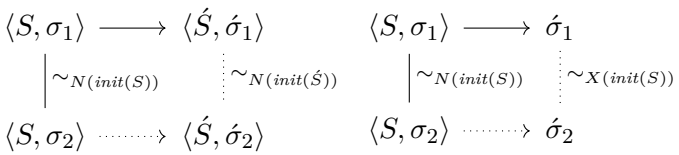
As a side remark: the while language is *deterministic*, meaning a program code $S$ and a state $\sigma$ *determines* the successor configuration (if we are not yet at the final configuration). Note also: the intra-block (and backward) definition of liveness *directly* gives that for an assignment $x := a$, 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").

**Lemma 2.2.13** (Preservation inter-block flow)**.** *Assume live* $\models \mathsf{LV}^{\subseteq}$*. If* $\sigma_1 \sim_{X(l)} \sigma_2$ *and* $l \to_{flow} l'$*, then* $\sigma_1 \sim_{N(l')} \sigma_2$*.*

**Theorem 2.2.14** (Correctness)**.** *Assume live* $\models \mathsf{LV}^{\subseteq}(S)$*.*

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

The theorem is captured by the following figure:




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 standard, and allows to express such properties in a short diagrammatic but still precise manner.

**Theorem 2.2.15** (Correctness (many steps))**.** *Assume live* $\models \mathsf{LV}^{\subseteq}(S)$

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

## 2.3 Monotone frameworks

We have seen 4 different classical analyses, which all shared some similarities. In this section, those analyses will systematically be put into a larger context, which is known as *monotone framework*. As unifying principle, this was first formulated by Kildall [3] 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.

**Abstract, general formulation**

$$\begin{array}{lcll} \mathit{Analysis}_\circ(l) & \sqsupseteq & \begin{cases} \iota & \text{if } l \in E \\ \bigsqcup\{\mathit{Analysis}_\bullet(l') \mid (l', l) \in F\} & \text{otherwise} \end{cases} & (2.17) \\ \mathit{Analysis}_\bullet(l) & \sqsupseteq & f_l(\mathit{Analysis}_\circ(l)) & \end{array}$$

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 are the **transfer functions**. 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.

So **forward** analysis uses $F = \mathit{flow}(S_*)$, the $E$ ("extremal node") refer to initial or entry nodes of the CFG, where we assume isolated entries,and $\mathit{Analysis}_\circ$ is the information for entries of blocks and $\mathit{Analysis}_\bullet$ for the exits. For **backward** analyses all is dual: For the flow, we have $F = \mathit{flow}^R(S_*)$, i.e., the reverse flow, $E$ represents the exit nodes of the control-flow graph, where we assume isolated exits, and $\mathit{Analysis}_\circ$, represents the exits and $\mathit{Analysis}_\bullet$ the entries of the blocks.

**May** analyses express properties for *some* path, and one is interested in the *smallest* solution. The lattice-related operations and relations $\sqcup$ and $\sqsupseteq$ from (2.17) are interpreted as $\cup$ and $\supseteq$, respectively. For **must**, all is dual, we are after properties for *all* paths, one is after the *largest* solutions, and the operations and relatations are $\cap$ and $\subseteq$.

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.

The **property space** (here called $\mathcal{L}$) 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, and all of our example analyses were of that form. Often, the lattice sattisfies the **ascending chain condition** (stabilization), especially **finite** lattices have that condition, and it guarantees termination of constraint solving, resp. fixpoint iteration.

Towards the end, let's say a few words about the **transfer functions**:

$$f_l : \mathcal{L} \to \mathcal{L}$$

with $l \in \mathbf{Lab}_*$. There is a transfer function associated to each block or label $l$ (determined by the instruction(s) in the block). The transfer functions need to be **monotone**, otherwise the whole approach does not work. The condition, however, is a very natural one.

The last condition in connection with transfer functions is likewise natural, so nothing to worry about normally, but listed here for completess. Namely that transfer functions can be **composed**. It means, that $f_1$ and $f_2$ are transfer functions, then alo $f_1 \circ f_2$ is a transfer function. So one has a domain $\mathcal{F}$ of monotone functions over $\mathcal{L}$ that comprises all transfer functions, it is closed under composition, and to round it off completely, the identity function.

Often, one assumes or works an additional condition on transfer functions, which has practical consequences, i.e., positive consequences. That is, one works with transfer functions that are **distributive**. This means, they satisfy the following condition

$$f(a_1 \sqcup a_2) = f(a_1) \sqcup f(a_2) \ .$$

In this case, one has a **distributive** monotone framework.

Let's finish up by summarizing the 4 classical data flow analyses we covered. For a label consistent program $S_*$, they are all *instances* of a monotone, distributive, framework. That the properties form a lattice in each case is immedate (as all instances are using sets ordered by subset or superset inclusion. The qascending chain condition is guaranteed, for all 4, the lattices are *finite*. The *closure* conditions on $\mathcal{F}$ (something we have not really discussed much as it is mostly unproblematic as a naturally given condition) are also fine. The transfer functions are monotone, and the set $\mathcal{F}$ of all transfer functions is closed under identity[6] and composition. All 4 instances are additionally **distributive**, which is assured by the fact that the analyses are formulated using kill- and generate-functions for the transfer function.

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

Table 2.2: Overview over the 4 example DFAs

**Remark 2.3.1.** *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*

[6] It means nothing else that the identity function is a transfer function

*staight-line code). In other accounts, the control flow graphs and/or the transfer functions may be differently represented, without changing anything relevant.*

*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.* □

## 2.4 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 (subset or superseet 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. This section just fleshes that out a bit.

### 2.4.1 Chaotic iteration (repetition)

We even have seen a very schematic formulation of an corresponding iterative approximation algo in the introduction, called *chaotic iteration* (see Listing 2.1).

```
Input:    equations for reaching defs
          for the given program
Output:   least solution: RD⃗ = (RD_1, ..., RD_12)
```

```
Initialization:
        RD_1 := ∅; ...; RD_12 := ∅
Iteration:
        while RD_j ≠ F_j(RD_1, ..., RD_12) for some j
        do
                RD_j := F_j(RD_1, ..., RD_12)
```

Listing 2.1: Chaotic iteration

The chaotic iteration from the introduction was formulated for reaching definitions, but it works for all monotone frameworks analogously. The chaotic iteration here is also specialized to the factorial-example that was used before, in that it mentions 12 variables $\mathsf{RD}_1 \ldots, \mathsf{RD}_{12}$.

The algorithm works randomly, i.e., it follows a **random** 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 arbitrary behavior still is guaranteed to terminate and find the fixpoint, all other more specific strategies or heuristics will find it as well. So while chaotic interation is not really practical, it gives the assurance: 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 comforting setting.

In this section, we don't do many different heuristics, we show just one has a rationally organize data in connection when doing fixpoint algorithm, and that can also be used to steer the approximation algorithm. That general form of algorithm is known as **worklist algorithm**. The name comes from a central data structure, the so-called **worklist.**

### 2.4.2 The idea of a worklist data structure

The worklist can help steering the algorithm through its seach space, which in this case means, which particular constraint to tackle next. It addresses also another inefficiency in the random iteration, resp. fleshes out a point that in the chaotic iteration formulation states in one basic step: take a constraint that is not already satisfied and repair it. What is left unspecified in the chaotic algorithm is how to detect constraints that are not satisfied? Checking all of them, effectively searching for one, for instance going through all the constraints and checking them one after the other until one finds one that is not satisfied and repair it is not very effective. It's better to store a list of constraints that need treatment, keep the list up-to date and just pick one constraint from the list to do the elementary fixpoint approximation step. That list is of course the work-list.

Since constraints in our data-flow setting are connected with edges in the graph, picking the next constraint to treat 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 (that might not even be called graph traversal).

The chaotic iteration is not a work-list algorithm, it lacks this data structure. Instead it uses non-determinism and not an explicit representation (as data structure) of the edges still to be treated.

Calling the data structure a "worklist" is 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 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

$$\mathsf{RD}_j \neq F_j(\vec{\mathsf{RD}}) \ . \tag{2.18}$$

Now, 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 later at the worklist algorithm, the worklist does *not* contain **exactly** all current situations described by equation (2.18). It contains an **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 are potentially in need for 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 checked if, at that point, the corresponding contraint actually needs treatment; it may well be that it's already satified, as the worklist contains an overapproximation of pieces of work.

### 2.4.3 Input, output of the worklist algorithm, and other data structures

The input for the algorithm is a (distributive) **monotone framework**. So it's a generic algorithm, that works uniformely for many different analyses including the ones we discussed). Another input is the set of constraint or equation system representing the data flow problem for a **given program.**

Actually, the algorithm is an implementation of "fixpoint" constraints solving over lattices and finding smallest (or largest) solutions for such constraints So they work analogously for other problems that can be formulated that way.

The algorithm is formulated in an imperative manner (as was the chaotic iteration). It represents the constraints variables as an array, it's called *Analysis* in Listing 2.2 (it corresponds to the "array" $\mathsf{RD}_1,\ldots,\mathsf{RD}_{12}$ in the chaotic interation for reaching definitions. This array contains the approximations to the ultimate (fixpoint) solution and is updated during the algorithm. The array has one slot for each basic block or label $l$.

**Remark 2.4.1.** *One might have expected that one has an array with two slots per label $l$, one for the entry and one for the exit. For instance, in the RD-analysis with chaotic iteration, there were 12 $\mathsf{RD}_i$-slots for 6 labels (in the factorial example). The code from Listing 2.2 uses only an array with $l$ slots, not $2l$ slots. That's just a small (not not important) implementation details. What is given back in the end are $2l$ values (given back as $MFP_\circ$ and $MFP_\bullet$), the latter is just one more application of the transfer function on the result of $MFP_\circ$ per label.* □

```
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 f_l(Analysis[l]) ⋢ Analysis[l']
                  then Analysis[l'] := Analysis[l'] ⊔ f_l(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) := f_l(Analysis[l])
```

Listing 2.2: Worklist algorithm

The algorithm works in three stages: initialization, a main loop, and finally processing the result for being returned. The main loop increases the approximation in *Analysis*, at least if treating a constraint that isn't solved yet, and when all constraints are solved, it gives back the solution which (basically) is the values contained in *Analysis*.

#### 2.4.3.1 Solving the analyses

- *2 solutions* for the monotone frameworks
  - **MFP**: "maximal fix point"
  - **MOP**: "meet over all paths"

**Remark 2.4.2** (Meet-over-all paths (MOP))**.** *That may be discussed later.* □

#### 2.4.3.2 MFP

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

#### 2.4.3.3 Worklist algorithms

- for more details (different traversal strategies): see Chap. 6 from [5]

#### 2.4.3.4 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 $\mathsf{RD}_i$ depends via $F$ on *all* $\mathsf{RD}_j$. 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.

#### 2.4.3.5 MFP: properties

**Lemma 2.4.3.** *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

□

#### 2.4.3.6 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

⇒

$$O(e \cdot h) \tag{2.19}$$

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

## 2.5 Interprocedural analysis

### 2.5.1 Introduction

This section makes the analysis, resp. the language a bit more realistic. So far it was a really 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 are handled as in the pure while language so far are of course also part of the run-time environment, it's just a simpler 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 and pop, etc. We do a stack (because that's what we need), but it's not so visible as if we would work with an explicit stack, it's more implicit (using a syntactic bind-construct, see later). Instead of a *state* (written $\sigma$) as before, the memory is represented more 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. The language is statically scoped, but the structure of the scopes is simple, as the language has no nested procedures or nested lexical scope.

What we also have to arrange is **parameter passing,** it will be a "mixture" of **call-by-value** annd **call-by-result**. Call-by-value is very standard, call-by-result slightly less so. The motivation for that choice is not because call-by-result is a central mechanism. It's more presentational, for the smallest possible extension of the imperative while-language. We integrate calls into the syntax as `statements`, which means that calls er evaluated for their side effects only. See further explanations below

### 2.5.2 Extending the syntax

The syntax is shown in equation (2.20), an extension of the grammar for the while-language. Procedures $P$ is a new syntactic category, for procedure declarations. Typical procedures may be called $p$ ... Procedure **calls** are statements. The abstract syntax is *labelled*, as we did earlier to obtain the control-flow graph. Note: the call statement has **2 labels**, which is needed to represent the call-return behavior of procedure calls. We assume in the following that (again) we have unique labelling, and furthermore that declared procedures are called and all procedures have different names (as we don't want to deal with erroneous behavior.

**Extending the syntax**

$$\begin{array}{lcll} P & ::= & \texttt{begin}\, D_*\ S_*\, \texttt{end} & \text{program} \\ D & ::= & \texttt{proc}\, p(\texttt{val}\, x, \texttt{res}\, y)\texttt{is}^{l_n} S \texttt{end}^{l_x} \mid D\ D & \text{proc. definitions} \\ S & ::= & \ldots\, [\texttt{call}\, p(a,z)]^{l_c}_{l_r} & \text{statements} \end{array} \tag{2.20}$$

The language is still rather simple. It features **first-order** procedures. The procedures, additiontally, cannot be **nested**. However, recursive procedure definitions are allowed, including mutual recursion.

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. Another difference (for now) is 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 using languages supporting **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. When dealing with side-effect only procedures, it should not even be called **evaluating** the procedure. The word "evaluating" conventionally 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.

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* 2.5.1 (Fibonacci).

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

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, and there are easy ways to write a tail-recursive and thus a more efficient version of this procedure (or a purely imperative one without procedures). However, for illustrating challenges with interprocedural analysis, we intend to have an example with *two* calls, and that's an easy example of that. □

### 2.5.3 Extending the semantics and the CFGs

Next comes the adaptation of the definition of the control-flow graph. To do so, we need to adapt and extend the definitions of flow, blocks, etc. Actually, it's pretty straightforward. 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 (on 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.

#### 2.5.3.1 Block, labels, flow: building the CFG

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

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

Note that there are two *new* kinds of flows (written slightly differently, namely an edge for *calling* and one for *returning*. That's actually not the end of it, later we will introduce the so-called interprocedural edges *IF*, that's when conceptually something new happens, namely "edges" which are 4-tuples, not just tuples.[7] So apart from the fact that the edges are written slighly differently, there's no real change, the new edges are treated the same way as the old.

#### 2.5.3.2 For procedure declarations

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

Let's start with the "standard" flow of a program, meaning that we leave out the treatment of procedure calls, the so-called *interprocedural flow* (IF) for later. So basically the definition here is nothing more than a repetition.

$$\begin{array}{rcl} \mathit{init}_* & = & \mathit{init}(S_*) \\ \mathit{finall}_* & = & \mathit{final}(S_*) \\ \mathit{blocks}_* & = & \bigcup\{\mathit{blocks}(p) \mid \texttt{proc}\, p(\texttt{val}\, x, \texttt{res}\, y)\texttt{is}^{l_n} S \texttt{end}^{l_x} \in D_*\} \\ & & \cup \mathit{blocks}(S_*) \\ \mathit{labels}_* & = & \bigcup\{\mathit{labels}(p) \mid \texttt{proc}\, p(\texttt{val}\, x, \texttt{res}\, y)\texttt{is}^{l_n} S \texttt{end}^{l_x} \in D_*\} \\ & & \cup \mathit{labels}(S_*) \\ \mathit{flow}_* & = & \bigcup\{\mathit{flow}(p) \mid \texttt{proc}\, p(\texttt{val}\, x, \texttt{res}\, y)\texttt{is}^{l_n} S \texttt{end}^{l_x} \in D_*\} \\ & & \cup \mathit{flow}(S_*) \end{array} \tag{2.21}$$

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

But now we start treating the new situation for real, introducing really **new kind of edges** representing the interprocedural flow (IF) The inter-procedural from goes from call-site to a procedure and back, i.e., include edges $(l_c; l_n)$ and $(l_x; l_r)$. But not separately, as two edges, but the two edges from one call-site to one procedure are coupled together. Doing that allows for a more *precise* (= better) capture of the control-flow. We use *IF*

[7] When taking about graphs in general, that's sometimes called hyper-edges, and consequently one would speak about hyper-graphs therefore. That's just terminology and for us, we speak about standard flow (the normal edges) and interprocedural flow, the ones we introduce soon after.

as abbreviation for *inter-flow*$_*$ or *inter-flow*$_*^R$ (for the forward or reverse situation). With the pair of edges for a call and for a corresponding return coupled together, that makes such inter-procedural flow edges 4-tuples:

**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 generalized graphs that allows that are called hyper-graphs. Not that it matters much in our context, how it's called in graph-theoretical terms.

*Example* 2.5.2 (Fibonacci flow). The CFG for the Fibonacci code from Example 2.5.1 is shown in Figure 2.6

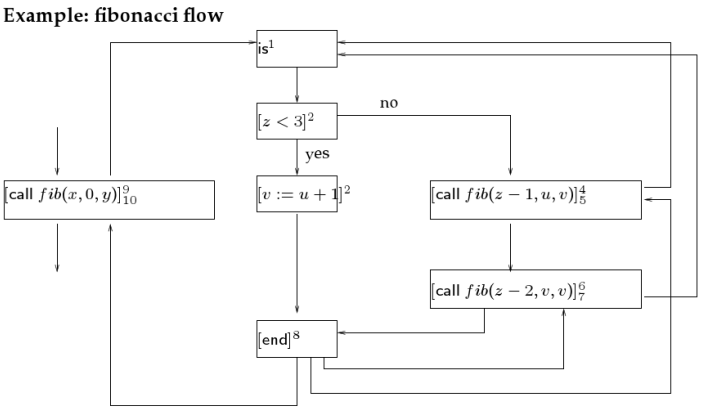



Figure 2.6: CFG for Fibonacci

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). □

#### 2.5.3.3 Semantics: stores, locations,...

The operational semantics, as before, will be defined by transitions between **configurations**. Configurations descibe the "situation" a program is currently in. That includes 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 fully *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 the content for the current version of $x$.

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 stored. That is not yet the same as having references or **pointers** in the language, 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 pointers 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.

**Representation of "memory"** The representation of memory is shown in equation (2.22).

$$\begin{array}{lcll} \xi & \in & \mathbf{Loc} & \text{locations} \\ \rho & \in & \mathbf{Env} = \mathbf{Var}_* \to \mathbf{Loc} & \text{environment} \\ \varsigma & \in & \mathbf{Store} = \mathbf{Loc} \to_{\mathit{fin}} \mathbb{Z} & \text{store} \end{array} \tag{2.22}$$

Remember how memory was represented in the simpler setting of the while-language without procedures, in a language with full static memory layout, it was called **state** and was a mapping from variables to (integer) values: $\sigma \in \mathbf{State} = \mathbf{Var}_* \to \mathbb{Z}$. Also in the setting, values are associated with variables, but it's via locations or addresses, i.e., here it's

$$\sigma = \varsigma \circ \rho$$

We want that $\sigma$ is *total* (to avoid that the semantics have to deal with errors or undefined behavior) and that implies $\Rightarrow$ $\mathit{ran}(\rho) \subseteq \mathit{dom}(\varsigma)$. We assume a **top-level environment** $\rho_*$, were all variables to **unique locations** (so there is no *aliasing*).[8]

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

[8]The language has no pointers or references, and the semantics will preserve that there never is aliasing.

### 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 are 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 is no syntax that can be (mis-)used for that. What we are doing then is to simply **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 generated 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 "semantic tricks" are actually quite common when designing an operational semantics. It's known as **run-time syntax**. For the simple while language without procedures, we did not see extra 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 steers the execution steps, and if that calls for new syntax called run-time syntax, why not?

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

But first to the "format" of the transition **steps**. The steps are formulated *relative* to an **environment** $\rho$. They look as follows.

$$\rho \vdash \langle S, \varsigma\rangle \to \langle \acute{S}, \acute{\varsigma}\rangle \qquad (2.23) \qquad\qquad \rho \vdash \langle S, \varsigma\rangle \to \acute{\varsigma} \qquad (2.24)$$

Since the shape of the steps have changed, also the old rules, those for the older constructs need to be adapted. The steps are now done relative to some environment, as said, and we write $\rho_*$ for the **global** environment, the environment that stores the values for the global variables of the program.

The adaptation of the old rules is straightworward. Nothing basically changes, except that state $\sigma$ is now split into two components and written in a strange way, using the $\vdash$ notation. *Writing* it like that is of course not central, a configuration is now a **triple,** instead of a tuple, at least in the non-terminated form.

In the rules covering old syntax (without calls), 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 $\sigma$). 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 $\rho$, is **fixed** per **scope.**

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

Before we go to the concrete rules: the transition in equation (2.23) captures that the program with the store executes one step *inside a given environement* $\rho$. 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 a local state space for locally scoped variables? The form of the rules seems not able to capture that since $\rho$ 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.

#### 2.5.3.4 Call-rule and introducing bind

Now we turn to 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. See Table 2.3.

This rule realizes **parameter passing**, more precisely 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 assume 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** $\xi_1$ and $\xi_2$ 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 $\xi_1$ represents the call-by-value mechanism for the arguments (here only one for simplicity). The address $\xi_2$ 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 $\sigma$ 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 pattern at the corresponding address(es) which can be seen as a random value; there is no such thing as a "empty" memory slot in reality. 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 $\rho_*$ is the "top-level", global 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 extra procedure-local variables (beyond the formal parameters).

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

Table 2.3: Call-rule

#### 2.5.3.5 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 statements (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 $\rho$ in a nested fashion in the derivation tree, which is an abstract representation of the stack frames.

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

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

Table 2.4: Call-rule

**Comparison with stack based run-time environments in compilers** The following discussion 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 some previous run of the lecture (in 2020), there had been some remarks 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 (SOS) 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 $\rho$. 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 $\rho_*$) *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 analogous concept. The semantics here has no need for that. Assuming two different different environments $\rho_1$ and $\rho_2$ at some point in the execution, one for the caller one for the callee, $\rho_1$ and $\rho_2$ are *unrelated* except of the fact that they share the same bindings for the global variables. I.e., both $\rho_1$ and $\rho_2$ extend $\rho_*$ but the callee has no access to bindings in $\rho_2$ (or other "earlier" 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).

### 2.5.4 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 simply 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. As a starting point, we are assuming a *mononote framework* $(\mathcal{L}, \mathcal{F}, F, E, \iota, f)$. To deal with procedures, we assume for each procedure **call** 2 transfer functions $f_{l_c}$ (call) and $f_{l_r}$ (return), and for each procedure **definition** further 2 transfer functions: $f_{l_n}$ (enter) and $f_{l_x}$ (exit).

The naive formulation treats the *IF* edges $(l_c; l_n)$ and $(l_x; l_r)$ as **ordinary** flow edges $(l_1, l_2)$. Additionally we take as *transfer* functions for procedure calls and procedure definitions the *identity* function.

#### 2.5.4.1 Equation system ("naive" version)

$$
\begin{array}{lcl}
A_\bullet(l) & = & f_l(A_\circ(l)) \\
A_\circ(l) & = & \bigsqcup\{A_\bullet(l') \mid (l', l) \in F \text{ or } (l'; l) \in F\} \sqcup \iota^l_E
\end{array}
\tag{2.25}
$$

with

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

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

The analysis is **safe**, but can be made more precise and less abstract. That the analysis 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 no attempt to figure out which one is taken, so the flow explores boths.

**Remark 2.5.3** (Dead code)**.** *Of course, in a program, even for a program with input, it may not be the case, that in reality, at run-time, both branches are taken. 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 suspicious situation or code smell: why did the programmer program a condition, when one 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 analyses similar to the ones we covered to try figuring out such situation.* □

Dead code is not our concern 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 and will be be taken by the level of abstraction of the analysis.

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 in 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.

**Remark 2.5.4** (Connected control-flow graph and call graphs)**.** *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. That, though, is a silly anomaly. One could simply assume that this does not occur, and it's easy to check if all procedures are called at least once. 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 returns to the point where it was 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's "unessessary" imprecision or an additional factor of imprecision. **Context-sensitive** analyses try to address and mitigate that additional imprecision, where the naive context-insensitive analysis ignores the potential of improvement.

### Path (in)sensivity

Before we do the we may further reflect on that imprecision. The data-flow analyses, the monotone framework, so far were 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 immediately 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. Of 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 they 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*, by 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 or *following* down a 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 paths 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 characterizes path-insensitive data flow. 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 in 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 remembered that place (or "information about that place").

That will be an approach the lecture will pursue.

**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 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 a similar spirit of context-sensitive 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-sensitive way is *the same* as context-sensitive interprocedural analysis. It's two enhancements in the same spririt:

> Add more information and you get more precise, but the analysis will take more effort, in particular 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** here and shed more light about some wordings we used in the discussion, namely that the analyses need to be enhanced by additional 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 2.5.5 below). We are dealing with specific graphs, control-flow graphs, which may obey 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 graph 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 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 2.5.5. 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 explore infinitely many paths one by one is not an 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 the worklist algo as discussed which ignores the notion of paths completely.

How about *interprocedural* analysis? As already mentioned, if we ignore the special nature of the calls 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.

### 2.5.5 Taking paths into account

Paths hier corresponds to the standard notion of a path through a graph. As discussed, paths so far 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.

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 know 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 composition (and includes the identity function).

- historically: MOP stands for **meet over all paths**
- here: dually also *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

**Definition 2.5.5** (Paths and transfer functions for paths)**.**

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

Given a path $\vec{l}$, the **transfer function** for that path is given as

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

#### 2.5.5.1 Meet over all paths

Analyses can be forward or backwards. Consequently we need to consider forwards paths or backward paths, but otherwise the two situations are equivalent. For the forward-situation,

we consider paths from the initial block to the entry of blocks. For the backward-situations, it's dual:

For the formulation of the MOP analyses, we need *two* variations of paths (for given $l$): Up-to but not including $l$, written $path_\circ(l)$, and up-to including $l$, witten $path_\bullet(l)$.

**Definition 2.5.6** (MOP)**.** Given a monotone framework, the **meet-over-all-path** solution(s) are given as follows:

$$\begin{aligned} MOP_\circ(l) &= \bigsqcup\{f_{\vec{l}}(\iota) \mid \vec{l} \in path_\circ(l)\} \\ MOP_\bullet(l) &= \bigsqcup\{f_{\vec{l}}(\iota) \mid \vec{l} \in path_\bullet(l)\} \end{aligned} \tag{2.26}$$

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 join), 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.

**Lemma 2.5.7** (Connection between MFP and MOP)**.**

$$MFP_\circ \sqsupseteq MOP_\circ \text{ and } MFP_\bullet \sqsupseteq MOP_\bullet \tag{2.27}$$

*In case of a distributive framework*

$$MFP_\circ = MOP_\circ \text{ and } MFP_\bullet = MOP_\bullet \tag{2.28}$$

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

#### 2.5.5.2 Valid (and complete) paths

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 set of 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").[9] 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 valid path, in the other about whether an analysis takes call-sites into account.

Context-free languages are typically given by context-free grammars, which is effectively what we do below. Actually before we cover 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 to the call site. A complete 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 call.

That can be illustrated by some prototypical example of context-free languages, namely those of well-balanced parentheses or brackets. 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 the example with the three kinds of 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 likewise be defined 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.

[9] 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.

### 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** Let's start by defining complete paths. Its definition is simpler than that for valid paths, and actually valid paths will build upon the notion of complete path. We are given a program $P_* = \texttt{begin}\, D_*\ S_*\, \texttt{end}$ or its control-flow graph and we want to define the set of **complete** paths from $l_1$ to $l_2$ (for which we write $CP_{l_1,l_2}$). That can be done by a **context-free grammar**, where $CP_{l_1,l_2}$ are the non-terminals and the labels $l$'s are the terminals. We assume forward analysis here. The productions are shown in Table 2.5.

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

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

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

Table 2.5: Complete paths

The notion of **complete** path is rather straightforward. It informally says that each call is answered by one corresponding later return, and also that each return is matched by one corresponding earlier 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 identified by the name of the function being called, but by the call-sites and 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* 2.5.8 (Fibonacci). See Figure 2.6 from before for the control-flow graph of the Fibonacci program. The following shows the concrete grammar generating the complete paths for that example.

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

□

**Valid paths** **Valid** paths are mildy more complex to define, but equally easy to understand. They could be called "incomplete" paths, in that a valid path is a **prefix** of a complete path and with the additional condition that the path has to start at an initial node (for a forward analysis). That means all procedure returns must have a matching call, but not the other way around: calls can be not yet closed by a matching return.

The corresponding grammar is shown in Table 2.6. It's formulated for both forward and backward analyses, so we start at an extremal node (from $E$), either the initial or one of the final nodes, depending on the setting.

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.

We simply have to adapt the previous definition for MOP, to take into account not all by but only *valid* ones. As before we have two flavors, one not including the label last label, and one including it. As before as well (in Definition 2.5.5), the notion of valid path implies that it states at an extremal node:

**Definition 2.5.9** (Valid paths)**.**

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

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

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

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

Table 2.6: Valid paths (generated from non-terminal $VP_*$)

**Definition 2.5.10** (MVP)**.** Given a monotone framework, the **meet-over-all-valid path** solution(s) are given as follows:

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

Now we have defined the concept of paths, resp. MOP or MVP. But since there are typically infinitely many paths, it's impractical. In particular one cannot first calculate all paths and then build the meet over them, as in Definition 2.5.10. The situation is similar as for Tarski's result, from Theorem 2.2.7. Also there, we build a meet over a set of lattice elements. Also there it was impractical, even if the set of lattice elements were often finite for the data flow analyses we discussed. We addressed that problem by relying on Kleene's result about fixpoints (for continuous functions), which lead to fixpoint iteration. This is also what we do next here, in the more complex interprocedural setting. So the goal will be to reconcile the path approach with "MFP" and fixpoint approximation.

### 2.5.6 Context-sensitive analysis

Next we finally 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 once (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, the 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.

Independently 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 do is to extend that approach:

> Extend 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 observation, of course: more information being tracked leads to a more precise analysis. One could add some information and render the analysis *path sensitive*, and then calling that *embellished*.

We focus here on adding information to make the analysis deal with procedures, making the analysis *context sensitive*. This information is called **context**.

We are doing data flow analysis, and that includes analysing the body of procedures. Each procedure has its own control-flow graph (which is analyzed the way data flow analyses work).

> The context when analyzing a procedure body is information about the "situation" where the procedure had been called. For instance, from which location it has been called. Inside one body, the context is of course the same, but it changes from caller to callee (and back). Done properly, i.e., taking into account that a call site location can be from a procedure that also had been called and is analyzed in its own context will lead to a choice for contexts known as call strings.

In that way, the particular context information represents the sequence of all open calls. As said, that will be called **call strings**, basically an abstract version of the current **call stack**, and call strings are **unbounded**; adding that yields the same precision as MVP, and that can is undecidable, and we will see how to deal with that.

Adding the full path resp. the call string as the stack of pending calls would lead to full disambiguation. But adding **path** information is *not* the only thing one can do the disambiguation. One can also add different information, namely information about the **data** with which a procedure is called. Perhaps at some places, a particular procedure is called with a non-negative argument, and at others 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.

#### 2.5.6.1 Contexts

Let's introduce the notion of **contexts**. Actually, no concrete contexts are introduced for now, like defining what data to use concretely for a context. Equation (2.29) simply introduces a symbol we use to refer to contexts, whatever those are. Only later will we look at a few different reasonable choices for concrete contexts.

For now, we assume we have some context $\delta$ and discuss how to extend a monotone framework to make use of that context, leading to the notion of embellished monotone framework.

$$\delta \in \Delta \tag{2.29}$$

So $\delta$ is the **context information**, for instance: representing/recording of the *path* taken (or other things). Adding context is called an "embellishment" of the framework:

**Embellished monotone framework**

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

When treating the body of a procedure, there is no change of the context, it's the context of the procedure call. Inside a procedure, one is doing a *intra*-procedural analysis, of course, and the $\delta$ is just some unchanged extra piece of information carried around. It's only when doing calls and return that the $\delta$ changes, in that the calling procedure has a different context than the one being called. In a way the contexts connect caller and callee.

Embellishment is notationally indicated by placing a $\hat{\text{hat}}$ 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 concerned, 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.

**Remark 2.5.11** (Notational remark)**.** *In the following we encounter silly problem. It's that sometimes (and following [5]) we used the letters $L$ and $l$ for different purposes. One is the lattice $\mathcal{L}$ with elements $l \in \mathcal{L}$. The other one is the set of labels* **Lab** *resp.* $\mathbf{Lab}_*$*, with typical elements $l$ as well. Actually, in the book [5], 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 from* **Lab** *(as before) and write $d$, $d'$, $d_i$, etc. for the elements or values of the lattice ($d$ standing 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 labels or nodes are part of the information stored in the lattice $\mathcal{L}$.*

*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 $\mathcal{L}$ contained labels $l$ from* $\mathbf{Lab}_*$*. 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 $\hat{\mathcal{L}}$, which is of the following form

**Embellished lattice $\hat{\mathcal{L}}$:**

$$\hat{\mathcal{L}} = \Delta \to \mathcal{L} \tag{2.30}$$

which contains functions taking a context from $\Delta$ to an element of the original, unembellished lattice $\mathcal{L}$. One can convince oneself that, if $\mathcal{L}$ is a lattice, then so is $\hat{\mathcal{L}}$. 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 (2.30) defines the embellished lattice $\hat{\mathcal{L}}$ by giving one lattice element $d \in \mathcal{L}$ per context $\delta \in \Delta$. This is a way of **lifting** the lattice to the new embellished setting. Of course we need not just to lift the lattice from $\mathcal{L}$ to $\hat{\mathcal{L}}$, we need also to lift the intraprocedural transfer functions, which are functions from $\mathcal{L} \to \mathcal{L}$ to $\hat{\mathcal{L}} \to \hat{\mathcal{L}}$ (unembellished). 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:

$$\hat{_} \;:\; (\mathcal{L} \to \mathcal{L}) \to (\hat{\mathcal{L}} \to \hat{\mathcal{L}}) \tag{2.31}$$

One could call that operation **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 $(A \to B) \to ((List\ A) \to (List\ B))$. One could say that map "lifts" the effect of $f$ from the underlying domains to lists over those domains.

We also need to lift the lattice relation, i.e., the **partial order** $\sqsubseteq$ on $\mathcal{L}$ to its counterpart on $\hat{\mathcal{L}}$. 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 **intra**procedural aspects 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 (2.35). The transfer function $\hat{f}_l$ (for some location $l$) is of type $\hat{\mathcal{L}} \to \hat{\mathcal{L}}$, i.e., a function of type

$$(\Delta \to \mathcal{L}) \to (\Delta \to \mathcal{L}) \tag{2.32}$$

So, it's a function that takes as argument a function of type $(\Delta \to \mathcal{L})$ and returns a function, namely one with $\Delta$ as argument and a return value from $\mathcal{L}$. That's defined in equation (2.35) below and shown on the slides: the "first" functional argument $\hat{d} : \hat{\mathcal{L}}$ is applied in the definition to the "second" argument $\delta$ and the results is fed into the unlifted transfer function $f_l$. Equation (2.35) may be written identically as shown in equation (2.33), making use of $\lambda$-abstraction, and adding the types of the arguments to make it more explicit. We show the lifting in a more explicit form here with $\lambda$, to shed light on how to think of the lifting, which may be more intuitive to get rid of the higher-order formulation, and starting from equation (2.32) may make that explanation clearer.

$$\hat{f}_l = \lambda\hat{d}{:}(\Delta \to \mathcal{L}).\ \lambda\delta{:}\Delta.\ f_l(\hat{d}(\delta)) \tag{2.33}$$

**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 $\delta$, 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

$$A \to B \to C \quad \text{and} \quad A \times B \to C \tag{2.34}$$

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 that one can manually re-write each function of one type equivalently into one of the other form. That might be plausible. What is more: there exists 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 (2.34) 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.

How does it helps to understand the "lifting", the $\hat{\mathcal{L}}$ and $\hat{f}$? We will come back to that later, when discussing equations (2.37) and (2.38), which are a curried resp. uncurried representation of the "same" function.

#### 2.5.6.2 Intra-procedural part: basically unchanged

The intra-procedural part is basically unchanged. Of course it's not really unchanged, as now the information used in the analysis now carries context information around, as the lattice has changed. It's only that the context information does not actually plays a role, it's just carried along as dead weight. Of course that changes later when we take calls and returns into account, as at this point, the context information will be used to discriminate between different call sites, and use that information to connect calls and returns. The degree of matching depends on what we use as concrete context information. Later also the context information **changes** when the data flow "moves" from caller to callee and back. As long as we stay within a single procedure, the context is **unchanged**, and thus just "carried along". It can be seen as information per procedure activation about the situation where that procedure activation came from.

$$\hat{f}_l(\hat{d})(\delta) = f_l(\hat{d}(\delta)) \tag{2.35}$$

The following are the flow equations for the intra-procedural part. They are basically "unchanged" from the settings without procedures, in the sense that the transfer function $\hat{f}$ corresponds to $f$, but lifted to the embellished setting, operating on the embellished lattice:

$$\begin{aligned} A_\bullet(l) &= \hat{f}_l(A_\circ(l)) \\ A_\circ(l) &= \bigsqcup\{A_\bullet(l') \mid (l',l) \in F \text{ or } (l';l) \in F)\} \sqcup \iota_E^{\hat{l}} \end{aligned} \tag{2.36}$$

In the equations, the labels $l$ resp. $l'$ are labels or nodes, **except** those for procedure calls, (i.e., not $l_c$ and $l_r$). That maybe compared to the naive formulation from equation (2.25), which looks quite similar, but which treats the edges involving $l_c$ and $l_r$ as ordinary flow edges (which makes the analysis "naive").

The embellished lattice $\hat{\mathcal{L}}$ is defined by the function type $\Delta \to \mathcal{L}$, and we have defined the interprocedural transfer functions (lifting them). But let's also have look at what the **solution** of the embellished data-flow problem now. Equivalently, let's have a look at the data structure $A$ (resp. $A_\bullet$ and $A_\circ$), mentioned in equation (2.36) 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 unembellished solution is of the type $N \to \mathcal{L}$, whereas *now*, the solution is of type $N \to \hat{\mathcal{L}}$, i.e.,

$$N \to \Delta \to \mathcal{L} \ . \tag{2.37}$$

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

$$(N \times \Delta) \to \mathcal{L} \ . \tag{2.38}$$

Seen like that, in the 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.

#### 2.5.6.3 Remarks concerning how to implement that

Let's ponder how one could implement that that. The unembellished setting, being simpler, is less problematic. Implementing and maintaining a function $N \to \mathcal{L}$ 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 is: there can be many possible contexts from $\Delta$, maybe even ininitely many. That poses a "challenge", independently of whether one bases the implementation on a curried or uncurried view (equation (2.37) vs. (2.38)).

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

#### 2.5.6.4 Sign analysis (intra-procedural)

So far, everything about embellishement has been abstract, it was just about how to lift a given analysis. In the following we illustrate the concepts on one **specific** analysis, known as **sign-analysis.** As a starting point we define that analysis first for the unembellished setting, and then lifting that to the embellished setting. Keep in mind, that we have *not* yet covered the interprocedural part of embellished framework. This will come later in Section 2.5.6.5, so here we also only cover the intra-procedural part of sign analysis.

.

The presentation will again be heavy on the notational side, though the analysis is rather mundane. Actually, at it's core, it 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, we still want to comment on the nature of the sign analysis. On 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 opportunity 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 monotone 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 try to find out what values there are, but it tries to find out an **abstraction** thereof. Namely whether 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 analyses. 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 which 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 (again not even approximatively).

This is **different here**. Tracing information about signs means, there is approximative information about data. It's in some rather 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, as we are interested in interprocedural analysis of procedures, it's not 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 $\sigma$ 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 here.

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 equations (2.39) and (2.40).

Two plausible **lattices** for sign-analysis:

$$\mathcal{L}_{sign} = 2^{\mathbf{Var}_* \to \mathbf{Sign}} \tag{2.39}$$

$$\mathcal{L}'_{sign} = \mathbf{Var}_* \to 2^{\mathbf{Sign}} \tag{2.40}$$

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, that should make the difference clear.

*Example* 2.5.12 (Different levels of abstraction). Assume two variables $x$ and $y$ and the following situation in the lattice $\mathcal{L}_{\mathbf{Sign}}$:

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

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

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

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

Now, we work with the more precise one. There is no particular reason for that, we could tell the same story, adding embellishement, lifting the transfer functions, integrating an interprocedural part of the analysis, also with the less precise one. After all, we are discussing contexts and interprocedural analysis; the sign analysis is just a starting point and example.

**Definition 2.5.13** (Unembellished sign analysis)**.** Let's define as **lattice** of the sign analysis:

$$\mathcal{L}_{sign} = 2^{\mathbf{Var}_* \to \mathbf{Sign}} \qquad \text{where} \qquad \mathbf{Sign} = \{-, 0, +\}$$

We assume the following semantic function on *expressions:* $[\![_]\!]^{\mathcal{A}_{sign}}_{_} : \mathbf{AExp} \to (\mathbf{Var}_* \to \mathbf{Sign}) \to 2^{\mathbf{Sign}}$. Let's also define the transfer function

**Definition 2.5.14** (Transfer function for assignments)**.** The **transfer function** for $[x := a]^l$ is given as follows

$$f_l^{sign}(Y) = \bigcup\{\phi_l^{sign}(\sigma^{sign}) \mid \sigma^{sign} \in Y\} \tag{2.41}$$

where $Y \in 2^{\mathbf{Var}_* \to \mathbf{Sign}}$ and

$$\phi_l^{sign}(\sigma^{sign}) = \{\sigma^{sign}[x \mapsto s] \mid s \in [\![a]\!]^{\mathcal{A}_{sign}}_{\sigma^{sign}}\} \tag{2.42}$$

The above definition of the **transfer function** proceeds, resp. is written in *3 steps*: At the core is the semantic function $[\![_]\!]$. This function is for *expressions,* and is already non-deterministic. Equation (2.42) reflects the effect of an assignment for one abstract state and equation (2.41) is finally the transfer function. Note in passing, it's another example of lifting: the function $\phi$, defined in equation (2.42) per "sign-state", is lifted to work on sets of state in the obvios manner (just on each individual). One also says, $\phi$ 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, there is not even a mentioning of $\Delta$ here. The real work is done in $\phi$: the overall input to that function is $Y$, which is a sets of states, and $\phi_l^{sign}$ just applies it pointwise, interpreting the expression on the right-hand side of the assignment and updating the state accordingly.

In the following, we will *embellish* the analysis. Since we are not yet in the inter-procedural part, the embellishment is not very interesting, just a "lifting" to the embellished setting. We know how that work already, abstractly (see equation (2.35)).

As mentioned before, the abstract lattice $\hat{\mathcal{L}}$ is of the form $\Delta \to \mathcal{L}$. Here the underlying lattice $\mathcal{L}$ 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 (2.43), 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 (2.44) below is not in the form of a higher-function as done generally before, in equation (2.35).

Let's proceed and embellished the sign-analysis. But first, in the next example, we focus on the **intra**-procedural part, the part where nothing actually happens.

*Example* 2.5.15 (Embellished sign-analysis: intra-procedural part)*.*

$$\begin{aligned} \hat{\mathcal{L}}_{sign} &= \Delta \to \mathcal{L}_{sign} \\ &= \Delta \to 2^{\mathbf{Var}_* \to \mathbf{Sign}} \simeq 2^{\Delta \times (\mathbf{Var}_* \to \mathbf{Sign})} \end{aligned} \tag{2.43}$$

And the transfer function for assignments $[x := a]^l$ looks as follows:

$$\hat{f}_l^{sign}(Z) = \bigcup\{\{\delta\} \times \phi_l^{sign}(\sigma^{sign}) \mid (\delta, \sigma^{sign}) \in Z\} \tag{2.44}$$

□

The transfer function is a special case of the intra-procedural embellished transfer function from equation (2.35) we have seen earlier, here exploiting uncurrying, i.e, without that "higher-order flavor" from before. The uncurrying is indicated in equation (2.43) in using $2^{\Delta \times (\mathbf{Var}_* \to \mathbf{Sign})}$ instead of the equivalent $\Delta \to 2^{\mathbf{Var}_* \to \mathbf{Sign}}$ (the equivalence is written as $\simeq$).

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 (2.44) just does **nothing** with the contexts $\delta$, since we are still **within** a single process. In the following we go to the inter-procedural 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.

#### 2.5.6.5 Inter-procedural part

Now, finally, we start taking the contexts seriously, namely treating the flow in an out from procedure calls. For **procedure definitions** (with syntax $\texttt{proc}(\texttt{val}\, x, \texttt{res}\, y)\ \texttt{is}^{l_n}\ S\ \texttt{end}^{l_x}$ as introduced earlier), we assume trivial transfer functions at the entry and the exit of procedures:

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

In the further discussion, we just assume a forward analysis, instead of keeping it parametric (and the sign-analysis we use for illustration is a forward analysis). Now to **procedure calls**. In connection with calls, we introduced new (hyper-)edges of the form $(l_c, l_n, l_x, l_r) \in IF$. There are 2 transfer functions for calls resp. returns and correspondingly 2 sets of equations in connection with those edges.

**2 transfer functions** For calls we assume a transfer functions $\hat{f^1}_{l_c} : (\Delta \to \mathcal{L}) \to (\Delta \to \mathcal{L}) = \hat{\mathcal{L}} \to \hat{\mathcal{L}}$, and for returns, a transfer function $\hat{f^2}_{l_c,l_r} : (\Delta \to \mathcal{L}) \times (\Delta \to \mathcal{L}) \to (\Delta \to \mathcal{L}) = \hat{\mathcal{L}}^2 \to \hat{\mathcal{L}}$. Given those two transfer functions, the corresponding constraints are:

$$A_\bullet(l_c) = \hat{f^1}_{l_c}(A_\circ(l_c)) \tag{2.45}$$

$$A_\bullet(l_r) = \hat{f^2}_{l_c,l_r}(A_\circ(l_c), A_\circ(l_r))) \tag{2.46}$$

There are two constraints in connection with a procedure call, one for calling and one for returning. The one for returning is the more interesting one. It has **two** arguments, unlike all the other transfer functions. This is basically the place which ties together the caller and the callee and where the embellishment and contexts play their role. 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).

Figure 2.7 illustrates the situation for procedure calls.

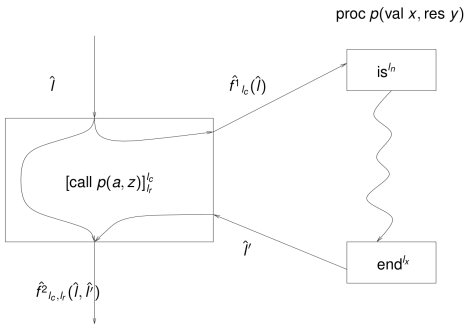



Figure 2.7: Transfer function for procedure calls

Note: *we* write $\hat{d}$ for elements of the embellished lattice (not $\hat{l}$ as in the pic). We mentioned earlier 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.

We will see that more concretely when working with a *concrete* embellishment, like for instance call-strings in Section 2.5.6.6.

#### 2.5.6.6 Context sensitivity

So far most is just general, we just sketched the form of constraints in connection with calls and returns. The only half-way concrete piece was that we introduced sign analysis, but so far we looked only at the intra-procedural part for that. Basically that meant that the context information was left untouched, i.e., lifted to the embellished setting. But what was *not* covered is what to concretely use for contexts. The purpose of the contexts is to **match returns** to corresponding **calls**. Generally, a context is used to identify the place where a call is done or at least contains information about potential call-sites. When analysing one procedure body —the interprocedural part— the context information is unchanged, as discussed, as all analysis is done for one particular call, i.e., one particular context. However, the context for a caller (which itself has been called from some other place) and a callee typically is different, and "interprocedural" constraints that mention both the caller and callee context is the way to match caller and callee analysis.

The context information may be precise enough to identify exactly the caller and to match exactly calls and returns. This leads then to a solution that corresponds to *MVP* (meet-over-all-valid-path). But one can use less specific information, which leads to less precision and more over-approximation. On the other end of the spectrum: if procedure bodies analysed *combining* flow information from **all** call-sites, that corresponds to **context insensitive** analysis, i.e., an analysis without contexts.

In the following, we will discuss **2 specializations**, i.e., two particular choices for context information.

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

By "control" we mean that the context identifies the place where the call is done. And "data" means information about the arguments. Of course one can imagine to do a combination of both kinds of information. Combinations of the two approaches, however, are not covered in the lecture. The call-strings corresponds more or less to the previously sketched MVP approach.

**Call strings** Call strings uses, as mentioned, control information to specify the call-site. Calling a procedure in a program, which in turns calls another etc. leads to the notion of a run-time stack, which grows and shrinks as the program progresses with calls and returns. That's mirrored in the analysis, where a call-string context corresponds to the sequence or stack of currently "active" calls. So the context a procedure body that has been called from a location contains not just that call site from the caller, but the caller itself has been called from a different context (presumably different call site but definitely a shorter call-string context). So the context here is a list of locations representing the stack of open calls and thus is (some form of) *path*, though it does not correspond exactly to the definition of paths we used earlier.

Let's fix the form of data that is known as call strings.

**Definition 2.5.16** (Call strings as data structure)**.** Assume flow-edges of the form $(l_c, l_n)$. The call-string information concentrates on calls, i.e.,labels of the form $l_c$ is recorded:

$$\Delta = \mathbf{Lab}^* \qquad \text{call strings} \tag{2.47}$$

The *extremal* value from the embellished lattice $\hat{\mathcal{L}} = \Delta \to \mathcal{L}$ is

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

The definition of $\hat{\iota} : \hat{\mathcal{L}}$ should be clear: at the beginning of the program, there are no calls; hence the call string is empty (represented by $\epsilon$). And $\iota$ is the "initial flow information" from the underlying monotone framework, that "seeds" the analysis at the initial resp. final nodes.. Note again the higher-order approach. The $\hat{\iota}$ is defined again point-wise.

While that may all look complex, it actually is not, resp. when seeing it in a concrete example, it should be clear enough.

*Example* 2.5.17 (Call strings for the Fibonacci example). Cf. the CFG for the Fibonacci example from Figure 2.6. For that example, the following lists a few **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 for call strings as contexts**

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

$$\begin{aligned}(\hat{f}^1_{l_c}\ \hat{d})\ [\delta; l_c] &= f^1_{l_c}(\hat{d}(\delta)) \\ (\hat{f}^1_{l_c}\ \hat{d})\ _ &= \bot\end{aligned} \tag{2.48}$$

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

$$\begin{aligned}(\hat{f}^2_{l_c,l_r}\ (\hat{d}, \hat{d}'))\ \delta &= f^2_{l_c,l_r}(\hat{d}(\delta), \hat{d}'([\delta; l_c])) \\ (\hat{f}^2_{l_c,l_r}\ (\hat{d}, \hat{d}'))\ _ &= \bot\end{aligned} \tag{2.49}$$

- 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 $\delta$) 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 (2.48). The "lifting" of $f_{l_c}$ is done analogously the same higher-order way as we have seen in the intraprocedural case (in equation (2.35). But the higher-order application of the underlying function is *not quite identical*. In equation (2.35), the context $\delta$ does not play a role at all: $(\hat{f}_l\ \hat{d})\ \delta = f_l\ (\hat{d}\ \delta)$. Now, for equation (2.48), the call-string context $\delta$ changes, that's the key thing. The change of $\delta$ 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 (2.49).

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 (2.48) and in (2.49) 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 (2.49).

Note the coupling.

In (2.48): we want to define the embellished transfer function $\hat{f}^1_{l_c} : \hat{\mathcal{L}} \to \hat{\mathcal{L}}$. Basically, it takes the unhatted function (assumed given) and applies it to $\hat{d}([\delta; l_c])$. Remember: $\hat{d}$ is a member of the embellished lattice $\Delta \to \mathcal{L}$. In other words: $\hat{d}(\delta)$ calculates the data $d$ in the original lattice using the calls string *without* the call. So the equation is to be interpreted as taking $\hat{d}$ as fixed, and then recur over the list of labels for a function of type $\Delta \to \mathcal{L}$. For the empty list, the function is undefined.

The original $f^1_{l_c}$ is defined where? Same for $f_{l_c,l_r}$, not really defined, they are assumed to exist.

The definition of the flow functions should be compared to the intra-procedural one, i.e., the pointwise lifting in equation (2.35) on page 70. In the point-wise definition, the $\delta$ is *not touched,* but just feeded into the hatted $\hat{f}$.

Here it is of course different, as we enter a call (for instance). As for the type: $\hat{f}_c : (\Delta \to \mathcal{L}) \to (\Delta \to \mathcal{L})$, and the question is, how to define that function. Perhaps it is better to separate the arguments: assume we have an $\hat{l}$ which is a member of the lattice $\hat{\mathcal{L}}$, i.e., a function. We are doing it forward, we are interested in how that affects $\hat{l}$, when entering the call.

After entering, it should still be a function $\Delta \to \mathcal{L}$, but now since we are in the procedure, the path is prolonged. So we still use the underlying, unembellished $f_c$, but now we insist that the path argument has the extended path. In some sense, one encodes the state (the path) in the argument of the function in a higher-order way.

For the *returns,* it works similarly. Remember that the second argument is the one from the *entry* of the call, i.e., in the extended context, whereas the first one is after/at returning, where the additional context is discarded again.

Interesting is equation (2.49). There, the match of the return with the corresponding call is done.

The case for calls is the simpler. We define the transfer function, and it's a "very" higher-order definition. Being a transfer function, the first argument is $\hat{l}$, an element from the embellished lattice, i.e., it's a function. which gives an element of $L$ depending on the

context argument. Now, the task is to encode the context-sensitivity: we need to define the flow exactly from a specific point (one can see it as the *pair* of location and the context, here the call string). So it "sets" the post-value for the $[\delta; l_c]$, but not for "others", therefore the else-part sets it to bottom. How is it defined? It needs to find the value of the lattice $L$ in the shorter context ($\hat{l}(\delta)$ and apply tje given "underlying" function $f^1_{l_c}$ to it.

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 (2.43)).

#### 2.5.6.7 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: $\hat{\mathcal{L}} \simeq 2^{\Delta \times (\mathbf{Var}_* \to \mathbf{Sign})}$ (see Eq. (2.43))
- before: standard embellished $\hat{f}^{\mathbf{Sign}}_l$ (with the help of $\phi^{\mathbf{Sign}}_l$)
- now: *inter-procedural*

We go back to the sign analysis from before. And before we come to look into the transfer function for the interprocedural setting, we have a look at how calls and return are treated in the unembellished setting, i.e., in context-insenstive manner.

*Example* 2.5.18 (Sign analysis: **underlying** transfer functions for calls and returns). For the transfer function other than the ones dealing with calls and returns, see example 2.5.13. We are still in the inembellished setting. For the definition of the transfer function for calls and returns, we are giving the definition of only the auxiliary functions $\phi$. Cf. equation (2.42).

We have to provide two transfer functions one for calling one for returning. They can be seen as abstract versions of **parameter passing** (passing the abstract sign data to the callee and back).

$$\begin{array}{rcl} \phi^{sign1}_{l_c}(\sigma^{sign}) & = & \{\sigma^{sign}[x \mapsto s][y \mapsto s'] \mid s \in [a]^{\mathcal{A}_{sign}}_{\sigma^{sign}},\ s' \in \{-, 0, +\}\} \\ \phi^{sign2}_{l_c,l_r}(\sigma^{sign}_1, \sigma^{sign}_2) & = & \{\sigma^{sign}_2[x, y, z \mapsto \sigma^{sign}_1(x), \sigma^{sign}_1(y), \sigma^{sign}_2(y)]\} \end{array}$$

To understand the transfer functions, one has to remember the conventions of the **syntax** of the language and the formulation of the operational semantics, the call-by-value-result passing mechanism. The **formal parameters** are $x$ and $, y$, where $y$ is the *result parameter*, the actual parameter is called $z$. □

The underlying functions $\phi$ 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 $[a]^{\mathcal{A}_{sign}}_{\sigma^{sign}}$. 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.

With the underlying transfer functions for calls and returns just given, let's proceed to make finally the analysis really *context-sensitive,* embellishing also the treatment of calls and returns.

*Example* 2.5.19 (Sign analysis: embellished $f$'s for call and returns with call-strings). The two embellished transfer functions for calls and returns for the sign-analysis and with call-string constext are given as follows:

$$\hat{f}^{sign1}_{l_c}(Z) = \bigcup\{\{\delta'\} \times \phi^{sign1}_{l_c}(\sigma^{sign}) \mid (\delta, \sigma^{sign}) \in Z, \delta' = [\delta; l_c]\} \tag{2.50}$$

$$\hat{f}^{sign2}_{l_c,l_r}(Z, Z') = \bigcup\{\{\delta\} \times \phi^{sign2}_{l_c,l_r}(\sigma^{sign}_1, \sigma^{sign}_2) \mid \begin{array}{l}(\delta, \sigma^{sign}_1) \in Z \quad \} \\ (\delta', \sigma^{sign}_2) \in Z' \\ \delta' = [\delta; l_c]\end{array} \tag{2.51}$$

Again, formal parameters are $x$ and $y$, the actual parameter is $z$. □

**calls: abstract parameter-passing + glueing calls-returns** Here we see the "uncurrying" we discussed before in action, exploiting the special form of the lattice for the sign analysis (equation (2.43)). 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 $\delta$ 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 $\phi^{sign1}_{l_c}$ in the case of calls in equation (2.50) extends $\delta$ from the caller to $\delta' = [\delta; l_c]$ in the callee. The same connecting of callee and caller is done for the returns in equation (2.51)

#### 2.5.6.8 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 with the call-strings is: turning the MVP approach as a instance of a monotone framework 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 $\mathcal{L}$ is finite (like in the 4 classical examples and also in the case of the sign analysis). The embellished lattice $\hat{\mathcal{L}}$ with call-strings, however, is definitely not finite. That means there's not guarantee 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 terminates, 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 guarantee of stabilization.

What can we do about? Well, after all that complex formalization, the idea is pretty simple and pragmatic. It simply sais; 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.

## 2.6 Static single assignment (not pensum)

### 2.6.1 Introductory remarks

This section is not covered by the book Nielson et al. [5]. It's added 2020 as it's an interesting and important angle on data flow analysis. [5] covers so-called **definition-use** or **use-definition** chains (aka du- and ud-chains), which are left out instead this time. Only so much for here: du- and du-chains are basically a slight variation or specialization of *reaching definitions*. Reaching definitions, as we know, calculates point by which definitions a point is reached and that's done for all points. Now, du-chains focus on uses, i.e., given a definition of a variable $x$ at some node, one is interested all points reched by it where it that assignment or definition is used. Points reached by the definition of $x$ but where $x$ is not used are of no interest. So indeed, it's just reaching definition, filtering out at the end non-interesting information, thus connecting a definition of a variable with places where it's used. That's a "forward" version of that information, called def-use or

du. There is also a "backward" version, connecting a point of use with all definitions that can reach the use. That's known as use-def (or ud). So basically, the concepts of def-use and use-def should be familiar (therefore we left out more formal details, and SSA can be seen as a generalization, in turn.

More than one ever wants to know can about SSA can be found in [4]. Early citations are Rosen et al. [7] Alpern et al. [1], and Cytron et al. [2]. So also historically, it's connected to value numbering.

- 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*.

**Tangential remarks** (Purely) functional languages and declarative language treat variables differently resp. more restrictively than imperative languages. The content of variables cannot be changed, variables cannot be re-assigned to. Variables can be introduced or defined only once, and later only used, when the value of the variable is neeeded. In this sitation, also the word "definition" of a variable is more appropriate. When introducing reaching definition analysis, we clarified, that a definition of a variable should probably be better called *assignment* and reaching definitions analysis maybe better called reaching assignment analysis. In a functional setting, since a variable is given a value only only once, it seems more appropriate to call that the variable's definition. We have written assignments like $x := a$ (though many languages would write $x = a$), but for functional languages, variables are often defined using `let`, as in for instance in `let` $x = 4$ `in` $e$. $x$ can be used in $e$ but not changed, so it's fair to say that $x$ is defined as 4 in $e$.

Of course one encounters situations like the following

```
let x = 4 in
    ...
    let x = "abc"
    in
          print x
    end
end
```

There is a nested let-situation, the outer let defines $x$ to be 4 in the surrounding **scope** and in the inner scope $x$ is a string. But that does *not* count as re-assignment, rather than that there are two different $x$'s

Now, with this regime, one can say, functional and declarative languages are **single assignment**. Same actually for logical formalisms and the use of variables in math (they are declarative, not imperative). Though one normally does not functional language as

being "single-assigment", as one feels there's not assignment statement at all, just `let` or similar definitorial mechanisms.

One way of characterizing purely functional or declarative languages as enjoying what's called **referential transparency**. It refers to an 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* (inside its scope).

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).

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, as said, 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" $v$ or is a name under which $v$ is known. 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 and/or make it more efficient. 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: SSA is an **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 use-def chains, mentioned above. 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.

### 2.6.2 A trivial starting point: straight-line code (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.

In this section we are using not the while language from earlier, 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 plausible form of 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).[10] 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.

[10] 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).

Let's look at the code from Listing 2.3. It's some 3AIC not in SSA format. The mentioned "numbering" trick to turn it to SSA should be obvious. See Listing 2.4. 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.

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

Listing 2.3: Three-address (intermediate) code (3A(I)C)

As a side remark: the variables $x$ and $y$ are not assigned at all and are use in that block of SLD "read only". Perhaps they are *input* variables of a procedure, or they are assigned to *"before"* the program (perhaps in a previous block of the control-flow graph.

```
a1 := x + y
b1 := a1 - 1
a2 := y + b1
b2 := x * 4
a3 := a2 + b2
```

Listing 2.4: 3AIC in SSA

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 times: for static analysis in general, 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 optimization and optimal compilation based on that is sometimes called *super-optimization*. Optimization is often understood as doing it better than without optimization, so omptimization is nothing else than an imporovement. But super-optimization is meant as getting an absolutely best result (given some critera). That word is of course, doubly-stupid. Optimization, as said, in compilers and static analysis does not mean getting something optimal (that being undecidable in general). To call the situation for SLC, where it may be obtainable "super-optimal", i.e., better than optimal, is pretty silly...

#### 2.6.2.1 3AC and temporaries

We are still discussing SSA in the almost trivial setting of SLC. It's not 100% trivial, one still needs some easy *data flow analysis* needed to get a fitting "versioning" for left-hand sides, a *one-pass* walk-through the code would suffice, as there are no loops. Only later it well get more complex. Before we proceed to the more general and more interesting setting, let's digress a bit and shed more lights on **temporaries** (as they also typically are used in a **single-assignment** manner.

As said, there's a difference between single-assignment and static single-assignment, but for SLC, there's no diffence as any line of code in stretch straight-line is executed exactly one time.[11]

The idea of temporaries straigtforward. They are variables introduced to store intermediate results. They store it only "for a while" (= temporarily), only as long as needed to evaluate an compound expression. Temporaries are therefore variables with a short live-span (live-span in the sense of the concept of **liveness**, as in liveness analysis). Why does the compiler need extra-variables (beyond those used by the programmers) to store intermediate results? Well, there comes a time in the chain of stages, where the compiler comes closer to actually generating code, and platforms cannot directly calculate complex expressions, like the one shown in Figure 2.8. That's the motivation to work with intermediate code formats, like 3AIC.

Figure 2.8 shows the abstract syntax tree for a simple compound expression. The leaves contain the basic expression (numbers a variables). The inner nodes contain compound (sub-)-expression and the root the overall expression `2*a+(b-3)`.

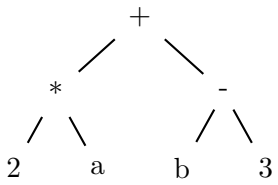



Figure 2.8: `2*a+(b-3)`

Listing 2.5 shows a plausible way intermediate code could like using a 3-address format. An intermediat code generater could generate temporaries $t_1$, $t_2$, $t_3$ to stored the intermediate results where the temporaries correspond to the inner nodes of the abstract syntax tree.

```
t1 = 2 * a
t2 = b - 3
t3 = t1 + t2
```

Listing 2.5: Three-address code with temporaries

[11] Seen globally, that may not be the case, if that piece of SLC is part of a loop etc. But of one does a local analysis for that SLC in isolation, single-assignment and static single assignment is the same.

3-address code is a popular form of linear intermediate code or an intermediate representation, closer to machine code. It characterized by the fact that each line can have only 3 variables (or less of course), correspond to 3 places or addresses in memory. The temporaries can be viewed as some abstraction on the concept of registers. As mentioned, temporaries have a short livespan, hence it's a good idea, to keep them into registers (and to have a live-variable analysis that figures out when a temporary's short live is over to recycle the register perhaps for the next temporary. Of course also user variables profit from being kept in temporaries, it's only that often the urgency and the return-of-investment of having in particularly temporaries in registers is perhaps higher.

It's an abstraction of registers, insofar that the intention is to keep intermediate values in registers. However, unlike for registers, one poses no bounds on how many temporaries one can use. So the (intermediate) code generator has a ulimited amount of temporaries (or "abstract registers") available, which is the reason it can use a fresh one for each intermediate result, and, as a consequence, the reason why temporaries follow a regine of **single assignment.**

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.

**Static** single assignment (and use also for user-level variables) just pushes that idea a bit further and makes sure that also ordinary variables 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, as illustrated in the code of Listing 2.5. The problem starts with conditionals (and loops). That will be treated next, and leads to the introduction of so-called Φ-functions.

### 2.6.3 Join points and phony functions

As said, the trouble starts with conditional (and loops). That's when we are no longer dealing with straigh-line code but places where the control-flow branches. Actually, it's not the branching which causes headache, it's more that when there is branching in a CFG, there are typically, though not necessarily, also places where the control-flow joins again.

We can illustrate the problem with a simple condition (i.e. in a situation typically not at 3AIC, as 3AIC normally does not support structured programming but works with (conditional) jumps instead. But also in that case, the CFG will show branching and joining, so the problem is the same.

Assume the following pseudo-code

```
if (z > 0)
then x := 1
else x := 2
fi;
y := x
```

The code is not in SSA format, as there are two assignments to $x$. Renaming them to $x_1$ and $x_2$ turns it into SSA-format, but we run into problems at the **join point** after the condition. The assignment $y := x$ makes no longer sense, after renaming $x$ into $x_1$ and $x_2$. And we statically don't know whether it should be $y := x_1$ or $y := x_2$. Actually it typically makes no sense to even assume it should (statically) correspond to either $y := x_1$ or else $y := x_2$, as it depends on the situation at run-time. In the example, in the value of $z$ in a particular situation.

The way out of that seems like a trick, it's the famous $\Phi$-function, a key ingredient of the SSA format. We simply say, the value of $y$ correponds to either $x_1$ or $x_2$, but statically it's not determined which it is. But it's fair to say that $y$ equals $\Phi(x_1, x_2)$, expressing that dependency of $y$ on $x_1$ and $x_2$. In order to take accound of that, one changes the code, introduces a new value $x_3$, the third value of $x$, besides $x_1$ and $x_2$, and sets $y$ to $x_3$, via $y := x_3$. That's illustrated in Figure 2.9

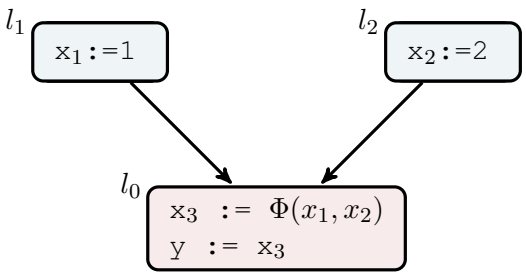



Figure 2.9: Join point and phony function

Note that the resulting small example with its *three* assignments to variables $x_1$, $x_2$, and $x_3$ is in SSA format. Note also that the the original code had only 2 assignments, both to $x$.

A core ingredient of SSA is use phony functions in assignment like $x_3 := Phi(x_1, x_2)$ from the small examples, where needed.

To place such phone functions for all nodes, which can be reached by more than one assignment to a variable in question, i.e., all nodes where there is ambiguity with respect to "reachable definitions" is wasteful. A straightforward improvement it: place $\Phi$-functions **only at join nodes**. But that's not the end of it. In the following we described how one can further improve the strategy. But first summarize the core of SSA.

**SSA in a nutshell**

SSA = using $\Phi$ + place $\Phi$-nodes smartly/economically + variable (re)naming scheme

The $\Phi$-construction also also called a **phone** function. As said before, it's a function that can be applied and evaluated statically, at compile time, like a macro. It's sometimes also called a "non-standard" function, it makes a decision based on (part of the) the control-flow. In the two-value dependency from the example, it reacts differently in a join-node depending on whether the last step in the execution comes "from the left or from the right". One may treat $\Phi$ function as "virtual", introduced as a kind of trick for purpose of analysis only, so that one obtains code in SSA format (that's what we are doing). One also could (later) implement them for real, i.e., generate code for the $\Phi$'s.

Turning a piece of code into SSA consists of a **two phase** algorithm, with the following phases in this order

1. strategical placement of $\Phi$-functions
2. renaming of variables

The main challange is a good **placement** of the $\Phi$ functions. The "**brainless"** SSA placement would place them everywhere, resp. as mentioned, to place them at all join nodes.

**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 $\geq 2$

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

That would guarantee SSA format (while preserving the semantics), so it's a **sound** approach. But it's an over-generous placement and undesirable. It leads to costly extra computations and subsequent analyses may suffer *loss of precision*.

Luckily there is **room for improvements**. Two obvious improvements are the following. One does not need $\Phi$-assignments for variables which no longer are **live.** In the previous example, if $x$ is not live at resp. after the join-node, there's not need to introduce $x_3$ which likewise would be dead.

A second improvement would be if (at a join point) there is a dependency "coming from the left or from the right" (in a join with 2 incoming edges) but it would not make a difference, as in $x_i \leftarrow \Phi(x_j, x_j)$ Obviously one would not need a phony function for that.

That's two improvements. In the following we sketch a more general one, based on graph-theoretic concepts, so we present an efficent general algo for $\Phi$-placement based on the notion of

**dominance**

**Core idea or observation:** Assume an 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 $n_2$.

The situation is illustrated in Figure 2.10

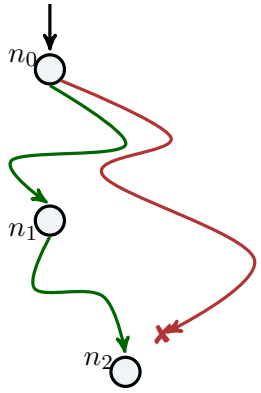



Figure 2.10: Illustration of dominance-based placement for $\Phi$

The idea is connected with the notion of domination. It's a general notion for directed graphs with some "start node". That's the form of CFGs, which are directed graph and do one one entry node (and one exit node, that is not relevant now). Also not relevan is the content of the nodes or basic blocks, as dominance comes generaly from graphs (not from CFGs where nodes carry some instructions).

**Definition 2.6.1.** Node $n_1$ **dominates** $n_2$, if all paths from entry $n_0$ to $n_2$ must pass through $n_1$. $n_1$ **stricly** dominats $n_2$ if $n_1$ dominats $n_2$ and $n_1 \neq n_2$. We write $dom(n)$ for the set of **dominators** of $n$, the set of nodes that dominate $n$.

The dominators can be determined by a simple data flow analysis (*must* and *forward*).

The above core idea told us where **not** to put $\Phi$ for a given assignment at $n$, namely at all nodes being dominated by $n$. All those nodes being dominated are a "safe zone", no need to worry about $\Phi$. Conversely, the undominated nodes are the "**danger zone**". We could then place $\Phi$ at all (join) nodes in that danger zone and that would be sound, but again we can do better.

We the placement algo does is to not cover all join nodes in the danger zone by a $\Phi$-function, but only the *first* ones, when entering the danger zone. I.e., when one starts at at a given node, the node dominates itself (though not strictly), following the edges, the subsequent node may still be dominated, and the the next one still, so all those nodes

are still in the "safe zone". But following the edges further there may come a point when one hits a non-dominated node, i.e., one enters the danger zone. And if there is not loop-back, one will stay in the danger size. If a node is undominated, also its successor is (unless there's a loop). In that picture, it suffices to place a Φ-assignment **at that very first node when entering the danger zone**. If that's covered by a phony function, subsequent nodes are as well (again loops need special consideration, as the concept of "subsequent nodes" is not really appropriate in a loop . . . ).

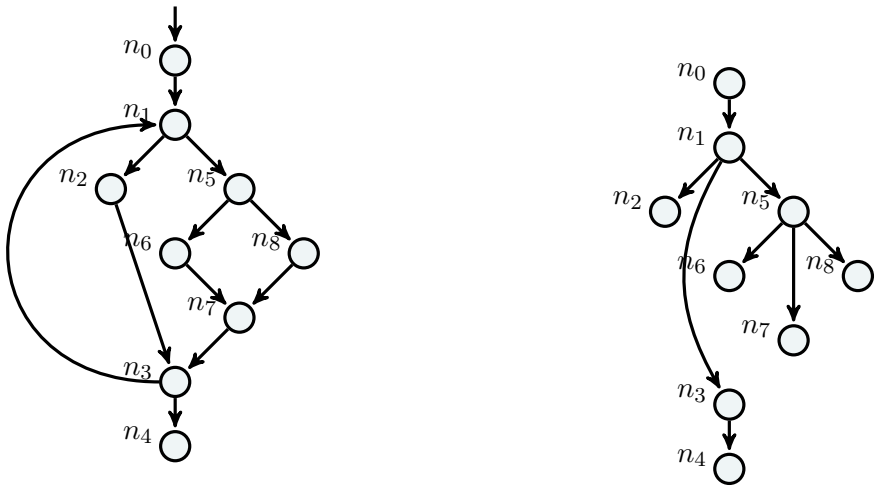
Anyway, the first nodes in a danger zone (i.e., undominated by a given node), that's known as **dominance frontier** of the node under consideration.[12]

**Definition 2.6.2** (Dominance frontier)**.** The *dominance frontier* $df(n)$ of a node is the collection of $m$ s.t.:

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

The notion of strict (non-dominance mentioned in the condition is needed to cover loops properly. Note also that the dominance frontier contains *join* nodes, only.

In connection with dominance, the concept of *dominator trees* is also useful. A node $n$ may have many strict dominators (or none), they are written as $dom(n)$. If we focus on the (strict) dominator closest to $n$ (if it exists), then that one is unique. Only the entry node of the graph has no immediate strict dominator. Because of that, the structure relating nodes with nodes they immediately dominate is a tree, the **dominator tree** (we can write $idom(n)$ to that immediate (strict) dominator, it's the parent node in the dominator tree. In the figure, the edges of the graph go from parent to children nodes. where the parent is the the $idomn$ of a child $n$.




(a) Some sample (control-flow) graph

(b) Dominator tree

Figure 2.11: Graph and dominator tree

[12] Note: the word "danger zone" is not a technical term from graph theory or SSA, but dominance and dominance frontier are.

The following shows in pseudo-code an algorithm to calculate the 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)
```

Listing 2.6: Dominance frontier

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 $\rightarrowtriangle^*$ 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 $\Phi$-functions, whereas the undominated territory is the "danger zone" where $\Phi$ 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 $\Phi$.

Now to the algorithm. The frontier is calculated for all nodes which have more than one predecessor ($\Phi$-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* 2.6.3 (dominance frontier for the previous graph).

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

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. □

Finally, a possible way to rename the variable in the second phase after the placement has been done

```
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])
```

Listing 2.7: Rename

# Bibliography

[1] Alpern, B., Wegman, M. N., and Zadeck, F. K. (1988). Detecting equalities of variables in programs. In [6], pages 1–11.

[2] 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.

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

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

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

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

[7] Rosen, B. K., Wegman, M. N., and Zadeck, F. K. (1988). Global value numbers and redundant computations. In [6], pages 12–27.

# Index

access link, 55
activation record, 54
analysis
    context-sensitive, 65
    embellished, 66
ascending chain condition, 38
available expression, 5
available expressions, 5

backward analysis, 9

call graph, 56
call string, 66
call-by-result, 46
calling conventions, 54
chaotic iteration, 40, 41
code generation, 89
common subexpression elimination, 4
common subsexpresion elimination., 4
complete lattice, 20, 38
complete path, 62
configurations, 49
context, 67
context-sensitive analysis, 65
control flow
    reverse, 3
control flow graph, 1
CSE, 4
currying, 69

dead code, 56
derivation sequence, 19
dynamic link, 55
dynamic memory, 45

embellished analysis, 66

$\hat{f}$ (embellished $f$), 68

hoisting, 10, 11

IF, 48
intraprocedural analysis, 1
isolated entry, 2

kill and generate, 6

label consistency, 3
lattice, 20
    complete, 38
location, 49

monotone framework, 37
MVP, 63

optimization problem, 23
order relation, 20

parameter passing, 46
partial order, 20
path, 55
    complete, 62
    valid, 62
path insensitivity, 73
path sensitive analysis, 57
program transformation, 11
property space, 38

reaching definitions, 8
referential opaqueness, 85
referential transparency, 84, 85
reverse control flow, 3
run-time environment, 45

sign analysis, 72
SOS, 16
stabilization, 38
stack frame, 54
static link, 55
static memory, 45
structural operational semantics, 16

temporaries, 86

uncurrying, 69
under-approximation, 5
unique entry, 2

valid path, 62
very busy, 11

worklist, 41
worklist algorithn, 41