Course Script

Static analysis and all that

IN5440 / autum 2018

Martin Steffen

Contents

Chapter 1 Introduction

What is it about?

Learning Targets of this Chapter

Apart from a motivational introduction, the chapter gives a high-level overview over larger topics covered in the lecture. They are treated hear just as a teaser and in less depth compared to later but there is already technical content.

Contents

1.1 Motivation

1.1.1 General remarks

Static analysis: why and what?

- what
  - *static*: at “compile time”
  - *analysis*: deduction of program properties
    - automatic/decidable
    - formally, based on semantics
- why
  - **error catching**

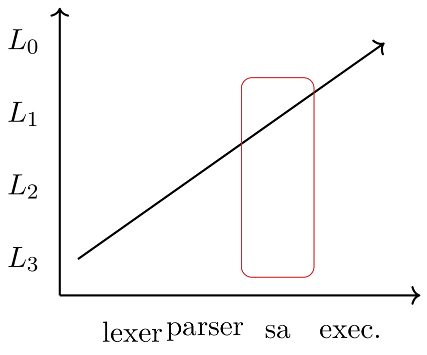
- catching common “stupid” errors without bothering the user much
- spotting errors early
- certain similarities to model checking
- examples: type checking, uninitialized variables, potential nil-pointer deref’s, unused code

- **optimization**: based on analysis, transform the “code”[1], such the the result is “better”
  - examples: precalculation of results, optimized register allocation ...

The nature of static analysis

- compiler with differerent *phases*
- corresponding to *Chomsky’s hierarchy*
- **static** = in principle: before run-time, but in praxis, “*context-free*”
- since: run-time most often: undecidable

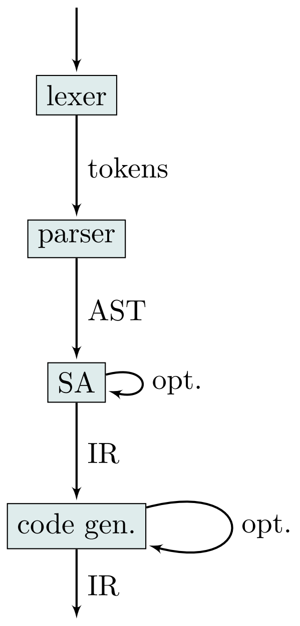
⇒ static analysis as **approximation**



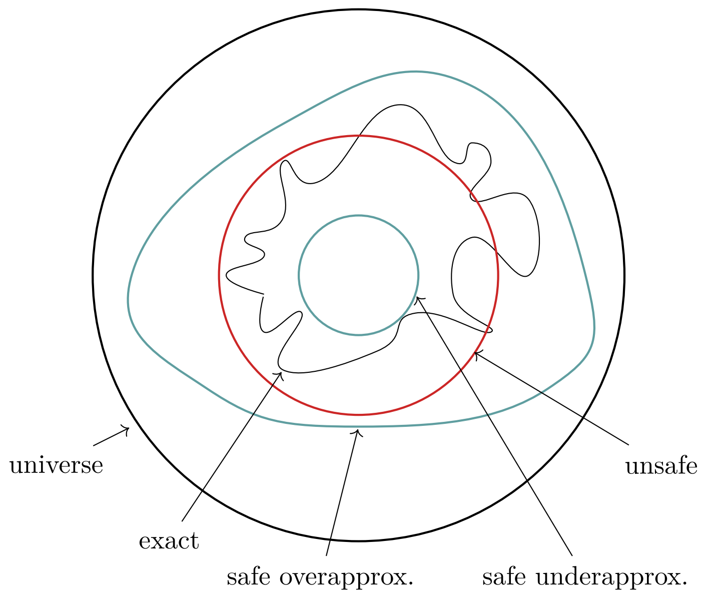
Concerning the notion of “static” analysis. Playing with words, one could call full-scale (hand?) verification ‘static” analysis, and likewise call lexical analysis a static analysis.

[1]source code, intermediate code at various levels

Phases



Static analysis as approximation



The figure is, of course, only an informal illustration. In general, program behavior is (for any non-trivial programming language) *undecidable*. It's a general fact ("Rice's theorem") that all non-trivial, semantical properties of programs are undecidable. Actually, there are exactly two properties which qualify as being trivial. That's the property "true" which holds for all progams, and "false", which holds for none. That means basically every single semantical property about programs is undecidable up front. *Semantical* properties are those that refer to the behavior of a programs, its semantics. Of course, *syntactical* properties *are* decidable, even if they are non-trivial in the above technical sense.

Optimal compiler?

Full employment theorem for compiler writers It's a (mathematically proven!) fact that for any compiler, there exists another one which beats it.

- slightly more than *non-existence* of optimal compiler or *undecidability* of such a compiler
- theorem
  - just states that there room for improvement is always *guaranteed*
  - does not say *how!* Finding a better one: *undecidable*

1.2 Data flow analysis

1.2.1 A simplistic while-language

While-language

- simple, prototypical imperative language
  - "untyped"
  - simple control structure: while, conditional, sequencing
  - simple data (numerals, booleans)
- abstract syntax $\neq$ concrete syntax
- disambiguation when needed: (...), or { ... } or begin ... end

Abstract syntax The given while-language here (and other languages later) is rather simplistic. This fact of being simple is, however, not the reason why we call the syntax here *abstract* (not like "it's a very abstract language, it has only 5 constructs"). Nonetheless, if we don't assume an upper bound on the memory, it is Turing complete.

In this lecture, we generally work with *abstract syntax*. That's different from concrete syntax. Remember from the phases of a compiler, that abstract syntax trees are typically the result of the parsing phase and the input of the static analysis phase (aka. semantical analysis). Focusing here on the semantic phase of a compiler, we assume that the lexer and parser have done their thing, and we start the considerations on abstract syntax. Abstract syntax is *specified* by context-free grammars, which is a formalism to specify structured trees. Thus we are completely not interested in *parse trees* (sometimes known as contrete syntax trees), or issues of precendence, associativity, etc. Things like that are covered for instance in the *compiler construction course* INF5110.

This program "written" in abstract syntax are thought of as *trees*. Since trees are notationally not easy to write down, we don't depict them as actual trees (even if that would be accurate). Instead, we use textual notations and to appeal to the understanding in which way that represents trees, and allow ourselves groupong constructs like parentheses (..) to disambiguate the tree structure, even if of course parentheses and similar constructs are general not part of abstract syntax trees; they are just needed to help the human reader here to understand the underlying tree structure.

Furthermore, and also as a general remark: the while language here (even in its restricted syntactic capabilities) resembles a “high-level” language, at least insofar that it supports “structured programming” in the form of while-loops, as opposed to conditional jumps. Later we might extend it also with further capabilites and programming abstractions, for instance, procedure calls. Anyway, techniques similar to the ones we cover, can also be applied to lower-level intermediate languages, not just abstract syntax of the source language. Also in that case, in a more formal account, one might start fixing the abstract syntax of the intermediate language for the programs one intends to analyze.

Types As mentioned, working directly with abstract syntax, we very much ignore syntactical questions. Besides that, and in particular for the while-language, we more or less ignore another issue, that of typing. Typing and type checking is one very important form of static analysis and will be covered in the course, just not really in connection with the while-language and its derivatives. The “type system” here is so impoverished, supporting basically only integers and booleans, that it's not much worth introducing a type system for doing that. Instead, sidestepping the question, we deal with it “syntactically”. *Arithmetic* expressions are represented by the nonterminal a, boolean expressions by b, and, as it happens, we have only variables of arithmetic type. In that sense the language *is* typed, only not very interestingly, and we assume the proper types have somehow been figured out already, so we concetrate for the while language on other things. For functional languages, resp. calculi (variations of the λ-calculus), we will take care of type checking and extensions of traditional type checking, as it becomes more challenging.

Labelling

- associate *flow* information

⇒ **labels**

- *elementary block* = labelled item
- identify basic building blocks
- consistent/unique labelling

Abstract syntax

$$\begin{array}{rcll} a & ::= & x \mid n \mid a \,\mathsf{op}_a\, a & \text{arithm. expressions} \\ b & ::= & \mathsf{true} \mid \mathsf{false} \mid \mathsf{not}\ b \mid b\,\mathsf{op}_b\, b \mid a\,\mathsf{op}_r\, a & \text{boolean expr.} \\ S & ::= & x := a \mid \mathsf{skip} \mid S_1; S_2 & \text{statements} \\ & & \mathtt{if}\ b\ \mathtt{then}\ S\ \mathtt{else}\ S \mid \mathtt{while}\ b\ \mathtt{do}\ S & \end{array}$$

Table 1.1: Abstract syntax

a	$::=$	$x \mid n \mid a \,\mathsf{op}_a\, a$	arithm. expressions
b	$::=$	$\mathsf{true} \mid \mathsf{false} \mid \mathsf{not}\; b \mid b \,\mathsf{op}_b\, b \mid a \,\mathsf{op}_r\, a$	boolean expr.
S	$::=$	$[x := a]^l \mid [\mathsf{skip}]^l \mid S_1; S_2$	statements
		$\texttt{if}\,[b]^l\,\texttt{then}\,S\,\texttt{else}\,S \mid \texttt{while}[b]^l\,\texttt{do}\,S$	

Table 1.2: Labelled abstract syntax

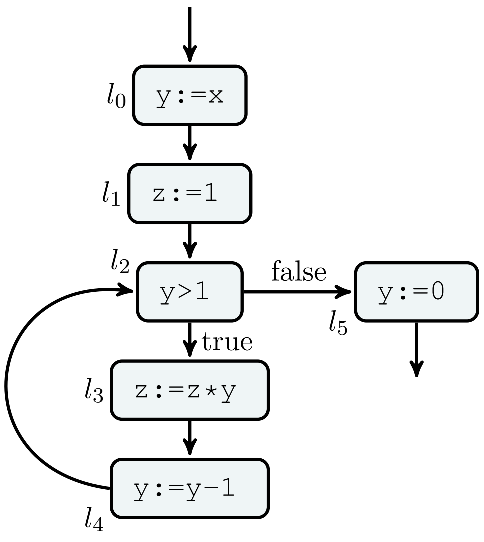
Example factorial

$$y := x; z := 1; \texttt{while}\;\; y > 1\,\texttt{do}(z := z * y; y := y - 1); y := 0$$

- input variable: x
- output variable: z

$$\begin{aligned}
&[y := x]^0;\\
&[z := 1]^1;\\
&\;\texttt{while}\;[y > 1]^2\\
&\;\texttt{do}([z := z * y]^3; [y := y - 1]^4);\\
&\;[y := 0]^5
\end{aligned} \tag{1.1}$$

CFG factorial



Control flow graph The factorial was used to illustrate the important concept of *control flow graphs*. Later in the lecture we will have another look in more detail how to actually *calculate* a control-flow graph, given a program in abstract syntax (i.e., in the form of an abstract syntax tree). Actually, it's not very complex; basically, one has to traverse the tree, "label" the nodes appropriately, which typically means, creating one new node of the graph for each encountered basic construct. In the illustrations here, the nodes are identified by unique labels $l_0, l_1, l_2 \ldots$ (in some way, the numbers serve themselves as identification). The above "labelled" abstract syntax from above is a different notation to illustrate the *nodes* of the control flow graph + the basic expression "contained in" or "associated with" the nodes (and the nodes of the graph are the same as the labels in the syntax notation). Missing in the labelled programm from equation (1.1) obviously are the *edges* of the graph, but, as said, it's not too hard to calculate them as well (while labelling the statements and traversing the syntax tree).

For participants of the course *compiler construction* (INF5110): the definition of the control flow graph was done there at a lower level, i.e., at some *intermediate code* (like three-address-code) without looping constructs, but conditional jumps instead. That intermediate code can be seen as a kind of "machine-independent machine language", i.e., an intermediate language already rather close to actual machine language, but not yet quite there. In any case, that intermediate language language had officially *labels* that could be used for conditional or unconditional *jumps* (which are like goto's). In other words, that intermediate language was "officially labelled" via a specific `label`-command (which was counted among the so-called *pseudo-instructions*). Because of that, that form of language almost immediately contains its control-flow graph, since the jumps corresponds obviously to the edges. More precisely, for conditional jumps, one of the edges is goes from the conditional jump-statement to the target label, the other edge is the "fall-through".

A final remark on the nodes and the code contained therein. In the lecture here, we make the assumption for the imperative language: "one node, one basic statement". In practice, it's often the case that one groups together sequences of statements like assignments together into larger block which contains no branching or jumps into it. Those larger blocks are sometimes called *basic blocks* and the grouping is done for efficiency. In the factorial example, l_0 and l_1 could be lumped together (as well as l_3 and l_4). While a good idea in practice, we don't care much here, as the principles of data flow would not don't change with this refinement.

Factorial: reaching definitions analysis

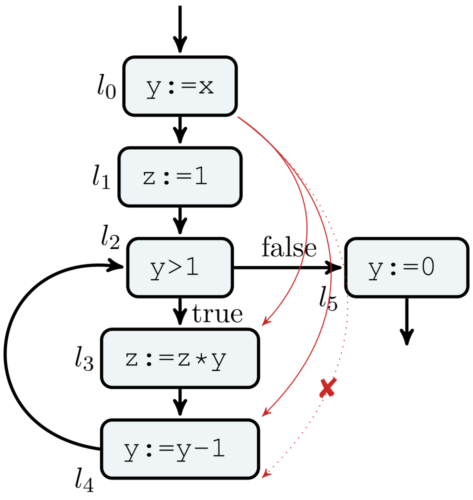
- "definition" of x: assignment to x: $x := a$
- better name: reaching assignment analysis
- first, simple example of **data flow** analysis

Reaching def's An *assignment* (= "definition") $[x := a]^l$ *may* reach a program point, if there *exists* an execution where x was *last assigned to* at l, when the mentioned program point is reached.

Reaching definitions The reaching definitions is just one of a family of similar analyses the lecture covers (and which themselves are only a small selection of many much more). It's a typical *data flow problem* and the techniques to solve this problem can be used analogously to a wide range of problem. This class of problems is known as *monotone frameworks* and goes back to **?**].

The data which is imagined to "flow" through the program, resp. rather flow through the control flow graphs as the current intermediate representation of the program, is not so much the actual program data (i.e., here, the integer values stored in the variables). It's rather *information about the data* that flows through the graph, which may be seen as an *abstraction*. Of course, there may be different kinds of information one could be interested in, and the choice determines the analysis. In case of the reaching definition analysis here, we are interested in the places, where variables are "defined", i.e., the nodes or labels where variables are being assigned to.

Factorial: reaching definitions



- data of interest: tuples of variable × label (or node)
- note: *distinguish* between *entry* and *exit* of a node.

Factorial Note in particular, that the *exit* of the node l_4 cannot be reached by the assignment to y in the node l_1, whereas the entry of that node l_4 may well be reached. That's indicated by the dotted, resp. the solid arrows. Not all points which are reachable via the choice of origin $(y, 0)$ are given in the figure, to keep it readable.

It should be also noted: accepting that the analysis works on tuples (x, l), we immediately see that the problem intuitively is decidable: there are only finitely many variables and finitely many nodes (= labels) in the control flow graph. One could analyze thereby the problem, by making some graph seach for all combinations of variables with start nodes and end nodes. That would be a very wasteful approach, and *data flow analysis* is about more efficient techniques.

Factorial: reaching assignments

- " **points** " in the program: *entry* and *exit* to elementary blocks/labels
- ?: special label (not occurring otherwise), representing *entry* to the program, i.e., $(x, ?)$ represents initial (uninitialized) value of x

- full information: pair of “functions”

$$\mathsf{RD} = (\mathsf{RD}_{entry}, \mathsf{RD}_{exit}) \tag{1.2}$$

- tabular form (array): see next slide

Factorial: reaching assignments table

l	RD_{entry}	RD_{exit}
0	$(x,?),(y,?),(z,?)$	$(x,?),(y,0),(z,?)$
1	$(x,?),(y,0),(z,?)$	$(x,?),(y,0),(z,1)$
2	$(x,?),(y,0),(y,4),(z,1),(z,3)$	$(x,?),(y,0),(y,4),(z,1),(z,3)$
3	$(x,?),(y,0),(y,4),(z,1),(z,3)$	$(x,?),(y,0),(y,4),\quad(z,3)$
4	$(x,?),(y,0),(y,4),\quad(z,3)$	$(x,?),\quad(y,4),\quad(z,3)$
5	$(x,?),(y,0),(y,4),(z,1),(z,3)$	$(x,?),(y,5),\quad(z,1),(z,3)$

The highlighted information in the table is not 100% consistently done. The intention is to highlight the information which is *generated* new in the corresponding block. That can be seen in the blocks (the lines in the table) corresponding to *assignments* (which are all except $l = 2$). At the exit of those blocks, a pair containing the corresponding label and the assigned variable is injected. That explains the highlights in the exit-column. Note that at label 3, whose block is side-effect free, the exit set corresponds to the entry set.

Now for the entry points: one needs to look at the graph, they are determined by the inter-block relations, i.e., the *edges*. In the easiest case (for instance $l = 1$), the set just coincides with the post-set of the predecessor block. Interesting is, of course, label $l = 2$, where the graph (interpreted forwardly) joins two arrows, i.e., the block 2 has *two predecessors*, 1 and 4. The question then is: is it the *union* or the *intersection* of the information. Intuitively: it must be *union*. That’s due to the nature of the analysis here: the question is, *may* an assignment reach a point in question.

Reaching assignments: remarks

- *elementary* blocks of the form
  - $[b]^l$: entry/exit information coincides
  - $[x := a]^l$: entry/exit information (in general) different
- at program exit: $(x, ?)$, x is input variable
- table: *“best”* information = *smallest* sets:
  - additional pairs in the table: still *safe*
  - removing labels: *unsafe*
- note: still an **approximation**
  - no *real* (= run time) data, no real execution, only *data flow*
  - *approximate* since
    - ∗ in *concrete* runs: at each point *in that run*, there is exactly *one* last assignment, not a *set*
    - ∗ *label* represents (potentially infinitely many) runs
  - e.g.: at program exit in concrete run: *either* $(z, 1)$ or else $(z, 3)$

Input variable As a result of the analysis, variable x is marked as ?. That can be interpreted as a sign that it's not assigned to. More precisely, the fact that there is the tuple $(x, ?)$ at the program exist *together* with the fact that there is *no other* tuple containing x indicates that x is never assigned to (since we are dealing with "may" information"). That's the reason why we can interpret it as *input variable*: It is often a convention that input variables are *not* assigned to inside the program, they are assumed to be given a value up front (indicated by ?).

If one analyses a function body —each function conventionally is represented by its own control flow graph capturing the control-flow of its body— the formal parameters are given their initial value from "outside" via parameter passing. It's often considered as bad style, at least when passing parameters by value, to assign to the formal parameters.

Of course, non-input variables which are not assigned to make no sense. Actually, if such variables occur, it may indicate a problem (unitialized variables) which may lead to random results or errors like nil-pointer exceptions.

For participants of the course "compiler construction": the situation about unitialized variables may be compared to the analgous situation for *liveness analysis*. There, the core analysis concentrated completely on analysing *one single node* or *basic block* (there called *local analysis*). Some variables where marked as *liveness status unknown* as they where locally not used, but concentrating on one local node only, one cannot determined whether from the global perspective they are live or dead. A special information (like ? here) was used to indicate that "locally unclear" situation. As a side remark: one difference of the liveness analysis works compared to reaching definition analysis is that it works *backward*. Liveness analysis will be covered also here later as one instance of the mentioned monotone framework.

Optimal solution Currently it may not yet been completely clear in what sense is the solution is minimal/optimal. Much of the underlying theory (covered later) is about assuring that such a minimal solution demonstratibly and uniquely exists (it will be based on the notion of lattices). Intuitively, the solution from the table is minimal (and thus best) in that we cannot *remove* one single entry of the table, without making it **wrong** (unsound, unsafe). Very intuitively, it can be compared also to the picture static analysis as approximation. An over-approximation is an area fully "covering" the actually, irregularly shaped behavior. Restricting to a certain class of analysis (for instance simple data flow, specifically say simple reaching definition) can be visualize by allowed overapproximation only in the form of a circle, let's say. The optimal solution is then the smallest such covering circle (which still is an opproximation). If one invests in a more detailed analysis, that might correspond to having, let's say, ellipses or six-edged polygons as (illustration of) allowed results of the analysis. In that case, potentially better and more precise approximations may be doable (typically at higher computational costs). It may also be the case, that no longer a single best solution exists in general. For instance, there may be *two* covering polygons, both of which cannot be made smaller without becoming unsound, but on the other hand incomparable (neither is contained in the other). In this lecture, we typically deal with settings were *optimal* solution do exists, though.

Data flow analysis

- standard: representation of program as control flow graph (aka flow graph)
  - nodes: elementary blocks with labels (or basic block)
  - edges: flow of control
- two approaches, both (especially here) quite similar
  - *equational* approach
  - *constraint-based* approach

1.2.2 Equational approach

From flow graphs to equations

- associate an **equation system** with the flow graph:
  - describing the "flow of information"
  - here:
    * the information related to reaching assignments
    * information imagined to flow forwards
- **solutions** of the equations
  - describe *safe* approximations
  - not unique, interest in the *least* (or *largest*) solution
  - here: give back RD of equation (1.2) on slide 9

Equations for RD and factorial: intra-block

first type: *local*, **intra**-block":

- flow through each individual block
- relating for each elementary block its exit with its entry

$$\text{elementary block: } [y := x]^0$$

$$\text{elementary block: } [y > 1]^2$$

$$\text{all equations with } \mathsf{RD}_{exit} \text{ as "left-hand side"}$$

$$\begin{aligned}
\mathsf{RD}_{exit}(0) &= \mathsf{RD}_{entry}(0) \setminus \{(y,l) \mid l \in \mathbf{Lab}\} \cup \{(y,0)\} \\
\mathsf{RD}_{exit}(1) &= \mathsf{RD}_{entry}(1) \setminus \{(z,l) \mid l \in \mathbf{Lab}\} \cup \{(z,1)\} \\
\mathsf{RD}_{exit}(2) &= \mathsf{RD}_{entry}(2) \\
\mathsf{RD}_{exit}(3) &= \mathsf{RD}_{entry}(3) \setminus \{(z,l) \mid l \in \mathbf{Lab}\} \cup \{(z,3)\} \\
\mathsf{RD}_{exit}(4) &= \mathsf{RD}_{entry}(4) \setminus \{(y,l) \mid l \in \mathbf{Lab}\} \cup \{(y,4)\} \\
\mathsf{RD}_{exit}(5) &= \mathsf{RD}_{entry}(5) \setminus \{(y,l) \mid l \in \mathbf{Lab}\} \cup \{(y,5)\}
\end{aligned} \tag{1.3}$$

Inter-block flow

second type: *global*, **inter**-block

- flow *between* the elementary blocks, following the control-flow **edges**
- relating the *entry* of each block with the *exits* of *other* blocks, that are connected via an *edge* (exception: the initial block has no incoming edge)
- *initial* block: mark variables as *uninitialized*

$$
\begin{aligned}
\mathsf{RD}_{entry}(1) &= \mathsf{RD}_{exit}(0) \\
\mathsf{RD}_{entry}(2) &= \mathsf{RD}_{exit}(1) \cup \mathsf{RD}_{exit}(4) \\
\mathsf{RD}_{entry}(3) &= \mathsf{RD}_{exit}(2) \\
\mathsf{RD}_{entry}(4) &= \mathsf{RD}_{exit}(3) \\
\mathsf{RD}_{entry}(5) &= \mathsf{RD}_{exit}(2) \\
\\
\mathsf{RD}_{entry}(0) &= \{(x,?),(y,?),(z,?)\}
\end{aligned}
\tag{1.4}
$$

There are 6 equations as there are 6 nodes. As far as the right-hand sides are concerned: there are *6 mentionings of* $\mathsf{RD}_{exit}(l)$. That indicates that the graph has *6 edges* (not counting the incoming edge into node l_0 and the outgoing edge at the exit).

The entry node has no internally incoming edge in this example (only one that is shown in the picture to come from “outside” which is therefore not part of the graph nor an edge, it’s just a conventional, graphical indication of the initial node). This is not a coincidence, insofar that one generally assumes that the control-flow graph does not have such an *initial loop*. In the unlikely event that the program would start immediately with a loop or similar, the compiler would arrange it so that there is an extra “skip” entry node (a *sentinel*). Alternatively, the generation of the control-flow graph might simply automatically add some extra sentinel, just in case.

Technically, nothing would go wrong allowing such initial loops, it’s just that the technical representation later (and the algorithms) are slighly simpler. Sure, not fundamentally simpler, only avoiding some extra corner cases, it just gets a tiny bit more smooth.

Later, we will encounter analyses, which work *backwards*, i.e., the flow will follow the edges of the flow graph in reverse direction. Live variable analysis is one prime example. In those cases, one tries to avoid “final loops” at the exit of the program.

For participants of “compiler constructions” (INF5110): that’s reminiscent of the treatment of context-free grammars for some constructions (like the LR(0)-DFA construction). For some constructions, it was assumed that the grammar’s start symbol, say S, does not show up on the right-hand side of any production. That would correspond to an loop back to the “intitial state” here. Since also there, one wanted to slightly simplify the treatment avoiding that case, the standard construction routinely simply added “another” start symbol S' and a production $S' ::= S$.

General scheme (for RD)

Intra
- for assignments $[x := a]^l$

$$\mathsf{RD}_{exit}(l) = \mathsf{RD}_{entry}(l) \setminus \{(x, l') \mid l' \in \mathbf{Lab}\} \cup \{(x, l)\} \tag{1.5}$$

- for other blocks $[b]^l$ (side-effect free)

$$\mathsf{RD}_{exit}(l) = \mathsf{RD}_{entry}(l) \tag{1.6}$$

Inter

$$\mathsf{RD}_{entry}(l) = \bigcup_{l' \to l} \mathsf{RD}_{exit}(l') \tag{1.7}$$

Initial l: label of the initial block (isolated entry)

$$\mathsf{RD}_{entry}(l) = \{(x, ?) \mid x \text{ is a program variable}\} \tag{1.8}$$

The equation system as fix point

- RD example: solution to the equation system = *12 sets*

$$\mathsf{RD}_{entry}(0), \ldots, \mathsf{RD}_{exit}(5)$$

- i.e., the $\mathsf{RD}_{entry}(l), \mathsf{RD}_{exit}(l)$ are the *variables* of the equation system, of *type*: sets of pairs of the form (x, l)
- *domain* of the equation system:
- $\vec{\mathsf{RD}}$: the mentioned twelve-tuple of variables

⇒ equation system understood as function F

Equations

$$\vec{\mathsf{RD}} = F(\vec{\mathsf{RD}})$$

Fix point equation The above fixpoint equation on vectors of RD variables may be broken down more explicitly 12 parts (the individual "equations") like for instance

$$F(\vec{\mathsf{RD}}) = (F_{entry}(1)(\vec{\mathsf{RD}}), F_{exit}(0)(\vec{\mathsf{RD}}), \ldots, F_{exit}(5)(\vec{\mathsf{RD}}))$$

After solving the equation system, we for instance could get as part of a solution:

$$F_{entry}(2) = (\ldots, \mathsf{RD}_{exit}(1), \ldots, \mathsf{RD}_{exit}(4), \ldots) = \mathsf{RD}_{exit}(1) \cup \mathsf{RD}_{exit}(4)$$

The least solution

- $\mathbf{Var}_* =$ variables "of interest" (i.e., occurring), $\mathbf{Lab}_*$: labels of interest
- here $\mathbf{Var}_* = \{x, y, z\}$, $\mathbf{Lab}_* = \{?, 1, \ldots, 6\}$

$$F : (2^{\mathbf{Var}_* \times \mathbf{Lab}_*})^{12} \to (2^{\mathbf{Var}_* \times \mathbf{Lab}_*})^{12} \tag{1.9}$$

- domain $(2^{\mathbf{Var}_* \times \mathbf{Lab}_*})^{12}$: *partially ordered* pointwise:

$$\vec{\mathsf{RD}} \sqsubseteq \vec{\mathsf{RD}}' \text{ iff } \forall i.\ \mathsf{RD}_i \subseteq \mathsf{RD}'_i \tag{1.10}$$

$\Rightarrow$ **complete lattice**

1.2.3 Constraint-based approach

This section is basically a rerun of the previous one: the constraint-based approach here is nothing much more than a variation of the equational approach. As for terminology, the distinction between "equational" vs. "constaint-based" here is slightly misleading. It's misleading insofar, as also the equational approach is based on constraints, namely *equational constraints*. In contrast, next we will use "inequations". In the RD analysis we are currently focussing on, the solutions are sets (of pairs (x, l)). Consequently, the inequations will be *subset* constraints $\subseteq$ on sets, instead of equality $=$ on sets.

In the current setting, a basic data flow problem for a simplistic imperative language, there's not much difference between the two approaches. Not only is it straightforward to replace the $=$ by $\subseteq$, and there you are. Also on a deeper level here, there are technical reasons resulting in the following fact: the best (here smallest) solution of the equational approach *coincides* with the smallest solution of the constraint-based approach. That means, for the purpose of program analysis, which is after the best, safe approximation, **both approaches are the same**!

The setting here with simple control-flow graphs, however, *is* indeed simple. In more complex setting (for instance functional languages with much more flexible control flow) it's not always the case that equational approaches and constraint based approaches give the same (best) analysis result. In general, the constraint based approach offers more flexibility (thus sometimes more precise analysis) in more complex settings, but not here. In the context of type and effect systems, we will encounter such more complex settings.

When stating that (here) the step from an equational to a constraint-based approach amounts to a trivial reformulation of the equations leaves, however, one question still to be answered:

> should $=$ be replaced by $\subseteq$ or $\supseteq$?

We will see that the answer to that question depends on the analysis we are doing (basically whether we are interested in safe over-approximations or under-approximation). In the case of the reaching definitions ("…may reach …"), a solution *larger* than a safe one is safe, as well. As a consequence, the "left-hand sides" of the previous equations need to be $\supseteq$-larger than the "right-hand sides".

A final remark: the fact that we move from equations to $\supseteq$ allow allows for a simple *rearrangement* of the constraints. Basically, instead of saying equationally $s_1 = s_2 \cup s_3$, reformulated to $s_1 \supseteq s_2 \cup s_3$, the inequation is split into

$$\begin{aligned} s_1 &\supseteq s_2 \\ s_1 &\supseteq s_3 \end{aligned}$$

Factorial program: intra-block constraints

elementary block: $[y := x]^0$

elementary block: $[y > 1]^2$

all equations with RD_{exit} as left-hand side

$$\begin{aligned}
&\mathsf{RD}_{exit}(0) \supseteq \mathsf{RD}_{entry}(0) \setminus \{(y,l) \mid l \in \mathbf{Lab}\}\\
&\mathsf{RD}_{exit}(0) \supseteq \{(y,0)\}\\
&\mathsf{RD}_{exit}(1) \supseteq \mathsf{RD}_{entry}(1) \setminus \{(z,l) \mid l \in \mathbf{Lab}\}\\
&\mathsf{RD}_{exit}(1) \supseteq \{(z,1)\}\\
&\mathsf{RD}_{exit}(2) \supseteq \mathsf{RD}_{entry}(2)\\
&\mathsf{RD}_{exit}(3) \supseteq \mathsf{RD}_{entry}(3) \setminus \{(z,l) \mid l \in \mathbf{Lab}\}\\
&\mathsf{RD}_{exit}(3) \supseteq \{(z,3)\}\\
&\mathsf{RD}_{exit}(4) \supseteq \mathsf{RD}_{entry}(4) \setminus \{(y,l) \mid l \in \mathbf{Lab}\}\\
&\mathsf{RD}_{exit}(4) \supseteq \{(y,4)\}\\
&\mathsf{RD}_{exit}(5) \supseteq \mathsf{RD}_{entry}(5) \setminus \{(y,l) \mid l \in \mathbf{Lab}\}\\
&\mathsf{RD}_{exit}(5) \supseteq \{(y,5)\}
\end{aligned}$$

Factorial program: inter-block constraints

cf. slide 12 ff.: inter-block equations:

$$\begin{aligned}
\mathsf{RD}_{entry}(1) &= \mathsf{RD}_{exit}(0)\\
\mathsf{RD}_{entry}(2) &= \mathsf{RD}_{exit}(1) \cup \mathsf{RD}_{exit}(4)\\
\mathsf{RD}_{entry}(3) &= \mathsf{RD}_{exit}(2)\\
\mathsf{RD}_{entry}(4) &= \mathsf{RD}_{exit}(3)\\
\mathsf{RD}_{entry}(5) &= \mathsf{RD}_{exit}(2)\\
\\
\mathsf{RD}_{entry}(0) &= \{(x,?),(y,?),(z,?)\}
\end{aligned}$$

splitting of composed right-hand sides + using $\supseteq$ instead of $=$:

$$\begin{array}{lcl}
\mathsf{RD}_{entry}(1) & \supseteq & \mathsf{RD}_{exit}(0) \\
\mathsf{RD}_{entry}(2) & \supseteq & \mathsf{RD}_{exit}(1) \\
\mathsf{RD}_{entry}(2) & \supseteq & \mathsf{RD}_{exit}(4) \\
\mathsf{RD}_{entry}(3) & \supseteq & \mathsf{RD}_{exit}(2) \\
\mathsf{RD}_{entry}(4) & \supseteq & \mathsf{RD}_{exit}(3) \\
\mathsf{RD}_{entry}(5) & \supseteq & \mathsf{RD}_{exit}(2) \\
 & & \\
\mathsf{RD}_{entry}(1) & \supseteq & \{(x,?),(y,?),(z,?)\}
\end{array}$$

Least solution revisited

instead of $F(\vec{\mathsf{RD}}) = \vec{\mathsf{RD}}$

- clear: solution to the equation system $\Rightarrow$ solution to the constraint system
- important: **least** solutions *coincides*!

Pre-fixpoint

$$F(\vec{\mathsf{RD}}) \sqsubseteq \vec{\mathsf{RD}} \tag{1.11}$$

1.3 Constraint-based analysis

1.3.1 Control-flow analysis

Control-flow analysis

Goal CFA which elem. blocks lead to which other elem. blocks

- for while-language: immediate (labelled elem. blocks, resp., graph)
- complex for: more *advanced* features, *higher-order* languages, oo languages ...
- here: prototypical higher-order functional language **λ-calculus**
- formulated as **constraint-based analysis**

Introduction

This section is called *constraint-based analysis*, but it's also mostly about functional, higher-order languages (and complications that entails for static analysis).

In the section about data-flow analysis, the analysis was building upon *control-flow graphs*. Given the control-flow graph of a program, the problem was turned into a set of data flow equations or data flow inequations (there called constraints), which in turn then have to be solved. It was stressed *there* (in that simple setting) that the equational approach and the approach based on subset constraints give the same result (in that the best solution coincides in both representations). The analysis therefore proceeds in two clearly separated stages: first the control-flow graph is determined, and, based on that, the data flow is analyzed. We did not cover really the calculation of the control-flow graph in first stage as it's rather simple.

For languages with higher-order functions, the dividing line between "data" and "'control" is blurred: a function on the one hand is "code", but on the other hand is also "data" in that functions can be passed as arguments to other functions and returned as values from function. As a consequence, one would expect static program analysis gets more involved for higher-order functions. That's even true taking into account that the setting for data flow analysis in the introduction so far was really simple in that we did no even consider (non-higher-order) functions there. Adding those would complicate things —later in the lecture we will cover that to some extent— but still the 2 stages of determining the control-flow graph first, and afterwards doing data flow constraints are clearly discernible.

Here, determining the control flow is a problem in itself, unlike in the simpler setting before, where determining the CFG was so simple that would not even bother calling it control-flow analysis (even if it is). Here, we are facing a non-trivial *control flow analysis* problem and we use a constraint-based approach (not an equational) to tackle it.

Another way of seeing it is as follows: the control-flow problem here (which is inseparable from data-flow aspects as functions are higher-order), will not even result in a control-flow graph. For a non-higher order language, each function body has its own CFG, all clearly separated, but such a depiction makes no sense any longer. On the other hand: in the data flow section, we slightly touched upon the close connection between control-flow graph and the constraints (equational or otherwise). We did not explore it to the end, but one can view each edge of a control-flow graph as constraints connecting (in a forward analysis) the solutions of a target node dependent (via a constraint) on the solution on the source nodes and the nodes (= labels) play the role of *variables* in the constraint system. The connection is so close that a constraint systems in the form we covered in the data-flow section *is nothing else* than some representation of the control-flow graph (together with constraints connecting the nodes). As illustration, an inequation

$$x_1 \supseteq f(x_2, x_3)$$

represents the dependence of (the solution for) x_1 on the solutions of x_2 and x_3 via some f, thereby representing two incoming edges in x_1, starting from x_2 resp. x_3. In Section 1.2, this representation was not made explicit, and we did not called the variables x_i but RD_i (and split between entries and exits), but in general the connection is between variables of the constraint system and the nodes of the control flow graph.

Now, in the more complex situation of higher-order functions, we operate directly with constraints; since they are now more complicated, they can no longer be visualized as expressing relations on values in some graph.

Simple example

```
let f  = fn x => x 1;
    g  = fn y => y + 2;
    h  = fn z => z + 3;
```

```
in  (f  g) + (f  h)
```

- *higher-order* function f
- for simplicity: untyped
- local definitions via let-in
- interesting *above:* x 1

Goal (more specifically) For each function application, **which function** may be applied.

Local definitions with let The above example uses `let` for local definition. That is slighty different than assignments. For once, it's "single assignment", secondly it has a local scope associate with it. In later chapters, the functional language may support that construct, for the introduction we ignore it.

Labelling

- more complex language ⇒ more *complex* **labelling**
- "elem. blocks" can be *nested*
- *all* syntactic constructs (expressions) are labelled
- consider:

Unlabelled abstract syntax

$$(\mathtt{fn}\, x \Rightarrow x)\ (\mathtt{fn}\, y \Rightarrow y)$$

Full labelling

$$[\ [\mathtt{fn}\, x \Rightarrow [x]^1]^2\ [\mathtt{fn}\, y \Rightarrow [y]^3]^4\ ]^5$$

- functional language: side-effect free

⇒ *no* need to distinguish *entry* and *exit* of labelled blocks.

Data of the analysis

Pairs $(\hat{C}, \hat{\rho})$ of mappings:

abstract cache: $\hat{C}(l)$: set of values/function abstractions, the subexpression labelled l may evaluate to

abstract env.: $\hat{\rho}$: values, x may be bound to

The constraint system

- ignoring "`let`" here: *three* syntactic constructs ⇒ *three* kinds of constraints
- relating $\hat{C}$, $\hat{\rho}$, and the program in form of subset constraints (subsets, order-relation)

3 syntactic classes

- function abstraction: $[\mathtt{fn}\, x \Rightarrow x]^l$
- variables: $[x]^l$
- application: $[f\ g]^l$

Constraint system for the small example

Labelled example

$$[\ [\mathtt{fn}\, x \Rightarrow [x]^1]^2\ [\mathtt{fn}\, y \Rightarrow [y]^3]^4\ ]^5$$

- application: connecting function entry and (body) exit with the argument but:
- also $[\mathtt{fn}\, y \Rightarrow [y]^3]^4$ is a candidate at 2! (according to $\hat{C}(2)$)
- function abstractions
- variables (occurrences of use)

$$\begin{array}{rcl}
\{\mathtt{fn}\, x \Rightarrow [x]^1\} & \subseteq & \hat{C}(2) \\
\{\mathtt{fn}\, y \Rightarrow [y]^3\} & \subseteq & \hat{C}(4) \\
\hat{\rho}(x) & \subseteq & \hat{C}(1) \\
\hat{\rho}(y) & \subseteq & \hat{C}(3) \\
\{\mathtt{fn}\, x \Rightarrow [x]^1\} \subseteq \hat{C}(2) \quad \Rightarrow \quad \hat{C}(4) & \subseteq & \hat{\rho}(x) \\
\{\mathtt{fn}\, x \Rightarrow [x]^1\} \subseteq \hat{C}(2) \quad \Rightarrow \quad \hat{C}(1) & \subseteq & \hat{C}(5) \\
\{\mathtt{fn}\, y \Rightarrow [y]^3\} \subseteq \hat{C}(2) \quad \Rightarrow \quad \hat{\mathbf{C}}(\mathbf{4}) & \subseteq & \hat{\rho}(y) \\
\{\mathtt{fn}\, y \Rightarrow [y]^3\} \subseteq \hat{C}(2) \quad \Rightarrow \quad \hat{C}(3) & \subseteq & \hat{\mathbf{C}}(\mathbf{5})
\end{array}$$

Explanation of the constraint system The example is rather small, but contains all constructs of the language (ignoring `let`), therefore it illustrates how a label program gives rise to a constraint system, in this case for control-flow analysis. It's still a bit of an illustration only, as we don't give the general construction, we just show how it works in the given example.

The slides present the construction in *2 stages* (for didactic reasons). The first stage gives "standard", *unconditional* constraints. The resulting system is fine in the sense that all solutions to it are *safe approximations* of the problem. The second stage is an improvement in that it gives more precise solutions. It is formulated with *conditional* constraints.

For programmes in the given calculus, there are *3 different classes* of constraints, since there are three classes of syntactial constructs. The classification is done according to the top-level construct at the label being considered. Remember that abstract syntax represents trees, and now the labels indentify *subtrees*. The three cases are the following:

1. One for abstractions, one for
2. variables, and one for
3. applications (the most complex).

They are shown in this order via overlays.

1. Abstractions
We have two abstractions, at label 2 and 4. The constraints in this category are relevant for the $\hat{C}$, only, as there are no variables "involved" (only the bound variables inside the function definition). Therefore, the abstract environment is not mentioned here. The inequations are easy. They just state that the expression at the given label (which labels, as said, an abstraction), *at least* evaluates to that abstraction. That's pretty obvious.
2. Variables
Variables are a bit more tricky, especially because the corresponding constraints *relates* the $\hat{C}$ and the abstract environment. Basically it relates directly the $\hat{C}$ for the label with the $\hat{\rho}$ variable carrying that label. The only perhaps tricky *question* is: how do they relate, in which order is the $\subseteq$. In the example, in the smallest solution, it's $=$, anyway, in the end.
It's the way that one "visualizes" the flow. Here we have a variable, and the information flows *from* the variable *to* the label. We do not write to the variable, but *read* from it and get the value from the variable into the program (at the given label). Since we have a forward-may-analysis and go for the smallest value, the *"post-state" (at l)* must be equal or **larger** than the *"pre-state" (at x).*
3. Applications
We have only one application (labelled 5). So we have to think of the flow. One is that the argument "flows into" the variable which constitues the formal argument of the function. The same reasoning (intuitive direction of the flow of information) determines the direction of the inequation. The *second* flow is the *output:* what comes *out of the body of the function function comes /out of the application.* That relates the $\hat{C}$ of the corresponding labels. This explains the first two inequations (which are not yet complete, there will be 2 steps to capture the flow (more precisely)). The first step deals with the fact that we don't really know which function a variable represents (higher-order). The second one is an optimization, introducing *conditional* constraints.
However, that is **not all** for applications. So far we have just looked "intuitively" at the code, but we need to relate to the analysis/an algorithm. This means, we should not just look at what function there "is" according to our intuition at label 2 (the place for the function in the application) but formally we need to think, what functions location 2 evaluates *according to the analysis.* In other words, we need to consult $\hat{C}(2)$. We have only a **lower bound** for $\hat{C}(2)$ (some slides earlier), therefore, in principle *all* possible abstractions (there are 2) are candidates to be at the applicator position in the application. I.e., the one *missing* is $[\texttt{fn}\, y \Rightarrow [\mathbf{y}]^{\mathbf{3}}]^4$. This adds *two more* inequations, again one for input and the other for output, this time for the second abstraction.
What we have so far *would work* in the sense of leading to a sound analysis. As mentioned before, we can do better, however, in the category for applications. Namely that we formulate the inequations *conditionally* making the information about which function for $l = 2$ evaluates to a *more precise* (smaller) solution. In the example, we will see that the analysis finds out that $\hat{C}(2)$ indeed only contains *one* function, the obvious one. This additional precision gives indeed a smaller solution.
The refinement can be explained as follows: it concerns the category for applications (location 5). Each "instance" of an application leads to *two* constraints "in" and

"out" (for parameter passing and returning the data). The question only is (as discussed): *which* function definition is actually meant where the data flows "in" and "out". What we know is that the function, whichever it is, is placed at location 2. Therefore the condition in the conditional refinement of the constraints expresses the following

> "if such-and-such function occurs at 2, then consider the "in" constraint (and "out" constraint) for that function"

The least (= best) solution

$$\begin{array}{rcl}
\hat{C}(1) & = & \{\mathtt{fn}\, y \Rightarrow [y]^3\} \\
\hat{C}(2) & = & \{\mathtt{fn}\, x \Rightarrow [x]^1\} \\
\hat{C}(3) & = & \emptyset \\
\hat{C}(4) & = & \{\mathtt{fn}\, y \Rightarrow [y]^3\} \\
\hat{C}(5) & = & \{\mathtt{fn}\, y \Rightarrow [y]^3\} \\
\hline
\hat{\rho}(x) & = & \{\mathtt{fn}\, y \Rightarrow [y]^3\} \\
\hat{\rho}(y) & = & \emptyset
\end{array}$$

One interesting bit here in the solution is: $\hat{\rho}(y) = \emptyset$: that means, the variable y never evaluated, i.e., the function is not applied at all.

1.4 Type and effect systems

1.4.1 Introduction

Standard type systems and others

In this section, we are mainly dealing with "non-standard" type systems, but let's have a look at *standard* type systems first.

Types and type systems are a well-established and central part of static analysis. Before continuing further: As far as the lecture is concerned, we are dealing mainly with *static* type systems. The typing part of the semantic analysis phase is also kind of familiar, as they are often visible in the programming language and the programmer has to learn to deal with them. For languages being typed at run-time (for instance some scripting languages), types may be less visible, even if they are still there (but dynamically typed languages are mostly out of scope for that lecture anyway). In any case, even novice programmers are aware that somewhere under the hood, the compiler checks whether the given program adheres to the language's typing discipline; that's the task of the type checker.

The "standard" role of types is to describe "data values" and type checking assures that meaningful use is made of the date. At the lowest level, it serves the compiler, for instance, also to know the "size" of data so as to allocate adequate amount of memory for storing and accessing it. The lecture here covers types mostly on a higher abstraction level, and types are seen as *specifications* of allowed uses of data. For instance, the type system may allow to use the logical operations `and` and `or` on boolean typed values, but not addition. On that level, type and type systems are very rudimentary and potentially very rigid (but still important). Modern (standard) type system are far more complex, basically due to the wish to add flexibility (different forms of so-called polymorphism) but still maintaining efficient, scalable, decidable static type checking.

Type theory —the study of type systems, their expressiveness, efficiency etc.— often deals with functional languages, one reason being that functional languages feature one particular expressive form of data, namely functions. There are other reasons why type systems for functional languages have been widely studied, but that's perhaps outside the scope of the lecture and we leave it at that.

At any rate, whether it's the more down to earth "size-of-memory" basic types for code generation, or advanced polymorphic type systems for some λ-calculus or other, standard type systems are always concerned with specifying sets of data values (with the purpose of regulating their usage).

So far for *standard* type systems, what about *non-standard* ones? The latter term here is used for all aspects *different* from describing *values* in a programming language. A value of a program is "what comes out at the end", that's why executing a program is sometimes called *evaluation*... Another name we will encounter is *reduction*, the intuition being that the program is "reduced" to its final *value*. Anyhow, non-standard type systems, in contrast, specify aspects that are not related to the final value. For instance, data flow or control flow information is certainly something that typically is not covered by type systems. *Effects*, in particular, cover aspects that happen "during" the execution. It's not a coincidence that this is a very broad definition, as basically everything that needs semantic analysis at compile time and which refers to "what happens when the program runs" (as opposed to "what's the final value") might be described by an effect system. Often, the description is combined with a type system, in which case it's called a *type and effect system*. The lecture will probably cover different effects (exceptions, communication, ...).

For Haskell progammers, the separation between the "pure", functional part and all "the other aspects" should be familiar. Haskell goes to great lengths to separate both aspects, encapsulating all impure effects in so-called *monads*. So one might call standard types all that captures the pure (effect-free) core of the language, and effects correspond to those covered by monads. The connection between effects and monads can even be made formal (giving a monadic interpretation to effects), but that's beyond this lecture.

Effects: Intro

- type system: "classical" static analysis:

$$t : T$$

- *judgment*: "term or program phrase has type T"
- in general: *context-sensitive* judgments (remember Chomsky ...)

Judgement :

$$\Gamma \vdash t : \tau$$

- Γ: *assumption* or *context*
- here: *"non-standard"* type systems: effects and annotations
- natural setting: typed languages, here: *trivial!* setting (while-language)

Type system is a classical *context sensitive* analysis, following parsing, which is the classical context-free analysis. It's thus not a coincidence, that the component Γ in the judgements is called "context". In implementations, it directly corresponds to the so-called *symbol table*. That's a data structure, often realized as hash table or similar, used to keep relevant information about syntactic entities during the semantical phase (and also for code generation), in particular about variables or other "symbols". One prominent piece of information is about the types, and that's what Γ contains. When coming to effect systems and some non-standard types, also other kind of information may be stored in the types, and therefore in Γ, as well.

For the participants for the compiler construction course (INF5110): information attached to syntactic elements were there also called *attributes*. In that parlance, a type is an attribute of a variable (or an expression etc.). Note in passing, that also the labelling of expressions can well be seen as attribuation.

Context-sensitive information assosciated with syntactic entities was called attributes basically because in the static analysis part, the lecture was working with so-called *attribute grammars*. In this lecture we don't bother with that general formalism. The type and effect systems will be represented in the form of *derivation systems*, but in priciple that can be viewed as some special notation for specific attribute grammars, as well.

"Trival" type system

- setting: while-language
- each statement maps: state to states
- Σ: type of *states*

judgement

$$\vdash S : \Sigma \to \Sigma \tag{1.12}$$

- specified as a *derivation* system
- note: *partial* correctness assertion

The "type system" here is *trivial* in a technical sense. Type systems are used to *distinguish* well-typed programs from *ill-typed* ones, and reject the latter. This is generally not a context-free task, and therefore cannot be done by the parser alone, even though, for simple languages, type checking can be done *while* parsing. Here, the "type system" (shown below) accepts *all* programs, all syntactically correct programs are already well-typed (remember that we are dealing with abstract syntax, i.e., trees, which represent syntactically correct programs). Since all programs are well-typed, there is actually no need for a type system at all. It's only used here to illustrate how to extend a (in this case trivial) type system with extra information, leading to *annotated* type system and *effect type system*, which then yield useful information. Note in passing that the above judgment from equation (1.12) has no context Γ, reflecting the fact that it's technically context-free and not context-sensitive, and thereby trivial.

The typing judgment, as is the case in general for standard type system, is intended to be a *partial correctness* assertion. The trivial type system here is a bad illustation for that fact (being so trivial), but in general, the meaning of a judgment $e : \ \tau$ is:

> *if* the statement, expression or progam e terminates, then the resulting value conforms to the type τ.

Type systems typically don't attempt have an opinion about *termination*, they only guarantee that the data at the end is ok, *provided* the program terminates thereby yielding said value. This restricted form of assertion is known as *partial correctness* (as opposed to total correctness). Another word used for such specifications is, that typing is conventionally concerned with *safety* properties (as opposed to *liveness* properties).

This is not intended to say, that it's impossible to devise type systems that try to capture "total correctness" in that they would guarantee termination or would warn against possible non-termination. In general, that would require "non-standard" augmented information.

As a final side remark: there is a connection between termination and standard type system in the following way: for typed λ-calculi (without recursion), well-typed programs guarantee termination, which requires non-trivial techniques for proving that, and furthermore makes pure, typed λ-calculi (without recursion) no real programming languages insofar as they are not Turing-complete. The λ-calculus we will encounter later does have a recursion operator for that reason.

"Trival" type system: rules

$$\vdash [x := a]^l : \Sigma \to \Sigma \qquad \textsc{Ass}$$

$$[\mathsf{skip}]^l : \Sigma \to \Sigma \qquad \textsc{Skip}$$

$$\frac{\vdash S_1 : \Sigma \to \Sigma \qquad S_2 : \Sigma \to \Sigma}{\vdash S_1; S_2 : \Sigma \to \Sigma}\ \textsc{Seq}$$

$$\frac{\vdash S : \Sigma \to \Sigma}{\vdash \texttt{while}[b]^l\,\texttt{do}\,S : \Sigma \to \Sigma}\ \textsc{While}$$

$$\frac{\vdash S_1 : \Sigma \to \Sigma \qquad \vdash S_2 : \Sigma \to \Sigma}{\vdash \texttt{if}[b]^l\,\texttt{then}\,S_1\,\texttt{else}\,S_2 : \Sigma \to \Sigma}\ \textsc{Cond}$$

As mentioned, the “type system” does not do anything useful. It shows, however, the general *style* of writing down type systems, namely in the form of derivation rules. In the current version, there are five rules, one for each construct. Each rule has a set of *premises* and one *conclusion*. In case, there are *no* premises, a rule is also called *axiom*. That’s the case for Ass and Seq, which are dealing with the *basic* constructs of the language. The compound statement are treated by the rules with a non-empty set of premises, where the premises deal with the sub-constructs.

The rules can be viewed a logical “implications”, reading them from top to bottom: If S_1 is well-typed and S_2 is well-typed, then so is the sequential $S_1; S_2$ composition (in rule Seq). A program S is well-typed, if there is a derivation tree using the givem rules such that $\vdash S : \Sigma \to \Sigma$ is *derivable*, i.e., is the root of the tree (derivation trees have their roots at the bottom and their leaves, corresponding to axioms, at the top. . .)

One can view the rules also as the specification of a recursive procedure: in order to establish that $S_1; S_2$ is well-typed, one has to recursively check S_1 and S_2 for well-typedness. It’s of course just a different angle on the same thing. This “reading” of the rules is sometimes called “goal-directed”: in order to establish the conclusion, establish the premises first.

Seen in this goal-directed manner, the rules directly can be seen as a recursive procedure (corresponding to a tree-traversal of the abstract syntax tree). It’s a very straight-forward divide and conquer strategy. Note in this context: the language has 5 syntactic constructs (2 basic ones and 3 compound ones) and the derivation system has 5 rules (including 2 axioms), exactly one for each contruct.

That entails that the top-level constuct of a subprogram *determines* which rule to apply recursively (in the goal-directed reading of the rules). Furthermore, the premises always deals with proper *subterms* compared to the term or statement in the conclusion, which in particular guarantees termination. Type systems, or derivation systems in general with these “one-rule-per-construct” and “in-the-premises-subterms-only” properties are called *syntax-directed*. In the terminology of attribute grammars: the types here correspond to *inherited* attributes (in their simplest form). But keep in mind that the “type system” here is trival to the point of being meaningless. More realistic type systems do *not* correspond to inherited attributes, they are more complex.

Unfortunately, not all type systems are given in a syntax-directed manner. That means, not all type system specifications can be immediately understood as an algorithm. Sometimes, that is for fundamental reasons: the rules describe a type system so complex that the question whether $\Gamma \vdash e : \tau$ is derivable or not is *undecidable*. It won’t be much the case in this lecture. A more common reason is: the rules of the type system are not a priori intended as an *algorithm*, they are rather intended as *specification* of the typing discipline and showing an algorithm would obscure that specification. A non-syntax directed specification for instance will be used when dealing with *subtyping* or other forms of polymorphism, resp. *subeffecting*. We will also touch upon the question: given a non-syntax-directed type system as specification, how can one turn it into a syntax-directed, thus algorithmic version. The algorithm, of course, corresponds to the *type checker*.

Types, effects, and annotations

$$\vdash S : \Sigma_1 \to \Sigma_2 \qquad (1.13) \qquad\qquad \vdash S : \Sigma \xrightarrow{\varphi} \Sigma \qquad (1.14)$$

type and effect system (TES)

- *effect* system + *annotated* type system
- borderline fuzzy
- **annotated type system**
  - Σ_i: property of state ("$\Sigma_i \subseteq \Sigma$")
  - "abstract" properties: invariants, a variable is positive, etc.
- **effect system**
  - "statement S maps state to state, with (potential ...) effect φ"
  - *effect* φ: e.g.: errors, exceptions, file/resource access, ...

1.4.2 Annotated type systems

Annotated type systems

- example again: *reaching definitions* for while-language
- 2 flavors
  1. annotated base types: $S : \mathsf{RD}_1 \to \mathsf{RD}_2$
  2. annotated type constructors: $S : \Sigma \xrightarrow[\mathsf{RD}]{X} \Sigma$

Here we see that the border line between annotated type systems and effect system is fuzzy. The first sub-flavor corresponds to the intuition we have used so far: the states are restricted. The second one, if we think of it as functional type, can be seen as if the "functional" type is annotated. However, the annotation is like an effect (as we will see later).

RD with annotated base types

judgement

$$\vdash S : \mathsf{RD}_1 \to \mathsf{RD}_2 \qquad (1.15)$$

- $\mathsf{RD} \subseteq 2^{\mathbf{Var} \times \mathbf{Lab}}$
- auxiliary functions
  - note: every S has one "initial" elementary block, potentially more than one "at the end"
  - *init*(S): the (unique) label at the entry of S
  - *final*(S): the set of labels at the exits of S

"meaning" of judgment $\vdash S : \mathsf{RD}_1 \to \mathsf{RD}_2$ "RD_1 is the set of var/label reaching the entry of S and RD_2 the corresponding set at the exit(s) of S":

$$\begin{array}{rcl}\mathsf{RD}_1 & = & \mathsf{RD}_{entry}(init(S)) \\ \mathsf{RD}_2 & = & \bigcup\{\mathsf{RD}_{exit}(l) \mid l \in final(S)\}\end{array}$$

Concerning the "meaning" of the judgment: the formulation is not 100% correct, it's too strict as we will see. The problem is the claim that RD is *the* set of . . . . . . As in the data flow section, there is not just one single *safe* solution, but many. There is (as before) exactly one *minimal*, i.e., best one. However, the effect system is "lax" in that it specifies all safe ones. That is completely in analogy to the previous constraint system approaches.

One may compare the sets RD to the analysis data used in the "original" reaching definitions analysis (in the equational or constraint based approach, it does not matter which). The "functional" type here expresses the pre- and post-states. For elementary blocks, that corresponds to the entry and the exit point of the node or label. As one rule (coming next) treats one block/statement (elementary or not) at a time, it's not the *12-tuple*, of course, (taking the *concrete* factorial example) but just 2 generic slots of it: the pre- and the post-state.

Concerning the auxiliary functions (initial and final): They calculate *implicitly* the *control flow graph*.

Rules

$$\vdash [x := a]^{l'} : \mathsf{RD} \to \mathsf{RD} \setminus \{(x,l) \mid l \in \mathbf{Lab}\} \cup \{(x,l')\} \qquad \textsc{Ass}$$

$$\vdash [\mathsf{skip}]^{l} : \mathsf{RD} \to \mathsf{RD} \qquad \textsc{Skip}$$

$$\frac{\vdash S_1 : \mathsf{RD}_1 \to \mathsf{RD}_2 \qquad \vdash S_2 : \mathsf{RD}_2 \to \mathsf{RD}_3}{\vdash S_1; S_2 : \mathsf{RD}_1 \to \mathsf{RD}_3}\ \textsc{Seq}$$

$$\frac{\vdash S_1 : \mathsf{RD}_1 \to \mathsf{RD}_2 \qquad \vdash S_2 : \mathsf{RD}_1 \to \mathsf{RD}_2}{\vdash \mathtt{if}[b]^l\,\mathtt{then}\,S_1\,\mathtt{else}\,S_2 : \mathsf{RD}_1 \to \mathsf{RD}_2}\ \textsc{If}$$

$$\frac{\vdash S : \mathsf{RD} \to \mathsf{RD}}{\vdash \mathtt{while}[b]^l\,\mathtt{do}\,S : \mathsf{RD} \to \mathsf{RD}}\ \textsc{While}$$

$$\frac{\vdash S : \mathsf{RD}_1' \to \mathsf{RD}_2' \qquad \mathsf{RD}_1 \subseteq \mathsf{RD}_1' \qquad \mathsf{RD}_2' \subseteq \mathsf{RD}_2}{\vdash S : \mathsf{RD}_1 \to \mathsf{RD}_2}\ \textsc{Sub}$$

The rules may be compared with the constraint-based formulation of the reaching definitions early, which was based on the control-flow graph. The *intra-block* equations are covered by the axioms here, and the *inter-block* equations are the compositional part, the rules.

Worth mentioning is also the fact that in the annotated type system now, we have *6 rules* (for 5 syntactic constructs). So, there goes the syntax-directedness. . . . As a consequence, the type system does not (directly) describe an algorithm.

The culprit is rule SUB (for *subsumption*). We will encounter subsumption many times (for subeffecting, and also subtyping), and it's one classical reason why type systems are not syntax directed. It breaks it in two ways: first, for each construct, there are now *two rules* to choose from (if one thinks in a goal-directed manner), as subsumption is *always* possible. as well. Secondly, the core judgment in the premise does not assert well-typedness for a proper *sub-term* of S in the conclusion. So, going from conclusion to the premise, S does not get "smaller". Suddenly, termination of type checking may become a non-trivial issue.

Meaning of annotated judgments

"Meaning" of judgment $S : \mathsf{RD}_1 \to \mathsf{RD}_2$: "$\mathsf{RD}_1$ is *the* set of var/label reaching the entry of S and RD_2 the corresponding set at the exit(s) of S":

$$\begin{array}{rcl} \mathsf{RD}_1 & = & \mathsf{RD}_{\mathit{entry}}(\mathit{init}(S)) \\ \mathsf{RD}_2 & = & \bigcup\{\mathsf{RD}_{\mathit{exit}} l \mid l \in \mathit{final}(S)\} \end{array}$$

- Be careful:

$$\texttt{if}\,[b]^l\,\texttt{then}\,S_1\,\texttt{else}\,S_2$$

- more concretely

$$\texttt{if}\,[b]^l\,\texttt{then}\;[x := y]^{l_1}\;\texttt{else}\;[y := x]^{l_2}$$

Derivation

$$\dfrac{[y := x]^1 : \mathsf{RD}_0 \to \{?_x, 1, ?_z\} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{c}[z := \_]^4 : \mathsf{RD}_{\mathit{body}} \to \{?_x, 1, 5, 4, \cancel{2}\} \\ [y := \_]^5 : \{?_x, 1, 5, 4\} \to \{?_x, 5, 4\}\end{array}}{f_{\mathit{body}} : \mathsf{RD}_{\mathit{body}} \to \{?_x, 5, 4\}}}{f_{\mathit{body}} : \mathsf{RD}_{\mathit{body}} \to \mathsf{RD}_{\mathit{body}}}\;\textsc{Sub}}{f_{\mathit{while}} : \mathsf{RD}_{\mathit{body}} \to \mathsf{RD}_{\mathit{body}}}}{f_{\mathit{while}} : \{?_x, 1, 2\} \to \mathsf{RD}_{\mathit{body}}}\;\textsc{Sub} \qquad [y := 0]^6 : \mathsf{RD}_{\mathit{body}} \to \mathsf{RD}_{\mathit{final}}}{f_3 : \{?_x, 1, 2\} \to \mathsf{RD}_{\mathit{final}}}}{f_2 : \{?_x, 1, ?_z\} \to \mathsf{RD}_{\mathit{final}}}}{f : \mathsf{RD}_0 \to \mathsf{RD}_{\mathit{final}}}$$

$$\mathsf{RD}_0 = \{?_x, ?_y, ?_z\} \quad \mathsf{RD}_{\mathit{final}} = \{?_x, 6, 2, 4\}$$

- abbreviate $f_3 = \texttt{while}\ldots; [y := 0]^6$
- *loop invariant*

$$\mathsf{RD}_{body} = \{?_x, 1, 5, 2, 4\}$$

1.4.3 Annotated type constructors

Annotated type constructors

- alternative approach of annotated type systems
- arrow constructor itself *annotated*
- annotion of $\rightarrow$: flavor of effect system
- judgment

$$S : \Sigma \xrightarrow[\mathsf{RD}]{X} \Sigma$$

- annotation with RD (corresponding to the post-condition from above) alone is *not enough*
- also needed: the *variables "being" changed*

Intended meaning "S maps states to states, where RD is the set of reaching definitions, S may produce and X the set of var's S must (= unavoidably) assign.

In the previous formulation (with annotated base types), each judgment mentioned two versions of the RD information, the one before and the one after. Now, there is only one RD information, which is interpreted as the reaching definitions the statement of the rule may produce. That means, the generated RD is considered very much like an *effect* of the statement.

RD is indeed the information we are interested in for this analysis. However, to make the system technically work and fit together: It's not enough to keep track of "reaching definitions" which are *generated* per construct. If that were the only information, we would never "remove" any tuples, just add them. That can in particularly seen in the treatment of sequential composition $S_1; S_2$, see rule Seq below.

In order to capture that, the sets X of assigned variables is kept track of, as well. In the monotone frameworks later, the removal of flow information is also called "killing" and, in many cases, the data flow equations and transfer functions can be decscribed as a combination of "generating" and "killing" data flow. It could also be noted: the RD here is "may" information, wheras the X is "must" information.

Rules

$$[x := a]^l : \Sigma \xrightarrow[\{(x,l)\}]{\{x\}} \Sigma \quad \textsc{Ass} \qquad [\mathsf{skip}]^l : \Sigma \xrightarrow[\emptyset]{\emptyset} \Sigma \quad \textsc{Skip}$$

$$\frac{S_1 : \Sigma \xrightarrow[\mathsf{RD}_1]{X_1} \Sigma \qquad S_2 : \Sigma \xrightarrow[\mathsf{RD}_2]{X_2} \Sigma}{S_1; S_2 : \Sigma \xrightarrow[\mathsf{RD}_1 \setminus X_2 \cup \mathsf{RD}_2]{X_1 \cup X_2} \Sigma} \textsc{Seq}$$

$$\frac{S_1 : \Sigma \xrightarrow[\mathsf{RD}]{X} \Sigma \qquad S_2 : \Sigma \xrightarrow[\mathsf{RD}]{X} \Sigma}{\mathtt{if}[b]^l\,\mathtt{then}\,S_1\,\mathtt{else}\,S_2 : \Sigma \xrightarrow[\mathsf{RD}]{X} \Sigma} \textsc{If}$$

$$\frac{S : \Sigma \xrightarrow[\mathsf{RD}]{X} \Sigma}{\mathtt{while}[b]^l\,\mathtt{do}\,S : \Sigma \xrightarrow[\mathsf{RD}]{\emptyset} \Sigma} \textsc{While}$$

$$\frac{S : \Sigma \xrightarrow[\mathsf{RD}']{X'} \Sigma \qquad X \subseteq X' \qquad \mathsf{RD}' \subseteq \mathsf{RD}}{S : \Sigma \xrightarrow[\mathsf{RD}]{X} \Sigma} \textsc{Sub}$$

In [3], the IF rule is formulated more complex:

$$\frac{S_1 : \Sigma \xrightarrow[\mathsf{RD}_1]{X_1} \Sigma \qquad S_2 : \Sigma \xrightarrow[\mathsf{RD}_2]{X_2} \Sigma}{\mathtt{if}[b]^l\,\mathtt{then}\,S_1\,\mathtt{else}\,S_2 : \Sigma \xrightarrow[\mathsf{RD}_1 \cup \mathsf{RD}_2]{X_1 \cap X_2} \Sigma} \textsc{If}$$

The formulations are, of course, equivalent. The one on the slides is a special case of the more complex one. For the reverse direction, one can use subsumption. In the presence of the more complex rule, one can remove subsumption!

Interesting is the WHILE-rule, especially the fact that the set above the arrow is $\emptyset$. That's because the while-loop may not be taken at all, so that corresponds to the *intersection* of X in the premise and $\emptyset$.

1.4.4 Effect systems

Effect systems

- this time: back to the *functional* language
- starting point: simple type system
- *judgment*:

$$\Gamma \vdash e : \tau$$

- Γ: *type environment* (or context), “mapping” from variable to types
- types: bool, int, and $\tau \to \tau$

The “language” here basically is known as the *simply typed λ-calculus.* It is a variant which is known as *Curry-style* formulation of the type system. That refers to the fact that abstractions are written as $\lambda x.e$ instead of $\lambda x{:}\tau.e$. In the latter case, it's know as “Church style”. In the less explicit Curry-style, one is facing a problem of so-called *type inference*, i.e., figuring out what the (omitted) type of x should be. That in itself is an important and interesting problem, which we might cover later, but in the introduction, we focus on the *effect* part, not the typing.

As far as the syntax is concerned: The π-subscript of the abstraction is *non-standard* as far as the typed λ-calculus and its typing is concerned. It's added for the purpose of the effect system later, only.

In comparison to the earlier “type-system” we used a starting point for the while language, now the system is non-trivial (even if completely standard).

Rules

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\ \textsc{Var}$$

$$\frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \ \mathtt{fn}_\pi x \Rightarrow e : \tau_1 \to \tau_2}\ \textsc{Abs}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}\ \textsc{App}$$

As mentioned, the flavor of the (simply typed) λ-calculus here is Curry-style. As before, an easy way to interpret the rules is as derivation rules, where the premises imply the conclusion.

It may be also instructive to look at them in a goal-directed manner, as if it were a recursive procedure: what is needed in order to derive a conclusion.

But even more instructive is to think what such a recursive interpretation is supposed to achieve. To be useful as conventional type checker, it's not meant as: *check* if the following is true

$$\Gamma \vdash e : \tau? \tag{1.16}$$

like confirming the user's guess whether or not it's true that e has type τ (given the assumptions Γ). Instead, more useful is the interpretation

$$\Gamma \vdash e : ? \tag{1.17}$$

i.e., an answer to the question: is e well-typed under the given assumption, and if so, what's its type? In other words, when interpreting the rules, one useful reading is to see Γ and e as *input* of a recursive procedure, and τ as the *output*.

With this interpretation in mind, especially rule ABS is interesting. Here, in the Curry-style formulation, the formal parameter x is mentioned in the conclusion *without type*. That means, that the "recursive call" corresponding to the premise *guesses* a type τ_1 for it. This guess may be right in that it leads to a successful type check of the premise and thus a sucessful type check and returning $\tau_1 \to \tau_2$ for the abstraction in the conclusion. Or it may also fail. The form of abstraction may determined that a recursive call according rule APP is in place, however, not information is given in the input Γ and $\lambda x.e$ to determine the subsequent recusive call (as τ_1 has to be guess). Note that there are *infinitely* many types to choose from. . . . In other words, the rules (in the interpretation given) do *not really qualify as algorithm*, and they would not qualify as syntax-directed (if we interpret the rules as describing a problem as in equation (1.17) and not as *confirming* the type as in equation (1.16).

Solving the "guessing problem" in a proper manner is known as *type inference* or *type reconstruction* and we might return to it later. Also note: if the abstraction would be of the form $\lambda x{:}\tau.e$, the guessing problem would go away as well. This form is also known a *Church-style typing* (as opposed to Curry-style).

Effects: Call tracking analysis

Call tracking analysis: Determine: for each subexpression: which function abstractions may be applied, i.e., called, **during** the subexpression's evaluation.

$\Rightarrow$ set of function names

annotate: function type with **latent effect**

$\Rightarrow$ *annotated* types: $\hat{\tau}$: base types as before, arrow types:

$$\hat{\tau}_1 \xrightarrow{\varphi} \hat{\tau}_2 \tag{1.18}$$

- functions from τ_1 to τ_2, where in the execution, functions from set φ are called.

Judgment

$$\hat{\Gamma} \vdash e : \hat{\tau} :: \varphi \tag{1.19}$$

It may be worthwhile to reflect what the connection is between the call-tracking analysis here and the constraint-based analysis we had before. Both analyses are not the same. In the constraint-based analysis from earlier, we wanted to know *at each point* to which functions it evaluates to answer the question for applications $f\ a$, which functions are actually called. Here, the perspective is different: we take an expression and think of it as being evaluated and ask, which functions are being called *during evaluation*. Both questions, however, are related. Remember that the analysis earlier was *not* formulated as a type system.

Later in the lecture, we may revisit the control-flow analysis in a type-based formulation. This will be easier to compare to the call-tracking analysis from here, basically because it's formulated similarly, namely as an type-based derivation system quite similar to the one here. This allows to see the differences and similarities more clearly. The earlier control-flow analysis was *not* formalized as type system, but with constraints. It's not, however, meant to mean that constraint based systems are fundamentally different from type and effect systems. After all they may tackle the same problems. Again it's more a stylistic question and type systems as the one here can also be seen as a particular way (using logical derivation rules as used for type system) to specify the constraint system.

Call tracking rules

$$\frac{\hat{\Gamma}(x) = \hat{\tau}}{\hat{\Gamma} \vdash x : \hat{\tau} :: \emptyset}\ \textsc{Var}$$

$$\frac{\Gamma, x{:}\hat{\tau}_1 \vdash e : \hat{\tau}_2 ::\ \varphi}{\Gamma \vdash \mathtt{fn}_\pi x \Rightarrow e : \hat{\tau}_1 \overset{\varphi\cup\{\pi\}}{\longrightarrow} \hat{\tau}_2 :: \emptyset}\ \textsc{Abs}$$

$$\frac{\hat{\Gamma} \vdash e_1 : \hat{\tau}_1 \overset{\varphi}{\to} \hat{\tau}_2 :: \varphi_1 \qquad \hat{\Gamma} \vdash e_2 : \hat{\tau}_1 :: \varphi_2}{\hat{\Gamma} \vdash e_1\ e_2 : \hat{\tau}_2 :: \varphi \cup \varphi_1 \cup \varphi_2}\ \textsc{App}$$

Call tracking: example

$$\frac{\dfrac{x{:}\mathsf{int} \overset{\{Y\}}{\to} \mathsf{int} \vdash x{:}\mathsf{int} \overset{\{Y\}}{\to} \mathsf{int} :: \emptyset}{\vdash (\mathtt{fn}_X x \Rightarrow x) : (\mathsf{int} \overset{\{Y\}}{\to} \mathsf{int}) \overset{\{X\}}{\to} (\mathsf{int} \overset{\{Y\}}{\to} \mathsf{int}) :: \emptyset} \qquad \vdash (\mathtt{fn}_Y y \Rightarrow y) : \mathsf{int} \overset{\{Y\}}{\to} \mathsf{int} :: \emptyset}{\vdash (\mathtt{fn}_X x \Rightarrow x)\ (\mathtt{fn}_Y y \Rightarrow y) : \mathsf{int} \overset{\{Y\}}{\to} \mathsf{int} :: \{X\}}$$

1.5 Algorithms

Introduction

This part is rather short. So far, we touched upon here and there on issues how to algorithmically treat the problems we encountered. In particular in the context of the type systems, some remarks tried to raise awareness under which circumstances the derivation rules could straightforwardly be interpreted as a recursive procedure, and when not (basically in all interesting cases). But we never really tackled the issue of obtaining an algorithm, especially not for the "flow problems" (data flow, effects etc.).

In particular: we saw how data flow problems such as reaching definitions can be described or specified as constraint systems (equational or otherwise). What we did *not* do so far is giving hints of how to actually *solve* such constraint system, i.e., how do do *constraint solving*.

The notion of "constraint system" is extremely broad and can capture all sorts of problems. The lecture is mostly concerned with particular forms of constraint systems which capture flow problems (or program analysis problem in general). In addition, we discuss under which circumstances those kind of constraint systems have solutions, in particular have *unique best* solutions. This is a very welcome sitation and deserve a thorough treatment.

Here, we just hint at a very simple strategy (also only sketched on a very high-level), which is non-deterministic. Being *non-deterministic* makes it not really directly an implementation unless one wishes to make use of some random-generator which helps to implement the non-determinism. There would in practice no point in doing so, instead one would aim for *deterministic* solution, perhaps realizing specific stratetgies or heuristic.

Why then bother with a non-deterministic description at all? Basically it's to separate the question of *correctness* of the algorithm from the question of *efficiency*. As it turns out, the kind of constraints we are considering and the solution domains have the *very, very* desirable property that

> when facing a non-deterministic, choice, no matter how one resolve it, the one never make a "wrong choice".

So the choice does not really matter except for how fast the algorithm terminates, i.e., how efficient the algo runs. Even there are perhaps smarter and less smart choices at each point, there are not real *wrong* ones. As a consequence, there is no **backtracking**. And that's crucial for being usable. As a side remark: it's well-known that many forms of constraints, even simple ones as boolean constraints ("SAT-solving") don't have this favorable property and there is basically no way around solving those by brute force *combinatorial exploration*. Indeed, SAT (boolean satisfiability) one of the most famous NP complete problems there is (and the first one for which that was proven). Fortunately, for simple data flow equations, things look much brighter.

Chaotic iteration

- back to data flow/reaching def's
- goal: **solve**

$$\vec{RD} = F(\mathsf{RD}) \quad \text{or} \quad \vec{RD} \sqsubseteq F(\mathsf{R\vec{D}})$$

- F: monotone, finite domain

straightforward approach

init $\mathsf{R\vec{D}}_0 = F^0(\emptyset)$

iterate $\mathsf{R\vec{D}}_{n+1} = F(\mathsf{R\vec{D}}_n) = F^{n+1}(\emptyset)$ until stabilization

- approach to implement that: **chaotic iteration**
- non-deterministic stategy
- abbreviate:

$$\vec{\mathsf{RD}} = (\mathsf{RD}_1, \ldots, \mathsf{RD}_{12})$$

Chaotic iteration (for RD)

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

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

1.6 Conclusion

The introductory part touched upon different topics. The approaches are also related, i.e., it's sometimes a bit of a matter of preference if one represents a problem directly as flow equations, type systems etc. Things not covered in the introduction (but probably later are complications in the while language, like procedure calls or pointers). Also, there will be other analyses besides reaching definitions (and a systematic common overview over similar analyses (knows as monotone framework). Also we go into the underlying theory (lattices) as well has considering in which way to establish that the various analysis are *actually* a sound overapproximation of the program behavior ("soundness", "correctness", "safe approximation", all mean the same).

Chapter 2 Data flow analysis

What is it about?

Learning Targets of this Chapter

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

Contents

2.1 Introduction

In this part we cover classical *data flow analysis*, first in a few special, specific analyses, among other ones, one more time reaching definitions. Besides that, also different other well-known ones. Those analyses are based on very similar common principles, which then lead to the notion of *monotone framework*. All of this is done for the simple while language from the general introduction. We also have a look at important extensions. One is the treatment of procedures. Those will be *first-order* procedures, not higher-order procedures. Nonetheless, they are already complicating the data flow problem (and its complexity), leading to what is known as *context-sensitive* analysis. Another extension deals with dynamically allocated memory on *heaps*. Analyses that deal with that particular language feature are known as *alias analysis*, *pointer analysis*, and *shape analysis*. Also we might cover SSA this time.

2.2 Intraprocedural analysis

2.2.1 Determining the control flow graph

While language and control flow graph

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

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

The latter 2 functions are of the following type:

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

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

(2.2)

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

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 unique entry is not the same as the notion of "isolated" entry (mentioned already in the introduction). Isolated would mean: the entry is not the *target* of any edge. That's not the case, for instance for the 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.

Using those, determining the edges, by a function

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

works as follows:

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

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

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

They are defined straightforwardly as follows:

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

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

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

Flow and reverse flow

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

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

Program of interest

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

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

"l labels *the* block B"

- even better: *unique* labelling

Concerning *label consistency*: indeed, unique labelling is better. Otherwise nodes of the graph are "overlaid", i.e., there will be confusion wrt. predecessors and successors. Unique labelling is an very natural condition. When labelling the syntax (and building the control flow graph), one simply generate one label or node after the other, then naturally the blocks are unqiuely labelled. Label consistency, but with non-unique labelling looks rather unnatural, at the first sight.

See also the operational semantics later, which preserves label consistency but not unique labelling (in the case of unrolling a while-construct). Actually, the semantics would preserve a better property, it seems to me. Not only is the labelling "consistent" in the sense defined here. But also the edges and neighbors of a node remains comparable. But the book does not point that out.

2.2.2 Available expressions

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

Of course, it 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 expression containing variable, the content of variables mentioned in such expression may or may not have changed comparing different occurences of the same expression, and that has to be figured out via a specific *data flow analysis*, namely "available expressions" analysis.

Avoid recomputation: Available expressions

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

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

- usage: avoid *re-computation*

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

Available expressions: general

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

2 ingredients of intra-block flow

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

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

In the introduction, the reaching definition analysis was done without explicitly mentioning kill and generate, but they where there *implicitly* anyway (for the intra-block equations).

Available expressions: types

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

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

- analysis: pair of functions

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

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

Intra-block flow specification: Kill and generate

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

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

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

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

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

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

split into

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

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

Flow equations for AE

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

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

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

Explanation 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 available expressions. One conceptual difference is the replacement of $\bigcap$ (must) by $\bigcup$ (may).

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

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

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

Available expressions

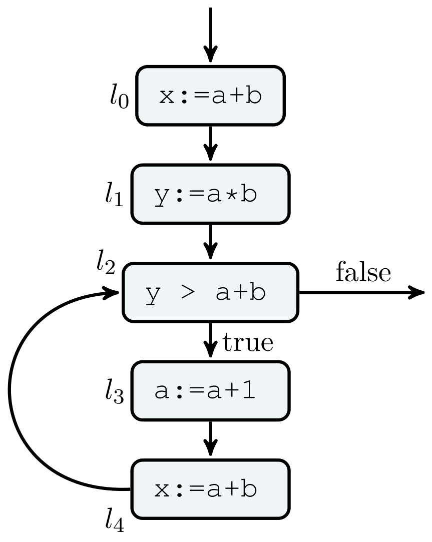
- *forward* analysis (as RD)
- interest in *largest* solution (unlike RD)

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

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

Example AE

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



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

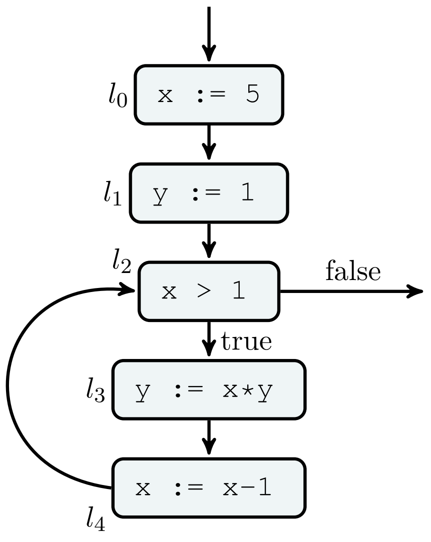
As a side remark: before we mentioned that available expressions analysis is useful for common sub-expression elimination. The example shows that one has to be careful with that, nonetheless.

2.2.3 Reaching definitions

Reaching definitions

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

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



Reaching definitions: types

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

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

- generation and killing:

$$\mathit{kill}_{\mathsf{RD}}, \mathit{gen}_{\mathsf{RD}} : \mathbf{Blocks}_* \to 2^{\mathbf{Var}_* \times \mathbf{Lab}_*^?}$$

- analysis: pair of mappings

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

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

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

Reaching defs: kill and generate

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

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

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

split into

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

Flow equations for RD

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

$$
\mathsf{RD}_{\mathit{exit}}(l) = \mathsf{RD}_{\mathit{entry}}(l) \setminus \mathit{kill}_{\mathsf{RD}}(B^l) \cup \mathit{gen}_{\mathsf{RD}}(B^l)
$$

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

- same order of kill/generate

2.2.4 Very busy expressions

Introduction

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

This change of perspective also means, that VB is an example of a *backward* analysis. The natural way of analysing very busy expressions is: at the place where an expression is actually used, immediately in front of that place it's definitely very busy. And then from there, let the information flow *backward*: 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 follow: if an expression is very busy, it may pay off to calculate it already now, i.e., it can be used for a program transformation, that moves the calculation of expression in an "eager" fashion as early as posssible. Transformations like this are known as expression "hoisting".

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

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

Very busy expressions

```
if    $[a > b]^1$
then  $[x := b - a]^2; [y := a - b]^3$
else  $[a := b - a]^4; [x := a - b]^5$
```

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

- usage: expression "hoisting"

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

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

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

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

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

Very busy expressions: types

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

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

- analysis: pair of mappings

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

Very busy expr.: kill and generate

core of the intra-block flow specification

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

A comparison with the kill and generate functions for AE might be interesting. First of all, in both cases, the functions have the *same types*, i.e., operate on the same domains. Of course, one difference is, that now the flow is backwards. For the blocks without side effects, this does not matter, i.e., the generate function is identical in both cases (the kill-function as well, of course). For the assignment, there are obviously differences. Let's first look at the kill-case. Literally, the two definitions *coincide,* but they have a different intuition (backward vs. forward). *Here* for VB we ask, because we are thinking backwards, which expressions are very busy at the *entry* of that block. Of course, also the killing works backwards: whatever was very busy at the exits 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 of *first* kill and *then* generate

:BEAMER$_{\text{env}}$: againframe :BEAMER$_{\text{ref}}$: frame.AE.killgenerate

Flow equations.: VB$^{=}$

split into

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

however: everything works backwards now

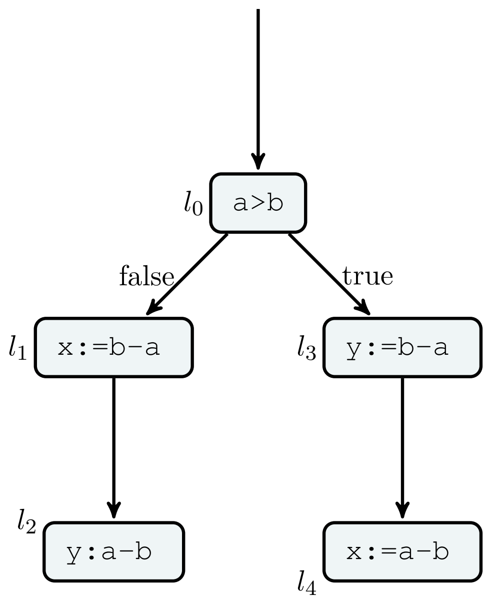
Flow equations: VB

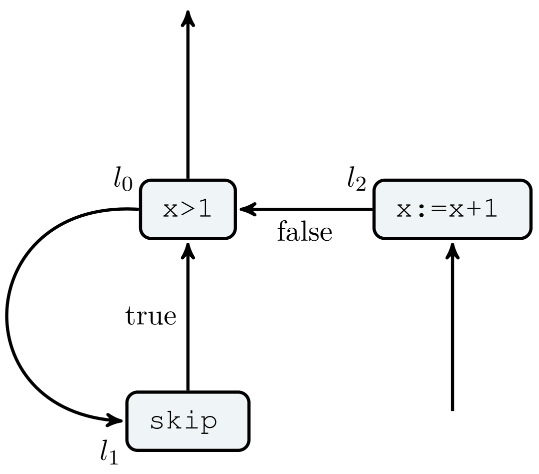
$$
\begin{aligned}
\mathsf{VB}_{exit}(l) &= \begin{cases} \emptyset & l \in \mathit{final}(S_*) \\ \bigcap\{\mathsf{VB}_{entry}(l') \mid (l', l) \in \mathit{flow}^R(S_*)\} & \text{otherwise} \end{cases} \\
\mathsf{VB}_{entry}(l) &= \mathsf{VB}_{exit}(l) \setminus \mathit{kill}_{\mathsf{VB}}(B^l) \cup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

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

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





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

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

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

Assuming that $x > 0$, 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* and the beginning of the program, as there is a possibility that x is used (in l_2) and that possiblity does not involve making an argument about infinite paths. Unlike the situation of the very busy expressions, this seems intuitively plausible.

2.2.5 Live variable analysis

Introduction

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

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

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

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

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

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

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

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

- usage: *register allocation*

Live variables are about: when is a variable still "needed". If not needed, one can *free* the space. In some sense and very generally, the question resembles reaching definition in a superficial sense, 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 variabe 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 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 loose efficiency. Making the opposite approximation, marking an actually live variable erronously as non-live, may lead to errors and is therefore unsafe.

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

Live variables: types

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

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

- analysis: pair of functions

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

Live variables: kill and generate

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

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

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

Flow equations $\mathsf{LV}^=$

split into

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

however: everything works backwards now

Flow equations LV

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

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

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

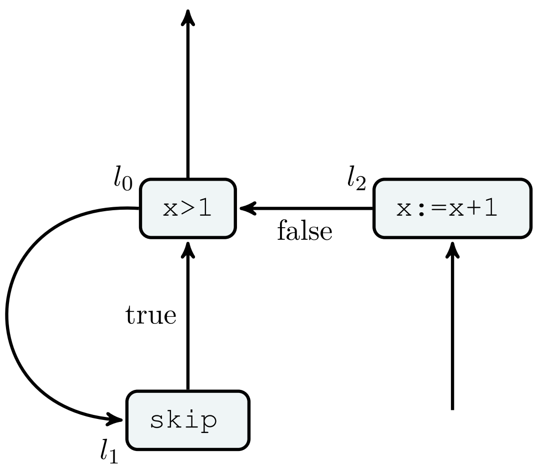
Example

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

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

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

Looping example



2.3 Theoretical properties and semantics

2.3.1 Semantics

Introduction

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 what the semantics is supposed to be also pays off when implementing a language, after all, the ultimately running program is expecteed to implement exactly the specified semantics, down to the actual machine code on a particular platform. Leaving the semantics up-to the implementation or the platform is not considered a very dignified engineering approach ("the semantics of a program is what happens if you run it, you'll see.").

In this section we will precisely define the semantics of the while-language. The semantics is defined on a rather high-level, on the level of the abstract syntax, and the task of the compiler would be to preserve exactly the syntax 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 described the *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., it's 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, even there different flavors and styles exists. Perhaps later, for the more complex functional languages, the lecture covers some variations.

Relating programs with analyses

- analyses
  - intended as (static) *abstraction* or overapprox. of real program behavior
  - so far: *without real connection* to programs
- soundness of the analysis: **safe** analysis

- but: *behavior* or *semantics* of programs not yet defined
- here: "easiest" semantics: *operational*
- more precisely: *small-step SOS* (structural operational semantics)

States, configs, and transitions

fixing some data types

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

Transitions

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

Semantics of expressions

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

simplifying assumption: no errors

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

clearly:

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

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

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

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

SOS

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

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

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

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

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

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 2.3.1.

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

Correctness of live analysis

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

LV constraint system

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

$$\mathsf{LV}_{entry}(l) \supseteq \mathsf{LV}_{exit}(l) \setminus \mathit{kill}_{\mathsf{LV}}(B^l) \cup \mathit{gen}_{\mathsf{LV}}(B^l)$$

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

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

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

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

Equational vs. constraint analysis

Lemma 2.3.2. *1. If live* $\models \mathsf{LV}^{=}$*, then live* $\models \mathsf{LV}^{\subseteq}$
2. The least solutions of live $\models \mathsf{LV}^{=}$ *and live* $\models \mathsf{LV}^{\subseteq}$ *coincide.*

2.3.2 Intermezzo: Lattices

Intermezzo: orders, lattices. etc.

as a reminder:

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

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 requires joins or "least upper bounds" $a \sqcup b$ and or "greatest lower bounds" $a \sqcap b$ to exists. 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 the montone framework is happy. In particular, if we are dealing with *finite* lattices, which is an important case, we don't need to consider *infinite* sets, and "standard" lattices with binary meets and joins (and least and largest elements) are complete already.

Fixpoints

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

- **fixpoint:** $f(l) = l$
$$\mathit{Fix}(f) = \{l \mid f(l) = l\}$$
- f *reductive* at l, l is a **pre-fixpoint** of f: $f(l) \sqsubseteq l$:
$$\mathit{Red}(f) = \{l \mid f(l) \sqsubseteq l\}$$
- f *extensive* at l, l is a **post-fixpoint** of f: $f(l) \sqsupseteq l$:
$$\mathit{Ext}(f) = \{l \mid f(l) \sqsupseteq l\}$$

Define "lfp" / "gfp"

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

The last display just gives the names to the two elements of the lattice defined by the corresponding right-hand sides. We know tha those elements are existing thanks to the fact that L is a complete lattice (and it's very easy to see that meets and joins are unique, that means the $\mathit{lfp}(f)$ and $\mathit{gfp}f$ are well-defined elements of the lattice). The chosen names somehow suggest that the two thusly defined elements are the least fixpoint, resp. the greatest fixpoint of the monotone function f.

But, so far *lfp* and *gfp* is just a suggestive choice of name. It requires an separate *argument* that the elements are actually fixpoints, and the least, resp. the largest fixpoint as that as well. Finally, if we take it really serious, an argument should be found that allows to speaking of *the* least fixpoint. If there is more than one least fixpoint, one should avoid talking about "the least fixpoint" (same for the largest fixpoint). The argument for uniquess of least fixpoints (or for greatest fixpoint) is very simple though, similar to arguing for the uniqueness of "the least upper bound" etc.

If one would carry out the argument, i.e., the proof, that all fits together in the sense that the $\mathit{lfp}(f)$ 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 lattices **is exactly what guarantees that those fixpoints exists**. Exactly that, nothing less and nothing more, If your framework has monotone functions and is based on a complete lattice, it works. If not, it does not work, very simple.

That explains the importance of lattices and monotone function. 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 $\mathit{lfp}(f)$ like above and see if I could prove something iteresting about it, perhaps it's a fixpoint?". But as said, that is speculation.

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

Tarski's theorem

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

Theorem 2.3.3. *L: complete lattice, $f : L \to L$ monotone.*

$$\begin{aligned} \mathit{lfp}(f) &\triangleq \bigsqcap \mathit{Red}(f) \in \mathit{Fix}(f) \\ \mathit{gfp}(f) &\triangleq \bigsqcup \mathit{Ext}(f) \in \mathit{Fix}(f) \end{aligned} \tag{2.6}$$

- Note: *lfp* (despite the name) is *defined* as glb of all pre-fixpoints
- The theorem (more or less directly) implies *lfp* is the *least* fixpoint

Fixpoint iteration

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

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

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

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

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

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

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

Basic preservation results

Lemma 2.3.4 ("Smaller" graph $\rightarrow$ less constraints)**.** *Assume live* $\models \mathsf{LV}^{\subseteq}(S_1)$. *If flow*$(S_1) \supseteq$ *flow*(S_2) *and blocks*$(S_1) \supseteq$ *blocks*(S_2)*, then live* $\models \mathsf{LV}^{\subseteq}(S_2)$.

Corollary 2.3.5 ("subject reduction")**.** *If live* $\models \mathsf{LV}^{\subseteq}(S)$ *and* $\langle S, \sigma\rangle \rightarrow \langle \acute{S}, \acute{\sigma}\rangle$*, then live* $\models$ $\mathsf{LV}^{\subseteq}(\acute{S})$

Lemma 2.3.6 (Flow)**.** *Assume live* $\models \mathsf{LV}^{\subseteq}(S)$. *If* $l \rightarrow_{\mathit{flow}} l'$*, then* $\mathit{live}_{\mathit{exit}}(l) \supseteq \mathit{live}_{\mathit{entry}}(l')$.

The three mentioned results are actually pretty straightforward, resp. express properties of the live variable analysis which should be (after some reflection) pretty obvious. Analgous results would hold for other data flow analyses. Lemma 2.3.4 compares the analyses results for two programs S_1 and S_2, where S_2 has a "smaller" control-flow graph (less edges and/or less blocks). Since the control flow graph directly corresponds to sets of constraints, removing parts of the graph means removing constraints. That means, *more* solutions are possible, which is expressed by the lemma (*live* $\models \mathsf{LV}^{\subseteq}(S)$ means that *live* (an assignment of liveness information to all variables of the constraint system) satisfies the constraint system of the program S.

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

The Corollary 2.3.5 is a direct consequence of that. In general, that's what the term "corollary" means: an immediate interesting follow-up of a preceding lemma or theorem etc.

However, the result is *not* without subtelty. It has to do with the step $\langle S, \sigma\rangle \rightarrow \langle \acute{S}, \acute{\sigma}\rangle$, 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.*

The last lemma is a direct consequence of the construction (backward may analysis).

These lemmas as such are not interesting in themselves.

Correctness relation

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

$\Rightarrow$

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

$$\sigma_1 \sim_V \sigma_2 \text{ iff } \forall x \in V. \sigma_1(x) = \sigma_2(x) \tag{2.7}$$

$$\begin{tikzcd}
\langle S, \sigma_1 \rangle \arrow[r] \arrow[d, dash, "\sim_V"] & \langle S', \sigma_1' \rangle \arrow[r] \arrow[d, dash, "\sim_{V'}"] & \ldots \arrow[r] & \langle S'', \sigma_1'' \rangle \arrow[r] \arrow[d, dash, "\sim_{V''}"] & \sigma_1''' \arrow[d, dash, "\sim_{X(l)}"] \\
\langle S, \sigma_2 \rangle \arrow[r] & \langle S', \sigma_2' \rangle \arrow[r] & \ldots \arrow[r] & \langle S'', \sigma_2'' \rangle \arrow[r] & \sigma_2'''
\end{tikzcd}$$

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

Correctness (1)

Lemma 2.3.7 (Preservation inter-block flow)**.** *Assume live* $\models \mathsf{LV}^{\subseteq}$. *If* $\sigma_1 \sim_{X(l)} \sigma_2$ *and* $l \to_{flow} l'$, *then* $\sigma_1 \sim_{N(l')} \sigma_2$.

Correctness

Theorem 2.3.8 (Correctness)**.** *Assume live* $\models \mathsf{LV}^{\subseteq}(S)$.

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

$$\begin{tikzcd}
\langle S, \sigma_1\rangle \arrow[r] \arrow[d, dash, "\sim_{N(init(S))}"] & \langle \acute{S}, \acute{\sigma}_1\rangle \arrow[d, dash, dotted, "\sim_{N(init(\acute{S}))}"] \\
\langle S, \sigma_2\rangle \arrow[r, dotted] & \langle \acute{S}, \acute{\sigma}_2\rangle
\end{tikzcd}
\qquad
\begin{tikzcd}
\langle S, \sigma_1\rangle \arrow[r] \arrow[d, dash, "\sim_{N(init(S))}"] & \acute{\sigma}_1 \arrow[d, dash, dotted, "\sim_{X(init(S))}"] \\
\langle S, \sigma_2\rangle \arrow[r, dotted] & \acute{\sigma}_2
\end{tikzcd}$$

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

Correctness (many steps)

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

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

2.4 Monotone frameworks

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

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

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

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

Monotone framework: general pattern

$$\begin{aligned} \mathit{Analysis}_\circ(l) &= \begin{cases} \iota & \text{if } l \in E \\ \bigsqcup\{\mathit{Analysis}_\bullet(l') \mid (l', l) \in F\} & \text{otherwise} \end{cases} \\ \mathit{Analysis}_\bullet(l) &= f_l(\mathit{Analysis}_\circ(l)) \end{aligned} \tag{2.8}$$

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

The definition is "generic" as it leaves open the alternatives "may" vs. "must" as well as "forward" vs. "backard". Also the domain flow information of interest is not fixed, neither is 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 one specific conditions of the intended analysis and the language. One final ingredient is the *transfer function*. We have encountered that implicitly for the *intra-block* data flow. It's only that we did not explicitly called 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 function (as many analysis domain are sets of information of interest), but not all. The general monotone framework simply requires a function that transform flow information on one end of a basic block to flow information at the other end. For a forward analysis, the flow information at the exit of a basic block is expressed as a function on the entry of the block, and for backward information, it's the other way around.

Monotone frameworks

direction of flow:

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

sort of solution

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

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

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

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

$$\begin{aligned}
\mathit{Analysis}_\circ(l) &= \iota^l_E \sqcup \bigsqcup\{\mathit{Analysis}_\bullet(l') \mid (l', l) \in F\} \\
&\quad \text{where } \iota^l_E = \begin{cases} \iota & \text{if } l \in E \\ \bot & \text{if } l \notin E \end{cases} \\
\mathit{Analysis}_\bullet(l) &= f_l(\mathit{Analysis}_\circ(l))
\end{aligned} \tag{2.9}$$

where $l \sqcup \bot = l$

Explanation

Let's compare it to equation (2.8), where we did it for isolated entries: First remember that E ("extremal") is an initial block (or a final one). Remember also that isolated entries does not mean that the is only *one* intial/final block, only that there is no *loop-back*. Let's consider the forward case. In this case, equation (2.8) makes sense. The distinguishing case is the one for *inter*-block flow, which says something for the *entry* of a block (in the forward case). And there are exactly two separate cases: if l refers to the initial block, then there is *only* the initial information ι. Note that this does not mean that there is only one initial label. Otherwise, if it is a non-initial label, then the intial flow is *not* mentioned in the equation/constraint. Note that it seems possible, that a non-initial block has no inflowing arc. Probably for the while-language that is not the case, however, the definitions seem to allow it. In this case the $\bigsqcup$ as $\bigcup$ gives the empty set, which probably makes sense. Obviously, (2.8) makes no sense without isolated entries, because for initial labels, the equations overlooks the flowing-back information. The new equation (2.9) repairs that in that it adds to the initial labels also the combined information from the posts of the connected nodes. However, it must of course *not inject* ι into non-initial nodes, hence the definition of ι^l_E.}

Basic definitions: property space

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

The *property space* (here called L) captures the "information of interest" (sets of variables ...). Technically, it needs to be some form of *lattice* (see the corresponding section later). 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 a special case of a lattice (and an important one in the context of the lecture).

Transfer functions

$$f_l : L \to L$$

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

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

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

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

Summary

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

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

Instead of the above condition, one might require

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

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

The 4 classical examples

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

Overview over the 4 examples

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

2.5 Equation solving

Solving the analyses

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

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

MFP

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

Chaotic iteration

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

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

Worklist algorithms

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

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

- for more details (different traversal strategies): see Chap. 6 from [3]

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 $\mathit{Analysis}_\circ$
- 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. Here, only the *entry* blocks are stored. An array is of course a good representation of a finite function.

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

Code

```
Input:   $(L, \mathcal{F}, F, E, \iota, f)$
Output:  $MFP_\circ, MFP_\bullet$
Method:  step 1: initialization
                $W := \mathsf{nil}$;
                for all $(l, l') \in F$ do  $W := (l, l') :: W$;
                for all  $l \in F$ or $\in E$ do
                   if  $l \in E$ then  $Analysis[l] := \iota$
                                else  $Analysis[l] := \bot_L$;
         step 2: iteration
                while  $W \neq \mathsf{nil}$ do
                   $(l, l')$ := ( fst(head(W)), snd(head(W)));
                   W := tail W;
                   if  $f_l(Analysis[l]) \not\sqsubseteq Analysis[l']$
                   then   $Analysis[l'] := Analysis[l'] \sqcup f_l(Analysis[l])$;
                          for all  $l''$ with  $(l', l'') \in F$ do
                                $W := (l', l'') :: W$;
          step 3: presenting the result:
                   for all  $l \in F$ or $\in E$ do
                      $MFP_\circ(l) := Analysis[l]$;
                      $MFP_\bullet(l) := f_l(Analysis[l])$
```

ML Code

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

MFP: properties

Lemma 2.5.1. *The algo*

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

Proof. • termination: ascending chain condition & loop is enlarging

- least FP:
  - invariant: array always below $\mathit{Analysis}_\circ$
  - at loop exit: array "solves" (in-)equations

□

Time complexity

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

$\Rightarrow$

$$O(e \cdot h) \tag{2.10}$$

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

2.6 Interprocedural analysis

2.6.1 Introduction

Adding procedures

- so far: *very simplified* language:
  - minimalistic imperative language
  - reading and writing to variables plus
  - simple controlflow, given as flow graph

- now: *procedures*: **interprocedural** analysis
- complications:
  - calls/return (control flow)
  - parameter passing (call-by-value vs. call-by-reference)
  - scopes
  - potential *aliasing* (with call-by-reference)
  - higher-order functions/procedures
- here: top-level procedures, mutual recursion, call-by-value parameter + call-by-result

Syntax

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

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

- procedure names p
- statements

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

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

Example: Fibonacci

2.6.2 Semantics

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

Explanation Next comes the adaptation of the definition of the flow graph. To do so, we need to adapt and extend the definitions of flow, block, etc. The new part deals, obviously, with the procedures. The basic trick is that we introduce *new* kinds of edges to deal with the procedures. The definition/presentation proceeds (in the slides) in *two* steps, first the call sites, afterwards the procedures themselves.

Block, labels, etc.

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

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

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

For procedure declaration

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

“Standard” flow of complete program

not yet interprocedural flow (IF)

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

side remark: S_*: notation for complete program “of interest”

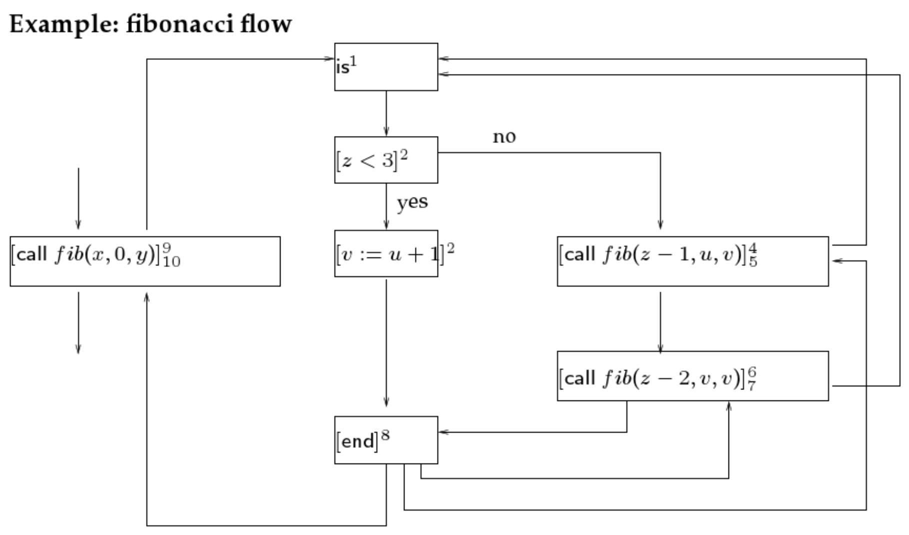
New kind of edges: Interprocedural flow (IF)

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

IF

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

Example: fibonacci flow


Example: fibonacci flow
is^1
$[z < 3]^2$
no
yes
[call $fib(x,0,y)]^9_{10}$
$[v := u + 1]^2$
[call $fib(z-1,u,v)]^4_5$
[call $fib(z-2,v,v)]^6_7$
[end]^8

Semantics: stores, locations,. . .

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

Representation of "memory"

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

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

Explanations $\to_{\mathit{fin}}$ represents *finite* partial functions.

SOS steps

- steps *relative* to *environment* ρ

$$\rho \vdash_* \langle S, \varsigma \rangle \to \langle \acute{S}, \acute{\varsigma} \rangle$$

or

$$\rho \vdash_* \langle S, \varsigma \rangle \to \acute{\varsigma}$$

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

Call-rule

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

Bind-construct

$$\frac{\acute{\rho} \vdash_* \langle S, \varsigma\rangle \to \langle \acute{S}, \acute{\varsigma}\rangle}{\rho \vdash_* \langle \mathsf{bind}\ \acute{\rho}\ \mathtt{in}\ S\ \mathtt{then}\ z := y, \varsigma\rangle \to \langle \mathsf{bind}\ \acute{\rho}\ \mathtt{in}\ \acute{S}\ \mathtt{then}\ z := y, \acute{\varsigma}\rangle}\ \textsc{Bind}_1$$

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

- bind-syntax: "runtime syntax"

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

2.6.3 Analysis

Transfer function: Naive formulation

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

Naive

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

Equation system ("naive" version")

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

with

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

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

The equational system here is *without* the assumption of isolated entries/exits. That corresponds to page 64 in the book.

[1]Not covered in the lecture.

2.6.4 Paths

Paths

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

Paths

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

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

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

Meet over all paths

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

MOP

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

MOP vs. MFP

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

Lemma 2.6.1.

$$MFP_\circ \sqsupseteq MOP_\circ \text{ and } MFP_\bullet \sqsupseteq MOP_\bullet \tag{2.11}$$

In case of a distributive framework

$$MFP_\circ = MOP_\circ \text{ and } MFP_\bullet = MOP_\bullet \tag{2.12}$$

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

MVP

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

⇒ capture the nesting structure
- from MOP to **MVP**: “meet over all **valid** paths”
- *complete* path:
  - appropriate call-nesting
  - all calls are answered

Complete paths

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

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

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

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

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

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

The interpretation is as follows. There is only one complete path from a label to itself, that's the trivial path. The second rule just splits off one label on the left (one could do also differently). Also in the third rule, there is a split-off of the first step. A *terminating* execution will have a complete path. There are only a finite number of productions.

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

Example: Fibonacci

- concrete grammar for fibonacci program:

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

Valid paths (context-free grammar)

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

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

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

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

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

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

MVP

- adapt the definition of paths

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

- **MVP** solution:

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

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

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

2.6.5 Context-sensitive analysis

Contexts

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

⇒ instead of MVP: **"embellish"** MFP

$$\delta \in \Delta \tag{2.13}$$

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

⇒ "embellishment": adding **contexts**

embellished monotone framework

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

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

Embellishment, notationally, is indicated by a hat on top. The following will proceed in two stages. The intra-procedural part and the interprocedural 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 slightly by embellishing it (indicated by the hatted syntax). That will involve a change in the lattice and other concomitant changes. Consequently, also the intraprocedual part needs to be adapted, basically taking care of the embellishement, i.e., taking care of the contexts. Taking care basically is rather trivial and 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 with the embelished transfer functions and other parts of the definition of the monotone framework. But the keyword is: the intra-procedural part is "basically unchanged".

Intra-procedural: basically unchanged

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

$$\hat{f}_l(\hat{l})(\delta) = f_l(\hat{l}(\delta)) \tag{2.14}$$

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

$$\begin{aligned} A_\bullet(l) &= \hat{f}_l(A_\circ(l)) \\ A_\circ(l) &= \bigsqcup\{A_\bullet(l') \mid (l', l) \in F \text{ or } (l'; l) \in F)\} \sqcup \iota_E^{\hat{l}} \end{aligned} \tag{2.15}$$

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

There is an unfortunate notational collision: Lattice L with its elements on the one hand and labels/nodes in the CFG l from $\mathbf{Lab}_*$

Apart from that: The above definitions define $\hat{L}$ as a function of type $\Delta \to L$. That gives raise to some "higher-order" explanations of what the embellished framework means (see already equation (2.14). That is a clean explanation of what is going on, but one may also see it as follows.

Remember that in the unembellished framework, a solution of a problem is a mapping from the nodes of the cfg to elements of the lattice. Let's use N for the nodes or labels (not L ...). To be very precise, we are interested not in a mapping from nodes to the lattice but from the entries and the exits of the nodes to the lattice, but let's ignore that for now.

That means, that an unembelished solution is of the type $N \to L$, whereas *now*, the solution is of type

$$N \to \Delta \to L\ .$$

That is the same as

$$(N \times \Delta) \to L\ .$$

Seen like that, 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.

Sign analysis (unembellished)

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

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

$$f_l^{sign}(Y) = \bigcup\{\phi_l^{sign}(\sigma^{sign}) \mid \sigma^{sign} \in Y\} \tag{2.16}$$

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

$$\phi_l^{sign}(\sigma^{sign}) = \{\sigma^{sign}[x \mapsto s] \mid s \in [\![a]\!]^{\mathcal{A}_{sign}}_{\sigma^{sign}}\} \tag{2.17}$$

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

As a side remark: for what is called sign-analysis, that's not the only possible choice. An *alternative* to L_{sign} would be to use a function $\mathbf{Var} \to 2^{\mathbf{Sign}}$. It would be a "state with abstract values" (where an abstract value is a set of concrete values) as opposed to an "abstract state" consistting of a set of concrete states. The alternative interpretation would be "weaker", i.e., more abstract.

Anyway: The above definition proceeds in *3 steps:* At the core is the semantic function $[\![\_]\!]$. This function is for *expressions,* and is already non-deterministic. Eq. (2.17) reflects the effect of an assignment for one abstract state and (2.16) is the transfer function (lifted pointwise).

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

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

On the next slides, we will *embellish* the analysis, but since we are not yet in the interprocedural part, the embellishment is not very interesting, just a "lifting" to the embellished setting.

Sign analysis: embellished

$$\begin{aligned} \hat{L}_{sign} &= \Delta \to L_{sign} \\ &= \Delta \to 2^{\mathbf{Var}_* \to \mathbf{Sign}} \simeq 2^{\Delta \times (\mathbf{Var}_* \to \mathbf{Sign})} \end{aligned} \tag{2.18}$$

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

$$\hat{f}_l^{sign}(Z) = \bigcup\{\{\delta\} \times \phi_l^{sign}(\sigma^{sign}) \mid (\delta, \sigma^{sign}) \in Z\} \tag{2.19}$$

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 (2.19) just does *nothing* with the δ, 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.

Inter-procedural

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

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

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

2 transfer functions

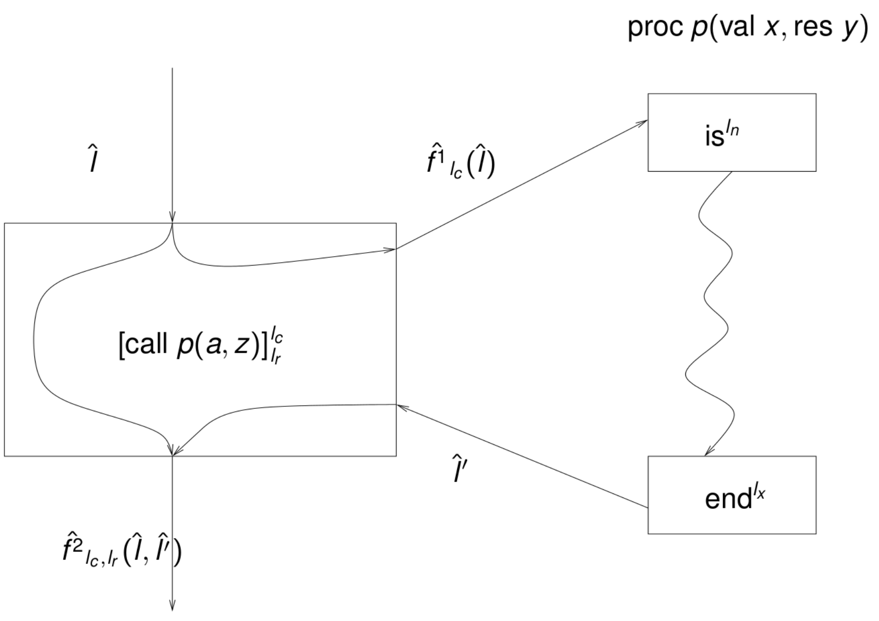
1. for calls: $\hat{f}1_{l_c} : (\Delta \to L) \to (\Delta \to L)$

$$A_\bullet(l_c) = \hat{f}1_{l_c}(A_\circ(l_c)) \tag{2.20}$$

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

$$A_\bullet(l_r) = \hat{f}2_{l_c,l_r}(A_\circ(l_c), A_\circ(l_r))) \tag{2.21}$$

Procedure call

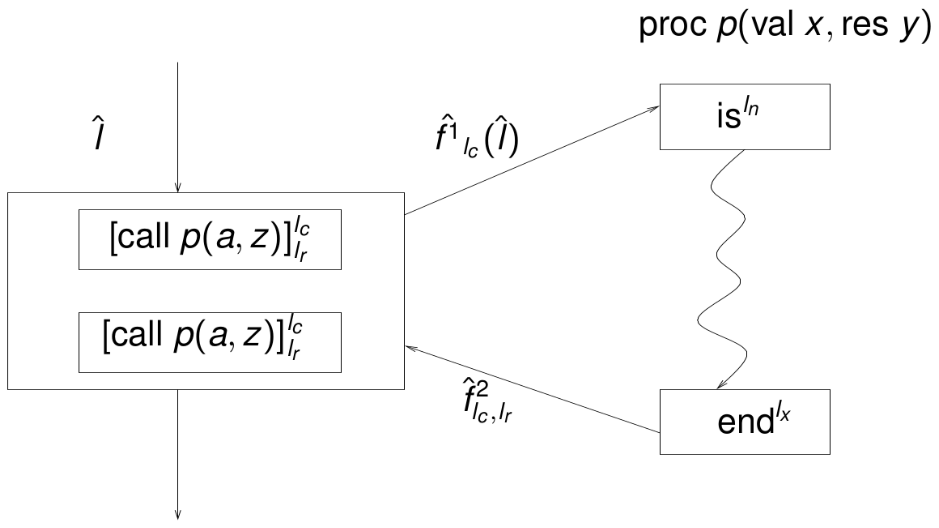

proc p(val x, res y)
$\hat{l}$
$\hat{f}1_{l_c}(\hat{l})$
is^{l_n}
[call $p(a, z)]^{l_c}_{l_r}$
$\hat{l}'$
end^{l_x}
$\hat{f}2_{l_c,l_r}(\hat{l}, \hat{l}')$

Note again the unfortunate notational collision: $\hat{l}$: element of embellished lattice (abstract value), l_c etc: nodes/labels in the control flow graph. The situation may become even more confusing for analyses like RD: there *labels* (which are nodes in the control-flow graph) are part of the values of interest and thus also elements of the lattice.

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

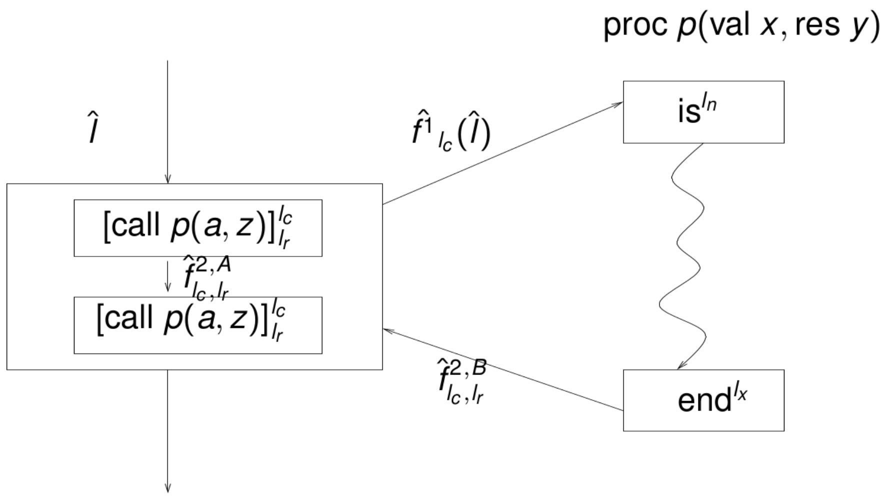
Ignoring the call context

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


proc p(val x, res y)
$\hat{l}$
$\hat{f}^1{}_{l_c}(\hat{l})$
is^{l_n}
$[\text{call } p(a,z)]^{l_c}_{l_r}$
$[\text{call } p(a,z)]^{l_c}_{l_r}$
$\hat{f}^2_{l_c,l_r}$
end^{l_x}

Merging call contexts

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


proc p(val x, res y)
$\hat{l}$
$\hat{f}^1{}_{l_c}(\hat{l})$
is^{l_n}
$[\text{call } p(a,z)]^{l_c}_{l_r}$
$\hat{f}^{2,A}_{l_c,l_r}$
$[\text{call } p(a,z)]^{l_c}_{l_r}$
$\hat{f}^{2,B}_{l_c,l_r}$
end^{l_x}

Context sensitivity

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

⇒ context *sensitive* analysis ⇒ more precision + more effort

In the following: 2 *specializations*:

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

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

Call strings

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

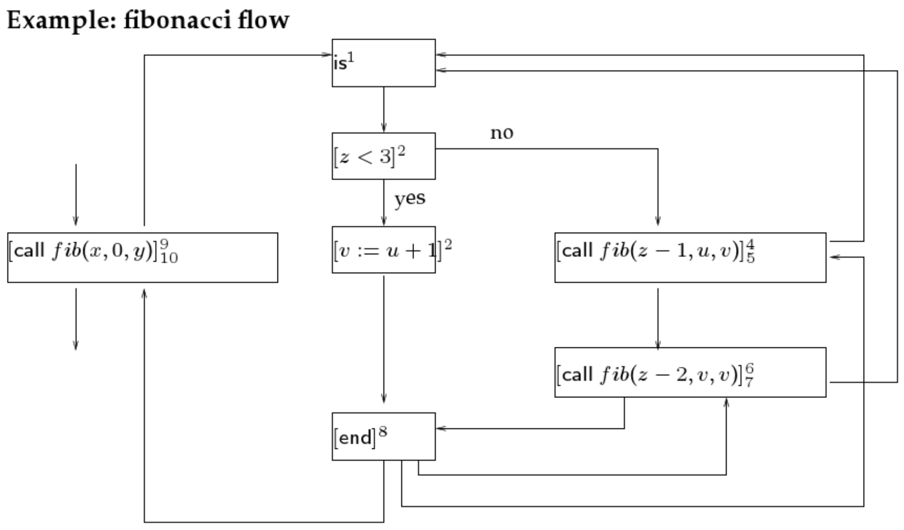
$$\Delta = \mathbf{Lab}^* \qquad \text{call strings}$$

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

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

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

Fibonacci flow


Example: fibonacci flow
is^1
$[z < 3]^2$
no
yes
[call $fib(x, 0, y)]^9_{10}$
$[v := u + 1]^2$
[call $fib(z-1, u, v)]^4_5$
[call $fib(z-2, v, v)]^6_7$
[end]^8

Fibonacci call strings

some call strings:

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

similar, but not same as valid paths The call strings here are *not* the same as the valid paths. Both concepts are related, though. The 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.

Note that if a function body calls another function 2 times, then the two call sites may still be confused!

Transfer functions for call strings

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

Transfer functions

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

$$\begin{aligned}\hat{f}^1_{l_c}(\hat{l})([\delta, l_c]) &= f^1_{l_c}(\hat{l}(\delta)) \\ \hat{f}^1_{l_c}(\_) &= \bot\end{aligned} \tag{2.22}$$

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

$$\hat{f}^2_{l_c,l_r}(\hat{l}, \hat{l}')(\delta) = f^2_{l_c,l_r}(\hat{l}(\delta), \hat{l}'([\delta, l_c])) \tag{2.23}$$

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

Sign analysis (continued)

- so far: "unconcrete", i.e.,
- given some underlying analysis: how to make it context-sensitive
- call-strings as context
- now: apply to some simple case: signs
- remember: $\hat{L} \simeq 2^{\Delta \times (\mathbf{Var}_* \to \mathbf{Sign})}$ (see Eq. (2.18))
- before: standard embellished $\hat{f}^{\mathbf{Sign}}_l$ (with the help of $\phi^{\mathbf{Sign}}_l$)
- now: *inter-procedural*

Sign analysis: aux. functions ϕ

still unembellished

calls: abstract parameter-passing

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

returns (analogously)

$$\phi^{sign2}_{l_c,l_r}(\sigma^{sign}_1, \sigma^{sign}_2) = \{\sigma^{sign}_2[x, y, z \mapsto \sigma^{sign}_1(x), \sigma^{sign}_1(y), \sigma^{sign}_2(y)]\}$$

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

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

Sign analysis

calls: abstract parameter-passing + glueing calls-returns

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

Returns: analogously

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

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

The sign analysis was introduced before. The start was unembellished, just the context-non-sensitive case. There, the ϕ was done, as a pre-step for the unembellished transfer functions. The underlying lattice was *not* a mapping from variables to sets of signs (which would have been possible) but the more precise *sets of such mappings*. Those were called abstract states. A that point we had already the *embellished transfer function*, but only for the non-procedure case. Note also: in this particular setting: the embellished lattice, in general, is a mapping from contexts to the old lattice. Here, the lattice is *isomorphic* to *sets of pairs* (see equation (2.18) for that).

How does this work? It works as before, i.e., as for the intra-procedural analysis. The auxiliary function ϕ (for signs) in (2.17) when the example sign-example was introduced which was already used in the *unembellished* setting to define the (unembellished) f^{sign}_l. In the *embellished* setting in the old intra-procedural part, the ϕ is also not used for touching the paths δ.

Here we now see, that the auxiliary ϕ^{sign}_l is *split* into 2 functions $\phi^{sign1}_{l_c}$ and $\phi^{sign2}_{l_c,l_r}$. This is done analogous to the splitting of the transfer functions (the ϕ's are just auxiliary constructions to it anyway). As before, the ϕ's have *nothing* to do with the paths. But they have to be different, because of the **parameter passing** (for x and y e.g., in the call). Note that the value for y is set arbitrarily.

The interesting coupling is in δ and δ' (resp. Z and Z')!

The definition can be best understood into two states, both for calls and for returns. As before, we assume that the abstract denotational semantics for expression is given (which is already non-deterministic and not repeated here). The two stages are 1) defined "f" for one abstract state (resp. for a pair, in case of the return) and then 2) lift it to sets of such.

1. that one is for *parameter passing* on the abstract level. The s' is just because of the call-by-result semantics for the result parameter, it's just the same as in the semantics (remember the SOS).
2. The second stage, the real *transfer function*, lifts it to sets. Cf. also the unembellished lifting function (for signs) in equation (2.17). The important change is that now we have the contexts (as call strings) δ in the lattice

Call strings of bounded length

- 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 ($\Delta = \{\epsilon\}$)

Assumption sets

- **alternative** to call strings
- not tracking the path, but assumption about the state
- assume here: lattice

$L = 2^D$

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

restrict to only the last call

dependency on data only $\Rightarrow$

$$\Delta = 2^D$$

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

Transfer functions

- calls

$$\begin{aligned} \hat{f}^1_{l_c}(Z) \quad = \quad \bigcup \{ \{\delta'\} \times \phi^1_{l_c}(d) \mid \ & (\delta, d) \in Z \wedge \\ & \delta' = \{ d'' \mid (\delta, d'') \in Z \} \quad \} \end{aligned}$$

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

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

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

Small assumption sets

- throw away even more information.

$$\Delta = D$$

- instead of $2^D \times D$: now only $D \times D$.
- transfer functions simplified
  - call

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

- return

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

Flow-(in-)sensitivity

- "execution order" influences result of the analysis:

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

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

Set of assigned variables

- for procedure p: determine

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

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

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

$$\mathsf{IAV}(p) = (\mathsf{AV}(S) \setminus \{x\}) \cup \bigcup \{\mathsf{IAV}(p') \mid p' \in CP(S)\} \tag{2.24}$$

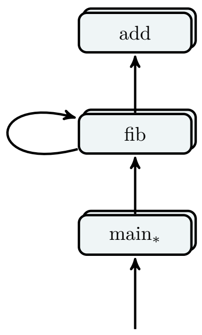
where $\texttt{proc}\, p(\texttt{val}\, x, \texttt{res}\, y)\, \texttt{is}^{l_n}\, S\, \texttt{end}^{l_x} \in D_*$

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

Example

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

Example



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

$\Rightarrow$ smallest solution

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

Chapter 3 Types and effect systems

What is it about?

Learning Targets of this Chapter

type systems
effects
functional languages
type inference and unification

Contents

3.1 Introduction

In this part, we cover various type systems. In the book, the focus is clearly on the "non-standard type systems" parts (covering flow information and effects). The slides reorder the material a bit, in particular we start covering standard type systems and type inference first (in case to have that covered, should the time run out).

In the introduction, we used the while-language to illustrate type and effect systems or annotated type systems in a very simple setting, where the typing part was trivial. In this section, we deal mainly with functional languages, i.e., λ-calculi.

In 2107 and 2018, we only came up-to the standard type inference part.

3.2 Type checking

Syntax

$$\begin{array}{rcll} e & ::= & c \mid x \mid \ \mathtt{fn}_\pi x \Rightarrow e \mid \ \mathtt{fun}_\pi f\ x \Rightarrow e \mid e\ e & \text{terms} \\ & \mid & \mathtt{if}\, e\, \mathtt{then}\, e\, \mathtt{else}\, e \mid \ \mathtt{let}\, x = e\, \mathtt{in}\, e \mid e\, \mathtt{op}\, e & \end{array}$$

Table 3.1: Abstract syntax

π	$\in$	**Pnt**	program points
e	$\in$	**Expr**	expressions
c	$\in$	**Const**	constants
op	$\in$	**Op**	operators
f, x	$\in$	**Var**	variables

The syntax is as a variation of an (untyped) λ-calculus, the prototypical "functional language". Instead of "λ" as symbol for functional abstraction, the language here uses `fn` and `fun`. The language distinguishes syntactivally between the "standard" abstraction `fn` and abstraction for recursive functions. Standard representations of the untyped λ-calculus would not bother to make that distinction: the untyped λ-calculus is Turing complete and therefore expressive enough to encode recursion. Likewise it could encode conditionals etc. However, the lecture uses the more elaborate syntax for two reasons. We are interested in illustrating program analysis, where control flow constructs like conditionals are most likely to be part of the abstract syntax. Even if conditional would be encodable in a more basic syntax, that will obscure the analysis (and make it less "realistic"). Likewise, recursion will pose specific problems, the treatment of which would be obscured if relying on an encoding. Finally, we do make use of *typed* versions of the language, and for the typed λ-calculus, things change: without adding recursion explicitly, the calculus *is not Turing complete* (which make it hard to argue that it's a simple form of a standard functional programming language).

What does *not* belong to the standard calculus are the labels. Those will be needed (as before) for the intended analysis *later* (not for the basic stuff like standard type checking and standart type inference).

It's worthwile to remember the CFA from the introduction, where the functional calculus was labelled, as well. Unlike here, the labelling in the introduction was *for all constructs* whereas here, we label only the two forms abstractions. The difference are motivated by the fact that here we will be interested in a slightly different analysis.

Examples

Example 3.2.1 (Application).

$$(\texttt{fn}_X\, x \Rightarrow x)\ (\texttt{fn}_Y\, y \Rightarrow y)$$

Example 3.2.2.

$$\begin{array}{lll}\texttt{let}\, g & = (\texttt{fun}_F\, f\ x \Rightarrow f(\texttt{fn}_Y\, y \Rightarrow y)) \\ \texttt{in} & \ g\ (\texttt{fn}_Z\, x \Rightarrow x)\end{array}$$

It's worthwile to think about the latter example and especially the role of `fun` (the "recursive" function abstraction). Its

Types

- *Curry*-style typing

$$\begin{array}{cclc}\tau & \in & \textbf{Type} & \text{types} \\ \Gamma & \in & \textbf{TEnv} & \text{type environment}\end{array}$$

Types

$$\tau ::= \mathsf{int} \mid \mathsf{bool} \mid \tau \to \tau$$

- base types:
  - bool and int
  - standard constants and operators assumed $(\mathsf{true}, 5, +, \leq, \ldots)$
  - each constant has a base type τ_c
- **type environments** (finite mappings)

$$\Gamma ::= [] \mid \Gamma, x{:}\tau$$

"Curry-style" means that abstractions like $\mathtt{fn}\, x.e$ *don't* mention a type for the formal parameter. The alternative with a syntax like $\mathtt{fn}\, x{:}\tau.e$ is called "Church style". Both variants exist in real programming languages. Also, leaving it up to a user whether mentioning a type or leaving it out on a case by case basis is possible. Some programmers may prefer to add types for being more explicit and documenting the type of expected arguments and prefer to leave it out, for compactness or convenience.

In Curry-style typing, it's a problem to figure out what the type of an argument actually is (if any). The process of figuring that out is commonly known as *type inference*; some people prefer the word *type reconstruction* for it (and also type synthesis is used here and there). Some other interpretations of the terminology of "type inference" also exists, but they are not so common. In case giving the types is *optional*, the problem is called *partial* type inference (or partial type reconstruction).

The *base* types won't play a prominent role in the development, the calculus simply picks some common ones, where Booleans are needed for the conditionals. There is one restriction for the constants here, namely they are typed by one of the *base types*. That means there are no functional constants. More precisely: the operators can be seen as built-in functional constants, but their arguments are base types, which means, the operators are not *higher-order*. That may well be different in real languages, and is done to slightly simplify the representation here.

As for *type environments* Γ. They play the role of "contexts" (also in the technical sense of dealing with context-sensitive analyses). It is assumed that they work as *finite mappings*, basically as a representation of the *symbol table*. Three fundamental things one can do with such a "table" is: creating an empty one, adding a new binding ("enter"), and looking up an entry ("lookup"). The first two operations are part of the definition from above, for the lookup of x in Γ we write $\Gamma(x)$. We may also write $\Gamma[x \mapsto \tau]$, and use $dom(\Gamma)$. Γ in general acts like a stack.

Judgments and derivation system

Type judgments

$$\Gamma \vdash_{\mathsf{UL}} e : \tau \tag{3.1}$$

- derivation system:
  - Curry-style formulation

  ⇒ *non-deterministic*
  - nonetheless: *monomorphic* let
- type reconstruction/type inference

The remark about *monomorphic let* on the slide is relevent at that point only for the ones who know, that a very famous contribution in type inference is the treatment of the so-called "polymorphic let". Polymorphism and monymorphim relate to the character of the type system for a language. A language is *monomorphic* if each program has (at most) one type. It's *at most* and not *exactly one* as a program may fail to be well-typed. Alternatively can can say, a language is monomorphic, if every well-typed program has *exactly one type*. A language is polymorphic, if it's not the case, i.e., if it's not monomorphic.

There are various forms of polymorphism, according to a classical classification by Cardelli and Wegner [1], there are 4 main variants. Arguably the two most important or most interesting are *parametric polymorphism* and *inclusion polymorphism* (both together called in [1] also as *universal polymorphism*). Inclusion polymorphism is also known as *subtype polymorphism* (as made popular by object-oriented languages).

Let-polymoprhism is of the *parametric* kind, and we might encounter it later. But as said: not right now. Subtype or inclusion polymorphism will probably *not* be covered in this lecture, at least not as far as the underlying type system is concerned. On the other hand: when dealing with the "non-standard" part of the type system (the annotations, the effects etc.) then there will be *"inclusion polymorphism"*. That is related to the "lattice treatment" for instance when dealing with data flow information as in the monotone frameworks (except that it was not formulated making use of type-theoretic notions).

Underlying type system (1) (Curry style)

$$\Gamma \vdash c : \tau_c \quad \textsc{Con} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\ \textsc{Var}$$

$$\frac{\Gamma \vdash e_1 : \tau^1_{\mathtt{op}} \qquad \Gamma \vdash e_2 : \tau^2_{\mathtt{op}}}{\Gamma \vdash e_1\, \mathtt{op}\, e_2 : \tau_{\mathtt{op}}}\ \textsc{Op}$$

Underlying type system (2) (Curry style)

$$\frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash\ \mathtt{fn}_\pi x \Rightarrow e : \tau_1 \to \tau_2}\ \textsc{Fn} \qquad \frac{\Gamma, x{:}\tau_1, f{:}\tau_1 \to \tau_2 \vdash e : \tau_2}{\Gamma \vdash\ \mathtt{fun}_\pi x \Rightarrow e : \tau_1 \to \tau_2}\ \textsc{Fun}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}\ \textsc{App}$$

$$\frac{\Gamma \vdash e_0 : \mathsf{bool} \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash\ \mathtt{if}\, e_0\, \mathtt{then}\, e_1\, \mathtt{else}\, e_2 : \tau}\ \textsc{If}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x{:}\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash\ \mathtt{let}\, x = e_1\, \mathtt{in}\, e_2 : \tau_2}\ \textsc{Let}$$

3.3 Type inference

3.3.1 Type inference problem

That was one of the sections covered 2017 and 2018. Basically, we left out *all* effect or flow parts, and covered *only* the basics here: type inference or reconstruction and the role *unification* plays in that. Therefore the order of the presentation deviates also from the book.

Inference algorithms

- take care of *terminology*
- so far: no *algorithm*! (price of laxness)
- *foresight* needed
- guessing wrong $\Rightarrow$ **backtracking** (and we seriously don't want that)

$\Rightarrow$ required: mechanism to make
  - tentative *guesses*
  - *refine* guesses

- we start first: with the *underlying system*

Next we tackle what is called *type inference*. The terminology of type inference is standard (it goes back to Milner etc.), but not all like the terminology. The reason why it might not the best name is: the type systems we are dealing with are given in the form of *inference systems* i.e. with the help of inference or derivation rules. Therefore the word "type inference" may be confusing to some. Consequently, the word *type reconstruction* is used sometimes.

No matter how it's called, it's here about *algorithms*. The type system so far did not correspond directly to any algorithm for type checking (or flow or effect analysis) . . .

It is useful here to revisit the previous rules of the "Curry-style" type system and reflect on *why* those rules are not directly an algorithm. The basic culprit is the rule for abstraction FN. In the formulation here, where we have also a rule for *recursive* function abstraction, also that rule is not "algorithmic" (rule FUN).

To get the right mental picture, one need to be clear about, in which way the rules at intended to describe an algoriths, at least ultimately. Rules can be interpreted as a "logical" derivation process: given all the premises, the conclusion can be derived. That's ok, but when it comes to the question how to establish a judgment, one better switches perspective: namely: to establish the conclusion, one has to establish the premises first. Nothing really changes, of course, but thinking in this *goal-directed* way ("what do I need to do to establish a goal") may lead to a recursive procedure. Concretely in our setting and our rule, though, we need to explict about what *is* actually the goal? Of course, we have judgements of the form $\Gamma \vdash e : \tau$. With that interpretation, the goal to establish is a boolan question ("yes/no"). Is it the case or not that e has type τ (given a context). For type inference, though, the question is $\Gamma \vdash e : ?$, which means: "is e well-typed, and if so, infer me the type". It's the *latter* way that we think about the judgements.

With this interpretation and concentrating here on FN: the culprit is the fact that the premise of the rule has to **guess** type τ_1. What is the status of τ_1? Well, τ_1 is a symbol we use to represent types (a non-terminal in the grammar for types). It's not a type itself. Another terminology is that it's a **meta-variable** representing types. That τ is a meta-variable for types means, we use that symbol to "speak about" the type system, but it's of course not itself a type. Especially it's not a variable in or of the type system.

The notion of "meta-variable" sounds esotheric, but actually it's not; think of how one would do an implementation. Let's try a recursive check procedure, assuming that the type rules where an inmpementation in the sense of a (non-deterministic) recursive procecure. The conclusion would be a call to a function, say `tcheck` (for do-a-type-check), with input a context, an expression, and a type) and a boolean return. That's a simplifying assumption, as we typically want to have the context Γ and expression e as input, and the type as output, but let's use the simpler "boolean" yes/no problem for illustration. That would mean, the type-check procedure could look like (in some unspecified programming language):

```
tcheck(gamma : Context, exp : Exp, type : Type) =
  if exp = app (exp1 exp2 ) ....
```

Now, `type` is a variable in the unspecified programming language, the formal parameter of the procedure, and is of type `Type`, which, one the one hand, is some concrete type in that said programming language and, on the other hand, is used to implement the types of the language whose type system we intend to implement. At any rate, `type` is a variable, but *not* of the language we are implementing, but of the language we use to implement it. Thus it's a *meta* variable, a variable of the language use to implement the target language (or the language used to "speak about" the target language).

One *core* "enabler" to make type inference work is to *internalize* the notion of "variable" from the meta-level to the level of the language we are dealing with. That means that the concept of types needs to be extended to include *type variables*. That will be called *augmented types*.

Augmented types

fancy name for: "we have added type variables"

$$\begin{array}{lll} \tau & \in & \textbf{AType} \quad \text{augmented types} \\ \alpha & \in & \textbf{TVar} \quad \text{type variables} \end{array}$$

$$\begin{array}{lll} \tau & ::= & \mathsf{int} \mid \mathsf{bool} \mid \tau \to \tau \mid \alpha \\ \alpha & ::= & \mathtt{'a} \mid \mathtt{'b} \mid \ldots \end{array}$$

Substitutions

Substitution (in general) mapping from variables to "terms"

- "syntactic mapping" here:
  - "terms" are (augmented) types
  - variables: type variables

$$\theta : \textbf{TVar} \to_{\mathit{fin}} \textbf{AType}$$

- considered as finite functions: we write $dom(\theta)$.
- **ground substitution**: mapping to *ordinary* types (no variables)
- substitutions: *lifted* to types in the standard manner
- composition of substitutions: $\theta_1 \circ \theta_2$ (or just $\theta_2\theta_1$)

Substitutions will play an important role in the following (and are an important concept in general). They are called "syntactic mapping" above, since terms (here types) are considered syntax (as opposed to values or similar, which is considered "semantics"). So, a state as a mapping from variables to values is considered a semantic thing and different from a substitution. Terms normally contains variables (otherwise there would be no point of substitutions anyhow ...) and terms without variables are generally called *ground terms*. Consequently, a substitution where the result does not contain variables is called a *ground substitution*.

By *lifting*, one simply means; if one knows the effect of a substitution on variables, then it's straightforward to use the substitution *also* as a mapping from terms to terms (here types to types), simply by replacing all the variables inside a term one by one. It's a simple recursive algorithm.

Above we defined substitutions as finite functions, i.e., function with a finite domain of variables. Sometimes, θ's are also considered as *total functions* (over "all" variables), setting $\theta(\alpha) = \alpha$ when $\alpha \notin dom(\theta)$.

Algorithm: basic idea

- instead of guessing type *now* $\Rightarrow$ *postpone* the decision

$\Rightarrow$ use of **type variables**

replace:

$$\frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \ \mathtt{fn}_\pi x \Rightarrow e : \tau_1 \to \tau_2}\ \textsc{Fn}$$

by

$$\frac{\Gamma, x{:}\alpha \vdash e : \tau_2}{\Gamma \vdash \ \mathtt{fn}_\pi x \Rightarrow e : \alpha \to \tau_2}\ \textsc{Fn}$$

- $x{:}\alpha$ when α is fresh (otherwise unused) means: type of x is completely arbitrary.
- syntax-directed now?
- τ_1: meta-variable for concrete types
- α: (still meta variable for) type variables

α's completely arbitrary?

Consider body

$$e = x\ g$$

for $\mathtt{fn}_\pi x \Rightarrow e$

$\Rightarrow$

- a function type: $\alpha = \beta \to \gamma$
- fit together with type of g $\Rightarrow$ condition or constraint on β
- judments "give back" not just the type, but also "restrictions" on type variables.
- represented as constraint[1]
- $\Rightarrow$

$$\Gamma \vdash e : (\tau, C)$$

Under the assumptions Γ (which might "assign" to (program) variables: type variables), program e possesses type τ (again potentially containing type variables) *and* imposes the restrictions "embodied" by C on the type variables.

Constraint generation algorithm The presentention on the slides *deviates* from the presentation in [3] (and previous years). It's a presentational issue, as the rules become (much) more readable. Concentwise, the content is the same. The difference is the following: generating constraints in the typing rules is a kind of *2-phase approach.* In the first phase, generate constraints, and afterwards, in a second phase, try to solve them (in our setting, but unficiation, see below). The alternative, which one often finds in the literature (for instance the original paper) is a *one-phase-approach*: one eagerly not just generates constraints while traversing the abstract syntax tree in the typing rule, one *solves them eagerly at the same time* (here by unification).

It should be noted that the "solve-the-constraint-while-you-go" works fine for the situation here (with unification). For more involved settings, though, it's not just a matter of presentation to do a 2-phase-approach and postpone the solution of the constraints. In more complex situations, the "on-the-fly" solution may no longer works.

[1]In the book, what is given back is a substitution instead.

The intuitive reason for that is as follows: The typing rules travers the syntax tree in a particular manner, collecting information bottom-up. As it turns out, for unification and for collecting information about “the most general type”, that works fine. If the information being collected is more complex, this “bottom-up” treatment of *solving* the constraints may not work anymore (collecting them bottom-up is just fine). An example would be data-flow analysis (take reaching definitions as examples). Solving them involves (as we know) a fixpoint algorithm, due to loops in the program or recursion. Type systems, however, typically don’t go “up-and-down” a synstax tree, it’s a “one-sweep” process. That “sweep” could be used to collect thus flow constraints, and in a second phase, they could be solved (which would involved fixpoint iteration).

Constraints

- generally:
  - constraint(s) is a formula with free variables
  - solving a constraint set: finding values for the variables such that here formula becomes true (satisfiability)

  . set of constraints = interpreted as $\wedge$ (conjuction)
- more precisely here: (term) unification constraints
- notation $\tau_1 =^? \tau_2$
- many other forms of “constraints” systems exists with specialized solving techniques
- here: term *unification*

Constraint generation

$$\frac{}{\Gamma \vdash c : (\tau_c, \emptyset)}\;\textsc{T-Const} \qquad \frac{}{\Gamma \vdash x : (\Gamma(x), id)}\;\textsc{T-Var}$$

$$\frac{\alpha \text{ fresh} \qquad \Gamma, x{:}\alpha \vdash e_0 : (\tau_0, C_0)}{\Gamma \vdash \ \mathtt{fn}_\pi x \Rightarrow e_0 : (\alpha \to \tau_0, C_0)}\;\textsc{T-Fn}$$

$$\frac{\alpha, \alpha_0 \text{ fresh} \quad \Gamma, f{:}\alpha \to \alpha_0, x{:}\alpha \vdash e_0 : (\tau_0, C_0) \quad C_1 = \{\tau_0 =^? \alpha\}}{\Gamma \vdash \mathtt{fun}_\pi f\ x \Rightarrow e_0 : (\alpha \to \tau_0, C_0, C_1)}\;\textsc{T-Fun}$$

$$\frac{\Gamma \vdash e_1 : (\tau_1, C_1) \quad \Gamma \vdash e_2 : (\tau_2, C_2) \quad \alpha \text{ fresh} \qquad C_3 = \{\tau_1 =^? (\tau_2 \to \alpha)\}}{\Gamma \vdash e_1\ e_2 : (\alpha, C_1, C_2, C_3)}\;\textsc{T-App}$$

$$\frac{\Gamma \vdash e_0 : (\tau_0, C_0) \qquad \Gamma \vdash e_1 : (\tau_1, C_1) \qquad \Gamma \vdash e_2 : (\tau_2, C_2) \qquad C_4 = \tau_0 =^? \mathsf{bool} \qquad C_5 = \tau_1 =^? \tau_2}{\Gamma \vdash \mathtt{if}\, e_0\, \mathtt{then}\, e_1\, \mathtt{else}\, e_2 : (\tau_2, C_1, C_2, C_3, C_4, C_5)}\;\textsc{If}$$

$$\frac{\Gamma \vdash e_1 : (\tau_1, C_1) \qquad \Gamma, x{:}\tau_1 \vdash e_2 : (\tau_2, C_2)}{\Gamma \vdash \mathtt{let}\, x = e_1\, \mathtt{in}\, e_2 : (\tau_2, C_1, C_1)}\;\textsc{Let}$$

$$\frac{\Gamma \vdash e_1 : (\tau_1, C_1) \qquad \Gamma \vdash e_2 : (\tau_2, C_2) \qquad C = \{\tau_1 =^? \tau_{\mathsf{op}}^1, \tau_2 =^? \tau_{\mathsf{op}}^2\}}{\Gamma \vdash e_1\, \mathsf{op}\, e_2 : (\tau_{\mathsf{op}}, C_1, C_2, C)}\;\textsc{Op}$$

We are a bit sloppy here with the notation (but the rules are intended to be precise nonetheless). The sloppyness refers to the fact that constraints C are seen as sets of elements of the form $\tau_1 =^? \tau_2$, but we not always write it as set $\{\tau_1 =^? \tau_2\}$. Also we we put two constrain sets together (by building set-union), we simply write C_1, C_2, as opposed to $C_1 \cup C_2$. Instead of seeing it as sets of basic unification constraints, could also see it as a *conjunction* (for instance using $\wedge$), but all of that is just notation. The intention is: a set of constraint is statisfied by a solution, if *all* the constraints inside the constraint set are satisfied, and that makes it a conjuction of basic constraints ("and")

3.3.2 Unification

Unification

- "classical" algorithm ([4])
- many applications (theorem proving, Prolog etc.)
- definition: substitution

Unifier A **unifier** of two types τ_1 and τ_2: a *substitution* θ such that

$$\theta(\tau_1) = \theta(\tau_2)$$

- *unfication problem* given τ_1 and τ_2, determine a *unifier* for them, if it exists

Remarks in other areas: formulas, terms ...

Ordering substitution (and unifiers)

- better formulation of **unfication problem**: given τ_1 and τ_2, determine *the best* = *most general unifier* for them (if they are unifiable).
- solve unification constraint $\tau_1 =^? \tau_2$
- easy generalizable to constraints: $\theta \models C$

Ordering: "less general", "more specific" $\theta_1 \lesssim \theta_2$ if $\theta_1 = \theta\theta_2$ (for some θ)

- most-general-unifier of two types = "the" least upper bound of all unifiers

The ordering of the constraints wrt. their "generality": a substitution θ_2 is more general than another θ_1, if it makes "less definite" choices. That is captured in the above definition.

Unification algorithm for underlying types

$$
\begin{array}{rcl}
\mathcal{U}(\mathsf{int}, \mathsf{int})) & = & \mathit{id} \\
\mathcal{U}(\mathsf{bool}, \mathsf{bool})) & = & \mathit{id} \\
\mathcal{U}(\tau_1 \to \tau_2, \tau_1' \to \tau_2') & = & \begin{array}{ll} \texttt{let} & \theta_1 = \mathcal{U}(\tau_1, \tau_1') \\ & \theta_2 = \mathcal{U}(\theta_1\tau_2, \theta_1\tau_2') \\ \texttt{in} & \theta_2 \circ \theta_1 \end{array} \\
\mathcal{U}(\tau, \alpha) & = & \begin{cases} [\alpha \mapsto \tau] & \text{if } \alpha \text{ does not occur in } \tau \\ & \text{or if } \alpha = \tau \\ \text{fail} & \text{else} \end{cases} \\
\mathcal{U}(\alpha, \tau) & = & \text{symmetrically} \\
\mathcal{U}(\tau_1, \tau_2) & = & \text{fail} \qquad \text{in all other cases}
\end{array}
$$

1-phase Type inference algorithm

- formulated here as *rule system*
- immediate correspondence to a *recursive* function:

$$\mathcal{W}(\Gamma, e) = (\tau, \theta)$$

instead of

$$\Gamma \vdash e : (\tau, \theta)$$

- not 2-phase, giving back a set of unification constraints C

For comparison: one phase approach

$$\frac{}{\Gamma \vdash c : (\tau_c, \mathit{id})}\ \text{T-Const} \qquad \frac{}{\Gamma \vdash x : (\Gamma(x), \mathit{id})}\ \text{T-Var}$$

$$\frac{\alpha \text{ fresh} \qquad \Gamma, x{:}\alpha \vdash e_0 : (\tau_0, \theta_0)}{\Gamma \vdash \ \texttt{fn}_\pi x \Rightarrow e_0 : (\theta_0\alpha \to \tau_0, \theta_0)}\ \text{T-Fn}$$

$$\frac{\alpha, \alpha_0 \text{ fresh} \quad \Gamma, f{:}\alpha \to \alpha_0, x{:}\alpha \vdash e_0 : (\tau_0, \theta_0) \quad \theta_1 = \mathcal{U}(\tau_0, \theta_0\alpha_0)}{\Gamma \vdash \texttt{fun}_\pi f\ x \Rightarrow e_0 : (\theta_1\theta_0\alpha \to \theta_1(\tau_0), \theta_1 \circ \theta_0)}\ \text{T-Fun}$$

$$\frac{\Gamma \vdash e_1 : (\tau_1, \theta_1) \quad \theta_1\Gamma \vdash e_2 : (\tau_2, \theta_2) \quad \alpha \text{ fresh} \quad \theta_3 = \mathcal{U}(\theta_2\tau_1, \tau_2 \to \alpha)}{\Gamma \vdash e_1\ e_2 : (\theta_3\alpha, \theta_3\theta_2\theta_1)}\ \text{T-App}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_0 : (\tau_0, \theta_0) \qquad \theta_0\Gamma \vdash e_1 : (\tau_1, \theta_1) \qquad \theta_1\theta_0\Gamma \vdash e_2 : (\tau_2, \theta_2) \\ \theta_3 = \mathcal{U}(\theta_2\theta_0\tau_0, \mathsf{bool}) \qquad \theta_4 = \mathcal{U}(\theta_3\tau_2, \theta_3\theta_2\tau_1)\end{array}}{\Gamma \vdash \texttt{if}\, e_0\, \texttt{then}\, e_1\, \texttt{else}\, e_2 : (\theta_4\theta_3\tau_2, \theta_4\theta_3\theta_2\theta_1\theta_0)}\ \text{If}$$

$$\frac{\Gamma \vdash e_1 : (\tau_1, \theta_1) \qquad \theta_1\Gamma, x{:}\tau_1 \vdash e_2 : (\tau_2, \theta_2)}{\Gamma \vdash \texttt{let}\, x = e_1\, \texttt{in}\, e_2 : (\tau_2, \theta_2\theta_1)}\ \text{Let}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : (\tau_1, \theta_1) \qquad \theta_2\Gamma \vdash e_2 : (\tau_2, \theta_2) \\ \theta_3 = \mathcal{U}(\theta_2\tau_1, \tau_{\mathsf{op}}^1) \qquad \theta_3 = \mathcal{U}(\theta_3\tau_2, \tau_{\mathsf{op}}^2)\end{array}}{\Gamma \vdash e_1\, \mathsf{op}\, e_2 : (\tau_{\mathsf{op}}, \theta_4\theta_3\theta_2\theta_1)}\ \text{Op}$$

Remarks Note again: we do not have the *full* famous Damas Milner type system with let-polymorphism. To do that we would need to add type schemes (see later).

“Classic” type inference

- we did **not** look at the *full* well-known Hindley-Damas-Milner type inference algorithm
- missing here: **polymorphic let**
- monomoprhic let: “almost useless” polymorphism
- Note the fine line
  - polymorphic let: yes
  - polymorphic functions as function arguments: **no!**

the classical type “inference” algo

- higher-order functions,
- polymorphic functions,
- but *no “higher-order polymorphic functions”*

- dropping the last restriction: type inference *undecidable*
- no type variables in the underlying type system (the “specification”), the type inference algo does
- types (with variables) and *type schemes* $\forall\alpha.\tau$

Bibliography

[1] Cardelli, L. and Wegner, P. (1985). On understanding types, data abstraction and polymorphism. *Computing Surveys*, 17(4):471–522.

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

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

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

[4] Robinson, J. A. (1965). A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41.

Index