

Course Script

Static analysis and all that

IN5440 / autumn 2022

Martin Steffen

Contents

1 Introduction **1**
1.1 Motivation . . . 2
1.1.1 About this document . . . 2
1.1.2 About this lecture . . . 2
1.1.3 About this chapter . . . 2
1.1.4 Connection to the compiler lecture . . . 2
1.1.5 What's static (program) analysis? . . . 4
1.1.6 General remarks . . . 6
1.2 Data-flow analysis . . . 10
1.2.1 A simplistic while-language . . . 12
1.2.2 Equational approach . . . 25
1.2.3 Constraint-based approach . . . 32
1.3 Control-flow analysis (constraint-based) . . . 34
1.4 Type and effect systems . . . 41
1.4.1 Introduction . . . 41
1.4.2 Annotated type systems . . . 46
1.4.3 Annotated type constructors . . . 50
1.4.4 Effect systems . . . 51
1.5 Algorithms . . . 54
1.6 Conclusion . . . 56

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 here just as a teaser and in less depth compared to later but there is already technical content.

Contents

1.1 Motivation 2
1.1.1 About this document 2
1.1.2 About this lecture . . 2
1.1.3 About this chapter . . 2
1.1.4 Connection to the compiler lecture . . . 2
1.1.5 What's static (program) analysis? 4
1.1.6 General remarks . . . 6
1.2 Data-flow analysis 10
1.2.1 A simplistic while-language 12
1.2.2 Equational approach . 25
1.2.3 Constraint-based approach 32
1.3 Control-flow analysis (constraint-based) 34
1.4 Type and effect systems . . . 41
1.4.1 Introduction 41
1.4.2 Annotated type systems 46
1.4.3 Annotated type constructors 50
1.4.4 Effect systems 51
1.5 Algorithms 54
1.6 Conclusion 56

1.1 Motivation

1.1.1 About this document

This is the script version of the slides. It contains additional material, explanations etc., often things I say in class. In case it will again be an online semester, instead of replaying a video or podcast, one may (also) read this script for information. Especially, for the more technical material, one may profit from a written explanation for studying, which one can peruse at one's own pace.

The document is also a *working* document, I.e., it will undergo changes during the semester, cosmetic ones for sure, rearrangements, but perhaps also extensions.

1.1.2 About this lecture

The topics in this lecture are rather standard. The presentation and notations follow often the well-known textbook Nielson et al. [5] (we cannot cover the whole book, though). It focuses on the *principles* of analyses techniques, including a certain amount of mathematical foundations.

Recently, in preparing this semester's version, I stumbled also over the course material or book Møller and Schwartzbach [4], which is also freely available on the net. I will probably also take inpiration there. I have not worked through it yet myself, but since the authors had good stuff in the past, including good textbooks or material, I expect it to be valuable and readable.

1.1.3 About this chapter

The first chapter is partly for motivation and also some cursory preview of things to come, i.e., touching upon issues that we will later dive into in more depth. Thus, the issues here more "impressionistic" as later ones but the sections of the introductory chapters correspond to some extent to larger chapters later.

1.1.4 Connection to the compiler lecture

Static analysis (sometimes also called semantic analysis) is one (complex) phase in most compilers. Often, compilers use many different individual static analyses, so the "static analysis phase" is in fact not one big monolithical phase. Anyway, the compiler lecture at least scratched on the topic of static analysis, mostly in the form of a type checker of a rather simple type system.

However, the lecture here does *not* require having followed the *INF5110* compiler course (or any other such course). In this course, focusing on static analysis, we never touch on lexing and parsing. We basically start under the assumption that parsing is done and we ignore issues of *concrete* syntax (i.e., how concretely a program is written). Instead, we use something called *abstract syntax* as *starting* point, like:

> "let's not talk about it too much, it's easy, and we assume, if someone wants to implement the stuff, we trust that this part (lexing and parsing) is straightforward, and even if not, it's not our concern. For us programs are given in abstract syntax and other intermediate reprentations on top of that.

That approach is common when covering static analysis: focus on the subject matter. It corresponds also in the standard modular architecture of compilers: the type checker or any other analysis phases could not care less how the parser pulled it off to come up with an abstract syntax and which techniques were used: bottom-up, top-down, is a look-ahead neeed, it's all unimportant, important is the abstract syntax tree and its intended semantics.

In the compiler lecture, in particular the oblig part, abstract syntax is a *data structure* which is the result of the lexical and syntactic phases of a compiler (the lexer and parser). We will encounter *abstract syntax* not concretely as represented in an implementation (like a bunch of classes and interfaces, or as inductive data types). Instead, we will write it mathematically with a standard notation (which not incidentally actually corresponds programmatically to inductive data types...). At any rate, knowledge about these issues which one may or may not have from following the compiler lecture is not needed, we start fresh.

One topic covered to some extent in the compiler lecture was *attribute grammars*. Those can be understood as some formalism for static or semantic analysis, basically operating on trees. This lecture here will *not* cover attribute grammars. It's a well-known and standard formalism, but both a bit too general and, perhaps for that reason, a bit out-of-fashion. Too general means, one can try to describe type systems by attribute grammars or "explain" symbol-tables via attribute grammars, but one is probably better off working with more specific approaches when dealing with specific questions like type systems or symbol tables or other aspects. For the same reason, one would not get much mileage out of working with the even more general concept of context-sensitive grammars, which would cover basially all of static analysis . "Better off" on a theoretic level insofar working with specific frameworks will give more *insight* into the topic at hand (like type systems etc), but also on a practical level. Being very general, basing for instance some type checker on a general solver for attribute grammars may well not result in an efficient checker. Finally, even if on the one hand attribute grammars are very general and may capture important aspects of, for example type systems or other specific static analyses, they are, on the other hand, perhaps to weak to elegantly capture other aspects of the targeted analysis and not all analyses will work on trees (notably data flow analysis works on graphs). For all of those reasons, we won't deal with attribute grammars.

There is a *small overlap* with the compiler lecture concerning *data-flow analysis*. This topic will take quite some time in the lecture here. In the compiler lecture, we have covered a small special case of data-flow analysis, namely in the form of *live variable analysis*. The difference to this lecture is, that here we look deeper at data-flow analyses (including live-variable analysis), including *principles* of data-flow analyses and comon foundations shared by a large class of different analyses. Also, the compiler lecture put emphasis on the base line of data-flow analysis, the most simplest setting namely for so-called basic blocks. Those are chunks of straight-line code which form the nodes of the important concept of control-flow graph, little was done on analysing whole control-flow

graphs. Those more global and challenging aspects are what we focus on here, including data-flow *across* procedure calls.

1.1.5 What's static (program) analysis?

If one takes the word "static" literally, one may say, static analysis is every form of effort on a program code before it's actually run. That's super-broad and would basically include everything a compiler does plus some more. That broad view would, for instance, include things like "code reviews" to static analysis. As broad as that literal interpretation of static analysis is, still it mostly does not include other aspect of program analysis and development. For instance, the development process itself, like coming from a problem specification or market-analysis and customer expectations, to use case scenarios etc. until one has as product in the form of a "program". Other important big field of program analyses which does not count as static program analysis is *testing*, which is conceptually done for running programs. The same applies to monitoring and run-time verification.

The static program analysis works on programs before they are run. Typically, the analysis is not done the program code itself, but it works on appropriate representations (like abstract syntax, control-flow graphs, other form of intermediate representations).

Furthermore, at least the way we use the term, static analysis is *algorithmic*, i.e., automatic, typically as part of a compiler. That rules out things like "code review" as static analysis technique, though some people count that in (but it's not covered in the lecture). Also, it excludes *program verification* at least to the extent that it involves the user, for instance in the form of interactive theorem proving. Though automatic theorem proving techniques, constraint solving, etc. do have a role in static analysis, and some may argue, even if the user is involved in establishing facts about a program, also that is static analysis (like code review, though more formalistic), most people would also rule that out.

For the sake of good order: in the above attempt to draw lines what is normally understood as static analysis and what not, we made statements like "the development process is not part of static analysis" (as coming earlier) or "testing is not static analysis" (as coming later, at run-time). That does not mean, that static analysis techniques cannot be used in connection with software development and testing. Actually, such techniques are used in connection with those activities. Before coming to a point in the development process working with actually programming language code, one often works with other "artifacts", maybe more or less abstract models or specifications of some sort or other. To be computer-processable, those are also written in specific notations or languages (and not just white-board snaps or strategy documents), maybe not called programming languages, but formal notations nonetheless. Also those notations need to be "parsed" and they may well be analysed automatically.

Even if, in this discussion, we narrowed down a bit the notion of static analysis, it's still broad and seems to cover everything a compiler does. That may be a defendable interpretation of "automatic static program analysis", but static analysis is more narrow for most: it excludes lexical and syntactic analyses. Though undeniably automatic and static and involving interesting techniques, they don't count (being too "simple", though it's not meant to say, that parsing and syntactical analysis cannot pose complex algorithmic

challenges). But static analysis is concerned with analysing *semantical* properties, and syntactic and lexical properties don't count among those. That's why static analysis is also called *semantic analysis*.

Testing is conceptually clearly separated from static analysis, though one may use static analysis to improve the quality of testing. Indeed, most kinds of tests should be done in an automated way. Just having some poor soul (the testing "engineer"?) to take a number of swings at a program, play around a bit until the release date, to see if something seems amiss, is not even called testing. So, the activity (for many kinds of tests) is either automatic, systematic etc, or it's not even considered testing... One useful approach to achieve a good degree of automatation would be to do *test generation*, often for achieving some criteria. Indeed, certain coverage criteria are closely connected to data-flow analysis and techniques for test generation would apply data-flow techniques or symbolic execution, both central for static analysis.

As shortly mentioned before: A conceptually (though very high-level) way of positioning static analysis is to connect it to the well-known hierarchy of "languages" by Chomsky and Schützenberger (see one of the introductory slides): static analysis is concerned with what is called *context-sensitive* languages or context-sensitive grammars, but not bothering to deal with regular languages and context-free languages (those are for lexers and parsers). Static analysis typically does not venture into the most general class of languages (Turing-complete languages or unrestricted grammars), simply because one prefers the compiler to terminate (often one even prefers efficient termination, not just termination with some immense complexity). When saying that "typically" one prefers decidabiliy, one should be aware, that there are, for instance, known examples of undecidable type system (sometimes by accident, sometimes not), and these include everyday languages like C^{++}(actually, already parsing C^{++}is undecidable...)

Static analysis is a rather broad field, there are basically no limits on what one could try to analyze about a program, and besides that there are new languages, new frameworks, platforms etc coming out all the time, which may pose new challenges or perhaps new angles to old challenges.

A concrete compiler, even an ambitious one, will of course not do *all* analyses and optimizations known on earth, but it could actually be quite many. Often, a number of them are optional; it's up to the user to decide, which ones to enable and/or which variant or "agressiveness-level" of an analysis or optimization is wished. As an indication, the list of compiler options (not the description of the options, just the compact list) of the gcc compiler scrolls down a quite some pages, and it's not even one new line per option (see the online doc here).

But anyway, a compiler will make some selections and compromises wrt. which analyses it supports and in which form, and which are left out. Program analysis is done because the compiler writer expects some "benefit" (for the users of the compiler), like more robubst programs by finding errors early, or resulting in more efficient code etc. But of course, it comes at a price: one can invest unlimited resources in program analysis, but ultimatly the costs will overshadow the benefits (as always there is the universal law of diminishing marginal utility). Likewise the lecture: we cannot cover all kinds of analyses, we have to make a selection as well. We cover basic techniques like plain data-flow analysis and basics of type systems, but also venture here and there to deeper territory.

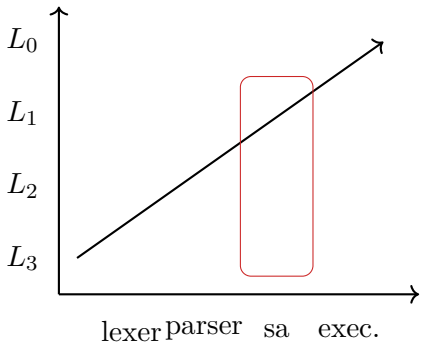


Figure 1.1: Chomsky's hierarchy

1.1.6 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 ...

Static analysis is a very old and well-established field. In the context.

success-story for formal methods

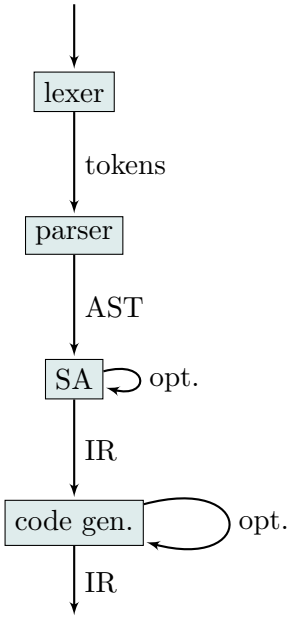
The nature of static analysis

- compiler with differerent *phases*
- corresponding to *Chomsky's hierarchy*
- **static** = in principle: before run-time
- since: run-time most often: undecidable

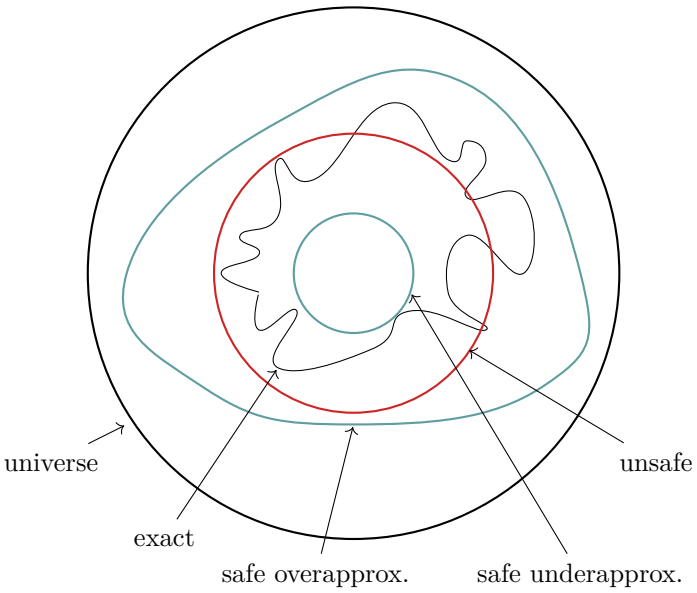
$\Rightarrow$ static analysis as **approximation**

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.

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 (known as "Rice's

[1] source code, intermediate code at various levels

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. Syntactical properties are those referring how the programm is *written*. They include trivialities like "the program has more than 100 lines" or more generally "number of lines of code". Others are the "keywords", or also whether the program adheres to a given context-free grammar, i.e., whether the text is syntactically correct or not.

Back to the picture: the actual, exact program behavior, its semantics, is undecidable. This is indicated by the irregular and complex "area" representing all possible behaviors. This fundamental restriction of being undecidable also holds for *all* nontrivial semantic properties one might be interested in. So, how do we deal with the sad fact that there is no hope whatsoever, to analyze *algorithmically* any semantic property for all behaviors for all programs in an exact manner. Again: no semantic property at all can be analyzed (except "true" or "false"). What can one do? If one cannot have it all, one has to compromize and give up on some items on the wishlist.

One thing that one cannot give up typically is, to say: I consider a programming language which is not Turing complete, then its behavior becomes analyzable. That's typically not an option.

We could insist on exact information and automatation, just not for all behaviors. That can be done by *testing* which runs each programs for a finite amount of test cases (and perhaps put a limit on how long each program test is run). Then one obtains terminating and exact information, though only about a fraction of a program's behavior. Actually, a super-tiny fraction of its behavior, typically. One can invest in selecting that tiny fraction in an intelligent way, so one can use heuristics that the test cases focus on corner cases, or cover each class of input, or covers each line of code, etc. Performing tests in a smart and organized way (with perhaps the help of semantic analysis which may come from static analysis) is very helpful in practice, but still, it is only a tiny fraction.

Static analysis sacrifices precision, but insists on *automation* and on being able to cover *all* behaviors of all programs. We want an answer, i.e., we insist on termination. One might say, non-termination, which is in a way the absence of an answer, is also some form of answer, but mostly that's not what static analysis does, one is not happy with semi-decision procedures. Being not exact or approximative could mean, in the simplest case, that the analysis sometimes errs: when the type checker says, the program is type correct, still some type-related errors may occur, and conversely, when spotting a type-related issue a compile time, the program actually may be free of such problems at run-time. One speaks sometimes of false-positives and and false-negatives. This "don't rely on the outcome either way" approximation is shown in the picture by the red circle (unsafe). Some analyses do it like that, and one can defend it: an analysis may have value for the user, if the correct answers outweigh the wrong ones, and if the consequences of making errors are not show-stoppers.

In this lecture, we *don't* do such hit-or-miss analyses, we insist on safe analyses. In a compiler, one typically would indeed not knowingly go down the "who cares"-path of

saying: often the produced code should be fine, often the analsyses will a correct answer, often the optimization will not mess up the progrem, but sometimes it goes all wrong. Something that provides results that are sometimes reliable and sometimes not (without being clear when is which) is simply not reliable at all and that is unacceptable in many compiler-related contexts. Of course, an analysis designed to be *safe* may still contain bugs when implemented and thereby produce wrong results, but that's a different from being intentionally unsafe as a matter of design.

By safe approximation, we mean, there may be false-positives or else false-negatives, but not both. It's either a safe underapproximation or safe overapproximation. Both are mirror-images of each other. Whether one goes for an over- or an under-approximation depends on the property one plans to investigate and/or also on the optimization or kinds of actions taken based in the information. For instance, when using live-variable analysis with the purpose of register allocation, the perspective is as follows: a register containing a value of a variable analyzed as non-live (= no longer used in the future) may as well be considered free and can be used to hold the value of another variable. With this perspective, judging a variable as live, i.e., potentially used later, when in fact it is not, may waste register space, but otherwise not change program behavior (apart from protentially degrading the performance). The converse error, judging a variable non-live (and thus reusing the register otherwise and throwing away the value), leads to erroneous behavior. That's unsafe. So, the intention of the live-variable analysis for register allocation in the way described dictates that one has to do over-approximation: judging statically in more situations a variable as live than actually is the case at runtime, that's ok. In this case one has at least half-way reliable information. If the analysis statically says, the variable is dead, then it is guaranteed to be so at run-time. If the analysis judges a variable as live at some point, this may or may not be the case.

Over- and under-approximations are, in a way, mirror images. Actually, if one would do "dead-variable analysis" instead of live-variable analysis with the same intention of register management, the overapproximation for live variables turns into an underapproximation for dead variables. But otherwise, it's exactly the same. What matters is the correct register mangement, not the way one views this particular analysis problem. I suspect, one always speaks of live variable analysis out of tradition, and perhaps because saying that variables are dead sounds less nice. Not for all analysis, it's *practical* to flip the perspective like that. For live resp. dead variable analysis for a finite amount of variables, it would perhaps simply amount to a bit-flip in a bit-array (in one plausible implementation). For other analyses, one way may be actually preferable or even the only possible way. For example, if one need to keep track of identifiers or similar that are being used, one can keep a list of the ones in use, but keeping the ones *not* in use does not work because the list would be "infinite", conceptually at least. If one could have an infinite bit-array of all possible identifiers each slot indicating if in use or not, then it would be the same symmetric setting as for the finite amount of variables in the live-vs-dead setting: Logically, it's still symmetric, but practically it is not.

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 the *non-existence* of optimal compiler or *undecidability* of such a compiler
- theorem
  - it just *guarantees* that there is room for improvement
  - does not say *how!* Actually finding a better one: *undecidable*

The proof is actually pretty simple. See Appel [1]. But let's not go into proving it in the lecture, let's leave it at that.

1.2 Data-flow analysis

About the importance of data-flow analysis

Data-flow analysis is one well-established, very standard, general form of program analysis. It is being routinely used for many purposes in compilers. It can take different forms (and can be done for different purposes), but all those are based on common notions and principles. We will look at those principles, here first more cursory, later in more depth.

It is a static analysis technique par excellence, in many aspects. It's widely used and widely applicable with different goals, and it's based on well-understood principles. It's pretty ancient, as well. Indeed, a form of data-flow analysis was done in one of the very first compilers overall, for Fortran. The Fortran compiler was not the absolutely first compiler, but it has been dubbed the first complete and commercially available compiler. Actually, it was not even called a compiler in the paper Backus et al. [2] from 1957, giving a short overview over the status of the FORTRAN project and the resulting compiler. In the title, it's called "automatic coding system" and the text talks about the *translator*, the word "compiler" does not show up, though it's mentioned that the Fortran translator *compiles* Fortran programs to IBM704 programs. Anyway, the document speaks about *flow analysis* (with the purpose of register allocation). This is done by "section 4 and 5" of that compiler (the phases of the compiler were called "sections"). The compiler also worked with the notion of *basic blocks* and *control-flow graphs* (though the latter word is not used). We will cover those concepts in this lecture, as well.

Another point why one should know about data-flow analysis is that it illustrates in a straightforward setting a number of general concepts or mind-sets underlying basically all static analyses. In particular a clear notion of working on *approximations* (or abstractions), central to basically all static analyses. Technically one can understand data-flow analysis to work with *constraints*, another central concept. For basic data-flow analysis, the involved constraints are likewise "basic". They are known as data-flow constraints and data-flow analysis is one simple example of *constraint solving*. If one sticks to basic settings (as we do for the beginning, especially in this warm-up section), data-flow analysis is typically manageable, complexity-wise. And, as mentioned, it was always around in all compilers, ever since basically the first one for Fortran.

About abstract syntax

This warm-up section will also give a first glimpse if what we will repeatedly see in the following. If one reads other literature, like publications explaining new forms of analyses, the way things are presented there often also resemble the things here. Static analysis normally does not work directly on programs texts (in the form of a string of characters in concrete syntax), but on more reasonable, structured representations of the program. One such intermediate representation is known as the **abstract syntax** of a programming language, and one particular program is represented by one concrete element of the abstract syntax, which is a tree, namely the **abstract syntax tree** of the given program. In a concrete implementation of a compiler, it's an important data structure as output from the parser, perhaps implemented using inductive data types (when working with a functional language), or with interfaces and sub-classes in a language like Java (and there are other ways how one can implement such trees).

How will *we* talk about abstract syntax? We will use something called *context-free grammars* for that, more specifically, a *notation* for that kind of grammars known as BNF resp. EBNF ((extended) Backus-Naur form). Also this concept originated from the Fortran language, at least their use for programming languages (the formal notion of grammars originated in linguistics). The Fortran folks did indeed some truly pioneering stuff. We will see BNFs describing abstract syntax variously in this lecture, starting in the next Section 1.2.1 with the abstract syntax of a small while-language.

Now, if one happens to know a bit about the parsing phase of a compiler, one knows, that parsing is obsessed with context-free grammars: What kind of grammar one can use for what kind of parsing machinery, how much look-ahead is supported, is the grammar left-recursive, ambiguous, normalized, and what not. We, in contrast, are not obsessed and we are not doing parsing, which is done for (a token stream of) *concrete syntax*. So we don't care about those issues. The usage we make of grammars in BNF is to describe the structure of trees, i.e., a means to concisely *communicate* the (abstract) syntax of the language at hand (for instance, the "while-language" coming up next). Therefore, we will not formally introduce grammars, BNFs etc, we just present the notation relying on that it's not only precise (which it is) but also clear enough, which may of course depend on the reader or listener. But after seeing different examples throughout the course, one sure just learns by "getting used to it".

Also typical for many textbook or similar presentations is, that one often chooses to work with a "simplified" version of the abstract syntax. The simplification refers to different aspects. One is that the abstract syntax is presented mathematically and compactly, omitting things that are included in abstract syntax tree data structures in concrete implementation (which may include line numbers, types and what not). There is another, more relevant way the abstract syntax is "simplified". We *focus* on aspects we are interested in. For example, the while-language is unrealistically simple for modern languages. There is no scoping, no procedures, no pointers, etc. That's partly pedagogical: one learns the basics one some cut-down focused core language, knowing that in a real language, there are other things to be dealt with as well. Actually, later in the lecture we may deal with some of those (like adding procedures, pointers, etc., thereby extending the abstract syntax more and more). However, still we won't cover the abstract syntax of a full realistic language.

But a simplified and lean abstract syntax is not only "pedagogical". Properly designed, it should consist of a carefully chosen selection of core concepts or features, each of which might only be represented once syntactically. Note that we are talking about the abstract syntax of the core language itself. On top of that language, many fancy things can be supported in libraries, which may indeed be massive. Also, in case the designer chooses to support different syntactic ways to express similar things in the language itself, there is often no need to clutter the *abstract* syntax with this. For illustration, if we had a simple imperative language (like the while language), the feature is looping and that loops can be nested. For the user, i.e., in concrete syntax, the language may support different ways of expressing that concept, like while-loops and repeat-until-loops and other variants. However, in the abstract syntax, it may be a good idea that just one form of loop is represented (let's say corresponding to while-loops). "Syntactic" variations of that sort, that are "compiled away" early on, before the real work starts, are also know as *syntactic sugar* and the process of getting rid of those and working with a "simplified" or cut-down version of the user syntax is called *desugaring*.

A last word about abstract syntax. We talked about it in a way that seem to indicate that abstract syntax captures the syntactical essence of the language the user sees (in the form of concrete syntax). Fair enough, but as stressed, BNF is a way to describe trees ("syntax" tree). Therefore, abstract syntax and corresponding abstract syntax trees may well apply to the syntax of various forms of intermediate code, even if the intermediate code is totally internal and there is no "concrete syntax" attached to it.

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.

As said, 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 concrete 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 grouping 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. As mentioned, later we might extend it also with further capabilities 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", i.e., *within* the grammar. *Arithmetic* expressions are represented by the non-terminal 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. Since we have only two types (and no polymorphism or other complications), we can bake the very rudimentary type directly into the grammar. 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.

Labeling

- associate *flow* information

$\Rightarrow$ **labels**

- *elementary block* = labeled item
- identify basic building blocks
- consistent/unique labeling

Abstract syntax

There will be two versions of the abstract syntax, one the "conventional" one. One can say, that's *the* abstract syntax of the while language. The other one is an augmented version, decorated with additional information. But let's start with the standard version first in Table 1.1.

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	::=	$x := a \mid \mathsf{skip} \mid S; S$	statements
	$\mid$	$\mathtt{if}\ b\ \mathtt{then}\ S\ \mathtt{else}\ S \mid \mathtt{while}\ b\ \mathtt{do}\ S$	

Table 1.1: Abstract syntax

The abstract syntax is written in BNF notation, using the "meta"-symbols ::= and |. The ::= is "definitorial", defining the entity of the left by the construction(s) on the right. The | reads "or" i.e., represents alternatives. The last column is just added for human readability, as a kind of commentary, in that sense it does not belong to the mathematical content of the grammar. For instance, the symbol S, which is supposed to represent "statements" can be given or *constructed* in exactly 5 different ways; the different cases, separated by | stretch out over 2 lines. The first two one could call basic statements, they represent assignments and an "empty" statement called skip. The last 3 cases construct composed statements from simpler ones; what they intend to represent should be self-explanatory.

As for notational or typographic conventions we try to follow (though not slavishly). The "constructors" of the abstract syntax, we use some specific `font`, like for `if`, `then`, etc. They can be seen as "keywords" of the language *terminals* of the grammars. That's perhaps a helpful way to see it, though the notion keywords (and tokens) belong rather to the use of grammars for parsing and thus for *concrete* syntax. We are using the BNF as a grammar to describe abstract syntax trees. So the `if _then _else` is a construction that can build a statement "node" when given 3 arguments, a boolean expression (represented by b) and two statements, both of type S. It's one single constructor, not three individual "keywords". We might have chosen to write it `ITE` b S S to stress it's conceptually better seen as one three-argument constructor. The letters (or sometimes strings) written in *italics* like S, x, etc. are in the terminology of grammars *non-terminals*.

For concreteness sake, Listing 1.1 shows how one could represent a date type for statements in a functional language. The implementation is pretty faithful, but "deviates" in two aspects. First, in contains positioning information; the `pos` data types would be something like containing line numbers and/or start- and end character position in the source code. That kind of information is just useful to have in a concrete implementation, for error localization. The other deviation is, that the S non-terminal is split into two data types, `stm` and `astmt`, the latter representing *atomic* statements. Why I choose to do it like that, I can't remember. In the implementation, the non-terminals are inductive data-type *constructors*, the non-terminals are the data types. Being an inductive data type of that form (with arguments) allows to capture the *recursive* nature of the grammar.

```
    astmt =
   | Assign of var * aexpr * pos
   | Callstmt of var * var * var
   | Skip
type stmt =
  | AStmt  of astmt
  | Seq    of (stmt list)  * pos
  | If     of  bexpr * stmt * stmt * pos
  | While  of  bexpr * stmt *        pos
```

Listing 1.1: Abstract syntax in ocaml

Adding "labels" The second version of the abstract syntax from Table 1.2 contains additional information, called labels l (written as superscript and with some [bracket] notation, which has no purpose except for clarity or readability). We don't explain the role of labels and how they can be "implemented" here much, since we are in the overview, and we pick up on the syntax, the labeling and other issues in a later chapter. The labels can be understood as being able to identify different points in a program.

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	::=	$[x := a]^l \mid [\mathsf{skip}]^l \mid S; S$	statements
	$\mid$	$\mathtt{if}\ [b]^l\ \mathtt{then}\ S\ \mathtt{else}\ S \mid \mathtt{while}\ [b]^l\ \mathtt{do}\ S$	

Table 1.2: Labeled abstract syntax

People who went to the compiler course will have encountered the notion of labels, those were basically the same concept. In more low-level languages (including certain intermediate code languages or also assembler) or in a higher-level language with goto's, labels (like here) allow to refer to specific points in the program. That allows to mark a line with a symbolic label like `label_1` or `error-handling` and then do things like `goto errorhandling`. Used this way, labels are symbolic representations of line numbers in the code. It's a slightly better and less brittle way of programming than doing things like `goto 137` where 137 is the line number (some versions of BASIC supported that). That's a perhaps helpful analogy, though again, we are not working with "lines of code", we are working with abstract syntax *trees*, described by the grammar, so there are no lines at all. Still the purpose of the labels is analogous. The labels will not be part of syntax the user can provide, the labeling is added *automatically*. In an actual implementation, one perhaps would use integers and a counter, to attach them to a given abstract syntax tree in a manner without handing out a label twice. In this lecture, we will use labels abstractly, for instance, the labeled version of the factorial from equation (1.2) uses labels l_0, l_1, l_2. It is implicitly assumed that the $l_0, l_1, \ldots$ are all different. But otherwise, labels are treated abstract and it's not intended to imply that l_0 is the integer 0 etc, or even that internally it's an integer, though in a concrete implementation, integers may be a valid representation.

Example: factorial

The following is a very simple example, what it does should be clear, but it's not too important anyway. The example uses most constructs. The skip-construct is missing (which is a boring one) and conditional, but the `while` works actually similar, like a conditional with the addition complication that it leads to a cycle. The notions of input and output variables are not officially part of the abstract syntax, it's more like, in this example, x is seen as the input and z the output.

Afterwards, we see the same program, but being properly labeled. We just use 0, 1, 2, not l_0, l_1, not to overdo it with subscripts or superscripts. The subsequent control flow graph is decorated with l's.

Note that x is not initialized in the program, and not assigned to. That's common for from input variables (and in a more realistic language, which supports those concept explicitly, it may be enforced by a (rather trivial) static analysis. That all non-input variables are properly initialized could be covered by a non-trivial but still basic static analysis, in fact by a standard data-flow analysis like the one were are about to look at, "reaching definition". We will later comment on that again, in the context of that particular analysis.

Example 1.2.1. □

$$y := x; z := 1; \texttt{while}\ y > 1\, \texttt{do}(z := z * y; y := y - 1); y := 0 \tag{1.1}$$

- input variable: x
- output variable: z

$$\begin{array}{l} [y := x]^0; \\ [z := 1]^1; \\ \ \texttt{while}\ [y > 1]^2 \\ \ \texttt{do}([z := z * y]^3; [y := y - 1]^4); \\ \ [y := 0]^5 \end{array} \tag{1.2}$$

CFG factorial

What's "control"? Previously we drew a parallel of the labels in our abstract syntax and labels abstract versions of program lines or indicating jump-targets. Another way of seeing it are marking specific *control point*. What are points of control? In the execution of a programs one can most often make a strict distinction between *data* and *control*. The control refers to the code of the program, the instructions, and the data is what the running program operates on, by reading and writing to it. Both kind of information is kept somewhere "in memory", but often in strictly separated parts of the memory. The separation is more or less enforced, by the operating system together with the hardware. That forbids things like self-modifying code, where the running program would write to the code-part of the memory, the one that contains the "control". Allowing that is typically seen as a bad idea. Data and control are often not 100% separated so people may still

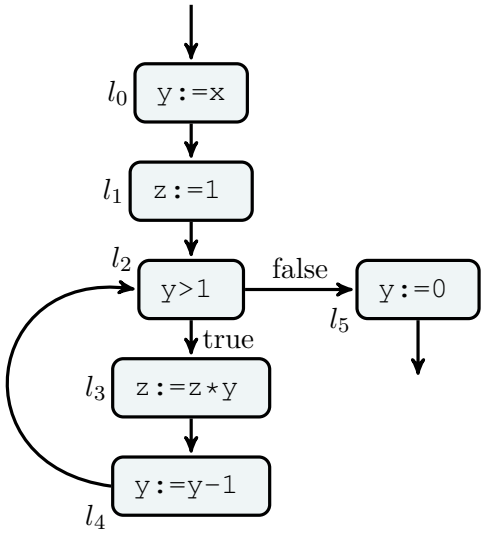


Figure 1.2: CFG factorial

mess things up. For instance, a buffer or stack *overflow* is in a way when data is written in memory at some place where it does not belong, especially at a place meant to store control information, and nit user data. For instance the return address in the call stack. Of course, programs are supposed to contain checks protecting against such occurrences, but not all programs are watertight. Managing to fiddle with the return address is not really using data to modify the *code* of the program, but still it messes with the control part of the program, namely manipulating where the control flow returns. It's a popular way of hijacking systems and subverting security protections.... And static analysis and specific data-flow analyses can check if a program suffers from such weaknesses.

Control flow graph The factorial used to illustrate the important concept of *control flow graphs*. Later in the lecture we will have another look in slightly 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 themselves serve as identification). The "labeled" abstract syntax from above is a different notation for 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 labeled program from equation (1.2) obviously are the *edges* of the graph, but, as said, it's not too hard to calculate them as well (while labeling the statements and traversing the syntax tree).

For participants of the course *compiler construction* (INF5110): the definition of the control flow graph was done in that course at a lower level, i.e., at some *intermediate code* (like three-address-code) without looping constructs, but conditional jumps instead. That

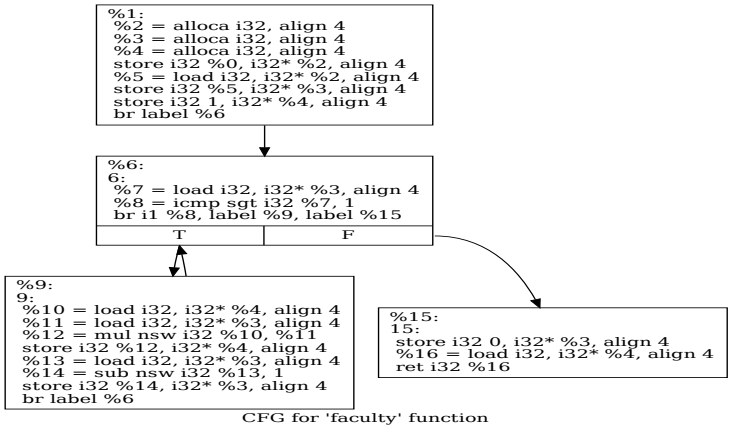
intermediate code can be seen as a kind of "machine-independent machine language", i.e., an intermediate language already rather close to actual machine language, but not yet quite there. In any case, that intermediate language language had *officially* labels, as part of the syntax, that could be used for conditional or unconditional *jumps* (which are like goto's). In other words, that intermediate language was "officially labeled" 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 (and fall-throughs) correspond 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". Unconditional jumps correspond to one single edge. In principle, the notion of control-flow graph there is analogous to the one used here (and elsewhere).

Basic blocks 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 blocks which contains no branching or jumps into it or out of it. Those larger blocks are sometimes called *basic blocks* and the grouping is done for efficiency in an implementation, when running some analysis like reaching definitions on the graph. In the factorial example, l_0 and l_1 could be lumped together (as well as l_3 and l_4). While making larger basic blocks is a good idea in practice for efficiency, we don't care much here, as the principles of data-flow would not change under this refinement.

CFG in "real life" (LLVM)

```
int faculty (int x) {
  int  y,z;
  y = x;
  z = 1;
  while(y > 1) {
    z = z*y;
    y = y-1;
  };
  y=0;
  return z;
};
```

Listing 1.2: Faculty, as C program



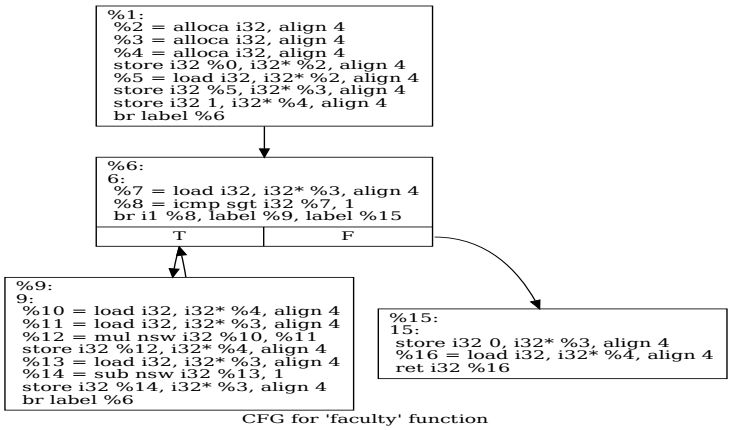
CFG for 'faculty' function

We may also have a glimpse of how the CFG for the faculty looks in reality. For that, I used a C version of the faculty function. The input variable z is now the formal parameter of the procedure, and the output variable is used in the return-statement.

```
int faculty (int x) {
  int  y,z;
  y = x;
  z = 1;
  while(y > 1) {
    z = z*y;
    y = y-1;
  };
  y=0;
  return z;
};
```

Listing 1.3: Faculty, as C program

The CFG is produced with LLVM and `clang` (and rendered with `graphviz`). It's more complicated than our class-room version, though it has lesser nodes: the CFG here uses basic blocks, containing maximal stretches of straight-line code.



CFG for 'faculty' function

When saying, "realistic", on the one hand, that's true, being LLVM and all. On the other hand, one should keep in mind shown is of course not the *actual* CFG data structure, but a graphical representation of it. It is generated with the support of `clang` and `opt` into half-way human-readable textual form (a print-out of the graph, so to say), which in turn is visualized by `graphviz`. But one can see the labels, represented here as numbers and how they are used by the jump-commands ("branch"). More could be said about CFGs in LLVM, including that it makes use of the so-called SSA format, but we can keep that for later).

If one wants to experiment with CFGs oneself, that's the commands I used

```
clang faculty.c  -O0 -S -emit-llvm  -o faculty.o
opt -dot-cfg faculty.o
dot -Tpdf .faculty.dot -o faculty.pdf
```

Reaching definitions

In the following, we look at a first instance of a data-flow analysis. The reaching definitions analysis 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 Kildall [3].

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. This information 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. We will illustrate the analysis with the example program from before, the faculty.

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.

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

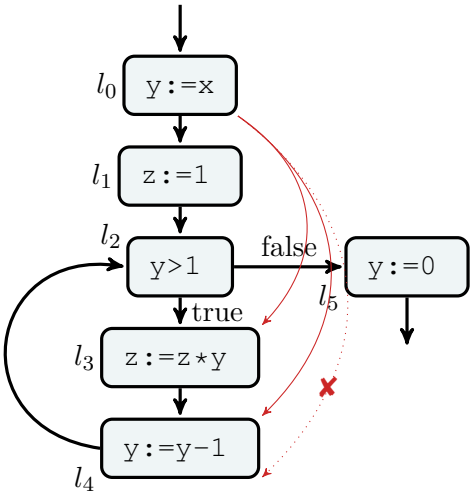


Figure 1.3: Factorial: reaching definitions

Note, the example illustrates, which points are reachable from the choice of origin $(y, 0)$, but, to keep it readable, not all reachable such points are indicated via red arrows. Note also, that by "points" in the program, one does not just say, the node labeled l_x is reachable. It's more fine-grained in that one distinguishes between the entry and the exit of a node. For instance, 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, in the illustration. For the given CFG of the faculty with 6 nodes, that gives altogether 12 "points" of interests. Since there are 6 nodes, there are 6×12 combinations for the question: can the assignment from a node reach the entry or exit of another node. Actually, in the program, l_2 is as starting point is uninteresting insofar it does not contain an assignment. Additionally, for reaching definitions, there are definitions or assignments to the variables, which play a role, which are *not* shown in the graph. Conceptually, it's assumed that all variables have some value "assigned" *prior* to the execution of the program, they have some initial values, though it's unclear what they are. Some languages may guarantee, that variables have specific initial values (0, or a nil-pointer), some don't. The analysis here uses ? as specific label or node for that.

It should be also noted: when 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 search 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"

$$\mathsf{RD} = (\mathsf{RD}_{entry}, \mathsf{RD}_{exit}) \tag{1.3}$$

- 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	$(x,?),(y,0),(y,4),(z,1),(z,3)$	$(x,?),(y,0),(y,4),(z,3)$
4	$(x,?),(y,0),(y,4),(z,3)$	$(x,?),(y,4),(z,3)$
5	$(x,?),(y,0),(y,4),(z,1),(z,3)$	$(x,?),(y,5),(z,1),(z,3)$

The table shows the *result* of the reaching definition analysis. For the of the entries and exits for each node, they show, by which definitions from which nodes they can be reached. Actually, it's not just *one* possible result of the reaching definition, it's the "canonical" one insofar that it is the smallest, the most precise, and thereby the "best" one can (at the given level of abstraction).

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 in the right-hand column, 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 We take up again the notion of input variables. 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 (initialized variables) which may lead to random results or errors like nil-pointer exceptions.

For participants of the course "compiler construction": the situation about initialized variables may be compared to the analogous situation for *liveness analysis*. There, the core analysis concentrated completely on analyzing *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 be completely clear in what sense the solution is minimal/optimal. Much of the underlying theory (covered later) is about assuring that such a minimal solution demonstrably 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 from before (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 "visualized" by allowing over-approximation only in the form of a *circle*, let's say. The optimal solution is then the smallest such covering circle (which still is an approximation). 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 where *optimal* solutions 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

So far we have a impression of the information we are after, at least in the reaching definition case. We have a feeling when the information is correct (i.e. when it's a safe approximation). Missing is: how to obtain that information? The way we proceed is: first show how to *specify* the data-flow problem for a given program, and afterwards show how to *solve* the data-flow problem.

A program gives rise to a data-flow problem in the form of *constraints*. The notion of constraints is a very general one, there are may different forms of constraints, also many in use for static analysis. Consequently there are many different algorithms to tackle specific classes of constraints and may different constraint solvers.

A constraint is nothing else then a formula of some form containing some (free) variables or unknowns. Solving a constraint is nothing else that filling out the unknowns with values in such a way that the formula becomes true. So the concept as such is very familiar, and one will have it encountered it already in early math classes in elementary school, when one has to solve things like $x + 6 = 14$ or what not. Some folks solve constraints for recreational purpose, filling out crossword puzzles or sudokus (the white squares are the unknowns or "variables").

For us: Data-flow problems are straightforwardly cast as a simple form of constraint system. We illustrate that in the following, presenting two variations, which are slightly different ways to do that. The difference is **non-essential** in that both representations give basically the same results and both representations will be solved algorithmically exactly the same way. Still we go through both variants, since the second one will be more flexible. The flexibility is irrelevant for the simple form of constraints right now (reaching definitions), but better suitable for more complex setting (like analyzing functional languages).

The first approach phrases the data-flow problem as a set of *equations*, the *equational* approach. Afterward we generalize the equations by *inequations*, i.e. subset-constraints, using $\subseteq$. That's the only difference.

For precision's sake: we have be careful with our statement claiming that both approaches "give the same results" because literally taken, it's not true that both versions have the same set of solution: inequations allow (naturally) more solutions. As analogy, $x + 4 \leq y$

has more solutions than $x + 4 = y$ (those are not data-flow constraints, but constraints involving equations and inequations nonetheless). But as far as the intended solution is concerned, both variants give the same in the particular mathematical setting of data-flow constraints (more later).

1.2.2 Equational approach

The constraint system takes a particularly simple form. Actually, it's not so much the fact that there are equations, and we don't want to imply, equations in the first approach are simple whereas the "inequations" cover later, using $\subseteq$ instead, are complex. They are equally simple.

What makes the constraint problem simple when compared to other forms of constraints is perhaps two facts. First is the *form* of constraints. Constraints are formulas with variables. In this case, the "formulas" are sets of equations (or inequations later), where the set can be seen as *conjunctions*, insofar that a solution must satisfy all (in-)equations. What makes it simple is, that one side of all questions, the variables are "isolated". As illustration: in an (numerical) equation system like

$$x = y - 16 \tag{1.4}$$
$$y = 4 \tag{1.5}$$

in the first line, x is "isolated" on the left, but not y, and in the second line y is "isolated". For inequations, one would require, that both lines would either be both $\leq$ or both $\geq$, a mixture would be no longer in the simple form characteristic of data-flow equations (and the techniques for solving data-flow equation would no longer apply). This specific restriction on the form of the (in-)equations is only aspect what makes data-flow constraint system simple (actually, in some presentations, the word "simple constraint system" is meant as technical term to denote said syntactic restriction). The other aspect that makes the data-flow setting quite straightforward is not visible in the form of the constraints. It has to do with the **domain** over which the constraints are solved to fill in the unknowns.

Before we go there, let's have a second look at the constraint system from (the non-data-flow) equations (1.4) and (1.5). A plausible domain over which the system is to be solved is integers. Another one could be the natural numbers, in which case there'd be no solution. So, let's say it's integers. In that case, to solve the equation system is not just easy, but super-easy. Everyone (even in elementary school) knows how to proceed. That it is so trivial has to do with the fact that the equation are simple, with variables isolated on the left. For y, the solution is directly given, and that we just have to set into the first equation, do a simple numerical calculation, and that gives the result -12 for x, and we are done.

Now, that was not just simple and easy, but outright trivial. Data-flow constraints are still simple in the way described. However, they involve typically **recursion**. We could change the above equation as follows, making the two equations *mutually recursive*.

$$x = y + 12 \tag{1.6}$$
$$y = 5 + x \tag{1.7}$$

A solution for x depends on a solution for y, and vice versa. Such equations are not typically encountered in elementary school... Adding recursion catapults the constraint system to a whole new level of complication (thought the form is still called "simple" since at least the variables on one side are "isolated").

The particular recursive equation system has *no* solution over integers nor natural numbers (nor in any standard numerical type). I say standard, since if one considered "infinity" a some non-standard number, one could reasonably say $x = \infty$ and $y = \infty$ is a valid solution.

However, data-flow equations are seldom solved over natural numbers or integers and equations would typically not involve plus and minus on numbers. Instead, they are solved over domains that have nicer mathematical properties. What is nice or not, of course, may be a matter of taste, but nice in the sense that the properties of the domain fit *exactly* to the needs that allow the data-flow problem to be solved. We will shortly see that in the example of reaching definitions, where the domain are **sets** of elements containing RD information (the pairs of the form $(x, ?)$ from before), ordered by $\subseteq$. Calculations over such kind of non-numerical domains are consequently also not plus or minus or things like that, but set union and set intersection. Later we discuss more in detail what properties such domains need to have (they are known as complete lattices) to guarantee that simple recursive equations like the one before do have solutions. Actually, not only that, working with such solution domains guarantees **unique best solutions** and that they can be **efficiently** calculated, at least in the sense that no backtracking is required.

We will cover the necessary background later. For now: the reaching definitions (and other data-flow questions) are solved over a domain which is a complete lattice. In a way, that's what basic data-flow analysis is all about:

> *data-flow analysis = solving simple (but typically recursive) equations over a lattice.*

As said, sets (ordered by $\subseteq$) are a lattice, the natural numbers (ordered by $\leq$) are *not* a lattice.

By the fact alone, that equations (1.6) and (1.7) have no solution in $\mathbb{N}$ shows, that it can't be a lattice, because being a lattice would *guarantee* that a solution exist. Well, to be really precise: to really guarantee that the system has a (best) solution, we have also take into account that $+$ is a *monotone* function (which it is), otherwise, a solution may exist or may not.

Indeed, if we would use $\mathbb{N}^\infty$ instead of the standard natural numbers, with and additional value ∞ representing an element $\geq$ than all other numbers, that would turn the domain into a (complete) lattice. Since $+$, properly extended, is still monotone for $\mathbb{N}^\infty$, the above equation system does have a (unique smallest) solution. Namely ∞, as we already mentioned.

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 (1.3) on slide 22

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

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

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

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

$$
\begin{array}{lcl}
\mathsf{RD}_{exit}(0) & = & \mathsf{RD}_{entry}(0) \setminus \{(y,l) \mid l \in \mathbf{Lab}\} \cup \{(y,0)\} \\
\mathsf{RD}_{exit}(1) & = & \mathsf{RD}_{entry}(1) \setminus \{(z,l) \mid l \in \mathbf{Lab}\} \cup \{(z,1)\} \\
\mathsf{RD}_{exit}(2) & = & \mathsf{RD}_{entry}(2) \\
\mathsf{RD}_{exit}(3) & = & \mathsf{RD}_{entry}(3) \setminus \{(z,l) \mid l \in \mathbf{Lab}\} \cup \{(z,3)\} \\
\mathsf{RD}_{exit}(4) & = & \mathsf{RD}_{entry}(4) \setminus \{(y,l) \mid l \in \mathbf{Lab}\} \cup \{(y,4)\} \\
\mathsf{RD}_{exit}(5) & = & \mathsf{RD}_{entry}(5) \setminus \{(y,l) \mid l \in \mathbf{Lab}\} \cup \{(y,5)\}
\end{array}
\tag{1.8}
$$

Remarks One thing should be pointed out: the label of the generated pair corresponds to the label the corresponding equation talks about ("intra"). Perhaps also interesting: this kind of equations expresses the *exits* as a function of *entries.* That's visible in the fact that the left-hand sides of the equations *isolate* the (here

1. *exits.* That reflects the fact that we are dealing with a

forward analysis. Similar remarks can be done about the second half of the equations (to come below), where the *entries* are isolated on the left-hand side, as functions from some exits, as regulated by the *forward* flow graph.

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*

$$\begin{array}{lcl} \mathsf{RD}_{entry}(1) & = & \mathsf{RD}_{exit}(0) \\ \mathsf{RD}_{entry}(2) & = & \mathsf{RD}_{exit}(1) \cup \mathsf{RD}_{exit}(4) \\ \mathsf{RD}_{entry}(3) & = & \mathsf{RD}_{exit}(2) \\ \mathsf{RD}_{entry}(4) & = & \mathsf{RD}_{exit}(3) \\ \mathsf{RD}_{entry}(5) & = & \mathsf{RD}_{exit}(2) \\ \\ \mathsf{RD}_{entry}(0) & = & \{(x,?),(y,?),(z,?)\} \end{array} \tag{1.9}$$

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 slightly 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 construction" (INF5110): that simplifying "restriction" is mildly reminiscent of the treatment of context-free grammars for some constructions (like the LR(0)-DFA construction). For some constructions on grammars, 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 a loop back to the "initial 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)

The following equations show the scheme of data-flow equations of a while-program, more precisely the flow equations given the control-flow graph of a while-program. Besides equations for the initial node, there are equations *per node* of the graph, and equations in connection with the *edges*. Since the nodes represent typically *basic blocks*, one speaks of equations representing *intra-block data-flow* (inside a block), and equations, representing *inter-block data-flow* (between blocks).

As said earlier, we allow ourselves to make nodes that contain only one statement, where typically in implementations, it would be maximal stretches of straight-line code. Nothing much would change though as far as the principles are concerned. The only complication would be: currently, for intra-block flow, there are exactly three cases to consider: conditions coming from loops or conditionals) and the two possible cases for statements, assignments or skip-statements. Those are covered by equations (1.10) and (1.11). In the realistic setting that nodes contains sequences of assignments, the nodes themselves cannot be by a simple finite case distinction, they require an analysis of their own. But that does not change anything, it's just that per a node, one would have a number of equations, which correspond to the equations we would get here, when stretching out the statements across multiple nodes.

If either representation, lumping straight-line code together into one large node or stretching them out to a linear sequence of nodes does no make a difference, why do real compilers bother to work with basic blocks? It's largely a matter of efficiency. Soon we will get an impression of how such equations can be solved. In any rate, if the program contains loops, and most programs do, the analysis will handle equations *repeatedly*. That's caused by the recursive nature of the equations, which is caused by loops in the program code. Because of that, one gains efficiency, if one calculates the accumulated effect of stretches of linear code in a first stage. Of course, if a large basic block is part of a loop, the overall effect of the basic block needs still may be repeatedly taken into account, but one does not repeatedly accumulate the effect of the block's individual elementary steps one over and over again, since one has *pre-computed* their summary effect.

For the participants of the compiler construction lecture: calculating the effect of straight-line code was called (block) *local* analysis.

Intra

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

$$\mathsf{RD}_{exit}(l) = \mathsf{RD}_{entry}(l) \setminus \{(x, l') \mid l' \in \mathbf{Lab}\} \cup \{(x, l)\} \tag{1.10}$$

- for other blocks ($[b]^l$ or $[\mathsf{skip}]^l$), side-effect free:

$$\mathsf{RD}_{exit}(l) = \mathsf{RD}_{entry}(l) \tag{1.11}$$

Inter

$$\mathsf{RD}_{entry}(l) = \bigcup_{l' \to l} \mathsf{RD}_{exit}(l') \tag{1.12}$$

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

$$\mathsf{RD}_{entry}(l) = \{(x, ?) \mid x \text{ is a program variable}\} \tag{1.13}$$

The formulas in equations (1.11), (1.12), and (1.13) are of course *schemas* of equational constraints, so they represent *sets* of equation. The bigger the control-flow graph, the larger the number of equational constraints, with variables of the form $\mathsf{RD}_{entry}l$ and $\mathsf{RD}_{exit}l$.

So far we have only talked about the equations, which can be seen as a **specification** of the flow problem. But we have not solved it. That comes next, where we speak about solving an equation. One needs also think about *how many* solutions such equations have. The equations seen so far are already in a form, which is rather restricted: the left-hand side of the equations are "isolated". That's almost solved, except for one (important) thing: recursion ... Both facts —"isolation" and recursion— will show up more clearly in that we turn it into a *fix-point* equation.

Fix point equations

Earlier, we commented on the specific nature of data-flow equations ("isolated" variables on one side, but recursive). We will soon see how to approach them, in particular how to tackle the phenomenon of recursion. As a first step towards that, we take a second look at those equation, to understand them as what is known as **fix-point** equations. What a fix-point is, is actually very straightforward. Assume one has some kind of function f of type $A \to A$ (where for the time being, A does not matter). Now, a *fix-point* of f is an element a from A such that

$$f(a) = a$$

In other words:

> A fix-point is a solution to the *fix-point equation*
>
> $$f(x) = x$$

i.e., finding a value for x in this constraint. The fix-point equation corresponds directly to the form of the data-flow constraints as we will explain, with the only difference that the data-flow equations typically involve more than one variable; they are of the form $f(\vec{x}) = \vec{x}$. That's actually not a very significant difference, one can see x also as a variable being of a type of a tuple; that's the same of having multiple variables $\vec{x}$.

Now that we know that data-flow equations are nothing else than fix-point equations, we will need to figure out how to solve those. Let's see first in some familiar mathematical setting, like "numbers". Assume as example the square function f_{square} defined as $\lambda y.y^2$ of type $\mathbb{R} \to \mathbb{R}$. So, what about fix-points of that function, i.e., what are solutions of the fix-point equation $f_{square}(x) = x$, i.e., solutions of

$$x^2 = x$$

In this case, it's easy, there are exactly two fix-points, 0 and 1. Those can be visualized by plotting the function x^2 in the standard manner, and the fix-points are where the graph intersects with the 45°-line (representing the identity function), see Figure 1.4.

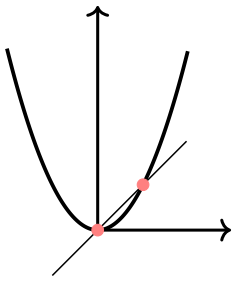


Figure 1.4: Two fix-points of x^2

If we modify the equation a bit to $x^2 + 2 = x$, then there are no fix-point solutions (at least none from $\mathbb{R}$). Other kind of numerical equations over $\mathbb{R}$ involving polynomials may have exactly one, or more than 2 solutions. So, the solution space can consist of 0, 1, 2, etc. solutions.

Data-flow analysis is typically facing fix-point constraints over *non-numerical* domains. We have seen an example characteristic for such domains in the reaching definition example. For such domains, the situation is nicer. For those it is *guaranteed* that the data-flow equations do have a solution, there cannot be *no* solution. Typically, there are many solutions. That, however, does not cause headache. Multiple solutions could be problematic, because in this case, one might not say what *the* solution of the data-flow problem is. However, the solution domains is of a form, that the constraint system has **exactly one most precise solution**. That's the one the analysis returns. That's good news because in this way, one can speak about *the* solution of the data-flow problem. But that's not all. The structure of the solution domain does not only guarantee the existence of a unique best solution. It also assures, on top of that, that it can be calculated without backtracking, i.e., **efficiently.**

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)
- $\vec{\mathsf{RD}}$: the mentioned twelve-tuple of variables

$\Rightarrow$ equation system understood as function F

Equations

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

Fix point equation The above fix-point equation on vectors of RD variables may be broken down more explicitly into 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 $\mathbf{Var}_* = \{x, y, z\}$, $\mathbf{Lab}_* = \{?, 1, \ldots, 6\}$

$$F : (2^{\mathbf{Var}_* \times \mathbf{Lab}_*})^{12} \to (2^{\mathbf{Var}_* \times \mathbf{Lab}_*})^{12} \tag{1.14}$$

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

$$\vec{\mathsf{RD}} \sqsubseteq \vec{\mathsf{RD}}' \text{ iff } \forall i.\ \mathsf{RD}_i \subseteq \mathsf{RD}'_i \tag{1.15}$$

$\Rightarrow$ **complete lattice** (see later)

1.2.3 Constraint-based approach

- next, for DFA: a simple *variant* of the equational approach
- *rearrangement* of the entry-exit relationships
- instead of equations: *inequations* (sub-set instead of set-equality)
- in more complex settings: constraints become more complex, no split in exit- and entry-constraints

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. "constraint-based" is slightly misleading. It's misleading insofar, as also the equational approach is based on constraints, namely *equational constraints*. In the following, instead of equations, we will use "inequations". In the current RD analysis, the solutions are sets (of pairs (x, l)) and 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 ($=$ vs. $\subseteq$). Not only is it straightforward to replace the $=$ by $\subseteq$. Also on a deeper level, there are technical reasons resulting in the following fact: the best (here the 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 settings (for instance functional languages with much more flexible control flow) it's not always the case that equational approaches and constraint based approaches result in the same formulation of the problem (and as a result give the same (best) analysis result).

In general, the subset-constraint-based approach offers more flexibility (thus sometimes more precise analysis) in more complex settings. But, as said, not right now, here it does not really matter. In the context of type and effect systems, we will encounter more complex settings, where $\subseteq$-constraints come in more handy.

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 remains 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 $\supseteq$ allows for a simple *rearrangement* of the constraints. Basically, instead of replacing $s_1 = s_2 \cup s_3$ by $s_1 \supseteq s_2 \cup s_3$, the latter is split into

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

Factorial program: Intra- and inter-block constraints

With this, the reformulation of the equations is straightforward. Again, we group them into intra-block and inter-block equations. Cf. also the equation sets (1.8) and (1.9).

$$\begin{aligned}
&\mathsf{RD}_{exit}(0) \supseteq \mathsf{RD}_{entry}(0) \setminus \{(y,l) \mid l \in \mathbf{Lab}\} \\
&\mathsf{RD}_{exit}(0) \supseteq \{(y,0)\} \\
&\mathsf{RD}_{exit}(1) \supseteq \mathsf{RD}_{entry}(1) \setminus \{(z,l) \mid l \in \mathbf{Lab}\} \\
&\mathsf{RD}_{exit}(1) \supseteq \{(z,1)\} \\
&\mathsf{RD}_{exit}(2) \supseteq \mathsf{RD}_{entry}(2) \\
&\mathsf{RD}_{exit}(3) \supseteq \mathsf{RD}_{entry}(3) \setminus \{(z,l) \mid l \in \mathbf{Lab}\} \\
&\mathsf{RD}_{exit}(3) \supseteq \{(z,3)\} \\
&\mathsf{RD}_{exit}(4) \supseteq \mathsf{RD}_{entry}(4) \setminus \{(y,l) \mid l \in \mathbf{Lab}\} \\
&\mathsf{RD}_{exit}(4) \supseteq \{(y,4)\} \\
&\mathsf{RD}_{exit}(5) \supseteq \mathsf{RD}_{entry}(5) \setminus \{(y,l) \mid l \in \mathbf{Lab}\} \\
&\mathsf{RD}_{exit}(5) \supseteq \{(y,5)\}
\end{aligned} \tag{1.16}$$

$$\begin{array}{lcl} \mathsf{RD}_{entry}(1) & \supseteq & \mathsf{RD}_{exit}(0) \\ \mathsf{RD}_{entry}(2) & \supseteq & \mathsf{RD}_{exit}(1) \\ \mathsf{RD}_{entry}(2) & \supseteq & \mathsf{RD}_{exit}(4) \\ \mathsf{RD}_{entry}(3) & \supseteq & \mathsf{RD}_{exit}(2) \\ \mathsf{RD}_{entry}(4) & \supseteq & \mathsf{RD}_{exit}(3) \\ \mathsf{RD}_{entry}(5) & \supseteq & \mathsf{RD}_{exit}(2) \\ \\ \mathsf{RD}_{entry}(0) & \supseteq & \{(x,?),(y,?),(z,?)\} \end{array} \tag{1.17}$$

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

$$F(\vec{\mathsf{RD}}) \sqsubseteq \vec{\mathsf{RD}} \tag{1.18}$$

Solutions of constraints like the one from equation (1.18) are called *pre-fixpoint*. We will later,in the main part, also encounter post-fixpoints (which are obtained when replacing $\sqsubseteq$ by $\sqsupseteq$).

Pre-fixpoints are also known under other names. A well-known terminology is "being closed under F": Solutions to the variable $\vec{\mathsf{RD}}$ from equation (1.18) are said to be *closed under* F. It's the same terminology when used like: "$\mathbb{N}$ is closed under $+$", because adding two natural numbers gives another natural number. In contrast, $\mathbb{N}$ is not closed under $-$.

1.3 Control-flow analysis (constraint-based)

As remarked in the data-flow analysis part, both the approach with data-flow **equations** as well as the one based on $\subseteq$ are data-flow **constraints**. 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). Here, determining the control-flow is a problem in itself, unlike in the simpler setting before, where determining the CFG was so simple that one would not even bother to call it control-flow analysis (even if it is). Here, we are facing a non-trivial **control-flow analysis** problem and we use a $\subseteq$-constraint-based approach (not an equational) to tackle it.

Now things change, insofar that we are dealing with a **functional language**. As sketched in the previous section, determining the control-flow for a very simple language like the while-language is really straightforward. On top of the control-flow graph of a given program, we to determined some data-flow (reaching definitions for instance). So, this is

clearly a two-stage approach, **first** determine the control-flow in the form of the control-flow graph, and **afterwards**, on top of that, the data-flow analysis. For precision's sake: of course the control-flow graph, strictly speaking, *does not* represent *the* control-flow. It represents a *static abstraction* of the control flow.

When working now with *functional* languages, things get more involved. The simple picture of control-flow on the one hand, that can easily visualized by a graph, and data on the other hand, that "flows" through that graph does no longer works. One reason for that is: characteristic for functional languages is that the line between "control" on one hand and "data" on the other gets blurred. Basically, all is data *and* control in the following sense: the central concept in functional languages is the notion of *functions*. Those are the things being executed, in this sense they are "code". On the other hand, they can be passed around as arguments of other functions and returned as results of functions. This, together with the possibility of nested, lexical scopes for functions, is known as **higher-order functions**. Functions as **first-class citizens** of the language, which means there is no longer a strict separation between "data" on the one hand, and functions on the other. Some languages have restrictions and don't treat functions as first-class citizens, for instance by regulating: the language has lexical scope, one can have local data inside a function, but unfortunately no local function definition. The language has functions, one can pass all kind of data as argument and as return value, but unfortunately not functions themselves etc.

In functional languages, there are typically no such restrictions. As a consequence, it's no surprise that 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 not even consider (non-higher-order) functions there. Adding first-order procedures 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. When saying that analysis for languages with higher-order functions get more involved, then that's fair to say; for instance, control-flow analysis becomes more involved.

Though it may be the case that static analysis becomes more simpler in some aspects, as well. Functional languages often downplay the role of **mutable state** (and compensate by the added flexibility of higher-order function). Still most functional languages support it, it just not encouraged; only purely functional or declarative languages put a total ban on mutable state. Mutable state poses problems for static analysis, so avoiding them makes some aspects of some analyses easier. In some languages that do *not* support higher-order functions, there are ways to recover some expressivity of that feature. Like "patterns" that (almost) work as if one had such functions, perhaps involving function pointers. Relying on (in principle mutable) pointers opens another box of Padora, the flexibility of mutable pointers are notoriosly hard to analyse (and are a notorious source of errors).

As for the importance of functional or decrarative ways of programming: Indeed, some intermediate representations for imperative languages turn the imperative code in some form of "functional representation" known as SSA-format (static single assignment). For instance, LLVM uses CFG+SSA as intermediate representation, because further analyses profit from such a "quasi-functional" intermediate representation. It does not mean, LLVM uses higher-order functions in that intermediate representation, it just treats variables in

an immutable, declarative way, which makes things easier afterwards (of course turning imperative code to SSA takes some effort in itself, but the fact that SSA is very popular shows, it's often worth it).

For the lecture, similar as what we did for the imperative while-language, we work with a stripped down core language capturing (some) core aspects of functional languages

> That's "the" λ-calculus.

There are many λ-calculi (especially, when it comes to typed versions, different type systems drastically change expressivity). Here, in this part of the intro, we restrict ourselves to the bare-bones version: the *untyped λ-calculus*.[2] We use that to show, how one does labels a program in that language. The labelling is more "complex", though not more complex in that it's harder to understand, the labels can in a way just be "nested". In the CFG for the while language, the labels corresponded to the nodes of the CFG. In that sense, one could consider that now nodes can be contained in nodes. On top of that, running the program would change the nesting structure, so the "nested-graph" would somehow dynamically change. In short, there is no good way to visualize the control-flow with some static graph structure. Still, we can capture the relationships between the labels by *constraints*. Solving those constraints then is a form of *control-flow analysis*.

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 longer sense. 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 call 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 as we said before.

[2] Later in the intro, we also present a typed version.

Control-flow analysis

The **general goal for CFA** here is to determine which elemementary blocks leads (or may lead) to which other elementary xblocks

Example 1.3.1. Let's look at a simple functional program. In the program, the function f takes a functional argument. The example does not involve a situation where a function *returns* a function.

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

Listing 1.4: Simple example

We assume the code to be untyped (at least statically untyped). Though it's clear from the code snippet that g and h takes numbers as argument, and f takes as argument functions, as the formal parameter x is applied to 1 in the body. This application is of course the interesting part of the code, as it requires analysis to figure out, which function(s) are actually called. In the example, it's of course immedately clear by looking at it, namely g and h, but in general it might be more involved.

The code also makes use of definitions via let-in, even though the current calculus won't have that contstruct. □

The **goal more specifically here is** to determine for each function application, **which function** may be applied.

The code snippet is a simple example of a functional program, covering in particular the possibility of having functions as argument. What is written here as `fn`$x \Rightarrow \ldots$ might be syntax one could find in a functional language. In λ-calculus notation, it would correspond to $\lambda x. \ldots$. Make sure, even if not familiar with functional programmig, that the short piece of code makes intuitive sense, i.e., one should have a feeling what it does when executed. There is no point in trying to get a feeling what the following analysis does when approximating the behavior of the program, if the behavior of the program is not at least half-way clear for a start.

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.

Labeling

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

Unlabelled abstract syntax

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

Full labelling

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

- functional language: side-effect free

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

As promised at the beginning of the section: the labelling may be conceptually be more complex (allowing labels "inside" labels, so to say) but actually not more difficult to understand how the labelling is done. compared to the while-language.

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

Constraint system for the small example

Labelled example

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

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

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

Explanation of the constraint system The example is rather small, but contains all constructs of the language (ignoring `let`), therefore it illustrates how a labeled 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 (but not the script) 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 kinds** 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 constitutes 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 resp. 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

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

One interesting bit here in the solution is: $\hat{\rho}(y) = \emptyset$: that means, the variable y is 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 this 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 data. At the lowest level, it serves the compiler, for instance, to know the "size" of data so as to allocate an 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 or need to add flexibility, for instance, offering different forms of so-called polymorphism, but still maintaining efficient, safe, scalable, and 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 *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" could sound 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:

$$t : \tau$$

- *judgment*: "term or program phrase has type τ"
- 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.

Remark 1.4.1. *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

$$\vdash S : \Sigma \to \Sigma \tag{1.19}$$

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

The "type system" here is *trivial* in a technical sense (not just meant to say the type system is "super-easy", which it also is). 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 whatsoever. 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 or an *effect type system*, which then yields useful information. Note in passing that the above judgment from equation (1.19) 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.

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

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

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

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

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

Table 1.3: "Trivial" type system: rules

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 *exists* 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, using the rules in a goal-directed manner, correspond to *synthesized* 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 synthesized attributes, they may be more complex (and they may not be syntaxt-directed).

Not all type systems are given in a syntax-directed manner. That means, not all type system specifications can be directly 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*. We won't see much of that 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 (normally the specification of the type system reads simpler and clearer). A non-syntax directed specification for instance will be used when dealing with *subtyping* or other forms of polymorphism, resp. *subeffecting*. To illustrate the point, let's have a look at subtypong. One could for instance specify the following:

$$\frac{\Gamma \vdash \tau_1 \leq \tau_2 \qquad \Gamma \vdash \tau_2 \leq \tau_3}{\Gamma \vdash \tau_1 \leq \tau_3}\ \text{S-TRANS}$$

That's a crystal clear specification requiring that subtyping is supposed to be *transitive*: If τ_1 is a subtype of τ_2, which in turn is a subtype of τ_3, then τ_1 is indeed a subtype of τ_3. That's what the rule unmistakingly stipulates when reading it as "implication" from

the premises to the conclusion. When trying to interpret in a goal-directed manner, the reading goes: To establish that τ_1 is a subtype of τ_3, one can "break down" the problem to the subproblems to check that τ_1 is a subtype of τ_2 and τ_2 a subtype of τ_3. The problem just is: what has been said, going from the goal in the conclusion to the subgoals in the premises cannot really be called an act of "breaking down" the problems. It's not clear in which way the problem has become smaller in the premises, they are not really sub-problems, and it's equally unclear which τ_2 to take. Actually, interpreting the rule in a goal directed manner just say "find such a τ_2 in the middle" to establish the goal. Typically, systems have infinitely many types, just randomly trying out all candidates is a no-go.

We will not dig deeper into the issue here, it's meant to illustrate the point that there is often a gap between the type system, and its algorithmic realization. Later in the lecture we touch upon this question again: how can one turn a type system specification it into a syntax-directed, thus algorithmic version. Sometimes it's far from trivial, sometimes impossible (the type system may be undecidable), but there is a standard route one can try, which we may look at, —and we don't go into too complex systems, anyway.

Types, effects, and annotations

$$\vdash S : \Sigma_1 \to \Sigma_2 \qquad (1.20) \qquad\qquad \vdash S : \Sigma \xrightarrow{\varphi} \Sigma \qquad (1.21)$$

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

$$\vdash S : \mathsf{RD}_1 \to \mathsf{RD}_2 \tag{1.22}$$

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

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

Concerning the "meaning" of the judgment: the formulation is not 100% correct (to say it more bluntly, it's incorrect when taking things precisely ...), 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*.

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 out of the window... 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

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

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

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

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

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

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

Table 1.4: Rules for RD with annotated base types

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":

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

- Be careful:

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

- more concretely

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

Meaning of annotated judgments

Once again: "Meaning" of judgment $S : \mathsf{RD}_1 \to \mathsf{RD}_2$: "if RD_1 is a set of var/label reaching the entry of S, then RD_2 is a corresponding set at the exit(s) of S":

$$\begin{array}{rrcl} & \text{if} \quad \mathsf{RD}_1 & \subseteq & \mathsf{RD}_{entry}(init(S)) \\ \text{then} & \forall l \in final(S).\ \mathsf{RD}_{exit}(l) & \subseteq & \mathsf{RD}_2 \end{array}$$

The example is chosen in such a way, that the two branches definitely have different results: one branch adds $(x, \{l_1\})$ and the other one $(y, \{l_2\})$ (and both potentially remove stuff) and in particular delete potential entries concerning the variables that the other *does not delete*. Even if the statement is embedded in some larger context (loop), the example makes sure, that at the end of the conditional, the RD-sets are different (if RD represents the intended *smallest* solution). The conditional rule, however, insists in its premises that the to branches have *the same* pre/post type. So, especially for the premises, covering the two branches, the given interpretation, namely that RD represents "the" reaching definition" at the beginning and the end, is not correct, if we wish to have that If-rule as given. And indeed, we should should have been warned. There is not something as "the" RD information. There is a thing like *the smallest* RD *information*, i.e., smallest safe approximation of the corresponding data flow constraints. The discussion here hightlights the fact, that the type and effect system does *not* capture that, it captures *all* safe solutions.

Example 1.4.2 (Factorial). Let's revisit the factorial from Example 1.2.1. One may also compare with the control-flow graph from Figure 1.2.

Derivation

$$\dfrac{[y := x]^0 : \mathsf{RD}_0 \to \{?_x, 1, ?_z\} \qquad \dfrac{\dfrac{\dfrac{\dfrac{[z := _]^3 : \mathsf{RD}_{body} \to \{?_x, 0, 4, 3, \cancel{1}\}}{f_{body} : \mathsf{RD}_{body} \to \{?_x, 4, 3\}}}{f_{body} : \mathsf{RD}_{body} \to \mathsf{RD}_{body}}\,\textsc{Sub}}{f_{while} : \mathsf{RD}_{body} \to \mathsf{RD}_{body}} \qquad [y := 0]^5 : \mathsf{RD}_{body} \to \mathsf{RD}_{final}}{f_{while} : \{?_x, 0, 1\} \to \mathsf{RD}_{body}}\,\textsc{Sub}}{f_2 : \{?_x, 0, 1\} \to \mathsf{RD}_{final}}}{f_1 : \{?_x, 0, ?_z\} \to \mathsf{RD}_{final}}}{f : \mathsf{RD}_0 \to \mathsf{RD}_{final}}$$

$$\mathsf{RD}_0 = \{?_x, ?_y, ?_z\} \quad \mathsf{RD}_{final} = \{?_x, 5, 1, 3\}$$

- abbreviate $f_2 = \texttt{while} \ldots; [y := 0]^5$
- *loop invariant*

$$\mathsf{RD}_{body} = \{?_x, 0, 4, 1, 3\}$$

1.4.3 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 be seen particularly 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 described 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

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

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

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

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

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

In Nielson et al. [5], 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).

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

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

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

Table 1.5: Simply typed λ-calculus: type system

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

$$\Gamma \vdash e : \tau? \tag{1.23}$$

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

$$\Gamma \vdash e : ? \tag{1.24}$$

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 successful 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 determine that a recursive call according rule App is in place; however, not enough information is given in the input Γ and $\lambda x.e$ to determine the subsequent recusive call (as τ_1 has to be guessed). 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 (1.24) and not as *confirming* the type as in equation (1.23).

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

> 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:

$$\hat{\tau}_1 \xrightarrow{\varphi} \hat{\tau}_2 \tag{1.25}$$

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

Judgment

$$\hat{\Gamma} \vdash e : \hat{\tau} :: \varphi \tag{1.26}$$

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 a type-based derivation system quite similar to the one here. This allows us 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.

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

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

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

Table 1.6: Call tracking rules

Example 1.4.3 (Call tracking). Let's use as a very simple example the application of the identity function to another instance of the idendity function.

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

The small derivation shows a small concrete deriviation. It's worth to look at the ultimate effect, the set $\{X\}$. The "important" information there is that Y is *not* mentioned, which means that the correspondingly-labelled function is *not* called. □

1.5 Algorithms

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 under which circumstances they have *unique best* solutions. Having that 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 be in practice no point in doing so. Instead one would aim for *deterministic* solution, perhaps realizing specific strategies or heuristics.

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**

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

- F: monotone, finite domain

straightforward approach

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

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

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

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

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).

Bibliography

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

[2] Backus, J. W., Beeber, J., Best, S., Goldberg, R., Haibt, L. M., Herrick, H. I., Nelson, R. A., Sayre, D., Sheridan, P. B., Stern, H., Ziller, I., Hughes, R. A., and Nutt, R. (1957). The Fortran automatic coding system. In *Proceedings of the Western Joint Computer Conference, Los Angeles, CA*.

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

[4] Møller, A. and Schwartzbach, M. I. (2019). Static program analysis. Technical report, Aarhus University. available at `https://cs.au.dk/ amoeller/spa/spa.pdf`.

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

Index

λ-calculus, 13
    typed, 44
$\Gamma \vdash t : \tau$, 42

abstract cache, 38
abstract data type, 14
abstract environment, 38
abstract syntax, 12
algorithms, 54
analysis
    control-flow, 34
attribute, 43
attribute grammar, 3, 43
axiom, 45

call tracking analysis, 53
chaotic iteration, 55
Church-style typing, 53
constraint solving, 54
constraint system, 54
context-sensitive, 43
control flow graph, 17
control-flow, 34
control-flow analysis, 34
control-flow graph, 47
correctness
    partial, 43
Curry-style typing, 52

data-flow analysis, 10
derivation rule, 45
desugaring, 12

effect, 42
effect system, 46
evaluation, 42

factorial, 49
fix-point, 30
FORTRAN, 10
full employment theorem for compiler writers, 9
functional language, 34

grammar
    attribute, 3

Haskell, 42
higher-order function, 34

input variable, 22

judgment, 42

labelling, 38
lattice, 23
    complete, 32

monad, 42
monitoring, 4

partial correctness, 43, 44
polymorphism, 41
pre-fixpoint, 34

reaching definitions, 20
Rice's theorem, 7
run-time verification, 4

semantic analysis, 5
subsumption, 47
symbol table, 43
syntactic sugar, 12
syntax-directed, 45

test generation, 5
testing, 4, 5, 8
total correctness, 44
type and effect system, 46
type system, 41, 45
    static, 41
type theory, 41
type vs. effect, 41

value, 42

while-language, 12