

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

2 Data flow analysis **57**
2.1 Introduction . . . 58
2.2 Intraprocedural analysis . . . 58
2.2.1 Determining the control flow graph . . . 58
2.2.2 Available expressions . . . 61
2.2.3 Reaching definitions . . . 65
2.2.4 Very busy expressions . . . 67
2.2.5 Live variable analysis . . . 72
2.3 Theoretical properties and semantics . . . 75
2.3.1 Semantics . . . 75
2.3.2 Intermezzo: Lattices . . . 80
2.3.3 Correctness of the liveness analysis . . . 87
2.4 Monotone frameworks . . . 90
2.5 Equation solving . . . 94
2.6 Interprocedural analysis . . . 99
2.6.1 Introduction . . . 99
2.6.2 Extending the semantics and the CFGs . . . 101
2.6.3 Naive analysis (non-context-sensitive) . . . 110
2.6.4 Taking paths into account . . . 114
2.6.5 Context-sensitive analysis . . . 120
2.7 Static single assignment . . . 144
2.7.1 Introductory remarks . . . 144
2.7.2 A trivial starting point: straight-line code (SLC) . . . 146

2.7.3 Join points and phony functions 150

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. [10] (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 [9], 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.

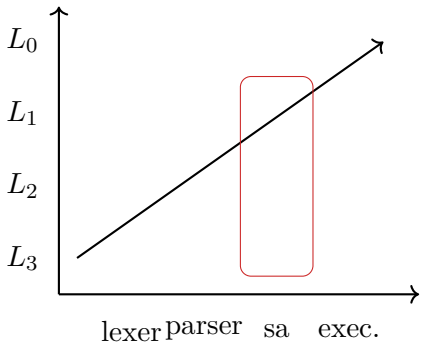


Figure 1.1: Chomsky's hierarchy

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

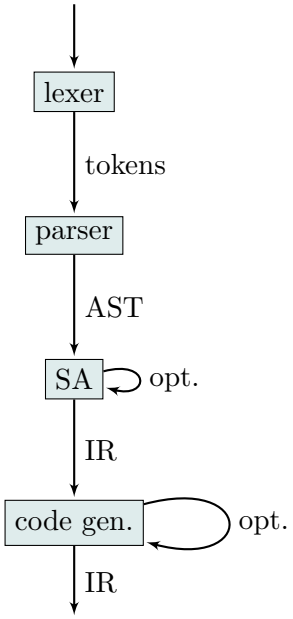
The nature of static analysis

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

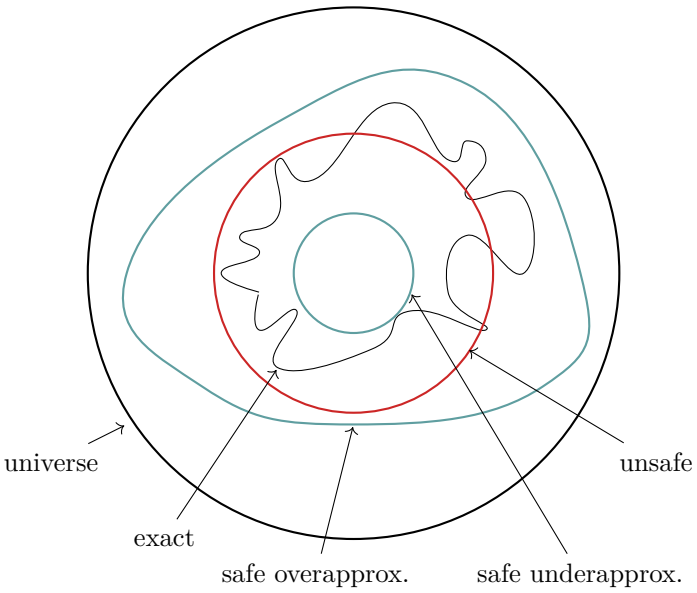
$\Rightarrow$ static analysis as **approximation**

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 [3]. 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. [4] 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
	|	$\mathtt{if}\ [b]^l\ \mathtt{then}\ S\ \mathtt{else}\ S \mid \mathtt{while}\ [b]^l\ \mathtt{do}\ S$	

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

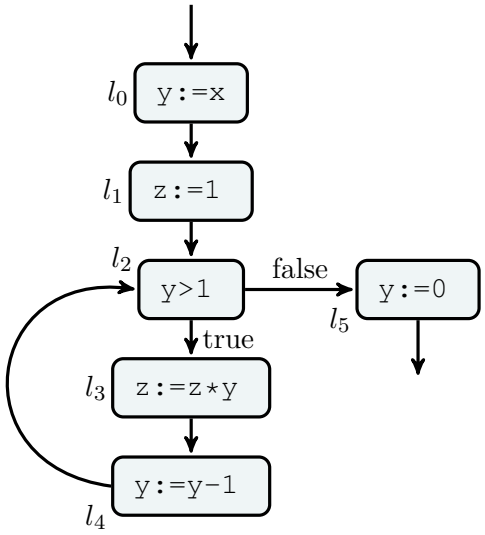


Figure 1.2: CFG factorial

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",

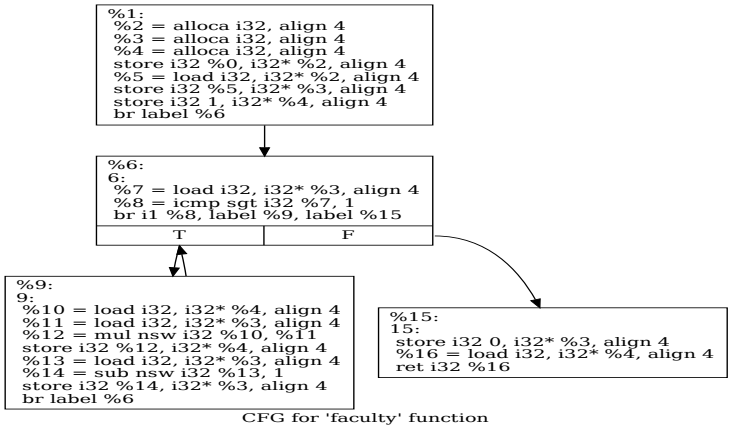
i.e., an intermediate language already rather close to actual machine language, but not yet quite there. In any case, that intermediate language language had *officially* labels, as part of the syntax, that could be used for conditional or unconditional *jumps* (which are like goto's). In other words, 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



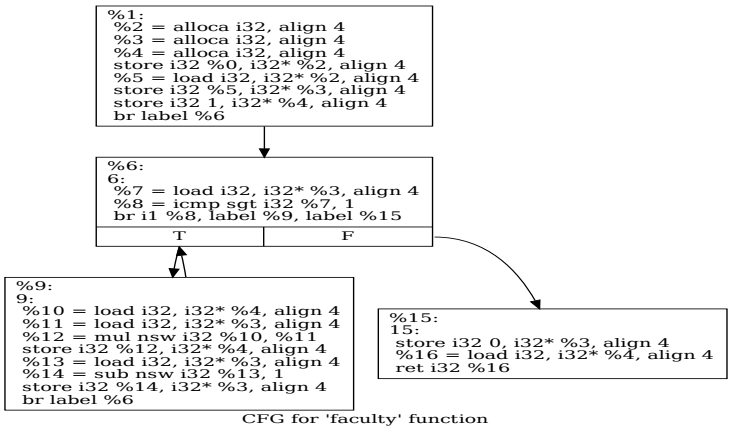
CFG for 'faculty' function

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 [7].

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.

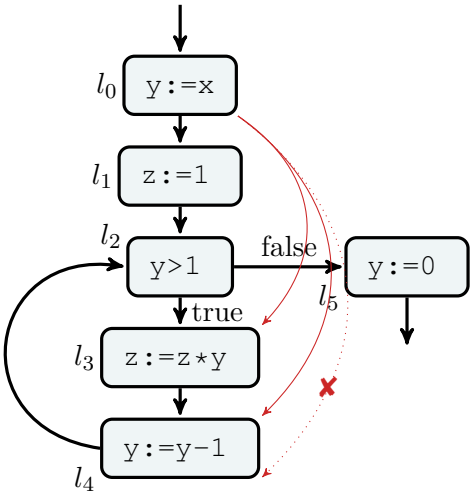


Figure 1.3: Factorial: reaching definitions

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.

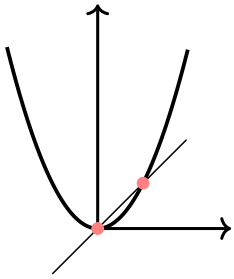


Figure 1.4: Two fix-points of x^2

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.

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

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

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

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

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

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. [10], 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.

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

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

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

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.

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

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).

Chapter 2
Data flow analysis

What is it about?

Learning Targets of this Chapter

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

Contents

2.1 Introduction 58
2.2 Intraprocedural analysis . . . 58
2.2.1 Determining the control flow graph 58
2.2.2 Available expressions . 61
2.2.3 Reaching definitions . 65
2.2.4 Very busy expressions 67
2.2.5 Live variable analysis 72
2.3 Theoretical properties and semantics 75
2.3.1 Semantics 75
2.3.2 Intermezzo: Lattices . 80
2.3.3 Correctness of the liveness analysis . . . 87
2.4 Monotone frameworks 90
2.5 Equation solving 94
2.6 Interprocedural analysis . . . 99
2.6.1 Introduction 99
2.6.2 Extending the semantics and the CFGs 101
2.6.3 Naive analysis (non-context-sensitive) . . . 110
2.6.4 Taking paths into account 114
2.6.5 Context-sensitive analysis 120
2.7 Static single assignment . . . 144
2.7.1 Introductory remarks 144
2.7.2 A trivial starting point: straight-line code (SLC) 146
2.7.3 Join points and phony functions 150

2.1 Introduction

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

2.2 Intraprocedural analysis

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

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

2.2.1 Determining the control flow graph

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

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

The latter 2 functions are of the following type:

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

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

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

(2.2)

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

Using those, determining the edges, by a function

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

works as follows:

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

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

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

They are defined straightforwardly as follows:

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

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

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

Flow and reverse flow

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

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

Program of interest

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

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

"l labels *the* block B"

- even better: *unique* labelling

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

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

individual block gets a new, fresh label. That would lead to *unique* labels, no label is reused, which is stronger (and much more natural) than the assumption here.

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

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

2.2.2 Available expressions

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

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

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

Example 2.2.1 (Available expressions). Consider the following very small "program".

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

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

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

•

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

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

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

Available expressions: general

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

2 ingredients of intra-block flow

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

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

In the introduction, the reaching definition analysis was done without explicitly mentioning the notions of "kill" and "generate", but they where there *implicitly* anyway (for the intra-block equations). Now, we formulate basically all analyses using kill and generate functions, so see the commonalities and differences. Not really all data flow analyses

(in the monotone framework setting) can actually be phrased that way, but very many, including many important ones.

Available expressions: types

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

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

- analysis: pair of functions

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

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

Intra-block flow specification: Kill and generate

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

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

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

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

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

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

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

split into

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

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

Flow equations for AE

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

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

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

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

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

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

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

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

Available expressions

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

$\Rightarrow$ **must** analysis (as opposed to *may*)

- expression is available: if *no path kills it*

- remember: informal description of AE: expression available on *all paths* (i.e., not killed on any)

One intuitive explanation for "why the largest solution?" is to ask oneself: which gives *more* (better) information? We are talking about available expressesion, and the more expressions we have found out to be available, the better.

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

Example 2.2.2 (Available expressions). Consider the following code snippet, with labelled abstract syntax.

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

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

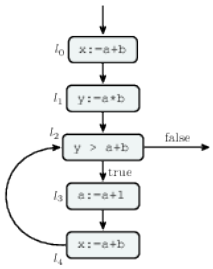


Figure 2.1: Available expressions example: CFG

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

2.2.3 Reaching definitions

This is basically a recap of the introduction, it's the **same** analysis, only phrased with the help of the new definitions, like the kill, and generate functions.

Example 2.2.3 (Reaching definitions). We may have a look at the following piece of code for a simple example.

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

The corresponding CFG is given in Figure **??**.

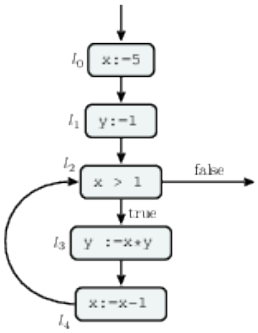


Figure 2.2: RD: Simple CFG

Reaching definitions: types

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

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

- generation and killing:

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

- analysis: pair of mappings

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

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

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

Reaching defs: kill and generate

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

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

Similar to the AE analysis: the interesting case is, of course, the one for assignments. The generation and killing is indeed also quite similar to before. It is the assignment *to*

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

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

split into

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

Flow equations for RD

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

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

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

2.2.4 Very busy expressions

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

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

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

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

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

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

Example 2.2.4 (Very busy expressions).

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

Definition 2.2.5 (Very busy expression)**.** An expression is *very busy* at the exit of a label, if for "all" paths from that label, the expression is used before any of its variables are "redefined" (= overwritten).

- usage: expression "hoisting"

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

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

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

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

very busy expression analysis works *backwards*, for forward analyses it's the corresponding information at the *entries* of the blocks.

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

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

Very busy expressions: types

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

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

- analysis: pair of mappings

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

Very busy expr.: kill and generate

core of the intra-block flow specification

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

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

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

that contain x are modified and thus are *not* very busy at the entry of the block (one could say, as there is no branching withing one block, they are not even busy at all), and thus the kill-function removes those. The reasoning for the AE case is similar, only working forward.

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

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

split into

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

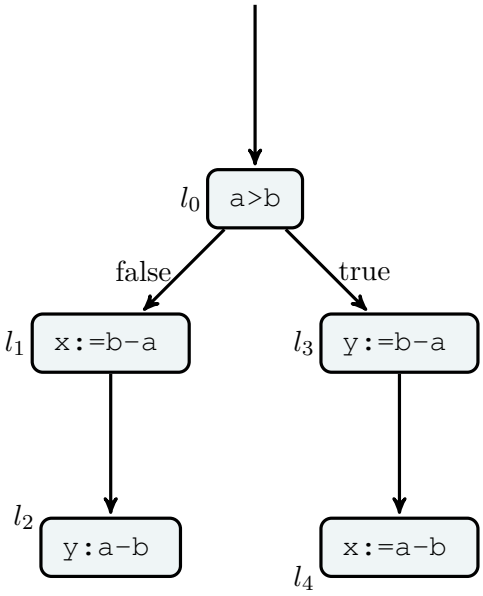
however: everything works backwards now

Flow equations: VB

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

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

Note: Doing a *backward* analysis, the roles of entries and exits are now reversed. The kill and generate functions now calculate the *entry* as function of the exit point. Analogously, the inter-block flow equations (of the graph) calculate the exit of a block as function of the entries of others.



Example 2.2.6.

Example 2.2.7 (Infinite loop). Let's have a look at the example from Figure 2.3. Since

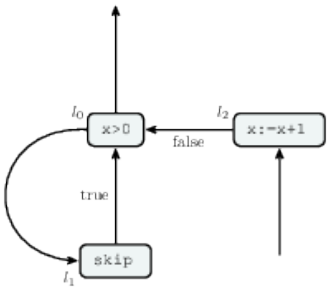


Figure 2.3: Infinite loop

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

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

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

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

On the other hand: it seems that the analysis could make the assumption that there *is* actually a path on which the $x + 1$ is *never, ever* evaluated. That seems to indicate that one should intuitively consider the expression $x + 1$ *not very busy*. If we don't know how

often the loop body is executed (if at all), and since we cannot exclude that the body is taken infinitely (as in this case), it seems plausible to say, there's a chance that $x+1$ may not executed and therefor count it as not very busy.

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

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

2.2.5 Live variable analysis

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

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

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

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

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

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

live: A variable is **live** (at the exit of a label) if there *exists* a path from the mentioned exit to the *use* of that variable which does not assign to the variable (i.e., redefines its value). **Goal** therefor: determine for each program point: which variables are live at the exit of that point.

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

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

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

Live variables: types

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

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

- analysis: pair of functions

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

Live variables: kill and generate

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

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

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

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

Flow equations $\mathsf{LV}^=$

split into

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

however: everything works backwards now

Flow equations LV

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

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

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

Example

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

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

For the participants of the compiler construction course (INF5110). The course presented a *local* live variable analysis, which concentrated on *straight line code*. The code was so-called three-address code and had two types of variables: normal ones and so-called "temporaries" (temporary variables). The standard variables were assumed *live* at the end of the straight-line code. The reason being that the SLC is code contained *inside one basic block*. Since the end if the block is not necessarily the end of the program, the analysis had to assume conservatively that chances are, that variables may by used by any potentially following block, and thus they variables were assumed live. Temporaries,

on the other hand, were treated as *dead* at the end, which was justfied by the fact that the code *never* used temporaries from a previous block. That, of course, depended on the knowledge how this code was actually generated. The general point is that of course the formulation of the live variable analysis (or others) must go hand in hand with what is actually going on, i.e., the semantics of the language and the assumptions about how the program is used ("will the content of the variables be returned to a caller after the program code or not", "might there by a block after the code being analyzed or not, and if so, will it make use of temporaries resp. variables or not").

Example 2.2.8 (Looping example). Let's revisit the CFG from Example 2.2.7 and Figure 2.3. □

2.3 Theoretical properties and semantics

2.3.1 Semantics

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

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

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

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

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

Relating programs with analyses

- analyses
  - intended as (static) *abstraction* or overapprox. of real program behavior
  - so far: *without real connection* to programs
- soundness of the analysis: **safe** analysis
- but: *behavior* or *semantics* of programs not yet defined
- here: "easiest" semantics: *operational*
- more precisely: *small-step SOS* (structural operational semantics)

States, configs, and transitions

fixing some data types

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

Transitions

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

Semantics of expressions

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

simplifying assumption: no errors

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

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

clearly:

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

In the introductory remarks, we mentioned that we will do some specific form of semantics, namely an *operational semantics*. That's not 100% true. For dealing with the control-flow structure of the while language, we will indeed formulate operational rules to describe

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

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

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

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

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

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

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

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

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

Each construct is convered by one or more rules (the second rule for conditionals is left out). Even if there are constructs where there is more than one rule, the alternatives are mutually excclusive. For instance, in case of the conditional, either the rule for the

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

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

Derivation sequences

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

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

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

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

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

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

Correctness of live analysis

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

LV constraint system

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

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

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

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

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

Equational vs. constraint analysis

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

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

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

2.3.2 Intermezzo: Lattices

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

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

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

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

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

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

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

The relation $\sqsubseteq$ allows to say when an element is smaller (or equal) than another, comparing two elements. Obviously, with $\sqsubseteq$ given, one can also say when one element is larger or than two elements. As boring and obvious as that is, there is a name for it, namely being an upper bound. So: c is an *upper bound* of a and b, if $c \sqsupseteq a$ and $c \sqsupseteq b$. Obviously, we

assume that the reader understands that $\sqsupseteq$ is "the same" relation than $\sqsubseteq$, just written the other way around. Anyway, the definition of upper bound or, dually, of lower bound of two elements is completely unspectactular, and it does not get more spectacular, if we "generalize" it to speaking of the upper bound of 3, 4, etc., elements. Like c is a lower bound of a_1, a_2 and a_3, if $c \sqsubseteq a_1$, $c \sqsubseteq a_2$, and $c \sqsubseteq a_3$. Actually, we might as well use the concepts on arbitrary sets of elements from A, speaking of a upper resp. lower bound of a set A' where $A' \subseteq A$.

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

But actually, that's not quite true, or at least we have to be careful what we mean by "upper and lower bounds exists" (let's say for $(\mathbb{N}, \leq)$. Certainly, if we take two numbers n_1 and n_2 as above, those bounds exist; the was nothing wrong with the above argumentation. Same for the upper bound of 3, 4, etc. numbers. But what about the following: is there an upper bound of *all even numbers*, i.e., the set $\{0, 2, 4, 6, \ldots\} \subseteq \mathbb{N}$? Well, there is *no* upper bound, at least not inside $\mathbb{N}$. The point here is that there is a qualitative step from asking for the existance of bounds for *finite* sets to the existence of bounds for *infinite* sets.

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

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

linear orders, this is very straightforward to see ("assume that there are *two* least upper bounds ...").

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

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

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

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

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

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

Why does one need such specific assumptions, insisting on working with lattices? We are trying to find solutions to a constraint problem (data flow equations or $\sqsubseteq$-constraints). We know already, from discussions earlier, that those constraints have more than one solution (there is more than one safe approximation), that solutions are ordered wrt. their precision, and that solving data-flow constraints corresponds to finding a *fixpoint* (resp. a pre-fixpoint or post-fixpoint). Trying to solve a constraint system is, very generally, a

search problem: find a solution if it exists (or maybe find all). In particular, since here solutions are ordered, find a good solution. The fact that one is after a good solution, for a given measure, makes the search problem to an *optimization* problem.

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

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

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

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

Intermezzo: orders, lattices etc.

as a reminder:

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

Here we are working with a specific form of lattice, called *complete* lattice. It's a very nicely behaved lattice, which makes it useful for the monotone framework. The important condition, which makes the lattice actually *complete* is rather hidden in the above definition. It's that meets and joins exists *for all subsets*. If we drop the "completeness", then one would still require joins or "least upper bounds" $a \sqcup b$ and or "greatest lower bounds" $a \sqcap b$ to exist. The difference here is we speak about *binary* such operations. To be complete, it's also required the meets and joins exists also for *infinite sets*. A priori a lattice is not required to be finite (and many interesting ones are not). For those situations, the existance of *binary* meets and joins implies the existance of *finite* meets and joins (those can always be expressed by a number of binary ones). What is *not* guaranteed it the existance of *arbitrary* meets and joins, including infinite ones. But that's required for a lattice to be complete. There are also other forms of lattices, for instance, if one only needs joins, but not meets, one can get away with a semi-lattice, and there are many more variations. For the lecture, we generally simply assume *complete lattices* and thus, the montone framework is happy. In particular, if we are dealing with *finite* lattices, which is an important case, we don't need to consider *infinite* sets, and "standard" lattices with binary meets and joins (and least and largest elements) are complete already.

Fixpoints

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

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

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

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

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

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

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

Define "lfp" / "gfp"

$$lfp(f) \triangleq \bigsqcap Fix(f) \quad \text{and} \quad gfp(f) \triangleq \bigsqcup Fix(f) \tag{2.7}$$

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

Equation (2.7) just gives the *names* to the two elements of the lattice defined by the corresponding right-hand sides. We know that those elements do exist, thanks to the fact that L is a complete lattice and $lfp(f)$ and $gfp(f)$ as defined are unique elements of the lattice. The chosen names suggest that the two thusly defined elements are the least fixpoint, resp. the greatest fixpoint of the monotone function f. But that requires a separate *argument*. Finally, if we take it really serious, an argument should be found that allows to speaking of *the* least fixpoint. If there is more than one least fixpoint, one should avoid talking about "the least fixpoint" (same for the largest fixpoint). The argument for uniquess of least fixpoints (or for greatest fixpoint) is very simple though, similar to arguing for the uniqueness of "the least upper bound" etc.

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

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

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

their precision. These other, non-lattice based kind of conditions one could use to ensure existance of fixpoints don't play a role for our purposes

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

Tarski's theorem

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

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

$$\begin{array}{lcll} \mathit{lfp}(f) & \triangleq & \bigsqcap \mathit{Red}(f) & \in \quad \mathit{Fix}(f) \\ \mathit{gfp}(f) & \triangleq & \bigsqcup \mathit{Ext}(f) & \in \quad \mathit{Fix}(f) \end{array} \tag{2.8}$$

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

Fixpoint iteration

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

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

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

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

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

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

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

2.3.3 Correctness of the liveness analysis

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

Basic preservation results

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

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

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

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

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

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

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

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

Correctness and correctness relation

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

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

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

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

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

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

But first to the induction. It's induction over the the number of steps $\rightarrow^*$. Induction is a general proof technique. It involves an something one likes to prove, a property. It's sometimes called *induction hypothesis*. Then the proof proceeds by establishing that it holds in the *base case* (or for all base cases) and establishing the *induction case* (or cases). The exact shape depends in the structure for which one proves something. Induction is often first encounterer as induction over the natural numbers, where the base case is 0, i.e., establishing the property for 0, and the induction case involves proving that the property holds for $n + 1$ under the assumption that it holds for n. It's also called induction step, and it steps from n to $n+1$, showing *preseveration* of the induction hypothesis when doing that "step". Of course there is also induction over trees (where the base case are the leaves and the induction step involves proceeding from children to parent nodes. Derivation trees could be an application in the context of this lecture.

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

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

steperror

There are 3 ingredients to the argument, where the first is for free, i.e., one does not have to establish anything.

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

proof by **induction**

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

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

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

$$\sigma_1 \sim_V \sigma_2 \text{ iff } \forall x \in V.\sigma_1(x) = \sigma_2(x) \tag{2.9}$$

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

2.4

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

Figure 2.4: Illustration: liveness variables

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

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

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

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

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

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

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

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

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

Theorem 2.3.11 (Correctness (many steps))**.** *Assume live* $\models \mathsf{LV}^{\subseteq}(S)$

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

2.4 Monotone frameworks

We have seen 4 different classical analyses, which all shared some similarities. In this section, those analyses will systematically be put into a larger context, which is known as *monotone framework*. As unifying principle, this was first formulated by Kildall [7] and

constitutes in a way the common orthodox and completely standardized understanding of what classical data flow analysis is.

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

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

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

Monotone framework: general pattern

$$\begin{array}{lcll} \mathit{Analysis}_\circ(l) & = & \begin{cases} \iota & \text{if } l \in E \\ \bigsqcup\{\mathit{Analysis}_\bullet(l') \mid (l', l) \in F\} & \text{otherwise} \end{cases} \\ \mathit{Analysis}_\bullet(l) & = & f_l(\mathit{Analysis}_\circ(l)) \end{array} \tag{2.10}$$

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

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

we did, via kill and transfer functions (as many analysis domains are sets of information of interest), but not all. The general monotone framework simply requires a function that transform flow information on one end of a basic block to flow information at the other end. For a forward analysis, the flow information at the exit of a basic block is expressed as a function on the entry of the block, and for backward information, it's the other way around.

Monotone frameworks

direction of flow:

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

sort of solution

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

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

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

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

Basic definitions: property space

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

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

Transfer functions

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

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

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

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

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

Summary

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

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

Instead of the above condition, one might require

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

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

Let's finish up by summarizing the 4 classical data flow analyses we covered. For a label consistent program S_*, they are all *instances* of a monotone, distributive, framework. That the properties form a lattice in each case is immedate (as all instances are using sets ordered by subset or superset inclusion. The qascending chain condition is guaranteed, for all 4, the lattices are *finite*. The *closure* conditions on $\mathcal{F}$ (something we have not really discussed much as it is mostly unproblematic as a naturally given condition) are also fine. The transfer functions are monotone, and the set $\mathcal{F}$ of all transfer functions is closed under identity[1] and composition. All 4 instances are additionally **distributivity**, which is assured by the fact that the analyses are formulated using kill- and generate-functions for the transfer function.

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

Table 2.2: Overview over the 4 example DFAs

2.5 Equation solving

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

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

formulation of an corresponding iterative approximation algo in the introcuction, called *chaotic iteration*.

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

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

Worklist data structure

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

Comparison to chaotic interation

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

$$\mathsf{RD}_j \neq F_j(\vec{\mathsf{RD}}) \ . \tag{2.11}$$

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

Looking carefully, the worklist do *not* contain exactly all current situations described by equation (2.11). It contains an **approximation** thereof! The worklist contains a *superset*

of the exact set of violated constraints, a set of constraints that are suspected to be violated and thus are potentially in need for further treatment.

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

Equation solving and traversal

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

Solving the analyses

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

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

MFP

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

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
$\mathsf{RD}_j := F_j(\mathsf{RD}_1, \ldots, \mathsf{RD}_{12})$

Listing 2.1: Chaotic iteration

The chaotic iteration is given on [10, page 25] in the introduction, for reaching def's. The chaotic iteration is not a work-list algorithm, it lacks this data structure. Instead it uses non-determinism and not an explicit representation (as data structure) of the edges still to be treated.

The chaotic iteration here is also specialized to the faculty-example, im that it mentions 12 variables.

Worklist algorithms

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

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

- for more details (different traversal strategies): see Chap. 6 from [10]

WL-algo for DFA

- WL-algo for *monotone frameworks*

⇒ input: instance of monotone framework

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

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

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

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

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

Listing 2.2: ML-code

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

MFP: properties

Lemma 2.5.1. *The algo*

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

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

□

Time complexity

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

$\Rightarrow$

$$O(e \cdot h) \tag{2.12}$$

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

2.6 Interprocedural analysis

2.6.1 Introduction

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

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

The formalization of the run-time environment, i.e. here, the organization of the memory for procedures is more complex (being *dynamic*). So far we had a finite amount of variables per program, statically fixed. Now, since procedures can be called recursively, the amount of memory to store the content of local variables resp. parameters is not longer statically fixed. The language extension supports *first-order* procedures of functions, but not higher-order procedures. As a consequence, the memory for the procedure local variables can be arranged in a **stack**. There are different ways how one can formalize that. For instance, one could explicitly introduce some stack with push an pop etc. We do a stack (because

that's what we need), but it's not so visible as if we would work with an explicit stack, it's more implicit (using a syntactic bind-construct, see later). Instead of a *state* (written σ) as before, the memory is represented more fine-grained, given in terms of **locations** and **stores**. Basically, that representation makes explicit that variables are connected to addresses (locations) and one can **allocate** new addressess when needed. That allocation works in a LIFO-manner, i.e., it works like a stack. Though, as said, we don't introduce an explicit stack structure.

Adding procedures

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

Extending the syntax

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

- procedure names p
- statements

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

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

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

Parameter passing What might look a bit unusual is the parameter passing mechanism. It highlights the fact that parameter passing is typically not a one-way street. It's not just about passing actual parameters from the caller to the callee. It also involves a mechanism the pass results back. Of course, one can evaluate a procedure with the intention of getting its side effect only (with a return type like `void` or similar). That is often done when using languages supporting **call-by-reference,** which we don't do. Side-effects would have to be done one the global variables in the current language. That probably should not be called "passing the result back" though it can be used that way. When dealing with side-effect only procedures, it should not even be called **evaluating** the procedure. The word "evaluating" conventionally means, executing something with the purpose of obtaining a **value** at the end and for a side-effect-only procedure, one cannot speak of a resulting value.

Now, the mechanism here has not only "input" parameters, it also has "output" or result parameters. The syntax restricts to just one, but it's easy to generalize, and actually, in the examples, we make use of multiple parameters. The mechanism for passing back the return value is called **by result**. The call site syntax `call` $p(a, z)$ can be read as $z := p(a)$: the result will be stored in z which is a variable in the scope of the caller.

Example 2.6.1 (Fibonacci).

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

The example is not meant to show how Fibonacci is best implemented. It's a rather inefficent way to do so. The reason for its inefficiency are the *two* recursive calls, and there are easy ways to write a tail-recursive and thus a more efficient versionof the function. However, for illustrating challenges with interprocedural analysis, we intend to have an example with *two* calls, and that's an easy example of that. □

2.6.2 Extending the semantics and the CFGs

Next comes the adaptation of the definition of the control-flow graph. To do so, we need to adapt and extend the definitions of flow, blocks, etc. Actually, it's pretty straightforward. The new part deals, obviously, with the procedures; also that is straightforward. The basic trick is that we introduce **new kinds of edges** to deal with the procedures and calling them. The definition/presentation proceeds (on the slides) in *two* steps, first the call sites, afterwards the procedures themselves. Besides the fact that we need "non-convential" edges (namely connecting 4 nodes), it's still straightforward.

Another thing one can learn from the example is parameter passing, in particular the way the return is passed "by reference". However, the focus in the following will not be how to

program with this syntax (it is not intented as concrete surface syntax anyway), but how to analyze it.

Block, labels, flow: building the CFG

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

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

Note that there are two *new* kinds of flows (written slightly differently, namely an edge for *calling* and one for *returning*. That's actually not the end of it, later we will introduce the so-called interprocedural edges *IF*, that's when conceptually something new happens, namely "edges" which are 4-tuples, not just tuples.[2] So apart from the fact that the edges are written slighly differently, there's no real change, the new edges are treated the same way as the old.

For procedure declarations

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

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

$$\begin{array}{rcl} \mathit{init}_* & = & \mathit{init}(S_*) \\ \mathit{finall}_* & = & \mathit{final}(S_*) \\ \mathit{blocks}_* & = & \bigcup\{\mathit{blocks}(p) \mid \texttt{proc}\, p(\texttt{val}\, x, \texttt{res}\, y)\texttt{is}^{l_n} S \texttt{end}^{l_x} \in D_*\} \\ & & \cup \mathit{blocks}(S_*) \\ \mathit{labels}_* & = & \bigcup\{\mathit{labels}(p) \mid \texttt{proc}\, p(\texttt{val}\, x, \texttt{res}\, y)\texttt{is}^{l_n} S \texttt{end}^{l_x} \in D_*\} \\ & & \cup \mathit{labels}(S_*) \\ \mathit{flow}_* & = & \bigcup\{\mathit{flow}(p) \mid \texttt{proc}\, p(\texttt{val}\, x, \texttt{res}\, y)\texttt{is}^{l_n} S \texttt{end}^{l_x} \in D_*\} \\ & & \cup \mathit{flow}(S_*) \end{array} \tag{2.13}$$

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

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

But now we start treating the new situation for real, introducing really **ew kind of edges** representing the interprocedural flow (IF) The inter-procedural from goes from call-site to a procedure and back, i.e., include edges $(l_c; l_n)$ and $(l_x; l_r)$. But not separately, as two edges, but the two edges from one call-site to one procedure are coupled together. Doing that allows for a more *precise* (= better) capture of the control-flow. We use *IF* as abbreviation for *inter-flow*$_*$ or *inter-flow*$^R_*$ (for the forward or reverse situation). With the pair of edges for a call and for a corresponding return coupled together, that makes such inter-procedural flow edges 4-tuples:

IF

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

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

Example 2.6.2 (Fibonacci flow). The CFG for the fibonacci example is shown in Figure 2.5

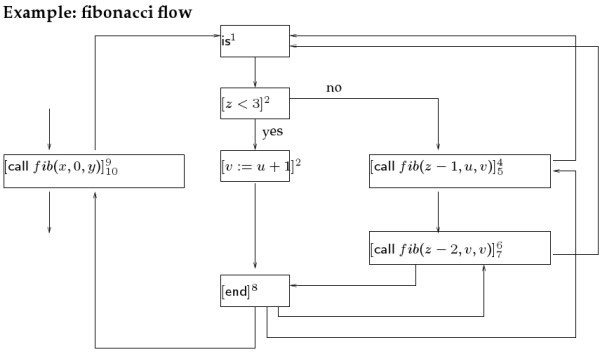


Figure 2.5: CFG for Fibonacci

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

Semantics: stores, locations,. . .

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

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

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

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

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

Representation of "memory"

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

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

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

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

SOS steps

Now to the semantics as transitions between *configurations*. We focus on the transitions for the new concepts, i.e., calls and returns. Before we do that, we need to **generalize** the notion of **configuration**. Part of it is achieved already by introducing locations or addresses and "splitting" the state into two separate mappings (the store and the environment). The environment is responsible to associate addresses to variables. It will also be used to realize the "stack-like" arrangement of accessible address space. There are different ways to do that. The way it's done here is that the envirenment part is "baked into the syntax part" of the configuration. That's one way of achieving the stack-like behavior we need.

Currently, however, the syntax is not prepared for that, there is no syntax that can be (mis-)used for that. What we are doing then is to simply *add* additional syntax. This syntax is not for the user, i.e., it won't show up in the program before it starts to run. It's ony generate on-the-fly, at run-time with the purpose of capturing some semantics concepts of behavior (in this case calls and returns and the corresponding stack-like run-time environment). Such "semantic tricks" are actually quite common. It's known as **run-time syntax**. For the simple while language without procedures, we did not see such run-time syntax, basically since the language is so trival. In many other situations, one works with such run-syntax. It's a way to formalize *run-time environments*, i.e., for instance in connection with memory treatment, in an abstract way and in a syntax directed way:

> the spirit of SOS (structural operational semantics) is: the structure of the syntax steers the execution steps, and if that calls for new syntax called run-time syntax, why not?

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

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

$$\rho \vdash \langle S, \varsigma\rangle \to \langle \acute{S}, \acute{\varsigma}\rangle \qquad (2.14) \qquad\qquad \rho \vdash \langle S, \varsigma\rangle \to \acute{\varsigma} \qquad (2.15)$$

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

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

In the rules covering old syntax (without calls), the environment part never changes, though of course executing an assignment updates the store part (in the analogous way that an assignment in the previous setting updated the state σ). That reflects the fact

that executing code inside one current procedure body (more generally, executing code inside one current scope), never changes the association of variables to addresses. That binding of variables to addresses, here represented as evironment ρ, is **fixed** per **scope.**

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

Before we go to the concrete rules: the transition in equation (2.14) captures that the program with the store executes one step *inside a given environement* ρ. And the environment does not change! How can we arrange that conceptually the body of a procedure is supposedly executed in a *different* environment, one that uses a local state space for locally scoped variables? The form of the rules seems not able to capture that since ρ is unchanged. That's a proper observation, and that motivates the addition bind-construct: the **bind-contruct** allows to add a "local" environment baked into the syntax, that's what the bind-syntax is introduced for.

Remark 2.6.3 (Run-time syntax)**.** *In the lecture 2022, I mentioned that introducing run-time syntax is a common technique, however, I could not remember an example that is plausible enough, and not to esotheric. In the meantime, I remembered one. For people who went to the IN2040 course, they were seeing an example of that (though it was not explained using the words "run-time syntax").*

The course is based on Scheme (as Lisp-dialect), and the course also explains a bit how Scheme is ***interpreted***[3]*. An interpreter nothing else than implementation of an operational semantics, actually an structural operational semantics in case of Scheme/Lisp, as it more or less directly operates on abstract syntax trees*[4]

Where is the run-time syntax coming in? One part where they use run-time syntax is when it's discussed how to achieqve ***lazy evaluation*** *(not eager evaluation which is the standard for Scheme). That can be done by avoiding and delaying an evaluation of some argument, but "freezing it up". That step put's the expression whose evaluation needs to be delayed into a particular construct. Traditionally it's known as* ***thunk****. That's a construct of* ***run-time syntax****, it's generated when the interpreter does a lazy call, turning its argument into "thunks" which for a while remain unevaluated (until needed). The mechanism is comparable to what we do here: a call here introduces* ***bind*** *and treats in with special rules. Of course* ***bind*** *serves a different purpose than* ***thunk****, but both a constructs are not from the official user syntax, but introduced by the interpreter (or operational semantics) to do its job.* □

Call-rule and introducing bind

Now we turn to the treatment of calls and returns (including the treatment of the bind-construct). Altogether, the behavior is covered by 3 rules. The first one deals with calls, and it introduces the new run-time construct. See Table 2.3

[3] the course is based on a famous book called SICP: Structure and *interpretation* of computer programs.

[4] Again the book does not mention that word "abstract syntax tree", but that what's done. Indeed for lisp the connection between syntax and data (=lists), resp. syntax trees and lists of lists of lists ... is particularly close.

It can be seen as realizing the parameter passing, actually the parameter passing from caller to callee, not yet passing back the return value (but preparing for it by allocating some memory for it). Calling a procedure p looks up the definition of that procedure in D_* (and we assume it can be found uniquely there). Since the procedure is defined with two formal parameters, we need two places where the parameters can be stored resp. passed (by value resp. by result). Those are the **fresh locations** ξ_1 and ξ_2 in the rule.

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

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

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

Table 2.3: Call-rule

Bind-construct

The bind-construct is covered by two rules. The construct is introduces when executing a call-statement and thus the rules for bind describe how to execute the body of a procedure after being called. There are two cases: during execution, and "termination" which in this case means, finishing with the body; in the larger context of being called, that means returning to the caller. When executing the body, it's done as for standard statments (covered by the other rules). Of course, that includes further calls. If that occurs, that leads to another execution of the CALL-rule, which in turn leads to execute the second procedure body in another environment. Note that the rule BIND_1 is a *recursive* derivation rule (BIND_2, as well, and actually as the prior rules covering sequential composition): the step of a bind construct is explained by recursively determining the step of the body. When a code calls a procedure, that in turn calls a further procedure, and that in turn a further one ..., that implicitly (in the derivation tree) leads to a number of environments ρ in a nested fashion in the derivation tree, which is an abstract representation of the stack frames.

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

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

Table 2.4: Call-rule

$\Rightarrow$ formulation of correctness must be adapted, too (Chap. 3)[5]

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

In this lecture (in 2020), there had been some remark and feedback from the audience about what to expect from a semantics. It was mentioned that it should for instance fix the order on which the arguments are passed, and the shape of the data structure where they are stored. In compiler terms, the data structure would be called *stack frame* or *activation record* and the conventions or specification about how to do the stepwise passing of the arguments (like, in which order) is known as *calling conventions*.

The call-rule here deals with that in a more *abstract* form. For example we don't pass arguments in some particular order, like fixing that the function arguments are to be stored "in reverse order" or something. In line with that: the locations, respresenting addresses,

[5] Not covered in the lecture.

are not even *ordered*. Besides that, since the language does not support side-effects in procedure arguments, the order does not really matter (though of course a compiler will arrange the copying in one particular order).

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

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

the execution proceeds to

$$S_p; S \ .$$

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

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

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

Finally, in standard stack frames, there is also the notion of a *dynamic link*, which allows the run-time enviroment to pop-off a stack frame (since stack frames are of difference size). Conceptually, we need to deal with that, but it's done implicitly in the formalization and

not with some pointers, in the same way that we conceptually return from the body of procedure call, but the formalization does not need to remember explicit return addresses (as explained above).

2.6.3 Naive analysis (non-context-sensitive)

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

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

Transfer function: Naive formulation

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

Naive

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

Equation system ("naive" version)

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

with

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

- analysis: *safe*

- unnecessarily imprecise, too abstract

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

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

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

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

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

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

But back to the remark about *unessessary* imprecision. In the setting with procedures, even if the call-graph is connected, not all paths through the CFG are actually possible

(assuming data abstraction). The reason for that is that, no matter the data, the caller necessarily returns to the point where it was called. If there are two or more places where a procedure is called, then ignoring that coupling between call-edge and return-edge in a path, as the context-insensitive analysis does, leads to an imprecision *on-top of that caused by abstracting data*. In that sense it's "unessessary" imprecision or an additional factor of imprecision. **Context-sensitive** analyses try to address and mitigate that additional imprecision, the naive context-insensitive analysis ignores the potential of improvement.

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

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

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

That will be an approach the lecture will pursue.

Path sensitivity and context-sensitivity and others In the above discussion, we used as illustation a conditional with $x \geq 0$ as branch condition; in that context we mentioned the terminology of *path sensitivity*. Classical data flow analysis is not path sensitive. It can be made more precise (becoming path-sensitive), if one extended the tracked information in the analysis and have it *remember* information about that choice. Similar remarks

apply to our setting here: one can make interprocedural analysis more precise, by remembering information about where a procedure had been called. This information can make the place to return to more precise. As far as terminology is concerned: the first situation, concerning branching is about path-insensitivity vs. path-sensitivity. The second one concerning procedure calls is called context-insensitivity vs. context-sensitivity of interprocedural analysis.

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

> add more information and you get more precise (but the analysis with take more effort, in particular time).

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

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

That very conventional notion of paths will be defined in Section 2.6.4. We know that data flow problems in the monotone framework do have unique best solutions. We also know that so far the monote framework does not operate on the notion of paths. An alternative problem is to ask: given a node in a graph and some data flow problem: what it the smallest value in the lattice at that location, considering all the different ways I can reach that location? That different from the standard data flow question, since the data flow equations don't "follow the edges", i.e., don't explore paths (remember the chaotic iteration). The follow-the-paths solution to a data-flow problem is called **meet-over-all-path (MOP)** or *join-over-all-paths*, depending on whether we do a must or a may analysis). Anyway, asking that question is more complex! In a typical program, with loops, there are **infinitely** many different paths by which nodes can be reached; some nodes of course still can be reached by finitely many paths (nodes at the "beginning" of a program resp. of a control-flow grahp, before encountering any loop).

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

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

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

2.6.4 Taking paths into account

It corresponds to the standard notion of a path through a graph. Note that so far that notion did not play a role. The data flow analysis, for instance in the form of a worklist algorithm, is concerned with solving equation, piecewise repairing violations of the equations. Each constraints correspond to an *edge* in the graph (or perhaps a set of edges in the equational approach). Anyway, repairing an constraint corresponds to treating an edge (or a set). But the solving procedure for the constraints is not required to follow the edges one after the other, like following paths through the control flow graph. The

concrete semantics and execution will do that, the analysis may choose other traversal strategies.

Paths

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

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

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

Definition 2.6.4 (Paths and transfer functions for paths)**.**

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

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

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

Meet over all paths

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

MOP

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

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

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

MOP vs. MFP

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

Lemma 2.6.5 (Connection between MFP and MOP)**.**

$$MFP_\circ \sqsupseteq MOP_\circ \text{ and } MFP_\bullet \sqsupseteq MOP_\bullet \tag{2.16}$$

In case of a distributive framework

$$MFP_\circ = MOP_\circ \text{ and } MFP_\bullet = MOP_\bullet \tag{2.17}$$

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

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

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

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

That kind of nesting can be described well by context-free grammars. The set of finite paths can be seen as some *language*, finite sequences of letters (where the letters are the labels or nodes). Without procedures, the corresponding language is *regular*, with the control-flow graph being considered a "finite-state automaton").[6] Now, valid paths for

[6] That in language-theoretic terms, languages are typically sequences of letters attached to the edges (of a FSA) and not sequences of nodes is a not-so-relevant detail.

procedures no longer form a regular language. Instead, they can be captured a **context-free language**.

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

Context-free languages are typically given by context-free grammars, which is effectively what we do in a few slides. Actually before we cover valid paths, which is what we are interested in, we define as stepping stone something slighty simpler but closely related: **complete** paths.

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

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

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

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

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

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

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

MVP

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

⇒ capture the nesting structure

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

Complete paths

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

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

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

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

Table 2.5: Complete paths

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

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

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

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

Example 2.6.6 (Fibonacci). See Figure 2.5 for the control-flow graph of the Fibonacci program. The following shows the concrete grammar generating the complete paths for

that example.

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

□

Valid paths

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

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

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

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

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

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

MVP

We simply have to adapt the previous definition for MOP, to take into account not all by but only *valid* ones

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

- **MVP** solution:

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

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

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

2.6.5 Context-sensitive analysis

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

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

Indepedently from that, for interprocedural analysis, making a set of paths **first** and **then** building the meet or join over it is also a bad algorithmic design. The chaotic iteration or the work-list algo did not "follow" some paths, they were solving constraints, following strategies as they felt fit (or doing it randomly). What we therefore do, is extending that approach:

> extending the definition of the data-flow problem and thereby also extending or rather reusing the algorithmic approaches for that, like the worklist algorithm. This will be called **embellished** monotone framework, i.e. the old framework with additional information addted (to the lattice).

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

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

If we added information about the path which leads to a procedure call, one could completely disambiguate the call and return situation and, in this way, obtain the MVP solution as an embellished MF. We will take a look at that, though one would not technically need all the paths in the form we have defined it so far. It will be enough to just add

the sequence of all open calls, because we have to return to the last still open call. Older calls and internal edges don't need to be remembered. That will be called **call strings**. Still, that call string information, basically an abstract version of the current call stack, is **unbounded**; adding that yields the same precision as MVP, and that is undecidable, and we will see what to do with that.

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

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

Contexts

Let's introduce the notion of **contexts**. Actually, not concrete context is introduced for now, like defining how a context looks like. Equation (2.18) simply introduces a symbol we use to refer to contexts, whatever those are. Only later we will look at different reasonable choices for concrete contexts. For now, we assume we have some context δ and discuss how to extend a monotone framework to make use of that context, leading to the notion of embellished monotone framework.

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

⇒ instead of MVP: **"embellish"** MFP

$$\delta \in \Delta \tag{2.18}$$

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

⇒ "embellishment": adding **contexts**

Embellished monotone framework

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

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

Embellishment is notationally indicated by placing a hat on top. The following will proceed in two stages. The intra- and the inter-procedural part. The first part is, of course, simpler. One might ask, why we need to consider the first part at all? Well, we change the framework by embellishing it (indicated by the hatted syntax). That will involve a change

in the lattice and other concomitant changes. When thinking in terms of an implementation, the type of the analysis changes, like the type of the involved lattice, insofar that a new component is involved, the context. Consequently, also the intraprocedual part needs to be adapted, taking care of the contexts. As far as the intra-procedural part is concert, that is rather trivial: it basicall consists of "ignoring" the context: as long as one deals with data-flow *within* one function body, the *context* remains **the same**. Nonetheless, the additional context-component has to be mentioned as being unchanged when dealing for the embellished transfer functions and other parts of the monotone framework. But the key message is: the intra-procedural part is *"basically unchanged"* by the embellishement, as one would expect.

Notational remark

In the following we encounter silly problem. It's that we used the letters L and l for different interpretations. One is the lattice $\mathcal{L}$ with elements $l \in \mathcal{L}$. The other one is the set of labels **Lab** resp. $\mathbf{Lab}_*$, with typical elements l as well. Actually, in the book [10], there are two "different kinds" of l's, typeset with two slightly different fonts. For the slides, it's difficult to do the same, as slides are typeset with sans serif fonts, and the difference disappears. Anyway it may be confusing. What we do here instead: we still use l for the labels or nodes from **Lab** (as before) and write d, d', d_i, etc. for the elements or values of the lattice (d standing for "data").

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

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

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

$$\hat{\mathcal{L}} = \Delta \to \mathcal{L} \tag{2.19}$$

which contains functions taking a context from Δ to an element of the original, unembellished lattice $\mathcal{L}$. One can convince oneself that, if $\mathcal{L}$ is a lattice, then so is $\hat{\mathcal{L}}$. But what to think about it? The context reflects (an abstraction of) the situation when inside a procedure body. When not doing a call or a return, i.e., staying within one procedure

body, doing intra-procedural analysis, the context is unchanged. Equation (2.19) defines the embellished lattice $\hat{\mathcal{L}}$ by giving one lattice element $d \in \mathcal{L}$ per context $\delta \in \Delta$. This is a way of **lifting** the lattice to the new setting. Of course we need not just to lift the lattice from $\mathcal{L}$ to $\hat{\mathcal{L}}$, we need also to lift the intraprocedural transfer functions, which are functions from $\mathcal{L} \to \mathcal{L}$ to $\hat{\mathcal{L}} \to \hat{\mathcal{L}}$ (unembellished). If we have a transfer function f, then we write $\hat{f}$ for its embellished counterpart. We can see this this "hatting" operation, lifting functions f to $\hat{f}$ as a higher-order function as follows:

$$\hat{_} \;:\; (\mathcal{L} \to \mathcal{L}) \to (\hat{\mathcal{L}} \to \hat{\mathcal{L}}) \tag{2.20}$$

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

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

The lifting from the underlying transfer function f to its embellished counterpart $\hat{f}$ is shown in equation (2.24). The transfer function $\hat{f}_l$ (for some location l) is of type $\hat{\mathcal{L}} \to \hat{\mathcal{L}}$, i.e., a function of type

$$(\Delta \to \mathcal{L}) \to (\Delta \to \mathcal{L}) \tag{2.21}$$

So, it's a function that takes as argument a function of type $(\Delta \to \mathcal{L})$ and returns a function, namely one with Δ as argument and a return value from $\mathcal{L}$. That's defined in equation (2.24) below and shown on the slides: the "first" functional argument $\hat{d} : \hat{\mathcal{L}}$ is applied in the definition to the "second" argument δ and the results is fed into the unlifted transfer function f_l. Equation (2.24) may be written identically as follows, making use of λ-abstraction, and adding the types of the arguments to make it more explicit.

$$\hat{f}_l = \lambda\hat{d}{:}(\Delta \to \mathcal{L}).\ \lambda\delta{:}\Delta.\ f_l(\hat{d}(\delta)) \tag{2.22}$$

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

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

$$A \to B \to C \quad \text{and} \quad A \times B \to C \tag{2.23}$$

essentially the same. Of course the two types are not really identical. A function of the first type is invoked as $(f\ a)\ b$, a function of the second type as $f(a, b)$, the tuple (a, b) as an element of $A \times B$. The two types are equivalent in the following sense. It's easy to take a function and turn it into a function of the second type, and also in the other direction. Each function of the first type can be "transformed" into one of the second type, but doing the same, and also in the other direction. Not only "transformed" in that one can manually re-write each function of one type equivalently into one of the other form. That might be plausible. What is more: there exists two functions, higher-order of course, that realize those transformation. Technically, one says the two types are *isomorphic*, and the two transformations are called **currying** and **uncurrying**. A function of the functional type in equation (2.23) is said to be in *curried* form, the one operating on pairs is said to be *uncurried*. The terminology is in honor of the logician *Haskell Brooks Curry*, though perhaps historically more correct would be to call the concept schoenfinkelling (after Mosses Schoenfinkel), not currying, but perhaps that sounded too weird...

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

Intra-procedural: basically unchanged

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

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

– transfer functions: **pointwise**

$$\hat{f}_l(\hat{d})(\delta) = f_l(\hat{d}(\delta)) \tag{2.24}$$

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

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

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

The embellished lattice $\hat{\mathcal{L}}$ is defined by the function type $\Delta \to \mathcal{L}$, and we have defined the interprocedural transfer functions (lifting them). But let's also have look at what the *solution* of the data-flow problem now is, in the embellished setting. Equivalently, let's have a look at the data structure A (resp. $A_\bullet$ and $A_\circ$), mentioned in equation (2.25) and used in the worklist algo.

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

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

$$N \to \Delta \to \mathcal{L} \ . \tag{2.26}$$

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

$$(N \times \Delta) \to \mathcal{L} \ . \tag{2.27}$$

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

Remarks concerning how to implement that

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

A practical problem now might be: there can be many possible contexts from Δ, maybe even ininitely many. That poses a "challenge", independently of whether one bases the implementation on a curried or uncurried view (equation (2.26) vs. (2.27)).

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

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

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

Sign analysis

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

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

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

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

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

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

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

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

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

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

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

No matter how, unavoidable is that abstraction leads overapproximation and sometimes "losing the overview". Not always one may loose overview, for instance, if one has an

expression $x + y$ where it's known that x and y are positive, then it's clear that the result is positive: "plus plus plus gives plus", as everyone knows. If we do a minus on two positive numbers, however, we don't know if the result is positive, negative, or 0, it may be any of those.

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

There is an alternative: one could say, one maps every variable to a *set of values* from **Sign**. Both choices, $2^{\mathbf{Var}_* \to \mathbf{Sign}}$ and $\mathbf{Var}_* \to 2^{\mathbf{Sign}}$ form a lattice. That's good news, after all we know that data flow analyses basically *need* to be based on complete lattices... The two lattices are shown in equation (2.28) and (2.29).

$$\mathcal{L}_{sign} = 2^{\mathbf{Var}_* \to \mathbf{Sign}} \tag{2.28}$$

$$\mathcal{L}'_{sign} = \mathbf{Var}_* \to 2^{\mathbf{Sign}} \tag{2.29}$$

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

That's sound pretty much the "same" (and it is, as long as we have only one single variable). But let's look at a more general example, that should make the difference clear.

Example 2.6.7 (Different levels of abstraction). Assume two variables x and y and a situation in the lattice $\mathcal{L}_{\mathbf{Sign}}$

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

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

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

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

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

Definition 2.6.8 (Unembellished sign analysis)**.** Let's define as **lattice** of the sign analysis

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

Let's call elements of the lattice **abstract states** and write $\sigma^{sign} \in \mathcal{L}_{sign}$.

We assume as given the following semantic function on *expressions:* $[_]^{\mathcal{A}_{sign}}_{_} : \mathbf{AExp} \to (\mathbf{Var}_* \to \mathbf{Sign}) \to 2^{\mathbf{Sign}}$.

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

$$f_l^{sign}(Y) = \bigcup \{\phi_l^{sign}(\sigma^{sign}) \mid \sigma^{sign} \in Y\} \tag{2.30}$$

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

$$\phi_l^{sign}(\sigma^{sign}) = \{\sigma^{sign}[x \mapsto s] \mid s \in [\![a]\!]^{\mathcal{A}_{sign}}_{\sigma^{sign}}\} \tag{2.31}$$

The above definition of the **transfer function** proceeds, resp. is written in *3 steps*: At the core is the semantic function $[_]$. This function is for *expressions,* and is already non-deterministic. Equation (2.31) reflects the effect of an assignment for one abstract state and equation (2.30) is finally the transfer function. Note in passing, it's another example of lifting: the function ϕ, defined in equation (2.31) per "sign-state", is lifted to work on sets of state in the obvios manner (just on each individual). One also says, ϕ is lifted pointwise.

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

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

In the following, we will *embellish* the analysis. Since we are not yet in the inter-procedural part, the embellishment is not very interesting, just a "lifting" to the embellished setting. We know how that work already, abstractly (see equation (2.24)).

As mentioned before, the abstract lattice $\hat{\mathcal{L}}$ is of the form $\Delta \to \mathcal{L}$. Here the underlying lattice $\mathcal{L}$ is of a particular form, namely $2^{\mathbf{Var}_* \to \mathbf{Sign}}$. With this one, we, in a way, make use of uncurrying to represent $\Delta \to 2^{\mathbf{Var}_* \to \mathbf{Sign}}$ slightly diffently (but isomorphically). In equation (2.32), we use the symbol $\simeq$ of that. Since we used a slightly different representation, not a function from contexts to sets of sign-states, but sets of pairs containing contexts and set-states, the transfer function in equation (2.33) below is not in the form of a higher-function as done generally before, in equation (2.24).

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

Example 2.6.9 (Embellished sign-analysis: intra-procedural part).

$$\begin{aligned}\hat{\mathcal{L}}_{sign} &= \Delta \to \mathcal{L}_{sign} \\ &= \Delta \to 2^{\mathbf{Var}_* \to \mathbf{Sign}} \simeq 2^{\Delta \times (\mathbf{Var}_* \to \mathbf{Sign})}\end{aligned} \tag{2.32}$$

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

$$\hat{f}_l^{sign}(Z) = \bigcup \{\{\delta\} \times \phi_l^{sign}(\sigma^{sign}) \mid (\delta, \sigma^{sign}) \in Z\} \tag{2.33}$$

□

The transfer function is a special case of the intra-procedural embellished transfer function from equation (2.24) we have seen earlier, here exploiting uncurrying, i.e, without that "higher-order flavor" from before.

The unembellished one so far was a simple instance of the monotone framework. The transfer function just "joins" all possible outcomes, where it is assumed that we have as function that calculates the set of signs for expression. That was completely standard. Now, it does not get really more complex: equation (2.33) just does **nothing** with the contexts δ, since we are still **within** a single process. In the following we go to the inter-procedural fragment and there, things get more complex, since for dealing with calls and returns we have to *connect* the contexts of the caller and the callee. It's a bit like parameter passing.

Inter-procedural

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

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

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

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

$$A_\bullet(l_c) = \hat{f}^1{}_{l_c}(A_\circ(l_c)) \tag{2.34}$$

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

$$A_\bullet(l_r) = \hat{f}^2{}_{l_c,l_r}(A_\circ(l_c), A_\circ(l_r))) \tag{2.35}$$

Figure 2.6 illustrates the situation for procedure calls.

Note: *we* write $\hat{d}$ for elements of the embellished lattice (not $\hat{l}$ as in the pic). Note the unfortunate notational collision in the picture: $\hat{l}$: element of embellished lattice (abstract value), l_c etc: nodes/labels in the control flow graph.

One way to understand the 2 arguments for the return is that often one wants to **match** the return with the call (via the context).

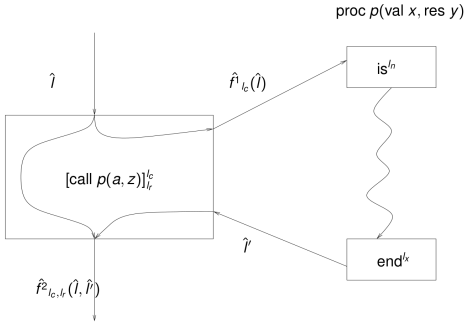
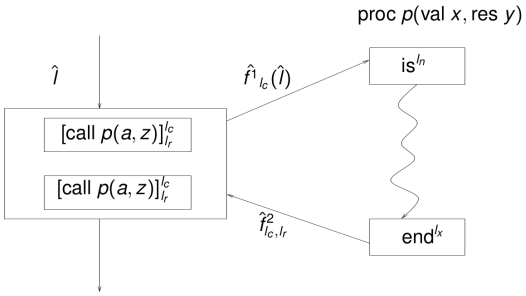
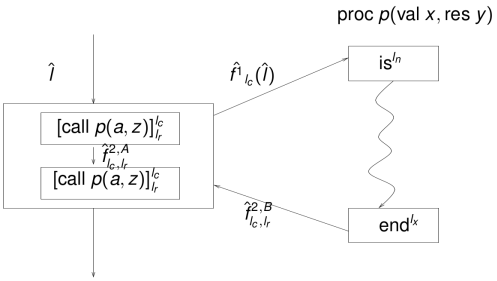


Figure 2.6: Transfer function for procedure calls

The following Figure 2.7 shows to two different **simplifications** for f^2. The one from Figure 2.7a simply **ignores** the call-contest, the one from Figure 2.7b **merges** the contexts.



(a) Ignoring the call-context



(b) Merging the call-context

Figure 2.7: Two simplifications

That can be captured by the following formulation of the transfer function

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

Context sensitivity

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

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

In the following:

2 *specializations*:

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

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

Call strings

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

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

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

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

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

Example 2.6.10 (Call strings for Fibonacci). Cf. the CFG for the fibonacci example from Figure 2.5. For that example, the following lists a few **call strings:**

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

□

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

Transfer functions for call strings

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

Transfer functions for call strings as contexts

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

$$\begin{aligned}(\hat{f}^1_{l_c}\ \hat{d})\ [\delta; l_c] &= f^1_{l_c}(\hat{d}(\delta)) \\ (\hat{f}^1_{l_c}\ \hat{d})\ _ &= \bot\end{aligned} \tag{2.36}$$

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

$$\begin{aligned}(\hat{f}^2_{l_c,l_r}\ (\hat{d}, \hat{d}'))\ \delta &= f^2_{l_c,l_r}(\hat{d}(\delta), \hat{d}'([\delta; l_c])) \\ (\hat{f}^2_{l_c,l_r}\ (\hat{d}, \hat{d}'))\ _ &= \bot\end{aligned} \tag{2.37}$$

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

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

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

Let's first have a look at how it's done for calls in equation (2.36). The "lifting" of f_{l_c} is done analogously the same higher-order way as we have seen in the intraprocedural case (in equation (2.24). But the higher-order application of the underlying function is *not quite identical*. In equation (2.24), the context δ does not play a role at all: $(\hat{f}_l\ \hat{d})\ \delta = f_l\ (\hat{d}\ \delta)$. Now, for equation (2.36), the call-string context δ changes, that's the key thing. The change of δ from the caller to the extended one $[\delta; l_c]$ effectively lifts $f^1_{l_c}$ to the embellished setting connecting analysis-results *only* for matching callers and callees and far as the call-string is concerned. That transfers the data flow information according to the underlying transfer function f_{l_c} from the caller to the callee, and the callee is analyzed with that information and in a context appropriately extended l_c. Remembering that caller-information distinguishes the different call-site instances and allows also allows also to connect the callee back to the site where it had been called in equation (2.37).

In the definitions, _ stands for a "wild-card" argument, like all cases not coverd by the first line. So the two lines of the definitions in (2.36) and in (2.37) are to be read in a

first-match manner (as in programming languages).

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

The same technique is use also for lifing the transfer function $f^2_{l,c}$ (again assumed given) for returns and matching the corresponding activations of callee and caller in equation (2.37).

Note the coupling.

In (2.36): we want to define the embellished transfer function $\hat{f}^1_{l_c} : \hat{\mathcal{L}} \to \hat{\mathcal{L}}$. Basically, it takes the unhatted function (assumed given) and applies it to $\hat{d}([\delta; l_c])$. Remember: $\hat{d}$ is a member of the embellished lattice $\Delta \to \mathcal{L}$. In other words: $\hat{d}(\delta)$ calculates the data d in the original lattice using the calls string *without* the call. So the equation is to be interpreted as taking $\hat{d}$ as fixed, and then recur over the list of labels for a function of type $\Delta \to \mathcal{L}$. For the empty list, the function is undefined.

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

The definition of the flow functions should be compared to the intra-procedural one, i.e., the pointwise lifting in equation (2.24) on page 125. In the point-wise definition, the δ is *not touched,* but just feeded into the hatted $\hat{f}$.

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

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

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

Interesting is equation (2.37). There, the match of the return with the corresponding call is done.

The case for calls is the simpler. We define the transfer function, and it's a "very" higher-order definition. Being a transfer function, the first argument is $\hat{l}$, an element from the embellished lattice, i.e., it's a function. which gives an element of L depending on the context argument. Now, the task is to encode the context-sensitivity: we need to define the flow exactly from a specific point (one can see it as the *pair* of location and the context, here the call string). So it "sets" the post-value for the $[\delta; l_c]$, but not for "others", therefore the else-part sets it to bottom. How is it defined? It needs to find the value of the lattice L in the shorter context ($\hat{l}(\delta)$ and apply tje given "underlying" function $f^1_{l_c}$ to it.

In the following, we apply the definition in the specific setting of the sign-analysis. The definition of how to cover calls and returns with the corresponding transfer function where "involved", due to the higher-order formulation. In the specific setting of the sign-analysis, we make use of *uncurrying* the representation (as we did before), i.e., we operate on sets of pairs rather than on applying functions (remember equation (2.32)).

Sign analysis (continued)

- so far: "unconcrete", i.e.,
- given some underlying analysis: how to make it context-sensitive
- call-strings as context
- now: apply to some simple case: signs
- remember: $\hat{\mathcal{L}} \simeq 2^{\Delta \times (\mathbf{Var}_* \to \mathbf{Sign})}$ (see Eq. (2.32))
- before: standard embellished $\hat{f}_l^{\mathbf{Sign}}$ (with the help of $\phi_l^{\mathbf{Sign}}$)
- now: *inter-procedural*

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

Example 2.6.11 (Sign analysis: **underlying** transfer functions for calls and returns). For the transfer function other than the ones dealing with calls and returns, see example 2.6.8. We are still in the inembellished setting. For the definition of the transfer function for calls and returns, we are giving the definition of only the auxiliary functions ϕ. Cf. equation (2.31).

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

$$\begin{array}{rcl} \phi_{l_c}^{sign1}(\sigma^{sign}) & = & \{\sigma^{sign}[x \mapsto s][y \mapsto s'] \mid s \in [a]_{\sigma^{sign}}^{\mathcal{A}_{sign}},\ s' \in \{-, 0, +\}\} \\ \phi_{l_c,l_r}^{sign2}(\sigma_1^{sign}, \sigma_2^{sign}) & = & \{\sigma_2^{sign}[x, y, z \mapsto \sigma_1^{sign}(x), \sigma_1^{sign}(y), \sigma_2^{sign}(y)]\} \end{array}$$

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

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

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

Example 2.6.12 (Sign analysis: embellished f's for call and returns with call-strings). The two embellished transfer functions for calls and returns for the sign-analysis and with call-string constext are given as follows:

$$\hat{f}^{sign1}_{l_c}(Z) = \bigcup\{\{\delta'\} \times \phi^{sign1}_{l_c}(\sigma^{sign}) \mid (\delta, \sigma^{sign}) \in Z, \delta' = [\delta; l_c]\} \tag{2.38}$$

$$\hat{f}^{sign2}_{l_c,l_r}(Z, Z') = \bigcup\{\{\delta\} \times \phi^{sign2}_{l_c,l_r}(\sigma^{sign}_1, \sigma^{sign}_2) \mid \begin{array}{l}(\delta, \sigma^{sign}_1) \in Z \quad \} \\ (\delta', \sigma^{sign}_2) \in Z' \\ \delta' = [\delta; l_c]\end{array} \tag{2.39}$$

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

calls: abstract parameter-passing + glueing calls-returns Here we see the "uncurrying" we discussed before in action, exploiting the special form of the lattice for the sign analysis (equation (2.32)). Well, actually the situation is not so special. Many analyses work with sets of this or that, i.e., with subset-lattices. So in very many cases, one can switch to an uncurried representation., like here. That allows a formulatiion *pairing* contexts δ with the rest of the information, as opposed to feeding the context as argument in some higher-order function, as in the general formulation.

The lifting of the underlying $\phi^{sign1}_{l_c}$ in the case of calls in equation (2.38) extends δ from the caller to $\delta' = [\delta; l_c]$ in the callee. The same connecting of callee and caller is done for the returns in equation (2.39)

Call strings of bounded length

Back from the specific sign analysis to the general setting of the embellished monotone framework, in particular, the embellishedment with call strings.

> What we have achieved with the call-strings is: turning the MVP approach as a instance of a monotone framework by making the lattice more complex and lifting the underlying transfer functions to the more complex setting ("embellished" as it's called here).

The embellished lattice, however, has become quite more complex. In many settings, the underlying lattice $\mathcal{L}$ is finite (like in the 4 classical examples and also in the case of the sign analysis). The embellished lattice $\hat{\mathcal{L}}$ with call-strings, however, is definitely not finite. That means there's not guarantee for *stabilization* for the worklist algo or the chaotic iteration. That perhaps should not come as surprise. It has been mentioned that **MVP is undecdable.** In general, at least, there may be particular programs where it terminates, of course. Still, since adding call strings is a way of encoding MVP, that cannot make a undecidable problem decidable, so there cannot be a guarantee of stabilization.

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

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

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

$\Rightarrow$ restrict the length

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

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

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

Another form of contexts: assumption sets

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

But one can use other information besides the return address, one can use the *data value* with which a function is called to at least narrow down where the function was called from. There's no garantee that it locates the call site instance precisely, as the return address, but still in can disambiguate the situation to some extent, thereby adding precision to some extent. In general, it won't be possible use actual, concrete data values for that. Like in our languages, integers. The data domains for arguments are often infinite, and anyway, the underlying analysis concretely not even track value. Thus, one will generally use *abstractions* as context, not concrete values. And actually, many analyses not even work with abstractions of data. For illustration: our toy language uses integers as data

values. However, none of the 4 classical analyses actually worked with abstractions in $\mathbb{Z}$! Only the sign-analysis later did that, abstracting (or partitioning) the infinite set $\mathbb{Z}$ into three classes, the positive, the negative, and the set containing 0.

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

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

$$\mathcal{L} = 2^D \tag{2.40}$$

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

Technically, we had defined (concrete) states as an element from $\mathbf{Var}_* \to \mathbb{Z}$. Though the formulation here is *not* meant as applying only to setting where done does assumptions on particular this definition on states. The idea can be used for lattices of the form as specified in equation (2.40). Which is basically for all analyses we have seen.

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

Conceptually, one can use the idea broadly. Technically the slides show how to lift the transfer function and connect calls and returns making use of a auxiliary function ϕ. We have seen an example of that in the sign-analysis. To define the transfer functions with the help auxiliary functions ϕ was done previously in the sign-analysis (and only there). The analysis made use of "abstract states" $\sigma^{sign} : \mathbf{AState} = \mathbf{Var}_* \to \mathbf{Sign}$. The auxliary function ϕ had been of type $\mathbf{AState} \to 2^{\mathbf{AState}}$, it captured the general case where the (abstract) state is changed. That happens for the sign analysis for assignments and for parameter passing. Working on abstract sign-information, the "result" of the auxiliary function is, in the general case, a *set* of abstract states, i.e., an element of $2^{\mathbf{AState}}$. That captures the "non-determinism" in the effect of the abstraction, in particular for the assignment. Remember in particular equation (**??**), which made that clear for the sign-analysis. This setting, where transfer functions of type $\mathcal{L} \to \mathcal{L}$, i.e., $2^D \to 2^D$ are defined making use of an auxilary function of type $D \to 2^D$ is characteristic for situations, when D

is seen as (abstract) state, and the sign-analysis is an example for that. That means, the other 4 analyses, where the lattices is perhaps not naturally called "state" and where the transfer functions are not defined via some auxiliary construction ϕ, will not be exactly formulated the way we do in the following. That does not imply, one cannot introduce context that contain information "data" information, like having $\Delta = \mathcal{L}$. One can do that, though the embellished transfer functions cannot be written exactly as in the formulas that follow. But one can of course just use the general curried, higher-order formulation from equation (2.22), that works for *all* kinds of contexts. Anyway, here it's in a setting that resembles more the sign-analysis, with D corresponding to the type of abstract states $\mathbf{AState} = \mathbf{Var}_* \to \mathbf{Sign}$.

Assumption sets

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

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

- dependency on "data" only $\Rightarrow$

$$\Delta = 2^D$$

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

Transfer functions

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

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

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

and the corresponding embellished transfer function is given in equation (2.41). It's analogous to the definition we did for the sign analysis and with call strings in equation (2.38).

As for the definition from equation (2.41): the type of the transfer function (for calls) is

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

Elements of the lattice are thus sets of pairs, the parts of each tuple consists of the current data or state d in the second position and a set of data or states δ in the first position, representing the *contexts*. As mentioned, the general picture here is: when calling a

function, use the current data d to identify the place of the call, where the identification is not 100% precise, so it's about narrowing down possible places where the procedure is called.

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

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

The transfer function for return from equation (2.42) works correspondingly. Connecting the callee back to the caller by requiring the connection between the shape of the caller context δ and the callee context δ'.

- calls

$$\hat{f}^1_{l_c}(Z) = \bigcup\{\{\delta'\} \times \phi^1_{l_c}(d) \mid \begin{array}{l}(\delta, d) \in Z \wedge \\ \delta' = \{d'' \mid (\delta, d'') \in Z\}\end{array}\} \tag{2.41}$$

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

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

$$\hat{f}^2_{l_c,l_r}(Z, Z') = \bigcup\{\{\delta\} \times \phi^2_{l_c,l_r}(d, d') \mid \begin{array}{l}(\delta, d) \in Z \wedge \\ (\delta', d') \in Z' \wedge \\ \delta' = \{d'' \mid (\delta, d'') \in Z\}\end{array}\} \tag{2.42}$$

Small assumption sets

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

As explained, the previous approach can be seen as taking "historical" context-information with a bound $k = 1$ into account. One can simplify that further, storing no such information, i.e., $k = 0$. That's not a context-insensitive approach, as it had been the case with call-strings: there $k = 0$ meant: no information is used whatsoever. Here, we use "data" (or state) information from D, and we simply use that as callee context, but *not* also the context of the caller. I.e., we used the data (here d) itself to distinguish different call instance and connect back the analysis when returning, but *ignore* the context of the

data. Basically, we identify all call activations with the same "arguments" from all possible callers, though of course the arguments are actual, concrete data, but the information used in the flow analysis. We ignore how, however, the context of the caller(s), as was done with the previous approach. That approach here known as *small assumption sets* (where as the previous one or similar are consequently *large* assumption set approaches).

- throw away even more information.

$$\Delta = D$$

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

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

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

Flow-(in-)sensitivity

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

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

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

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

Technically, flow insensitivity does not mean, all "orderings" in the program can be ignores. Technically, it means that the order imposed by sequential composition ("semicolon") is irrelevant, see equation (2.43). But there may be other, less obvious "ordings" in a program. For instance, one may be interested in a so-called *call-graph*: which procedure calls which one, i.e., which proceure may call which one, for instance in the while-language with procedure (in higher-order languages, things get much more involved; we have stumbled upon that already in the introduction). When doing such a call graph, one is interested in that, for instance f calls g, in that sense that the callee g can start executing only *after* that f has started (unless g may call f as well, in which case one might not now). At any rate, the call graph can contain some form of ordering but others not. for instance, if f calls uniformely g_1 an afterwards g_2, then the ordering between g_1 "before" g_2 is not part of the call-graph. The corresponding information is flow-insensitive, in the sense defined. Any it's also not ridiculously trivial, one has to build up some graph, but it's still fairly simple, at least for a while-language with procedures of the form we have seen. Also the analysis IAV later (implicitly) performs such a call-graph analysis.

- "execution order" influences result of the analysis:

$$S_1; S_2 \quad \text{vs.} \quad S_2; S_1 \tag{2.43}$$

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

Set of assigned variables

- for procedure p: determine

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

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

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

$$\mathsf{IAV}(p) = (\mathsf{AV}(S) \setminus \{x\}) \cup \bigcup \{\mathsf{IAV}(p') \mid p' \in CP(S)\} \tag{2.44}$$

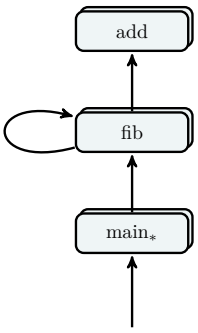
where $\texttt{proc}\, p(\texttt{val}\, x, \texttt{res}\, y)\texttt{is}^{l_n} S \texttt{end}^{l_x} \in D_*$

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

Example

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

Example



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

$\Rightarrow$ smallest solution

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

2.7 Static single assignment

2.7.1 Introductory remarks

This section is not covered by the book Nielson et al. [10]. It's added 2020 as it's an interesting and important angle on data flow analysis. [10] covers so-called **definition-use** or **use-definition** chains (aka du- and ud-chains), which are left out instead this time. Only so much for here: du- and du-chains are basically a slight variation or specialization of *reaching definitions*. Reaching definitions, as we know, calculates point by which definitions a point is reached and that's done for all points. Now, du-chains focus on uses, i.e., given a definition of a variable x at some node, one is interested all points reched by it where it that assignment or definition is used. Points reached by the definition of x but where x is not used are of no interest. So indeed, it's just reaching definition, filtering out at the end non-interesting information, thus connecting a definition of a variable with places where it's used. That's a "forward" version of that information, called def-use or du. There is also a "backward" version, connecting a point of use with all definitions that can reach the use. That's known as use-def (or ud). So basically, the concepts of def-use and use-def should be familiar (therefore we left out more formal details, and SSA can be seen as a generalization, in turn.

More than one ever wants to know can about SSA can be found in [8]. Early citations are Rosen et al. [12] Alpern et al. [2], and Cytron et al. [6]. So also historically, it's connected to value numbering.

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

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

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

Of course one encounters situations like the following

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

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

Now, with this regime, one can say, functional and declarative languages are **single assignment**. Same actually for logical formalisms and the use of variables in math (they are declarative, not imperative). Though one normally does not functional language as being "single-assigment", as one feels there's not assignment statement at all, just `let` or similar definitorial mechanisms.

One way of characterizing purely functional or declarative languages as enjoying what's called **referential transparency**. It refers to an in generally welcome property of expressions (in this case variables). It means that a value of an expression (or here a variable) is *independent of where the expression appears* (inside its scope).

The statement should perhaps be qualified in that the value of a variable is always the same (unless there is *no value*, i.e., the variable is undefined). If we ignore the "being undefined" as special status, then a referentially transparent value means indeed the value of the variable is "always the same" (and that means the value is **immutable** and the variable is **single assignment**, and there might be other characterizations as well).

If one deals with referentially transparent variables, many things become easier, the concept of variable becomes more *logical*. Variables in imperative programming languages have the flavor of being names to memory addresses, whereas variables in mathematical text books, equations, or in logical formulas do not give that feeling. The reason being that expressions and variables are intuitvely understood as being referentially transparent (not that any math text would ever point that out, it simply does not cross anybody's mind what a variable has to do with a mutable memory cell ...) Also in the lecture, as said, we carefully try to separate assigmnents $x := e$ from equations $x = e$, the latter being understood as referentially transparent (or declarative ...). The opposite of referentially transparent is also known as referentially opaque.

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

what it means to be "equal", the standard mathematical, logical (referentially transparent, declarative ...) meaning of equality:

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

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

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

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

2.7.2 A trivial starting point: straight-line code (SLC)

The core idea for using SSA is actually pretty simple, the following shows an example. If one intends that each "variable" is used only once, one simply has to use enough variables. If the program to start with violates the SSA assumption, like assigning to a variable more than once, one simple "**renames**" the variable, using a new variable each time an assignment targets the variable.

In this section we are using not the while language from earlier, but some intermediate language formate. It could be called **three address intermediate code**, because the basic data manipulation operation ("assignment") works with three "addresses": two source variables and the target variable. Of course, it's not 100% strict, also something like `x := y` is in three-address format, even if only two variables are involved.

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

A format like 3AC is some plausible form of so-called **intermediate code**. So it's no longer user-level abstract syntax, but closer to a typical machine code level (which may be in

3A-format or 2A-format).[7] Turning assignments of the general form of the while language to 3AC is not hard, and the translation resembles what needs to be done for SSA. Simply introducing new variables when one the 3AC cannot handle a compound expression. The additional variables are also called *tempory variables* or just **temporaries**. We illustrte the idea in some later slide.

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

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

Let's look at the code from Listing 2.3. It's some 3AIC not in SSA format. The mentioned "numbering" trick to turn it to SSA should be obvious. See Listing 2.4. The subscripts here is just a way of systematically make clear, that b_1, b_2 etc. all have they origin in the variable b. This convention is helpful for reading such pieces of code; how it is implemented is a different issue, but using natural numbers for different versions of a variable is certainly an option.

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

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

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

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

Listing 2.4: 3AIC in SSA

[7] There is also something called 1A-format, but that's not used for hardware, only for virtual machines and interpreters, for instance for byte-code (though historically there had been attempts in HW).

The transformation for straight-line code is *straightforward*. It's so simple that one could ask oneself: what's the big deal? Why is SSA said to be such an important format, if the whole thing seem pretty trivial? Well, as we mentioned a few times: for static analysis in general, straight-line code is not too complex. Actually, analysing straight-line code allows basically exact analysis, without approximation. There is no question whether a variable may or may not be used on the future: it *is* used in the future or else *it is not* (liveness or dead-ness of a variable is clear). Same for reaching definition etc. Trying to *exacly* exactly analyse straigh-line code and making some optimization and optimal compilation based on that is sometimes called *super-optimization*. Optimization is often understood as doing it better than without optimization, so omptimization is nothing else than an imporovement. But super-optimization is meant as getting an absolutely best result (given some critera). That word is of course, doubly-stupid. Optimization, as said, in compilers and static analysis does not mean getting something optimal (that being undecidable in general). To call the situation for SLC, where it may be obtainable "super-optimal", i.e., better than optimal, is pretty silly...

3AC and temporaries

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

As said, there's a difference between single-assignment and static single-assignment, but for SLC, there's no diffence as any line of code in stretch straight-line is executed exactly one time.[8]

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

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

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

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

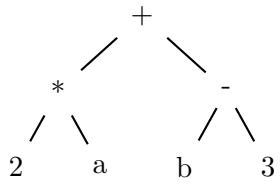


Figure 2.8: `2*a+(b-3)`

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

Listing 2.5: Three-address code with temporaries

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

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

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

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

If we were sticking to straight-line code, without conditional jumps, conditionals, loops etc., everything would be rather simplistic and straightforward, as illustrated in the code of Listing 2.5. The problem starts with conditionals (and loops). That will be treated next, and leads to the introduction of so-called Φ-functions.

2.7.3 Join points and phony functions

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

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

Assume the following pseudo-code

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

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

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

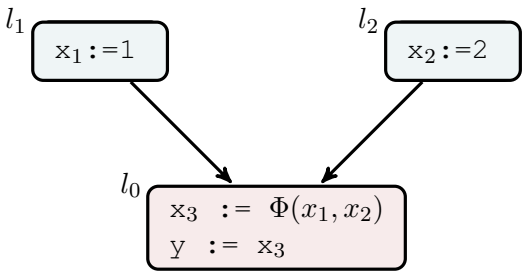


Figure 2.9: Join point and phony function

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

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

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

SSA in a nutshell

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

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

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

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

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

Maximal SSA recipe:

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

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

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

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

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

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

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

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

dominance

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

The situation is illustrated in Figure 2.10

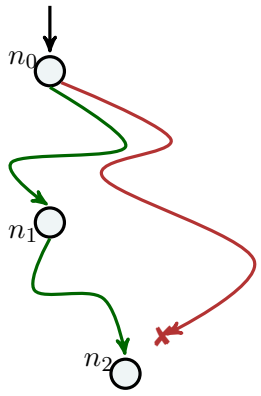


Figure 2.10: Illustration of dominance-based placement for Φ

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

Definition 2.7.1. Node n_1 **dominates** n_2, if all paths from entry n_0 to n_2 must pass through n_1. n_1 **stricly** dominats n_2 if n_1 dominats n_2 and $n_1 \neq n_2$. We write $dom(n)$ for the set of **dominators** of n, the set of nodes that dominate n.

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

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

We the placement algo does is to not cover all join nodes in the danger zone by a Φ-function, but only the *first* ones, when entering the danger zone. I.e., when one starts at at a given node, the node dominates itself (though not strictly), following the edges, the subsequent node may still be dominated, and the the next one still, so all those nodes are still in the "safe zone". But following the edges further there may come a point when one hits a non-dominated node, i.e., one enters the danger zone. And if there is not loop-back, one will stay in the danger size. If a node is undominated, also its successor is (unless there's a loop). In that picture, it suffices to place a Φ-assignment **at that very first node when entering the danger zone**. If that's covered by a phony function, subsequent nodes are as well (again loops need special consideration, as the concept of "subsequent nodes" is not really appropriate in a loop ...).

Anyway, the first nodes in a danger zone (i.e., undominated by a given node), that's known as **dominance frontier** of the node under consideration.[9]

Definition 2.7.2 (Dominance frontier)**.** The *dominance frontier* $df(n)$ of a node is the collection of m s.t.:

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

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

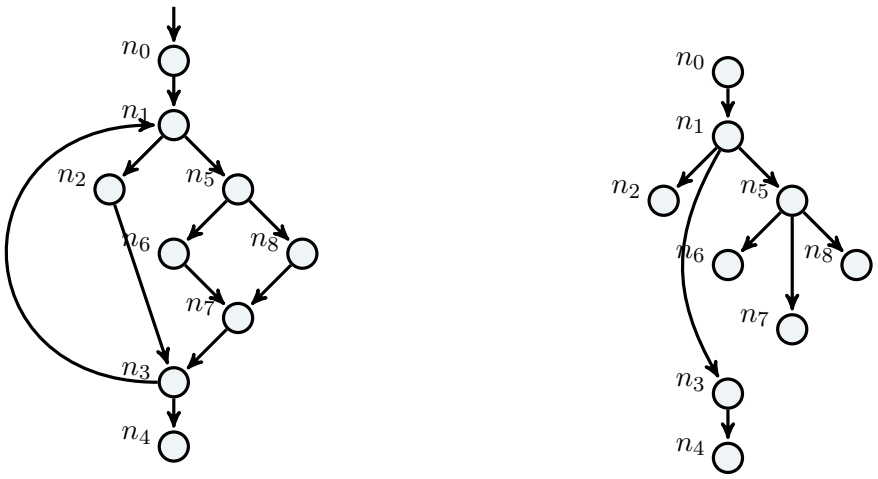
In connection with dominance, the concept of *dominator trees* is also useful. A node n may have many strict dominators (or none), they are written as $dom(n)$. If we focus on the (strict) dominator closest to n (if it exists), then that one is unique. Only the entry node of the graph has no immediate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

The following shows in pseudo-code an algorithm to calculate the dominance frontier.

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

for all nodes  n
```

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



(a) Some sample (control-flow) graph

(b) Dominator tree

Figure 2.11: Graph and dominator tree

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

Listing 2.6: Dominance frontier

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

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

The algo is simple enough, and needs both the original graph as well as the immediate dominator tree. First one takes all predecessors in the CFG of the chosen node (and there has to be more than one, as we concentrate on join nodes). From each of those predecessors, we walk up the dominator tree. That walk is "deterministic", as we use the dominator *tree* upwards. Note that we are not only following the tree upwards, it's also

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

Example 2.7.3 (dominance frontier for the previous graph).

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

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

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

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

  for each operation x ← y ⊕ z
     rewrite y with subscript top(stack[y])
     rewrite z with subscript top(stack[z])
     rewrite x as newname(x)
  for each successor of b in the CFG
```

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

Listing 2.7: Rename

Bibliography

[1] Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2007). *Compilers: Principles, Techniques and Tools*. Pearson,Addison-Wesley, second edition.

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

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

[4] 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*.

[5] Cooper, K. D. and Torczon, L. (2004). *Engineering a Compiler*. Elsevier.

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

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

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

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

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

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

[12] Rosen, B. K., Wegman, M. N., and Zadeck, F. K. (1988). Global value numbers and redundant computations. In [11], pages 12–27.

Index

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

abstract cache, 38
abstract data type, 14
abstract environment, 38
abstract syntax, 12
access link, 109
activation record, 108
algorithms, 54
analysis
    context-sensitive, 120
    control-flow, 34
    embellished, 120
ascending chain condition, 93
assumption set, 137
attribute, 43
attribute grammar, 3, 43
available expression, 62
available expressions, 62
axiom, 45

backward analysis, 67

call string, 121
call tracking analysis, 53
call-by-result, 101
calling conventions, 108
chaotic iteration, 55, 95
Church-style typing, 53
code generation, 149
common subexpression elimination, 61
common subsexpresion elimination., 61
complete lattice, 80, 93
complete path, 117
configurations, 103
constraint solving, 54
constraint system, 54
context, 120
context-sensitive, 43
context-sensitive analysis, 120
control flow
    reverse, 60
control flow graph, 17, 58
control-flow, 34
control-flow analysis, 34
control-flow graph, 47
correctness
    partial, 43
CSE, 61
Curry-style typing, 52
currying, 123

data flow analysis, 58
data-flow analysis, 10
derivation rule, 45
derivation sequence, 78
desugaring, 12
dynamic link, 109
dynamic memory, 99

effect, 42
effect system, 46
embellished analysis, 120
evaluation, 42

$\hat{f}$ (embellished f), 123
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
hoisting, 68

IF, 103
input variable, 22
intraprocedural analysis, 58
isolated entry, 59

judgment, 42

kill and generate, 62

label consistency, 60

labelling, 38
lattice, 23, 80
    complete, 32, 93
lazy evaluation, 106
location, 104

monad, 42
monitoring, 4
monotone framework, 58, 91
MVP, 117

optimization problem, 83
order relation, 80

parameter passing, 101
partial correctness, 43, 44
partial order, 80
path, 115
    complete, 117
    valid, 116
path insensitivity, 127
path sensitive analysis, 112
polymorphism, 41
pre-fixpoint, 34
program transformation, 68
property space, 93

reaching definitions, 20, 65
referential opaqueness, 145
referential transparency, 144, 145
reverse control flow, 60
Rice's theorem, 7
run-time environment, 99
run-time verification, 4

semantic analysis, 5
sign analysis, 126
SOS, 75
SSA, 58
stabilization, 93
stack frame, 108
static link, 109
static memory, 99
structural operational semantics, 75
subsumption, 47
symbol table, 43
syntactic sugar, 12
syntax-directed, 45

temporaries, 147
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

uncurrying, 123
under-approximation, 62
unique entry, 59

valid path, 116
value, 42
very busy, 68

while-language, 12
worklist, 95
worklist algorithn, 95